EE – 518

VLSI LAB III

Course Project on

Floating Point Square Root

Submitted To :

Dr Gaurav Trivedi

Ankita Tiwari

Meenali Janveja

Presented By:

Abhishek Verma (214102401)

Chaitanya Roop Tejaswi (214102408)

Pawan Kumar (214102412)

Introduction

With the fast development of sub-micron technology, the density of silicon grows rapidly and the cost of hardware decreases, more and more functionalities are transferred into hardware to realize. In order to meet the ever increasing demand in high performance applications including scientific computations, digital signal processing, computer graphics, multimedia, etc. the performance of square root computation is becoming more and more important. In this design we have used **Newton Raphson method** to implement square root of a number. The iteration of NR method results in a doubled accuracy which leads to faster execution time.

Several methods are commonly employed for the computation of floating-point square root. For example:
1) Digital recurrence, it's simple, easy to implement, but the latency is long, especially for large operand sizes.
2) Functional iteration, such as NR and Goldschmidt iteration algorithms. It requires more hardware cost, and the result is not fully accurate, but it is fast, scalable, easy to pipeline, and has high precision.
3) Very high radix arithmetic, it is fast, but very complicated to implement.
4) Table look up, it is simple, fast, with big hardware cost and bad extensibility.
5) Variable latency, it is fast with more hardware cost.

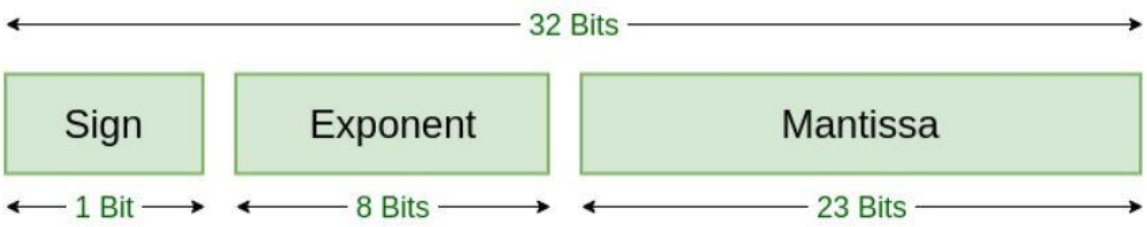
IEEE 754 standard

There are several ways to represent floating point number but IEEE 754 is the most efficient in most cases. IEEE 754 has 3 basic components:

1. **The Sign of Mantissa**
   This is as simple as the name. 0 represents a positive number while 1 represents a negative number.

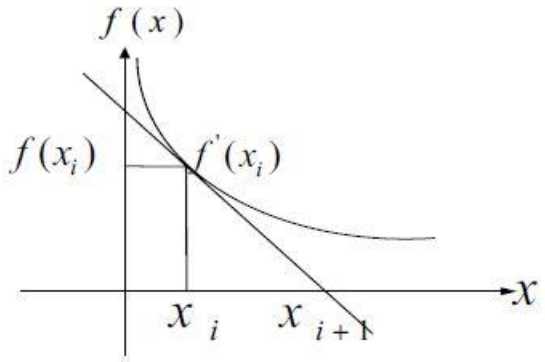
2. **The Biased exponent**
   The exponent field needs to represent both positive and negative exponents. A bias is added to the actual exponent in order to get the stored exponent.

3. **The Normalised Mantissa**
   The mantissa is part of a number in scientific notation or a floating-point number, consisting of its significant digits. Here we have only 2 digits, i.e. O and 1. So a normalised mantissa is one with only one 1 to the left of the decimal.



Newton Raphson Method:

Because of the simplicity and fast convergence, NR iteration method is widely used for complicated computation. The main purpose of this method is to find a zero point of the function.



$$x_{i+1} = x_i - f(x_i) / f'(x_i)$$

This is the NR formula, where x_i is the value at the i^{th} iteration, $f(x_i)$ is the function value at x_i ,and $f'(x_i)$ is the function derivative at x_i.

The iteration formula for square root computation can be derived as follows:

Operation of sign: $S_q = 0$ (if S_b is 1 then square root is not possible)

Operation of exponent: $E_q = E_b$ + bias

Operation of Mantissa: square root of mantissa is calculated using NR method.

Square Root Algorithm:

Newton Raphson Iteration is used to find the square root. It Uses 3 divide, 3 add and 2 Multiply Instances.

