Télécom Paris

Design of generic square root computing architectures on FPGA using VHDL

Advanced Digital Electronics

Amirhossein YOUSEFVAND

December 2025

1 Architecture 1 : Newton Method

Based on the Newton Method, the square root will be found using the following equation :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{1}$$

in which :

$$f(x) = x^2 - A \tag{2}$$

$$f'(x) = 2x \tag{3}$$

Final equation :

$$\begin{aligned} x_{n+1} &= x_n - \frac{x_n^2 - A}{2x_n} \\ &= x_n - \frac{1}{2}(x_n + \frac{A}{x_n}) \end{aligned}$$

$$x_{n+1} = \frac{1}{2}(x_n + \frac{A}{x_n}) \tag{4}$$

According to Eq. 4, we need to divide A by x_n, add the result to x_n, and then shift the result to the right to perform the division by 2.

```
x_next_temp := resize(shift_right(x_temp + unsigned(A) / x_temp, 1), n);
```

Snippet 1 – VHDL Code of Equation 4

The following state machine has been used for the behavioral implementation of Architecture 1 :

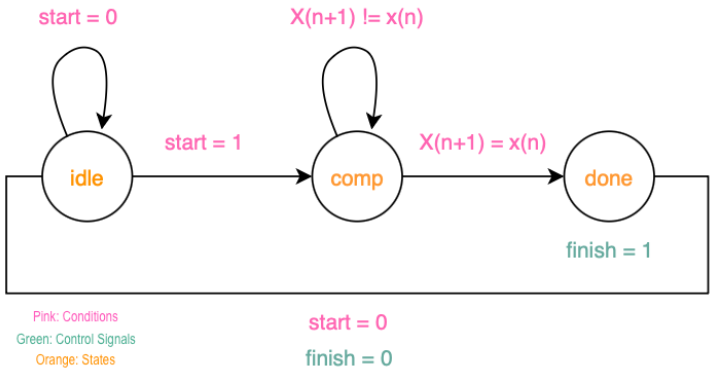


FIGURE 1 – State Machine of Architecture 1 (Newton Method)

As shown in Fig. 1, when the start signal is zero, the system stays in the idle state. When the start signal becomes one, we initialize the "x" value to "100...0", which represents $\frac{x}{2}+1$. There are other forms of initialization, each with its own advantages and disadvantages, but one of the optimal ways in terms of area and timing is initialization to $\frac{x}{2}+1$. Then the computation phase starts :

```
when comp =>
    x_temp := resize(x, 2*n);
    if x_temp = 0 then
            x_next_temp := (others => '0');
    else
            x_next_temp := resize(shift_right(x_temp + unsigned(A) / x_temp, 1), n);
    end if;
```

Snippet 2 – Computation State of Architecture 1

When the result of the computation is equal for two consecutive iterations, it means the root has been found and the state changes to "done". Subsequently, the "finish" signal will be equal to one.

1.1 Results

Nine different values were used for the verification of the code. The values and the square roots used in the testbench are listed in Table 1.

TABLE 1 – Simulation Input Values and Expected Square Roots

Input Value (A_tb)	**Integer Square Root** ($\lfloor\sqrt{A}\rfloor$)
5,499,030	2,345
1,194,877,489	34,567
3	1
7	2
0	0
1	1
512	22
15	3
4,294,967,295	65,535

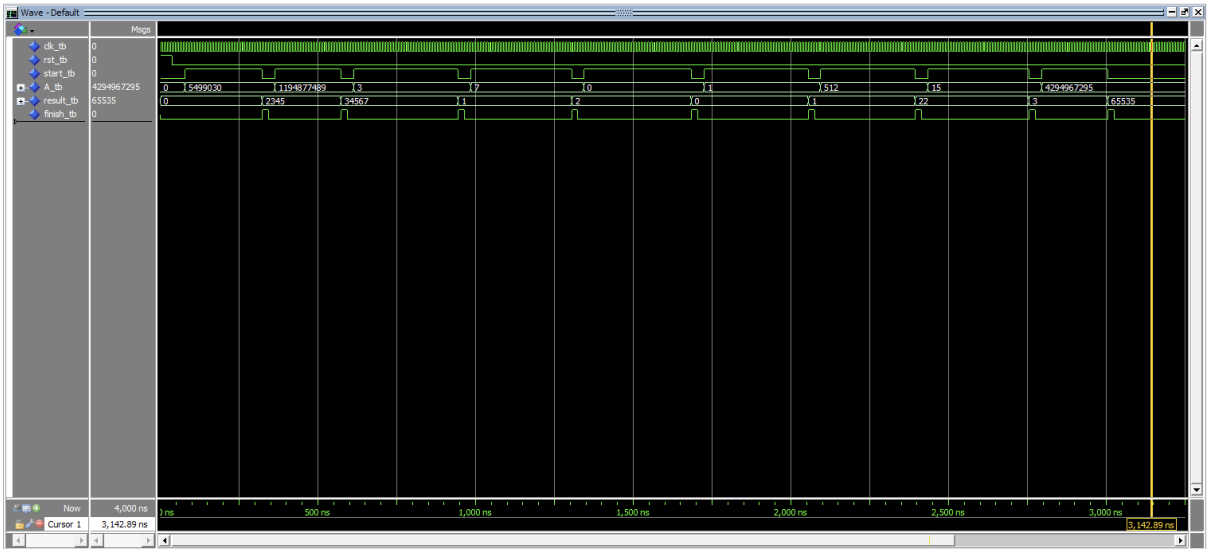
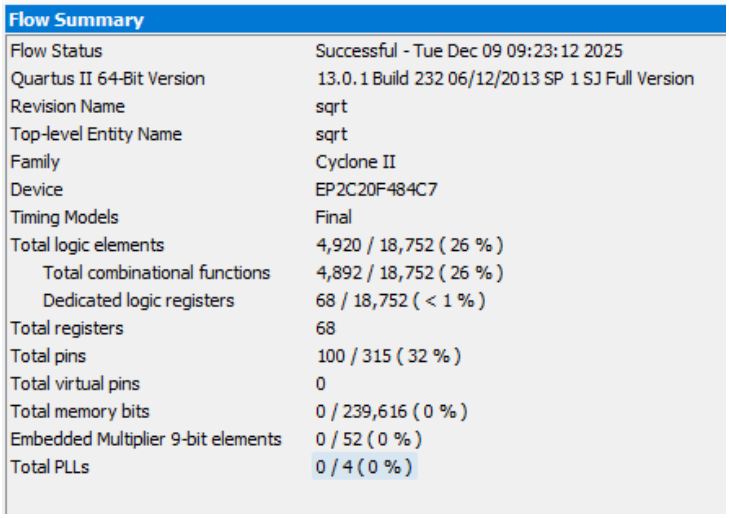


FIGURE 2 – Verification of Architecture 1

As we can see in Fig. 2, all of the results correspond with the values in Table 1.



