Laboratory #1: Floating Point Multiplication

Submission Date: February 14th 2025

CEG3156 - Computer Systems Design

Winter 2025

School of Electrical Engineering and Computer Science at the
University of Ottawa

Course Coordinator: Rami Abielmona, PhD, P. Eng
Teaching Assistant: Pavly Saleh

Nida Taj 300239050
August Zhang 300310509

Group 6

Theoretical Part

Introduction

The objective of this lab is to be able to design, implement and test both a floating point adder and floating point multiplier unit using VHDL. This lab is important because floating point arithmetic is a critical concept in computing. It allows for the representation of a wide variety of numbers, ranging from very small to very large with a great deal of precision. This lab furthers the understanding of this concept, through the successful implementation of both floating point arithmetic units.

Discussion of problem

The problem addressed in this lab regards the ability to successfully implement a floating point adder and multiplier unit. Both units will take two inputs, A and B, each consisting of a single sign bit, an 8-bit mantissa, and a 7-bit exponent, that will be represented in excess 63 format. The two inputs will then be added/multiplied together, depending on the unit being used, and output the final sign bit, 8-bit mantissa, and 7-bit exponent in excess 63 format. The design for both components will be done using the ASM methodology. The proposed solution can be seen in figure 1.1 below:

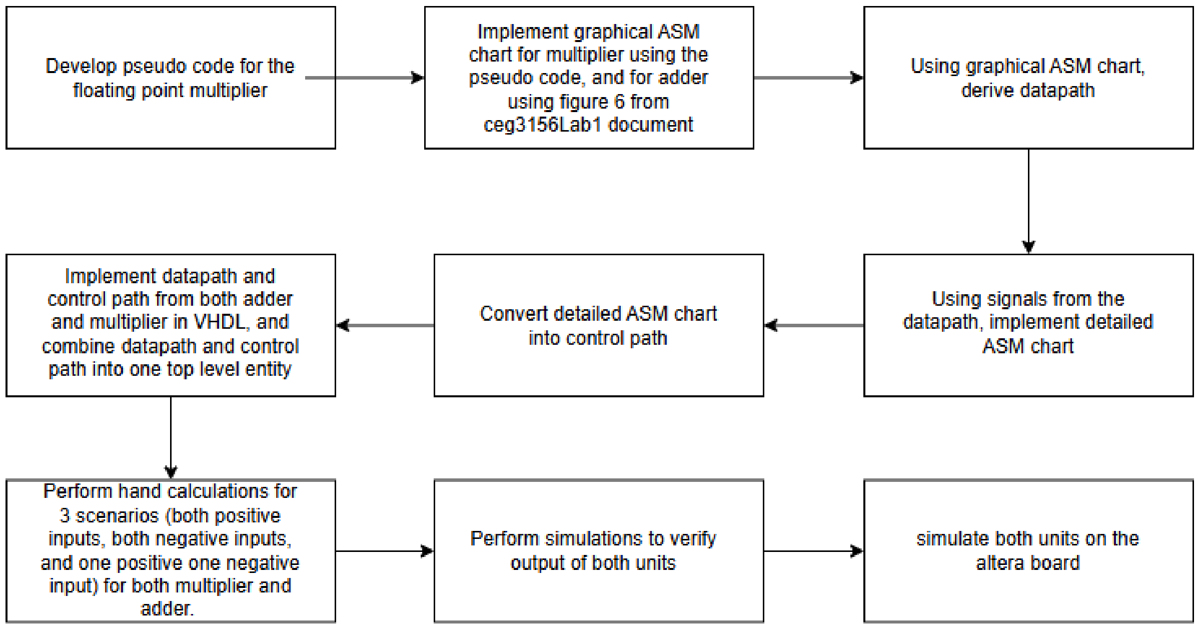

Develop pseudo code for the floating point multiplier
Implement graphical ASM chart for multiplier using the pseudo code, and for adder using figure 6 from ceg3156Lab1 document
Using graphical ASM chart, derive datapath
Using signals from the datapath, implement detailed ASM chart
Convert detailed ASM chart into control path
Implement datapath and control path from both adder and multiplier in VHDL, and combine datapath and control path into one top level entity
Perform hand calculations for 3 scenarios (both positive inputs, both negative inputs, and one positive one negative input) for both multiplier and adder.
Perform simulations to verify output of both units
simulate both units on the altera board

Figure 1.1 - Flowchart representation of proposed solution for both floating point arithmetic units

Discussion of algorithmic solution

The ASM methodology was employed for both the adder and the multiplier units. A graphical chart, datapath,detailed ASM chart, and control path were all created over the course of this lab, and are presented below:

<u>Multiplier</u>

```
$RM_A \leftarrow 1.M_A$ , $RM_B \leftarrow M_B$
$RS_A \leftarrow S_A$ , $RS_B \leftarrow S_B$ , $RS_m \leftarrow 0$
$RE_A \leftarrow E_A$ , $RE_B \leftarrow E_B$          # bias
$RE_M \leftarrow RE_A + RE_B$
$RE_M \leftarrow RE_m - 63$   # this will be exponent for the output.
$Flag \leftarrow 0$   # use for overflow detection
$RM_m \leftarrow RM_A \times RM_B$   # multiplication result

normalization if necessary
{ if ($RM_m \geq 2$)
      then begin
          $RE_m \leftarrow RE_m + 1$      # increment exponent
          $RM_m \leftarrow RM_m >> 1$     # Shift product right
      end;

overflow detection
{ if ($-63 \geq RE_m$ OR $RE_m \geq 64$)
      then begin
          $Flag \leftarrow 1$     # Overflow detected
          $RM_m \leftarrow \infty$
      end;

Sign
{ if ($S_A \oplus S_B == 1$)
      $RS_m \leftarrow 1$
  else
      $RS_m \leftarrow 0$
```

Figure 1.2 - Floating point multiplier pseudo code

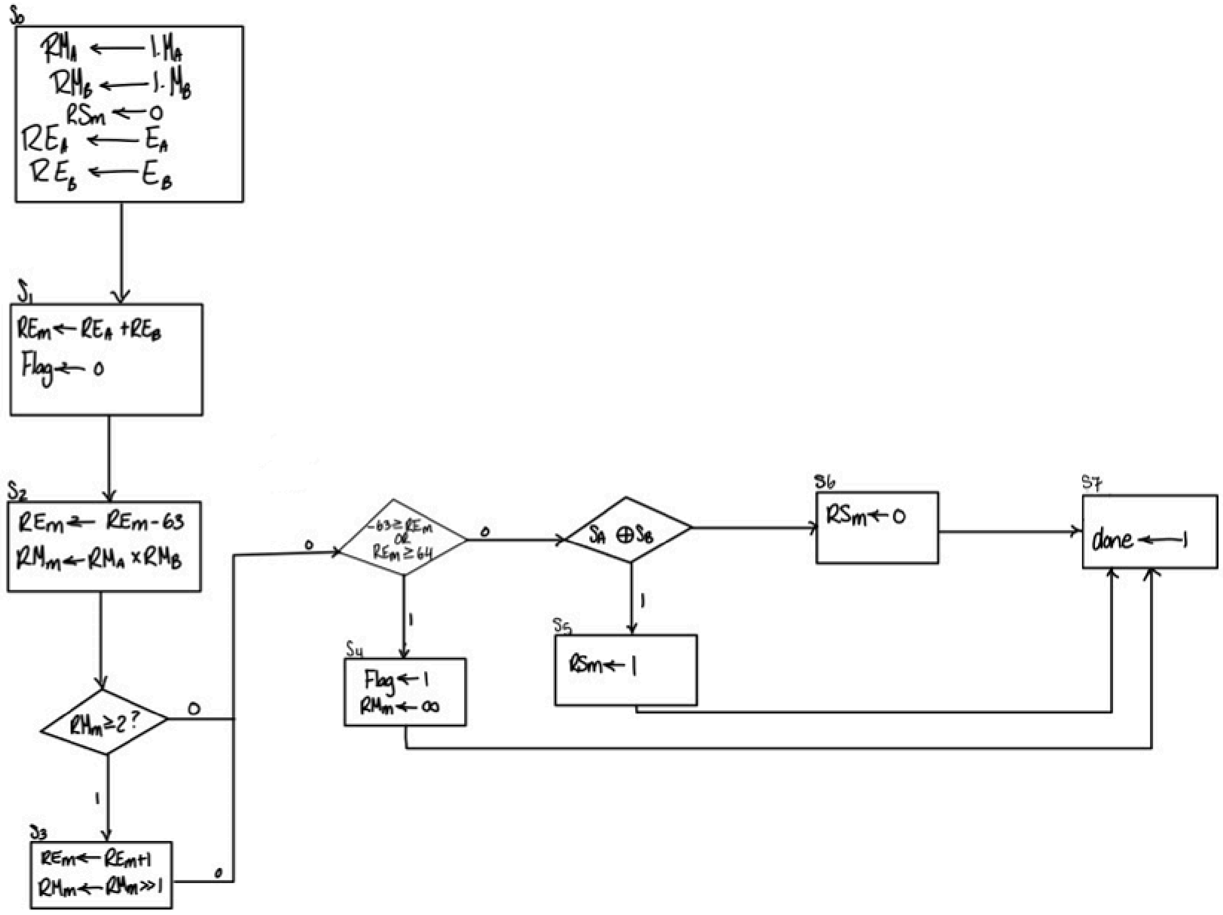


Figure 1.3 - Floating point multiplier graphical ASM chart

Figure 1.3 shows the graphical ASM chart implementation, which consists of a total of eight states. The first state loads all of the registers that will take in the mantissa, exponent, and sign bits of both inputs A and B. Once loaded, the flag register will be set to 0, as a default for the overflow detection, and the exponent output register (REm) will take the sum of the two inputs exponents. The bias is then subtracted from the exponent to get the final exponent output. The input mantissas are then multiplied, and if the output of the multiplication is greater than or equal to 2 (i.e. 10.xxxxxxxx…) then the product needs to be normalized, which happens in the third state, by incrementing the exponent by 1, and right shifting the mantissa. The REm register is then checked to make sure it is within the range of -63 and +64, if not, the product is set to infinity, and the overflow flag is set to 1. Finally, the sign bit is computed, using a simple XOR operation, and the computation for AxB is complete.

