

Figure 1: Architecture 1 - Testing  $\text{sqrt}(0)$  and  $\text{sqrt}(1)$

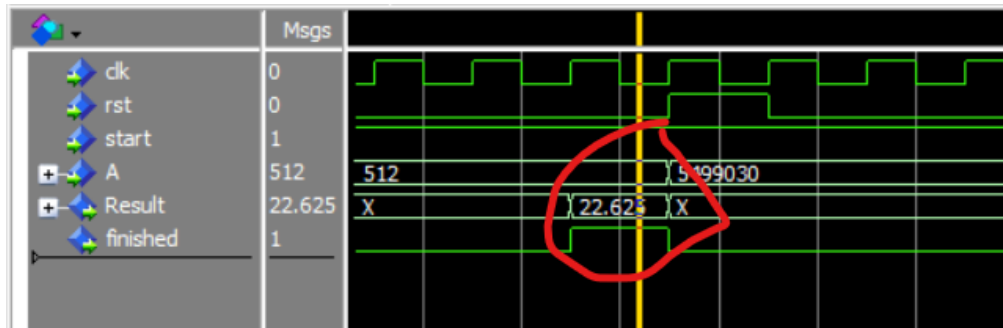


Figure 2: Architecture 1 - Testing  $\sqrt{512}$

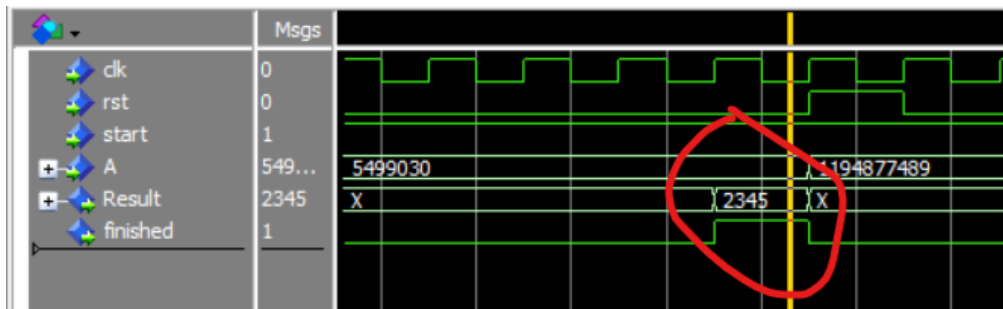


Figure 3: Architecture 1 - Testing  $\sqrt{5499030}$

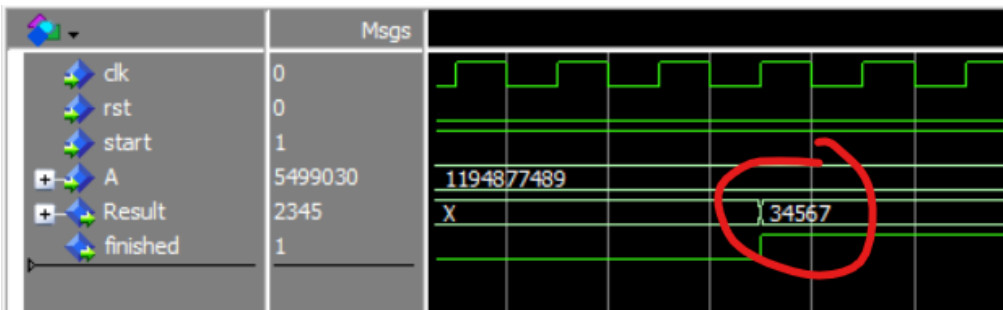


Figure 4: Architecture 1 - Testing  $\sqrt{1194877489}$

As shown in the figures above, a testbench was used to test the square root of the following values: 0, 1, 512, 5499030, and 1194877489.

The results were very accurate and most of the values did not need to complete the whole loop. However, as we mentioned earlier, this method would result in a very complex hardware with a huge number of gates due to the presence of multipliers and dividers.

## Architecture 2

For this architecture, we used another algorithm that was introduced in the *ICCD'96* by Yamin Li and Wanming Chu [1]. It is also an iterative algorithm that computes the bits of the result sequentially. It starts with the MSB, based on a modified non-restoring integer square root computing algorithm [1].

