KLE Technological University

Creating Value

Leveraging Knowledge

B. V. Bhoomaraddi Engineering & Technology College Campus, Hubballi - India

SQUARE ROOT ON FPGA

ADLD

COURSE PROJECT

Sno.	Name	USN
1.	Sneha S Anchekar	01FE20BEC291
2.	Sneha Kumari	01FE20BEC272

INTRODUCTION

Introduction:

The square root calculation is an essential mathematical operation required in various applications, including signal processing, image processing, and cryptography. In this report, we will discuss the implementation of the square root calculation using the long division method in Verilog on an FPGA (Field-Programmable Gate Array). The long division method is a widely used technique to approximate the square root of a number iteratively. By leveraging the power of FPGAs, we can design efficient hardware circuits to perform square root calculations in real-time, making it suitable for applications that require high-speed processing.

The objective of this project is to develop a square root calculator that can accurately and efficiently calculate the square root of binary numbers using the long division method. By implementing this functionality in Verilog on an FPGA, we can leverage the parallel processing capabilities of the FPGA to achieve high-performance square root calculations.

In the following sections, we will delve into the details of the long division method and explain its adaptation to binary numbers. We will discuss the algorithmic steps involved in finding the square root using the long division method. We will also explain the design considerations and optimizations that can be applied when implementing the square root calculator in Verilog on an FPGA.

Additionally, we will explore the benefits of using FPGAs for implementing square root calculations. FPGAs offer reconfigurability, allowing us to modify and optimize the hardware design to achieve better performance. Moreover, FPGAs enable parallel processing, enabling us to perform multiple calculations simultaneously, leading to improved throughput.

To validate the functionality and performance of our square root calculator, we will implement and synthesize the Verilog design on an FPGA development board. We will evaluate the accuracy and efficiency of the implemented square root calculator by testing it with a range of binary numbers and comparing the results against software-based calculations.

In conclusion, this report aims to present an in-depth analysis of the implementation of the square root calculation using the long division method in Verilog on an FPGA. By leveraging the parallel processing capabilities and reconfigurability of FPGAs, we can develop a high-performance square root calculator suitable for real-time applications.

The Long Route

Example: √1111001

The number we're finding the root of is known as the **radicand**.

For our example, the radicand is `1111001` (121 in decimal).

Our algorithm works with pairs of digits. Before we begin our calculation, we need to split the radicand into pairs starting with the least significant. Thus `1111001` becomes `01 11 10 01`.

1 0 1 1 Our answer: one digit for each pair of digits in the radicand

√01 11 10 01 Our radicand

01 Bring down the most significant pair

- 01 Subtract 01

——

```
  00          Our answer is NOT negative, so our first answer digit is 1

  00 11       Bring down the next pair of digits and append to previous result
- 01 01       Append 01 to our existing answer, 1, and subtract it
———————
  11 00       Our result is negative, so our second answer digit is 0
              We discard the result because it's negative

  00 11 10    Keep the existing digits, 00 11, and append the next pair
-   10 01     Append 01 to our existing answer, 10, and subtract it
———————
    01 01     Our answer is NOT negative, so our next answer digit is 1

    01 01 01  Bring down the next pair of digits and append to previous result
-   01 01 01  Append 01 to our existing answer, 101, and subtract it
————————
        00    Our answer is NOT negative, so our next answer digit is 1
```

There are no more pairs of digits, and the result of our last step is 0, so our answer is exact:

`√1111001 = 1011` or in decimal `√121 = 11`.

We'll look at what happens for irrational roots, such as √2, when we come to fixed-point numbers.

Algorithm Implementation

To make our algorithm usable on an FPGA, we need to turn the steps into simple operations we can represent in Verilog. The main change is using shifts to select numbers to work on. We'll use four registers in our algorithm:

- **X** - input radicand we want the square root of
- **A** - holds the current value we're working on
- **T** - result of sign test
- **Q** - the square root

Input: X=01111001 (decimal 121)

Step A X T Q Description

———

00000000 01111001 00000000 0000 Starting values.

