The Forgotten Horseman: Digital Implementation of Arithmetic Division and Resources to Learn and Teach Its Complexities

Dr. Peter Jamieson, Miami University

Dr. Jamieson is an associate professor in the Electrical and Computer Engineering department at Miami University. His research focuses on Education, Games, and FPGAs.

Nathaniel David Martin, Miami University

The Forgotten Horseman - Digital Implementation of Arithmetic Division and Resources to Learn and Teach Its Complexities

Abstract

Of the four arithmetic functions, Division is the most complex. When we learn arithmetic in school it is the last of the arithmetic operations taught after first learning addition, subtraction, and multiplication. This is the case, again, since division is the most complex of the four operations to learn, and similarly, from a digital implementation standpoint, division is the most complex. In undergraduate digital system design courses, we typically teach addition, subtraction, and maybe multiplication, but division is treated as a more complex algorithm that is rarely touched upon. We believe this is the case because of the complexity of division and its general lack of use in most algorithms. From our exploration of where division is used, it appears in the Diffie-Helmann algorithm and pseudo-random number generators. We hypothesize the second reason division is not typically explored in undergraduate education is that the complexity of division implementation is beyond most instructors and there are few tools to help us all understand division.

In this work, we investigate the efficiency of different division algorithms as the bit width of the division increases, specifically for unsigned integer division. Our target architecture is a Field Programmable Gate-Array (FPGA) where we measure each divider's area (measured by Logic Elements of the FPGA) and the speed of division. We investigate non-restoring division, Radix-2 SRT division, Radix-4 SRT division, and Goldschmidt division at widths ranging from 8 bits to 1024 bits. Each of these dividers at each bit-width is tested for functionality and is measured based on speed (a combination of critical path and number of clock cycles to complete) and area (number of logic elements (LEs) needed for the divider). Our results provide a general trend for the area and speed for these division algorithms as the width of the division grows for FPGAs, and our tool is open source so that others can implement these dividers with detailed examples of their stepwise operation so that educationally we can develop an understanding of division algorithms.

1. Introduction

The arithmetic operation of division for computing is the most difficult to implement in hardware (HW) of the four basic arithmetic operations: addition, subtraction, multiplication, and division. Addition, subtraction, and multiplication are used in many algorithms and are taught as part of a digital design course. There has been significant development to improve the HW design to make these arithmetic operations fast and efficient. Division HW, however, has not been a subject of as much focus mainly because the number of algorithms that use division is small, and the complexity of division makes it hard to improve the efficiency of the HW. This work studies various wide-bit dividers to examine how different design approaches impact the speed and area consumption of HW when targeting FPGA technology. Our goal is to provide an understanding of how bit-width impacts division implementations so that future work in large-width divisions can focus on improving the most appropriate implementation of the target divider for large-bit-width applications. What we learned in this study is that previous literature and educational resources for division are limited, and a few worked-out examples have propagated deep into the educational space. The result is that HW division has many implementation details that are hidden from the average designer. To help alleviate this we provide tools in our work to help educate on the details of implementing division including the capability to see worked-out examples beyond the few examples that exist in slide decks and textbooks.

This work performs an evaluative case study of different algorithmic approaches to implementing division on FPGA HW, and we will analyze how these algorithmic approaches are quantified (as measured by speed and area on an FPGA) as the number of bits (bit-width) in the division operation increases. The importance of this study is driven by a few algorithmic cases where division is useful. These example cases could utilize a specialized HW divider for large bit division including:

- Diffie-Hellman algorithm [1]
- Pseudo-random Number generation [2]

This work explores the relationship between HW division bit-width and the speed and area of the circuit implemented on an FPGA. For an FPGA, speed is measured as the time taken to complete a division operation, and this can be measured by the number of clock cycles needed to complete a division operation algorithm multiplied by the critical path of the circuit resulting in a time measurement. Area, on FPGAs, is measured by the number of resources needed as expressed by ALMs, DSP blocks, and memory bits used for the FPGA design. For this study, we measure area using reports generated by an Industrial FPGA CAD tool: Quartus by Intel [3], which synthesizes our HW dividers to be programmed to a target FPGA. To effectively measure the speed of our dividers, we will implement each of our dividers as a custom arithmetic block on a RISC-V soft processor, and measure the number of clock cycles needed to complete the division operation. The RISC-V processor we use is called Trireme and was created by Arizona State University [4].

The contributions of this work are:

- A detailed study of HW division implemented on FPGAs in terms of their area and speed results. The parameters of this study are the choice of HW algorithm and the bit-width of the division operation.
- A methodology to evaluate custom division instructions on a RISC-V processor.
- Open-source scripts[1] that allow HW designers and educators a means to quickly generate specific bit-width dividers for different HW algorithms as Verilog design modules.
- Within the above tool, a process to generate step-by-step examples of the internal algorithm operation of non-restoring, Radix-2 SRT, Radix-4 SRT, and Goldschmidt divisions for any

[1]`https://github.com/marti693/Variable-Width-Division-Scripts`

two inputs and any bit-width.

The last contribution is both significant for future implementations and a surprising missing piece from the existing literature. Though many people have implemented these division algorithms previously, and several textbooks and online resources describe the designs of these algorithms, the details of actual implementation including how to align and recover the correct results of a division are left out or hidden in the mathematical details. Therefore, to help future designers of these arithmetic division algorithms we have included output steps of the algorithms in our scripts so that instead of propagating one example of these division algorithms throughout the literature, designers can now dynamically implement any division example and see the internal steps of the algorithm.

The paper is organized as follows: Section provides background on the division algorithms we explore, and describes in detail how they are implemented using HW. Section describes our methodology for creating and testing the dividers at variable bit-widths. Section provides the results from our experimental testing. Section discusses the results and concludes the work.

2. Background

2.1 Division HW Implementations

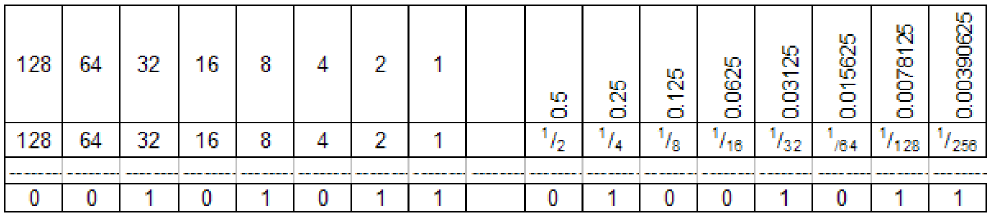
Division is a mathematical operation in which a number, the dividend ($Dividend$), is separated into equal pieces. The number being separated is called the dividend, and the number of equal pieces is called the divisor ($Divisor$). The result of the division operation is referred to as the quotient ($Quotient$). For Natural number division, the remainder ($Remainder$) is the leftover value that cannot be further divided by the divisor. Natural number division is shown in the following equation:

$$\frac{Dividend}{Divisor} = Quotient + \frac{Remainder}{Divisor} \tag{1}$$

In binary systems, there are different approaches for representing numbers, which then impacts the design of the division HW. Each number system representation has different ways of being implemented for division. Integer division is the operation where the $Dividend$, $Divisor$ and $Quotient$ are all strictly integers. In integer division, there is also a $Remainder$ value which is the output of the modulus operation. In many programming languages, the "/" operator is the division operation, and "%" is the modulus operation.