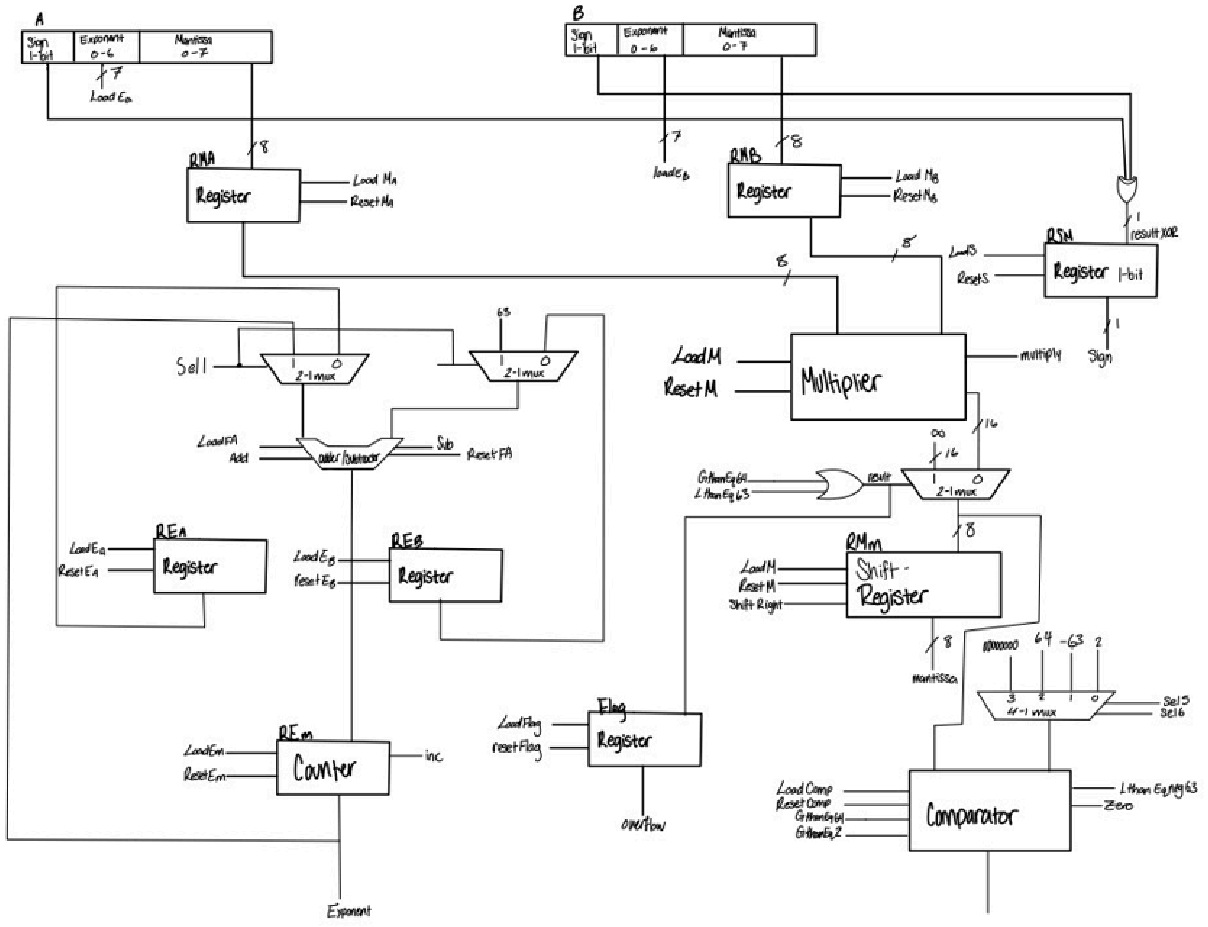


Figure 1.4 - Floating point multiplier ASM datapath

Figure 1.4 shows the datapath for the multiplier. The sign bit is calculated using an XOR gate, with inputs being the sign of A and B which is then stored in the RSm register, and outputted in the final state. The exponent is calculated by first taking exponent A and loading it into the REA register, and the same is done with exponent B to REB. The output of these two registers are sent to two different two-to-one multiplexers who's outputs are connected to an adder/subtractor. Both multiplexers take the same select signal, which will send exponent A and B to be added when set to 0, which will be stored in REm. when sel1 is equal to 1, the REm register will be subtracted by 63, and the final exponent will be computed. The mantissa will be calculated by first storing the A and B mantissas into their respective RMA or RMB registers, who's outputs are connected to a multiplier. The multiplier output is connected to a 2-1 multiplexer, which also takes as input an infinity signal. The select signal for this multiplexer is the OR operation of the GthanEq64 and LthanEq63 signals, from the comparator, which compares the exponent with the inputs from the 4-1 mux and asserts the relevant signal. Depending on the signal, the product or infinity will be sent to the RMm register, which will then be the final mantissa for the computation. The Flag register will also take the OR operation of the GthanEq64 and LthanEq63 signals and output the overflow based on those two signals.

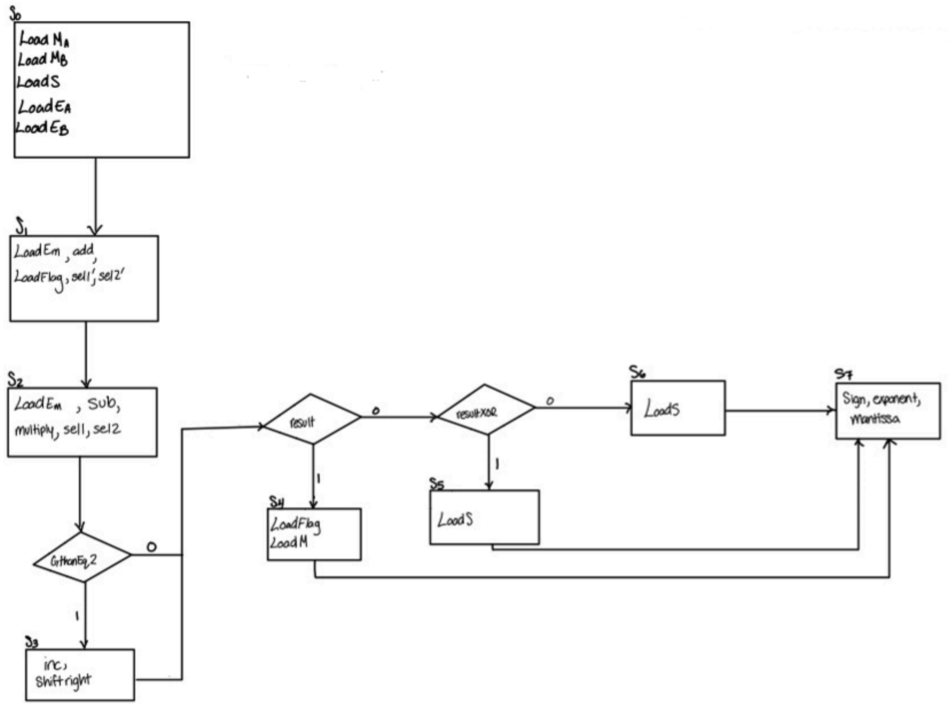


Figure 1.5 - Floating point multiplier detailed ASM chart

The control path (figure 1.5) uses signals found in the datapath. The first state (S0) takes all the load signals for the initial inputs. The decisions are made using the signals from the comparator, and the OR gate from the datapath in figure 1.4. If the GthanEq2 signal is on, then normalization will take place. The result signal is the output signal of the OR operation between the two signals GthanEq64 and LthanEq63 which will decide whether there is an overflow or not, and finally, the resultXOR signal which is the output signal of the XOR operation, which decides what the sign bit will be depending on the imputed sign bits.

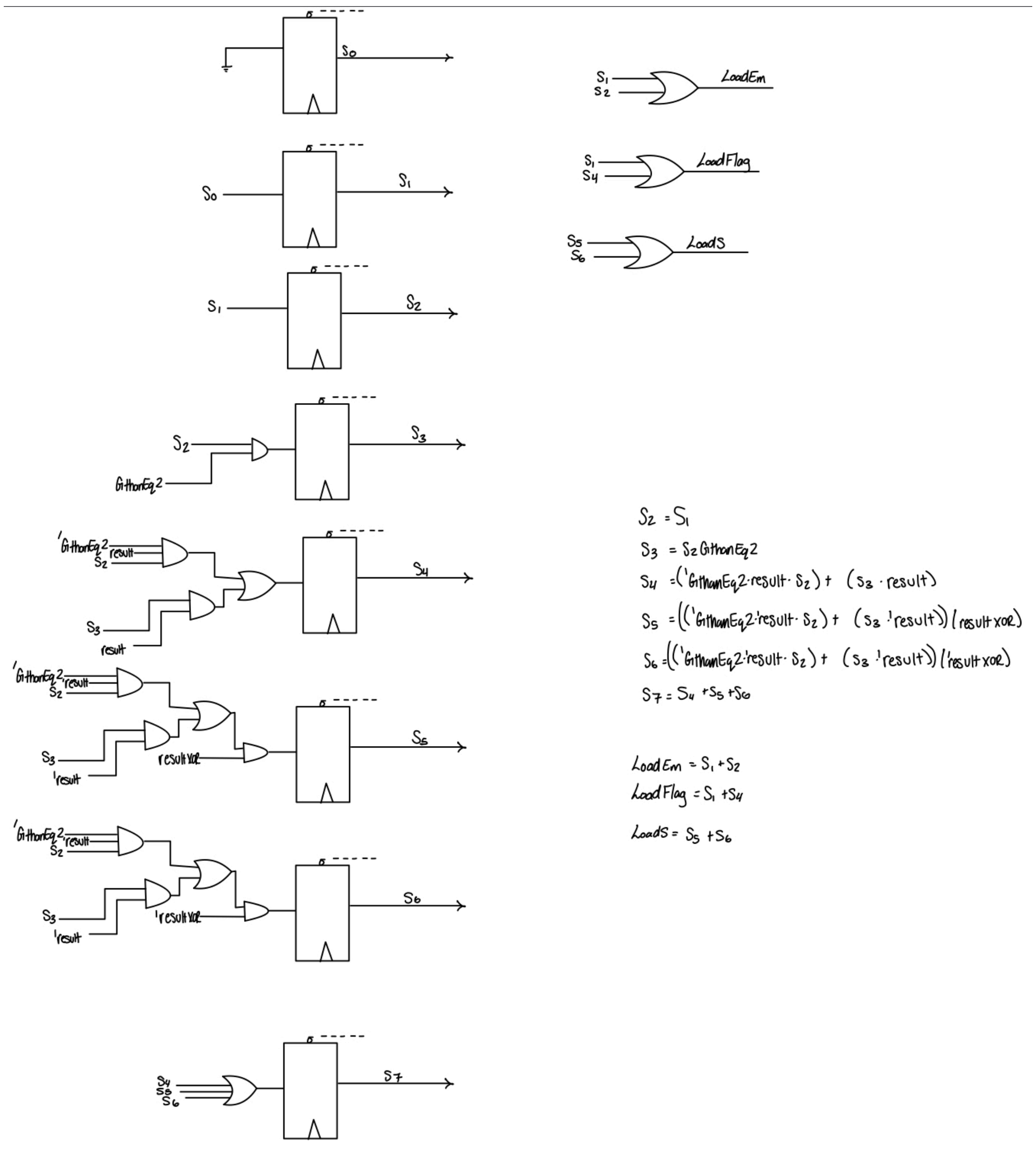


Figure 1.6 - Floating point multiplier ASM control path

The control path is shown in figure 1.6, and consists of 8 D-type flipflops for the 8 possible states. The S0 flip flop is connected to ground, and imputed for S1. The rest of the states take a combination of signals from the detailed ASM and datapath in order to output the correct state. There are three OR gaters,one each for LoadEm, LoadFlag, and LoadS, because these signals are asserted in more than one state.

