Report On

Floating Pt Square-Root Implementation

submitted to

Dr. Gaurav Trivedi
Ankita Tiwari
Meenali Janveja

by

Abhishek Verma (214102401)
Chaitanya Tejaswi (214102408)
Pawan Kumar (214102412)

April 26, 2022

Department of Electronics and Electrical Engineering
Indian Institute of Technology, Guwahati
Guwahati-781039, India

Aim

Perform post-synthesis simulations & functionality testing for Floating Pt Square-Root Implementation.

Theory

Introduction

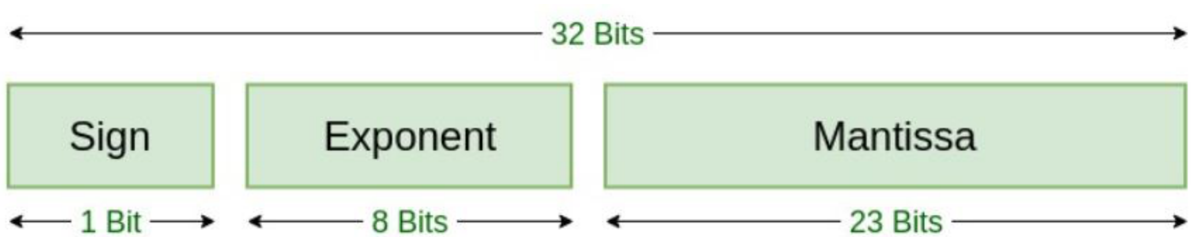
With the fast development of sub-micron technology, the density of silicon grows rapidly and the cost of hardware decreases, more and more functionalities are transferred into hardware to realize. In order to meet the ever increasing demand in high performance applications including scientific computations, 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:

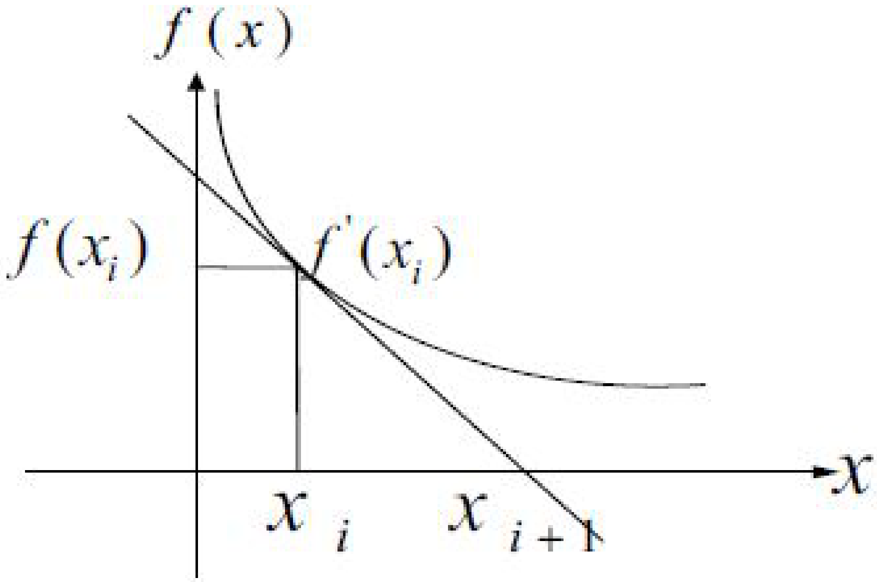
- Digital recurrence, itâs simple, easy to implement, but the latency is long, especially for large operand sizes.
- Functional iteration, such as NR and Goldschmidt iteration algorithms. It requires more hardware cost, and the result is not fully accurate, but it is fast, scalable, easy to pipeline, and has high precision.
- Very high radix arithmetic, it is fast, but very complicated to implement.
- Table look up, it is simple, fast, with big hardware cost and bad extensibility. 5) Variable latency, it is fast with more hardware cost.

IEE754 Representation of Floating Pt Numbers

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:

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

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

This is the NR formula, where x_i is the value at the ith 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:

1. Operation of sign: $S_q = 0$ (if $S_b = 1$ then square root is not possible).
2. Operation of exponent: $E_q = E_b + bias$.
3. Operation of Mantissa: square root of mantissa is calculated using NR method.