Flow Summary	
Flow Status	Successful - Tue Dec 09 09:23:12 2025
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version
Revision Name	sqrt
Top-level Entity Name	sqrt
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Total logic elements	4,920 / 18,752 (26 %)
Total combinational functions	4,892 / 18,752 (26 %)
Dedicated logic registers	68 / 18,752 (< 1 %)
Total registers	68
Total pins	100 / 315 (32 %)
Total virtual pins	0
Total memory bits	0 / 239,616 (0 %)
Embedded Multiplier 9-bit elements	0 / 52 (0 %)
Total PLLs	0 / 4 (0 %)

FIGURE 3 – Resources Used for Architecture 1 (Newton Method)

Compilation Report - sqrt

Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	2.42 MHz	2.42 MHz	clk	

FIGURE 4 – Maximum Working Frequency of Architecture 1 (Newton Method)

The number of clock cycles required to perform the square root computation varies with the input "A".

2 Architecture 2

The algorithm has been implemented based on Figure 2 in the project description. For better optimization, instead of using multiplication and division, I used left and right shifts.

```
if R_temp >= 0 then
    R_temp := shift_left(R_temp, 2) + signed(resize(shift_right(D_temp, 2*n-2), R_temp'
        length)) - signed(resize((shift_left(Z_temp, 2) + 1), R_temp'length));
else
    R_temp := shift_left(R_temp, 2) + signed(resize(shift_right(D_temp, 2*n-2), R_temp'
        length)) + signed(resize((shift_left(Z_temp, 2) + 3), R_temp'length));
end if;

if R_temp >= 0 then
    Z_temp := shift_left(Z_temp, 1) + 1;
else
    Z_temp := shift_left(Z_temp, 1);
end if;
D_temp := shift_left(D_temp, 2);
```

Snippet 3 – VHDL Code of Architecture 2

The following state machine has been used for the behavioral implementation of Architecture 2 :

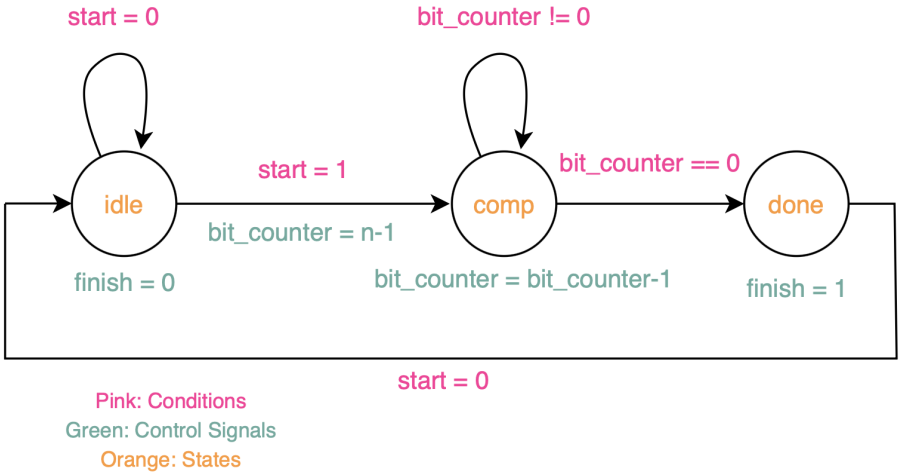


FIGURE 5 – State Machine of Architecture 2

As shown in Fig. 5, when the start signal is zero, the system stays in the idle state. The algorithm used for Architecture 2 computes the bits of the result sequentially. Thus, when the start signal becomes one, a "bit counter" is initialized to "n-1" and is decremented in each iteration. When all bits of the result have been calculated, the "bit counter" will equal zero, and the state changes to "done".

2.1 Results

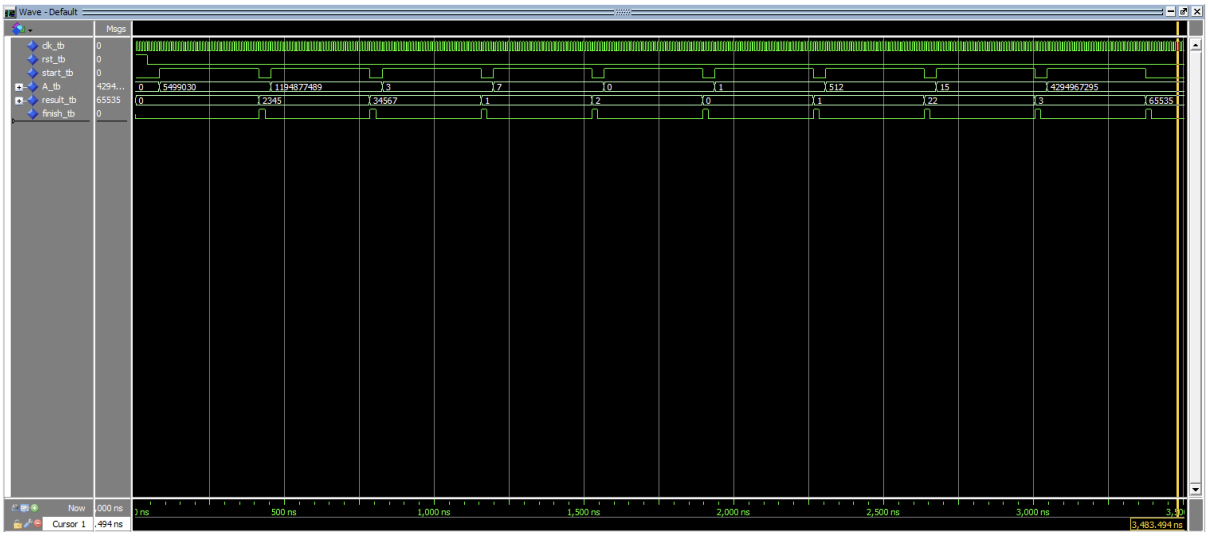


FIGURE 6 – Verification of Architecture 2

As we can see in Fig. 6, all results correspond with the values in Table 1.

