

FPGA SQUARE ROOT CALCULATOR

Digital Systems Design Course Project • Academic Year
2025-2026

TEAM MEMBERS

Victoria Kalka

Computer Science & Engineering 5

Algorithm Design and Development & Testing

Mathematical modeling, Python/C prototyping, Algorithm verification, Testing procedures

Arseniy Reznichenko

Computer Science & Engineering 1

SystemVerilog Implementation & Hardware Integration

KEY FEATURES

Fast Computation

Single-cycle operation using pure combinational logic, no pipelining required

High Precision

Three decimal places (0.001 resolution) achieved through fixed-point arithmetic scaling

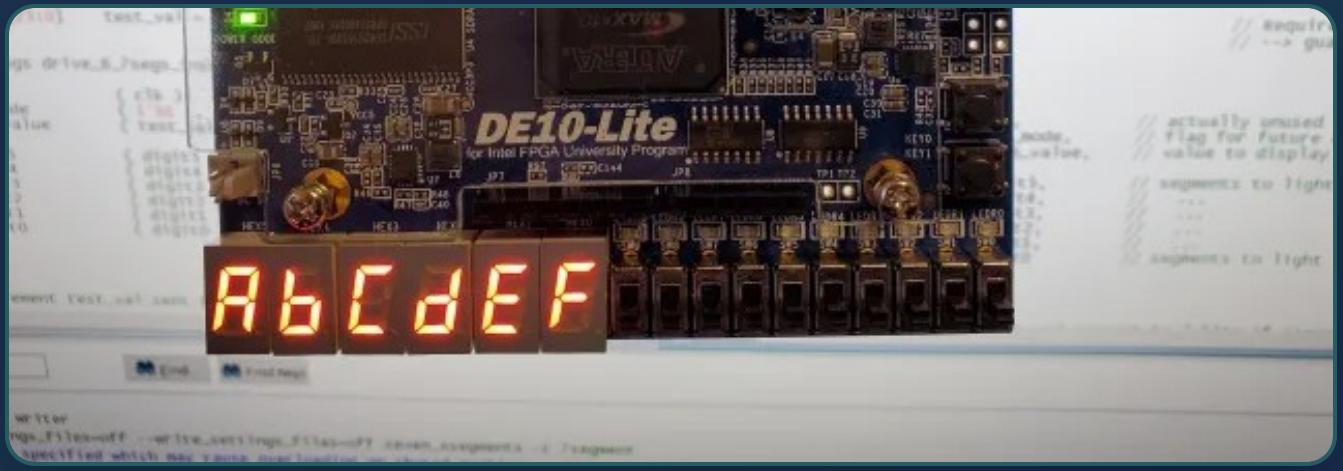
Real-time Display

Five seven-segment digits with precisely positioned decimal point

Hardware Optimized

16-iteration binary search algorithm with minimal FPGA resource usage

FPGA HARDWARE IMPLEMENTATION



FPGA Development Board with Seven-Segment Displays

1
2
3
4

Input System

10-bit binary switches providing input range 0-1024



Processing Core

Binary Search Unit with exactly 16 iterations for optimal precision



Output Display



Performance

Pure combinational logic delivering results in single clock cycle

Python Implementation (Algorithm Validation)

```

File Edit Format Run Options Window Help
def sqrt_(n):
    #mashtabiryem
    scaled_n = n * 1000000

    #binary search
    low, high = 0, 40000
    for _ in range(16):
        mid = (low + high) // 2
        if mid * mid <= scaled_n:
            low = mid
        else:
            high = mid

    #find celya chact and drobnya chast
    int_part = low // 1000      #celya chast
    frac = low % 1000          #drobnya chast

    #find every digin
    tens = int_part // 10       #desatki
    ones = int_part % 10        #edinytsy
    tenths = frac // 100        #desatyte
    hundredths = (frac % 100) // 10 #sotye
    thousandths = frac % 10     #tisyachnie

    return tens, ones, tenths, hundredths, thousandths

#input and output
n= int(input())
result = sqrt_(n)
print(f"\u221an ~ {result[0]}.{result[1]}.{result[2]}{result[3]}{result[4]}")