The following is the new non-restoring square root algorithm written in the C language.

```
unsigned squart(D, r) /*Non-Restoring sqrt*/
unsigned D; /*D:32-bit unsigned integer to be square rooted */
int *r;
{
    unsigned Q=0; /*Q:16-bit unsigned integer (root)*/
    int R=0; /*R:17-bit integer (remainder)*/
    int i;
    for (i=15;i>=0;i--) { /*for each root bit*/
        if (R>=0) { /*new remainder:*/
            R=(R<<2)|((D>>(i+1))&3);
            R=R-((Q<<2)|1); /*-Q01*/
        } else { /*new remainder:*/
            R=(R<<2)|((D>>(i+1))&3);
            R=R+((Q<<2)|3); /*+Q11*/
        }

        if (R>=0) Q=(Q<<1)|1; /*new Q:*/
        else Q=(Q<<1)|0; /*new Q:*/
    }

    /*remainder adjusting*/
    if (R<0) R= R+((Q<<1)|1);
    *r=R; /*return remainder*/
    return(Q); /*return root*/
}
```

## Architecture 2 results

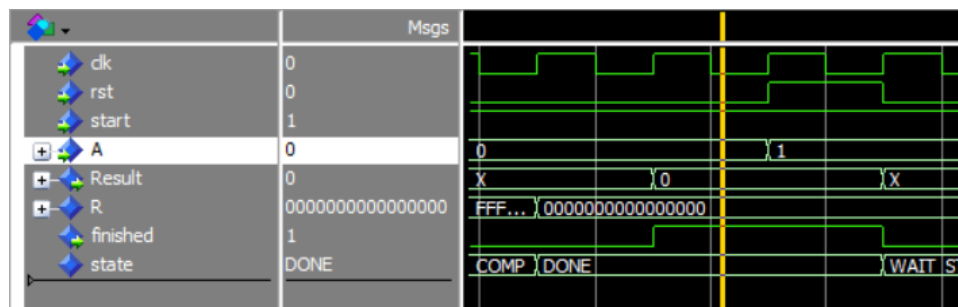


Figure 5: Architecture 2 - Testing sqrt(0)

