A NOVEL DIVISION ALGORITHM FOR PARALLEL AND SEQUENTIAL PROCESSING

K. Tatas, D. J. Soudris, D. Siomos, M. Dasygenis and A. Thanailakis

VLSI Design and Testing Center, Dept. of Electrical & Computer Eng., Democritus University of Thrace, Xanthi, Greece, {ktatas, dsoudris, mdasyg, thanail}@ee.duth.gr

ABSTRACT

A new algorithm for reducing the division operation to a series of smaller divisions is introduced. Partitioning the dividend into segments, we perform divisions, shifts, and accumulations taking into account the weight of dividend bits. Each partial division can be performed by any existing division algorithm. From an algorithmic point of view, computational complexity analysis is performed in comparison with existing algorithms. From an implementation point of view, since the division can be performed by any existing divider, the designer can choose the divider which meets his specifications best. Two possible implementations of the algorithm, namely the sequential and parallel are derived, with several variations, allowing performance, cost, and cost/performance trade-offs.

1. INTRODUCTION

Division exhibits the largest latency among basic arithmetic operations. Although less frequent than addition and multiplication, there are important applications such as rendering systems, artificial intelligence, and graphics compression, that require this operation [1].

Many algorithms for division have been developed, such as the digit recurrence algorithms (restoring, non-restoring and non-performing division [1], [2], [3], SRT division [1], [6]), division by convergence (Newton-Raphson method [6]), and implemented in different ways (parallel array, serial divider, etc.), aiming at maximizing performance, while taking area cost into account. Recently, power dissipation has also become a consideration [4], [5].

Employing larger radices is an obvious way of speeding up the division operation [7], but this scheme additionally increases hardware complexity and therefore area cost and cost/performance ratio, a factor by which designs are generally judged. Moreover, in Digital Signal Processing (DSP) applications, the data are derived as a series of sets (or packets), since the throughput rate is more critical than the overall system latency. The above two issues provide the motivation for the work that follows.

In this paper a novel algorithm for reducing the division operation to a series of smaller divisions is introduced. We partition the dividend into partitions of fixed wordlength and by executing divisions (using smaller dividends), shifts, and accumulations, taking into account the weight of dividend bits, we calculate the correct quotient and remainder. The partial divisions can be executed in a sequential or parallel fashion. The idea of performing division for a large dividend against a small divisor has been suggested in the past [8], but in a digit recurrence (iterative), not in a partitioning fashion. Furthermore, the algorithm was implemented only in software and for very large radices.

Computational complexity analysis and exhaustive comparisons between the new algorithm and existing ones are performed. The required divisions can be performed by any existing algorithm. Two possible implementations of the algorithm, namely the sequential and parallel are derived and comparison with existing ones in terms of area cost, performance, and cost/performance is done. The results prove the efficiency of the proposed algorithm.

2. THE PROPOSED ALGORITHM

The basic idea behind the proposed algorithm is that the operation of division can be considered as a fraction \(X/D\), whose numerator and denominator correspond to dividend and divisor, respectively. We assume an \(n\)-bit divisor and a \(4n\)-bit dividend. Let us also assume two unsigned numbers

\[P_1 = \sum_{i=0}^{2n-1} x_{2i+1} 2^{2i} \quad \text{and} \quad P_2 = \sum_{i=0}^{2n-1} x_i 2^i,\]

so that \(P_1 + P_2 = X\). Then, \(X/D\) can be rewritten as:
\[
\frac{p_1 + p_2}{D} = \frac{p_1}{D} + \frac{p_2}{D}
\]. Thus, we have to calculate a sum of fractions. Actually, we “partition” the numerator into two numbers with appropriate weights.

### 2.1 The developed algorithm for \( p=2 \)

For the sake of simplicity, we provide the new algorithm for \( p=2 \), i.e. two partitions and radix \( b=2 \). It can be generalized for every \( p>2 \) with slight modifications.

**STEP 1.** The \( 2n \) most significant bits of the dividend are used as input for our divider implementation. Performing the division with divisor \( D \), a pair of partial quotient, \( q_1 \), and partial remainder \( r_1 \) are derived.

