I.J. Information Engineering and Electronic Business, 2014, 1, 1-14
Published Online February 2014 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijieeb.2014.01.01

Design of FPGA based 32-bit Floating Point Arithmetic Unit and verification of its VHDL code using MATLAB

Naresh Grover, M.K.Soni
Faculty of Engineering and Technology, Manav Rachna International University, Faridabad, India
grovernr@rediffmail.com, dr_mksoni@hotmail.com

Abstract — Most of the algorithms implemented in FPGAs used to be fixed-point. Floating-point operations are useful for computations involving large dynamic range, but they require significantly more resources than integer operations. With the current trends in system requirements and available FPGAs, floating-point implementations are becoming more common and designers are increasingly taking advantage of FPGAs as a platform for floating-point implementations. The rapid advance in Field-Programmable Gate Array (FPGA) technology makes such devices increasingly attractive for implementing floating-point arithmetic. Compared to Application Specific Integrated Circuits, FPGAs offer reduced development time and costs. Moreover, their flexibility enables field upgrade and adaptation of hardware to run-time conditions. A 32 bit floating point arithmetic unit with IEEE 754 Standard has been designed using VHDL code and all operations of addition, subtraction, multiplication and division are tested on Xilinx. Thereafter, Simulink model in MAT lab has been created for verification of VHDL code of that Floating Point Arithmetic Unit in Modelsim.

Index Terms — Floating Point, Arithmetic Unit, VHDL, Modelsim, Simulink.

1. Introduction

The floating point operations have found intensive applications in the various fields for the requirements for high precious operation due to its great dynamic range, high precision and easy operation rules. High attention has been paid on the design and research of the floating point processing units. With the increasing requirements for the floating point operations for the high-speed data signal processing and the scientific operation, the requirements for the high-speed hardware floating point arithmetic units have become more and more exigent. The implementation of the floating point arithmetic has been very easy and convenient in the floating point high level languages, but the implementation of the arithmetic by hardware has been very difficult. With the development of the very large scale integration (VLSI) technology, a kind of devices like Field Programmable Gate Arrays (FPGAs) have become the best options for implementing floating hardware arithmetic units because of their high integration density, low price, high performance and flexible applications requirements for high precious operation.

Floating-point implementation on FPGAs has been the interest of many researchers. The use of custom floating-point formats in FPGAs has been investigated in a long series of work [1, 2, 3, 4, 5]. In most of the cases, these formats are shown to be adequate for some applications that require significantly less area to implement than IEEE formats [6] and to run significantly faster than IEEE formats. Moreover, these efforts demonstrate that such customized formats enable significant speedups for certain chosen applications. The earliest work on IEEE floating-point [7] focused on single precision although found to be feasible but it was extremely slow. Eventually, it was demonstrated [8] that while FPGAs were uncompetitive with CPUs in terms of peak FLOPs, they could provide competitive sustained floating-point performance. Since then, a variety of work [2, 5, 9, 10] has demonstrated the growing feasibility of IEEE compliant, single precision floating point arithmetic and other floating-point formats of approximately same complexity. In [2, 5], the details of the floating-point format are varied to optimize performance. The specific issues of implementing floating-point division in FPGAs have been studied [10]. Early implementations either involved multiple FPGAs for implementing IEEE 754 single precision floating-point arithmetic, or they adopted custom data formats to enable a single-FPGA solution. To overcome device size restriction, subsequent single-FPGA implementations of IEEE 754 standard employed serial arithmetic or avoided features, such as supporting gradual underflow, which are expensive to implement.

In this paper, a high-speed IEEE754-compliant 32-bit floating point arithmetic unit designed using VHDL code has been presented and all operations of addition, subtraction, multiplication and division got tested on Xilinx and verified successfully. Thereafter, the new feature of creating Simulink model using MAT lab for verification of VHDL code of that 32-bit Floating Point Arithmetic Unit in Modelsim has been explained. The simulation results of addition, subtraction, multiplication and division in Modelsim wave window

have been demonstrated.

The rest of the paper is organized as follows. Section 2 presents the general floating point architecture. Section 3 explains the algorithms used to write VHDL codes for implementing 32 bit floating point arithmetic operations: addition/subtraction, multiplication and division. The Section 4 of the paper details the VHDL code and behaviour model for all above stated arithmetic operation. The section 5 explains the design steps along with experimental method to create Simulink model in MAT lab for verification of VHDL code in Modelsim and the results are shown and discussed in its section 6 while section 7 concludes the paper with further scope of work.

2. Floating Point Architecture

Floating point numbers are one possible way of representing real numbers in binary format; the IEEE 754 [11] standard presents two different floating point formats, Binary interchange format and Decimal interchange format. This paper focuses only on single precision normalized binary interchange format. Figure 1 shows the IEEE 754 single precision binary format representation; it consists of a one bit sign (S), an eight bit exponent (E), and a twenty three bit fraction (M) or Mantissa.

32 bit Single Precision Floating Point Numbers IEEE standard are stored as:

S EEEEEEEE MMMMMMMMMMMMMMMMMMMMMMM

S: Sign – 1 bit
E: Exponent – 8 bits
M: Mantissa – 23 bits Fraction

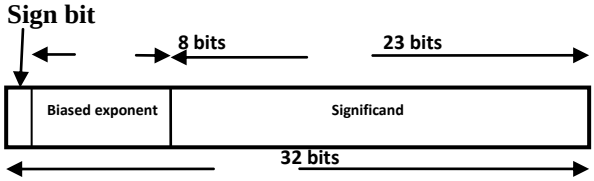

Figure.1: IEEE 754 single precision binary format representation

The value of number V:

If E=255 and F is nonzero, then V= Nan ("Not a Number")

If E=255 and F is zero and S is 1, then V= - Infinity

If E=255 and F is zero and S is 0, then V= Infinity

If 0<E<255 then V= (-1) **S * 2 ** (E-127) * (1.F) (exponent range = -127 to +128)

If E=0 and F is nonzero, then V= (-1) **S * 2 ** (-126) * (0.F) ("un-normalized" values")

If E=0 and F is zero and S is 1, then V= - 0
If E=0 and M is zero and S is 0, then V = 0

An extra bit is added to the mantissa to form what is called the significand. If the exponent is greater than 0 and smaller than 255, and there is 1 in the MSB of the significand then the number is said to be a normalized number; in this case the real number is represented by (1)

$$V = (-1^{s}) * 2^{(E - Bias)} * (1.M) \quad (1)$$

Where $M = m22\ 2^{-1} + m21\ 2^{-2} + m20\ 2^{-3} + \ldots + m1\ 2^{-22} + m0\ 2^{-23}$; Bias = 127.

3. Algorithms for Floating Point Arithmetic Unit

The algorithms using flow charts for floating point addition/subtraction, multiplication and division have been described in this section, that become the base for writing VHDL codes for implementation of 32-bit floating point arithmetic unit.

3.1 Floating Point Addition / Subtraction

The algorithm for floating point addition is explained through flow chart in Figure 2. While adding the two floating point numbers, two cases may arise. Case I: when both the numbers are of same sign i.e. when both the numbers are either +ve or –ve. In this case MSB of both the numbers are either 1 or 0. Case II: when both the numbers are of different sign i.e. when one number is +ve and other number is –ve. In this case the MSB of one number is 1 and other is 0.

