SRT Simulator

Bikash Shrestha

March 2020

1. Abstract

This report describes the implementation of the SRT division method for binary floating-point numbers. SRT division algorithm is a very efficient method for dividing two binary floating-points. A simple simulator was created using a python programming language to simulate the operations that are performed in the SRT algorithm. The simulator takes two floating-point numbers, Dividend and Divisor, as Hexadecimal number, unless, otherwise they are defined as Binary. All the operations performed in the simulator are in the binary numbers. This simulator has different modules to perform different binary operations like module for finding 2's complement, Full Adder module, Carry Select Adder module, a module to check overflow, a module for shift (both right and left) operations and so on. The output of the simulator shows each step of the algorithm with print outs showing the content. Also, the execution time of the simulator is calculated in terms of Δt. There is a plot between the length of operand vs the execution time to show that an increase in length will increase the execution time of the division.

2. Introduction and Motivation

The most common implementation of binary division in the computer system is SRT division, taking its name from the initials of Sweeney, Robertson, and Tocher, who developed the algorithm independently at approximately the same time. SRT division uses subtraction as the fundamental operator to remove a fixed number of quotient bits in each iteration. SRT division is considered as a fast division method. SRT division is somewhat similar to the Restoring division method but with quite different unique features.

The reason behind doing this project is to understand how the basic binary operations are used efficiently to build a complex system. Since the SRT method is the most efficient division algorithm. It is interesting to see how the execution of the algorithm happens on the computer. It also provides a better and efficient way of testing and finding the results of binary division. It provides the flexibility of testing all sorts of binary numbers from smaller bits to larger bits and investigates the execution time with respect to the length of the numbers. So it can be an important application for analyzing this algorithm. It also provides a better way of learning and understanding different binary operations.

3. Methodology

This simulator was built using the python programming language. The project is divided into different modules and was combined together to simulate the SRT division algorithm. The simulator takes two inputs as a dividend and divisor which is in hexadecimal format unless it is specified as a binary number. These numbers are provided as a string to the program. The dividend length and divisor length should be within this group (8, 4), (12, 6), (16, 8), and (24, 12) bits. So if the inputs do not satisfy these criteria, then additional zero bits are added manually to make them as required by the program. Now it is difficult to perform binary operations in the string, so they are converted into arrays of ones and zeros to perform a binary operation on them. As per the algorithm, the given pseudocode was applied in this project:

- Get the inputs, dividend (AQ) and divisor (B)
- Check for overflow (if the higher part of the dividend is greater or equal to the divisor)
    - If there is an overflow, then right-shift the dividend until the higher part of it becomes less than the divisor.
- Normalize the B
- Take a 2's complement of the B
- Adjust the AQ
- Shift over zeros if there are zeros in the most significant bits else go to next step
- Subtract B from the higher-order part of AQ i.e A
- While (shiftCount != (length of divisor + 1)) do
    - Check the most significant bit of the result.
        - If there is 1 in the MSB which means it is a negative value, then in the next step
            - left shift AQ and insert 0 to the LSB of the AQ (i.e $Q_0 \leftarrow 0$)
            - Shift over ones if there are zeros in the most significant bits else go to next step
            - Add B to the higher-order part of AQ i.e A
        - Else if it is a positive value.
            - Left shift AQ and insert 1 to the LSB of the AQ (i.e $Q_0 \leftarrow 1$)
            - Shift over zeros if there are zeros in the most significant bits else go to next step
            - Subtract B from the higher-order part of AQ i.e A
- Quotient = Q part of AQ
- If in the last iteration there was a negative value, then
    - Right shift the higher-order part of AQ (i.e A)

    - Add B to A

- Remainder = Most significant Part of A except all the bits that were shifted after the first adjustment

Moreover, the execution time of each operation was noted and added to calculate the total execution time of the whole SRT division. The execution time is in terms of Δt. Shift operation is $3\Delta t$, and 2's complement operation is $n^* \Delta t$ (n is the operand length). We are using Carry Select for the addition of two binary numbers and the execution time of Carry Select adder was calculated using the following equation:

$$\text{Exec time} = 10\Delta t + ((n - 4)/4) * 2\Delta t$$

where n is the length of the total bits in the number

Each module will calculate it's own execution time and return it to the main program which is then added to find the total execution time. The figures given in the result and analysis section will show the execution time after each step while performing the division.