We will focus on unsigned integer division in this work, but there also exist fixed and floating point number representations. These number formats can represent a subset of real numbers on a digital system where the radix point (decimal point) is either floating or fixed in a location. Fixed-point numbers have a fixed radix point, which means the number of bits reserved for the integer and fraction part of the number is constant. Floating point numbers have a flexible radix point, which allows for a larger range of magnitudes to be represented, but less precision.

A fixed-point representation of binary numbers has a defined location for a "decimal point". To the left of the decimal point, the values for each digit of the binary value are the same as an integer, and the right side is defined by (2^{-n}) where n is the number of digits from the decimal point. As can be seen in Figure 1, the values at each location to the right are 0.5, 0.25, 0.125, etc.



128	64	32	16	8	4	2	1		0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125	0.00390625
128	64	32	16	8	4	2	1		$^1/_2$	$^1/_4$	$^1/_8$	$^1/_{16}$	$^1/_{32}$	$^1/_{64}$	$^1/_{128}$	$^1/_{256}$
--------	--------	--------	--------	--------	--------	--------	--------	--------	--------	--------	--------	--------	--------	--------	--------	--------
0	0	1	0	1	0	1	1		0	1	0	0	1	0	1	1

Figure 1: Fixed-point Binary Representation. The binary number is represented on the bottom row of the figure, which consists of only ones and zeros. The base 10 number can be reconstructed by adding each of the corresponding powers of two where the binary digit is a 1.

To convert the number into base 10, the values corresponding to the 1s in the number get summed. Just like a binary integer representation most real numbers can be approximately represented this way. The example in Figure 1 is 00101011.01001011_2, which equals $(32 + 8 + 2 + 1 + .125 + .03125 + .0078125 + .00390625 = 43.16796875_{10})$.

Although this work only looks at unsigned integer division, many of our implementations rely on floating point numbers to perform the division algorithm (such as SRT division and the Newton Methods of division). In these operations, we will be shifting our dividend and divisor into a fixed-point number where the divisor is less than 1. For instance, if we were performing the division $(15/3)$, our inputs would be (1111_2) and (0011_2). This will be converted into $(11.11_2 = 3.75)$ and $(0.11_2 = 0.75)$. The result of this division $(3.75/.75 = 5)$ is the same product and quotient as $(15/3)$. The importance of this shift into a fixed-point number less than 1 will be discussed further in the following sections.

According to Obermann and Flynn, there are 4 common ways to implement HW division in a binary system [5]. The four categories they propose are:

1. Digit recurrence [6]
2. Functional iteration [7]
3. Very high radix division [8]
4. Variable latency algorithms [9] [10]

Their review also investigates some common HW division optimization approaches. Of these categories, items 1-3 are all unique implementations of division, and the fourth item is a type of optimization able to be applied to any division algorithm. We will now discuss the differences between methods 1-3, and explain how they function. We will also discuss variable latency algorithms briefly and their contribution to optimizing the other three algorithms. When discussing these algorithms, we will use Big O notation [11] to describe the computational complexity or speed of these algorithms.

2.2 Digit Recurrence

Digit Recurrence uses repeated subtraction of the divisor from the dividend to find the quotient and remainder, very similar to how long division is done with pencil and paper. The algorithmic efficiency of digit recurrence is $O(n)$ where n is the bit-width of the divisor. Due to its simplicity and extremely small area footprint, it is attractive as a division solution. However, the simplest digit recurrence algorithms do not perform well enough for modern computing systems. Figure 2 is a simple example of digit recurrence division. In this example the dividend is 153_{10}

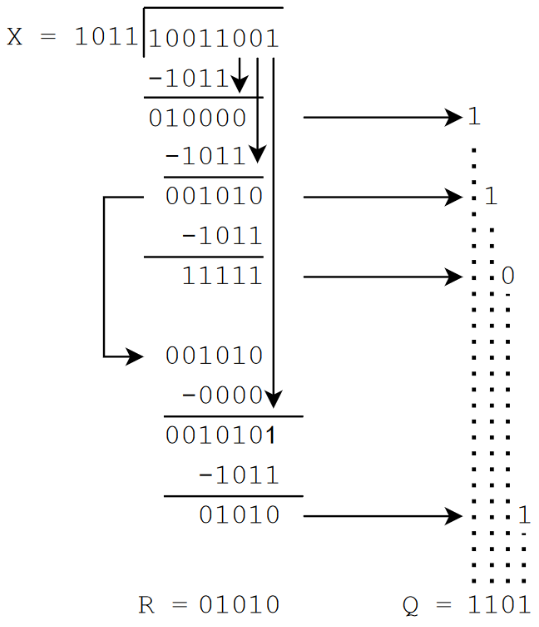


Figure 2: Digit Recurrence example

(10011001_2) and the divisor is 11_{10} (1011_2). Exactly like long division in base 10, we are subtracting a shifted version of the divisor from the dividend, and then bringing down another digit after the subtraction takes place. If the result of the subtraction is positive, we add a 1 to the quotient. If the result of the subtraction is negative, we add a 0 to the quotient and restore it to the previous number. Once we have done a subtraction down to the one's place, we have a quotient and remainder. In this case, $(153/11 = 13r10)$.

There are several methods to implement digit recurrence, such as restoring, non-restoring and SRT division (SRT stands for Sweeney, Robertson, and Tocher who all independently discovered the algorithm [12] [13] [14]). The example shown in Figure 2 is known as restoring division. In practical applications, SRT is by far the most commonly used due to its ability to increase its radix and be pipelined. This method works only with floating point numbers, and it functions as follows: Repeated subtractions are performed as normal, and due to the floating number representation, we can determine if the result is $(> 1/2)$, $(< -1/2)$ or in between $(1/2)$ and $(-1/2)$. Much like how we assign values to the quotient based on whether the result of the subtraction was negative or not, we will either add 1, 0, or $\overline{1}$ (-1) to the quotient based on the comparisons as mentioned above. Then we shift the result of the subtraction and compare again. We continue this shifting (which is equivalent to a multiplication by 2) until the number becomes $(>= 1/2)$, in which case we add a 1 to the quotient, and do another subtraction. Figure 3 is an example of SRT division, which performs $(63/256) = 0.00111111_2$ divided by $(9/16) = 0.1001_2$ resulting in $(7/16) = 0.0111_2$ and no remainder. This SRT division would be said to have a radix of 2. The radix number for SRT division is determined by raising 2 to the power of the number of quotient bits chosen in each iteration.

2.3 Higher Radix Algorithms

SRT division can have a higher radix, which is the method of computing more bits of the quotient

$r_0 = X$		0	.0	0	1	1	1	1	1	1	
$2r_0$		0	.0	1	1	1	1	1	1	0	$< 1/2$ set $q_1 = 0$
$2r_1$		0	.1	1	1	1	1	1	0	0	$\geq 1/2$ set $q_2 = 1$
Add $-D$	$+$	1	.0	1	1	1					
r_2		0	.0	1	1	0	1	1	0	0	
$2r_2$		0	.1	1	0	1	1	0	0	0	$\geq 1/2$ set $q_3 = 1$
Add $-D$	$+$	1	.0	1	1	1					
r_3		0	.0	1	0	0	1	0	0	0	
$2r_3$		0	.1	0	0	1	0	0	0	0	$\geq 1/2$ set $q_4 = 1$
Add $-D$	$+$	1	.0	1	1	1					
r_4		0	.0	0	0	0	0	0	0	0	zero final remainder

Figure 3: Radix-2 SRT Division performed with dividend = 0.00111111_2 and divisor = 0.1001_2. The quotient results in 0.0111_2 with a remainder of 0.

at each cycle. For instance, the simplest higher radix algorithm is Radix-4 SRT division which computes 2 bits of the quotient every iteration, which roughly halves the time for the division to compute. Radix-8 SRT division would compute 3 bits of the quotient at once, and so on. In our Radix-2 SRT division example, there were 2 cutoff lines line at $1/2$ and $-1/2$. When we increase the radix, we need to have more cutoff lines, and they become much more complex.