**STEP 2.** A magnitude comparison on the \((2n+1)\)-th dividend bit is performed:

1. If it is “0”, then the remaining \( 2n \) bits of the dividend are used as dividend in the second division, resulting into a new pair of partial quotient \( q_2 = \sum_{i=0}^{n-1} q_{2,i} 2^i \) and partial remainder: \( r_2 = \sum_{i=0}^{n-1} r_{2,i} 2^i \). Then, we shift \( q_2 \) and \( r_2 \) to the right \( 2n \) positions. Thus, we obtain:
   \[
   q_2' = \sum_{i=0}^{n-1} q_{2,i} 2^{(2n+1)+i},
   \]
   \[
   r_2' = \sum_{i=0}^{n-1} r_{2,i} 2^{(2n+1)+i}.
   \]

2. If it is “1”, we use the next \( 2n-1 \) bits forcing a “0” in the MSB position. Executing the second division a new quotient \( q_2 = \sum_{i=0}^{n-1} q_{2,i} 2^i \) and remainder \( r_2 = \sum_{i=0}^{n-1} r_{2,i} 2^i \) are obtained. Then, we shift \( q_2 \) and \( r_2 \) to the right \( 2n-1 \) positions. Thus, we obtain:
   \[
   q_2' = \sum_{i=0}^{n-1} q_{2,i} 2^{2n+i}\quad \text{and}\quad r_2' = \sum_{i=0}^{n-1} r_{2,i} 2^{2n+i}.
   \]

The above two steps can be executed in parallel.

**STEP 3.** Sum of the partial quotients and remainders calculated in steps 1 and 2, produces the final quotient: \( q = q_1 + q_2' \) and final remainder: \( r = r_1 + r_2' \).

**STEP 4.** The condition \( r < d \) is checked. If it is true then the execution of the algorithm is over, i.e. \( q=Q \) and \( r=R \). If not, then an additional step is required.

**STEP 5.** i) Perform the operation \( r-D \). The difference is the final remainder \( R \). ii) Perform the operation \( q+1 \).

The sum is the final quotient \( Q \).

The proposed algorithm is illustrated in flowchart in Fig. 1. Since it is based on a fundamental mathematical property (i.e. sum of fractions), it is independent from the chosen division scheme for performing the partial divisions. Therefore, a designer can select any of the existing basic algorithms, such as restoring or non-restoring division [1]. Furthermore, the partial divisions can be executed in a parallel or sequential fashion (recursively). Also, the algorithm is precisely the same in the case that dividend and divisor are fractions. Simply, all the sums mentioned above will have negative exponents instead of positive.

**2.2 Computational complexity analysis**

A standard way of comparing algorithms without getting into implementation details is through computational complexity analysis. Computationally, the new algorithm requires (for a \( 4n \) – bit dividend and an \( n \)-bit divisor):

1. Two \( n \)-bit divisions

![Flowchart of the proposed algorithm](image-url)
ii) Two shifts by 2n bit positions
iii) Two additions of 3n-bit wordlength numbers
iv) One 3n-bit magnitude comparison
v) Possibly an additional 3n-bit subtraction

A comparison between the proposed algorithm and various well-known existing division algorithms for \( p = 2 \) is shown in Table 1.

In the computational complexity analysis of the proposed algorithm, we assume both sequential and parallel execution of the partial divisions both by a non-restoring scheme. The reason the parallel scheme is not twice as fast as the sequential is that the magnitude comparison and the possible corrective subtraction in the end exist in both versions of the algorithm and cannot be executed in parallel, since the first is the condition for the execution of the second.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Dividend wordlength</th>
<th>Number of computations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restoring Unsigned</td>
<td>4n</td>
<td>4 ( n^2 )</td>
</tr>
<tr>
<td>Non-Restoring Unsigned</td>
<td>4n</td>
<td>Best case: 4 ( n^2 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Worst case: 4 ( n^2 + 2n )</td>
</tr>
<tr>
<td>SRT Radix-2</td>
<td>4n</td>
<td>Best case: ( 4n )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Worst case: ( 8n^2 + 4n )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Average: 8n^2 ( /2.67 )</td>
</tr>
<tr>
<td>Knuth’s Software Algorithm [8]</td>
<td>4n</td>
<td>( \left( \frac{6n(2n+1)(n+1)}{2} \right) )</td>
</tr>
<tr>
<td>Proposed Algorithm (parallel)</td>
<td>4n</td>
<td>Best case: ( n^2 + 6n )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Worst case: ( n^2 + 9n )</td>
</tr>
<tr>
<td>Proposed Algorithm (sequential)</td>
<td>4n</td>
<td>Best case: ( 2n^2 + 6n )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Worst case: ( 2n^2 + 9n )</td>
</tr>
</tbody>
</table>

Table 1. Computational complexity analysis and comparisons between division algorithms

From the calculations seen in Table 1, it can be easily derived that for values of \( n \) greater than 2 (hence for any practical wordlength) the proposed algorithm has a smaller number of computations than the restoring and non-restoring division algorithms. Therefore, the new algorithm can also be used to speed up the non-restoring division algorithm or Knuth’s algorithm [8]. SRT division though, being of low complexity can not be accelerated by this process, because the additional steps overcome the gains of executing two smaller divisions in this case.