Case I: - When both numbers are of same sign

Step 1:- Enter two numbers N1 and N2. E1, S1 and E1, S2 represent exponent and significand of N1 and N2 respectively.

Step 2:- Is E1 or E2 ='0'. If yes; set hidden bit of N1 or N2 is zero. If not; then check if E2 > E1, if yes swap N1 and N2 and if E1 > E2; contents of N1 and N2 need not to be swapped.

Step 3:- Calculate difference in exponents d=E1-E2. If d = '0' then there is no need of shifting the significand. If d is more than '0' say 'y' then shift S2 to the right by an amount 'y' and fill the left most bits by zero. Shifting is done with hidden bit.

Step 4:- Amount of shifting i.e. 'y' is added to exponent of N2 value. New exponent value of E2= (previous E2) + 'y'. Now result is in normalize form because E1 = E2.

Step 5:- Check if N1 and N2 have different sign, if 'no';

Step 6:- Add the significands of 24 bits each including hidden bit S=S1+S2.

Step 7:- Check if there is carry out in significand addition. If yes; then add '1' to the exponent value of either E1 or new E2. After addition, shift the overall result of significand addition to the right by one by making MSB of S as '1' and dropping LSB of significand.

Step 8:- If there is no carry out in step 6, then previous exponent is the real exponent.

Step 9:- Sign of the result i.e. MSB = MSB of either N1 or N2.

Step 10:- Assemble result into 32 bit format excluding 24th bit of significand i.e. hidden bit.

Case II: - When both numbers are of different sign

Step 1, 2, 3 & 4 are same as done in case I.

Step 5:- Check if N1 and N2 have different sign, if 'Yes';

Step 6:- Take 2's complement of S2 and then add it to S1 i.e. S=S1+ (2's complement of S2).

Step 7:- Check if there is carry out in significand addition. If yes; then discard the carry and also shift the result to left until there is '1' in MSB and also count the amount of shifting say 'z'.

Step 8:- Subtract 'z' from exponent value either from E1 or E2. Now the original exponent is E1-'z'. Also append the 'z' amount of zeros at LSB.

Step 9:- If there is no carry out in step 6 then MSB must be '1' and in this case simply replace 'S' by 2's complement.

Step 10:- Sign of the result i.e. MSB = Sign of the larger number either MSB of N1or it can be MSB of N2.

Step 11:- Assemble result into 32 bit format excluding 24th bit of significand i.e. hidden bit.

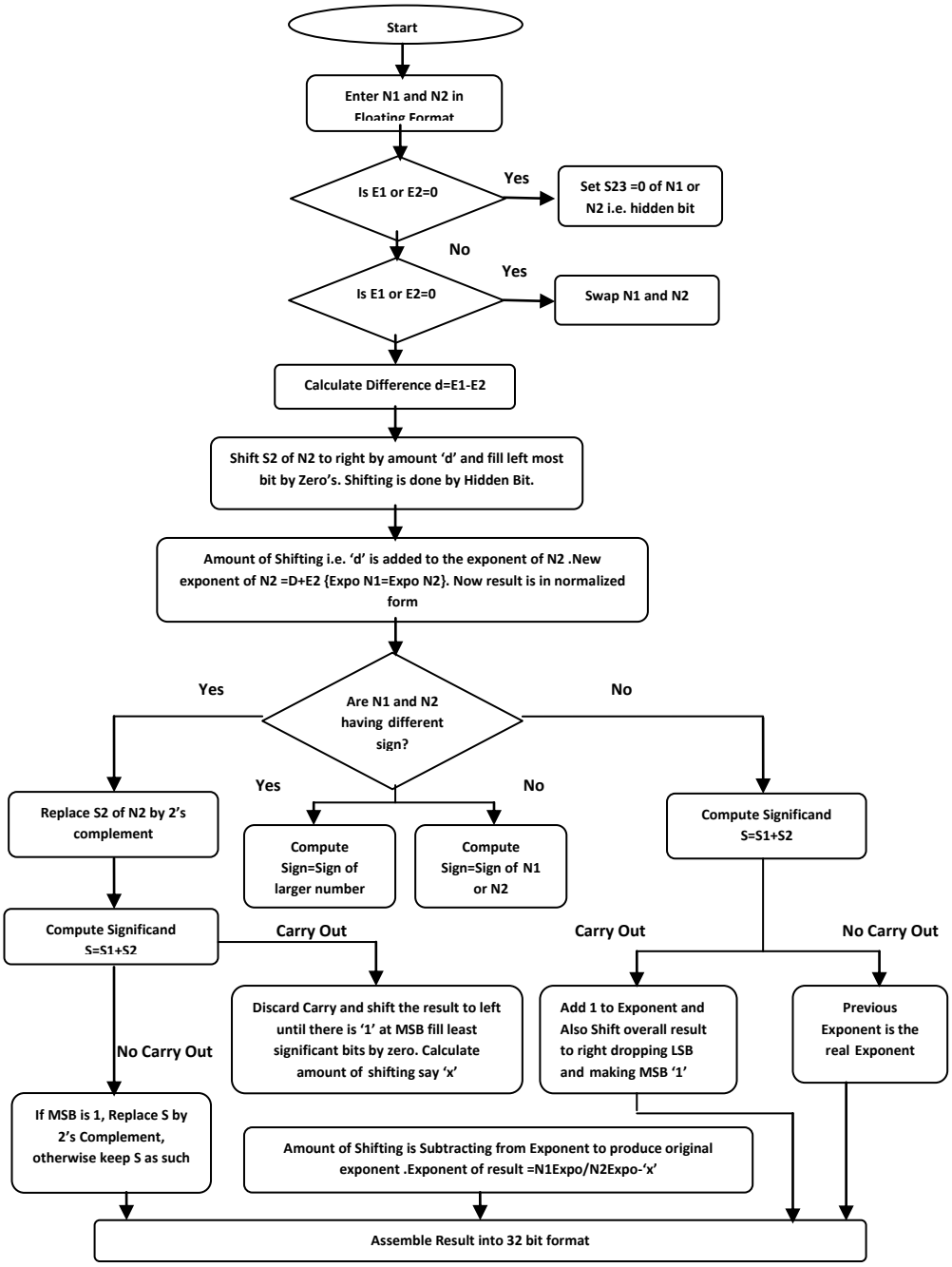

Figure. 2: Flow Chart for floating point Addition/Subtraction

In this algorithm three 8-bit comparators, one 24-bit and two 8-bit adders, two 8-bit subtractors, two shift units and one swap unit are required in the design.

First 8-bit comparator is used to compare the exponent of two numbers. If exponents of two numbers are equal then there is no need of shifting. Second 8-bit comparator compares exponent with zero. If the exponent of any number is zero set the hidden bit of that number zero. Third comparator is required to check whether the exponent of number 2 is greater than number 1. If the exponent of number 2 is greater than number 1 then the numbers are swapped.

One subtractor is required to compute the difference between the 8-bit exponents of two numbers. Second subtractor is used if both the numbers are of different sign than after addition of the significands of two numbers if carry appears. This carry is subtracted from the exponent using 8-bit subtractor.