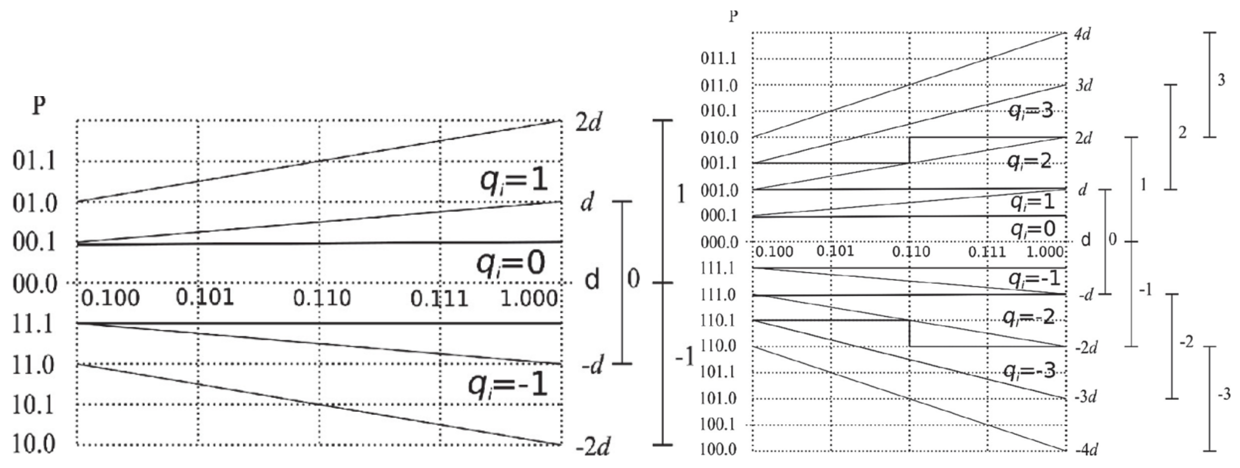


Figure 4: P-D plot for Radix-2 division and Radix-4 SRT division.

Figure 4 shows a graphical representation of the cutoff lines where a cutoff line is defined for the partial remainder and divisor. This graph shows partial remainder on the y-axis versus divisor on the x-axis and is called a P-D plot. In the Radix-4 division example pictured on the right in Figure 4, a partial remainder of 001.1_2 with a divisor of 0.111_2 is within the $q_i = 2$ area. This indicates that when we are performing a division, if our divisor is 0.111, and we find a partial remainder of 001.1_2, we need to add a 2 to our next quotient bits. Since this form of radix adds 2 bits at each iteration, setting our quotient bits to 2 equates to assigning 10_2 to that part of our quotient. When implemented in hardware, a look-up table has to be built, which gives instructions for what the next quotient bit is depending on the divisor and partial remainder. When building the lookup table, we can use comparison logic to check whether the partial remainder is greater than or less than a cutoff based on the divisor. The cutoffs must stay within the sloped lines, which follow multiples of the slope d. The right of each section of the figure shows the ranges where it is acceptable to select each part of the quotient. For instance, on the right graph if the divisor is 1, and the partial remainder is anywhere between $2d$ and $4d$, you are allowed to either select 2 or 3. The cutoff line for selecting a 3 in this plot is made above $2d$ and below $3d$, and it must stay within the range for all divisors d. Pictured on the left of Figure 4, we use a horizontal cutoff. The horizontal line is easy to implement and fits completely within the range as defined by the sloped lines. For higher radix, a stair-step technique has to be used to set the thresholds of the look-up table accurately. This causes the look-up tables of larger radix divisions to be much larger in the FPGA area when implemented in HW.

The advantage of the SRT method of division is the ability to increase the radix. A Radix-4 divider converts the dividend and divisor into base 4 and does the same process. For the Radix-4 division, you must determine the quotient bit to be between -3 to 3 rather than -1 to 1. This process can be scaled up to any level of radix, as long as the number of bits in the dividend is larger than the radix demands. This type of algorithm is used very often for processors because of its speed, reliability, and ability to be very easily pipelined, and processors typically have 32 and 64-bit representations. However, the Big O efficiency of this algorithm is still $O(n)$, even though a Radix-4 divider is twice as fast as a Radix-2 divider (and Radix-8 is 3x as fast, and so on). As you increase the radix, the lookup table also becomes extremely large, which causes Radix-16 to be the highest radix people commonly implement. In common Radix-16 algorithms, two Radix-4 dividers are being run in parallel to determine the four bits of the quotient, each of them computing two bits each[15]. This method helps circumvent the need to build a larger lookup table.

2.4 Functional Iteration

Functional Iteration is a division method that is usually exemplified by the Goldschmidt or Newton-Raphson algorithms [16] [7]. These algorithms use multiplication to estimate the quotient, and each iteration improves the accuracy of the answer. They approach the exact answer quadratically, but in theory, they will never achieve the exact answer due to the nature of the algorithm. The longer the algorithm runs, the more significant figures are confirmed to be correct in the estimation. The efficiency of this algorithm is $O(log(n))$, which makes it better than recurrence algorithms from an algorithmic speed standpoint. In our research, this will result in less number of clock cycles to complete the calculation.

The Goldschmidt method of division multiplies the numerator and denominator iteratively by factors that cause the denominator to approach 1. As the denominator approaches 1, the numerator will approach the quotient if it is being multiplied by the same values. The values to be multiplied to the numerator and denominator are calculated by $(2 - D_i)$ where D_i is the value of the denominator at iteration i. The example below shows a division using the Goldschmidt algorithm. In this example, the answer is correct to four decimal places after only three iterations. $R(N)$ is the value being multiplied to the denominator for the Nth iteration, and $D(N) = D * R(0) * R(1) * ... * R(N-1)$

$N = 0.0110100000000000_2 = 0.40625_{10}$
$D = 0.1100000000000000_2 = 0.75_{10}$

$R(0) = 2 - D = 1.010_2$
$D(1) = D * R(0) = 0.11110_2$
$R(1) = 2 - D(1) = 1.00010_2$
$D(2) = D(1) * R(1) = 0.111111110_2$
$R(2) = 2 - D(2) = 1.000000010_2$
$D(2) = D(2) * R(2) = 0.11111111111_2$
$Q = N * R(0) * R(1) * R(2) = .1000101010101_2$
$Q = 0.54165_{10}$
$0.40625_{10}/0.75_{10} = 0.5416666_{10}$

The Newton-Raphson method follows a similar process, except it approaches the inverse of the denominator, and multiplies the inverse of the denominator with the numerator. The algorithm for this method is $x_{i+1} = x_1(2 - D * x_i)$, where i is the number of iterations, and D is the denominator. After multiple iterations, x_i will approach the inverse of the denominator. These two algorithms are based on the same mathematical principle but have different implementations.

