An Efficient Implementation of Radix-4 Division by Recurrence

By

Kevin Hill

Bachelor of Science in Computer Engineering
Oklahoma State University
Stillwater, Oklahoma
2017

Submitted to the Faculty of the
Graduate College of
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2018

An Efficient Implementation of Radix-4 Division by Recurrence

Thesis Approved:

Dr. James E. Stine

Thesis Advisor

Dr. Keith A. Teague

Dr.Weili Zhang

ACKNOWLEDGMENTS

I would like to acknowledge the Air Force Research Laboratory for funding this research under contract FA8750-17-1-0261.

Name: Kevin Hill

Date of Degree: DECEMBER, 2018

Title of Study: An Efficient Implementation of Radix-4 Division by Recurrence

Major Field: ELECTRICAL AND COMPUTER ENGINEERING

Abstract: This paper presents the design of a radix-4, 32-bit integer divider which uses a recursive, non-restoring division algorithm. The primary focus for this design is on high-speed operation while maintaining low power consumption. This implementation accepts 32-bit unsigned integers as input, and returns the quotient, remainder, and a special case divide-by-zero flag. Included in this paper is the motivation for this design, background information necessary to understand the algorithm in use, details of the algorithm's implementation, and the evaluation of the proposed design implemented in IBM/GF 32nm SOI technology, which is then compared against other division implementations.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Division is one of the basic arithmetic operations present in many arithmetic-logic-units (ALU) along with addition, subtraction, multiplication, and more. While it is common to think arithmetic operations may only occur if a user is using a calculator or similar function, these operations are actually occurring almost constantly as processors manipulate data for general operation.

Other arithmetic operations, such as addition, are generally used much more frequently than division [1]. This could beg the question why should research focus on an operation that may take up less than 5% of all arithmetic operations [1]. The answer to this question is largely two-fold. First being that division, particularly integer division, makes up for low usage in that it has both a high, and unpredictable cycle count when compared to other operations [2]. When a particular operation has such a relatively high cycle count, this can lead to a dramatic reduction in overall processor throughput as the processor must wait for division to complete before moving onto the next instruction. In fact, according to [1], while floating-point (FP) divides only account for approximately 5% of all FP instructions, it accounts for over 40% of the floating-point unit's (FPU) stall time.

Beyond relatively slow speeds, division also suffers from implementation complexity and the elusiveness of errors. Division implementations are notoriously complex and difficult to understand, which makes it easier for problems to be missed until it is late in a product development cycle, and thus more difficult and expensive to fix. An example of this can be seen in the infamous 1994 Intel Pentium floating-point divide (FDIV) bug [3]. This bug in the Pentium FPU meant that it was possible that the FDIV operation could return an incorrect result. If this bug had not been caught shortly after the processor's release, it could have caused catastrophic errors, particularly for computers used in fields like finance, government and military, or health-care. Speaking to the error detection difficulties, this bug showed that despite testing methods that were considered rigorous, division bugs still prove themselves difficult to detect, making an intimate knowledge of a divider's design critical.

When combining the issues of relatively slow speeds and design complexity, it is easy to see why further research into division methods is of critical importance to increase processor performance, as well as overall knowledge of the subject so improvements can be made well into the future. This leads to the integer divide (IDIV) implementation presented in this work. While many previous works [2], [4], [5] discuss higher radix division algorithms and results of implementations, they are short on details for how to implement the algorithm they discuss, which is the main goal of this work; that is, to provide necessary background and implementation details for both a greater understanding of the IDIV algorithm, and allow for replication of the implementation.

The design detailed in this work is an integer divider, an important data-path element for any ALU, specifically allowing the use of instructions such as modulus and IDIV [2]. There are two generic ideas behind the implementation of IDIV. The first of these is a combinational implementation, and the other is sequential implementation. While combinational dividers are convenient in that they finish the calculation in one clock cycle, their downside is a tremendous reduction in processor clock speed due to heavy delay. On top of this, combinational dividers occupy a large area, draw significantly more power, and are difficult to pipeline making them non-ideal for general use [2]. That is not to say they do not have purpose; asynchronous circuits and other application-specific ALU's may benefit from a combination implementation.

With that said, the division implementation described in this work is a sequential design. Sequential designs are desirable for most applications for both general-use System-on-Chip (SoC), and application-specific integrated circuits (ASIC). The main idea behind sequential circuits is to break up all of the work done in one clock cycle with a combinational design, and spread it out over multiple cycles; which is ideal for recursive algorithms like the one used in this design. It also provides for implementations that are low in area and promote low-power tendencies.

The process of designing today's newest architectures has undergone a rapid shift from previous generations. This is because many general-purpose systems are now becoming more application-specific. Therefore, traditional architectures and digital systems that have long been designed with speed in mind must now combine different approaches for optimization, such as energy, power dissipation and application-specific functions (e.g., machine learning), in addition to speed. It has also been argued that today's architectures need new approaches or, better yet, tried-and-true techniques to get better optimization of constraints than those that were previously conceived [6].

This paper presents a radix-4 unsigned digit integer divider using recurrence that exhibits low-power tendencies [7]. In addition, as opposed to other implementations that traditionally use radix-2 implementations [5] or carry-save adders [8], this paper illustrates methods for implementing integer division by recurrence using higher radices. More importantly, the major novelty in this paper is the demonstration of a method for deploying a radix-4 unsigned adder using hierarchy for integer division to allow effective deployment of logic.