One 24-bit adder is required to add the 24-bit significands of two numbers. One 8-bit adder is required if both the numbers are of same sign than after addition of the significands of two numbers if carry appears. This carry is added to the exponent using 8-bit adder. Second 8-bit adder is used to add the amount of shifting to the exponent of smaller number.

One swap unit is required to swap the numbers if N2 is greater than N1. Swapping is normally done by taking the third variable. Two shift units are required one is shift left and second is shift right.

3.2 Floating Point Multiplication

The algorithm for floating point multiplication is explained through flow chart in Figure 3. Let N1 and N2 are normalized operands represented by S1, M1, E1 and S2, M2, E2 as their respective sign bit, mantissa (significand) and exponent. Basically following four steps are used for floating point multiplication.

1. Multiply signifcands, add exponents, and determine sign

M=M1*M2

E=E1+E2-Bias

S=S1XORS2

2. Normalize Mantissa M (Shift left or right by 1) and update exponent E

3. Rounding the result to fit in the available bits

4. Determine exception flags and special values for overflow and underflow.

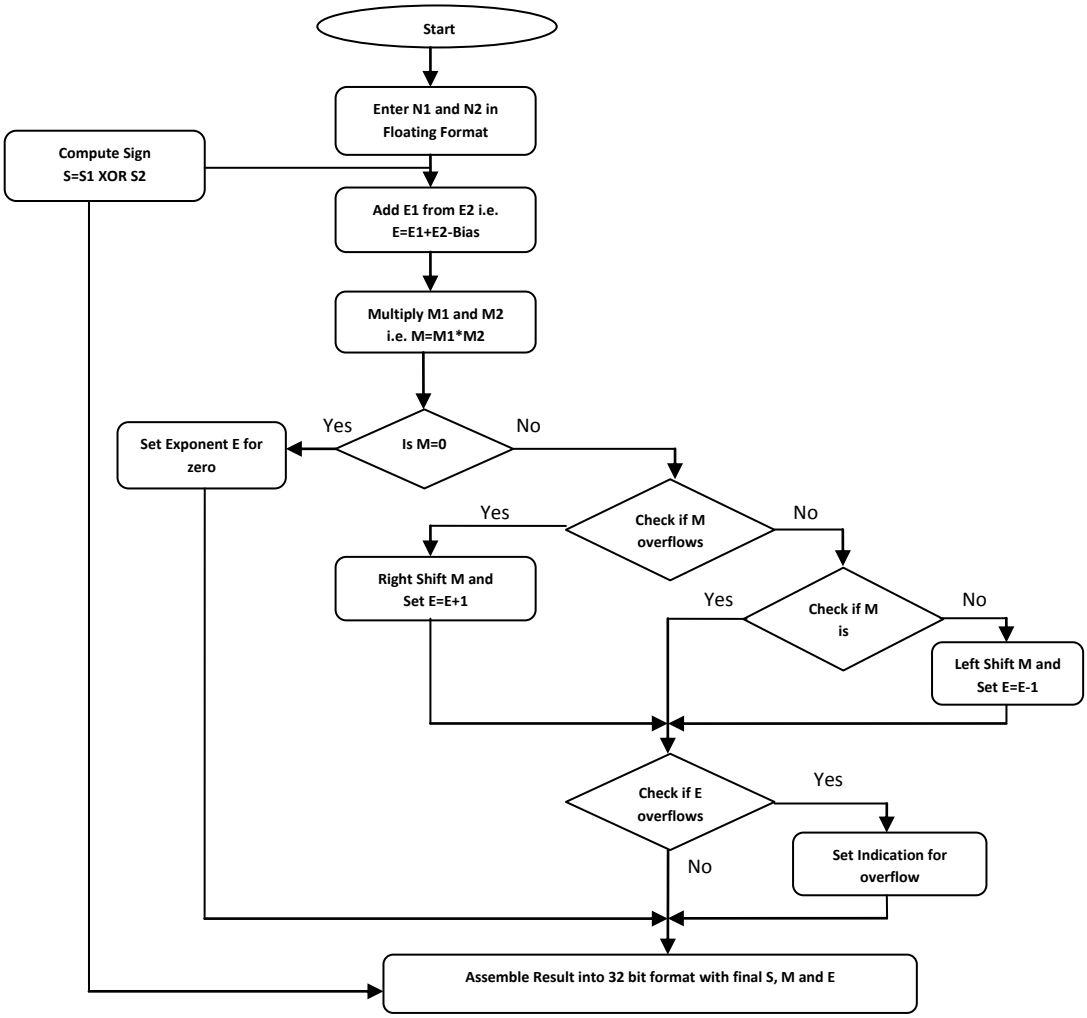

Figure. 3: Flow Chart for floating point Multiplication

Sign Bit Calculation: The result of multiplication is a negative sign if one of the multiplied numbers is of a negative value and that can be obtained by XORing the sign of two inputs.

Exponent Addition is done through unsigned adder for adding the exponent of the first input to the exponent of the second input and after that subtract the Bias (127) from the addition result (i.e. E1+E2 - Bias). The result of this stage can be called as intermediate exponent.

Significand Multiplication is done for multiplying the unsigned significand and placing the decimal point in the multiplication product. The result of significand multiplication can be called as intermediate product (IP). The unsigned significand multiplication is done on 24 bit.

The result of the significand multiplication (intermediate product) must be normalized to have a leading '1' just to the left of the decimal point (i.e. in the bit 46 in the intermediate product). Since the inputs are normalized numbers then the intermediate product has the leading one at bit 46 or 47. If the leading one is at bit 46 (i.e. to the left of the decimal point) then the intermediate product is already a normalized number and no shift is needed. If the leading one is at bit 47 then the intermediate product is shifted to the right and the exponent is incremented by 1.

Overflow/underflow means that the result's exponent is too large/small to be represented in the exponent field. The exponent of the result must be 8 bits in size, and must be between 1 and 254 otherwise the value is not a normalized one .An overflow may occur while adding the two exponents or during normalization. Overflow due to exponent addition can be compensated during subtraction of the bias; resulting in a normal output value (normal operation). An underflow may occur while subtracting the bias to form the intermediate exponent. If the intermediate exponent < 0 then it is an underflow that can never be compensated; if the intermediate exponent = 0 then it is an underflow that may be compensated during normalization by adding 1 to it .When an overflow occurs an overflow flag signal goes high and the result turns to ±Infinity (sign determined according to the sign of the floating point multiplier inputs). When an underflow occurs an underflow flag signal goes high and the result turns to ±Zero (sign determined according to the sign of the floating point multiplier inputs).

3.3 Floating Point Division

The algorithm for floating point multiplication is explained through flow chart in Figure 4. Let N1 and N2 are normalized operands represented by S1, M1, E1 and S2, M2, E2 as their respective sign bit, mantissa (significand) and exponent. If let us say we consider x=N1 and d=N2 and the final result q has been taken as "x/d". Again the following four steps are used for floating point division.