2.5 Very High Radix

As you increase the radix of the SRT division, the quotient digit selection becomes very expensive in HW area. Because of this, selecting around 10+ digits at a time is considered very high radix, and there are various alternative methods to perform very high radix division. Wong explores a very high radix algorithm that calculates sections of the quotient using truncated versions of the dividend and divisor [8]. This method is iterative and approaches a quotient very similar to the Newton method of division. The method can be equated to performing a functional iteration algorithm on small chunks of the division at a time. Many very high radix algorithms have been found that follow a similar method of iteration, but modify either early steps by using lookup tables [17] [18] or make optimizations in rounding and prescaling [19] [20].

Due to the ability to find a very large number of bits at a time, very high radix algorithms could be an effective solution for wide-width dividers. However, the investigation of very high radix division algorithms is out of the scope of this work.

2.5 Variable Latency

Variable latency algorithms are extra HW modules built into dividers to lower the average latency of each step. One example of optimization is to have a cache hold the value of previous divisions, and whenever a division command is executed, the cache is checked for the answer [10]. This method is useful if your processor is running the same divisions multiple times in a row.

Another method of variable latency is to speculate quotient digits [9]. This speculation is done by passing incomplete remainder and divisor information into high radix SRT division. It turns out this method provides a higher average speed, even when the speculation is incorrect, and the division must be re-done.

These optimizations are not implemented in our HW designs because we are only trying to investigate the general trends of each algorithm about bit-width rather than trying to build an optimal divider for large widths.

2.6 FPGA Studies of Division

The different types of division, and their algorithms have already been discussed, so this section will mention previous work that has analyzed division algorithms, or implemented them on an FPGA.

The major research works on FPGA division include the following:

- Sutter et. al. compared different functional iteration algorithms on an FPGA, such as non-restoring division, Radix-2, Radix-4, Radix-8, and Radix-16 SRT division among others [21].
- Khan investigated a fixed iteration algorithm, similar to SRT, and an algorithm utilizing variable latency optimizations that greatly reduce the number of iterations [22].
- Matthews et. al. designed a variable latency algorithm called Quick-Div and compared it to Radix-2/4/8/16 SRT algorithms implemented on the FPGA [23]. Much like our work, Matthews et. al. implemented all of these algorithms on a soft processor to analyze speed and timing constraints. Their work used a small suite of benchmarks for division, which differ from the benchmarks we use. In their case, they were testing for 32-bit divisions only, and due to variable latency algorithms implemented into Quick-Div, they had to test realistic applications and account for best and worst-case scenarios.

All of this research either offers something new to the exploration of dividers on FPGAs or focuses on improving the efficiency of existing division algorithms. Our work aims to add to this field of research by not trying to optimize or improve any algorithms but rather analyzing the general trends of these algorithms as the width of the division increases. This direction of research has never been explored for FPGAs. Additionally, we provide means to replicate HW division to understand the approaches in detail.

3. Methodology and Tools

This section has two parts that will explain the methodology of this work. Section details all of the steps that a Python script we built performs to run tests, collect data on our divider implementations, and provide examples of division for other learners. Section provides a more detailed explanation of the division algorithms that we implemented, and discusses some design choices that we made for each divider.

3.1 Python Script Workflow

We aim to build HW division modules with varying bit widths using different division algorithms. To accomplish this, a script was created in Python which can build and test a divider of any implemented algorithm and any bit-width. The script creates a Verilog HDL file which is an implementation of the chosen divider at a given bit-width. This file is incorporated into a soft processor and analyzed for FPGAs via the Quartus tool.

Figure 5 shows a summary view of our script. On the left of this figure are the inputs into the script which include the division algorithm and width for the divider. The script uses these inputs to generate the divider, as well as a testbench to be used for simulation. The divider gets compiled as a unique Verilog module in Quartus to obtain metrics on the FPGA area and critical path [3]. The divider is also attached to the Trireme processor [4]. This processor, along with the custom divider, gets synthesized by Quartus to test and verify accuracy using ModelSim. The simulation collects measurements for the number of clock cycles the divider takes in execution as well as checks the correctness of division answers generated by the custom HW. We will explain the relevant details of this flow in the remainder of this section.

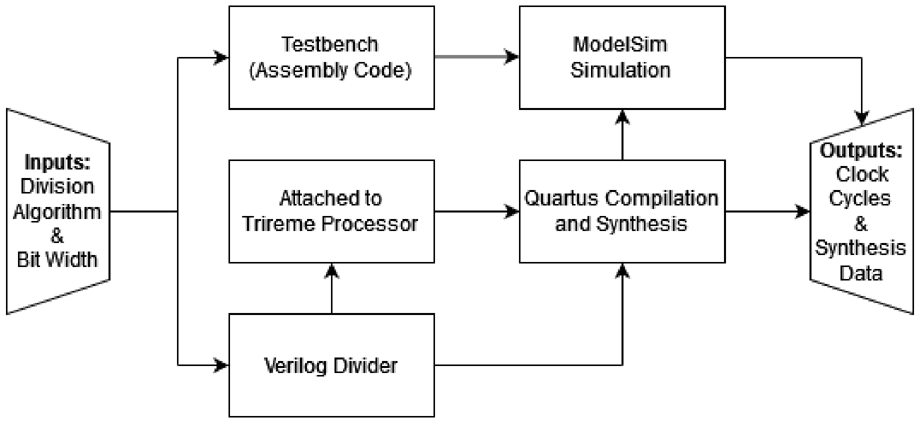

Testbench (Assembly Code)
ModelSim Simulation
Inputs: Division Algorithm & Bit Width
Attached to Trireme Processor
Quartus Compilation and Synthesis
Outputs: Clock Cycles & Synthesis Data
Verilog Divider

Figure 5: A diagram of our Python script workflow.

In our data collection, the Trireme soft processor [4] includes a divider and divider wrapper module attached to it. The script will rebuild the divider module according to the method of division, as well as edit the divider wrapper according to the width of the divider. The main processor connects to the wrapper via memory mapping. Memory mapping specifies a specific address section of memory that rather than storing or loading information from and to the RAM, instead accesses different modules in the processor. Our modified processor catches memory instructions targeted to the specific divider address range and reroutes those store operations to the divider wrapper for the division. Once all of the bits of a wide division are collected, the division algorithm starts.

The script automatically writes the assembly code needed to test the custom divider and compiles it into a form that the soft processor can use. This code is a series of specific store instructions in memory that will load the data into the wrapper. The Trireme processor needs instructions in a specific format for simulation, and the script converts the written code into that form. These instructions load a series of random bits into the processor for division by default, but any numbers can be loaded to test the dividers if desired.

Once the divider and wrapper are created, our entire design, which includes our HW divider, the wrapper, and the edited Trireme processor will be synthesized with the Quartus tool. We also compile just the HW divider in its module to get an accurate measure of speed and area for just the divider. This will give us a result for the area of this divider, as well as the critical path for the divider implementation. The synthesis of the entire project is needed for ModelSim to carry out the simulation and test the divider. The ModelSim simulator will simulate the testbench written for it and check for correctness. The number of clock cycles to complete the HW division is recorded from this simulation.

In our experiments, our division algorithms will be run for bit-widths starting at 8 bits up to 1024 bits. Not every width between 8 and 1024 will be tested, rather enough points for an accurate trend line will be sufficient.