CHAPTER 2

BACKGROUND

As stated in the introduction, the proposed divider uses a higher-radix, non-restoring division (NRD) algorithm, allowing division to be completed in fewer cycles by retiring more bits per cycle. This chapter gives the background information necessary for the understanding of the NRD algorithm and its implementation found in Chapter 3. This chapter is broken into three sections. The first, Restoring Division, details the base algorithm from which the NRD algorithm is developed. The next section, Signed-Digit Representation, describes the non-canonical digit-set required for this algorithm. Lastly, the Non-Restoring Division section explains the improvements to the restoring division algorithm which make the final implemented NRD algorithm.

2.1 Restoring Division

The main idea behind division by recurrence is actually quite simple. In fact, it is the same way that many first learn to do division by hand in elementary school. Following the example seen in Figure 2.1, the problem given is to divide 100 by 3 where the first operand is called the dividend and the second operand is called the divisor. First, multiply a chosen quotient value (in this case 3) by the divisor to create new value (9) then subtract this value from the dividend. This process continues until the residual is smaller than the divisor, and it becomes the final remainder.

We can take this process and use it to arrive at a simplified equation to compute the quotient iteratively. It is assumed that the input operands are the dividend (n) and the divisor (d). This equation is radix-based (r) and every iteration produces the

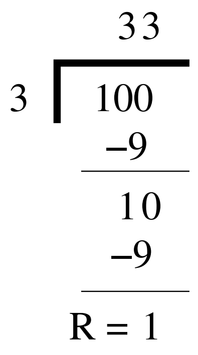


Figure 2.1: Simple Division Example

quotient q_{j+1}:

$$w_{j+1} = r \cdot w_j + q_{j+1} \cdot d \ .$$

where w is the residual, and $w_0 = n$. This equation for the $j+1$ residual is calculated each iteration along with a selected quotient digit. The process of selecting the quotient every iteration is called the Quotient Selection Table (QST) process [8].

The theory for selecting the quotient relies heavily on determining containment and continuity, which are covered in sections 2.3.1 and 2.3.2, respectively, and in even greater detail in [8]. QST architectures rely on paper-pencil techniques for subtractive-based division by looking at the divisor and shifted partial remainder. That is, the QST is selected based on:

$$q_{j+1} = QST\{r \cdot w_j, D\} \ .$$

Each selected value of q_{j+1} is then stored in a register, and shifted left each iteration to get a final quotient Q. A flow diagram for this entire process can be seen in Figure 2.2 [8].

2.2 Signed-Digit Representation

One of the requirements for use of the NRD algorithm is that quotient digits must be represented as signed digits rather than standard canonical digits. The reason

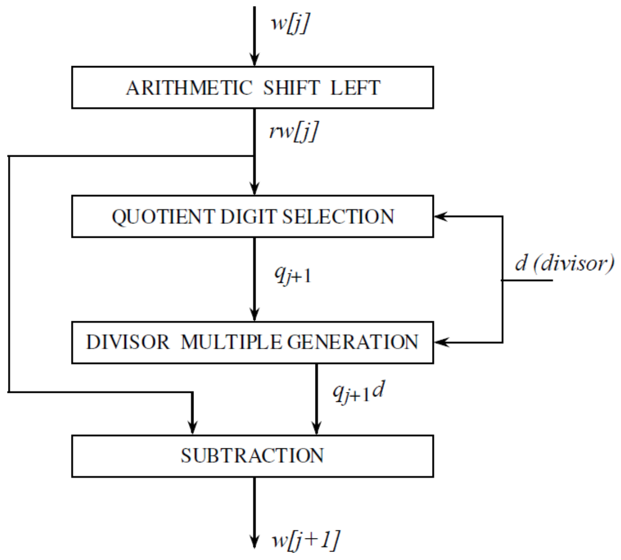


Figure 2.2: Restoring Division Flow (Adapted from [8])

for this requirement is explained later in section 2.3. The idea behind signed-digit (SD) representation is to use multiple or redundant digits to represent a single value. Although mathematically this idea has been around for several hundred years, it was not until the 1960s that it gained traction for its uses in digital arithmetic. This is mainly due to the work of Avizienis [9] that illustrated that SD representations can be helpful in promoting carry-free logic by allowing a digit to represent more than r values, where r is the integer's radix. In contrast, most fixed radix systems employed by digital devices has the digit set restricted to $\{0, \ldots, r-1\}$, which is known as a canonical or a conventional digit-set representation.

Although the SD numbering system is useful, it is somewhat cumbersome to convert from SD notation back to a conventional binary representation [8] using standard techniques. This conversion typically involves the use of a carry-propagate adder (CPA) [8], which will end up setting the device's critical path and should be avoided if possible. Unfortunately, for compatibility with other devices, the quotient must usually be reverted back to a canonical form [8].

SD arithmetic is helpful in that it provides redundancy in its definition of a number [8]. This ultimately leads to simplification of boolean logic for digital circuits. The largest benefit of redundancy for the purpose of division is that it simplifies the selection of the quotient bits [4] in the QST. Most SD numbers usually employ symmetric digit set representations such that:

$$D = \{\overline{a}, \ldots, -1, 0, 1, \ldots, a\} \ .$$

where $\overline{a} = -a$. In contrast to canonical digit sets, redundant digit sets allow a single digit to represent more than one value. For example, with $a = 2$ and $r = 4$, known as radix-4 representation, the value of 2_4 can be represented as 02_4 or $1\overline{2}_4$.