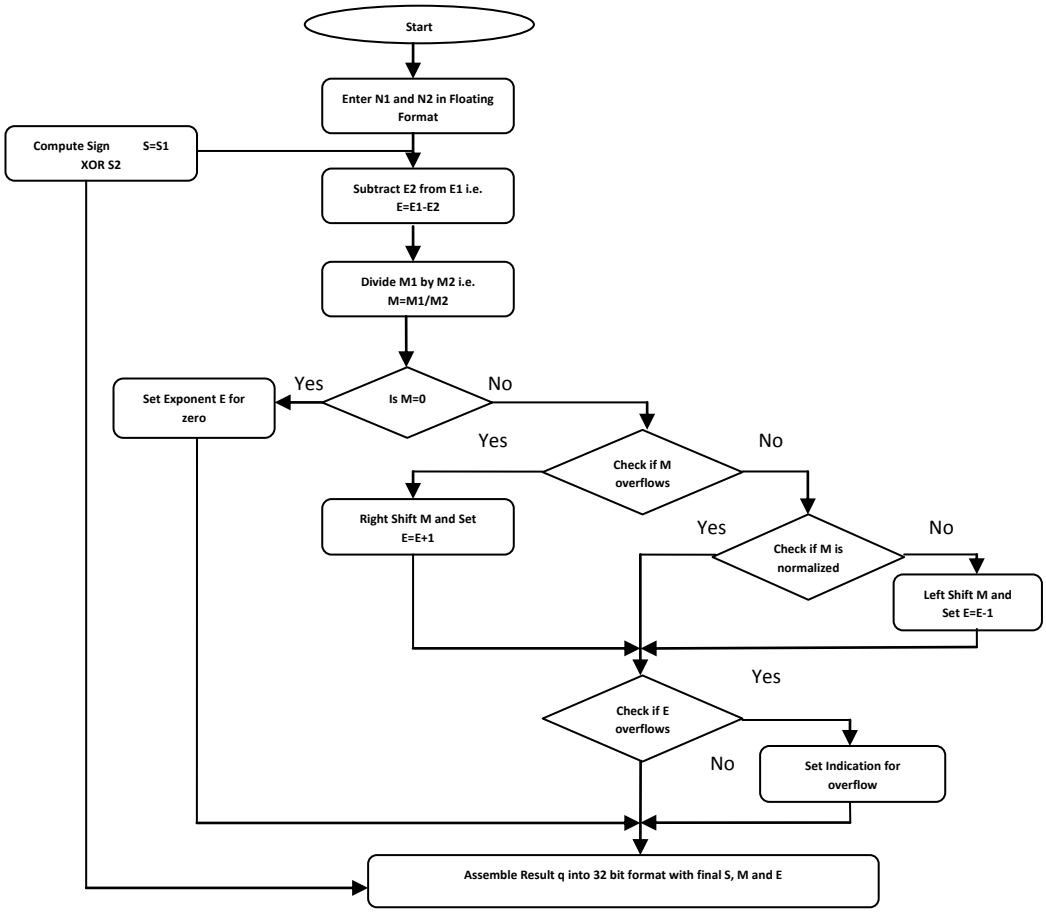

Figure. 4: Flow Chart for floating point Division (q = x/d; N1=x and N2=d)

1. Divide signifcands, subtract exponents, and determine sign

M=M1/M2
E=E1-E2
S=S1XORS2

2. Normalize Mantissa M (Shift left or right by 1) and update exponent E

3. Rounding the result to fit in the available bits

4. Determine exception flags and special values

The sign bit calculation, mantissa division, exponent subtraction (no need of bias subtraction here), rounding the result to fit in the available bits and normalization is done in the similar way as has been described for multiplication.

4. VHDL Code

This section illustrates the main steps of VHDL code that has been used to implement the 32-bit floating point arithmetic functions: addition/subtraction, multiplication and division. It includes the arithmetic structure followed by behavior model for different arithmetic functions for 32-bit floating point format following IEEE 754 standards.

ARITHMETIC UNIT STRUCTURE

```
entity fp_alu is
port(in1,in2:in std_logic_vector(31 downto 0);
clk:in std_logic;
sel:in std_logic_vector(1 downto 0);
output1:out std_logic_vector(31 downto 0));
end fp_alu;
architecture fp_alu_struct of fp_alu is
component divider is
port(
    clk     : in std_logic;
    res     : in std_logic;
    GO      : in std_logic;
    x       : in std_logic_vector(31 downto 0);
    y       : in std_logic_vector(31 downto 0);
    z       : out std_logic_vector(31 downto 0);
    done    : out std_logic;
    overflow : out std_logic
  );
          end component;
component fpa_seq is
          port(
          n1,n2:in std_logic_vector(32 downto 0);
          clk:in std_logic;
          sum:out std_logic_vector(32 downto 0)
          );
end component;
component fpm is
port(in1,in2:in std_logic_vector(31 downto 0);
out1:out std_logic_vector(31 downto 0)
);
end component;
signal out_fpa: std_logic_vector(32 downto 0);
signal out_fpm,out_div: std_logic_vector(31 downto 0);
signal in1_fpa,in2_fpa: std_logic_vector(32 downto 0);
begin
in1_fpa<=in1&'0';
in2_fpa<=in2&'0';
fpa1:fpa_seq port map(in1_fpa,in2_fpa,clk,out_fpa);
fpm1:fpm port map(in1,in2,out_fpm);
fpd1:divider port map(clk,'0','1',in1,in2,out_div);
process(sel,clk)
begin
if(sel="01")then
output1<=out_fpa(32 downto 1);
elsif(sel="10")then
output1<=out_fpm;
elsif(sel="11")then
output1<=out_div;
end if;
end process;
end fp_alu_struct;
```

FPA BEHAVIOUR

```
entity fpa_seq is
port(n1,n2:in std_logic_vector(32 downto 0);
clk:in std_logic;
sum:out std_logic_vector(32 downto 0));
end fpa_seq;
architecture Behavioral of fpa_seq is
```

```
--signal f1,f2:std_logic_vector(23 downto
0):="00000000000000000000000";
signal sub_e:std_logic_vector(7 downto
0):="00000000";
--signal addi:std_logic_vector(34 downto 0);
signal c_temp:std_logic:='0';--_vector(34 downto 0);
signal shift_count1:integer:=0;
signal num2_temp2: std_logic_vector(32 downto
0):="00000000000000000000000000000000";
signal s33:std_logic_vector(23 downto
0):="00000000000000000000000";
signal s2_temp :std_logic_vector(23 downto
0):="00000000000000000000000";
signal diff:std_logic_vector(7 downto 0):="00000000";
----------sub calling----------------------------------------------
--------------------
sub(e1,e2,d);
if(d>="00011100")then
sum<=num1;
elsif(d<"00011100")then
shift_count:=conv_integer(d);
shift_count1<=shift_count;
num2_temp2<=num2;
--s2_temp<=s2;
--------------shifter calling-------------------------------------
--------------------
shift(s2,shift_count,s3);
--s33<=s3;
------------sign bit checking------
 if (num1(32)/=num2(32))then
 s3:=(not(s3)+'1');------2's complement
 adder23(s1,s3,s4,c_out);
 if(c_out='1')then
  shift_left(s4,d_shl,ss4);
  sub(e1,d_shl,ee4);
  sum<=n1(32)& ee4 & ss4;
 else
   if(s4(23)='1')then
    s4:=(not(s4)+'1');------2's complement
            sum<=n1(32)& e1 & ss4;
 end if;
 end if;
 else
 s3:=s3;
 -- end if;
 ----------------------same sign start
---------------adder 8 calling---------------
adder8(e2,d,e3);
sub_e<=e3;
num1_temp:=n1(32)& e1 & s1;
num2_temp:=n2(32)& e3 & s3;
---------------adder 23 calling---------------
adder23(s1,s3,s4,c_out);
--s2_temp<=s4;
c_temp<=c_out;
if(c_out='1')then
--shift1(s4,s_1,s5);
--s2_temp<=s5;
s33<=s4;
s5:='1' & s4(23 downto 1);
s2_temp<=s5;
adder8(e3,"00000001",e4);
e3:=e4;
--sub_e<=e4;
sum<=n1(32)& e3 & s5;
else
sum<=n1(32)& e3 & s4;
end if;
end if;
end if;
end if;----same sign end
end if;
------final result assembling----------
--sum_temp<=n1(32)& e1 & s4;
--sum<=n1(32)& e3 & s4;
end process;
end Behavioral;
```