The two metrics we are measuring are the area of the designs and the speed of the division. For our designs, no variable latency algorithms are implemented, which means our algorithms will have a predictable run-time for every division. For instance, a non-restoring division algorithm will always take n clock cycles where n is the bit-width of the dividend. The number of cycles needed for the Newton and Goldschmidt methods is predetermined, as each iteration correctly identifies twice as many bits as the previous iteration (also known as quadratic convergence) [24]. The equation that we used to predetermine the number of iterations is $\lceil \frac{log_2(n+1)}{log_2(17)} \rceil$ where n is the width of the divider. This will ensure that we stop the algorithm after calculating enough significant digits. The division operations tested in the testbench will be random, as our most likely use cases for wide dividers would more closely resemble random strings. Most of our dividers have a consistent amount of iterations per width due to their lack of variable latency algorithms, so we do not feel the need to run multiple divisions per algorithm per width. The only dividers that take a variable amount of time are the SRT dividers due to their normalization steps. Since a random string of bits will have on average 1 normalization step, keeping all of the testing divisions random will give a somewhat uniform performance for any width. After measuring the clock cycles, area, and critical path of each divider, we were able to compile our results.

3.2 Division Algorithms

The previous section already discussed how the different division algorithms in this work function. However, when implementing these algorithms on FPGAs, some details might differ from how the theoretical methods are reported in the literature. This section walks through our implementations of each of the dividers and provides any important details about our dividers not discussed in the background section.

3.2.1 Non-Restoring Divider

The non-restoring divider is the most simple of our dividers. It works as described in the background section. First, the algorithm initializes the partial remainder of the dividend. Every iteration, it will check the Most-significant bit (MSb) of our partial remainder. If the MSb is 1, then it will add the divisor to the partial remainder and assign a 0 to the most significant unassigned bit of the quotient. The partial remainder will then shift left by 1, and the process repeats for n iterations (where n is the width of the divider). Once this is done, the quotient will have n bits assigned to it, and the final partial remainder is the remainder of the division. If the final remainder's MSb is 1, then the divisor needs to be added to it one last time, as a correction step.

3.2.2 Radix-2 SRT Divider

For our Radix-2 SRT divider, there are many details important for implementation that are not discussed in theoretical explanations of the algorithm. The most important aspect is that the divisor is half the width of the dividend. In our analysis, if the divider is 32 bits that means our dividend is 32 bits and our divisor is 16 bits. That important difference and its implications will be discussed further in the results section of this paper.

To perform SRT division, the divisor must be between $1/2$ and 1, which requires a normalization step. Our divisor and dividend are loaded as integer numbers, so we need to place an assumed decimal place, and shift the divisor to be within that range. In our implementation, the normalization algorithm is done within the division module, and it can shift up to 16 places at once. The amount that the number gets shifted depends on the amount of leading zeros. Take for example a 16-bit divider module. This means our divisor is going to be 8 bits long. If the divisor is 00000111_2, we would shift the number until we got 0.11100000_2. In this case, the divisor gets shifted left by 5 spots which equates to a multiplication of 32. Because of this, at the end of the division algorithm, there is a shift to the right by 5 which equates to a division of 32. Due to the transitive property of multiplication and division, this process will preserve the correct answer.

The number of iterations our Radix-2 SRT division takes is also dependent on how much the number is shifted. More specifically, the amount of iterations is dependent on the difference in the number of bits between the divisor and dividend minus 2.

Take for example a dividend of 10110101_2 and a divisor of 0011_2. This is an 8-bit division where the divisor is half the width of the dividend, but the divisor has two leading zeros. Since we shift the divisor to get rid of the leading zeros, the algorithm must run for two more iterations. According to our algorithm, this division would take 4 clock cycles. If the divisor was 0110_2 instead, the division would only take 3 clock cycles. This is an essential part of the implementation of this algorithm, because after normalization both 0110_2 and 0011_2 become 0.1100_2, which would result in the same answer if we ran the algorithm of the same amount of iterations. This is part of what gives SRT an advantage over non-restoring division because it's as if we are skipping the first two iterations by removing the two leading zeros in normalization (those iterations in the non-restoring division would only result in setting the quotient bit to zero and shifting left by 1).

The method to calculate the quotient we used was the on-the-fly conversion algorithm [25]. This means we do not store the string of 1, 0, and $\overline{1}$ and convert it at the end. Instead, when we found the digit for a specific location we added it to the quotient during that iteration. For example, if our quotient was $10\overline{1}1_2$, we would have found the most significant bit first and added it shifted left by 3 making our initial quotient 1000_2. The next iteration was zero so no addition is made. The next iteration is $\overline{1}$, so we add -1 shifted left by 1: $1000_2 + 1110_2 = 0110_2$. Finally, the last digit is 1 so we add one to get our final quotient: $0110_2 + 0001_2 = 0111_2$. This is normally done with a carry-save adder, which will decrease the critical path of the addition operations as compared to a traditional adder [26, 27, 25]. However, we chose to implement this using traditional adders for two main reasons. First, we were aiming to create simplified versions of these dividers, not striving to implement common optimizations, because we are only looking for general trends of the algorithm, rather than the performance of specific examples of these methods. Second, our implementations are on FPGAs which are very well optimized for traditional adders, and we hypothesize that the FPGA will have an easier time fitting a traditional adder of that size rather than a custom-built carry-save adder (however, this is left to others if they choose to implement these dividers for their work).

Another detail of Radix-2 SRT is the correction step at the end of the division. We already discussed the shift to the right corresponding to the normalization, but there is an extra correction that needs to be done if the final remainder is negative. In the negative case, you need to add the divisor to the remainder just like we do in the non-restoring division, and you need to subtract 1 from the quotient. This correction accounts for the error that can be present in the selection between -1, 0, and 1 in the quotient bits. If the final remainder is negative, it means the final iteration was within the range where two options were possible. The higher of the two options is selected incorrectly. This simple correction step allows for the algorithm to function correctly.

3.2.3 Radix-4 SRT Divider

The Radix-4 SRT divider is very similar to the implementation of the Radix-2 SRT divider. Our implementation uses the same normalization HW, and it also uses an on-the-fly conversion algorithm. The correction algorithm for when the final remainder is negative is the same as well. There are, however, details to be discussed that are unique to our Radix-4 SRT implementation.

An important detail of the Radix-4 SRT division is the specific P-D plot used. Since there is some flexibility with how you graph your P-D plot we think it is important to share what design we decided to use. We modeled our P-D plot after a plot with a quotient set from [-2, 2] from Behrooz Parhami's book on computer arithmetic [28], and it can be seen in Figure 6.