Flow Summary	
Flow Status	Successful - Tue Dec 09 09:38:45 2025
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version
Revision Name	sqrt
Top-level Entity Name	sqrt
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Total logic elements	255 / 18,752 (1 %)
Total combinational functions	223 / 18,752 (1 %)
Dedicated logic registers	171 / 18,752 (< 1 %)
Total registers	171
Total pins	100 / 315 (32 %)
Total virtual pins	0
Total memory bits	0 / 239,616 (0 %)
Embedded Multiplier 9-bit elements	0 / 52 (0 %)
Total PLLs	0 / 4 (0 %)

FIGURE 7 – Resources Used for Architecture 2

Slow Model Fmax Summary				
	Fmax	Restricted Fmax	Clock Name	Note
1	163.19 MHz	163.19 MHz	clk	

FIGURE 8 – Maximum Working Frequency of Architecture 2

The number of clock cycles to perform the square root computation is equal for all inputs. Specifically, 32 (n) clock cycles are needed for the calculation of the 32 bits of the result, plus one clock cycle for the "done" state.

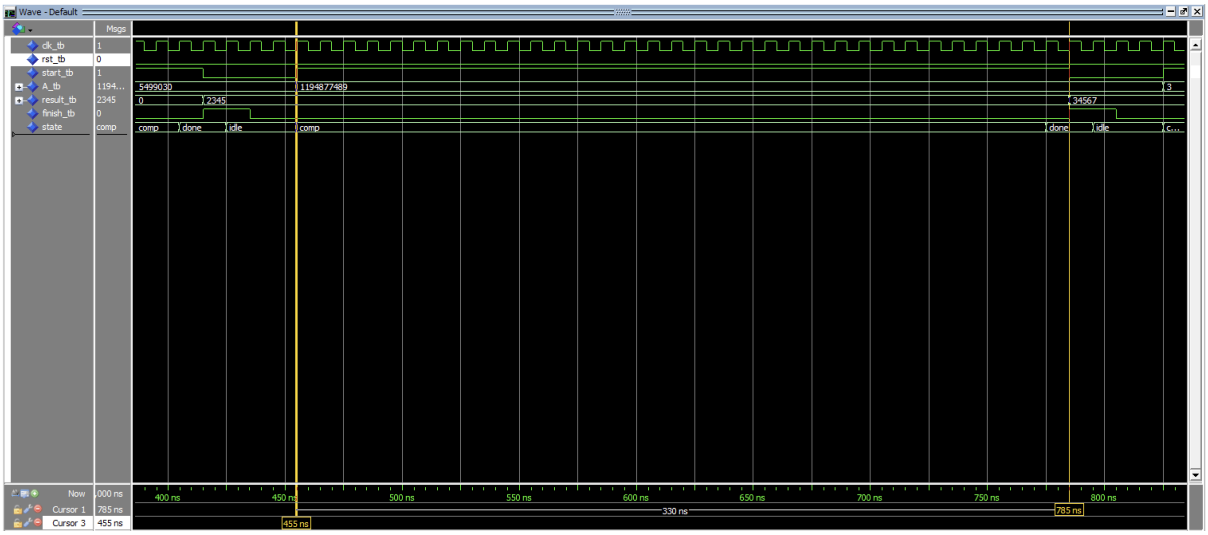


FIGURE 9 – Maximum Required Number of Clock Cycles for Computation

Based on Fig. 9, $\frac{330\ ns}{10\ ns\ (clock\ period)} = 33\ clock\ cycles$ are needed for each square root computation.

3 Architecture 3

In this part, a fully combinatorial circuit has been implemented, featuring an input "A" and an output "Result". The following loop is used to calculate the square root instantly.

```
for i in n-1 downto 0 loop
    if R_temp >= 0 then
        R_temp := shift_left(R_temp, 2) + signed(resize(shift_right(D_temp, 2*n-2),
            R_temp'length))- signed(resize((shift_left(Z_temp, 2) + 1), R_temp'length));
    else
        R_temp := shift_left(R_temp, 2) + signed(resize(shift_right(D_temp, 2*n-2),
            R_temp'length))+ signed(resize((shift_left(Z_temp, 2) + 3), R_temp'length));
    end if;
    if R_temp >= 0 then
        Z_temp := shift_left(Z_temp, 1) + 1;
    else
        Z_temp := shift_left(Z_temp, 1);
    end if;
    D_temp := shift_left(D_temp, 2);
end loop;

result <= std_logic_vector(Z_temp);
```

Snippet 4 – VHDL Code of Architecture 3

3.1 Results

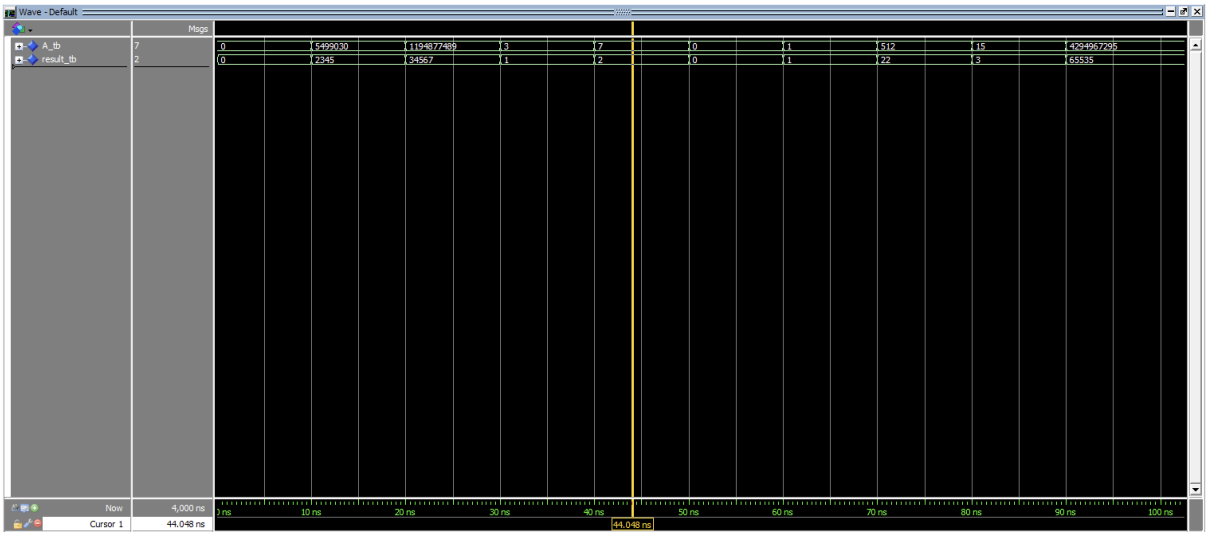


FIGURE 10 – Verification of Architecture 3

As we can see in Fig. 10, all results correspond with the values in Table 1. All results are ready instantly after changing the "A" value.