SquareRoot Algorithm

We perform NR square-root appoximation using the function: $x_{i+1} = \frac{x_i}{2}(3 + dx_i^2)$. The steps are as follows:

1. NR approximation of x_n.
2. Partial Product
3. Addition
4. Normalization, to get IEEE754 floating format.

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: $X = Y \times C$

$2^{(E/2)}$ 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 Listing

Newton-Raphson (Full):

```
`timescale 1ns/1ps
module tst(
    input [31:0]A,
    input clk,
    input reset,
    output [31:0] f_sqrt
);
    wire [7:0] A_exponent;
    wire [22:0] A_mantissa;
    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;

    // LookUp Values
    assign x0 = 32'h3f5a827a;
    assign sqrt_1by05 = 32'h3fb504f3;
    assign sqrt_2 = 32'h3fb504f3;
    assign sqrt_1by2 = 32'h3f3504f3;

    // Number in IEEE754 Format
    assign A_sign = A[31];
    assign A_exponent = A[30:23];
    assign A_mantissa = A[22:0];

    // First Iteration
    divide D1(.A({1'b0,8'd126,A_mantissa}),.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_mantissa}),.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_mantissa}),.B(x2),.C(temp5));
    add A3(.A(temp5),.B(x2),.C(temp6));
    assign x3 = {temp6[31],temp6[30:23]-1,temp6[22:0]};
    multiply M1(.A(x3),.B(sqrt_1by05),.C(temp7));

    assign pos = (A_exponent>=8'd127) ? 1'b1 : 1'b0;
    assign Exp_2 = pos ? (A_exponent-8'd127)/2 : (A_exponent-8'd127-1)/2 ;
    assign remainder = (A_exponent-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;
```

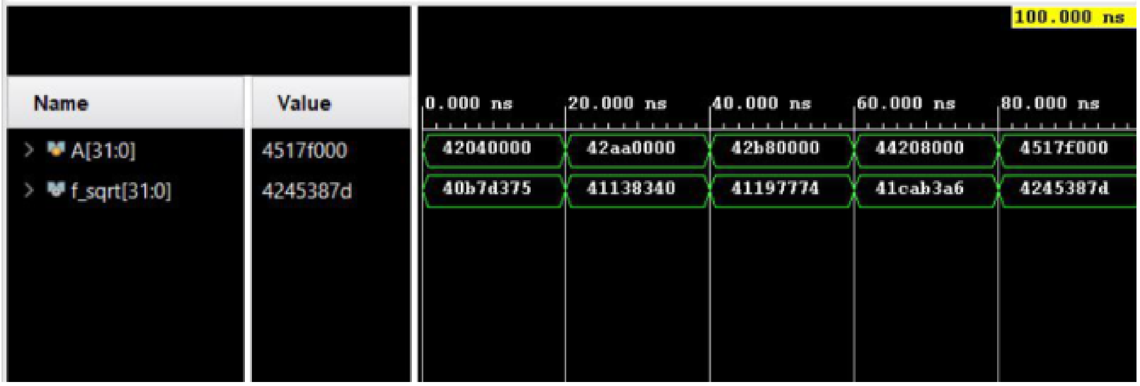
```
    wire [31:0] x0,x1,x2,x3;
    wire [31:0] temp1,temp2,temp3,temp4,temp5,temp6,temp7,debug;

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

    //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));
    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));

    //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_mantissa,B_mantissa;
    reg [22:0] C_mantissa;
    reg [47:0] Temp_mantissa;
    reg [7:0] A_exponent,B_exponent,Temp_exponent,C_exponent;
    reg A_sign,B_sign,C_sign;

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

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

        Temp_exponent = A_exponent+B_exponent-127;
        Temp_mantissa = A_mantissa*B_mantissa;
        C_mantissa = Temp_mantissa[47] ? Temp_mantissa[46:24] : Temp_mantissa[45:23];
        C_exponent = Temp_mantissa[47] ? Temp_exponent+1'b1 : Temp_exponent;
        C_sign = A_sign^B_sign;
    end
    assign C = {C_sign,C_exponent,C_mantissa};

endmodule


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