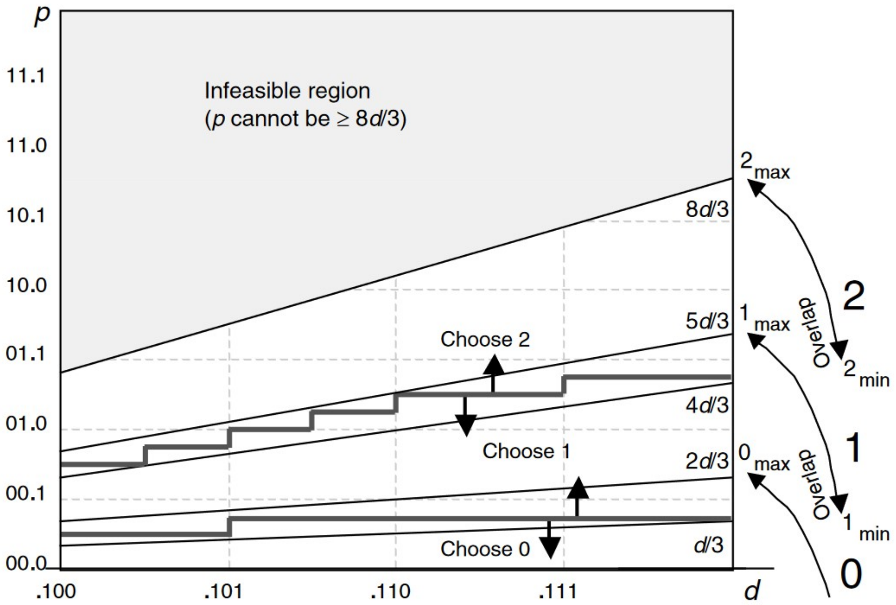


Figure 6: P-D plot from Computer Arithmetic [28] which we used for reference to create our P-D plot.

Since there are two bits of the quotient being decided in one iteration, the Radix-4 SRT algorithm shifts the remainder by 2 every iteration. This can occasionally cause alignment problems, which we need to address in the normalization. Consider the example where the divisor is 0110_2. After normalization, the divisor will get normalized to 0.1100_2, and we keep track of how far we normalized to know the location of our first digit of the quotient. Now consider the example where the divisor is 1100_2, which is the same as the previous example shifted to the left by two. The divisor will again get normalized to 0.1100_2, but the amount it was normalized is 1 less. Due to the encoding, each iteration will compute one digit of the quotient, so the last four iterations of the second example would compute Q[7,6], Q[5,4], Q[3,2], and Q[1,0]. Since the first example was shifted one more bit in normalization, the last four iterations are computing Q[8,7], Q[6,5], Q[4,3], and Q[2,1]. This will omit the last bit, and cause an incorrect answer. Because of this if the normalization of the divisor is shifted by an odd number, then we shift the dividend left by 1 and we assign the bits of the quotient shifted one to the right.

3.2.4 Goldschmidt Divider

Our Goldschmidt divider also requires steps for normalization, which we carry out using the same normalization hardware as the SRT dividers. To perform the multiplications, internal registers are twice the width of the divider. For example, in an 8-bit divider, the internal registers are 16-bit wide because an 8-bit number multiplied by another 8-bit number creates a 16-bit number. As mentioned previously, each iteration is calculated by $(2 - D) * D$ where D is the previous iteration of the divisor. Each iteration also gets multiplied by the dividend. To perform that multiplication, we do not truncate any bits of D, which means our quotient is 3 times the width of the divider. In an 8-bit example, D would be something like 00.11111101011001_2, and it would be multiplied by the dividend which would be something like $00000111.0011100001101101_2$. Once the algorithm finishes, we take the 8 most significant bits of the dividend and truncate all values after the decimal place.

4. Results for Increasing Width of Division

This section reports our results obtained through the testing of four division algorithms: Non-restoring, Radix-2 SRT, Radix-4 SRT, and Goldschmidt division. For each of these algorithms, we ran our analysis for bit-widths from 8 bits to 1024 bits. Not every width in that range is analyzed, and as the width increases, the gap between tests increases. The reason for this is to get a more accurate curve while the width is small, where someone is more likely to make a divider, and as the width increases we can lower the granularity of our data and still understand the general trends. Section will discuss the area of each divider, and Section will discuss the speed of each divider.

An important note to make is the range of our data for each divider. For all of our dividers, the largest width we could synthesize was 1120 bits, no matter the method. This limitation is most likely due to the FPGA's ability to connect two ALMs during the routing process, where a wire with a width larger than 1120 cannot be connected between two ALMs. The data we report only goes up to a maximum bit-width of 1024, so this limitation is not reflected in our graphs. Also, the Goldschmidt divider has a smaller range than the other dividers because it was not able to synthesize above a width of 244. This is due to the limited number of DSP blocks.

4.1 Area

The FPGA used in these tests is the 5CGXFC9E7F35C8 from the Cyclone V line. This FPGA is chosen due to its large amount of available ALMs and DSP blocks. The maximum ALMs that our

FPGA has in this study is 113,560. Very few dividers in this study approached this maximum number of ALMs as many other reasons caused placement to fail before this limit was reached (such as the inability to connect a wire of width larger than 1120 between ALMs). This FPGA also has 342 DSP blocks which our Goldschmidt divider approached around the width of 244. Running out of DSP blocks and the difficulty routing between DPS blocks with such a large utilization are the major factors that caused our Goldschmidt divider to not be able to synthesize beyond the width of 244.

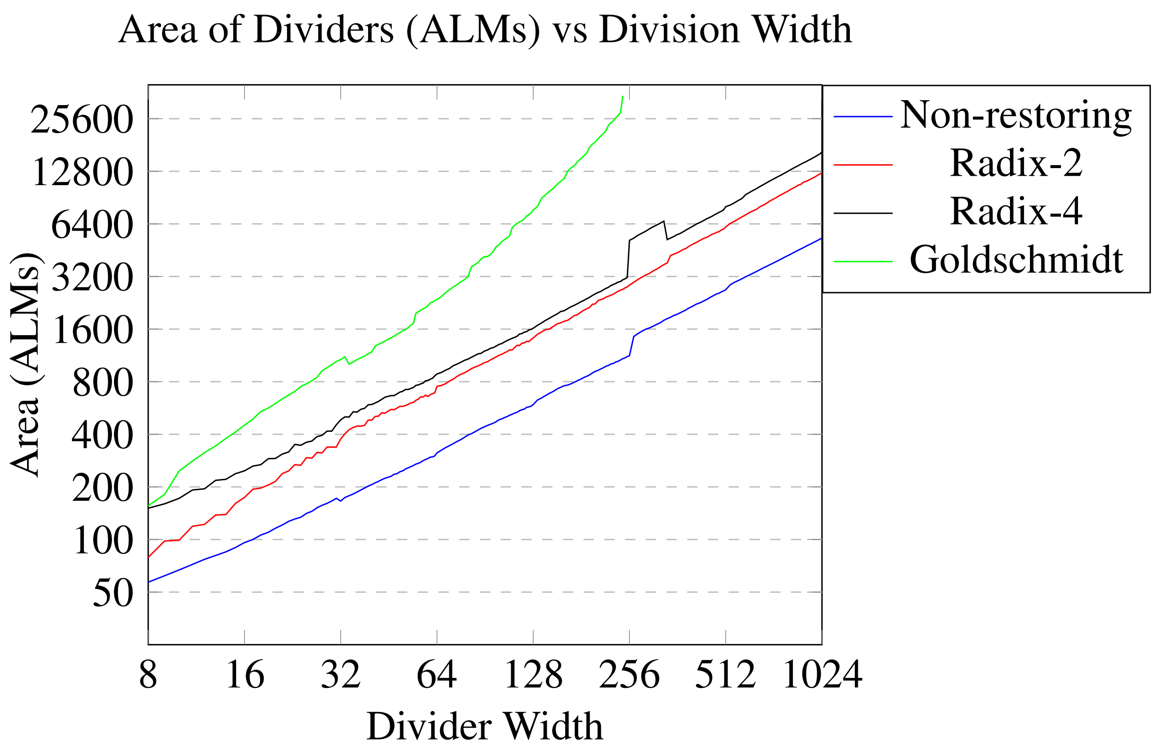


Figure 7: The area of dividers (ALMs) vs the bit-width of dividers. Both the x and y axis of this plot are logarithmic to show finer detail at lower bit-widths and maintain the linearity of the trend lines.