Flow Summary

Flow Status	Successful - Tue Dec 09 09:52:03 2025
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version
Revision Name	sqrt
Top-level Entity Name	sqrt
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Total logic elements	2,675 / 18,752 (14 %)
Total combinational functions	2,675 / 18,752 (14 %)
Dedicated logic registers	0 / 18,752 (0 %)
Total registers	0
Total pins	96 / 315 (30 %)
Total virtual pins	0
Total memory bits	0 / 239,616 (0 %)
Embedded Multiplier 9-bit elements	0 / 52 (0 %)
Total PLLs	0 / 4 (0 %)

FIGURE 11 – Resources Used for Architecture 3

Because this architecture is combinatorial, I placed it between two registers to determine its maximum working frequency :

Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	6.19 MHz	6.19 MHz	clk	

FIGURE 12 – Maximum Working Frequency of Architecture 3

Since the architecture is fully combinatorial and results change with the inputs, a "maximum required number of clock cycles" metric is not applicable.

4 Architecture 4

To convert the previous architecture to a pipeline, we use arrays of registers for R, D, Z, and start :

```
type R_array is array (n downto 0) of signed(n+1 downto 0);
type D_array is array (n downto 0) of unsigned(2*n-1 downto 0);
type Z_array is array (n downto 0) of unsigned(n-1 downto 0);
type start_array is array (n downto 0) of std_logic;
signal R_pipe : R_array;
signal D_pipe : D_array;
signal Z_pipe : Z_array;
signal start_pipe : start_array;
```

Snippet 5 – VHDL Code of Architecture 4

We then apply the same algorithm used in Architecture 3 while passing R, D, Z, and start through the pipeline. This effectively unrolls the loop into 32 distinct physical stages. Data flows sequentially from stage n down to 0, where the "start_pipe" shift register tracks valid data to assert the "finish" signal exactly when the final result exits the pipeline.

```
D_pipe(n) <= unsigned(A);
R_pipe(n) <= (others => '0');
Z_pipe(n) <= (others => '0');
start_pipe(n) <= start;
for i in n-1 downto 0 loop

    R_temp := R_pipe(i+1);
    D_temp := D_pipe(i+1);
    Z_temp := Z_pipe(i+1);
    if R_temp >= 0 then
        R_temp := shift_left(R_temp, 2) + signed(resize(shift_right(D_temp, 2*n-2),
            R_temp'length))- signed(resize((shift_left(Z_temp, 2) + 1), R_temp'length));
    else
        R_temp := shift_left(R_temp, 2) + signed(resize(shift_right(D_temp, 2*n-2),
            R_temp'length))+ signed(resize((shift_left(Z_temp, 2) + 3), R_temp'length));
    end if;
    if R_temp >= 0 then
        Z_temp := shift_left(Z_temp, 1) + 1;
    else
        Z_temp := shift_left(Z_temp, 1);
    end if;
    D_temp := shift_left(D_temp, 2);
    R_pipe(i) <= R_temp;
    D_pipe(i) <= D_temp;
    Z_pipe(i) <= Z_temp;

    start_pipe(i) <= start_pipe(i+1);
end loop;
result <= std_logic_vector(Z_pipe(0));
finish <= start_pipe(0);
```

Snippet 6 – VHDL Code of Architecture 4

4.1 Results

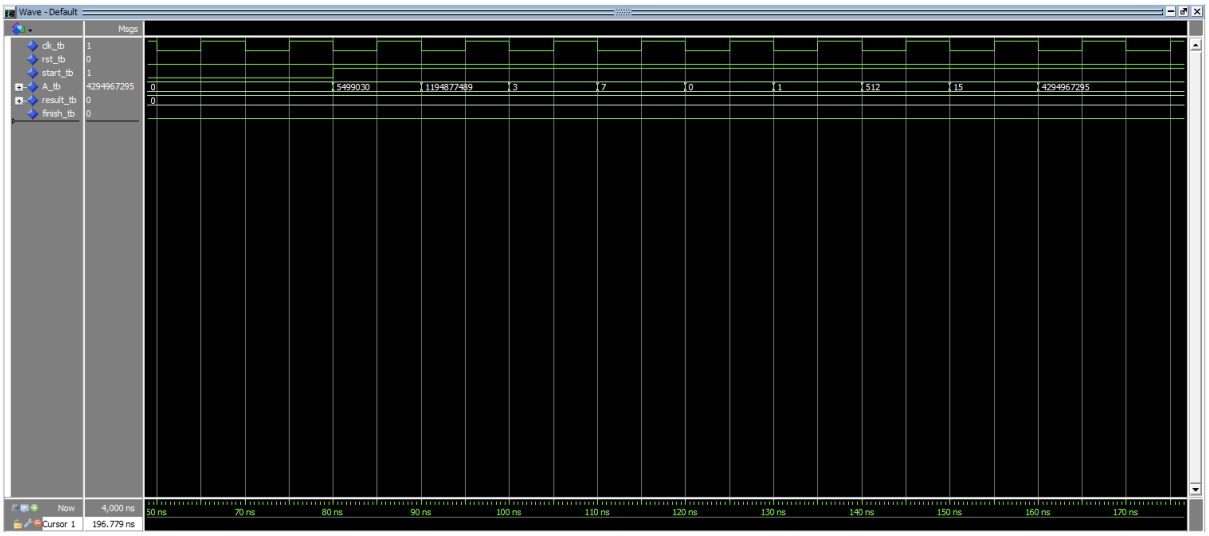


FIGURE 13 – Verification of Architecture 4 (Inputs)