A measure of the redundancy of a symmetric SD representation can made using the measure of redundancy (ρ), and the radix (r), defined as follows:

$$\rho = \frac{a}{r-1} \ni \frac{1}{2} < \rho \leq 1 \ .$$

Redundant number representations employ redundancy factors that are $\geq \lceil r/2 \rceil$. The choice of redundancy factor can limit or enhance representations of numbers and subsequently their implementations. Many systems, including this work, typically utilize minimally-redundant representations such that $\rho = 2/3$. The reason for this is simply that the multiply by 2 operation is easier to accomplish in hardware than a multiply by 3 operation, or other non-power of two value.

2.3 Nonrestoring Division

As explained in Section 2.1, the restoring division algorithm is an effective method for correctly calculating an integer division operation. Unfortunately, when the calculated residual is found to be negative, it performs the restore operation. The restore operation is the process of adding the divisor back to the remainder, and subtracting one unit in the last position (*ulp*) from the quotient in order to ensure the remainder is positive. Ultimately, the restore operation is inevitable, but it can be seen by combining the restoration with the next subtraction of the divisor, restoration can be eliminated during the iterative process. Subsequently, the CPA that would be required to do the restore operation is also eliminated saving power, area, and delay expenses. There are two conditions that must be met however in order to use the NRD algorithm. The first is containment and the second is continuity.

2.3.1 Containment

The containment condition [8] sets up the selection intervals necessary for computing the subsequent quotient digit. For a given quotient digit, q_{i+1} can be chosen to be k. That is, an interval of allowable partial remainders. These regions are defined by the interval $[L_k, U_k]$ such that L is the lower value and U is the upper value of the shifted partial remainder, $r \cdot w_i$, so that the subsequent shifted partial remainder is bounded. In other words, the interval is chosen based on the range of redundancy to avoid the quotient from not representing the final result properly. The equations for the upper and lower bounds can be given as [8]:

$$\begin{aligned} U_k &= (k + \rho) \cdot d \\ L_k &= (k - \rho) \cdot d \ . \end{aligned}$$

2.3.2 Continuity

The next condition, continuity, exists because of the overlap that occurs in the containment condition. From the equations given above for U_k and L_k, it's easy to see that the upper bound U_{k-1} will overlap with the lower bound L_k. Unfortunately, because of this overlap, a quotient digit may be chosen from either minimum value. To help understand the recurrence, a visualization is utilized called Robertson's diagram (Figure 2.3) that plots the recurrence relationship for a given quotient digit (i.e., a plot of w_{j+1} versus $r \cdot w_j$). The choice of redundancy within the SD notation will incur an overlap between L_k and U_{k-1} such that s_k (i.e., the selection interval)

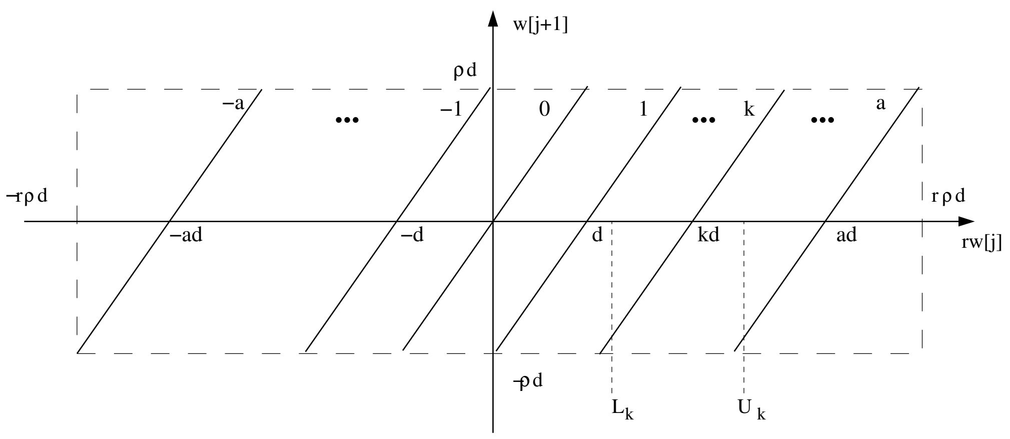


Figure 2.3: Robertson's Diagram (w_{j+1} vs $r \cdot w_j$).

can either be $k-1$ or k. Since the containment equations are defined, it is easy to measure this overlap as

$$U_{k-1} - L_k = (k-1+\rho) \cdot d - (k-\rho) \cdot d = (2 \cdot \rho - 1) \cdot d \ .$$

The simplest selection function is to make s_k constant and perform a comparison on a constant value. Thus, many implementations for QSTs resort to memory lookups or possibly programmable logic array (PLA) elements. The constants should satisfy the following containment condition for selection [8]:

$$max(L_k(d_i), L_k(d_{i+1})) \leq m_k(i) \leq$$
$$min(U_{k-1}(d_i), U_{k-1}(d_{i+1})) \ .$$

The max and min are required because the containment produces regions that are positive and negative with the PD plot which can be seen in Figure 2.4. A PD plot is a remapped Robertson's Diagram used to plot the values necessary for correct QST selection [4] (i.e., $r \cdot w_j$ vs. D).

The PD plot is a useful visual aid to see all of the selection regions possible for a given implementation. As the selection process goes into the negative quadrant, the selection regions are switched [8].