```verilog

    reg [23:0] a_mantissa,b_mantissa,temp_mantissa;
    reg [22:0] C_mantissa;
    reg [7:0] C_exponent,A_exponent,B_exponent,temp_exponent,diff_exponent;
    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_mantissa = comp ? {1'b1,A[22:0]} : {1'b1,B[22:0]};
        A_exponent = comp ? A[30:23] : B[30:23];
        A_sign = comp ? A[31] : B[31];

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

        diff_exponent = A_exponent-B_exponent;
        b_mantissa = (b_mantissa >> diff_exponent);
        {carry,temp_mantissa} = (A_sign ~^ B_sign)? a_mantissa + b_mantissa : a_mantissa-b_mantissa ;
        exp_adjust = A_exponent;
        if(carry) begin
            temp_mantissa = temp_mantissa>>1;
            exp_adjust = exp_adjust+1'b1;
        end else begin
            load =0;
            for (i=0;i<24;i=i+1) begin
                if (temp_mantissa[23] ==0 && load ==0) begin
                   temp_mantissa = temp_mantissa<<1;
                   exp_adjust =  exp_adjust-1'b1;
                end else load =1;
            end
        end

        C_sign = A_sign;
        C_mantissa = temp_mantissa[22:0];
        C_exponent = exp_adjust;
        C= {C_sign,C_exponent,C_mantissa};

    end

endmodule
```

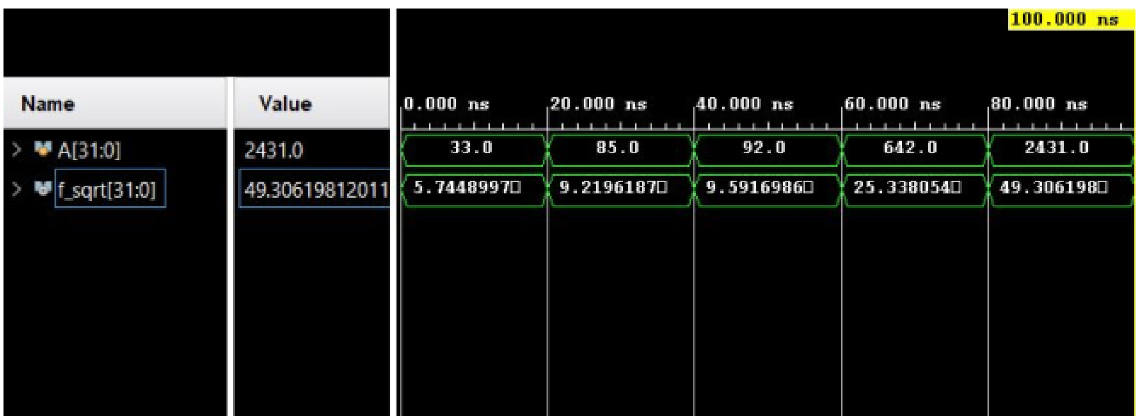
TestBench:

```
`timescale 1ns/1ps
module tsttb();
    reg [31:0] A;
    wire [31:0] f_sqrt;
    tst s1 (.A(A),.f_sqrt(f_sqrt)
);
    initial begin
    $monitor("A=%h, f_sqrt=%h", A, f_sqrt);
        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