Adder

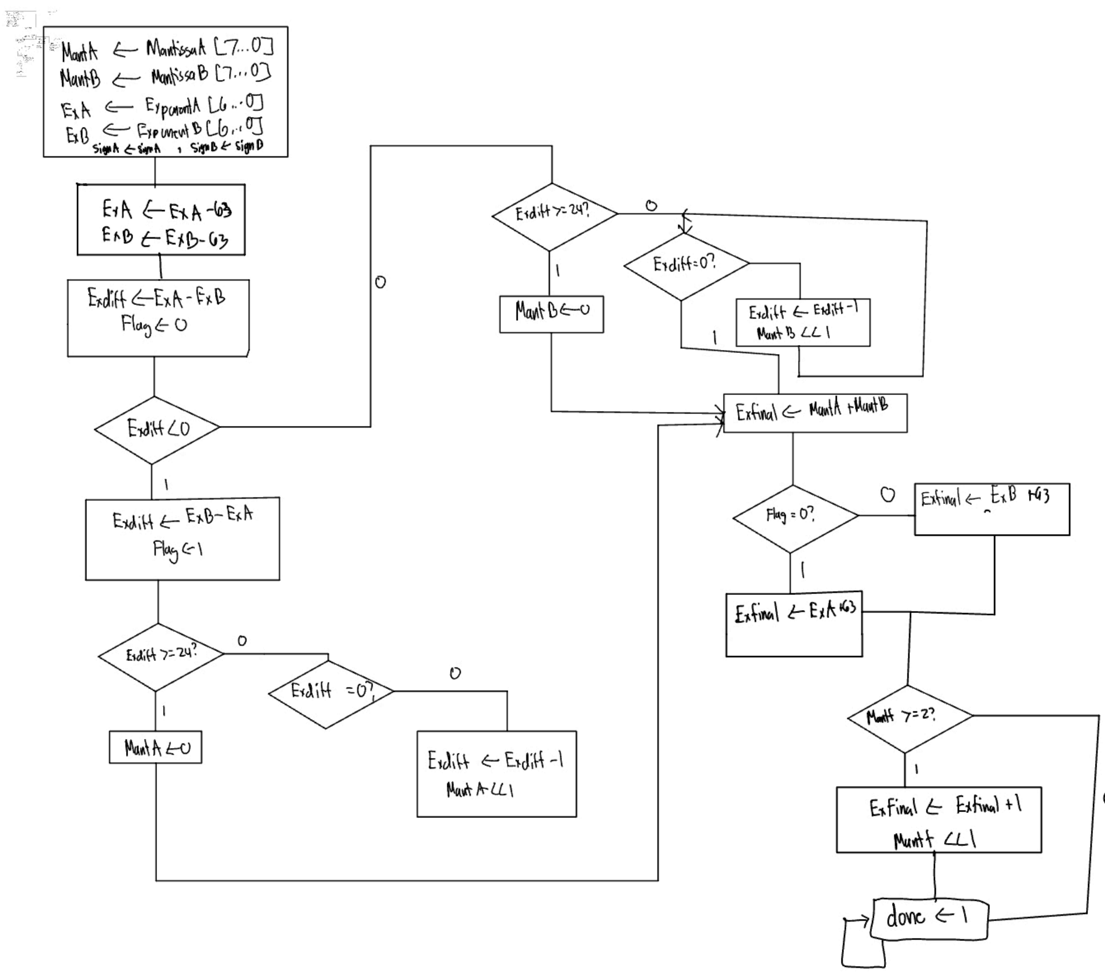


Figure 1.7 Floating Point Adder ASM Chart

The Floating Point Adder ASM chart, shown in Figure 1.7, begins by adjusting the exponents of both inputs by subtracting 63 to eliminate the excess 63 bias. Next, the exponent difference is computed to determine which of the two exponents is larger. The mantissa corresponding to the smaller exponent is then right-shifted, while its exponent is incremented until both exponents match. Once aligned, the mantissas are added, and the result undergoes a normalization process to ensure it fits within the valid range. Finally, the sum is checked for proper normalization and potential overflow before completing the computation.

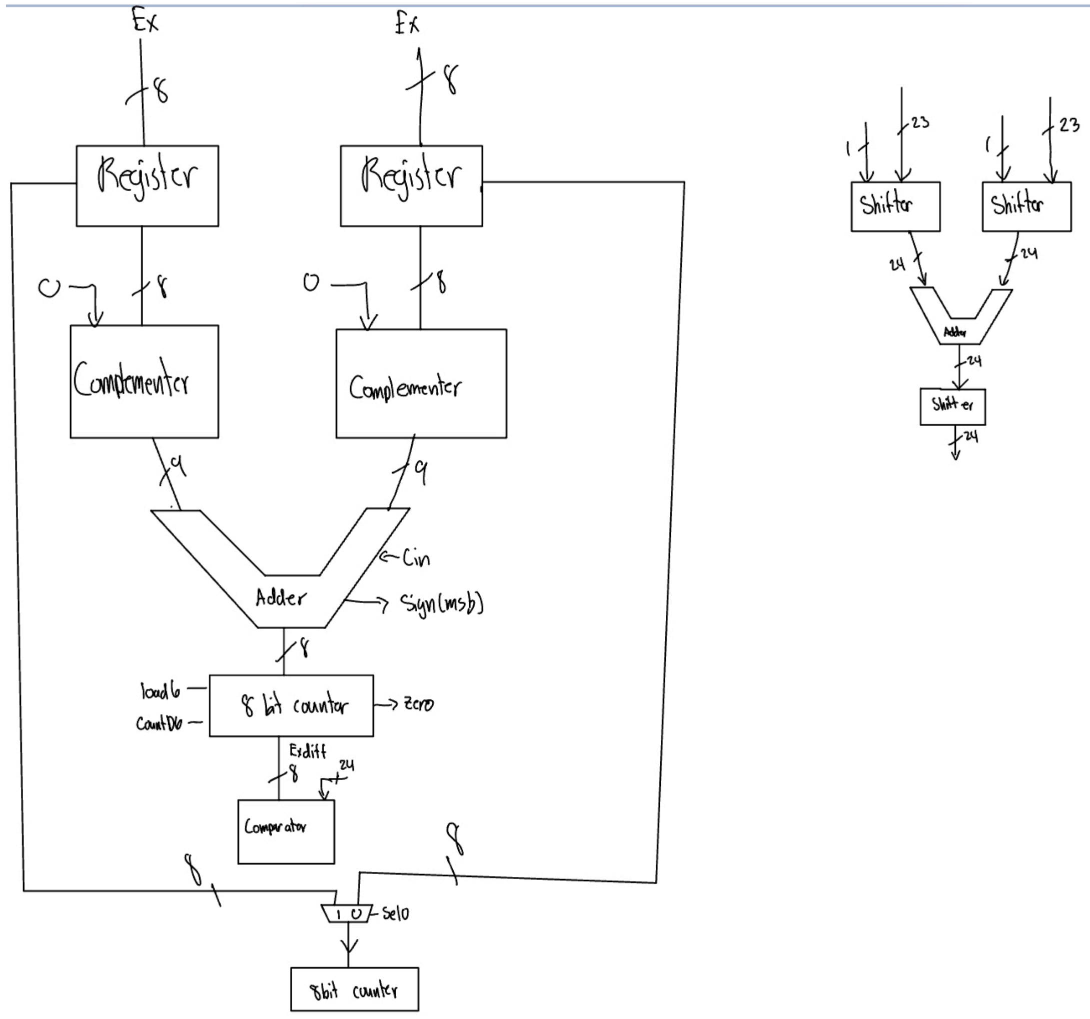


Figure 1.8 Floating Point Adder Datapath

Shown in Figure 1.8 is the Floating Point Adder Datapath. This floating-point datapath consists of two primary sections: exponent processing and mantissa alignment and addition. The exponent section begins with two registers that store the exponents of the input floating-point numbers. Each exponent passes through a complementor, which likely helps compute the exponent difference. These complemented values are then fed into an adder, which determines the difference and identifies the larger exponent. The result is stored in an 8-bit counter, which tracks shifts needed for alignment. A comparator then checks if the exponents are equal, guiding further processing. On the mantissa side, the two mantissas are fed into shifters, which align them based on the exponent difference. The aligned mantissas are then added together, and the result passes through another shifter for normalization. This ensures the final sum maintains proper floating-point representation before further processing or storage.

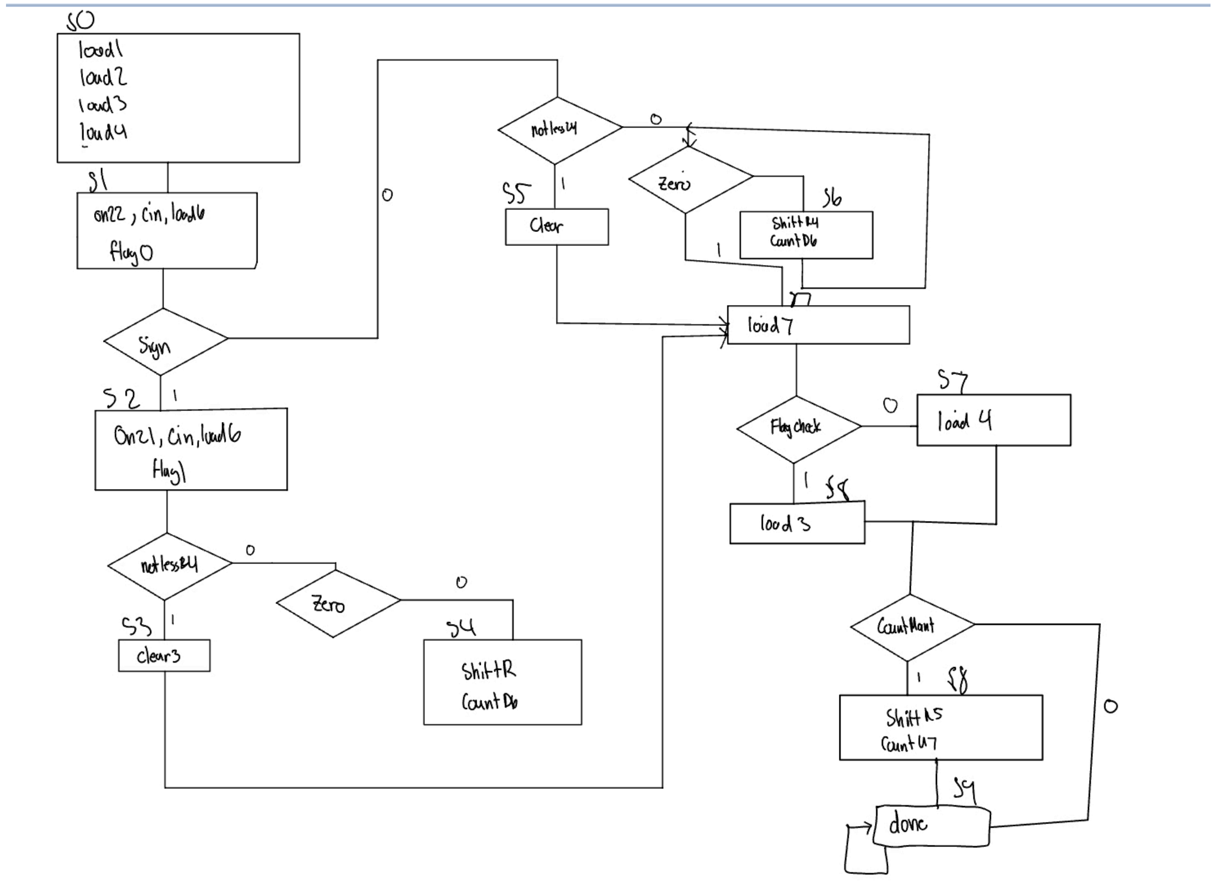


Figure 1.9 Floating Point Detailed ASM diagram

In Figure 1.9, the Floating Point Detailed ASM diagram is shown. It starts at S0, where initial values are loaded. At S1, operations such as OR-ing and loading specific values are performed, and the sign of the inputs is checked. Depending on the sign, the diagram either branches off or continues to S2, where another OR operation and additional computations take place. If the value is not set properly, it proceeds to S3 to clear registers. If the value is zero, it moves to S4, where a right shift operation is performed while decrementing a counter. If further adjustments are needed, the process flows through S5 and S6, where clearing, shifting, and exponent alignment occur. At S7, another load operation takes place, followed by flag checking at S8. If the flag check passes, it proceeds to load additional values in S9. Finally, in S10, the mantissa count is verified, and if necessary, shifting and counting adjustments are performed before reaching the final done state in S11, indicating the completion of the floating-point addition process.