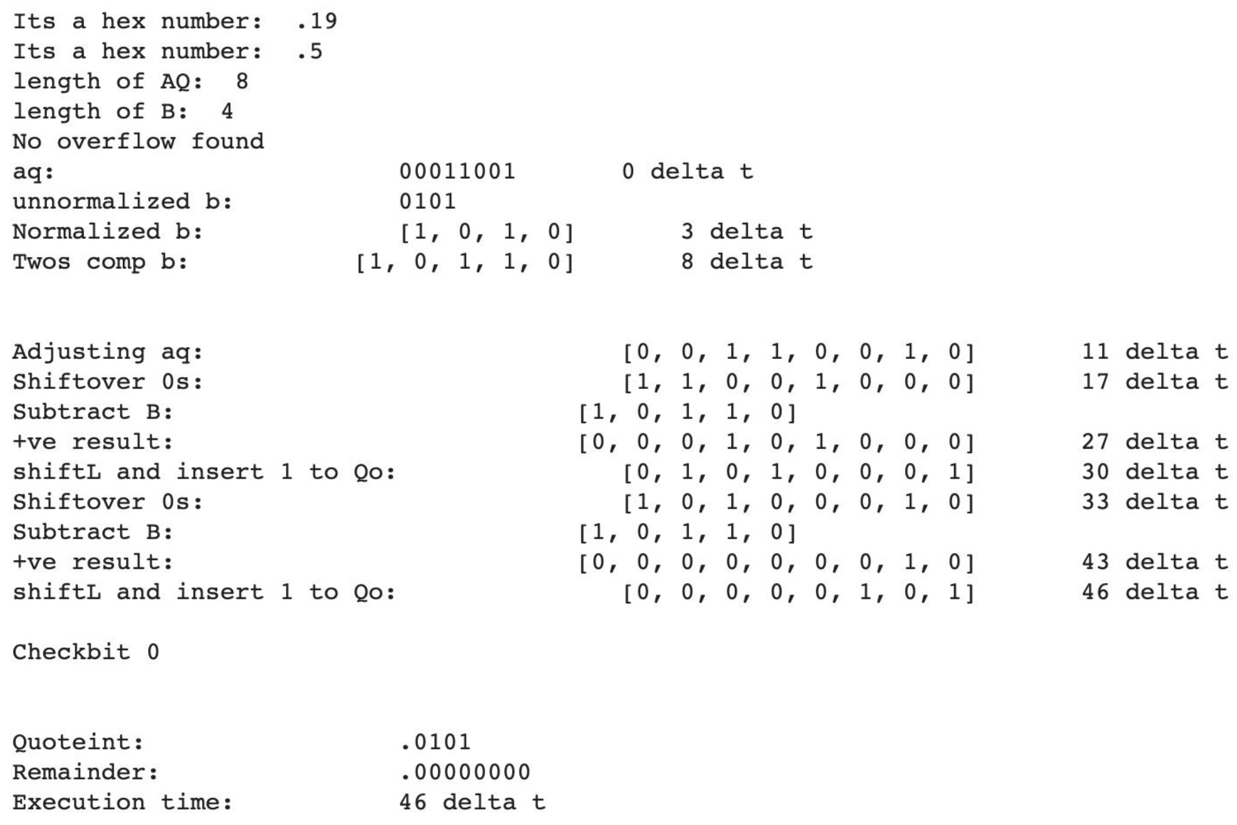
4. Result and Analysis

Different numbers with different bits were used in this project to test the performance of the simulator. Basically, bits length ranging from 4 to 12 for divisor and 8 to 24 for dividends were tested. The given table below shows the numbers that were tested in the project using the simulator with the results (quotient and remainder) and execution time.