FPM BEHAVIOUR

```
entity fpm is
port(in1,in2:in std_logic_vector(31 downto 0);
```

```
out1:out std_logic_vector(31 downto 0));
end fpm;
architecture Behavioral of fpm is
procedure adder( a,b:in std_logic_vector(7 downto 0);
        sout : out  STD_LOGIC_VECTOR (8 downto 0))is
variable g,p:std_logic_vector(7 downto 0);
variable c:std_logic_vector(8 downto 0);
variable sout1 :STD_LOGIC_VECTOR (7 downto 0);
begin
c(0):='0';
for i in 0 to 7 loop
g(i):= a(i) and b(i);
p(i):= a(i) xor b(i);
end loop;
for i in 0 to 7 loop
c(i+1):=(g(i) or (c(i) and p(i)));
end loop;
for i in 0 to 7 loop
sout1(i):=c(i) xor a(i) xor b(i);
end loop;
sout:=c(8) & sout1;
end adder;
-------------------------------------------multiplier-------------------------
procedure multiplier ( a,b : in  STD_LOGIC_VECTOR (23 downto 0);
        y : out  STD_LOGIC_VECTOR (47 downto 0))is
variable temp,prod:std_logic_vector(47 downto 0);
begin
temp:="000000000000000000000000"&a;
prod:="000000000000000000000000000000000000000000000000";
for i in 0 to 23 loop
if b(i)='1' then
prod:=prod+temp;
end if;
temp:=temp(46 downto 0)&'0';
end loop;
y:=prod;
end multiplier;
-------------------------end multipier-------------------------------------
begin
process(in1,in2)
variable sign_f,sign_in1,sign_in2: std_logic:='0';
variable e1,e2: std_logic_vector(7 downto 0):="00000000";
variable add_expo:std_logic_vector(8 downto 0):="000000000";
variable m1,m2: std_logic_vector(23 downto 0):="000000000000000000000000";
variable mantisa_round: std_logic_vector(22 downto 0):="00000000000000000000000";
variable prod:std_logic_vector(47 downto 0):="000000000000000000000000000000000000000000000000";
variable mul_mantisa :std_logic_vector(47 downto 0):="000000000000000000000000000000000000000000000000";
variable bias:std_logic_vector(8 downto 0):="001111111";
variable bias_sub:std_logic_vector(7 downto 0):="00000000";
variable inc_bias:std_logic_vector(8 downto 0):="000000000";
variable bias_round:std_logic_vector(8 downto 0):="000000000";
begin
sign calculation
sign_in1:=in1(31);
sign_in2:=in2(31);
sign_f:=sign_in1 xor sign_in2;
```

FPD BEHAVIOUR

```
entity divider is
port(
    clk     : in std_logic;
    res     : in std_logic;
    GO      : in std_logic;
    x       : in std_logic_vector(31 downto 0);
    y       : in std_logic_vector(31 downto 0);
    z       : out std_logic_vector(31 downto 0);
    done    : out std_logic;
```



```
    overflow : out std_logic
  );
end divider;
architecture design of divider is
 signal x_reg      : std_logic_vector(31 downto 0);
 signal y_reg      : std_logic_vector(31 downto 0);
 signal x_mantissa : std_logic_vector(23 downto 0);
 signal y_mantissa : std_logic_vector(23 downto 0);
 signal z_mantissa : std_logic_vector(23 downto 0);
 signal x_exponent : std_logic_vector(7 downto 0);
 signal y_exponent : std_logic_vector(7 downto 0);
 signal z_exponent : std_logic_vector(7 downto 0);
 signal x_sign     : std_logic;
 signal y_sign     : std_logic;
 signal z_sign     : std_logic;
 signal sign       : std_logic;
 signal SC         : integer range 0 to 26;
 signal exp        : std_logic_vector(9 downto 0);
 signal EA         : std_logic_vector(24 downto 0);
 signal B          : std_logic_vector(23 downto 0);
 signal Q          : std_logic_vector(24 downto 0);
 type states is (reset, idle, s0, s1, s2, s3, s4);
 signal state : states;
begin
  x_mantissa <= '1' & x_reg(22 downto 0);
  x_exponent <= x_reg(30 downto 23);
 x_sign <= x_reg(31);
 y_mantissa <= '1' & y_reg(22 downto 0);
 y_exponent <= y_reg(30 downto 23);
 y_sign <= y_reg(31);
  process(clk)
   begin
    if clk'event and clk = '1' then
     if res = '1' then
       state <= reset;
       exp <= (others => '0');
       sign <= '0';
       x_reg <= (others => '0');
       y_reg <= (others => '0');
       z_sign <= '0';
       z_mantissa <= (others => '0');
       z_exponent <= (others => '0');
       EA <= (others => '0');
       Q <= (others => '0');
       B <= (others => '0');
       overflow <= '0';
       done <= '0';
     else
      case state is
      when reset =>  state <= idle;
       when idle =>
                if GO = '1' then
                 state <= s0;
                 x_reg <= x;
                 y_reg <= y;
                end if;
       when s0 =>     state <= s1;
                 overflow <= '0';
                 SC <= 25;
                 done <= '0';
                 sign <= x_sign xor y_sign;
                 EA <= '0' & x_mantissa;
                 B <= y_mantissa;
                 Q <= (others => '0');
                 exp <= ("00" & x_exponent) + not ("00"
& y_exponent) + 1 + "0001111111";
          when s1 => if (y_mantissa = x"800000" and
y_exponent = x"00") then
                 overflow <= '1';
                 z_sign <= sign;
                 z_mantissa <= (others => '0');
                 z_exponent <= (others => '1');
                 done <= '1';
                 state <= idle;
elsif exp(9) = '1' or exp(7 downto 0) = x"00" or
(x_exponent = x"00" and x_mantissa =   x"00") or
(y_exponent = x"FF" and y_mantissa = x"00")  then
                 z_sign <= sign;
                 z_mantissa <= (others => '0');
                 z_exponent <= (others => '0');
                 done <= '1';
```

```
        state <= idle;
      else
       EA <= EA  + ('0' & not B) + 1;
       state <= s2;
      end if;
when s2 =>
       if EA(24) = '1' then
        Q(0) <= '1';
       else
        Q(0) <= '0';
        EA <= EA + B;
       end if;
        SC <= SC - 1;
        state <= s3;
when s3 => if SC = 0 then
       if Q(24) = '0' then
        Q <= Q (23 downto 0) & '0';
        exp <= exp - 1;
       end if;
        state <= s4;
      else
       EA <= EA(23 downto 0) & Q(24);
        Q <= Q(23 downto 0) & '0';
        state <= s1;
        end if;
when s4 =>  if exp = x"00" then
          z_sign <= sign;
          z_mantissa <= (others => '0');
          z_exponent <= (others => '0');
        elsif exp(9 downto 8) = "01" then
          z_sign <= sign;
          z_mantissa <= (others => '0');
          z_exponent <= (others => '1');
        else
          z_sign <= sign;
          z_mantissa <= Q(24 downto 1);
          z_exponent <= exp(7 downto 0);
        end if;
          done <= '1';
          state <= idle;
    end case;
   end if;
  end if;
 end process;
z <= z_sign & z_exponent & z_mantissa(22 downto 0);
end design;
```