Figure 7 shows the number of ALMs each divider took to synthesize. This metric shows roughly the area each divider would take at each bit-width. We will use this graph format for all of our results and note that the y-axis will change depending on the results. In this case, the x-axis is the bit-width, and the y-axis shows the number of ALMs used. Both the x and y axis are on a logarithmic scale. Each of the lines plotted on the graph will have the same color scheme where non-restoring is blue, Radix-2 SRT is red, Radix-4 SRT is black, and Goldschmidt is green.

There are multiple things to note about Figure 7. First off, the Radix-2 SRT divider and Radix-4 SRT divider are very similar in size, with the main difference between them being a larger look-up table based on the P-D plot and larger multiplexers for decision-making in bit selection. Their area is a little over twice as large as the area of the non-restoring division for any given width. The area of the Goldschmidt division increases at a much faster rate than all the digit recurrence algorithms.

Also in Figure 7 there are some anomalies in the graphs. In each trend line, there is a sudden change at 32 bits, and in non-restoring and Radix-4 SRT, there is a sudden change at 256 bits. This is likely due to the optimization tools in the Quartus tool flow. Since 32 bits is a common width for calculations we suspect that Quartus can better optimize the design when dealing with numbers at or greater than 32 bits. In particular, the Quartus architecture is designed with 8, 16, and 32-bit computations in mind. There are also jumps up at 64, 128, and 256 bits, which provides more evidence that the architecture is designed for certain bit widths. There is also an anomaly in the Radix-4 SRT division's area starting at a width of 256 and continuing until a width of 328. We do not have an explanation for this sudden bump in our graph, or the drop in area around the width of 328.

We have reported how many ALMs are in use in the dividers to represent the size of each divider, but our Goldschmidt divider also utilizes DSP blocks in its design. As mentioned earlier, the DSP blocks have a set width, and any given width of the divider would use the next largest DSP block to perform the multiplication portion of the algorithm. The number of DSP blocks increases as the width of the Goldschmidt divider increases, and the maximum usage of DSP blocks for the chosen FPGA occurs at around a width of 244 due to all the DSPs being used. For FPGAs without DSP blocks, the cost of the area is going to suffer even further [29] [30] [31]. We estimated that each DSP block costs around 150 ALMs. The performance and cost of the Goldschmidt divider are significantly worse without the utilization of DSP blocks

4.2 Speed

This section will discuss how fast each divider can complete the division. We will be analyzing the maximum clock frequency of each divider as well as how many clock cycles the divider takes to complete the division. These two metrics are used to estimate the general speed of each divider, which allows us to conclude their effectiveness.

In terms of cycles, all digit recurrence algorithms have a linear relationship between divider width and clock cycles. Radix-2 SRT division takes half as many clock cycles as the non-restoring division. Algorithmically these two methods should have the same amount of cycles to complete, but in our definition of the dividers, the bit width of the divider refers to the width of the dividend, and the Radix-2 SRT divider's divisor is half the width of the dividend. This means that on average our Radix-2 SRT divider took half the amount of clock cycles as the non-restoring divider. Our Radix-4 SRT divider takes half as many clock cycles as the Radix-2 divider because it uses the same algorithms but selects twice as many bits per cycle. The Goldschmidt divider lies on a logarithmic curve, and as the divider width increases, the gap between it and the digit recurrence algorithms grows larger. At our largest size of the Goldschmidt divider, it takes one-fourth the number of clock cycles as the Radix-4 SRT divider.

To explore what the best-performing divider was for speed, we created a theoretical lowest possible time for division. This number is created by converting our maximum clock frequency into the minimum possible period and multiplying it by the number of clock cycles for that width. Our results are shown in Figure 8. At very low bit-widths, the SRT algorithms are the best and they are comparable in speed. As the bit-width increases, the Radix-4 SRT divider is the best divider in terms of requiring the least time per division. Also of note is the time for the

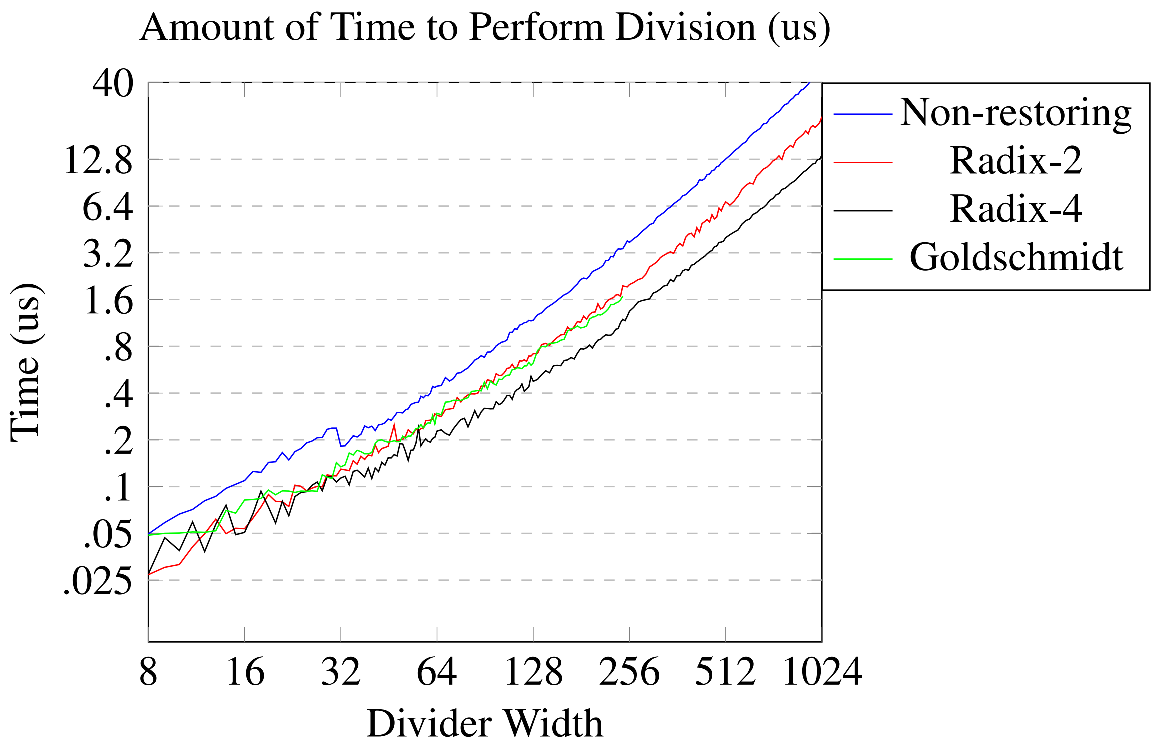


Figure 8: Amount of time in the theoretical best case scenario for each division operation. The time for division is calculated by $1/[clockfrequency] * clockcycles$. These results are under the assumption that the divider is being clocked at its maximum frequency. Both the x and y axes are logarithmic to help show the linearity of these trends as well as give finer detail of the low widths.