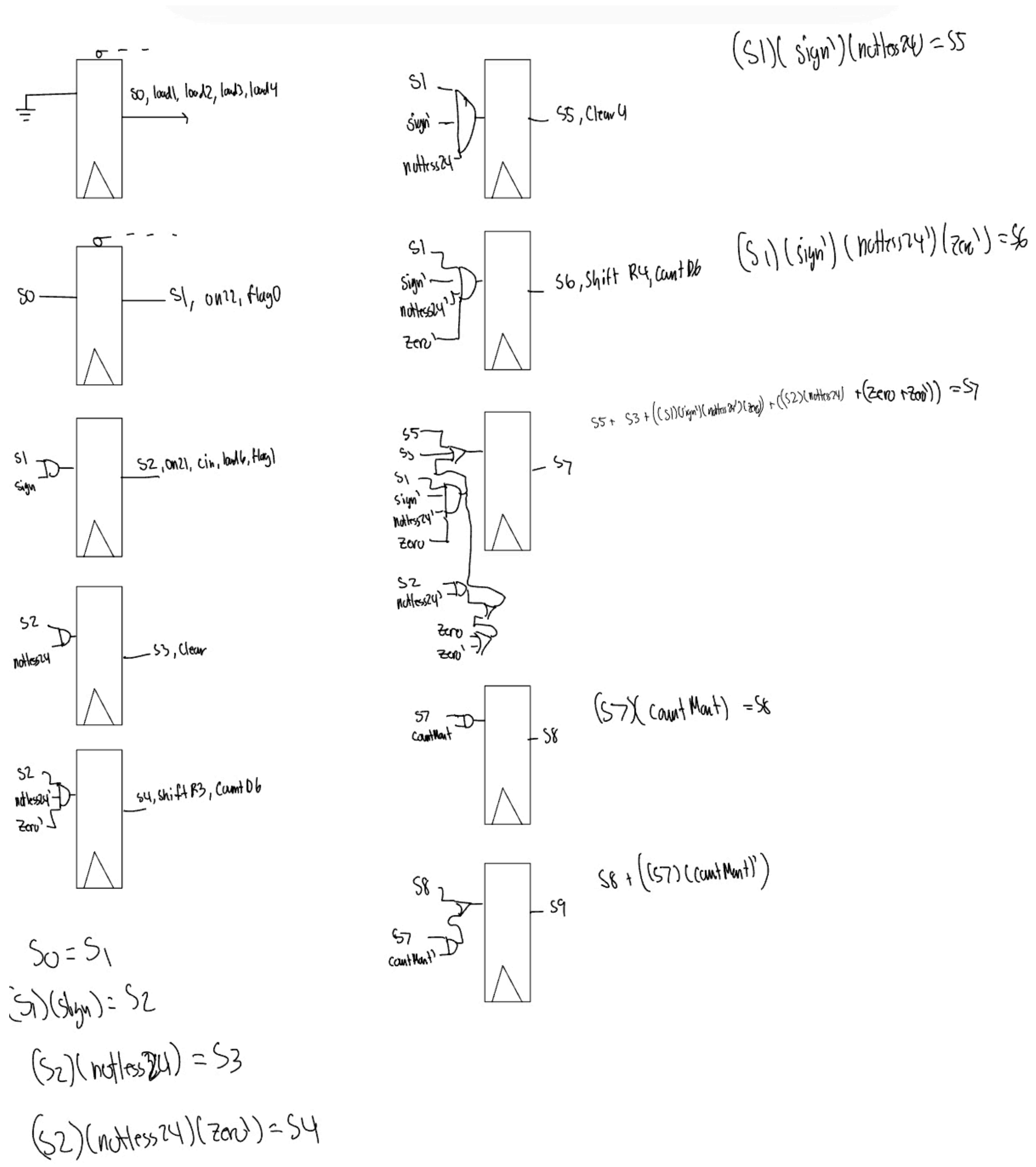


Figure 1.10 Floating Point ASM Control Path

The Floating Point ASM Control Path shown in Figure 1.10 consists of 10 Flip Flops for the 10 possible states. S0 is grounded and then imputed along with other signals to create S2. The other states also use signals taken from the detailed ASM diagram.

Design Part

Discussion of used components

Multiplier

The two main components used in the design of the multiplier were the control path and the datapath, designed using the ASM methodology, and further implemented in VHDL.

The floating point multiplier datapath is split into 3 categories, the sign bit, the exponent, and the mantissa (see Appendix A). The sign bit takes as input the user given signA and signB bits, and stores the XOR of that result in a D-type flip flop (enARdFF_2 component). The exponent consists of only a 7-bit adder/subtractor component, and an nBitRegister component, where n is set to 7. The user inputted exponents (ExponentA and ExponentB) are first added in the adderSubtractor component, and then that output (exponentAdd) is sent to another adderSubtractor, which subtracts 63 (0111111) from the exponentAdd value, to get the finalExponent value, which is the input for the RexpM register, that stores the final exponent value and outputs it to the user. There are intermediate registers as well, RexpB and RexpA, which take the imputed exponents and store then to be used in the first add operation. Finally, the mantissa portion of the datapath has two registers, RMA and RMB which store the imputed mantissa values from the user with an additional “1” appended to the start of the input. These registers subsequently pass those appended values to the topLevelMultiplier component which outputs a product that is sent to a twoToOneMux component which toggles between the product and infinity, depending on the signals obtained from the comparator. The nbitShiftRegister is used only if the product needs to be normalized. The flag component is set as a D-type flip flop, which will take as input GthanEq64 OR LthanEq63 to detect overflow. Finally, the comparator, which compares the exponent to ensure it remains within a range of -63 to 64, as well as compares to ensure that the product is not greater than 2 (otherwise normalize) and also sets a zero flag if the product is zero.

The control path contains 8 D-type flip flops for the 8 different possible states that the floating point multiplier traverses through (see Appendix A). The D-type flip flops correspond to the enARdFF_2 component. Within this architecture, dff1 corresponds to S0, dff2 corresponds to S1, and so one for each of the 8 states. The input to dff1 is connected to ‘1’ because it is the ground state as indicated in the ASM design from figure 1.6. There are otherwise three signals in this architecture that can be toggled in multiple states, LoadEM, loadFlag, and LoadS, which can be seen at the bottom of the control path with their respective state possibilities combined using an OR operation. The d0-d7 outputs are then connected to their respective state signals.

Adder

The components used in the floating-point adder consists of two main stages: exponent alignment and mantissa addition with normalization. In the exponent alignment stage, two 8-bit registers store the exponents of the input floating-point numbers. These exponents pass through complementers and an adder to compute the exponent difference, determining the necessary shift for mantissa alignment. An 8-bit counter tracks the exponent adjustment, while a comparator identifies the larger exponent to control the shift counter. In the mantissa addition and normalization stage, two 23-bit shifters align the mantissas based on the computed exponent difference. A 24-bit adder then performs the mantissa addition, and the result undergoes further shifting through a normalization shifter to produce the final correctly aligned sum.

Discussion of actual solution

Multiplier

The final floating point multiplier component was synthesized by combining the multiplier datapath and control path into one single multiplier top level entity (see Appendix A). This entity takes as input a signA and signB input used to get the signOut output. It also takes as input mantissaA and mantissaB, that will produce the final product outputted into mantissaOut, as well as exponentA and exponentB, that will produce the exponentOut output. The entity will also output overflow, if detected, and also take a clk and Greset signal. In the case of this top level entity, when Greset is high, the output will be seen. If Greset is set to '0', a global reset will occur. The two components in this entity are the datapath and the control path, where the datapath takes in all the user inputs, and the control path receives the declared signals.

Adder

The floating-point adder consists of a control unit and a datapath unit working together to execute the addition process. The control unit, implemented using D flip-flops, manages state transitions based on signals like Sign, Notless24, and Zero, determining operations such as shifting, loading, and adding. The datapath performs the arithmetic operations through key submodules, including register banks for operand storage, complementors for handling subtraction, a 9-bit adder for exponent alignment, an 8-bit counter for leading zero detection, a comparator for exponent comparison, and a 24-bit adder for mantissa addition. The addition process begins with exponent comparison, aligning the smaller mantissa using shift registers, followed by mantissa addition or subtraction. The result undergoes normalization, adjusting for leading zeros, and rounding if necessary.

Top Level Entity

The final top level entity was not successfully realized over the course of this lab, but it should combine both the floating point adder and multiplier developed over the course of this lab. It will take as input a signA and signB input used to get the signOut output, it will also take mantissaA and mantissaB, that will produce mantissaOut, as well as exponentA and exponentB, that will produce the exponentOut output. The entity also takes a clk and Greset input and a select input as well, to decide if multiplication or addition is the desired operation, and finally, an overflow output. The two components in this entity are the top level adder and multiplier entities, where both components take all the given user inputs, and if the user sets the select signal to ‘1’, the inputs will go through the floating point addition operation, otherwise, floating point multiplication will occur.

Discussion of tool

In this lab, the tools used were the Quartus II software and the Altera DE2 Board. The Quartus II software was used to code the VHDL and design the hardware implemented in the ASM methodology. The verification and simulation of the VHDL code was performed using the Altera DE2 board. More specifically, the switches on the board were used to manually set values for the inputs to the multiplier and the adder (sign, mantissa, and exponent). The calculations were simulated on the LEDS of the board.

Discussion of challenging problems

One of the challenges faced regarding the floating point multiplier specifically, was the exponent calculation. As can be seen in figure 1.4, the original ASM datapath for the multiplier regarding the exponent used two multiplexers, which took as input the RexpA and RexpB outputs for the addition of the exponents the first time around, and also the adderSubtractor output and 0111111 for the subtraction the second time through. However, this design proved to be problematic, since it created a loop, where the adderSubtractor output kept getting fed back in and recomputed, updating the RexpM register with every clock cycle, which was not the desired output. The solution was to replace that design with two adders instead, which can be seen in the datapath vhdl for the floating point multiplier (see Appendix A). This ensured that the imputed exponents were only added together once, and once added that result is fed into the next adderSubtractor component, where it is subtracted from the bias, only once, and that output is sent to the RexpM register.

Real Implementation

Shown simulation/synthesis results

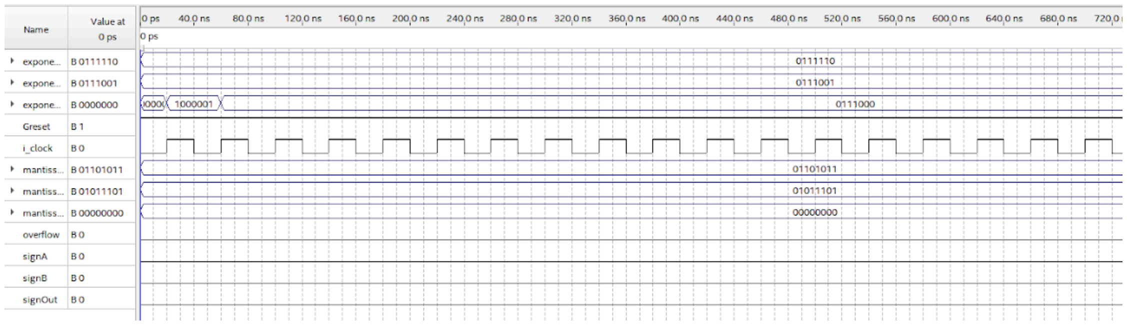


Figure 3.1 Floating Point Multiplier - Two positive numbers

The exponent calculation using excess 63 notation and the sign bit operation can be seen to work as expected in the above simulation waveform. When given two inputs, 62 and 57, the output exponent should be 56, which is the case for this simulation. The sign bits are both set to 0 to simulate the input of two positive numbers. The output sign bit is also working as expected.

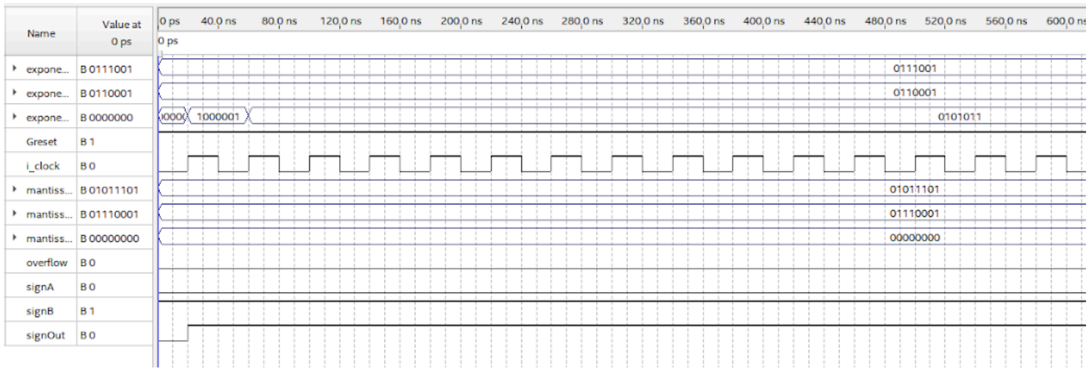


Figure 3.2 Floating Point Multiplier - One positive one negative number

Similar to the above simulation, this one is also outputting the exponent correctly, but here one of the numbers is negative, and therefore the output sign bit should be 1, which is the case, as can be seen in figure 3.2.

