Design of High Performance Floating Point SRT Divider Using Divisor and Partial Remainders Estimates

Saifur Rahman¹, Ahmad Hesham Ansri²
¹Department of Electronics Communication Engineering, Integral University, Lucknow, INDIA.
²Department of Electronics Communication Engineering, Integral University, Lucknow, INDIA.

ABSTRACT

SRT dividers are common in modern floating point units. Higher division performance is achieved by retiring more quotient bits in each cycle. Previous research has shown that realistic stages are limited to radix-2 and radix-4. Higher radix dividers are therefore formed by a combination of low-radix stages. In this paper, we present an analysis of the n-bit divider and Comparative analysis of different dividers in case of delays and performance. We show the performance and area results for a wide variety of divider architectures and implementations. We conclude that divider performance is only weakly sensitive to reasonable choices of architecture but significantly improved by restoring and non restoring techniques.

Keywords— VLSI, SRT Division, n-bit Divider, Delays.

I. INTRODUCTION

Arithmetic circuits have received relatively little attention from the verification community, except by those using methods based on theorem proving, e.g., [6]. This inattention is due to two main reasons. First, many perceive that arithmetic circuit design is fairly straightforward—the same implementation techniques have been used for years, and designers are confident of their ability to detect errors using conventional simulation. Intel’s recent experience with its Pentium floating point divider [5] has exposed the error in this thinking. There are many places one can make mistakes in designing these circuits, some of which may be very hard to detect with the limited number of cases that can be tested by simulation. Second, these circuits are especially challenging for methods based on ordered Binary Decision Diagrams (BDDs), the most popular alternative to theorem proving [3]. The BDDs representing the outputs of a multiplier grow exponentially with the word size [2], making them impractical for word sizes much beyond 16 bits. A similar result has been shown for representing the outputs of a divider [7]. On the other hand, the outputs of simpler units such as adders, subtractors, and comparators are represented very efficiently with BDDs.

In this paper, we demonstrate that restoring and non restoring based dividers can be usefully applied to complex arithmetic circuits. Even though it is not feasible to verify the overall circuit functionality, just verifying one iteration can uncover many possible design errors. We demonstrate this by showing the desired behavior for one iteration of radix4 SRT division [1], as used in the Pentium divider, can be specified and verified using BDDs. This verification will detect incorrect entries in the “PD” table, used to generate a quotient digit on each division step, such as occurred in the Pentium, as well as other potentially subtle design errors. Going on beyond verification, we show that a correct PD table can be generated automatically. Our method extends the correction method described by Madre and Coudert [4] to handle logic blocks with larger numbers of inputs and outputs.

The intention of this paper is not to advance the state of the art in formal verification, but rather to illustrate how existing technology could be applied to real life designs. Current tools fall well short of the ultimate goal of verifying complete floating point hardware designs against a high level, mathematical specification, e.g., the IEEE Floating Point Standard. Nonetheless, they can be applied to key components of a system, enhancing overall design quality. The decision of which components to verify requires a weighing of three factors: 1) how cleanly the functionality of the component can be specified, 2) whether the subsystem is tractable for existing formal verification tools, and 3) the chances that the subsystem may contain subtle design errors that cannot be detected by less formal measures, such as simulation. Although this “opportunistic verification” does not guarantee complete system correctness, any measures that can reduce the chances of design errors are worthy of consideration. The decision of whether and where to apply such verification should be based largely on economic grounds. That is, the cost of formally specifying and verifying a component should be compared to the cost of other verification
methods, such as simulation, as well as to the cost of any design errors that could be missed by these less comprehensive approaches.

The Pentium FDIV problem provides a clear illustration for the potential value of opportunistic verification. Compared to the $475 million charge that Intel took against its 1994 revenues to replace defective Pentium chips, one can easily justify applying verification tools to a number of subsystems, including many parts of the floating point unit.

II. SRT DIVISION

A. Definitions:

In this analysis, the input operands are assumed to be represented in a normalized floating point format with n bit significands in sign-and-magnitude representation. The algorithms presented here are applied only to the magnitudes of the significands of the input operands. Techniques for computing the resulting exponent and sign are straightforward.

The most common format found in modern computers is the IEEE 754 standard for binary floating point arithmetic. This standard defines single and double precision formats, where n=24 for single precision and n=53 for double precision. The significand consists of a normalized quantity, with an explicit or implicit leading bit to the left of the implied binary point, and the magnitude of the significand is in the range $[1,2)$. However, to simplify the presentation, this analysis assumes fractional quotients normalized to the range $[0.5,1)$.