The VHDL code written has been tested and verified on Xilinx ISE 8.1i for all operation. The design utilization summary has been shown in Figure 5.

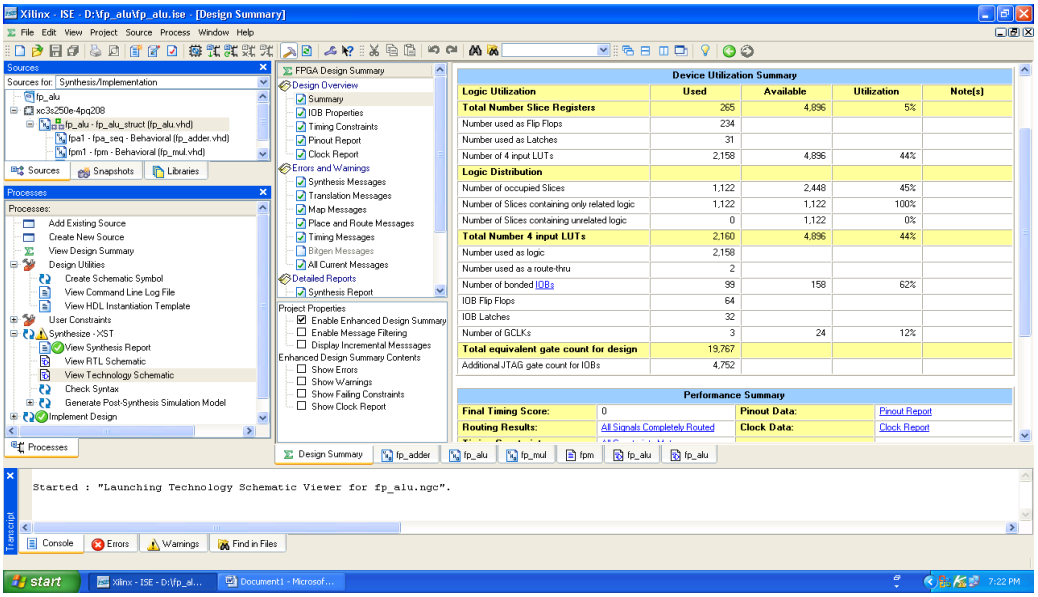

Figure. 5: Design Utilization Summary of Floating Point Arithmetic Unit on FPGA

5. Generation and verification of HDL code using MATLAB

Generation and verification of HDL code using MATLAB requires compatible versions of MATLAB (Simulink) and HDL Simulator 'Modelsim' to be loaded on the same system [13, 14, 15]. The basic design steps to create Simulink model for verification of VHDL code in Modelsim HDL Simulator is shown in the flow chart of Fig. 6.

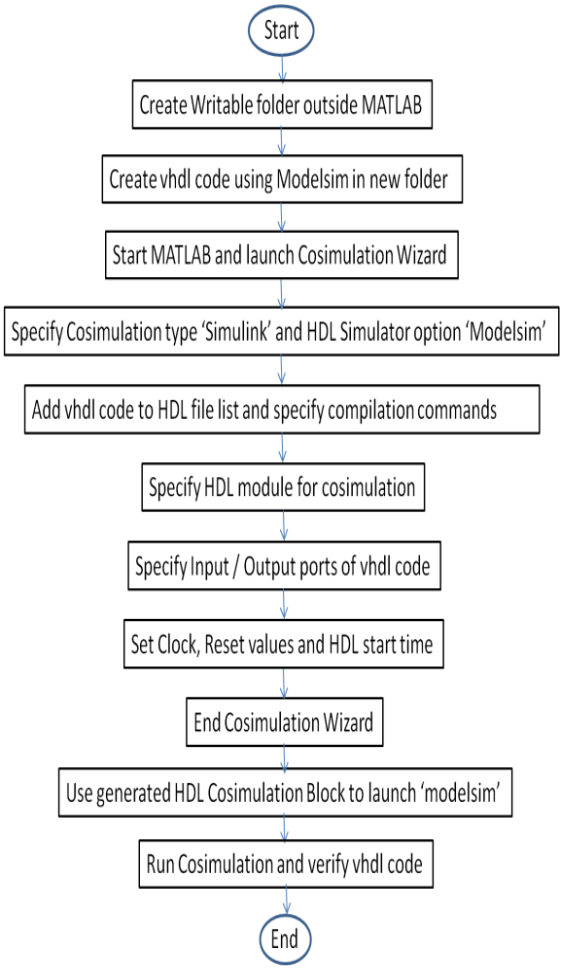

Figure. 6: Design steps to create Simulink model for verification of VHDL code in Modelsim

The Simulink Model to generate and verify Floating Point arithmetic created is shown in Figure 7. Input 1 and Input 2 are the two 32 bit floating point inputs to the model and 'Select' is set to '01' for Adder, '11' for Divider and '10' for Multiplier. It also has a scope to view the output. A sub-system is created to launch the Modelsim Simulator from Simulink as shown in Fig. 8.

