Analysis of Restoring and Non-Restoring Division Algorithm

By Tim Kellermann, Sudeep Potluri

Due: 29 November 2022

**Abstract**

Throughout this article we will verify the complexity of both the Restoring and Non-Restoring division algorithms. Within this analysis we will include useful metrics such as number of additions/subtractions that will quantify the execution time of both algorithms. Along with verification we hope to prove that the non-Restoring algorithm requires on average less additions/subtractions than the Restoring algorithm.

# Motivation

Our goal is to create a software routine which performs a sequence of subtractions. An algorithm is a well-defined procedure that takes an input and produces some output. This definition is rather broad, many things in everyday life can be written as an algorithm. Brushing your teeth, making your bed, tying your shoes, etc. However, there are many ways to achieve the same solution. For example, I could brush my teeth with a toothpick. This procedure would lead to great inefficiencies. Recognizing these inefficiencies, we have the option to choose what algorithm best fits our needs and minimizes the execution time which then leads to a faster machine. Within a machine the Arithmetic Logic Unit performs integer addition, subtraction, multiplication, and division. Among those operations listed, division is the most taxing on a system. Therefore, throughout this article we hope to provide an algorithm that reduces the number of clock cycles consumed.

# Introduction

Within a machine, division is executed using a sequence of subtractions. This technique produces many inefficiencies, due to this an efficient algorithm was proposed. The Restoring division algorithm functions by reverting the contents of the partial remainder when the divisor is larger than the partial remainder.

The following equation denotes the Restoring algorithm:

(

Therefore, for the Restoring algorithm it is required to subtract, restore and shift within one iteration. The second iteration will also execute subtraction.

Differing, the Non-Restoring algorithm does not revert the contents of the partial reminder if the divisor is larger than the partial remainder. It continues to the second iteration where the divisor will be added to the partial remainder.

The given equation for the non-Restoring division Algorithm:

This illuminates that utilizing non-Restoring techniques the first iteration only requires subtraction and shifting. The following iteration actions will be influenced by the prior subtraction.

Our aspiration is to develop a simulator to mimic the behavior of Restoring and non-Restoring Division techniques. Thus, highlighting the brilliance of the non-Restoring division algorithm.

# Implementation

Our implementation is done using Python. This was decided due to pythons built in functions and easy manipulation of Strings/Arrays in comparison with C/C++. Most of our testing was completed with our own personal computers but can work on the CS department Linux machine. Due to division and multiplication taking similar forms as add and shift or vice versa, the hardware requirements are identical.

B

B

AC

Parallel Adder

Q

Least

Significant Bit

s

Q

s

AC

s

C

out

For out implementation, we decided to separate the parts of Restoring and Non-Restoring into different functions. While the solution is constructed in our Restoring and non-Restoring functions the trivial operations such as two’s complement, add, shift left, etc are their own functions for sake of consistency.

The following steps correspond to the Restoring Algorithm:

1: If upper half of dividend is greater than or equal to divisor, Divide Overflow occurs Terminate operation

2: Initialize the registers i.e. Q = lower half of dividend, B=divisor, A = upper half of dividend, N= number of bits in the dividend.

3: Shift left contents of A and Q by one position

4: Perform A=A-B.

5: If the result is positive, Q­­­0 = 1. Else, Q0 = 0 and the contents of register A are restored.

6: Decrement the value of N by one.

7: Repeat steps 3 through 6 until N=0.

8: Q = quotient and A = remainder

**Algorithm 1** RESTORING (Dividend, Divisor)

1: **procedure** Restoring(Dividend, Divisor)

2: **if(***divideOverflow(*Dividend, Divisor*)***) then**

3: **return** *“Divide Overflow”*

4: **end if**

5: *n = length(Divisor)*

6: *A = upper half of Dividend*

7: *Q = lower half of Dividend*

8: *B = Divisor*

9: **while** *n > 0* **do**

10: *shiftLeft(A,Q)*

11: *subtract(A,B)*

12: **if(***A>B***) then**

13: *Q0 = 1*

14: **end**

15: **if(***A<0***) then**

16: *Q0 = 0*

17: *restore(A)*

18: **end if**

19: *n = n-1*

19: **end while**

20: **return** *A,Q*

21: **end procedure**

The following steps correspond to the Non-Restoring Algorithm:

1: If upper half of dividend is greater than or equal to divisor, Divide Overflow occurs Terminate operation

2: Initialize the registers i.e. Q = lower half of dividend, B=divisor, A = upper half of dividend, N= number of bits in the dividend.