Goldschmidt divider, which follows the Radix-2 divider very closely. Even though the Goldschmidt divider has a logarithmic amount of clock cycles as compared to the linear clock cycles of Radix-2 SRT, the design also decreases its maximum clock frequency at a shockingly similar rate. This results in the Goldschmidt divider performing comparably to the Radix-2 SRT divider no matter the width.

All of this data points to the Radix-4 SRT divider being the best method of division in most cases. However, there are many situations where Radix-2 SRT or non-restoring division would be desirable due to size or timing constraints. section will explore further the considerations of which divider to use in any given application.

5. Discussion and Conclusion

Digital divider implementation is not a new topic, but in our exploration in this space, the main idea we discovered is that the details on actual working implementations are either hidden in the mathematical description or non-existent due to the demonstration example selected (that seems to propagate into the education space as these examples are difficult to create). For this reason, we have replicated much of this work with a new focus on evaluating different division implementations as bit-width increases, and we have added the key feature to our open-source work (`https://github.com/marti693/Variable-Width-Division-Scripts`) where for each of our division implementations, the scripts generate a step-by-step output of an example division for any numbers.

From a collective understanding of division implementations, as well as other digital function designs such as the Cordic cores, floating-point cores, etc. we believe that this approach makes the contribution much more significant. We argue that the research and development open community should include a feature that outputs examples such that learners and other designers can both evaluate the design and see how the system works. We have been pushing academics to release digital application work as open-source since we don't believe new ideas and research have value if the artifact is not available to the community. Going forward, we believe an exemplar open-source generator should be included in contributions and will have similar value to the community. In particular, this will provide learners with examples that will help them understand the design of these systems.

The wide division work presented in this paper is also a small contribution to research. The main conclusion from this aspect of the work shows that high-radix dividers (as currently exist in modern processors) are the main winners in terms of speed and area for division implementation. Going forward, the challenge in a wide division is automating the SRT high-radix generation into the 32-, 64-, and beyond radixes.

References

[1] M. Hellman *et al.*, "New directions in cryptography," *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*, 1976.

[2] R. S. Katti and R. G. Kavasseri, "Secure pseudo-random bit sequence generation using coupled linear congruential generators," in *2008 IEEE International Symposium on Circuits and Systems*. IEEE, 2008, pp. 2929–2932.

[3] Altera, *Quartus II Handbook, Volumes 1, 2, and 3*, 2004.

[4] E. Gur, Z. E. Sataner, Y. H. Durkaya, and S. Bayar, "Fpga implementation of 32-bit risc-v processor with web-based assembler-disassembler," in *2018 International Symposium on Fundamentals of Electrical Engineering (ISFEE)*, 2018, pp. 1–4.

[5] S. F. Obermann and M. J. Flynn, "Division algorithms and implementations," *IEEE Transactions on computers*, vol. 46, no. 8, pp. 833–854, 1997.

[6] D. E. Atkins, "Higher-radix division using estimates of the divisor and partial remainders," *IEEE Transactions on computers*, vol. 100, no. 10, pp. 925–934, 1968.

[7] M. J. Flynn, "On division by functional iteration," *IEEE Transactions on Computers*, vol. 100, no. 8, pp. 702–706, 1970.

[8] D. C. Wong and M. J. Flynn, "Fast division using accurate quotient approximations to reduce the number of iterations," in *Proceedings 10th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society, 1991, pp. 191–192.

[9] J. Cortadella and T. Lang, "Division with speculation of quotient digits," in *Proceedings of IEEE 11th Symposium on Computer Arithmetic*. IEEE, 1993, pp. 87–94.

[10] S. E. Richardson, "Exploiting trivial and redundant computation," in *Proceedings of IEEE 11th Symposium on Computer Arithmetic*. IEEE, 1993, pp. 220–227.

[11] D. E. Knuth, “Big omicron and big omega and big theta,” *ACM Sigact News*, vol. 8, no. 2, pp. 18–24, 1976.

[12] O. L. MacSorley, “High-speed arithmetic in binary computers,” *Proceedings of the IRE*, vol. 49, no. 1, pp. 67–91, 1961.

[13] J. E. Robertson, “A new class of digital division methods,” *IRE transactions on electronic computers*, pp. 218–222, 1958.

[14] K. D. Tocher, “Techniques of multiplication and division for automatic binary computers,” *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 11, no. 3, pp. 364–384, 1958.

[15] T. M. Carter and J. E. Robertson, “Radix-16 signed-digit division,” *IEEE transactions on Computers*, vol. 39, no. 12, pp. 1424–1433, 1990.

[16] R. E. Goldschmidt, “Applications of division by convergence,” Ph.D. dissertation, Massachusetts Institute of Technology, 1964.

[17] W. Briggs and D. W. Matula, “A 17/spl times/69 bit multiply and add unit with redundant binary feedback and single cycle latency,” in *Proceedings of IEEE 11th Symposium on Computer Arithmetic*. IEEE, 1993, pp. 163–170.

[18] D. Matula, “Highly parallel divide and square root algorithms for a new generation floating point processor,” in *SCAN-89, International Symposium on Scientific Computing, Computer Arithmetic, and Numeric Validation*, 1989.

[19] M. D. Ercegovac, T. Lang, and P. Montuschi, “Very high radix division with selection by rounding and prescaling,” in *Proceedings of IEEE 11th Symposium on Computer Arithmetic*. IEEE, 1993, pp. 112–119.

[20] P. Montuschi and T. Lang, “Boosting very-high radix division with prescaling and selection by rounding,” *IEEE Transactions on Computers*, vol. 50, no. 1, pp. 13–27, 2001.

[21] G. Sutter, G. Bioul, and J.-P. Deschamps, “Comparative study of SRT-dividers in FPGA,” in *International Conference on Field Programmable Logic and Applications*. Springer, 2004, pp. 209–220.

[22] S. Khan, “Vhdl implementation and performance analysis of two division algorithms,” Ph.D. dissertation, Sir Syed University, 2015.

[23] E. Matthews, A. Lu, Z. Fang, and L. Shannon, “Rethinking integer divider design for fpga-based soft-processors,” in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 289–297.

[24] G. Even, P.-M. Seidel, and W. E. Ferguson, “A parametric error analysis of goldschmidt’s division algorithm,” *Journal of Computer and System Sciences*, vol. 70, no. 1, pp. 118–139, 2005.

[25] M. D. Ercegovac and T. Lang, *Digital arithmetic*. Elsevier, 2004.

[26] J.-P. Deschamps, G. D. Sutter, and E. Cantó, *Guide to FPGA implementation of arithmetic functions*. Springer Science & Business Media, 2012, vol. 149.

[27] I. Koren, *Computer Arithmetic Algorithms, 2nd edition*. Prentice-Hall Inc, 2002.

[28] B. Parhami, *Computer arithmetic*. Oxford university press New York, NY, 2010, vol. 20.

[29] P. Jamieson and J. Rose, “Architecting Hard Crossbars on FPGAs and Increasing their Area-Efficiency with Shadow Clusters,” in *IEEE International Conference on Field-Programmable Technology*, 2007, pp. 57–64.

[30] ——, “Mapping Multiplexers onto Hard Multipliers in FPGAs,” in *3rd International IEEE Northeast Workshop on Circuits & Systems*, 2005, pp. 215–226.

[31] ——, “Enhancing the area-efficiency of FPGAs with hard circuits using shadow clusters,” in *IEEE International Conference on Field-Programmable Technology*, 2006, pp. 1–8.