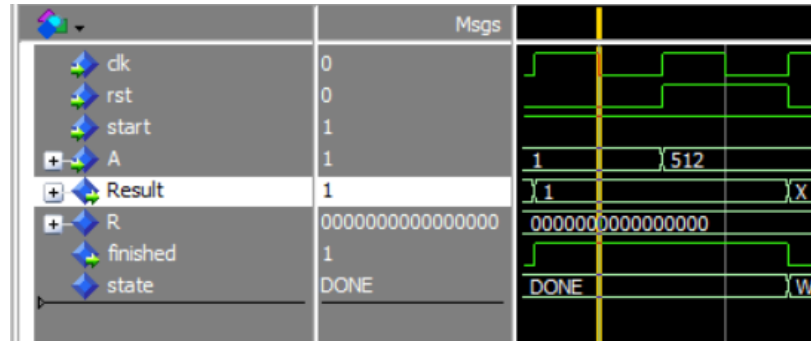


Figure 6: Architecture 2 - Testing  $\text{sqrt}(1)$

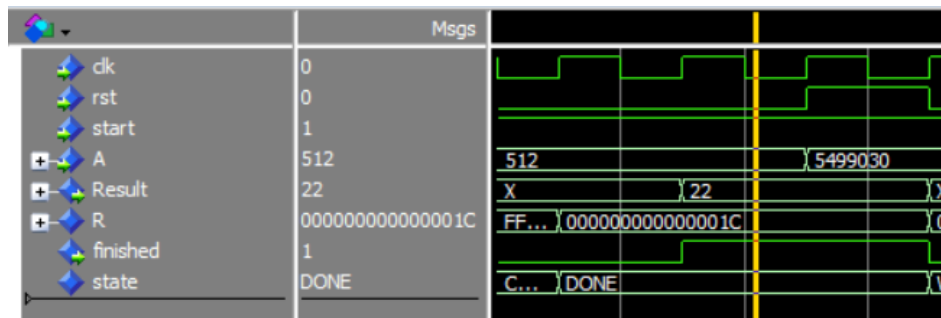


Figure 7: Architecture 2 - Testing  $\text{sqrt}(512)$

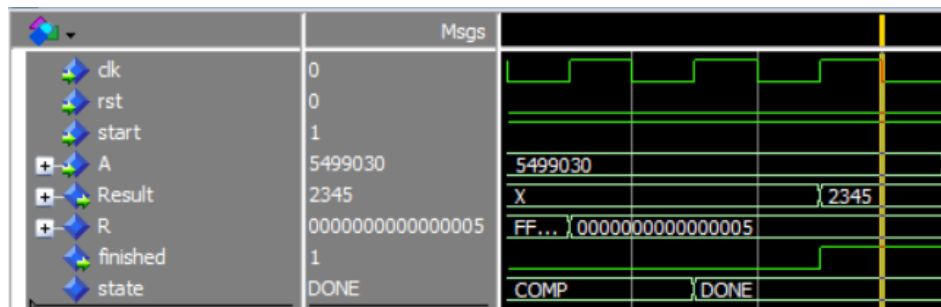


Figure 8: Architecture 2 - Testing  $\text{sqrt}(5499030)$

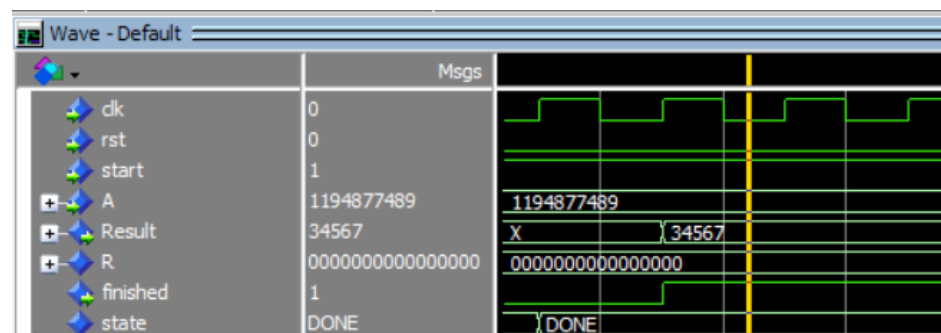


Figure 9: Architecture 2 - Testing  $\text{sqrt}(1194877489)$

As shown in the figures above, the result is correct for all the tests, and I also showed the remainder (R) as some results have a decimal values so the remainder counts for these values. This algorithm is much better than the first algorithm as it doesn't need any multipliers or dividers, it just needs some shifters and adders which have way less gates than the multiplier.

### Architecture 3

For this architecture, we used the same algorithm in the paper but with a different design to be able to make it a combinatorial circuit. The circuit is very simple as shown in fig. 10. It needs only shifters, registers, adders and subtractors. For the left side of the adder/subtractor, it simply consists of the remainder, and the most 2 significant bits in the input. For the right side, it consists of the output, MSB of the remainder and '1'. The input should be shifted to the left by 2 bits each iteration. For the remainder, it will be formed by adding or subtracting the left and right sides based on the MSB of the remainder of the previous iteration. Finally, the output is formed by shifting the previous output to the left and concatenating the MSB of the remainder with it.