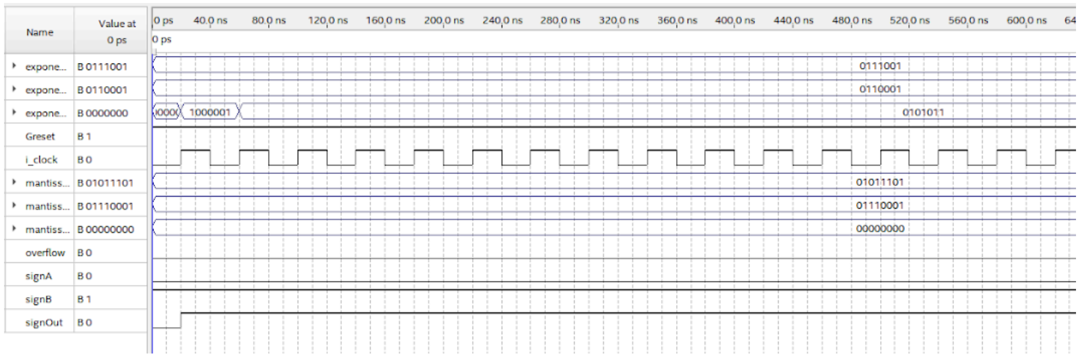
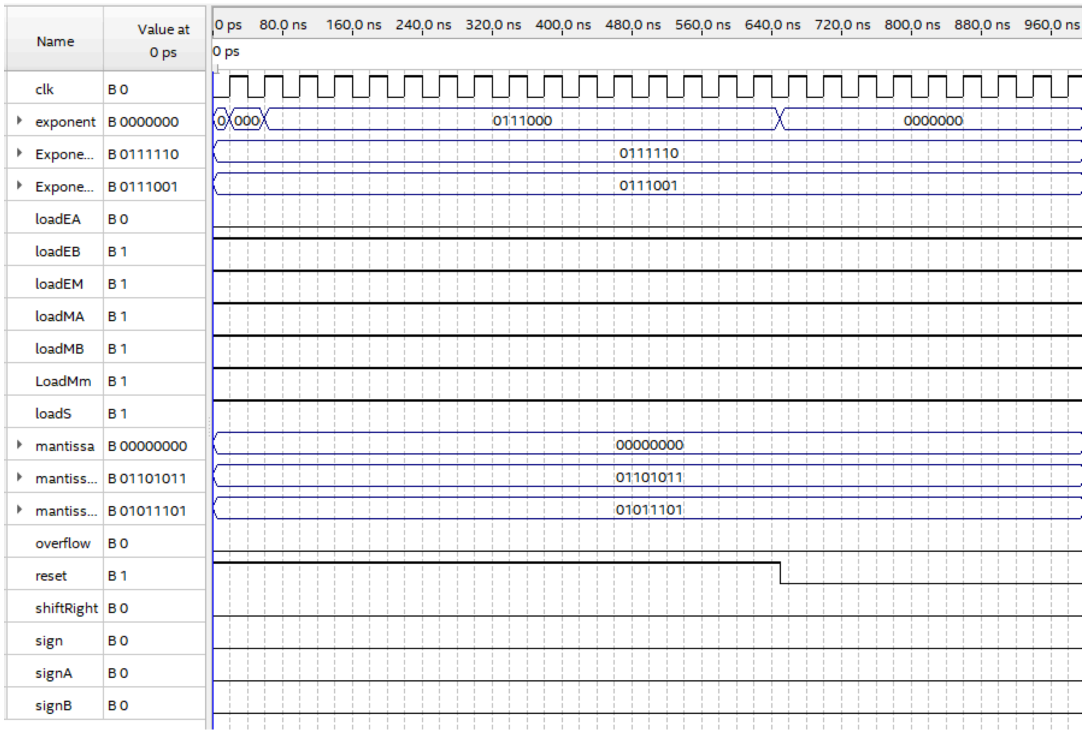
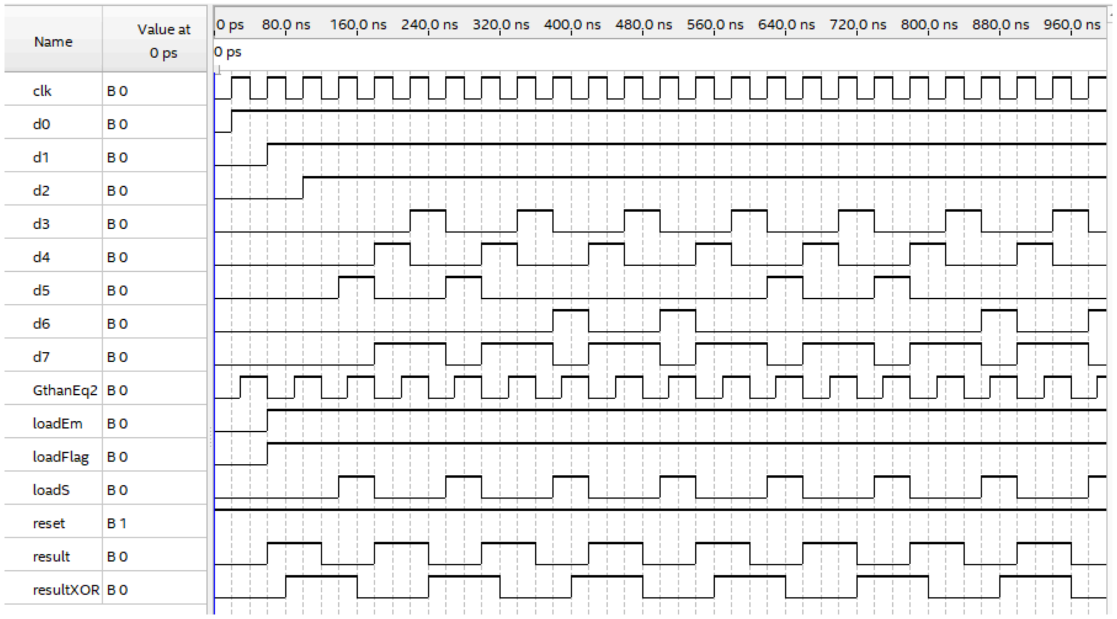


Figure 3.3 Floating Point Multiplier - Two negative numbers



Name	Value at 0 ps
clk	B 0
exponent	B 0000000
Expone...	B 0111110
Expone...	B 0111001
loadEA	B 0
loadEB	B 1
loadEM	B 1
loadMA	B 1
loadMB	B 1
LoadMm	B 1
loadS	B 1
mantissa	B 00000000
mantiss...	B 01101011
mantiss...	B 01011101
overflow	B 0
reset	B 1
shiftRight	B 0
sign	B 0
signA	B 0
signB	B 0

Figure 3.4 Floating Point Multiplier Datapath



Name	Value at 0 ps
clk	B 0
d0	B 0
d1	B 0
d2	B 0
d3	B 0
d4	B 0
d5	B 0
d6	B 0
d7	B 0
GthanEq2	B 0
loadEm	B 0
loadFlag	B 0
loadS	B 0
reset	B 1
result	B 0
resultXOR	B 0

Figure 3.5 Floating Point Multiplier Control Path

Verification

During this lab, the datapath, control path, and top level entity for the multiplier were all implemented, with successful implementation of the exponent and sign calculation. However, the mantissa did not produce the correct result. The adder in this lab was partially implemented. Both components were not demonstrated on the Altera board. The results obtained indicate that there were still errors present in the initial ASM design used to synthesize the VHDL code, particularly in the datapath implementation of the ASM process.

Discussion

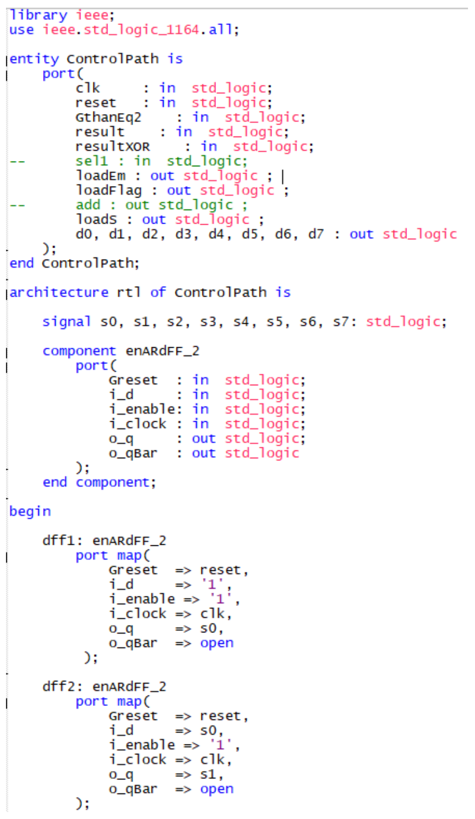
There were a few errors encountered over the course of this lab, specifically concerning the mantissa calculations, the output was always 0 in the floating point multiplier, which could be related to the logic of the topLevelMultiplier component used in the datapath to multiply the two inputs together. Another issue as discussed previously was regarding the exponents in the floating point multiplier, which was resolved by making changes within the datapath logic itself.

Conclusion

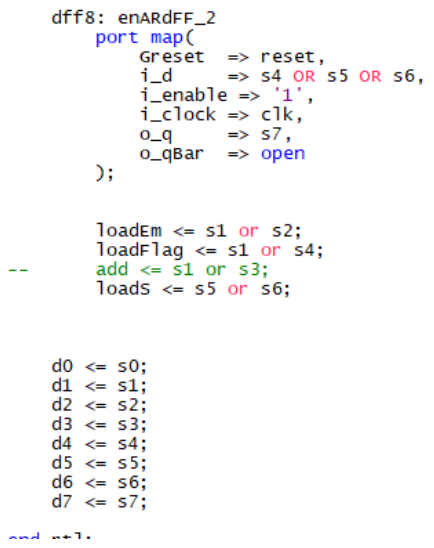
In conclusion, the objective of this lab was to design and implement both a floating point multiplier and a floating point adder using VHDL, and in the case of this implementation, the ASM methodology for the circuit design . Both multiplier and adder were partially successful, but a complete implementation was not reached within the scope of this lab.