3: Check the sign of contents in register A.

4: If positive, shift left AQ and perform A=A-B. If negative, shift left AQ and perform A=A+B.

5: Check the sign of contents of register A.

6: If positive, Q0 = 1. If negative, Q0 = 0.

7: Decrement the value of N by one.

8: Repeat steps 3 through 7 until N = 0.

9: Check the sign of contents of A. If negative, perform A=A+B.

10: Q = quotient and A = remainder

**Algorithm 2** NONRESTORING (Dividend, Divisor)

1: **procedure** NonRestoring(Dividend, Divisor)

2: **if(***divideOverflow(*Dividend, Divisor*)***) then**

3: **return** *“Divide Overflow”*

4: **end if**

5: *n = length(Divisor)*

6: *A = upper half of Dividend*

7: *Q = lower half of Dividend*

8: *B = Divisor*

9: **while** *n > 0* **do**

10: **if(***A[0] = 1* **) then**

11: *shiftLeft(A,Q)*

12: *add(A,B)*

13: **end if**

14: **if(***A[0] = 0* **) then**

15: *shiftLeft(A,Q)*

16: *sub(A,B)*

17: **end if**

18: **if(***A[0] = 0* **) then**

19: *Q0 = 1*

20: **end if**

21: **if(***A[0] = 1* **) then**

22: *Q0 = 0*

23: **end if**

24: *n = n-1*

25: **end while**

26: **if(***A[0] = 1* **) then**

27: *add(A,B)*

28: **end if**

29: **return** *A,Q*

30: **end procedure**

The following output is generated from the simulator. The given inputs are Dividend = 010100011 and Divisor = 01011 in signed magnitude. The following verification has been done alongside an example homework problem.

Table

Description automatically generated

Table

Description automatically generatedA screenshot of a computer

Description automatically generated with low confidence

**The following data was collected from the test data that was provided.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Length of Divisor** | **Length of the Dividend** | **# of iterations** | **Non-Restoring Add/Sub Count** | **Restoring Add/Sub Count** |
| 9 | 5 | 4 | 4 | 4 |
| 9 | 5 | 4 | 4 | 5 |
| 9 | 5 | 4 | 4 | 5 |
| 9 | 5 | 4 | 4 | 7 |
| 13 | 7 | 6 | 7 | 8 |
| 13 | 7 | 6 | 6 | 8 |
| 13 | 7 | 6 | 7 | 8 |
| 17 | 9 | 8 | 8 | 8 |
| 17 | 9 | 8 | 8 | 10 |
| 17 | 9 | 8 | 9 | 12 |
| 17 | 9 | 8 | 8 | 11 |
| 25 | 13 | 12 | 12 | 12 |
| 25 | 13 | 12 | 12 | 18 |
| 25 | 13 | 12 | 12 | 16 |

**Chart, scatter chart

Description automatically generated**

Figure 1: Graph displaying number of Additions/Subtractions with number of iterations vs length.

# Experimental Analysis

The arithmetic logic unit performs integer division as a sequence of subtractions. However, with each macro operation there a several micro operations that consume many clock cycles. Therefore, it is crucial to minimize the amount of macro operations to lessen the load on the machine.

It is now our goal to verify that the non-Restoring division algorithm performs less additions/subtractions than the Restoring algorithm which enables the machine to perform better. The graph shown above illuminates that as the length of the operands grow the number of additions/subtractions will also increase. This phenomenon occurs for both the Restoring and the non-Restoring case; however, number of additions/subtractions for the non-Restoring case is worst case *n+1.*  Furthermore, for the Restoring algorithm it is consistently utilizing more additions and subtractions as a way to counteract when the Divisor is greater than the partial dividend.

# Conclusions

Here you answer any questions raised in the previous sections. You should end with a wrap–up and firm statement of what a great accomplishment this is.

# References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein., *Introduction to Algorithms, 3rd Edition*. MIT press Cambridge, 2001.

# Appendix

An Appendix starts on a new page after everything else. This is were your code will be.