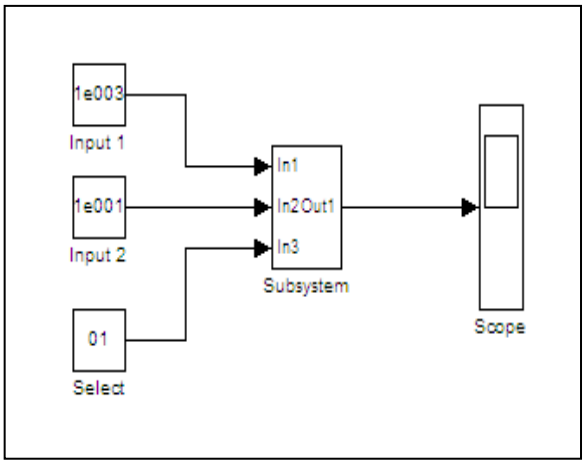

Figure. 7: Simulink model to generate and verify Floating Point arithmetic

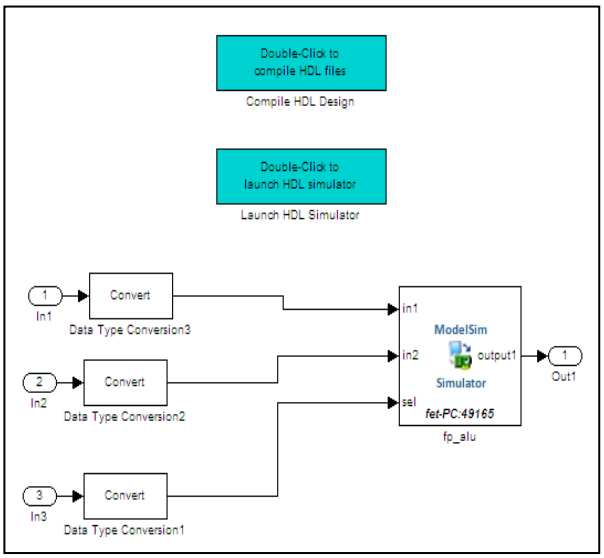

Figure. 8: Simulink sub-system to launch HDL Simulator

6. RESULTS

Double clicking the 'Launch HDL Simulator' in the Simulink model loads the test bench for simulation. The ModelSim Simulator opens a display window for monitoring the simulation as the test bench runs. The wave window in Figure 9 shows the simulation of two exponential inputs and Select set to '01'for 'adder' result as HDL waveform. Figure 10 shows the simulation of two decimal inputs for 'adder'. Figure 11 and 12 show the simulation of two decimal inputs for 'divider'. Figure 13 and 14 show the simulation of two decimal inputs for 'multiplier'.

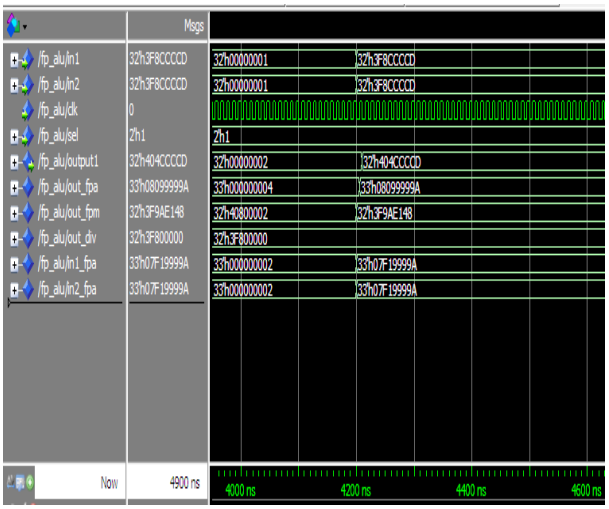

Figure. 9: Simulation result of decimal inputs 1.1 & 1.1 for 'adder' in Modelsim wave window

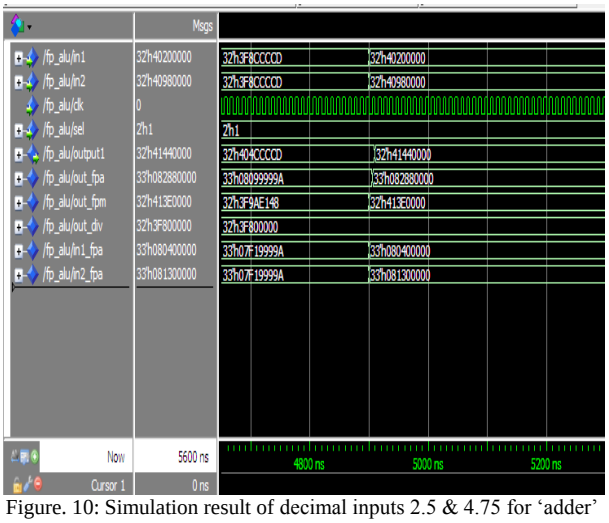

Figure. 10: Simulation result of decimal inputs 2.5 & 4.75 for 'adder' in Modelsim wave window

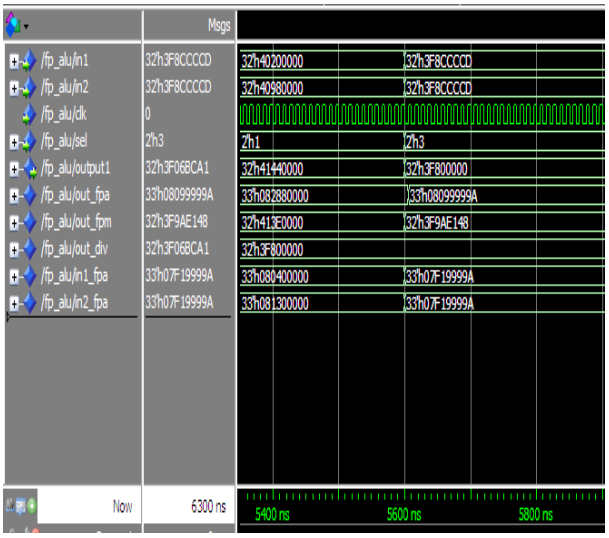

Figure. 11: Simulation result of decimal inputs 1.1 & 1.1 for 'divider' in Modelsim wave window

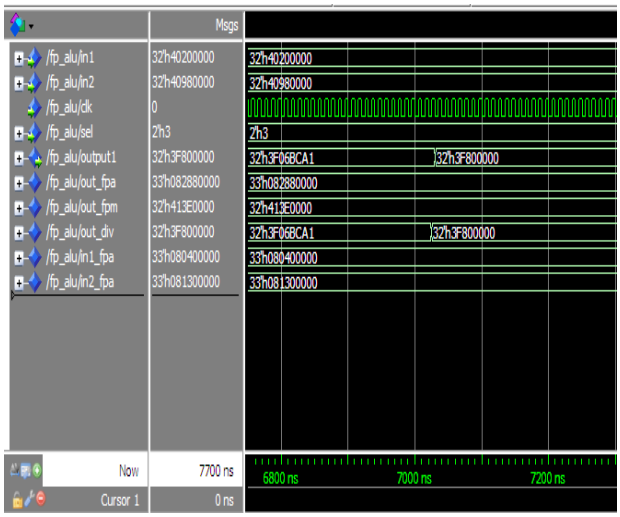

Figure. 12: Simulation result of decimal inputs 2.5 & 4.75 for 'divider' in Modelsim wave window

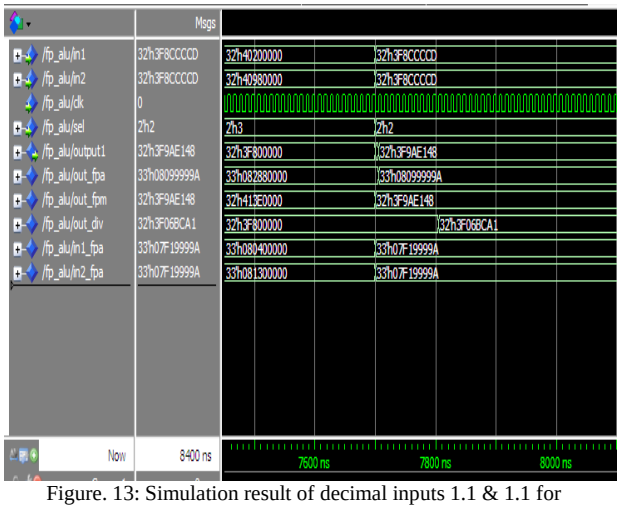

Figure. 13: Simulation result of decimal inputs 1.1 & 1.1 for 'multiplier' in Modelsim wave window