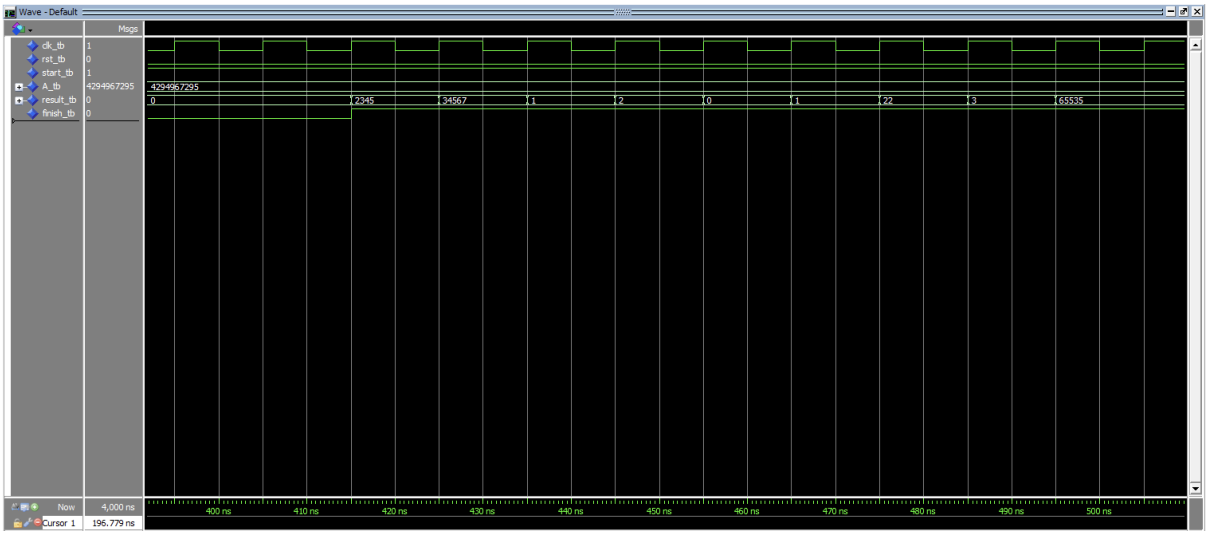


FIGURE 14 – Verification of Architecture 4 (Outputs)

As we can see in Fig. 14, all results correspond with the values in Table 1.

Flow Summary	
Flow Status	Successful - Tue Dec 09 09:55:47 2025
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version
Revision Name	sqrt
Top-level Entity Name	sqrt
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Total logic elements	3,352 / 18,752 (18 %)
Total combinational functions	3,024 / 18,752 (16 %)
Dedicated logic registers	1,957 / 18,752 (10 %)
Total registers	1957
Total pins	100 / 315 (32 %)
Total virtual pins	0
Total memory bits	688 / 239,616 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 52 (0 %)
Total PLLs	0 / 4 (0 %)

FIGURE 15 – Resources Used for Architecture 4

Slow Model Fmax Summary				
	Fmax	Restricted Fmax	Clock Name	Note
1	132.94 MHz	132.94 MHz	clk	

FIGURE 16 – Maximum Working Frequency of Architecture 4

The number of clock cycles required to perform the square root computation is equal for all inputs. 33 clock cycles are needed to calculate the 32 bits of the result and present them at the component output.

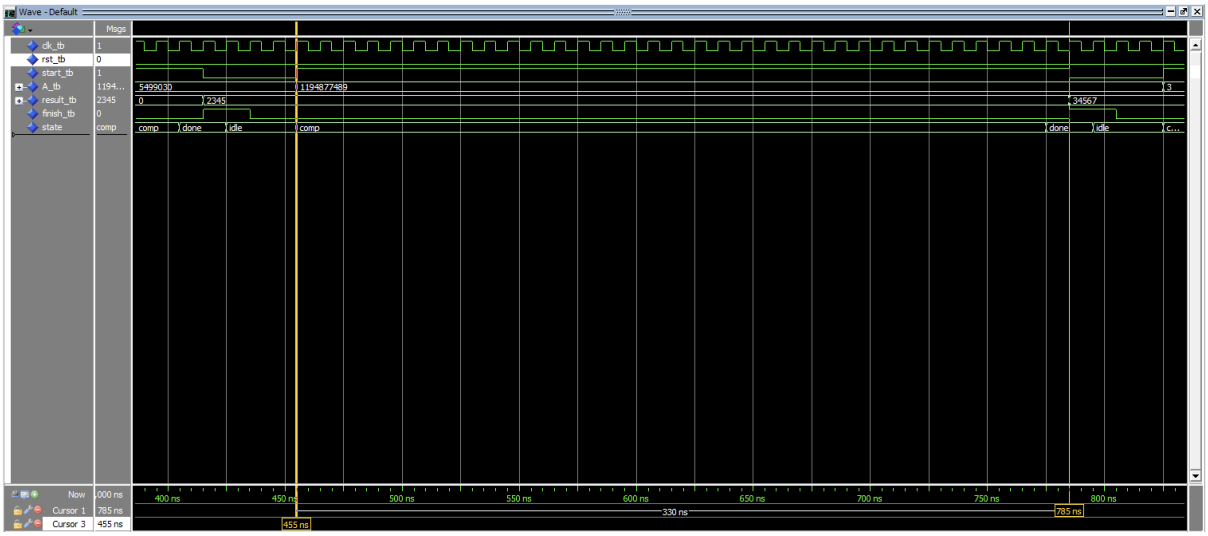


FIGURE 17 – Maximum Required Number of Clock Cycles for Computation

Based on Fig. 17, $\frac{330\ ns}{10\ ns\ (clock\ period)} = 33\ clock\ cycles$ are needed for each square root computation.