Square root Algorithm: $A^{0.5}$

A split into two parts => $M \times 2^E$

$A^{0.5} = (M \times 2^E)^{0.5}$
$A^{0.5} = M^{0.5} \times 2^{(E/2)}$
$X = M^{0.5}$ and $Z = 2^{(E/2)}$

M adjusted to fit the range 0.5-1 by replacing exponent with 8'd126 (actual exponent = 126-127 = -1).

$X = (M \times 1)^{0.5}$

$X = (M \times 2^{(126-127)} / 2^{(126-127)})^{0.5}$

$X = (M \times 2^{(126-127)})^{0.5} / 2^{(-0.5)}$

$X = (M \times 2^{(126-127)})^{0.5} \times (1/ 2^{(-0.5)})$

Let C = 1/ 2^(-0.5) which is already known (constant) and multiplied at the end

Let $Y = (M \times 2^{(126-127)})^{0.5}$ which is computed using Newton Raphson Iterations and Inserted in the equation
thus X becomes: X = Y x C

$2^{(E/2)}$ is basically exponent adjust and based on the value of E (Multiple of 2 or not).The resulting expression is multiplied by $2^{(0.5)}$ if the exponent is not a multiple of 2 and according to that values are re-adjusted at the end.

Initial Seed : x0 = 0.853553414345

Newton Raphson Iterations :
x1 = 0.5*(x0 + X/x0)
x2 = 0.5*(x1 + X/x1)
x3 = 0.5*(x2 + X/x2)

the exponent value of x3 is adjusted and multiplied with square root of 2, if necessary to produce the final result.

Code:

```
module tst(input [31:0]A,
input clk,
input reset,
output [31:0] f_sqrt);

parameter s1=32'h3fa0624e;

wire [7:0] A_exp;
wire [22:0] A_man;
wire A_sign;
wire [31:0] temp1,temp2,temp3,temp4,temp5,temp6,temp7,temp8,temp;
wire [31:0] x0,x1,x2,x3;
wire [31:0] sqrt_1by05; // 1/sqrt(0.5)
wire [31:0] sqrt_2; // sqrt(2)
wire [31:0] sqrt_1by2; // sqrt(0.5)
wire [7:0] Exp_2;
wire remainder;
wire pos;

assign x0 = s1;
assign sqrt_1by05 = 32'h3fb504f3;
assign sqrt_2 = 32'h3fb504f3;
assign sqrt_1by2 = 32'h3f3504f3;
assign A_sign = A[31];
assign A_exp = A[30:23];
assign A_man = A[22:0];

generate
   begin
//First Iteration
divide D1(.A({1'b0,8'd126,A_man}),.B(x0),.C(temp1));
add  A1(.A(temp1),.B(x0),.C(temp2));
assign  x1 = {temp2[31],temp2[30:23]-1,temp2[22:0]};

//Second Iteration
divide D2(.A({1'b0,8'd126,A_man}),.B(x1),.C(temp3));
add  A2(.A(temp3),.B(x1),.C(temp4));
assign  x2 = {temp4[31],temp4[30:23]-1,temp4[22:0]};

//Third Iteration
divide D3(.A({1'b0,8'd126,A_man}),.B(x2),.C(temp5));
add  A3(.A(temp5),.B(x2),.C(temp6));
assign  x3 = {temp6[31],temp6[30:23]-1,temp6[22:0]};
   end
   endgenerate

multiply M1(.A(x3),.B(sqrt_1by05),.C(temp7));

assign pos = (A_exp>=8'd127) ? 1'b1 : 1'b0;
```

```
assign Exp_2 = pos ? (A_exp-8'd127)/2 : (A_exp-8'd127-1)/2 ;
assign remainder = (A_exp-8'd127)%2;
assign temp = {temp7[31],Exp_2 + temp7[30:23],temp7[22:0]};
//assign temp7[30:23] = Exp_2 + temp7[30:23];
multiply M2(.A(temp),.B(sqrt_2),.C(temp8));
assign f_sqrt = remainder ? temp8 : temp;
endmodule
//////////////////////////////////////////////////////////////////////////////

module divide(input [31:0]A,
input [31:0]B,
output [31:0] C);

wire [7:0] exp_Brec;
wire [31:0] B_reciprocal;
wire [31:0] x0,x1,x2,x3;
wire [31:0] temp1,temp2,temp3,temp4,temp5,temp6,temp7,debug;

generate
begin
//Initial value
multiply M1(.A({{1'b0,8'd126,B[22:0]}}),.B(32'h3ff0f0f1),.C(temp1));   //verified 1.88235294819
assign debug = {1'b1,temp1[30:0]};
add A1(.A(32'h4034b4b5),.B({1'b1,temp1[30:0]}),.C(x0));   //2.82352948189

//First Iteration
multiply M2(.A({{1'b0,8'd126,B[22:0]}}),.B(x0),.C(temp2));
add A2(.A(32'h40000000),.B({!temp2[31],temp2[30:0]}),.C(temp3));  //2.0
multiply M3(.A(x0),.B(temp3),.C(x1));

//Second Iteration
multiply M4(.A({1'b0,8'd126,B[22:0]}),.B(x1),.C(temp4));
add A3(.A(32'h40000000),.B({!temp4[31],temp4[30:0]}),.C(temp5));
multiply M5(.A(x1),.B(temp5),.C(x2));

//Third Iteration
multiply M6(.A({1'b0,8'd126,B[22:0]}),.B(x2),.C(temp6));
add A4(.A(32'h40000000),.B({!temp6[31],temp6[30:0]}),.C(temp7));
multiply M7(.A(x2),.B(temp7),.C(x3));
end
endgenerate

//Reciprocal : 1/B
assign exp_Brec = x3[30:23]+8'd126-B[30:23];
assign B_reciprocal = {B[31],exp_Brec,x3[22:0]};

//Multiplication A*(1/B)
multiply M8(.A(A),.B(B_reciprocal),.C(C));
endmodule
//////////////////////////////////////////////////////////////////////////////
```