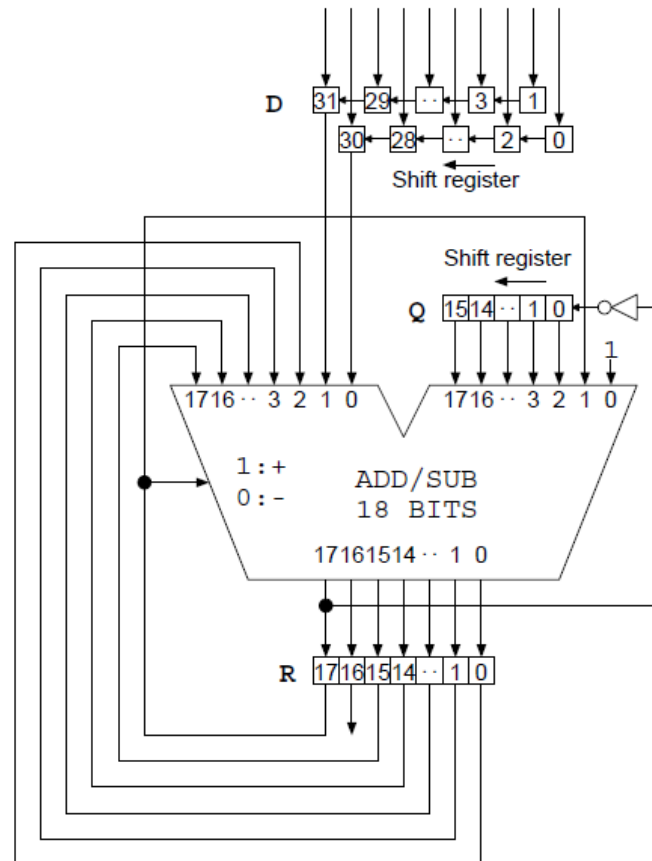
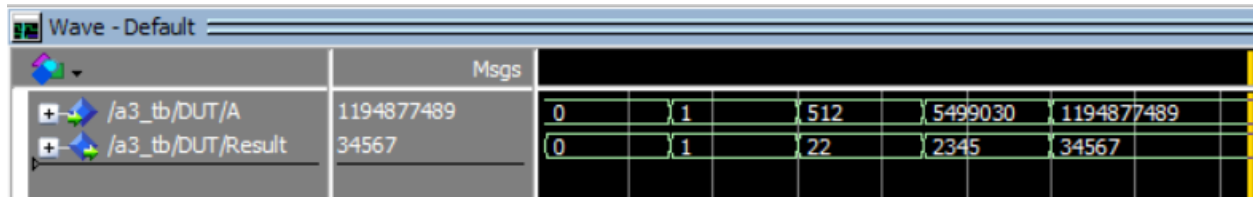


Figure 10: Architecture 3

## Architecture 3 results



Wave - Default		Msgs					
+ /a3_tb/DUT/A	1194877489	0	1	512	5499030	1194877489	
+ /a3_tb/DUT/Result	34567	0	1	22	2345	34567	

Figure 11: Architecture 3 results

## Architecture 4

This architecture is based on the figure shown below which is taken from [1]. Each signal or variable are made as arrays to be able to hold the values when we enter different inputs each clock cycle. For this architecture we need around 32 adder/subtractor, one for each stage. We need also an array of registers for the output and reminders.

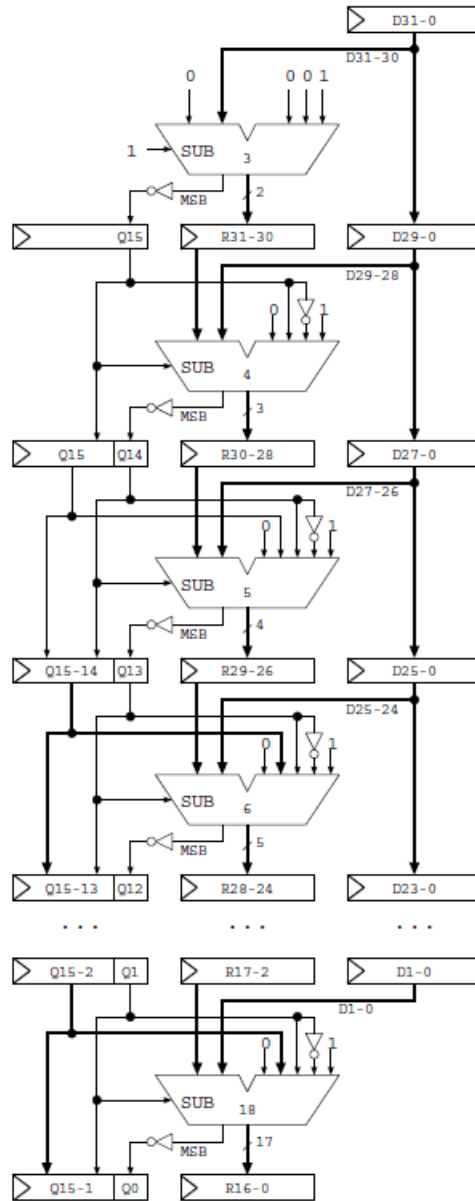


Figure 12: Architecture 4 FSM

## Architecture 4 results

As shown in the figure below, the inputs are inserted into the architecture one each clock cycle and after the 32 cycles are done, the result is shown and with each extra cycle after the 32 cycles a new output is calculated.