5 Architecture 5

For Architecture 5, we use a structural approach, which involves designing distinct datapath and controller components. For the datapath, I utilized the design described in [1] with minor modifications to parameterize the architecture. We require two shift registers for "D" and "Q" (the name used in paper

[1] instead of "Z") and a register for "R". Additionally, an adder/subtractor is needed to perform the required operations in each clock cycle :

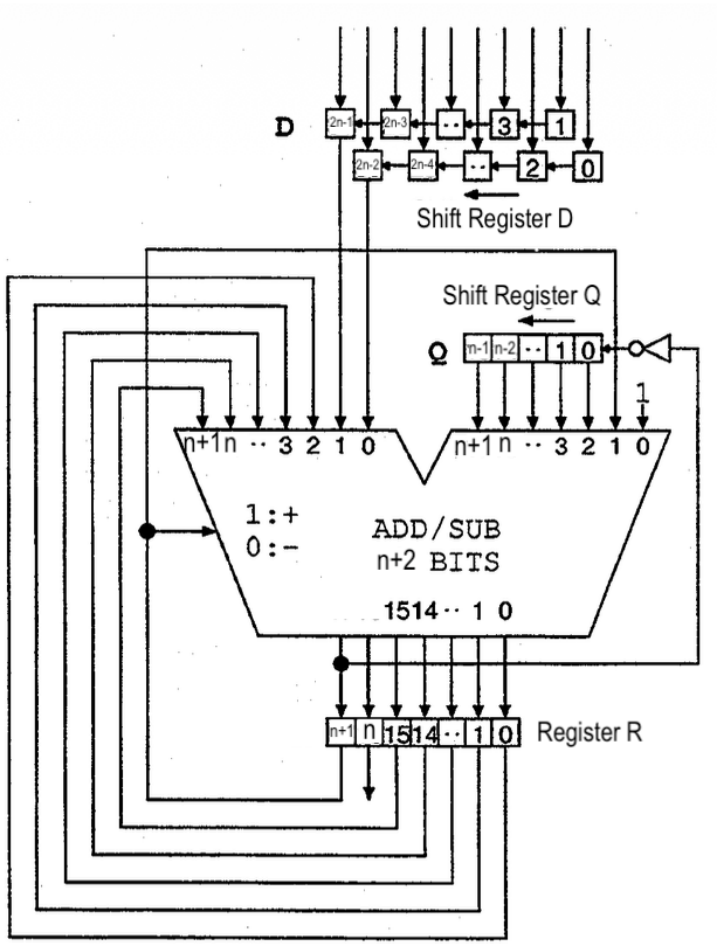


FIGURE 18 – Datapath of Architecture 5 (Image from [1] with Minor Modifications)

5.1 Adder/Subtractor

This component features a control signal "add_sub" and performs addition when this signal is one :

```
process(a, b, add_sub)
begin
    if add_sub = '1' then
        result <= std_logic_vector(signed(a) + signed(b));
    else
        result <= std_logic_vector(signed(a) - signed(b));
    end if;
end process;
```

Snippet 7 – VHDL Code of Adder/Subtractor

5.2 Shift Register D

When the "enable" signal is one, the register shifts the data to the left by 2 and places the two most significant bits on the "top_2" output.

```
process(clk, reset)
begin
    if reset = '1' then
        reg <= (others => '0');
    elsif rising_edge(clk) then
        if load = '1' then
            reg <= unsigned(data_in);
        elsif enable = '1' then
            reg <= shift_left(reg, 2);
        end if;
    end if;
end process;

top_2 <= std_logic_vector(reg(2*n-1 downto 2*n-2));
```

Snippet 8 – VHDL Code of Register D

5.3 Shift Register Q

When the "enable" signal is one, the register shifts the data to the left by 1 and places "bit_in" into the least significant position.

```
process(clk, reset)
begin
    if reset = '1' then
        reg <= (others => '0');
    elsif rising_edge(clk) then
        if enable = '1' then
            reg <= reg(n-2 downto 0) & bit_in;
        end if;
    end if;
end process;
data_out <= std_logic_vector(reg);
```

Snippet 9 – VHDL Code of Register Q

5.4 Register R

This simple register is placed at the output of the Add/Sub component (Width $= n + 2$).

5.5 Controller

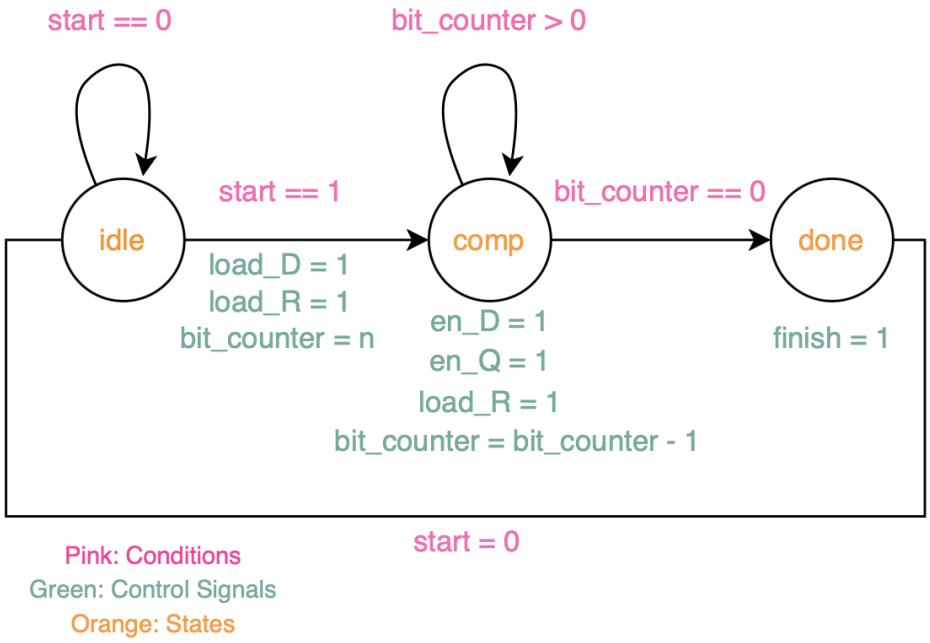


FIGURE 19 – Controller of Architecture 5

As shown in Fig. 19, when the start signal is zero, the system stays in the idle state. When start is one, the "D" and "R" registers load the data, and a "bit_counter" is initialized to "n" and decremented in each iteration. Furthermore, in each iteration, the D and Q shift registers shift the data (en=1) and register "R" loads the data. Once all bits of the result are calculated, the "bit_counter" will equal zero, and the state changes to "done".