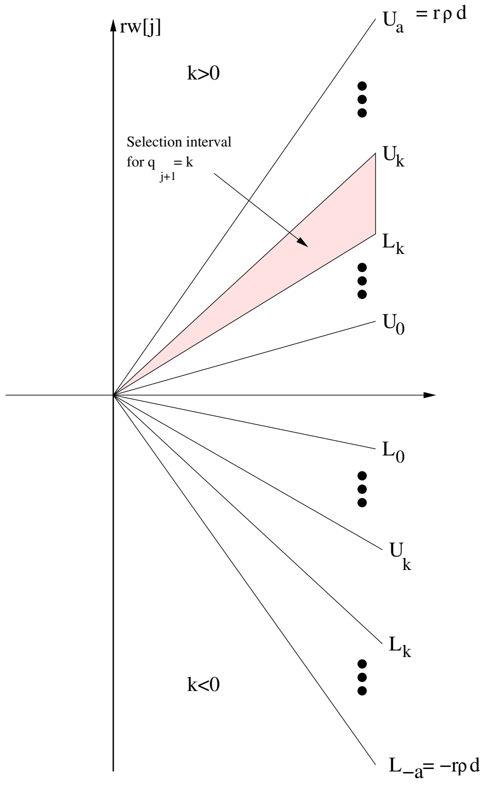


Figure 2.4: P-D diagram.

Since a single set of selection constants is used for a given interval of length $2^{-\delta}$, only the δ most-significant bits are used in the QST function. From this, we now have to figure out the limited number of bits needed to be examined from the shifted residual. Any size can be utilized, however, a smaller number of bits is better since this minimizes the QST table regardless if implemented with a memory lookup or combinationally. Therefore, the selection constant has the following form:

$$m_k(i) = A_k(i) \cdot 2^{-c} \ ,$$

where A_k is an integer.

CHAPTER 3

Implementation

This chapter details the architectural implementation for the NRD algorithm shown in Chapter 2. The final architecture for the divider design is shown in Figure 3.1. The overall design allows for a maximum of 19 iterations to return a correct result. The critical path is set by the CPA leading to the final output `rem`. To preserve cycles, the number of iterations is calculated and passed to a finite state machine (FSM), not pictured, for signal control, rather than assuming worst case for each operation.

From Figure 3.1, the iterative nature of the recursive algorithm is easily seen, with the carry and sum looping back to the multiplexers (mux) at the top. Most of the architecture is straight forward to understand and replicate from the figure. There are however some exceptions to this.

The first, is that for 32-bit division, there are 36 bits of internal precision, starting with the shifted divisor. Previously, Section 2.1 indicates that we iteratively select and store a quotient q_{j+1} to get a final quotient. In the figure however, it is seen that instead of `Q` coming from `QST`, it comes from previously unmentioned blocks, `ls_control` and `OTF`. Lastly, it is stated in sections 2.1 and 2.3 that restoration is eventually required, and the final CPA blocks at the bottom of the architecture diagram show this in place. The following sections help to explain each of these architecture components and their purpose.

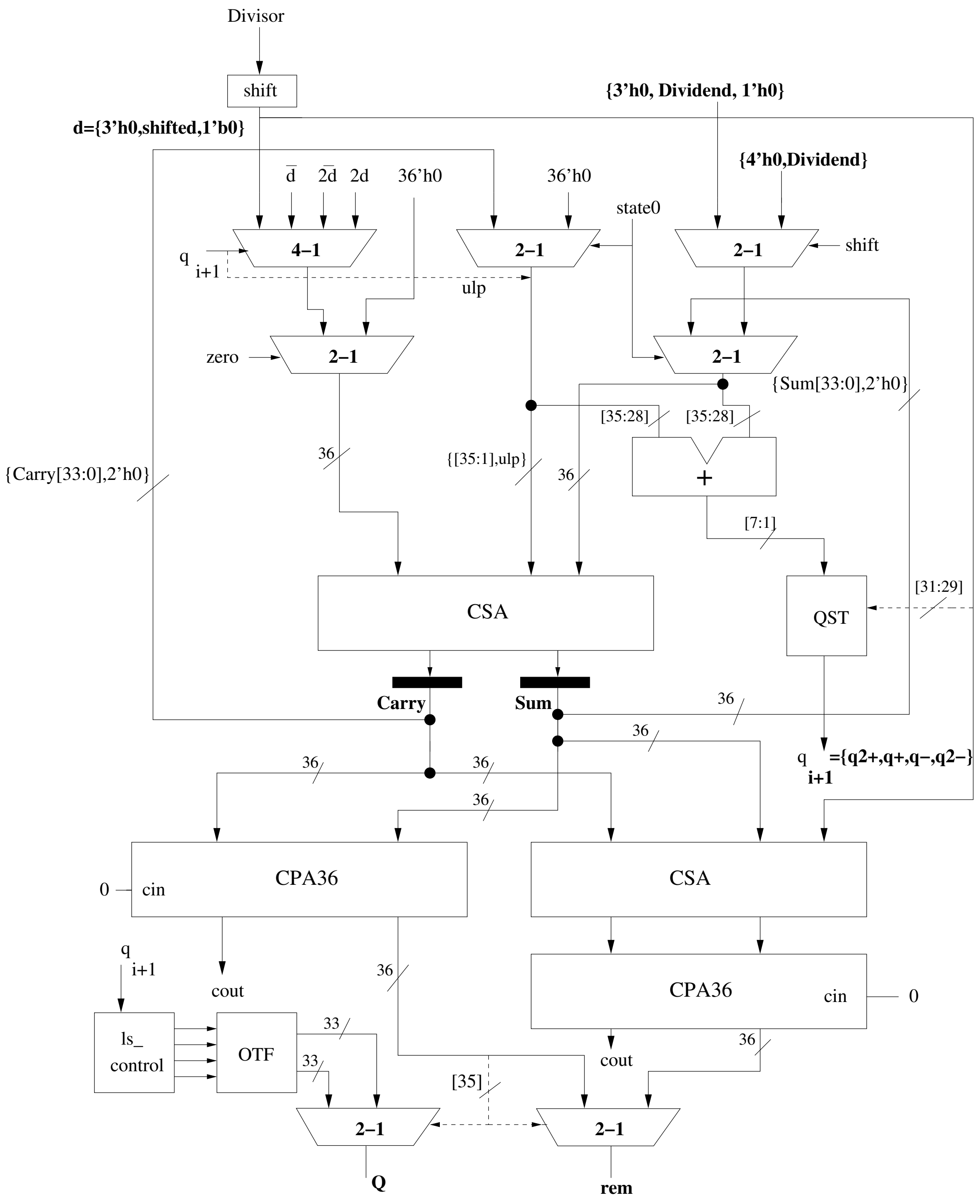


Figure 3.1: Architecture of the Proposed Integer Divider.