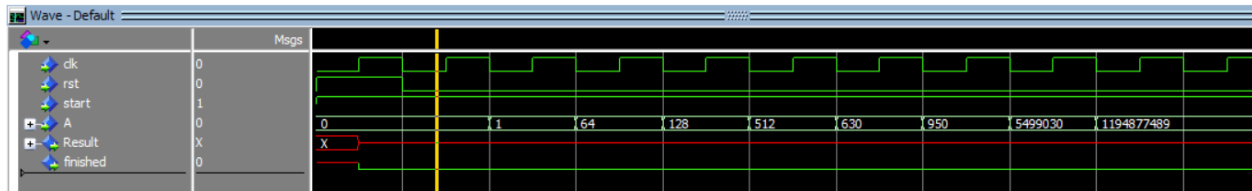


Figure 13: Inputs

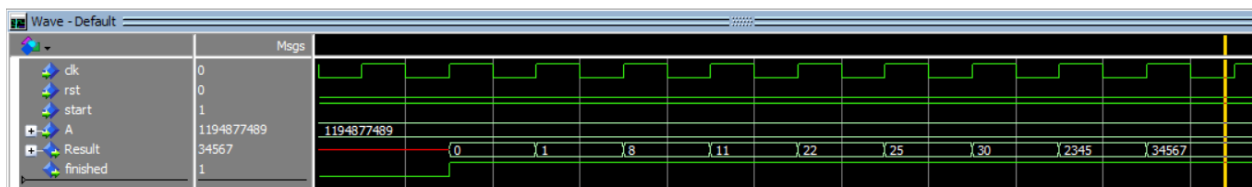


Figure 14: Pipeline Output



## Architecture 5

For the last architecture, a structural approach was used to better control the architecture of the circuit. For this architecture, the design shown in figure below was used. It only uses 2 shifters (1 for input shifting and another one for the output) and an 18-bits adder/subtractor. A design was made for the 2 shifters and the adder/subtractor and a FSM was used as a control unit to control the flow of the data throughout the whole process.

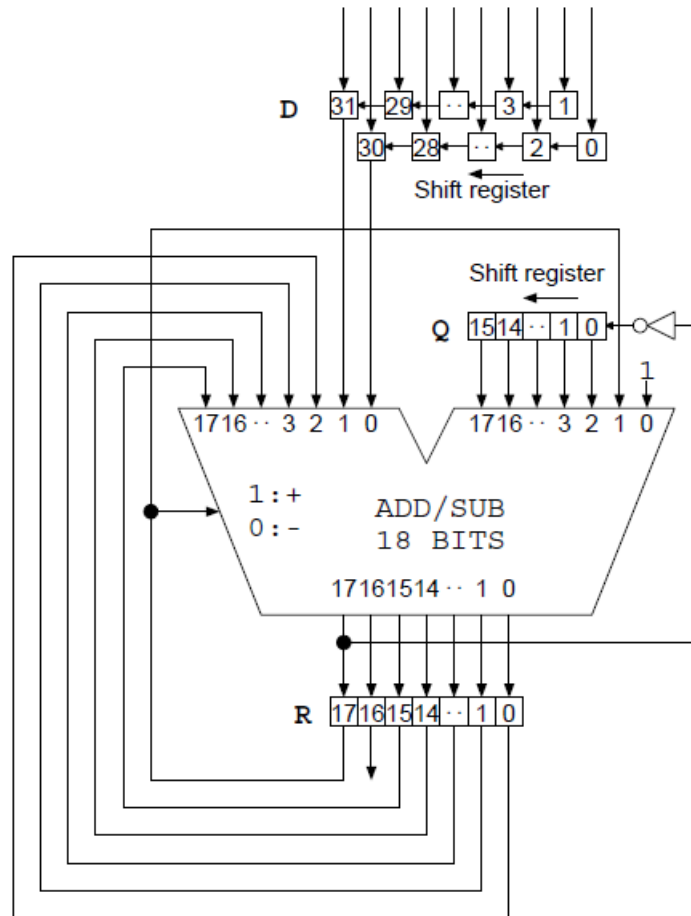


Figure 15: Low-cost version

The FSM used is shown in the figure below:

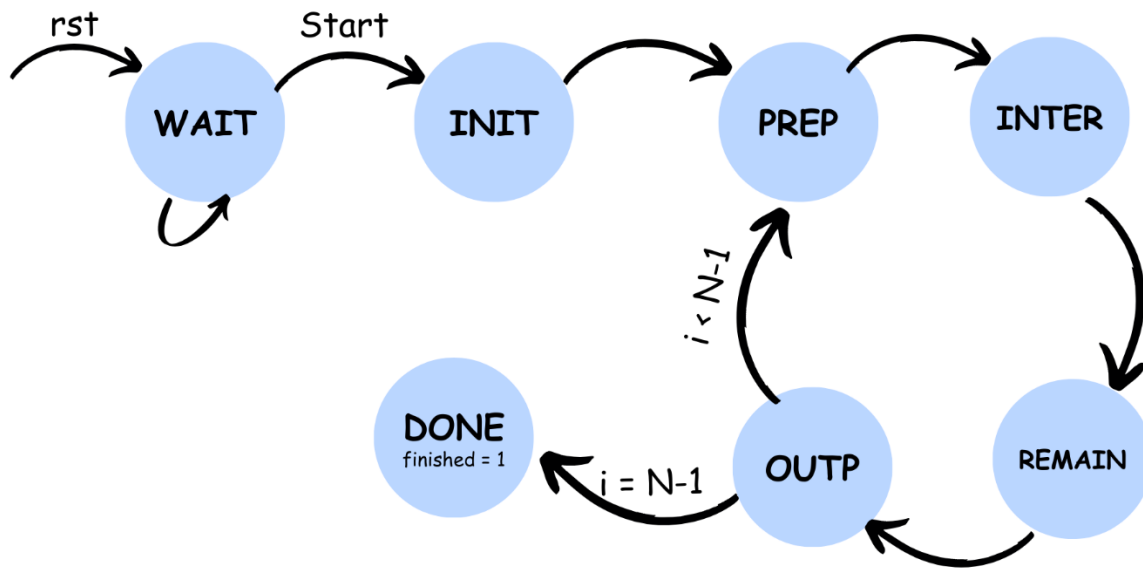


Figure 16: Architecture 5 FSM

Here's a brief about what each state does:

- 1) WAIT state: It is the first state when we reset the circuit, it checks on the *start* signal and goes to the next state when the start signal is high.
- 2) INIT state: In this state, we put the input in a variable to operate on and initialize a counter for the loop.
- 3) PREP state: In this state, the first shifter is enabled to prepare the input shifter for the next state, the right and left sides of the adder/subtractor are calculated.
- 4) INTER state: In this state, the input is shifted, and the adder enable is activated to prepare the adder/subtractor to work for the next state.
- 5) REMAIN state: In this state, the remainder is calculated by the adder/subtractor, and the output shifter is enabled to prepare it for the next state.
- 6) OUTP state: In this state, the output is shifted, and a check was done on *i* to know whether the loop is finished, or we should go back to the PREP state.
- 7) DONE state: This is the final state, in this state, we assign the output to the result signal, raise the *finished* signal, and stay there until the *start* signal is lowered or *reset* is done.