Dividend	Divisor	Quotient	Remainder	Exec time
.DE	.E	.1111	.00001100	43Δt
.19	.5	.0101	.00000000	46Δt
.CD	.A	.1010	.00000000	43Δt
.156	.101110 (binary)	.000111	.000000010100	51Δt
.232	.011111 (binary)	.010010	.000000000100	74Δt

.1111	.22	.10000000	.0000000000010001	57Δt
.3333	.11	.11000000	.0000000000001100	78Δt
.5AC2	.79	.11000000	.0000000000000010	66Δt
.9CDE11	.ABC	.111010011101	.000000000000000011000101	136Δt
.672300	.DEF	.011101100110	.000000000000110011000110	153Δt
.ABCDEF	.987	.100100000100	.000000000000000111011011	97Δt

The given figures below show some of the operations with their actual output on the simulator:



```
Its a hex number:  .19
Its a hex number:  .5
length of AQ:  8
length of B:  4
No overflow found
aq:                          00011001       0 delta t
unnormalized b:              0101
Normalized b:                [1, 0, 1, 0]       3 delta t
Twos comp b:             [1, 0, 1, 1, 0]       8 delta t


Adjusting aq:                          [0, 0, 1, 1, 0, 0, 1, 0]       11 delta t
Shiftover 0s:                          [1, 1, 0, 0, 1, 0, 0, 0]       17 delta t
Subtract B:                         [1, 0, 1, 1, 0]
+ve result:                         [0, 0, 0, 1, 0, 1, 0, 0, 0]       27 delta t
shiftL and insert 1 to Qo:             [0, 1, 0, 1, 0, 0, 0, 1]       30 delta t
Shiftover 0s:                          [1, 0, 1, 0, 0, 0, 1, 0]       33 delta t
Subtract B:                         [1, 0, 1, 1, 0]
+ve result:                         [0, 0, 0, 0, 0, 0, 0, 1, 0]       43 delta t
shiftL and insert 1 to Qo:             [0, 0, 0, 0, 0, 1, 0, 1]       46 delta t

Checkbit 0


Quoteint:                    .0101
Remainder:                   .00000000
Execution time:              46 delta t
```

```
Its a hex number:  .CD
Its a hex number:  .A
length of AQ:  8
length of B:  4
There is an overflow
aq:                         01100110       3 delta t
No overflow found
aq:                         01100110       3 delta t
unnormalized b:             1010
Normalized b:               [1, 0, 1, 0]       3 delta t
Twos comp b:             [1, 0, 1, 1, 0]       8 delta t
Shiftover 0s:                             [1, 1, 0, 0, 1, 1, 0, 0]       11 delta t
Subtract B:                            [1, 0, 1, 1, 0]
+ve result:                            [0, 0, 0, 1, 0, 1, 1, 0, 0]       21 delta t
shiftL and insert 1 to Qo:                [0, 1, 0, 1, 1, 0, 0, 1]       24 delta t
Shiftover 0s:                             [1, 0, 1, 1, 0, 0, 1, 0]       27 delta t
Subtract B:                            [1, 0, 1, 1, 0]
+ve result:                            [0, 0, 0, 0, 1, 0, 0, 1, 0]       37 delta t
shiftL and insert 1 to Qo:                [0, 0, 1, 0, 0, 1, 0, 1]       40 delta t
Shiftover 0s:                             [0, 1, 0, 0, 1, 0, 1, 0]       43 delta t

Checkbit 0


Quoteint:                   .1010
Remainder:                  .00000010
Execution time:             43 delta t
```

```
Its a hex number:  .156
Its a binary number .101110 (binary)
length of AQ:  12
length of B:  6
No overflow found
aq:                         000101010110       0 delta t
unnormalized b:             101110
Normalized b:               [1, 0, 1, 1, 1, 0]       0 delta t
Twos comp b:             [1, 0, 1, 0, 0, 1, 0]       7 delta t
Shiftover 0s:                          [1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0]       16 delta t
Subtract B:                         [1, 0, 1, 0, 0, 1, 0]
-ve result:                         [1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0]       26 delta t
shiftL and insert 0 to Qo:             [1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0]       29 delta t
Shiftover 1s:                          [0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1]       38 delta t

Checkbit 1
*** Correcting the remainder by shifting high order half of AQ right and adding b ***
Shift right a:                         [1, 0, 0, 1, 1, 0]       41 delta t
Add B:                                 [1, 0, 1, 1, 1, 0]
After adding B:                        [0, 1, 0, 1, 0, 0]       51 delta t


Quoteint:                   .000111
Remainder:                  .000000010100
Execution time:             51 delta t
```

```
Its a hex number:  .ABCDEF
Its a hex number:  .987
length of AQ:  24
length of B:  12
There is an overflow
aq:                       010101011110011011110111       3 delta t
No overflow found
aq:                       010101011110011011110111       3 delta t
unnormalized b:           100110000111
Normalized b:             [1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1]       3 delta t
Twos comp b:           [1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1]       16 delta t
Shiftover 0s:                            [1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0]       19 delta t
Subtract B:                           [1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1]
+ve result:                           [0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0]       33 delta t
shiftL and insert 1 to Qo:               [0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1]       36 delta t
Shiftover 0s:                            [1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0]       42 delta t
Subtract B:                           [1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1]
+ve result:                           [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0]       56 delta t
shiftL and insert 1 to Qo:               [0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1]       59 delta t
Shiftover 0s:                            [1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0]       74 delta t
Subtract B:                           [1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1]
+ve result:                           [0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0]       88 delta t
shiftL and insert 1 to Qo:               [0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1]       91 delta t
Shiftover 0s:                            [0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0]       97 delta t

Checkbit 0


Quoteint:                 .100100000100
Remainder:                .000000000000000111011011
Execution time:           97 delta t
```