3.1 Normalization and Alignment

One of the potentially standout items that can be seen in Figure 3.1 is the internal bus sizes. For a 32-bit integer divide operation, the internal buses for the carry and sum calculation are all 36 bits. This is due to the normalization of the divisor that must occur to allow for the QST to operate correctly.

In order for the selection functions to work properly, the divisor must have a leading-one integer bit. Therefore, the divisor is shifted accordingly via a leading-one detector [8]

$$d^{*} = 2^{m} \cdot d$$

where d^{*} is the shifted divisor, and m is the number of shifts required to have a leading-one. This gives us a modified divisor in the range of $[1/2, 2)$.

Since this computation involves a higher radix version of division, the bits that get retired could be complicated by a non-power of two boundary. That is, a correct remainder is required for the bit of the quotient so that it is aligned to the radix-r boundary. This requires extra bits within the intermediate computation to allow for the computation to take place.

Moreover, as mentioned earlier, Robertson's diagram lays out the range of values that be utilized during a recurrence. This ultimately causes the recurrence to go out of bounds. Therefore, the resulting bounds within the computation require extra integer bits as stipulated by Robertson's diagram (i.e., $[-r \cdot \rho \cdot D_{min}, r \cdot \rho \cdot D_{max}]$). The radix-4 case with $a = 2$, in a $[1, 2)$ domain, results in a range of $[-16/3, 16/3]$. This means that any recurrence computation requires 3 integer bits for the unsigned integer case in order to avoid loss of precision.

Division is harder than most computations because it has to deal with bits from the most-significant portion towards the least-significant digits, sometimes called an online algorithm [8]. Therefore, since the range of the final quotient is modified (i.e.,

$1/2 < q < 2$), the dividend has to also be shifted appropriately. As specified in [8], this can achieved by shifting the dividend right by $v + s$ bits where $v = 2$ for $\rho < 1$ such that:

$$(m + v + s)\ mod\ k = 0\ ,$$

for m fractional bits and $k = 2$ for radix 4 (i.e., $r = 2^k$).

Integer division requires a remainder to be produced, which is one advantage of using division by recurrence compared to other division methods that employ multiplication. However, since division by recurrence is implemented using a non-restoring division algorithm, the remainder needs to be modified appropriately. Consequently, the remainder needs to be shifted accordingly after N cycles [8]:

$$rem = \begin{cases} w[N] \cdot 2^{(n \cdot log_2 r - m)} & \text{if } w[N] \geq 0 \\ (w[N] + D) \cdot 2^{(n \cdot log_2 r - m)} & \text{if } w[N] < 0\ . \end{cases}$$

3.2 Quotient Generation

Chapter 2 mentions that there are multiple ways to implement a QST. Since this QST is chosen to have each selection region constant, there are generally two ways to implement this type of QST. The first way is using a memory-lookup. While memory-lookup could be a viable option for larger integer division, it is impractical for this implementation due to a relatively small QST (58 product terms). Instead, the QST uses combinational logic in order to avoid the increased complexity of introducing a memory component along with the high area and power requirements. In addition, the proposed design incorporates specialized logic to maintain a one-hot encoding for the quotient selection in order to save power and spurious switching.

Chapter 2 indicates that the QST is responsible for the generation of the final quotient. While this is correct, it must be remembered that the output of the QST is in SD notation that can only be handled by other devices that are specifically designed

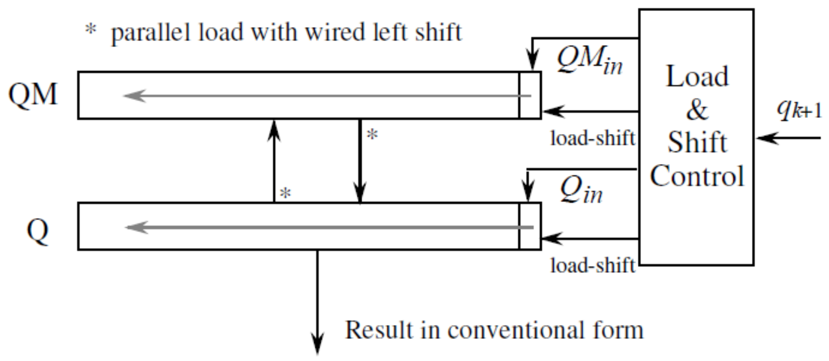
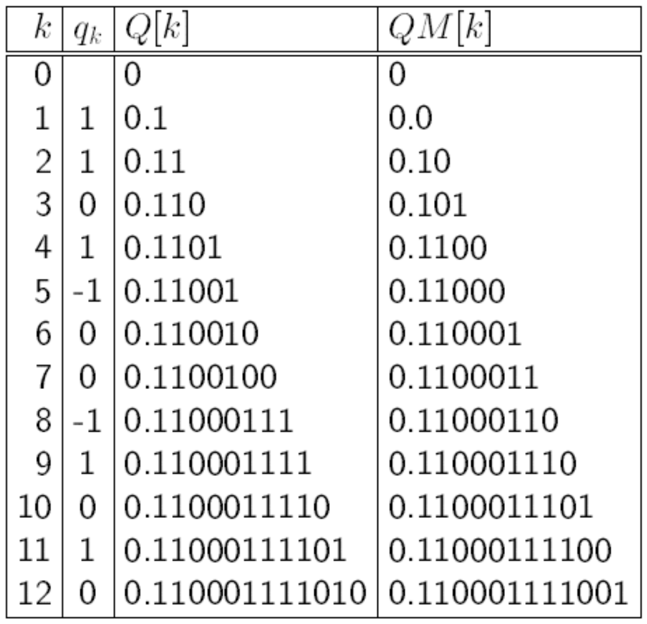