Appendix A

A – 1 Multiplier Control Path VHDL Code



```
library ieee;
use ieee.std_logic_1164.all;

entity ControlPath is
    port(
        clk      : in  std_logic;
        reset    : in  std_logic;
        GthanEq2    : in  std_logic;
        result     : in  std_logic;
        resultXOR    : in  std_logic;
--      sel1 : in  std_logic;
        loadEm : out std_logic ;
        loadFlag : out std_logic ;
--      add : out std_logic ;
        loadS : out std_logic ;
        d0, d1, d2, d3, d4, d5, d6, d7 : out std_logic
    );
end ControlPath;

architecture rtl of ControlPath is

    signal s0, s1, s2, s3, s4, s5, s6, s7: std_logic;

    component enARdFF_2
        port(
            Greset   : in  std_logic;
            i_d      : in  std_logic;
            i_enable: in  std_logic;
            i_clock : in  std_logic;
            o_q      : out std_logic;
            o_qBar   : out std_logic
        );
    end component;

begin

    dff1: enARdFF_2
        port map(
            Greset   => reset,
            i_d      => '1',
            i_enable => '1',
            i_clock => clk,
            o_q      => s0,
            o_qBar   => open
         );

    dff2: enARdFF_2
        port map(
            Greset   => reset,
            i_d      => s0,
            i_enable => '1',
            i_clock => clk,
            o_q      => s1,
            o_qBar   => open
        );
```



```
    dff8: enARdFF_2
        port map(
            Greset   => reset,
            i_d      => s4 OR s5 OR s6,
            i_enable => '1',
            i_clock => clk,
            o_q      => s7,
            o_qBar   => open
        );


        loadEm <= s1 or s2;
        loadFlag <= s1 or s4;
--      add <= s1 or s3;
        loadS <= s5 or s6;


    d0 <= s0;
    d1 <= s1;
    d2 <= s2;
    d3 <= s3;
    d4 <= s4;
    d5 <= s5;
    d6 <= s6;
    d7 <= s7;

end rtl;
```

A – 2 Multiplier Datapath VHDL Code

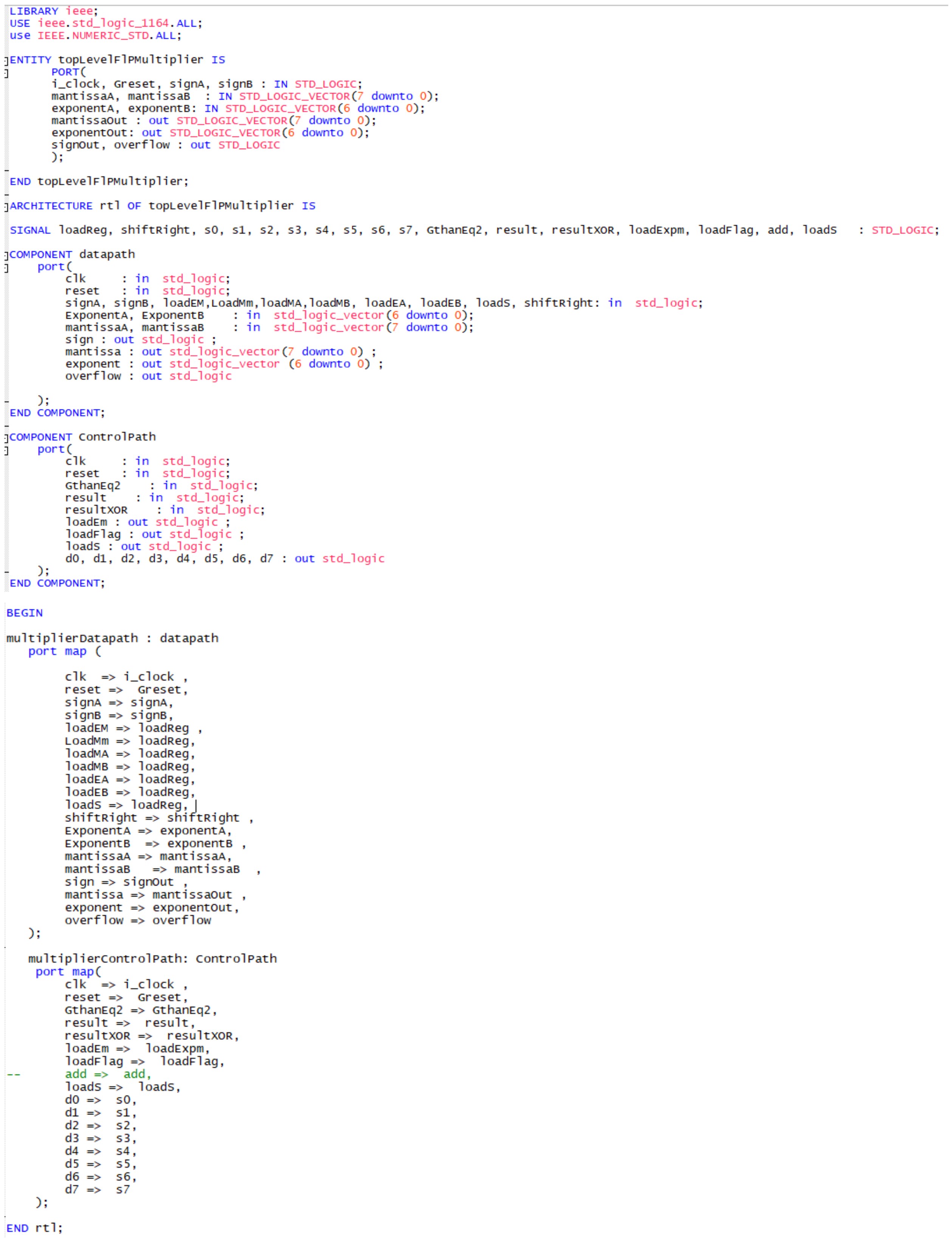
```
library ieee;
use ieee.std_logic_1164.all;

entity datapath is
    port(
        clk     : in  std_logic;
        reset   : in  std_logic;
        signA, signB, loadEM,LoadMm,loadMA,loadMB, loadEA, loadEB, loadS, shiftRight: in  std_logic;
        ExponentA, ExponentB     : in  std_logic_vector(6 downto 0);
        mantissaA, mantissaB     : in  std_logic_vector(7 downto 0);
        sign : out std_logic ;
        mantissa : out std_logic_vector(7 downto 0) ;
        exponent : out std_logic_vector (6 downto 0) ;
        overflow : out std_logic

    );
end datapath;

architecture rtl OF datapath IS

SIGNAL mux1out, mux2out, exponentOut, exponentAout, exponentAdd, exponentBout, Artexponent, finalExponent, mux4out : std_logic_vector(6 downto 0);
SIGNAL  sel3, sel4, cin, cout,addSub, GthanEq64, LthanEqNeg63, eq2, zero, loadComp, loadFlag, loadCount, signOut, select1, Greset: std_logic ;
SIGNAL mantissaAout, mantissaBout :  std_logic_vector(8 downto 0);
SIGNAL mux3out  :  std_logic_vector(7 downto 0);
SIGNAL product, inf:  std_logic_vector(17 downto 0);
SIGNAL operationCounter : STD_LOGIC_VECTOR(1 DOWNTO 0) := "00";


 COMPONENT sevenBtwoToOneMux
     PORT(
        D0, D1 : IN STD_LOGIC_VECTOR(6 downto 0);
        S0: in STD_LOGIC;
        M: out STD_LOGIC_VECTOR(6 downto 0)
        );
 END COMPONENT;

  COMPONENT eightBtwoToOneMux
     PORT(
        D0, D1 : IN STD_LOGIC_VECTOR(7 downto 0);
        S0: in STD_LOGIC;
        M: out STD_LOGIC_VECTOR(7 downto 0)
        );
 END COMPONENT;

 COMPONENT adderSubtractor
    GENERIC(
            N : INTEGER := 7
        );
        PORT(
              cin        : IN STD_LOGIC;
              a      : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
              b      : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
              cout       : OUT STD_LOGIC;
              sub    : IN STD_LOGIC;
              fsout       : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0)
        );
    END COMPONENT;
    COMPONENT nBitRegister

 | COMPONENT nBitRegister
     GENERIC (
         N : INTEGER
     );
     PORT(
        Greset   : IN STD_LOGIC;
        i_load   : IN STD_LOGIC;
        i_clock  : IN STD_LOGIC;
        i_Value  : IN STD_LOGIC_VECTOR(N-1 downto 0);
        o_Value  : OUT STD_LOGIC_VECTOR(N-1 downto 0)
     );
 END COMPONENT;

  COMPONENT enARdFF_2
     PORT(
         Greset  : in  std_logic;
         i_d     : in  std_logic;
         i_enable: in  std_logic;
         i_clock : in  std_logic;
         o_q     : out std_logic;
         o_qBar  : out std_logic
     );
 end COMPONENT;

 COMPONENT topLevelMultiplier
   PORT(
   i_clock, Greset : IN STD_LOGIC;
   A_in, B_in  : IN STD_LOGIC_VECTOR(8 downto 0);
   Product : out STD_LOGIC_VECTOR(17 downto 0)
   );
 end COMPONENT;

 COMPONENT  nbitShiftRegister
 GENERIC (
     N : INTEGER
 );
 PORT(
     ClrA    : IN STD_LOGIC;
     loadA   : IN STD_LOGIC;
     rightA  : IN STD_LOGIC;
     i_clock : IN STD_LOGIC;
     i_Value : IN STD_LOGIC_VECTOR(N-1 downto 0);
     o_Value : OUT STD_LOGIC_VECTOR(N-1 downto 0)
 );
 end COMPONENT;

  COMPONENT FourToOneMux
   PORT(
     D0, D1, D2, D3 : IN STD_LOGIC_VECTOR(6 downto 0);
     S0, S1: in STD_LOGIC;
     M: out STD_LOGIC_VECTOR(6 downto 0)
   );
 end COMPONENT;


  COMPONENT Comparator
   Port (
     A       : in  STD_LOGIC_VECTOR(6 downto 0);
     B       : in  STD_LOGIC_VECTOR(6 downto 0);
     load    : in  STD_LOGIC;
     equal   : out STD_LOGIC;
     greater : out STD_LOGIC;
     less    : out STD_LOGIC
 );
 end COMPONENT;
```

```
      COMPONENT counter
        PORT(
            Greset   : IN  std_logic;
            i_clock  : IN  std_logic;
            Enable   : IN  std_logic;
            o_Value   : OUT std_logic_vector(1 downto 0)
        );
    END COMPONENT;



   BEGIN

---Sign Start------
    dff1: enARdFF_2
        port map(
            Greset  => reset,
            i_d     => signA XOR signB,
            i_enable => '1',
            i_clock => clk,
            o_q     => signOut,
            o_qBar  => open
         );


---Sign End------

---Exponent Start------ |

    exponentCalcAdd: adderSubtractor
       PORT MAP(
           cin  => '0',
           a    => exponentAout,
           b      => exponentBout,
           cout  => open,
           sub  => '0',
           fsout => exponentAdd
   );

    exponentCalcSub: adderSubtractor
       PORT MAP(
           cin  => '0',
           a    => exponentAdd,
           b      => "0111111",
           cout  => open,
           sub  => '1',
           fsout => finalExponent
   );


   RexpA: nBitRegister
        GENERIC MAP (
            N => 7
        )
        PORT MAP(
           Greset   =>  reset,
           i_load  => '1',
           i_clock => clk,
           i_Value => ExponentA,
           o_Value => exponentAout
        );
```

```vhdl
RexpB: nBitRegister
      GENERIC MAP (
            N => 7
      )
      PORT MAP(
         Greset   => reset,
         i_load  => '1',
         i_clock => clk,
         i_Value => ExponentB,
         o_Value => exponentBout
      );

RexpM: nBitRegister
      GENERIC MAP (
            N => 7
      )
      PORT MAP(
         Greset   => reset,
         i_load  => '1',
         i_clock => clk,
         i_Value => finalExponent,
         o_Value => exponentOut
      );

---Exponent End------

---Mantissa Start------
   RMA: nBitRegister
      GENERIC MAP (
            N => 9
      )
      PORT MAP(
         Greset   => reset,
         i_load  => loadMA,
         i_clock => clk,
         i_Value => '1' & mantissaA,
         o_Value => mantissaAout
      );
   RMB: nBitRegister
      GENERIC MAP (
            N => 9
      )
      PORT MAP(
         Greset   => reset,
         i_load  => loadMB,
         i_clock => clk,
         i_Value => '1' & mantissaB,
         o_Value => mantissaBout
      );

   multiply: topLevelMultiplier
      PORT MAP(
      i_clock => clk,
      Greset => reset,
      A_in => mantissaAout,
      B_in => mantissaBout,
      Product => product
      );
```