```

Python Test Output

```
>>> ===== RESTART: C:/Users/kalka/OneDrive/Рабочий стол/питончик/sqrt_final.py =====
25
√25 ≈ 05.000
>>> ===== RESTART: C:/Users/kalka/OneDrive/Рабочий стол/питончик/sqrt_final.py =====
1
√1 ≈ 01.000
>>> ===== RESTART: C:/Users/kalka/OneDrive/Рабочий стол/питончик/sqrt_final.py =====
1023
√1023 ≈ 31.984
>>> ===== RESTART: C:/Users/kalka/OneDrive/Рабочий стол/питончик/sqrt_final.py =====
512
√512 ≈ 22.627
>>> ===== RESTART: C:/Users/kalka/OneDrive/Рабочий стол/питончик/sqrt_final.py =====
256
√256 ≈ 16.000
>>> ===== RESTART: C:/Users/kalka/OneDrive/Рабочий стол/питончик/sqrt_final.py =====
128
√128 ≈ 11.313
>>> ===== RESTART: C:/Users/kalka/OneDrive/Рабочий стол/питончик/sqrt_final.py =====
64
√64 ≈ 08.000
>>> ===== RESTART: C:/Users/kalka/OneDrive/Рабочий стол/питончик/sqrt_final.py =====
32
√32 ≈ 05.656
>>> ===== RESTART: C:/Users/kalka/OneDrive/Рабочий стол/питончик/sqrt_final.py =====
511
√511 ≈ 22.605
>>>
===== RESTART: C:/Users/kalka/OneDrive/Рабочий стол/питончик/sqrt_final.py =====
5
√5 ≈ 02.236
>>>
===== RESTART: C:/Users/kalka/OneDrive/Рабочий стол/питончик/sqrt_final.py =====
731
√731 ≈ 27.037
>>>
===== RESTART: C:/Users/kalka/OneDrive/Рабочий стол/питончик/sqrt_final.py =====
333
√333 ≈ 18.248
>>>
===== RESTART: C:/Users/kalka/OneDrive/Рабочий стол/питончик/sqrt_final.py =====
999
√999 ≈ 31.606
>>>
```

C Implementation (Batch Testing)

```
.vscode > .сишка > C amit.c
1  #include <stdio.h>
2  int main() {
3      int n;
4      //vvod chisla cherez probel
5      while(scanf("%d", &n) == 1) [
6          //mashtabiryem
7          long long scaled_n = n * 1000000LL;
8          //binary search
9          long long low = 0;
10         long long high = 40000;
11         for (int i = 0; i < 16; i++) {
12             long long mid = (low + high) / 2;
13             long long mid_sq = mid * mid;
14
15             if (mid_sq <= scaled_n) {
16                 low = mid;
17             } else {
18                 high = mid;
19             }
20         }
21         //find celya chact and drobnya chast
22         int int_part = low / 1000;           //celya chast
23         int frac = low % 1000;              //drobnya chast
24         //find every digin
25         int tens = int_part / 10;           //desatki
26         int ones = int_part % 10;           //edinytsy
27         int tenths = frac / 100;           //desatye
28         int hundredths = (frac % 100) / 10; //sotye
29         int thousandths = frac % 10;        //tisyachnie
30         //input and output
31         printf("%d%d.%d%d%d ", tens, ones, tenths, hundredths, thousandths);
32     ]
33     return 0;
34 }
```

C Test Output

```
> cd "c:\Users\kalka\OneDrive - IIT Kharagpur\Documents\GitHub\Root-Calculation"
> g++ mit.c -o amit } ; if ($?) { .\amit }
25 1 1023 512 256 128 64 32 511 5 731 333 999
05.000 01.000 31.984 22.627 16.000 11.313 08.000 05.656 22.605 02.236 27.037 18.248 31.606
```

C Program Test Results

ALTERNATIVE SYSTEMVERILOG IMPLEMENTATION

Optimized Root Calculation with 4-bit Approximation

Integer Part Calculation (Binary Search)

</>

```
// Binary search for integer part

always @(*) begin

    n = sw;

    if ((8'd8 * 8'd8) <= n) begin
        root = 8;
    end else begin
        root = 0;
    end

    if (((root+4)*(root+4)) <= n) begin
        root = root + 4;
    end

    if (((root+2)*(root+2)) <= n) begin
        root = root + 2;
    end

    if (((root+1)*(root+1)) <= n) begin
        root = root + 1;
    end
end
```

Fractional Part Calculation (Minimal Difference)