Figure 3.2: OTF Conversion Block Diagram (Adapted from [8])

to take it as input; which makes it necessary to return back to standard binary. As previously stated, this would typically require the use of a CPA after the entire quotient had been calculated. Instead, this design uses a technique called on-the-fly (OTF) conversion to convert from the SD redundant notation back to conventional binary [8]. The `OTF` unit is responsible for the translation of each quotient digit during a recurrence and is controlled by the `ls_control` block. The hardware visualization for this relationship can be seen in Figure 3.2.

On-the-fly conversion [8] is an extremely efficient method of conversion since recurrence methods for division, and even square root, are online algorithms [8]. Most OTF designs need a clock to store elements every iteration and subsequently have setup and hold constraints that require the need to handle this storage with the generation of the quotient. The logic for OTF conversion is based on knowing what the most-significant digit of the previous conversion (Q) is and then either selects the previous version or selects another value of the conversion that is one *ulp* smaller ($QM = Q - ulp$) [8]. It is easier to describe OTF conversion as a series of careful shifts and loads. The simplified example below (Figure 3.3) in radix-2 can be used to understand how on-the-fly conversion takes the returned QST values ($0.1101\bar{1}00\bar{1}1010$) and converts it to the standard binary representation (0.110001111010). Q and QM registers are initialized to 0, and depending on the chosen q_k, the values are shifted



k	q_k	$Q[k]$	$QM[k]$
0		0	0
1	1	0.1	0.0
2	1	0.11	0.10
3	0	0.110	0.101
4	1	0.1101	0.1100
5	-1	0.11001	0.11000
6	0	0.110010	0.110001
7	0	0.1100100	0.1100011
8	-1	0.11000111	0.11000110
9	1	0.110001111	0.110001110
10	0	0.1100011110	0.1100011101
11	1	0.11000111101	0.11000111100
12	0	0.110001111010	0.110001111001

Figure 3.3: OTF radix-2 example (Adapted from [8])

between registers and then shifted left.

In this example, when q_k is 1, $Q[k-1]$ is shifted to $QM[k]$, and left shifted with a 0 stored in the least significant bit (LSB). Then, $Q[k]$ is left shifted with a 1 stored in the LSB. When q_k is 0, $Q[k]$ is left shifted and a 0 is stored, while $QM[k]$ stores a 1. Lastly, When q_k is -1, the value from $QM[k-1]$ is shifted into the $QM[k]$ register and a 1 is stored, and $QM[k]$ is simply left shifted with a 0 stored in the LSB. The equations for OTF operation are given in [8] as:

$$Q[j+1] = \begin{cases} (Q[j], q_{j+1}) & \text{if } q_{j+1} \geq 0 \\ (QM[j], (r - |q_{j+1}|) & \text{if } q_{j+1} < 0 \end{cases} .$$

$$QM[j+1] = \begin{cases} (Q[j], q_{j+1} - 1) & \text{if } q_{j+1} \geq 0 \\ (QM[j], ((r-1) - |q_{j+1}|) & \text{if } q_{j+1} < 0 \end{cases} .$$

The result of this at the end of the calculation is that the QM register will store the value of the quotient one *ulp* less than the value stored in the Q register. If the remainder returns negative, then the final quotient will be chosen as the value stored in QM to complete the required restore operation.

3.3 Remainder Generation

As discussed in Chapter 2, restoring division is not practical for a high-speed implementation due to the restoration step. To refresh with restoring division, when the residual w_{j+1} is negative, the divisor is added back to the remainder and the quotient has one *ulp* subtracted (Figure 3.4). To avoid the restore step, the nonrestoring division algorithm along with SD representation is implemented. Unfortunately, restoration must still occur in order to get a correct and positive remainder. Figure 3.4 can be utilized to visualize this occurrence. In this example, if the remainder correction is not performed, it will be returned as negative which is mathematically valid ($3 \cdot 34 - 2 = 100$), but is typically not helpful as output. It also goes against mathematical convention of a positive remainder. The two CPA's and mux at the bottom of Figure 3.1 choose between the remainder that has the divisor added back if it is negative, or the normal remainder if no restore is needed.

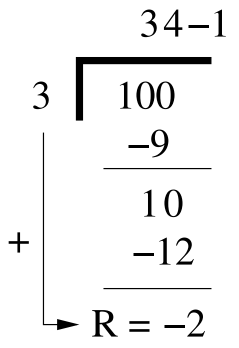


Figure 3.4: Restoration Operation

CHAPTER 4

RESULTS

4.1 Simulation and Verification

The proposed divider is designed and simulated using RTL-compliant SystemVerilog. In order to ensure correct operation, a testbench was created to randomly generate approximately 32 million vectors as input to the device. Values returned by the divider are compared against true results within the testbench given by the division and modulus operators to assure correctness.