JAVA Source Code: **fragment.java**

/ / Sample r o u t i n e wi t h a s s e r t s t a t e m e n t s p u b l i c v o i d Build Max Heap(*<*T*>* A [ ] )

*{*

/ / Author : G Howser

/ / Purpose : B u i l d s a Max Heap i n A[ ] wi t h r o o t A[ 1 ]

/ / Heap Test : R e t u r n s ” t r u e ” i f f A[ j ] Max Heap f o r j , j + 1 , j + 2 , . . . n

/ /

/ / Pre c o n d i t i o n : A[ ] i s a non*−*empty a r r a y of comparable o b j e c t s

/ / I n v a r i a n t : A[ j ] i s r o o t of a Max Heap f o r j = i + 1 , i + 2 , . . . n i f ( debug )

*{*

a s s e r t ( ( ( A[ 1 ] *>* 0 ) *| |* (A[ 1 ] *<*= 0 ) ) ) ;

*}*;

i n t h e a p s i z e ;

i n t n = A. l e n g t h *−* 1 ;

h e a p s i z e = A. l e n g t h *−* 1 ; / / NOTE: z e r o r e l a t i v e a r r a y s b u t we a r e i g n o r i n g A[ 0 ] f o r ( i n t i = Math . f l o o r ( ( n / 2 . 0 ) ) ; i *−−*; i *<* 1 )

*{*

Max Heapify ( A, i ) ; i f ( debug )

*{*

a s s e r t ( ( Heap Test (A[ ] , i ) ) ;

*}*;

/ / P o s t c o n d i t i o n : A[ ] i s a Max Heap i f ( debug )

*{*

a s s e r t ( ( Heap Test (A[ ] , 1 ) ) ;

*}*;

*}*

JAVA Source Code: **fragment2.java**

/ / Sample r o u t i n e wi t h a s s e r t s t a t e m e n t s a n d a method t o e v a l u a t e an I n v a r i a n t p u b l i c v o i d Build Max Heap 2 (A [ ] )

*{*

/ / Author : G Howser

/ / Purpose : B u i l d s a Max Heap i n A[ ] wi t h r o o t A[ 1 ]

/ / Heap Test : R e t u r n s ” t r u e ” i f f A[ j ] Max Heap f o r j , j + 1 , j + 2 , . . . n

/ /

/ / Pre c o n d i t i o n : A[ ] i s a non*−*empty a r r a y of comparable o b j e c t s

/ / I n v a r i a n t : A[ j ] i s r o o t of a Max Heap f o r j = i + 1 , i + 2 , . . . n i f ( debug )

*{*

a s s e r t ( ( ( A[ 1 ] *>* 0 ) *| |* (A[ 1 ] *<*= 0 ) ) ) ;

*}*;

i n t h e a p s i z e ;

i n t n = A. l e n g t h *−* 1 ;

h e a p s i z e = A. l e n g t h *−* 1 ; / / NOTE: z e r o r e l a t i v e a r r a y s b u t we a r e i g n o r i n g A[ 0 ] f o r ( i n t i = Math . f l o o r ( ( n / 2 . 0 ) ) ; i *−−*; i *<* 1 )

*{*

Max Heapify ( A, i ) ; i f ( debug )

*{*

a s s e r t ( ( Heap Test (A[ ] , i , h e a p s i z e ) ) ;

*}*;

/ / P o s t c o n d i t i o n : A[ ] i s a Max Heap i f ( debug )

*{*

a s s e r t ( ( Heap Test (A[ ] , 1 , h e a p s i z e ) ) ;

*}*;

*}*

p r i v a t e boo l Heap Test (A[ ] , i n t i , i n t h S i z e )

*{*

/ / Author : G Howser

/ / Purpose : I n v a r i a n t f o r v a r i o u s r o u t i n e s i n Heap S o r t

/ / Heap Test : R e t u r n s ” t r u e ” i f f A[ j ] Max Heap f o r j , j + 1 , j + 2 , . . . n

/ /

/ / Pre c o n d i t i o n : A[ ] i s a non*−*empty a r r a y of i n t e g e r s

/ / P o s t c o n d i t i o n : R e t u r n s t r u e i f and on l y i f A[ 1 . . . h S i z e ] i s a heap i f ( debug )*{*

a s s e r t ( ( ( A[ 1 ] *>* 0 ) *| |* (A[ 1 ] *<* = 0 ) ) ) ; *}* ;

bo o l r e s u l t = t r u e ;

i n t h e a p s i z e = h S i z e ; i n t n = A. l e n g t h *−* 1 ;

/ / Code needs t o be w r i t t e n , so f o r now a lways r e t u r n t r u e i f ( debug )

*{*System . o u t . p r i n t l n ( ”TODO: Code need t o be w r i t t e n *}*; r e t u r n r e s u l t ;

*}*