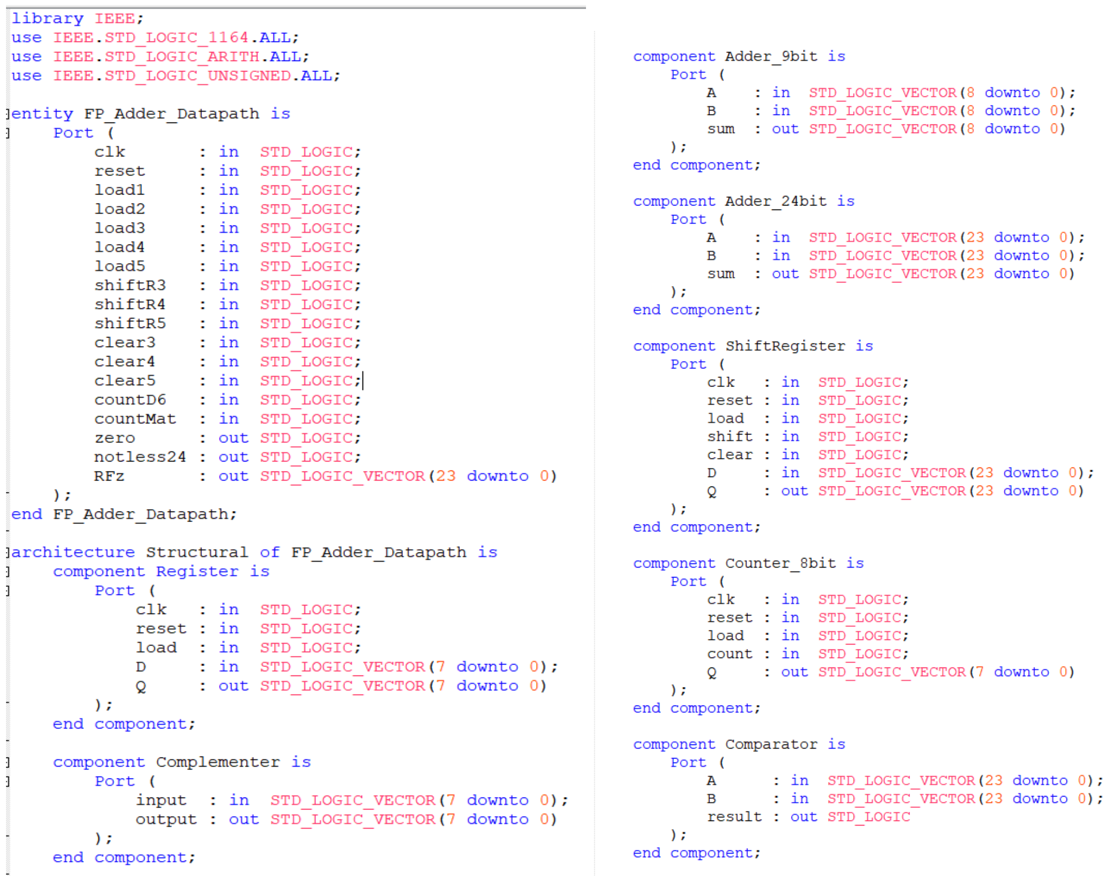
```vhdl
   mux3: eightBtwoToOneMux
      PORT MAP(
      D0 => inf(15 downto 8),
      D1 => product(15 downto 8),
      S0 => GthanEq64 OR LthanEqNeg63,
      M => mux3out
   );

   RMm: nbitShiftRegister
      GENERIC MAP (
            N => 8
      )
      PORT MAP(

         ClrA    => reset,
         loadA  => loadMB,
         rightA => shiftRight,
         i_clock => clk,
         i_Value => mux3out,
         o_Value => mantissa
      );

    mux4: FourToOneMux
      PORT MAP(
         D0 => "0000000",
         D1 => "1000000",
         D2 => "0111111",
         D3 => "0000010",
         S0 => sel3,
         S1 => sel4,
         M => mux4out
      );
   compare: Comparator
      PORT MAP (
         A    =>    mux4out,
         B     =>  finalExponent,
         load    => loadComp,
         equal   => eq2,
         greater => GthanEq64,
         less   => LthanEqNeg63
      );

      Flag: enARdFF_2
         port map(
            Greset   => reset,
            i_d      => LthanEqNeg63 OR GthanEq64,
            i_enable => '1',
            i_clock => clk,
            o_q      => overflow,
            o_qBar   => open
         );

---Mantissa End------

exponent <= exponentOut;
sign <= signOut;


END rtl;
```

A – 3 Multiplier Top Level Entity VHDL Code

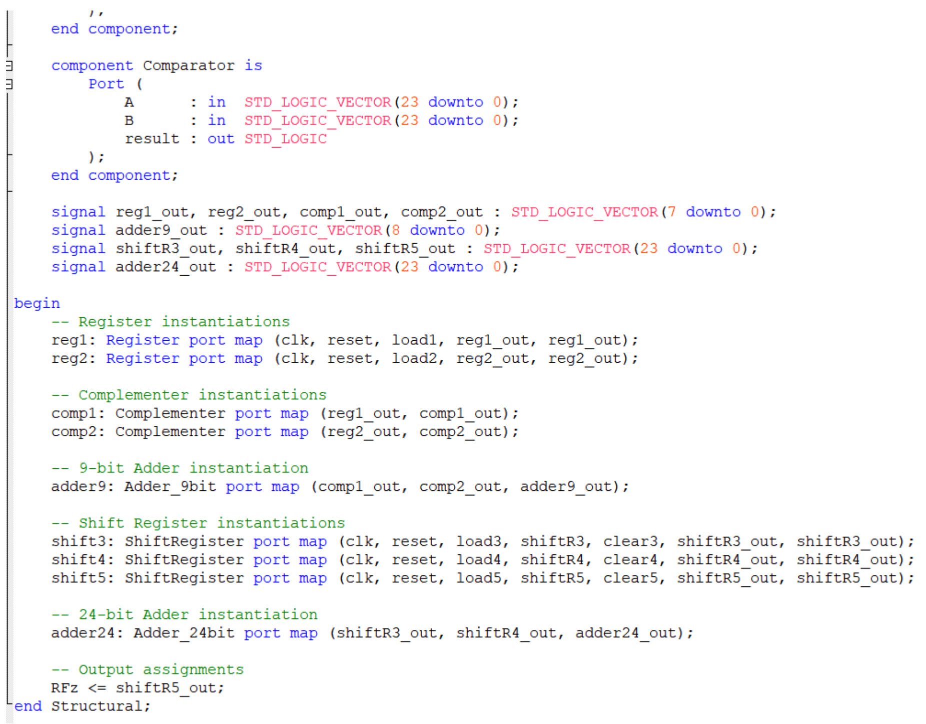


```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

ENTITY topLevelFlPMultiplier IS
      PORT(
      i_clock, Greset, signA, signB : IN STD_LOGIC;
      mantissaA, mantissaB  : IN STD_LOGIC_VECTOR(7 downto 0);
      exponentA, exponentB: IN STD_LOGIC_VECTOR(6 downto 0);
      mantissaOut : out STD_LOGIC_VECTOR(7 downto 0);
      exponentOut: out STD_LOGIC_VECTOR(6 downto 0);
      signOut, overflow : out STD_LOGIC
      );

END topLevelFlPMultiplier;

ARCHITECTURE rtl OF topLevelFlPMultiplier IS

SIGNAL loadReg, shiftRight, s0, s1, s2, s3, s4, s5, s6, s7, GthanEq2, result, resultXOR, loadExpm, loadFlag, add, loadS   : STD_LOGIC;

COMPONENT datapath
    port(
        clk     : in  std_logic;
        reset   : in  std_logic;
        signA, signB, loadEM,LoadMm,loadMA,loadMB, loadEA, loadEB, loadS, shiftRight: in  std_logic;
        ExponentA, ExponentB     : in  std_logic_vector(6 downto 0);
        mantissaA, mantissaB     : in  std_logic_vector(7 downto 0);
        sign : out std_logic ;
        mantissa : out std_logic_vector(7 downto 0) ;
        exponent : out std_logic_vector (6 downto 0) ;
        overflow : out std_logic

    );
END COMPONENT;

COMPONENT ControlPath
    port(
        clk     : in  std_logic;
        reset   : in  std_logic;
        GthanEq2    : in  std_logic;
        result     : in  std_logic;
        resultXOR    : in  std_logic;
        loadEm : out std_logic ;
        loadFlag : out std_logic ;
        loadS : out std_logic ;
        d0, d1, d2, d3, d4, d5, d6, d7 : out std_logic
    );
END COMPONENT;

BEGIN

multiplierDatapath : datapath
   port map (

        clk   => i_clock ,
        reset =>  Greset,
        signA => signA,
        signB => signB,
        loadEM => loadReg ,
        LoadMm => loadReg,
        loadMA => loadReg,
        loadMB => loadReg,
        loadEA => loadReg,
        loadEB => loadReg,
        loadS => loadReg, |
        shiftRight => shiftRight ,
        ExponentA => exponentA,
        ExponentB  => exponentB ,
        mantissaA => mantissaA,
        mantissaB    => mantissaB   ,
        sign => signOut ,
        mantissa => mantissaOut ,
        exponent => exponentOut,
        overflow => overflow
   );

   multiplierControlPath: ControlPath
    port map(
        clk   => i_clock ,
        reset =>  Greset,
        GthanEq2 => GthanEq2,
        result =>  result,
        resultXOR =>  resultXOR,
        loadEm =>  loadExpm,
        loadFlag =>  loadFlag,
--      add =>  add,
        loadS =>  loadS,
        d0 =>  s0,
        d1 =>  s1,
        d2 =>  s2,
        d3 =>  s3,
        d4 =>  s4,
        d5 =>  s5,
        d6 =>  s6,
        d7 =>  s7
    );

END rtl;
```

A - 4 Adder Datapath VHDL Code



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

entity FP_Adder_Datapath is
    Port (
        clk       : in  STD_LOGIC;
        reset     : in  STD_LOGIC;
        load1     : in  STD_LOGIC;
        load2     : in  STD_LOGIC;
        load3     : in  STD_LOGIC;
        load4     : in  STD_LOGIC;
        load5     : in  STD_LOGIC;
        shiftR3   : in  STD_LOGIC;
        shiftR4   : in  STD_LOGIC;
        shiftR5   : in  STD_LOGIC;
        clear3    : in  STD_LOGIC;
        clear4    : in  STD_LOGIC;
        clear5    : in  STD_LOGIC;
        countD6   : in  STD_LOGIC;
        countMat  : in  STD_LOGIC;
        zero      : out STD_LOGIC;
        notless24 : out STD_LOGIC;
        RFz       : out STD_LOGIC_VECTOR(23 downto 0)
    );
end FP_Adder_Datapath;

architecture Structural of FP_Adder_Datapath is
    component Register is
        Port (
            clk   : in  STD_LOGIC;
            reset : in  STD_LOGIC;
            load  : in  STD_LOGIC;
            D     : in  STD_LOGIC_VECTOR(7 downto 0);
            Q     : out STD_LOGIC_VECTOR(7 downto 0)
        );
    end component;

    component Complementer is
        Port (
            input  : in  STD_LOGIC_VECTOR(7 downto 0);
            output : out STD_LOGIC_VECTOR(7 downto 0)
        );
    end component;

    component Adder_9bit is
        Port (
            A   : in  STD_LOGIC_VECTOR(8 downto 0);
            B   : in  STD_LOGIC_VECTOR(8 downto 0);
            sum : out STD_LOGIC_VECTOR(8 downto 0)
        );
    end component;

    component Adder_24bit is
        Port (
            A   : in  STD_LOGIC_VECTOR(23 downto 0);
            B   : in  STD_LOGIC_VECTOR(23 downto 0);
            sum : out STD_LOGIC_VECTOR(23 downto 0)
        );
    end component;

    component ShiftRegister is
        Port (
            clk   : in  STD_LOGIC;
            reset : in  STD_LOGIC;
            load  : in  STD_LOGIC;
            shift : in  STD_LOGIC;
            clear : in  STD_LOGIC;
            D     : in  STD_LOGIC_VECTOR(23 downto 0);
            Q     : out STD_LOGIC_VECTOR(23 downto 0)
        );
    end component;

    component Counter_8bit is
        Port (
            clk   : in  STD_LOGIC;
            reset : in  STD_LOGIC;
            load  : in  STD_LOGIC;
            count : in  STD_LOGIC;
            Q     : out STD_LOGIC_VECTOR(7 downto 0)
        );
    end component;

    component Comparator is
        Port (
            A      : in  STD_LOGIC_VECTOR(23 downto 0);
            B      : in  STD_LOGIC_VECTOR(23 downto 0);
            result : out STD_LOGIC
        );
    end component;
```