**3. IMPLEMENTATION OF THE ALGORITHM**

We have come up with two possible implementations of the discussed algorithm: a sequential and a parallel one, allowing trade-offs between performance and cost. Figs. 2 and 3 illustrate the sequential and parallel implementation respectively. For simplicity reasons, we examine the case of \( p = 2 \). The implementation

![Fig. 2. Sequential implementation of generalized algorithm.](image-url)

![Fig. 3. Parallel implementation of the proposed algorithm](image-url)
of the algorithm requires certain additional modules to the division operation we choose as basis, in this case the parallel array divider [1, 2]. This is by no means a limitation of the proposed algorithm, it simply allows us to evaluate the implementations in terms of cost (area), performance and cost/performance ratio.

In Fig. 2, initially the dividend is loaded on a 4n bits register. The preprocessing unit sets a “0” in the second partition’s MSB position The multiplexer determines the order of partitions loading to the divider. After the first division is performed, the two n-bit quotients and two n-bit remainders are loaded to four separate registers, by demultiplexing not illustrated here. The remaining operations of the implementation are rather straightforward. Each accumulator adds the two remainders/quotients. The magnitude comparator checks the condition \( r < d \). If it is true, the accumulator’s output is the calculated remainder. Otherwise, a subtractor should perform the operation \( r - d \) and the quotient should be increased by one.

A sequential implementation does not always affect the divider performance positively, because the dividend’s partitions are handled in a recursive manner. Significant improvement in performance at the expense of area cost can be achieved by performing the divisions of two partitions in parallel fashion. The parallel architecture exhibits almost the same structure with the sequential one, except for the second divider and the mux, respectively.

The above implementations were described in VHDL and implemented using Xilinx ISE tools. Table 2 gives the comparison results between the proposed and existing implementations for dividend wordlength 128 bit \( (n=32) \) for a Xilinx Virtex-E 1000 device. Column 2 comprises a list of the existing and proposed implementations, while the columns 3, and 4 the corresponding values for longest combinational path delay in ns (which determine the maximum frequency for our implementations) and slice count respectively. It is assumed that the proposed implementations use the parallel array divider, since it has high complexity and low performance. [1]. It should be noted that these realizations were optimized for area. Had they been optimized for timing, shorter delay, and greater slice count would have occurred.

<table>
<thead>
<tr>
<th>( N )</th>
<th>Implementation</th>
<th>Delay(ns)</th>
<th>Slices</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>Parallel – array [2]</td>
<td>3592</td>
<td>4316</td>
</tr>
<tr>
<td></td>
<td>Proposed Sequential ( n=16,p=2 )</td>
<td>1712</td>
<td>2136</td>
</tr>
<tr>
<td></td>
<td>Proposed Parallel, ( n=16, p=2 )</td>
<td>1498</td>
<td>3050</td>
</tr>
</tbody>
</table>

Table 2. Comparisons between the proposed (using array divider) and various existing implementations.

The sequential implementation exhibits significant area efficiency (less than half slices) compared to the parallel array. Its execution time is also reduced if no correcting steps are required. If correcting steps are required, the efficiency of the sequential implementation degrades.

In the parallel implementation, the delay is reduced slightly, but the latency will also be dramatically since only one division is performed by each divider. If no corrective steps are required, the delay is less than half that of the parallel array divider. The total slice count is increased, since two dividers are now required. Still, the parallel array increases in size four times each time its wordlength doubles, while its execution time also doubles. These seemingly long (for FPGA implementation) execution times can be dramatically improved by using pipelining. These implementations are not meant for FPGA implementation, but we did so for sake of comparison and evaluation.

4. CONCLUSIONS – FUTURE WORK

A novel algorithm for reducing the division operation to a series of smaller divisions was presented. The proposed algorithm and two implementations of it were compared with many existing algorithms and implementations in terms of computations, area cost, performance, and cost/performance. The derived architectures exhibited better characteristics than many existing implementations. Future plans include extending the algorithm to other number representations, the development of a new algorithm for square root calculations (division and square root are similar operations) and their VLSI design taking into account the issue of power dissipation.

5. REFERENCES