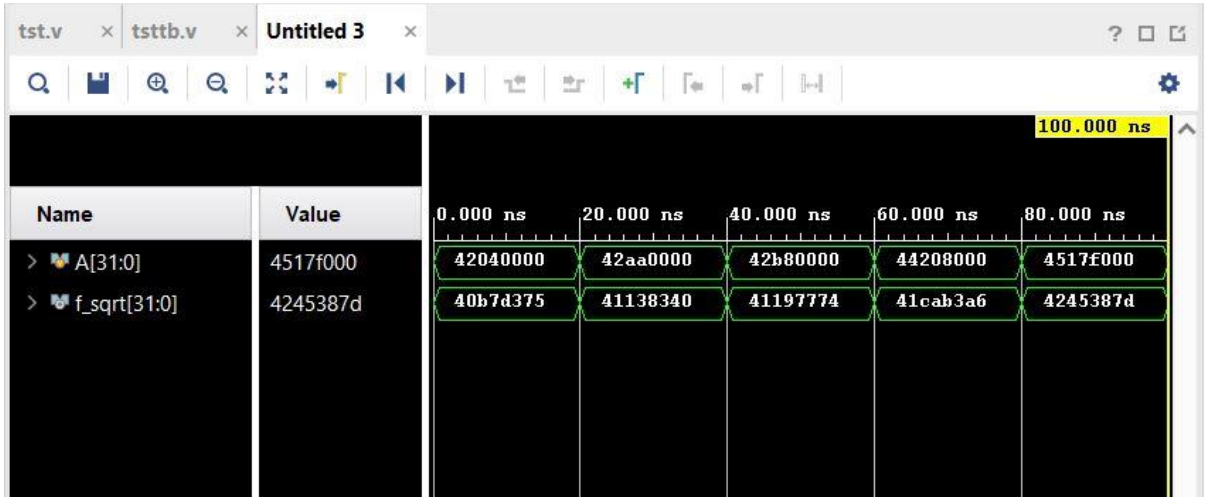
```
module multiply (input [31:0]A,
input [31:0]B,
output [31:0] C);

reg [23:0] A_man,B_man;
reg [22:0] C_man;
reg [47:0] Temp_man;
reg [7:0] A_exp,B_exp,Temp_exp,C_exp;
reg A_sign,B_sign,C_sign;

always@(*)
begin
A_man = {1'b1,A[22:0]};
A_exp = A[30:23];
A_sign = A[31];

B_man = {1'b1,B[22:0]};
B_exp = B[30:23];
B_sign = B[31];

Temp_exp = A_exp+B_exp-127;
Temp_man = A_man*B_man;
C_man = Temp_man[47] ? Temp_man[46:24] : Temp_man[45:23];
C_exp = Temp_man[47] ? Temp_exp+1'b1 : Temp_exp;
C_sign = A_sign^B_sign;
end
assign C = {C_sign,C_exp,C_man};
endmodule
////////////////////////////////////////////////////////////////////////////////

module add(input [31:0]A,
input [31:0]B,
output reg [31:0] C);
reg [23:0] a_man,b_man,temp_man;
reg [22:0] C_man;
reg [7:0] C_exp,A_exp,B_exp,temp_exp,diff_exp;
reg C_sign,A_sign,B_sign,Temp_sign;
wire MSB;
reg [32:0] Temp;
reg carry,load;
reg comp;
reg [7:0] exp_adjust;
integer i;

always @(*)
begin

comp = (A[30:23] >= B[30:23])? 1'b1 : 1'b0;

a_man = comp ? {1'b1,A[22:0]} : {1'b1,B[22:0]};
A_exp = comp ? A[30:23] : B[30:23];
```