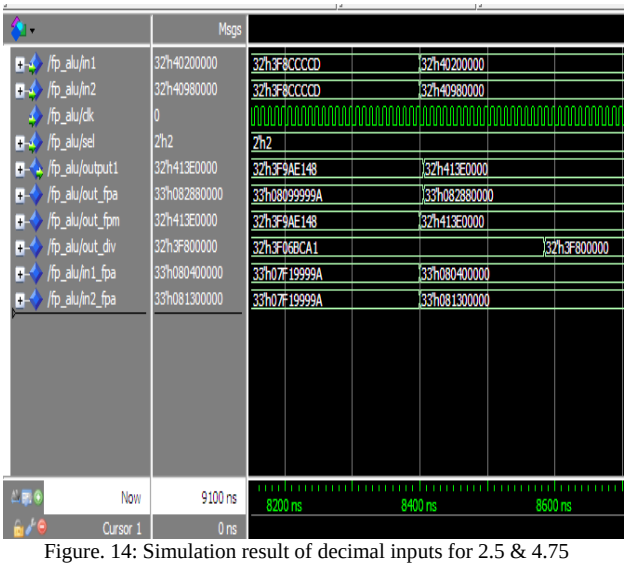

Figure. 14: Simulation result of decimal inputs for 2.5 & 4.75 'multiplier' in Modelsim wave window

Table-I below shows the input output details of the Floating point arithmetic architecture designed and linked using Simulink and Modelsim.

Table I

Wave	Select	Input 1	Input 2	Output
Figure 9	01	32'h3F8CCCCD	32'h3F8CCCCD	32'h404CCCCD
Figure 10	01	32'h40200000	32'h40980000	32'h41440000
Figure 11	11	32'h3F8CCCCD	32'h3F8CCCCD	32'h3F06BCA1
Figure 12	11	32'h40200000	32'h40980000	32'h3F800000
Figure 13	10	32'h3F8CCCCD	32'h3F8CCCCD	32'h3F9AE148
Figure 14	10	32'h40200000	32'h40980000	32'h413E0000

7. Conclusion and Future Scope of Work

The VHDL code written for complete 32-bit floating point arithmetic unit has been implemented and tested on Xilinx. A process described to create Simulink model in MAT lab for verification of VHDL code in Modelsim HDL Simulator has been used on the same VHDL code and results were found in order. Once the Simulink model has been created using MAT lab for VHDL code, the same can be optimized in MAT lab and the VHDL code can be regenerated with the optimized results and tested on Xilinx to see the improvement in the parameters.

References

[1] N. Shirazi, A. Walters, and P. Athanas, "Quantitative analysis of floating point arithmetic on fpga based custom computing machines," in Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, pp. 155–162, 1995.

[2] P. Belanovic and M. Leeser, "A library of parameterized floating-point modules and their use,"in Proceedings of the International Conference on Field Programmable Logic and Applications, 2002.

[3] J. Dido, N. Geraudie, L. Loiseau, O. Payeur, Y. Savaria, and D. Poirier, "A flexible floating-point format for optimizing data-paths and operators in fpga based dsps," in Proceedings of the ACM International Symposium on Field Programmable Gate Arrays, (Monterrey, CA), February 2002.

[4] A. A. Gaar, W. Luk, P. Y. Cheung, N. Shirazi, and J. Hwang, "Automating customisation of floating-point designs," in Proceedings of the International Conference on Field Programmable Logic and Applications, 2002.

[5] J. Liang, R. Tessier, and O. Mencer, "Floating point unit generation and evaluation for fpgas," in Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, (Napa Valley, CA), pp. 185–194, April 2003.

[6] IEEE Standards Board, "IEEE standard for binary floating-point arithmetic," Tech. Rep. ANSI/IEEE Std. 754-1985, The Institute of Electrical and Electronics Engineers, New York, 1985.

[7] B.Fagin and C. Renard, "Field programmable gate arrays and floating point arithmetic," IEEE Transactions on VLSI, vol. 2, no. 3, pp. 365–367, 1994.

[8] W. B. Ligon, S. P. McMillan, G. Monn, F. Stivers, K. Schoonover, and K. D. Underwood, "A re-evaluation of the praticality of floating-point on FPGAs," in Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, (Napa Valley, CA), pp. 206–215, April 1998.

[9] Z. Luo and M. Martonosi, "Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques," IEEE Transactions on Computers, vol. 49, no. 3, pp. 208–218, 2000.

[10] X. Wang and B. E. Nelson, "Tradeoffs of designing floating-point division and square root on virtex fpgas," in Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, (Napa Valley, CA), pp. 195–203, April 2003.

[11] IEEE 754-2008, IEEE Standard for Floating-Point Arithmetic, 2008.

[12] Simulink HDL Coder 1; User's Guide; 2006-2010 by the MathWorks, Inc.

[13] Hikmat N. Abdullah and Hussein A. Hadi "Design and Implementation of FPGA Based Software Defined Radio Using Simulink HDL Coder". Engineering and Technology Journal, Iraq, ISSN 1681-6900 01/2010; Vol.28 (No.23):pp.6750-6767.

[14] B. K. Mishra, S. Save , R. Mane. A frame work for model based designing of analog circuits using Simulink. ICWET '11 Proceedings of the International Conference & Workshop on Emerging Trends in Technology. Pages 1225-1228;

ACM New York, NY, USA ©2011 ISBN: 978-1-4503-0449-8.

[15] Alejandro A. Valenzuela, Hikmat N. Abdullah. A Joint Matlab/FPGA Design of AM Receiver for Teaching Purposes. Electromagnetics and Network Theory and their Microwave Technology Applications , 2011, pp 189-199.

Naresh Grover did his B.Sc (Engg.) in 1984 and M.Tech in Electronics and Communication Engineering in 1998 from REC Kurukshetra (Now NIT Kurukshetra). He has a rich experience of 29 years in academics. He has authored two books on Microprocessors and is a co-author for a book on Electronic Components and Materials. His core area of interest is Microprocessors and Digital System Design. Presently he is doing his research work in the area of FPGA based digital system designs.

Dr. M. K Soni did his B.Sc (Engg.) in 1972 and M.Sc (Engg.) in 1975 from REC Kurukshetra (Now NIT Kurukshetra) and thereafter completed his Ph.D from REC Kurukshetra (in collaboration with IIT Delhi) in 1988. He has a total 40 years of rich experience into Academics. His area of interest is microprocessor based control systems and digital system design. He has more than 100 research papers in the International and National Journals to his credit. Presently he is Executive Director & Dean, Faculty of Engineering and Technology, Manav Rachna International University, Faridabad.