endmodule
```

Results

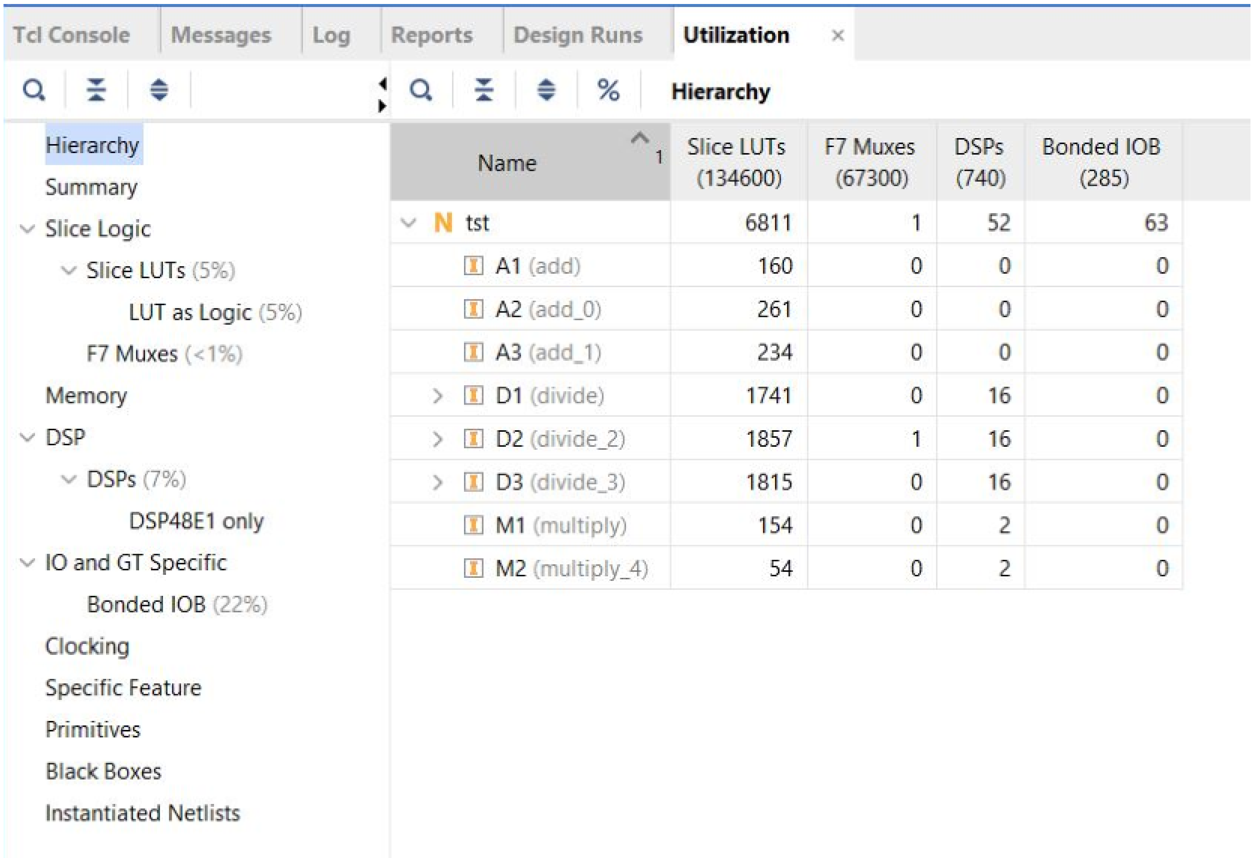


Here values are in hexadecimal format of IEEE754 standard, but when we convert it to decimal number system it results in:



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

Simulation Results



Name	Slice LUTs (134600)	F7 Muxes (67300)	DSPs (740)	Bonded IOB (285)
tst	6811	1	52	63
A1 (add)	160	0	0	0
A2 (add_0)	261	0	0	0
A3 (add_1)	234	0	0	0
D1 (divide)	1741	0	16	0
D2 (divide_2)	1857	1	16	0
D3 (divide_3)	1815	0	16	0
M1 (multiply)	154	0	2	0
M2 (multiply_4)	54	0	2	0

Utilization Results

Conclusions

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

Appendix

Work done prior to mid-semester.

Floating Pt Square-Root Implementation

Midsem Report

Abhishek Verma (214102401)
Chaitanya Tejaswi (214102408)
Pawan Kumar (214102412)

20-04-2022

Aim

Perform post-synthesis simulations & functionality testing for Floating Pt Square-Root Implementation.

Implementation#1: 32bit Integer Square-Root

Approach

We square-root using pen-paper ("restoring") approach. Examples for decimal/binary square-roots this way are given below:

Decimal

		q_1 q_2 q_3 q_4 q_5 q_6	
		1 7 3 2 0 5	; Square root
	1	$\sqrt{}$ 03 00 00 00 00 00	; Radicand
$(q_1 = 1)$	1	1	; $1 \times 1 = 1$ (guess 1)
	27	2 00	; $1 + 1 = 2$
$(q_2 = 7)$	7	1 89	; $27 \times 7 = 189$ (guess 7)
	343	11 00	; $27 + 7 = 34$
$(q_3 = 3)$	3	10 29	; $343 \times 3 = 1029$ (guess 3)
	3462	71 00	; $343 + 3 = 346$
$(q_4 = 2)$	2	69 24	; $3462 \times 2 = 6924$ (guess 2)
	34640	1 76 00	; $3462 + 2 = 3464$
$(q_5 = 0)$	0	0	; $34640 \times 0 = 0$ (guess 0)
	346405	1 76 00 00	; $34640 + 0 = 34640$
$(q_6 = 5)$	5	1 73 20 25	; $346405 \times 5 = 1732025$ (guess 5)
		2 79 75	; Remainder

Binary

		q_1 q_2 q_3 q_4	
		1 0 1 1	; Square root (11_{10})
	1	$\sqrt{}$ 01 11 11 11	; Radicand (127_{10})
$(q_1 = 1)$	1	1	; $1 \times 1 = 1$ (guess 1)
	100	0 11	; $1 + 1 = 10$
$(q_2 = 0)$	0	0	; $100 \times 0 = 0$ (guess 0)
	1001	11 11	; $100 + 0 = 100$
$(q_3 = 1)$	1	10 01	; $1001 \times 1 = 1001$ (guess 1)
	10101	1 10 11	; $1001 + 1 = 1010$