5.6 Results

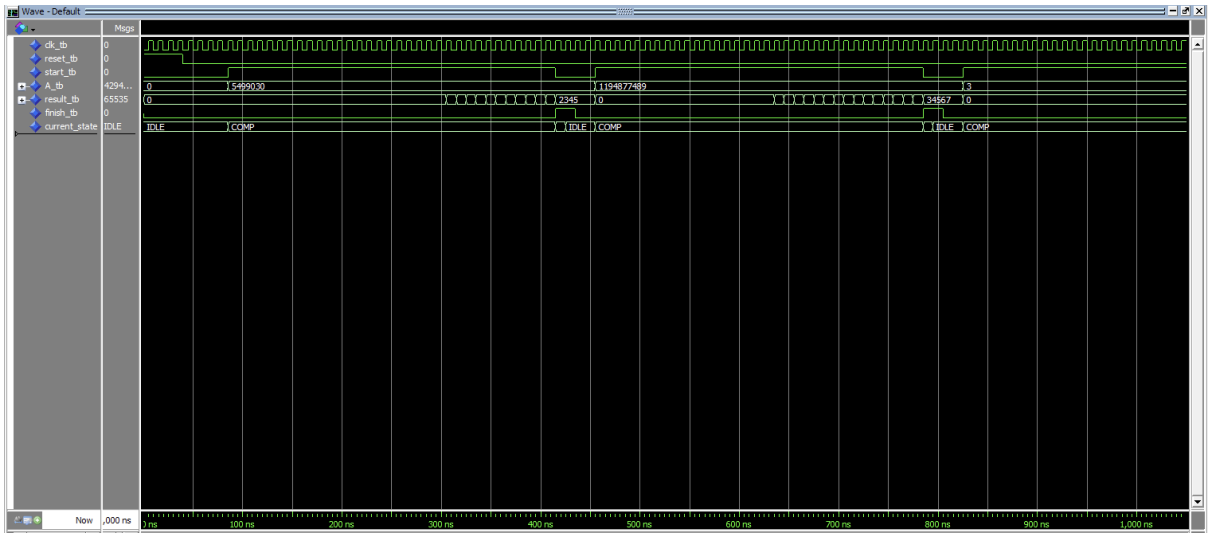


FIGURE 20 – Verification of Architecture 5

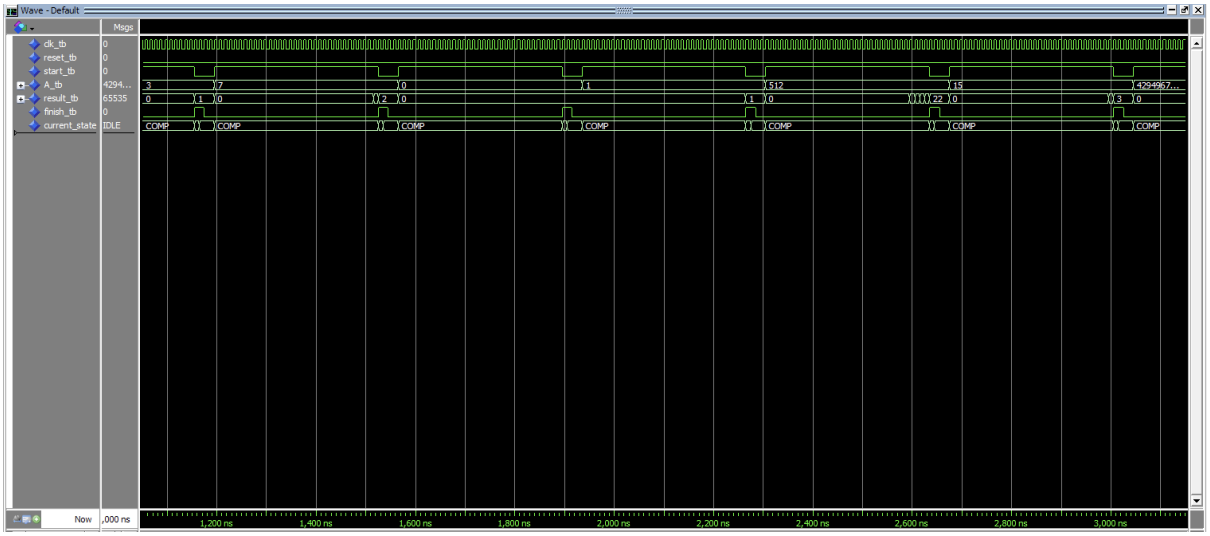


FIGURE 21 – Verification of Architecture 5

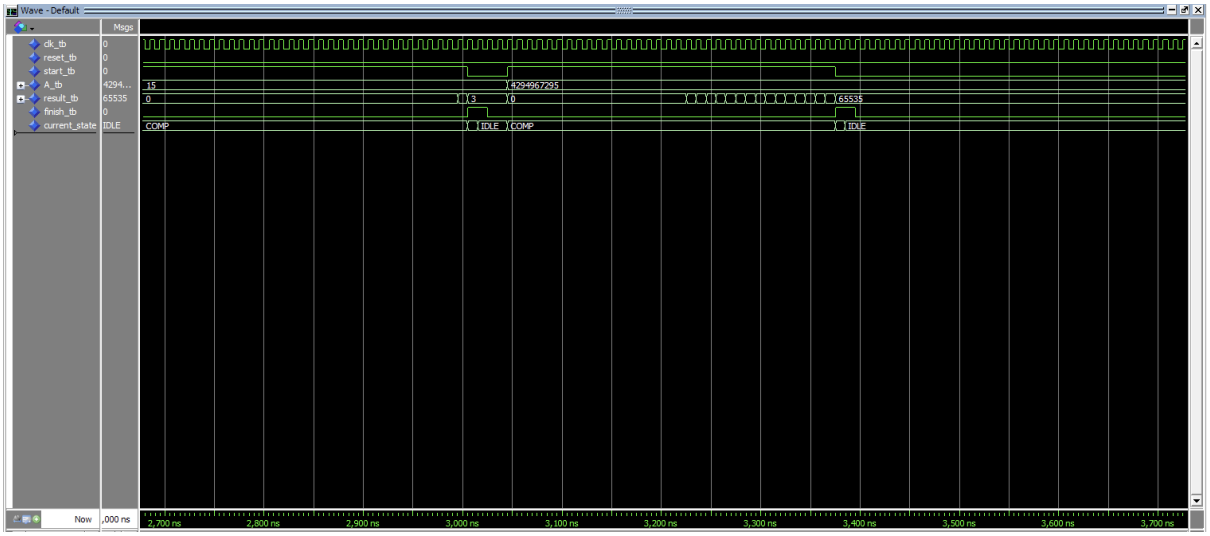


FIGURE 22 – Verification of Architecture 5

As we can see in Fig. 20, 21, and 22, all results correspond with the values in Table 1.