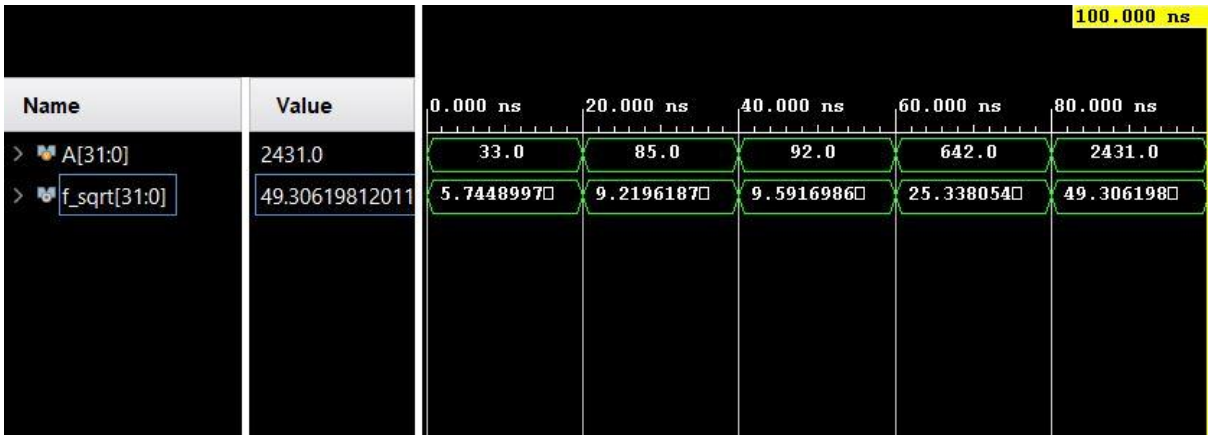
```
A_sign = comp ? A[31] : B[31];

b_man = comp ? {1'b1,B[22:0]} : {1'b1,A[22:0]};
B_exp = comp ? B[30:23] : A[30:23];
B_sign = comp ? B[31] : A[31];

diff_exp = A_exp-B_exp;
b_man = (b_man >> diff_exp);
{carry,temp_man} = (A_sign ~^ B_sign)? a_man + b_man : a_man-b_man ;
exp_adjust = A_exp;
if(carry)

begin
temp_man = temp_man>>1;
exp_adjust = exp_adjust+1'b1;
end

else
 begin
   load =0;
   for (i=0;i<24;i=i+1)
      begin
         if (temp_man[23] ==0 && load ==0)
         begin
           temp_man = temp_man<<1;
           exp_adjust =  exp_adjust-1'b1;
         end
         else load =1;
      end
   end
C_sign = A_sign;
C_man = temp_man[22:0];
C_exp = exp_adjust;
C= {C_sign,C_exp,C_man};
end

endmodule
```

Testbench:

```
module tsttb(  );

reg [31:0] A;
wire [31:0] f_sqrt;

tst s1 (.A(A),.f_sqrt(f_sqrt));

initial
begin

A = 32'h42040000;  // 33
#20
A = 32'h42aa0000;  // 85
#20
A = 32'h42b80000;  // 92
#20
A = 32'h44208000;  // 642
#20
A = 32'h4517f000;  // 2431

#20 $finish;

end

initial
begin
$monitor("A=%b, f_sqrt=%b ",$time,A,f_sqrt);
end

endmodule
```