$(q_4 = 1)$	1	1 01 01	; $10101 \times 1 = 10101$ (guess 1)
		01 10	; Remainder (6_{10})

Implementation#2: 32bit Floating Pt Square-Root

Approach

We perform NR square-root appoximation using the function: $x_{i+1} = \frac{x_i}{2}(3 + dx_i^2)$. The steps are as follows:

1. NR approximation of x_n.
2. Partial Product
3. Addition
4. Normalization, to get IEEE754 floating format.

Newton/Raphson SquareRoot Approximation

1.) $\sqrt{0.25} = 0.5$
$\sqrt{0x0.40000000} = \sqrt{\frac{4}{16}} = 0x0.80000000 = \frac{8}{16} = 0.5$
2.) $\sqrt{0.75} = 0.8660254038$
$\sqrt{0x0.c0000000} = \sqrt{\frac{12}{16}} = 0x0.ddb3d743 = \frac{13}{16^1} + \frac{13}{16^2} + \frac{11}{16^3} + \frac{3}{16^4} + \frac{13}{16^5} + \frac{7}{16^7} + \frac{4}{16^4} + \frac{4}{16^3} = 0.8660254038$

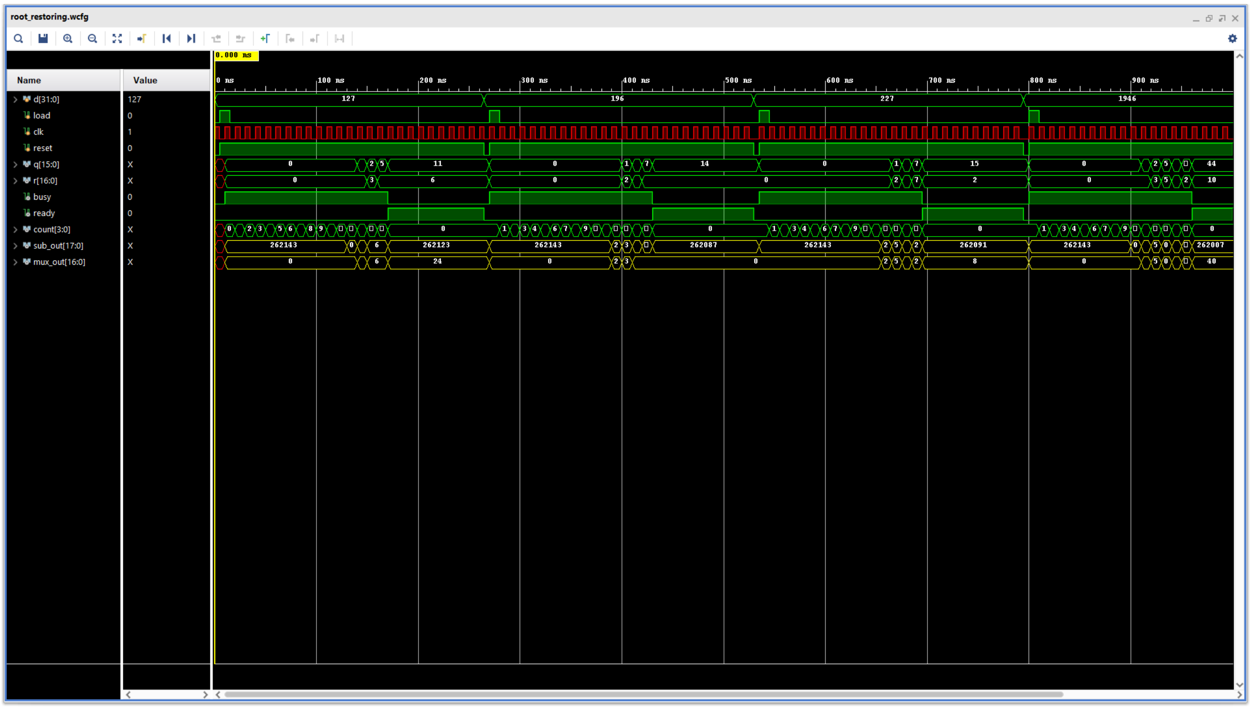
(correct upto 12 decimal places).

Let d such that $\frac{1}{4} \leq d < 1$, ie: $(= 0.1xxx..., 0.01xxx..., 0.001xxx..., ...)$
Calculate $q = \sqrt{d}$
We calculate $x_n = \frac{1}{\sqrt{d}}$, then obtain $q = dx_n = d\frac{1}{\sqrt{d}} = \sqrt{d}$
Let $f(x) = \frac{1}{x^2} - d$, then $f(x) = 0$ for $x = \frac{1}{\sqrt{d}}$.

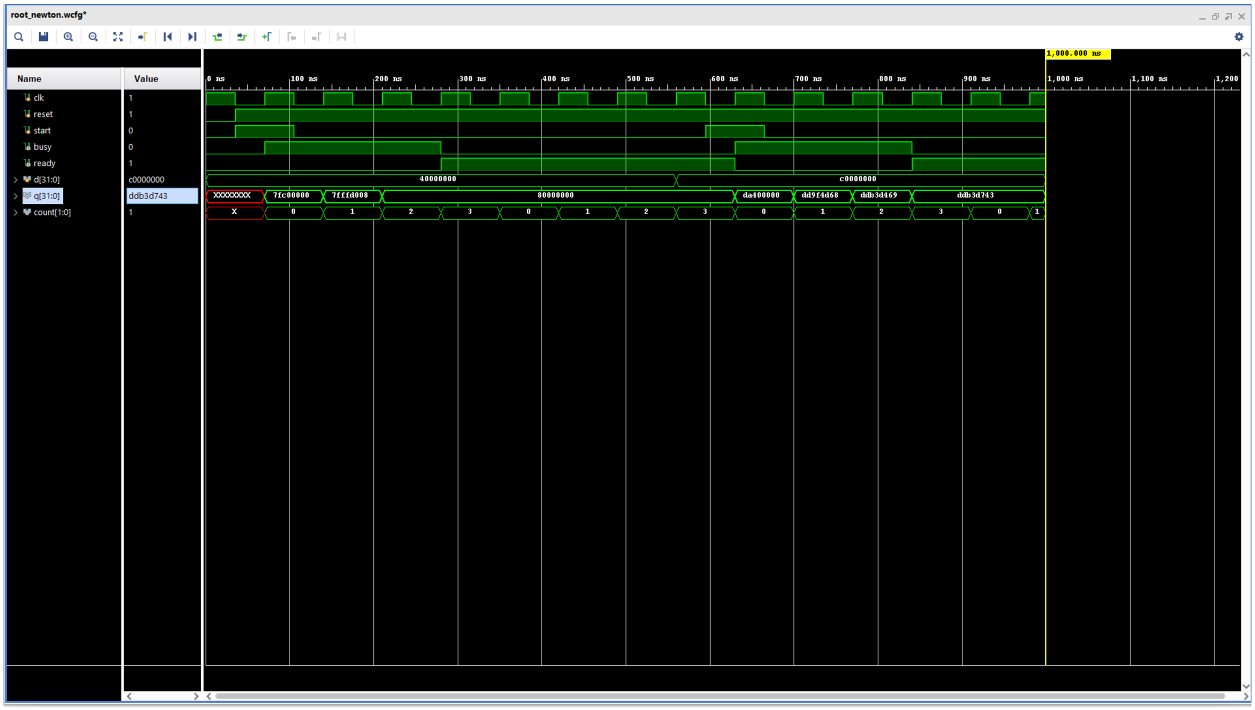
$$\begin{aligned} x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} \\ &= x_i - \frac{\frac{1}{x_i^2} - d}{-\frac{2}{x_i^3}} \\ &= \frac{x_i}{2}(3 + dx_i^2) \end{aligned}$$

Procedure

1. Create new project in Vivado, and source appropriate files for each (one for logic, one for testbench).
2. Run behavioral simulation, and RTL analysis/synthesis/implementation.
3. Using the Tcl commands **report_timing_summary** & **report_power**, note down the delay (slack), power consumption, and no. of LUTs/FFs used for each implementation.



Implementation 1: Restoring SquareRoot



Implementation 2: Newton-Raphson (Basic) SquareRoot

Implementation 2: Newton-Raphson (Full) SquareRoot

Results

.	**LUTs**	**FFs**	**Delay**	**Power**
32-Bit Integer Sqrt	111	12	13.187ns	$3.326W$
32-Bit FloatingPt Sqrt	1326	54	22.47ns	$16.475W$

Table 1: Timing/Power Values for Implementations

Code Listing

Restoring:

```
`timescale 1ns/1ns
module root_restoring (d,load,clk,reset,q,r,busy,ready,count);
    input  [31:0] d;                  // Input
    input         load;
    output reg    busy, ready;
    input         clk, reset;
    output [15:0] q;                  // Root
    output [16:0] r;                  // Remainder
    output  [3:0] count;
    reg    [31:0] reg_d;
    reg    [15:0] reg_q;
    reg    [16:0] reg_r;
    reg     [3:0] count;
    wire   [17:0] sub_out = {reg_r[15:0],reg_d[31:30]} - {reg_q,2'b1}; // -
    wire   [16:0] rem_out = sub_out[17]?                           // restoring
                    {reg_r[14:0],reg_d[31:30]} : sub_out[16:0]; // or not
    assign q = reg_q;
    assign r = reg_r;
    always @(posedge clk or negedge reset) begin
        if (!reset) begin
            busy  <= 0; ready <= 0;
        end else begin
            if (load) begin
                reg_d <= d; reg_q <= 0; reg_r <= 0;
                busy  <= 1; ready <= 0; count <= 0;
            end else if (busy) begin
                // Next-State Logic
                reg_d <= {reg_d[29:0],2'b0};
                reg_q <= {reg_q[14:0],~sub_out[17]};
                reg_r <= rem_out;
                count <= count + 4'b1;
                if (count == 4'hf) begin             // max 16 iterationns
                    busy  <= 0; ready <= 1;          // done! q,r ready
                end
            end
        end
    end
endmodule
```

TestBench:

```
`timescale 1ns/1ns
module root_restoring_tb;
    reg  [31:0] d;
    reg         load;
    reg         clk,reset;
    wire [15:0] q;
    wire [16:0] r;
    wire        busy;
    wire        ready;
    wire  [3:0] count;
    root_restoring rt (d,load,clk,reset,q,r,busy,ready,count);
    initial begin
        $dumpfile ("root_restoring.vcd");
        $dumpvars (0, root_restoring);
        $monitor ("At ",$time, ": D=%d, Root=%d, Remainder=%d",d,q,r);
             reset  = 0; load  = 0; clk   = 1; d     = 32'h0000007f;
        #5   reset  = 1; load  = 1;
        #10  load  = 0;
        #250 reset  = 0; load  = 0; clk   = 1; d     = 32'h000000c4;
        #5   reset  = 1; load  = 1;
        #10  load  = 0;
        #250 reset  = 0; load  = 0; clk   = 1; d     = 32'h000000e3;
        #5   reset  = 1; load  = 1;
        #10  load  = 0;
        #250 reset  = 0; load  = 0; clk   = 1; d     = 32'h000000e3;
        #5   reset  = 1; load  = 1;
        #10  load  = 0;
        #250 \$finish;
    end
    always #5 clk = !clk;
endmodule
```

Newton-Raphson (Basic):