Flow Summary	
Flow Status	Successful - Tue Dec 09 10:04:39 2025
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version
Revision Name	sqrt
Top-level Entity Name	sqrt
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Total logic elements	210 / 18,752 (1 %)
Total combinational functions	192 / 18,752 (1 %)
Dedicated logic registers	142 / 18,752 (< 1 %)
Total registers	142
Total pins	100 / 315 (32 %)
Total virtual pins	0
Total memory bits	0 / 239,616 (0 %)
Embedded Multiplier 9-bit elements	0 / 52 (0 %)
Total PLLs	0 / 4 (0 %)

FIGURE 23 – Resources Used for Architecture 5

Slow Model Fmax Summary				
	Fmax	Restricted Fmax	Clock Name	Note
1	160.0 MHz	160.0 MHz	clk	

FIGURE 24 – Maximum Working Frequency of Architecture 5

The number of clock cycles to perform the square root computation is equal for all inputs. Specifically, 33 $(n+1)$ clock cycles are needed for the calculation.

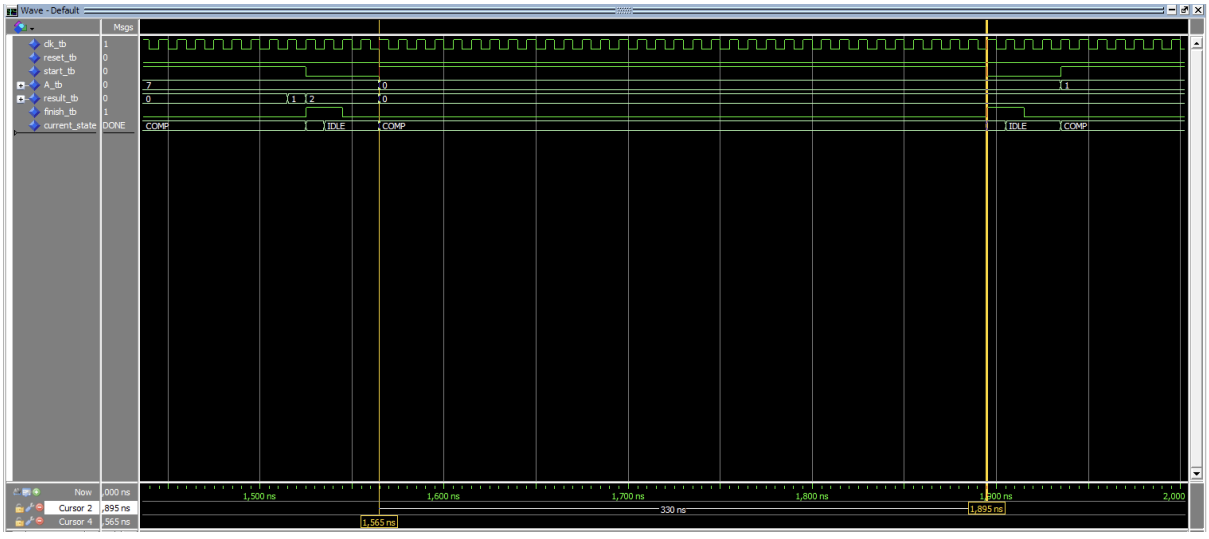


FIGURE 25 – Maximum Required Number of Clock Cycles for Computation

Based on Fig. 25, $\frac{330\ ns}{10\ ns\ (clock\ period)} = 33\ clock\ cycles$ are needed for each square root computation.

6 Results Comparison

The five square root architectures are compared in terms of hardware area, maximum operating frequency, latency, and efficiency, based on the results presented in Tables 2, 3, and 4.

Architecture 1 (Newton Method) consumes the largest amount of hardware resources and achieves the lowest maximum operating frequency, as shown in Table 2. This is primarily due to the use of division

operations, which are costly in FPGA implementations. Additionally, its computation latency depends on the input value.

Architectures 2 and 5 are the most area-efficient solutions, each utilizing approximately 1% of the FPGA logic resources, as reported in Table 2. By relying on shift and add/subtract operations, these architectures achieve high operating frequencies and a fixed, predictable latency of 33 clock cycles.

Architecture 3 computes the square root instantaneously using a fully combinational approach. However, as indicated in Table 2 and Table 3, this architecture suffers from a low maximum operating frequency and relatively high combinational complexity.

Architecture 4 introduces pipelining to improve throughput. Although this approach increases the number of registers and overall area usage, as shown in Table 3, it supports a high operating frequency and is capable of accepting one new input per clock cycle after the pipeline is filled.

Finally, the efficiency metric ($F_{\max}$ divided by total logic elements) presented in Table 4 shows that Architecture 5 achieves the best performance in terms of $F_{\max}$ per logic element, followed closely by Architecture 2. Architectures 1 and 3 exhibit the lowest efficiency values.

Overall, Architecture 5 provides the best trade-off between area, speed, and deterministic latency, making it the most suitable choice for FPGA-based square-root computation.

Table 2 – Comparison of Square Root Computing Architectures on FPGA

Architecture	**Total Logic Elements**	**Fmax**	**Clock Cycles per Computation**
Architecture 1	4,920 (26%)	2.42 MHz	Varies with input A
Architecture 2	255 (1%)	163.19 MHz	33 cycles
Architecture 3	2,675 (14%)	6.19 MHz*	Instant (Combinatorial)
Architecture 4	3,352 (18%)	132.94 MHz	33 cycles
Architecture 5	210 (1%)	160.0 MHz	33 cycles

*Fmax for Architecture 3 was determined by placing the combinatorial logic between two registers.

Table 3 – Resource Utilization Comparison for Square Root Architectures

Architecture	**Total Combinational Functions**	**Dedicated Logic Registers**
Architecture 1	4,892	68
Architecture 2	223	171
Architecture 3	2,675	0
Architecture 4	3,024	1,957
Architecture 5	192	142

Table 4 – Efficiency Metric Comparison (Fmax per Logic Element)

Architecture	**Efficiency (MHz / LE)**
Architecture 1	0.0005
Architecture 2	0.6400
Architecture 3	0.0023
Architecture 4	0.0397
Architecture 5	0.7619