```vhdl
        );
    end component;

    component Comparator is
        Port (
            A      : in  STD_LOGIC_VECTOR(23 downto 0);
            B      : in  STD_LOGIC_VECTOR(23 downto 0);
            result : out STD_LOGIC
        );
    end component;
```

```vhdl
    signal reg1_out, reg2_out, comp1_out, comp2_out : STD_LOGIC_VECTOR(7 downto 0);
    signal adder9_out : STD_LOGIC_VECTOR(8 downto 0);
    signal shiftR3_out, shiftR4_out, shiftR5_out : STD_LOGIC_VECTOR(23 downto 0);
    signal adder24_out : STD_LOGIC_VECTOR(23 downto 0);

begin
    -- Register instantiations
    reg1: Register port map (clk, reset, load1, reg1_out, reg1_out);
    reg2: Register port map (clk, reset, load2, reg2_out, reg2_out);

    -- Complementer instantiations
    comp1: Complementer port map (reg1_out, comp1_out);
    comp2: Complementer port map (reg2_out, comp2_out);

    -- 9-bit Adder instantiation
    adder9: Adder_9bit port map (comp1_out, comp2_out, adder9_out);

    -- Shift Register instantiations
    shift3: ShiftRegister port map (clk, reset, load3, shiftR3, clear3, shiftR3_out, shiftR3_out);
    shift4: ShiftRegister port map (clk, reset, load4, shiftR4, clear4, shiftR4_out, shiftR4_out);
    shift5: ShiftRegister port map (clk, reset, load5, shiftR5, clear5, shiftR5_out, shiftR5_out);

    -- 24-bit Adder instantiation
    adder24: Adder_24bit port map (shiftR3_out, shiftR4_out, adder24_out);

    -- Output assignments
    RFz <= shiftR5_out;
end Structural;
```

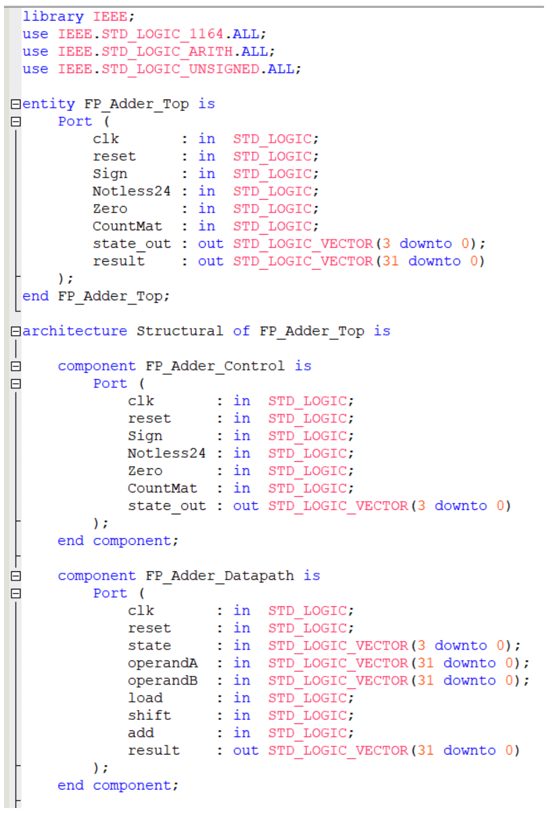
A - 5 Adder Control Path VHDL Code

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

entity FP_Adder_Control is
    Port (
        clk       : in  STD_LOGIC;
        reset     : in  STD_LOGIC;
        Sign      : in  STD_LOGIC;
        Notless24 : in  STD_LOGIC;
        Zero      : in  STD_LOGIC;
        CountMat  : in  STD_LOGIC;
        state_out : out STD_LOGIC_VECTOR(3 downto 0)
    );
end FP_Adder_Control;

architecture Structural of FP_Adder_Control is
    component enARDFF_2 is
        Port (
            Greset   : in  STD_LOGIC;
            i_d      : in  STD_LOGIC;
            i_enable : in  STD_LOGIC;
            i_clock  : in  STD_LOGIC;
            o_q      : out STD_LOGIC;
            o_qBar   : out STD_LOGIC
        );
    end component;

    signal S0, S1, S2, S3, S4, S5, S6, S7, S8, S9 : STD_LOGIC;

begin
    -- State Flip-Flops Implementation
    dff0: enARDFF_2 port map (
        Greset   => reset,
        i_d      => '1',
        i_enable => '1',
        i_clock  => clk,
        o_q      => S0,
        o_qBar   => open
    );
```

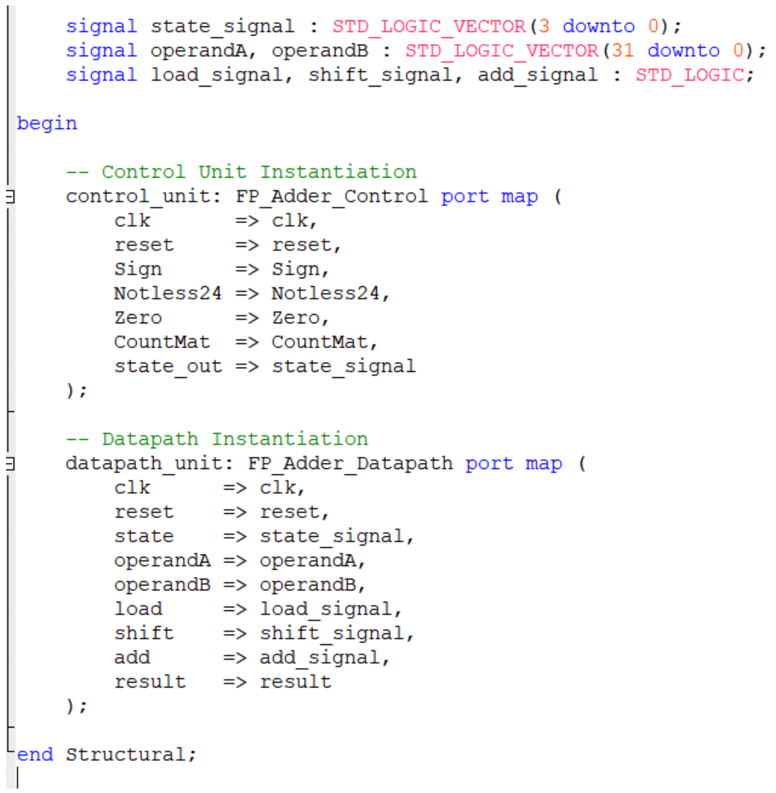
```vhdl
    dff1: enARDFF_2 port map (
        Greset   => reset,
        i_d      => S0 and Sign,
        i_enable => '1',
        i_clock  => clk,
        o_q      => S1,
        o_qBar   => open
    );

    dff2: enARDFF_2 port map (
        Greset   => reset,
        i_d      => S1 and Sign,
        i_enable => '1',
        i_clock  => clk,
        o_q      => S2,
        o_qBar   => open
    );

    dff3: enARDFF_2 port map (
        Greset   => reset,
        i_d      => S2 and Notless24,
        i_enable => '1',
        i_clock  => clk,
        o_q      => S3,
        o_qBar   => open
    );

    dff4: enARDFF_2 port map (
        Greset   => reset,
        i_d      => S2 and Notless24 and (not Zero),
        i_enable => '1',
        i_clock  => clk,
        o_q      => S4,
        o_qBar   => open
    );

    dff5: enARDFF_2 port map (
        Greset   => reset,
        i_d      => S1 and (not Sign) and Notless24,
        i_enable => '1',
        i_clock  => clk,
        o_q      => S5,
        o_qBar   => open
    );
```

```vhdl
    dff6: enARDFF_2 port map (
        Greset   => reset,
        i_d      => S1 and (not Sign) and (not Notless24) and (not Zero),
        i_enable => '1',
        i_clock  => clk,
        o_q      => S6,
        o_qBar   => open
    );

    dff7: enARDFF_2 port map (
        Greset   => reset,
        i_d      => S5 or S3 or (S1 and (not Sign) and (not Notless24) and Zero) or (S2 and Notless24),
        i_enable => '1',
        i_clock  => clk,
        o_q      => S7,
        o_qBar   => open
    );

    dff8: enARDFF_2 port map (
        Greset   => reset,
        i_d      => S7 and CountMat,
        i_enable => '1',
        i_clock  => clk,
        o_q      => S8,
        o_qBar   => open
    );

    dff9: enARDFF_2 port map (
        Greset   => reset,
        i_d      => S8 or (S7 and (not CountMat)),
        i_enable => '1',
        i_clock  => clk,
        o_q      => S9,
        o_qBar   => open
    );

    -- Output State Assignment
    state_out <= S9 & S8 & S7 & S6;

end Structural;
```

A - 6 Adder Top Level Entity VHDL Code



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

entity FP_Adder_Top is
    Port (
        clk       : in  STD_LOGIC;
        reset     : in  STD_LOGIC;
        Sign      : in  STD_LOGIC;
        Notless24 : in  STD_LOGIC;
        Zero      : in  STD_LOGIC;
        CountMat  : in  STD_LOGIC;
        state_out : out STD_LOGIC_VECTOR(3 downto 0);
        result    : out STD_LOGIC_VECTOR(31 downto 0)
    );
end FP_Adder_Top;

architecture Structural of FP_Adder_Top is

    component FP_Adder_Control is
        Port (
            clk       : in  STD_LOGIC;
            reset     : in  STD_LOGIC;
            Sign      : in  STD_LOGIC;
            Notless24 : in  STD_LOGIC;
            Zero      : in  STD_LOGIC;
            CountMat  : in  STD_LOGIC;
            state_out : out STD_LOGIC_VECTOR(3 downto 0)
        );
    end component;

    component FP_Adder_Datapath is
        Port (
            clk       : in  STD_LOGIC;
            reset     : in  STD_LOGIC;
            state     : in  STD_LOGIC_VECTOR(3 downto 0);
            operandA  : in  STD_LOGIC_VECTOR(31 downto 0);
            operandB  : in  STD_LOGIC_VECTOR(31 downto 0);
            load      : in  STD_LOGIC;
            shift     : in  STD_LOGIC;
            add       : in  STD_LOGIC;
            result    : out STD_LOGIC_VECTOR(31 downto 0)
        );
    end component;
```



```vhdl
    signal state_signal : STD_LOGIC_VECTOR(3 downto 0);
    signal operandA, operandB : STD_LOGIC_VECTOR(31 downto 0);
    signal load_signal, shift_signal, add_signal : STD_LOGIC;

begin

    -- Control Unit Instantiation
    control_unit: FP_Adder_Control port map (
        clk       => clk,
        reset     => reset,
        Sign      => Sign,
        Notless24 => Notless24,
        Zero      => Zero,
        CountMat  => CountMat,
        state_out => state_signal
    );

    -- Datapath Instantiation
    datapath_unit: FP_Adder_Datapath port map (
        clk      => clk,
        reset    => reset,
        state    => state_signal,
        operandA => operandA,
        operandB => operandB,
        load     => load_signal,
        shift    => shift_signal,
        add      => add_signal,
        result   => result
    );

end Structural;
```