```
module root_newton (d,clk,reset,start,q,busy,ready,count);
    input  [31:0] d;           // Input
    input         clk, reset;
    input         start;
    output [31:0] q;           // Root
    output reg    busy, ready;
    output  [2:0] count;
    reg    [31:0] reg_d;
    reg    [33:0] reg_x;
    reg     [1:0] count;
    // Partial Sums for: X[i+1] = Xi * (3 - Xi * Xi * d) / 2
    // X2, X2d, (3-X2d), X(3-X2d), Xd, X0
    wire   [67:0] X2     = reg_x * reg_x;
    wire   [67:0] X2d    = reg_d * X2[67:32];
    wire   [33:0] k_X2d  = 34'h300000000 - X2d[65:32];
    wire   [67:0] Xk_X2d = reg_x * k_X2d;
    wire   [65:0] Xd     = reg_d * reg_x;
    wire    [7:0] X0     = lut(d[31:27]);
    assign        q      = Xd[63:32] + |Xd[31:0]; // RoundOff
    always @(posedge clk or negedge reset) begin
        if (!reset) begin
            busy  <= 0; ready <= 0;
        end else begin
            if (start) begin
                reg_d <= d; reg_x <= {2'b1,X0,24'b0};
                busy  <= 1; ready <= 0; count <= 0;
            end else begin
                reg_x <= Xk_X2d[66:33]; count <= count + 2'b1;
                if (count == 2'h2) begin            // max 3 iterations
                    busy <= 0; ready <= 1;          // done! q,r ready
                end
            end
        end
    end
    function [7:0] lut; // 1/sqrt(d)
        input [4:0] d;
        case (d)
            5'h08: lut = 8'hff;
            5'h09: lut = 8'he1;
            5'h0a: lut = 8'hc7;
            5'h0b: lut = 8'hb1;
            5'h0c: lut = 8'h9e;
            5'h0d: lut = 8'h9e;
            5'h0e: lut = 8'h7f;
            5'h0f: lut = 8'h72;
            5'h10: lut = 8'h66;
            5'h11: lut = 8'h5b;
            5'h12: lut = 8'h51;
            5'h13: lut = 8'h48;
            5'h14: lut = 8'h3f;
            5'h15: lut = 8'h37;
            5'h16: lut = 8'h30;
            5'h17: lut = 8'h29;
            5'h18: lut = 8'h23;
            5'h19: lut = 8'h1d;
            5'h1a: lut = 8'h17;
            5'h1b: lut = 8'h12;
            5'h1c: lut = 8'h0d;
            5'h1d: lut = 8'h08;
            5'h1e: lut = 8'h04;
            5'h1f: lut = 8'h00;
          default: lut = 8'hff;
        endcase
    endfunction
endmodule
```

TestBench:

```
`timescale 1ns/1ns
module root_newton_tb;
    reg  [31:0] d;
    reg         start;
    reg         clk,reset;
    wire [31:0] q;
    wire        busy;
    wire        ready;
    wire  [1:0] count;
    root_newton root (d,clk,reset,start,q,busy,ready,count);
    initial begin
        $dumpfile ("root_newton.vcd");
        $dumpvars (0, root_newton);
        $monitor ("At ",$time, ": D=0.%h, Root=0.%h",d,q);

        // d = 0x0.40000000 = 0.25
        // sqrt(d) = 0x0.80000000 = 0.5
             reset=0; start=0; clk=1; d=32'h40000000;
        #35  reset=1; start=1;
        #70  start= 0;

        // d = 0x0.c0000000 = 0.75
        // sqrt(d) = 0x0.ddb3d743 = 0.8660254038
        #455 d    = 32'hc0000000;
        #35  start= 1;
        #70  start= 0;

        // d = 0x0.f8000000 = 0.96875
        // sqrt(d) = 0.9842509843 = 0x0.fbe00000
        #455 d    = 32'hf8000000;
        #35  start= 1;
        #70  start= 0;

        #455 \$finish;
    end
    always #35 clk = !clk;
endmodule
```