The quotient is defined to comprise $k$ radix-$r$ digits with

$$r = 2^b$$

$$k = n/b$$

where a division algorithm that retires $b$ bits of quotient in each iteration is said to be a radix-$r$ algorithm. Such an algorithm requires $k$ iterations to compute the final $n$ bit result and thus has a latency of $k$ cycles. The cycle time of the divider is defined as the maximum time to compute one iteration of the algorithm. Depending upon the implementation, this may or may not be the same as the cycle time of the processor.

The following recurrence is used in every iteration of the SRT algorithm:

$$rP_0 = \text{dividend}$$

$$P_{j+1} = rP_j - q_{j+1}\text{divisor}$$

where $P_j$ is the partial remainder, or residual, at iteration $j$. In each iteration, one digit of the quotient is determined by the quotient-digit selection function:

$$q_{j+1} = \text{SEL} (rP_j, r, \text{divisor})$$

The final quotient after $k$ iterations is then

$$q = \sum_{j=1}^{k} q_{j}r^{-j}$$

B. Divider Parameters:

1) Choice of Radix:

The fundamental method of decreasing the overall latency (in machine cycles) of the algorithm is to increase the radix $r$ of the algorithm, typically chosen to be a power of 2. However, this latency reduction does not come for free. As the radix increases, the quotient-digit selection becomes more complicated, which may increase the cycle time. Moreover, the generation of all required divisor multiples may become impractical for higher radices. Oberman [8] shows that the delay of quotient selection tables increases linearly with increasing radix, while the area increases quadratically. While prescaling of the input operands [9] reduces table complexity at the expense of additional latency, nevertheless the difficulty in generating all of the required divisor multiples for radix 8 and higher limits practical divider implementations to radix 2 and radix 4.

2) Choice of Quotient Digit Set:

For a given choice of radix $r$, some range of digits is decided upon for the allowed values of the quotient in each iteration. The simplest case is where, for radix $r$, there are exactly $r$ allowed values of the quotient. However, to increase the performance of the algorithm, a redundant digit set is used. This allows a quotient digit to be selected based upon an approximation of the partial remainder, permitting the use of a redundant remainder representation as discussed in the next section. Small errors in the quotient due to the remainder approximation are corrected in later iterations. Such a digit set is composed of symmetric signed-digit consecutive integers, where the maximum digit is $a$. The digit set is made redundant by having more than $r$ digits in the set. By using a larger number of allowed quotient digits, the complexity and latency of the quotient selection function is reduced. However, choosing a smaller number of allowed digits for the quotient simplifies the generation of the multiple of the divisor. Specifically, for radix 2, the digit set is $\{-1; 0; 1\}$. For radix 4, there are two typical choices for the digit set: minimally redundant $\{-2; -1; 0; 1; 2\}$ and maximally redundant $\{-3; -2; -1; 0; 1; 2; 3\}$. The quotient selection logic for a maximally-redundant radix 4 digit set is about 20% faster and 50% smaller than for a minimally redundant digit set [10]. However, maximally-redundant radix 4 requires the computation of the $3x$ divisor multiple, which typically requires extra initial delay and area.

3) Choice of Remainder Representation:

The partial remainder also can be represented in two different forms, either redundant or nonredundant. Each iteration of the algorithm requires a subtraction to compute the next partial remainder. If this partial remainder is in a nonredundant form, then this operation requires a time-consuming full-width carry-propagate-
adder, increasing the cycle time. Therefore, the partial remainder is typically stored in redundant form so that a fast carry-free adder, such as a carry-save adder (CSA), can be used in the partial remainder calculation.

![Block Diagram](image)

Figure 1: SRT divider block diagram

### III. PREVIOUS APPROACH

In recent years computer applications have increased in their computational complexity. High-speed floating-point hardware is a requirement in many VLSI systems to meet these increasing demands. An important component of the floating point unit is the divider. There are many methods for designing division hardware, including quadratically converging algorithms, such as Newton–Raphson, and linear converging algorithms, the most common of which is SRT [15]. SRT division computes a quotient one digit at a time, with an iteration time independent of the operand length.

The theory of SRT division is discussed thoroughly in Atkins [10]. Several SRT implementations have been reported, including radix 2 dividers by Knowles [14], radix 4 by Birman [16], radix 8 by Fandrianto [13], and radix 16 by Carter [12]. SRT dividers with simplified quotient-digit selection using operand range restriction have been presented. Harris [17] discusses detailed algorithmic and circuit tradeoffs in SRT divider design.