To attempt to adhere to the IEEE-1801 standard for Design and Verification of Low-Power Integrated Circuits [10], the power numbers provided in Table 4.1 and Table 4.2 are generated using Synopsys' PrimeTime using $28,344$ vectors. In order to obtain good estimations of energy consumption, PrimeTime requires a Switching Activity Interchange Format (SAIF) file. The purpose for this file is to assist in the extraction and storing of switching activity from a design [10], which can make large differences in reported power numbers. It also assists in power approximations from the characterization liberty file. The SAIF file for this design was generated using a combination of simulation and Synopsys®tools.

4.2 Design Comparison

For a baseline comparison, two other designs, `DW_div_seq` and `DW_div`, from the Synopsys DesignWare Intellectual Property (IP) library are also implemented [11] in the same IBM/GF 32nm SOI process. These designs were chosen for comparison for several reasons. The first is that the Synopsys IP library is widely used in both industry and academia in order to speed up design time, particularly if the device needed is outside of the expertise of the organization. Next, because Synopsys®creates both the synthesizer and the design, the finished device is highly optimized, so it serves well as "optimal" design to be compared against.

`DW_div_seq` is a sequential design which allows for both a radix-2, and a radix-4 synthesis model. In order to ensure the two designs are as similar as possible, the radix-4 implementation is used in the comparison and the number of cycles is set to 19 via the `num_cyc` variable. `DW_div` is a combinational implementation which is included not just for comparison, but also to illustrate the importance sequential design. Not only does the combinational design exhibit a dramatic increase in both area and power, it is also the only of the three designs to not meet the 1 GHz timing requirement, although it was relatively close (706.2 MHz).

Both divider designs were synthesized using Synopsys' Design Compiler (DC) topographical mode, optimized for minimum delay. DC topographical mode is desirable over the typical wire load model (WLM) in order to ensure a stronger correlation to the area, power consumption, and delay of a post-layout physical implementation. To give some comparison of synthesis, the number of logic levels is also reported from its Quality-of-Route (QoR) output. The number of logic levels is an effective method of

Table 4.1: Post-synthesis Results for the Proposed Design in cmos32soi 32nm GF technology 1 GHz using RVT cells

Methods	# Cells	Area [um^2]	Logic Levels	Power [mW]			
				Dynamic	Static	Leakage	Total
Proposed	2,191	1,922.08	22	0.0192	0.1002	0.5983	0.7177
`DW_div_seq`	829	1,000.58	43	0.0939	0.7136	0.3097	1.1137
`DW_div`	18,337	20,497.23	139	7.1251	5.3012	11.7124	24.1387

determining the efficiency of logic deployment by stating the number of gates that defines the critical path. Since energy and power are input dependent [12], all synthesis runs were completed using a 1 GHz cycle time.

Estimates for area, power, and delay for each component used in the design can be seen in Table 4.1 and Table 4.2. There were five different threshold voltage libraries obtained from ARM®used during synthesis: HVT, MVT, RVT, SVT, and UVT. HVT signifies the "high" threshold voltage, MVT signifies the "mezzanine" threshold voltage, RVT signifies the "regular" threshold voltage, SVT signifies "super-high" threshold voltage, and UVT signifies "ultra-high" threshold voltage, respectively. Due to the higher threshold voltage, the HVT, SVT, and UVT libraries have a lower static power but higher dynamic power. The lower threshold voltage MVT and RVT libraries have a lower dynamic power but a slightly higher static power. By synthesizing the designs to different combinations of these libraries, the set of libraries that optimizes the proposed design in terms of lowering area, delay, and power is determined experimentally. Therefore, Table 4.1 is synthesized for only RVT cells concentrating on delay, whereas, Table 4.2 is synthesized for lower leakage. It is important when designing any circuit with multiple libraries to determine which library or combination of libraries best fits design goals and constraints.

Table 4.2: Post-synthesis Results for the Proposed Design in cmos32soi 32nm GF technology at 1 GHz using Low-Leakage cells

Methods	# Cells	Area [um^2]	Logic Levels	Power [mW]			
				Dynamic	Static	Leakage	Total
Proposed	2,200	1,957.53	19	0.0176	0.0948	0.0531	0.1655
`DW_div_seq`	1,690	1,333.57	31	0.1032	0.6937	0.0044	0.8014
`DW_div`	19,583	21,824.71	146	7.3791	5.6150	9.2842	22.2783

CHAPTER 5

CONCLUSIONS

This paper presents a novel implementation for a radix-4, 32-bit integer divider. The benefit of higher radix is a reduction in the number of cycles required for a complete calculation. The maximum number of cycles in this design is 19 and it has a clock rate of 1 GHz, which gives a maximum calculation time of 19 nanoseconds (ns). The critical path for this design is set by the remainder generating CPA rather than the quotient thanks to the use of the efficient on-the-fly conversion technique.

The basis for the implementation is a nonrestoring division algorithm which is based off the restoring division algorithm. The key difference being that the non-restoring algorithm does not restore the divisor when the residual is negative. In order to accomplish this, it must use signed-digit representation, to give digits a measure of redundancy, which reduces the complexity of the quotient selection table.

The area, power, and timing estimates are given for an implementation in IBM/GF 32SOI 32-nm technology synthesized assuming a 1 GHz clock rate. The results indicate that this divider is slightly larger than a comparable divider from Synopsys' DesignWare IP, but has significantly smaller amounts of power dissipation and number of logic levels, making this design desirable for low power applications.