Simulation Results:

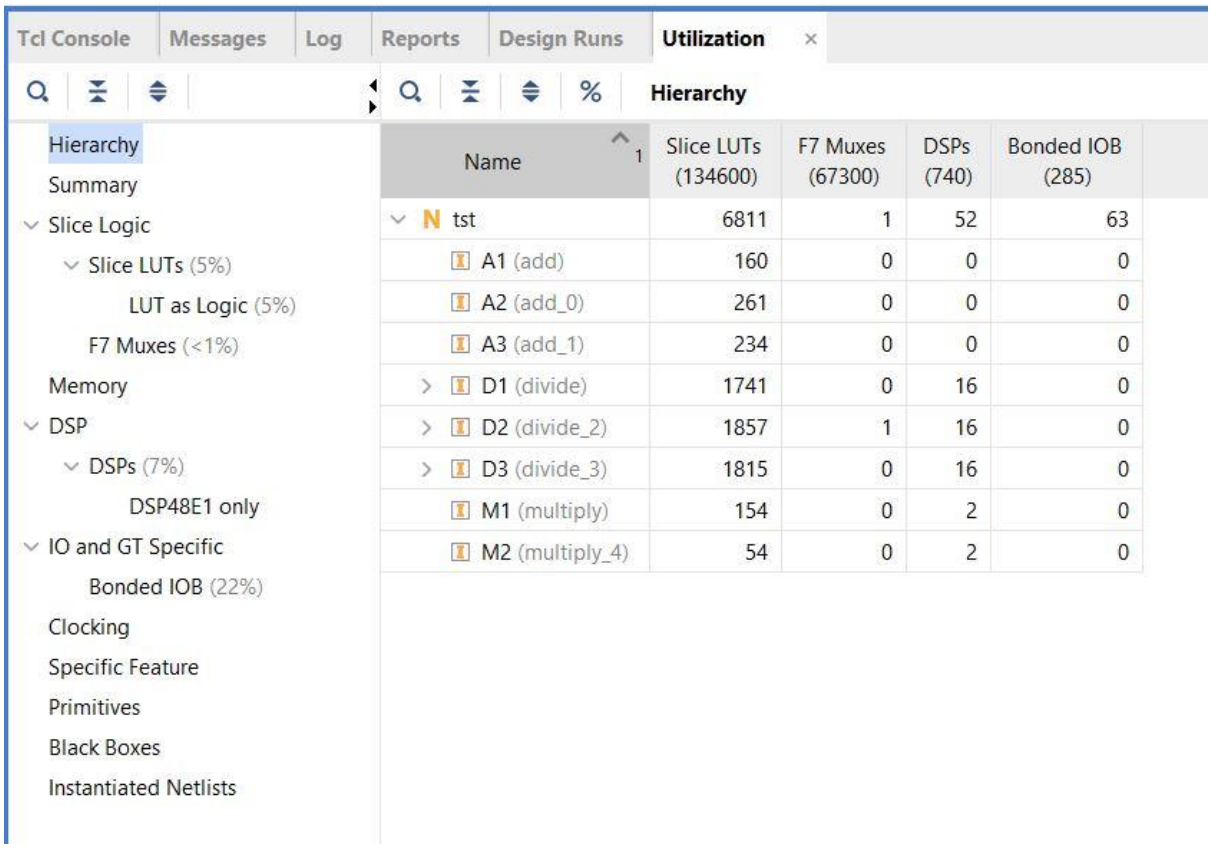




Here values are in hexadecimal format of IEEE754 standard, but when we convert it to decimal number system it results in:

Input (IEEE754) (Hexadecimal representation)	Input (Decimal Number System)	Output (IEEE754) (Hexadecimal representation)	Output (Decimal Number System)	Actual square root
42040000	33	40b7d375	5.74456262589	5.74456264653
42aa0000	85	41138340	9.21954345703	9.21954445729
42b80000	92	41197774	9.5916633606	9.59166304662
44208000	642	41cab3a6	25.3377189636	25.33771891863
4517f000	2431	4245387d	49.3051643372	49.30517214248

Utilization Report:



Conclusion

1. The implemented design is fully compatible with IEEE754 standard. Using the Newton Raphson method, the design is fast, and has high precision.
2. The implemented block can be used for implementing multiplier/divider blocks for and single-precision floating point as well.