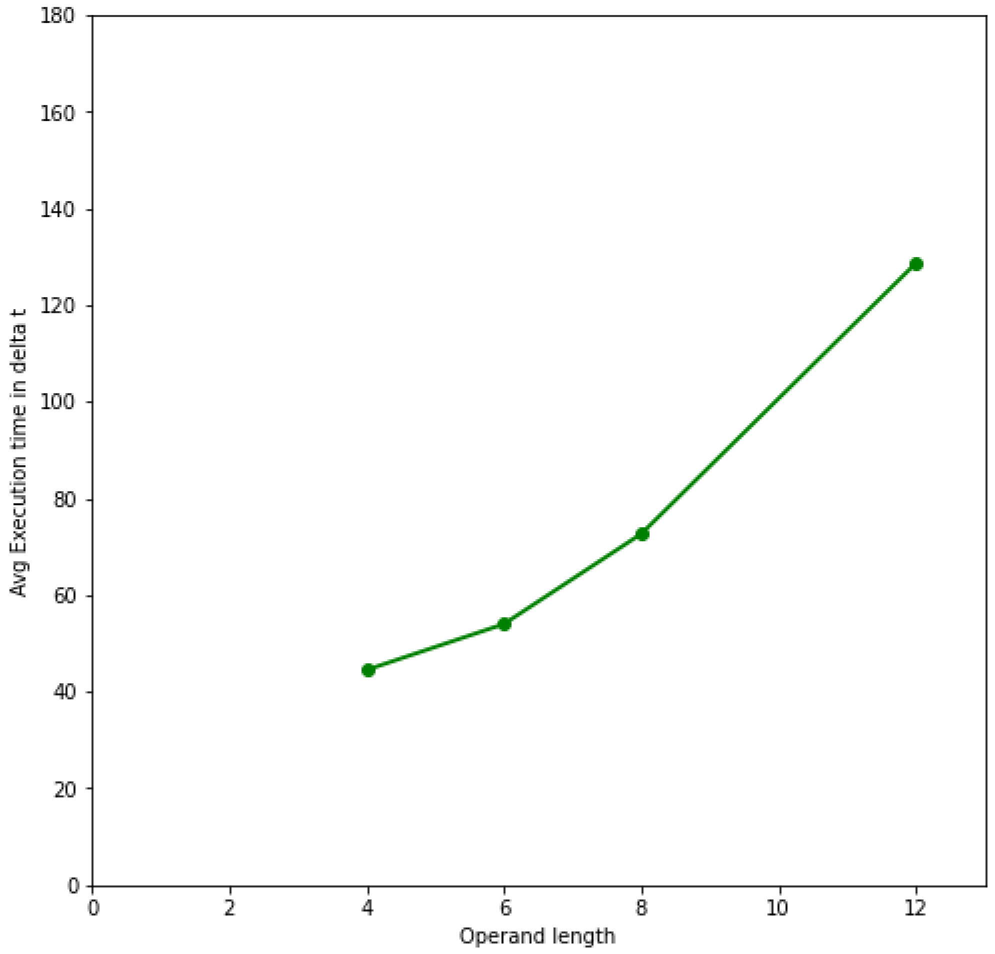
```
Its a hex number:  .9CDE11
Its a hex number:  .ABC
length of AQ:  24
length of B:  12
No overflow found
aq:                       100111001101111000010001       0 delta t
unnormalized b:           101010111100
Normalized b:             [1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0]       0 delta t
Twos comp b:           [1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0]       13 delta t
Subtract B:                           [1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0]
-ve result:                           [1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1]       27 delta t
shiftL and insert 0 to Qo:               [1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0]       30 delta t
Shiftover 1s:                            [0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1]       39 delta t
Add B:                                   [1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0]
-ve result:                           [1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1]       53 delta t
shiftL and insert 0 to Qo:               [0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0]       56 delta t
Add B:                                   [1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0]
+ve result:                           [0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0]       70 delta t
shiftL and insert 1 to Qo:               [0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1]       73 delta t
Shiftover 0s:                            [1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0]       76 delta t
Subtract B:                           [1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0]
-ve result:                           [1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0]       90 delta t
shiftL and insert 0 to Qo:               [1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0]       93 delta t
Shiftover 1s:                            [0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1]       102 delta t
Add B:                                   [1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0]
-ve result:                           [1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1]       116 delta t
shiftL and insert 0 to Qo:               [0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0]       119 delta t
Add B:                                   [1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0]
+ve result:                           [0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0]       133 delta t
shiftL and insert 1 to Qo:               [0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1]       136 delta t

Checkbit 0


Quoteint:                 .111010011101
Remainder:                .000000000000000011000101
Execution time:           136 delta t
```