5.1 Future Work

It is important on any design to look towards the future for possible further research and improvements. This design is no exception. One way to expand on the work done in this implementation is to further increase the radix. A higher radix will allow more bits per cycle to be retired, which in turn decreases the total number of cycles required to complete a calculation. Unfortunately as the radix increases, the combinational logic required to correctly calculate the quotient for each cycle also increases in complexity.

As the radix continues to increase, it approaches an, effectively, combinational design. From the results for the `DW_div` implementation, this has dramatic consequences for the area, power consumption, and single-cycle delay. The benefits gained by increasing the radix will begin to plateau due to a nonlinear relationship between the radix and the logic required to complete the calculation. This idea is given as a visual in Figure 5.1 [8] which shows the estimated factor of area increase and speedup for a chosen radix. When choosing a radix for a divider, these consequences must be taken into account for the specific application being used.

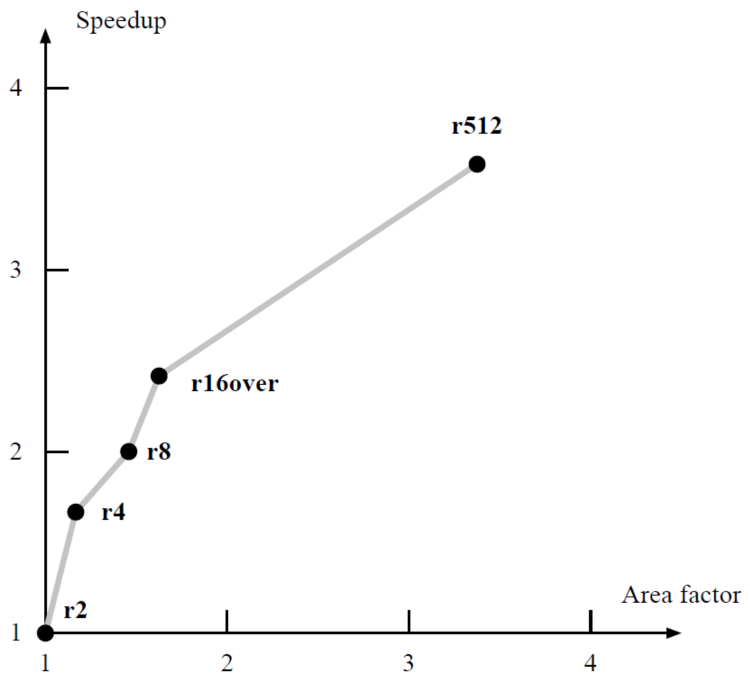


Figure 5.1: Radix vs Area and Speedup (Adapted from [8])

BIBLIOGRAPHY

[1] S. F. Oberman and M. J. Flynn, "Design issues in division and other floating-point operations," *IEEE Transactions on Computers*, vol. 46, pp. 154–161, Feb 1997.

[2] N. Takagi, S. Kadowaki, and K. Takagi, "A hardware algorithm for integer division," in *17th IEEE Symposium on Computer Arithmetic (ARITH'05)*, pp. 140–146, June 2005.

[3] D. Price, "Pentium FDIV flaw-lessons learned," *IEEE Micro*, vol. 15, pp. 86–88, Apr 1995.

[4] D. E. Atkins, "Higher-radix division using estimates of the divisor and partial remainders," *IEEE Transactions on Computers*, vol. C-17, pp. 925–934, Oct 1968.

[5] J. Ebergen and N. Jamadagni, "Radix-2 division algorithms with an over-redundant digit set," *IEEE Transactions on Computers*, vol. 64, pp. 2652–2663, Sept 2015.

[6] J. Dean, D. Patterson, and C. Young, "A new golden age in computer architecture: Empowering the machine-learning revolution," *IEEE Micro*, vol. 38, pp. 21–29, Mar 2018.

[7] K. Hill and J. Stine, "An efficient implementation of radix-4 integer division by recurrence for low-power applications," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4, May 2019.

[8] M. D. Ercegovac and T. Lang, *Digital Arithmetic.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2003.

[9] A. Avizienis, "Signed-digit numbe representations for fast parallel arithmetic," *IRE Transactions on Electronic Computers*, vol. EC-10, pp. 389–400, Sept 1961.

[10] "IEEE standard for design and verification of low-power, energy-aware electronic systems," *IEEE Std 1801-2015 (Revision of IEEE Std 1801-2013)*, pp. 1–515, March 2016.

[11] Synopsys, Inc., *DesignWare Building Block IP User Guide*, 2018.

[12] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective.* USA: Addison-Wesley Publishing Company, 4th ed., 2010.

VITA

Kevin Hill

Candidate for the Degree of

Master of Science

Thesis: An Efficient Implementation of Radix-4 Division by Recurrence

Major Field: Electrical Engineering

Biographical:

Education:

Received the B.S. degree from Oklahoma State University, Stillwater, Oklahoma, 2017, in Computer Engineering

Completed the requirements for the degree of Master of Science with a major in Electrical Engineering at Oklahoma State University in December, 2018.

Experience:

Intern at Enercon Services: Summer 2014

Intern at Tulsa County Government: Summer 2016

Research Assistant at Oklahoma State University: Summer 2017 - Fall 2018

Teaching Assistant at Oklahoma State University: Fall 2017 & Fall 2018

Intern at Skyworks Solutions: Summer 2018

Publications

K. Hill and J. Stine, ”An Efficient Implementation of Radix-4 Integer Division by Recurrence for Low-Power Applications,” *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, Sapporo, 2019, pp. 1-4. (submitted)