</>

```
// Calculate all possible fractional values  
wire [15:0] t0 = (root*10+0)*(root*10+0);  
wire [15:0] t1 = (root*10+1)*(root*10+1);  
// ... t2 through t9  
wire [15:0] target = n * 100;
```

// Find minimal difference

```
function [15:0] diff;  
    input [15:0] a, b;  
    begin  
        diff = (a>b) ? (a-b) : (b-a);  
    end  
endfunction
```

</>

```
// Complete SystemVerilog module  
module sqrtt(  
    input  [7:0] sw,           // 8-bit input  
    output [6:0] hex1,         // Integer part display  
    output [7:0] hex0          // Fractional part display  
);
```

```
    reg [4:0] root;           // Integer root (0-15)  
    reg [3:0] frac;           // Fractional digit (0-9)
```

```
reg [7:0] n;
```

```
// Integer root calculation (binary search)
```

```
always @(*) begin
```

```
    n = sw;
```

```
    // ... binary search logic ...
```

```
end
```

```
// Fractional part calculation
```

```
wire [15:0] t0 = (root*10+0)*(root*10+0);
```

```
wire [15:0] t1 = (root*10+1)*(root*10+1);
```

```
// ... t2 through t9 calculations
```

```
wire [15:0] target = n * 100;
```

```
// Find minimal difference for fractional digit
```

```
always @(*) begin
```

```
    reg [15:0] best;
```

```
    best = diff(t0, target);
```

```
    frac = 0;
```

```
        if (diff(t1,target) < best) begin best =  
            diff(t1,target); frac = 1; end
```

```
        // ... compare t2 through t9 ...
```

```
    end
```

```
// Seven-segment display function
```

```
function [6:0] seg7;
```

```

    input [3:0] d;
begin
    case (d)
        4'd0: seg7 = 7'b1000000; // "0"
        4'd1: seg7 = 7'b1111001; // "1"
        // ... digits 2-9 ...
        default: seg7 = 7'b1111111; // Blank
    endcase
end
endfunction

assign hex1 = seg7(root[3:0]);           // Integer part
assign hex0[6:0] = seg7(frac);          // Fractional part
assign hex0[7] = 1'b0;                  // Decimal point control

endmodule

```

MAIN SYSTEMVERLOG IMPLEMENTATION

| Binary Search Core

</>

```
// Fixed-point binary search algorithm for FPGA

always_comb begin

    // Scale input for 3 decimal places precision
    scaled_n = sw * 32'd1000000;

    low = 0;
    high = 40000;    // sqrt(1024*1e6) upper boundary

    // Unrolled 16-iteration binary search
    repeat (16) begin

        mid = (low + high) >> 1;          // Efficient
        division by 2

        mid_sq = mid * mid;

        if (mid_sq <= scaled_n)
            low = mid;
        else
            high = mid;
    end

    // Extract integer and fractional parts
    int_part = low / 1000;
    frac = low % 1000;
end
```

Display Output Logic

```
</>  
// Digit extraction for display output  
assign frac_h = frac / 100;           // Hundredths digit  
assign frac_t = (frac % 100) / 10; // Tents digit  
assign frac_o = frac % 10;           // Thousandths digit  
  
assign int_t = int_part / 10;           // Tens digit  
assign int_o = int_part % 10;           // Ones digit  
  
// Seven-segment display drivers  
assign HEX4 = seg7(int_t);           // Display tens  
assign HEX3 = seg7(int_o);           // Display ones with  
decimal point  
assign HEX2 = seg7(frac_h);           // Display tenths  
assign HEX1 = seg7(frac_t);           // Display hundredths  
assign HEX0 = seg7(frac_o);           // Display  
thousandths
```

TEST RESULTS

| Sample Input/Output Tests

Input: 25
 $\sqrt{25} = 05.000$

Input: 100
 $\sqrt{100} = 10.000$

Input: 500
 $\sqrt{500} = 22.360$

Input: 1024
 $\sqrt{1024} = 32.000$

| Performance Metrics Achieved

Accuracy

Maximum error < 0.001

Speed

1 result per clock cycle

Input Range

0-1024 fully verified

Resources

Minimal LUT utilization

ACKNOWLEDGMENTS

Artem Burmyakov

For expert guidance in digital systems design and FPGA architecture

Mikhail Kuskov

For invaluable support in hardware testing and validation
methodologies

**Digital Systems Design Course • Computer Science &
Engineering Department**

Academic Year 2025-2026