Gathering all the results from all those divisions, the plot was made using the length of the operand and the average execution time. The data with the same operand length was combined and average execution time was calculated for those. Here's the plot for average execution vs length of the operand.

Operand Length	Execution time in Δt	Avg Execution time
4	43, 46	44.5 Δt
6	51, 57	54 Δt
8	66, 74, 78	72.66 Δt
12	97, 136, 153	128.66 Δt



5. Conclusion

In conclusion, different binary operations were applied in this project to simulate the SRT algorithm which helps to investigate the operations in detail. The simulation of the steps of the algorithm shows how quickly the output can be computed which shows the efficiency of the algorithm. Results can be obtained quickly in comparison with other division algorithms like the Restoring and Non-restoring algorithm. The results and graph show that the execution time keeps increasing if we increase the Operand length. When the operand length was 4 bits, the execution time was around 44 Δt but after the bits increases to 6, the execution time also increases to 54 Δt. Similarly, it keeps on increasing and when the length reaches 12 bits, the execution time increases to 128 Δt. So it concludes that increase in length of operands increases the execution time.

6. References

- SRT Division: Architectures, Models, and Implementations, David L. Harris, Stuart F. Oberman, and Mark A. Horowitz
- SRT FAST DIVISION ALGORITHMS: SIMULATION AND PERFORMANCE EVALUATION, Selvihan Nazlı Yavuzer Ahmet Sertbaş