## Architecture 5 results

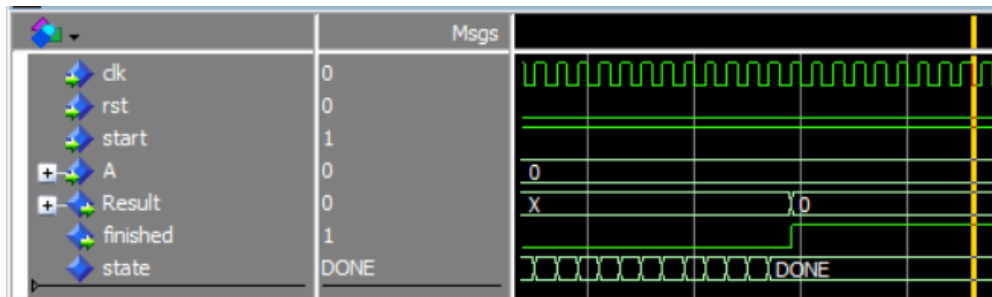


Figure 17: Architecture 4 - Testing  $\text{sqrt}(0)$

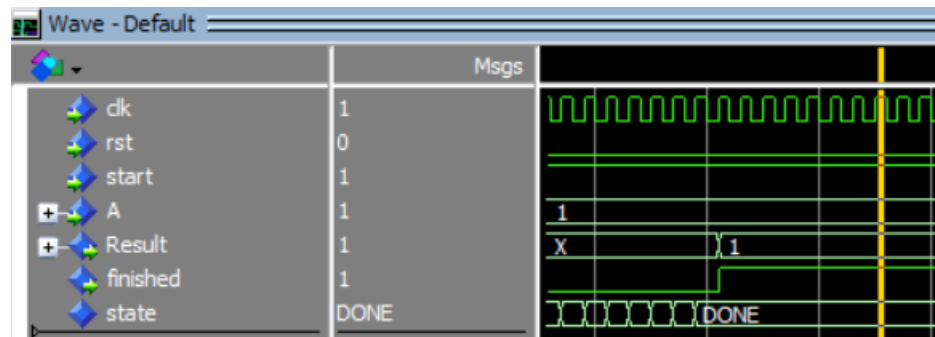


Figure 18: Architecture 4 - Testing  $\text{sqrt}(1)$

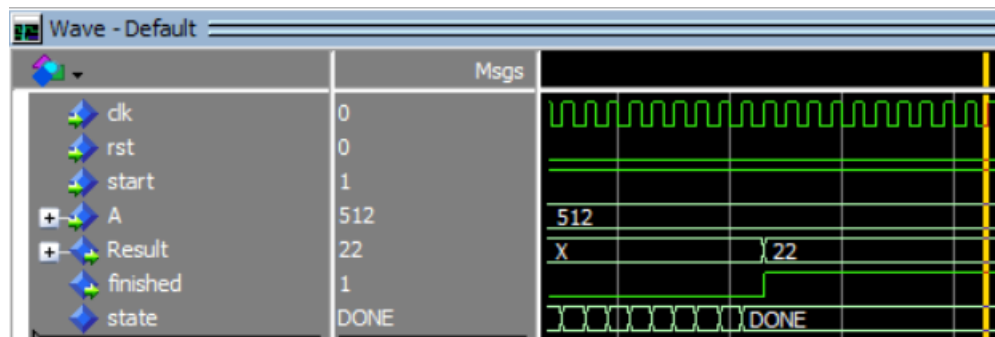


Figure 19: Architecture 4 - Testing  $\text{sqrt}(512)$

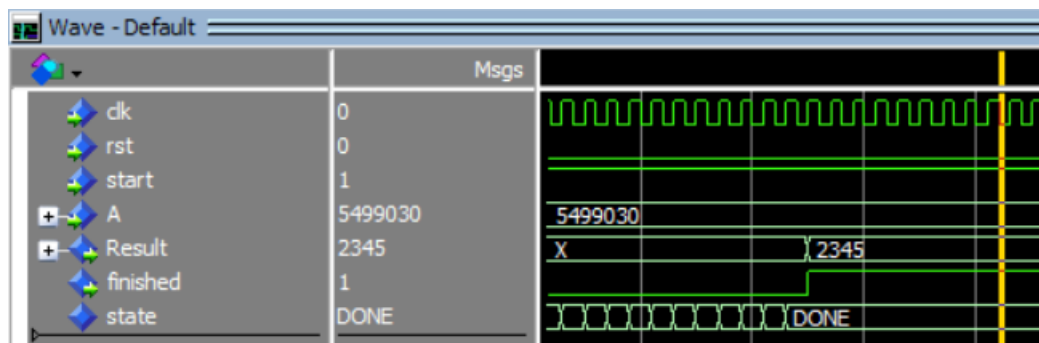


Figure 20: Architecture 4 - Testing  $\text{sqrt}(5499030)$