1 00000001 11100100 Left shift X by two places into A.

00000000 Set T = A - {Q,01}: 01 - 01.

0000 Left shift Q.

00000000 0001 Is T≥0? Yes. Set A=T and Q[0]=1.

2 00000011 10010000 Left shift X by two places into A.

11111100 Set T = A - {Q,01}: 11 - 101.

0010 Left shift Q.

Is T≥0? No. Move to next step.

3 00001110 01000000 Left shift X by two places into A.

00000101 Set T = A - {Q,01}: 1110 - 1001

0100 Left shift Q.

00000101 0101 Is T≥0? Yes. Set A=T and Q[0]=1.

4 00010101 00000000 Left shift X by two places into A.

00000000 Set T = A - {Q,01}: 10101 - 10101.

1010 Left shift Q.

00000000 1011 Is T≥0? Yes. Set A=T and Q[0]=1.

——

Output: Q=1010 (decimal 11), R=0 (remainder taken from final A).

Verilog Module

```
module adld_sqrt_cp(
input clk,
input start,
output reg busy,
output reg valid,
input [7:0] rad,
output reg [7:0] root,
output reg [7:0] rem
   );
parameter WIDTH=8;
reg [7:0] x, x_next;
reg [7:0] q, q_next;
reg [9:0] ac, ac_next;
reg [9:0] test_res;

   localparam ITER = WIDTH >> 1;
   reg [$clog2(ITER)-1:0] i;
```

```
  always @(*)
 begin
     test_res = ac - {q, 2'b01};
     if (test_res[WIDTH+1] == 0) begin
        {ac_next, x_next} = {test_res[WIDTH-1:0], x, 2'b0};
        q_next = {q[WIDTH-2:0], 1'b1};
     end
  else
  begin
        {ac_next, x_next} = {ac[WIDTH-1:0], x, 2'b0};
        q_next = q << 1;
     end
  end

  always @(posedge clk) begin
     if (start) begin
        busy <= 1;
        valid <= 0;
        i <= 0;
        q <= 0;
        {ac, x} <= {{WIDTH{1'b0}}, rad, 2'b0};
     end else if (busy) begin
        if (i == ITER-1) begin
           busy <= 0;
           valid <= 1;
           root <= q_next;
           rem <= ac_next[WIDTH+1:2];
        end else begin
           i <= i + 1;
           x <= x_next;
           ac <= ac_next;
           q <= q_next;
        end
     end
  end

endmodule
```

Verilog Testbench

```
module adld_sqrt_tb;


 // Inputs
 reg clk;
 reg start;
 reg [7:0] rad;
```

```
// Outputs
wire busy;
wire valid;
wire [7:0] root;
wire [7:0] rem;

// Instantiate the Unit Under Test (UUT)
adld_sqrt_cp uut (
 .clk(clk),
 .start(start),
 .busy(busy),
 .valid(valid),
 .rad(rad),
 .root(root),
 .rem(rem)
);


parameter CLK_PERIOD = 10;
   parameter WIDTH = 8;

 always #(CLK_PERIOD / 2) clk = ~clk;

initial begin
 clk = 1;

      #100 rad = 8'b00000000; // 0
            start = 1;
      #10 start = 0;

      #50 rad = 8'b00000001; // 1
            start = 1;
      #10 start = 0;

      #50 rad = 8'b01111001; // 121
            start = 1;
      #10 start = 0;

      #50 rad = 8'b01010001; // 81
            start = 1;
      #10 start = 0;

      #50 rad = 8'b01011010; // 90
            start = 1;
      #10 start = 0;

      #50 rad = 8'b11111111; // 255
            start = 1;
      #10 start = 0;

      #50 $finish;
```

end

endmodule

RESULT