There are many performance and area tradeoffs when designing an SRT divider. One metric for comparison of different designs is the minimum required truncations of the divisor and partial remainder for quotient-digit selection. Atkins [10] provide such analyzes of the divisor and partial remainder precisions required for quotient-digit selection. Burgess and Williams [11] present in more detail allowable truncations for divisors and both carry-save and borrow-save partial remainders. However, a more detailed comparison of quotient-digit selection complexity between different designs requires more information than input precision. This paper analyzes in detail the effects of algorithm radix, redundancy, divisor and partial remainder precision, and truncation error on the complexity of the resulting table. Complexity is measured by the number of product terms in the final logic equations, and the delay and area of standard-cell implementations of the tables. These metrics are obtained by an automated design flow using the specifications for the quotient-digit selection table as input, a Gray-coded PLA as an intermediate representation, and an LSI Logic 500 K standard-cell implementation as the output. This paper also examines the effects of additional techniques such as table folding and longer external carry-assimilating adders on table complexity. Using the methodology presented, it is possible to automatically generate optimized high radix quotient-digit selection tables.

### IV. SIMULATION RESULTS

In case of 16-bit divider we took 16-bit dividend is “0000000010101010” and 16-bit divisor is “0000000000010000” then the reminder is “0000000000000000”. In fig 2 we have shown the simulation result for 16-bit divider.

![Simulation Result](image)

Figure 2: Simulation Result for 16 Bit Divider

In case of 8-bit divider we took 8-bit dividend is “10101010” and 8-bit divisor is “00000010” then the reminder is “00000000”. In fig 3 we have shown the simulation result for 8-bit divider.

![Simulation Result](image)

Figure 3: Simulation Result for 8 Bit Divider

In case of 4-bit divider we took 4-bit dividend is “1010” and 4-bit divisor is “0010” then the reminder is “0000”. In fig 4 we have shown the simulation result for 4-bit divider.

![Simulation Result](image)

Figure 4: Simulation Result for 4 Bit Divider
In case of n-bit divider we took as an example 11-bit dividend is “01010101010” and 11-bit divisor is “00000000010” then the remainder is “00000000000”. In fig 5 we have shown the simulation result for n-bit divider.

V. CONCLUSION

We have shown how to divide n-bits by implementing n-bit divider. In this work we also compare results of 16-bit divider, 8-bit Divider, 4-bit Divider, and n-Bit divider. And we found that the logic delay and route delays are almost same. So in place of use 16-bit divider, 8-bit Divider, 4-bit Divider... etc. we can use n-bit divider. It will be reduce the cost, low power consumption and less area. In modern VLSI implementations, these tradeoffs directly affect the time and space required since custom designs use only the required number of bits.

In 4-bit divider the logic delay is 5.753 ns and route delay is 1.755 ns then the total delay for 4-bit divider is 7.508 ns. In case of 8-bit divider the logic delay is 5.753 ns and route delay is 1.332 ns then the total delay for 8-bit divider is 7.085 ns. In case of 16-bit divider the logic delay is 5.753 ns and route delay is 1.566 ns then the total delay for 16-bit divider is 7.319 ns. In case of n-bit divider the logic delay is 6.302 ns and route delay is 2.601 ns then the total delay for n-bit divider is 8.903 ns. If we look simply then we can say that the total delay of n-bit divider is greater than the rest of Dividers but we cannot decide which divider is required is everywhere. So that if we required three different dividers like 4, 8, 16-bit dividers then total delay will be 21.912 ns. That is why we use n-bit divider in place of these different dividers. Then the delay will be just 8.903 ns. This delay is clearly less than those all these dividers. Here we are talking about just three dividers but practically we need more bit dividers and more dividers.

So if we did not take n-bit divider then we have to design many more different dividers for different bit. They will definitely take more time, area, power and also cost. But in VLSI we have to save all of these. Then the result is n-bit divider is taking less power, less area and reduce the cost.

<table>
<thead>
<tr>
<th>TABLE I</th>
<th>RESULT COMPARISON</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divider</td>
<td>Logic Delay (ns)</td>
</tr>
<tr>
<td>4-Bit Divider</td>
<td>5.753</td>
</tr>
<tr>
<td>8-Bit Divider</td>
<td>5.753</td>
</tr>
<tr>
<td>16-Bit Divider</td>
<td>5.753</td>
</tr>
<tr>
<td>n-Bit Divider</td>
<td>6.302</td>
</tr>
</tbody>
</table>

REFERENCES


