Rochester Institute of Technology
RIT Scholar Works

Theses | Thesis/Dissertation Collections

5-2017

Vedic Based Division Core Design

Ryan Hinkley
rnh8227@rit.edu

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
Hinkley, Ryan, "Vedic Based Division Core Design" (2017). Thesis. Rochester Institute of Technology. Accessed from

This Master's Project is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

VEDIC BASED DIVISION CORE DESIGN

by

Ryan Hinkley

A Graduate Paper Submitted

in

Partial Fulfillment

of the

Requirements for the Degree of

MASTER OF SCIENCE

in

Electrical Engineering

Approved by:

PROF. ______________________________

(GRADUATE PAPER ADVISER - MARK A. INDOVINA)

PROF. ______________________________

(DEPARTMENT HEAD - DR. SOHAIL A. DIANAT)

DEPARTMENT OF ELECTRICAL AND MICROELECTRONIC ENGINEERING

COLLEGE OF ENGINEERING

ROCHESTER INSTITUTE OF TECHNOLOGY

ROCHESTER, NEW YORK

MAY, 2017

I would like to dedicate this work to my family, Sue, Don, Kara, and Abby for all their love and support throughout my entire life, and also to my friends for their love and inspiration throughout all of my schooling.

Declaration

I hereby declare that except where specific reference is made to the work of others, that all content of this Graduate Paper are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This Graduate Project is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Ryan Hinkley

May, 2017

Acknowledgements

I would like to thank my adviser, Mark A. Indovina, for his support, guidance, and feedback through the past two years in both classes and in completion of this graduate paper. I would also like to thank my colleague Julia Okvath for her assistance in finalizing this work. I am also thankful for my other teachers that have taught me throughout my five years of college. Without all of their help, I would not be where I am today.

Abstract

Processors and embedded systems perform algebraic manipulations such as addition, subtraction, multiplication and division on integers in order to complete and execute programs. Unfortunately, most division cores in these devices take significantly more time to compute than other manipulations. Because of this, research has been conducted on finding a division algorithm that does not take as long to execute. This graduate paper discuses and proposes a division core design based on ancient Vedic mathematics. The design is created and simulated as a 4-bit system for completion and analyzed for timing, power consumption, and cell area usage. Estimations on larger designs are completed and used to compare this design to similar ones. It is found that the timing and cell area usage are very minimal compared to other designs based on the same algorithm. However, the power consumption is significantly higher. Possible optimizations and improvements to the design are proposed for future use. At the end of this graduate paper, recommendations are given to optimize and upgrade the current design.

Contents

Contents v

List of Figures vii

List of Tables viii

1 Introduction 1

1.1 Research Goals 1

1.2 Contributions 2

1.3 Organization 3

2 Preliminaries and Background Research 4

2.1 Pipelined System 4

2.2 Vedic Division Algorithm 4

2.3 Similar Architectures 7

2.4 Verilog 7

2.5 Python 8

2.6 Simulation and Analysis Software 8

3 Binary Division Algorithm Implementation 9

3.1 Single Manipulation Algorithm 11
3.2 Final Manipulation Algorithm 14

4 Simulation and Evaluation of the 4-bit Architecture 17
4.1 Simulation 17
4.2 Timing Analysis 18
4.3 Power Consumption Analysis 18
4.4 Area Usage Analysis 21

5 Future Work 24
5.1 Future and Complex Implementations 24
5.2 Optimizations 28

6 Conclusion 29

References 31

I Verilog Design Code 33

II Python Test Bench Generator 53

List of Figures

2.1 Paravartya Sutra in the decimal number system 5

2.2 Paravartya Sutra in the binary number system 6

2.3 Paravartya Sutra in the binary number system 6

3.1 Overall Algorithm 10

3.2 Data flow 10

5.1 Area estimation for rnh_div_singleX 25

5.2 Total area estimation 26

List of Tables

4.1 Power consumption distribution 19
4.2 Power consumption by module 20
4.3 Cell area by module 21

5.1 Estimated single manipulation module values 25

Chapter 1

Introduction

Computers at their lowest level are just bit manipulators. They conduct arithmetic operations to process, compute, and evaluate binary numbers. These numbers can be ASCII characters, pixel colors, condition values for operations, and many more identifiers. The most common operations are addition, subtraction, and logic AND, OR, and NOT. Part of the reason for these operations are their simplicity and ability to compute the data in a short period of time. However, many processors use additional operations to compute greater amounts data efficiently and effectively. Examples of these operations are multiplication and division. However, to use these operations, large arithmetic blocks must be designed for each function and most likely have a latency in their output to the device. This paper describes a pipe-lined division core that can be used in this case.

1.1 Research Goals

The first goal of this graduate paper is to research and understand a division algorithm and then use the algorithm to design a division core. The design will consist of Boolean logic formulas based on the conditions of the algorithm. The chosen algorithm will be based on simplicity to

work in a pipe-lined system and on the uniqueness of the actual design.

The second goal of this work is to create, simulate, and analyze using Verilog. The analysis will include an investigation on timing, power consumption, and cell area usage which using software. This will be used to predict the behavior and model of larger designs using the same algorithm.

The third, and most important goal of this paper, is to determine if the chosen algorithm could be successful when implemented in an actual processor or other devices. This will be done by analyzing the output data and complexity of the final design.

1.2 Contributions

The first contribution is the research of possible algorithms that can be applied to a pipe-lined division core and creation of a logic design based on this algorithm. Extensive research is completed in the search of algorithms that can be easily pipe-lined and modeled using Boolean logic. The algorithm must also have the ability to scale from different size bits ranging from small to large. In addition, choosing an algorithm that is not as well-known will create a more unique and possibly better division core. From the chosen algorithm, a logic based design is created. This design is modeled using a hardware description language (HDL) which is easily simulated and analyzed.

The second contribution of this research refers to the simulation of the design. A test bench generator must be created to output a test bench that simulates all possible combinations. This is created using a high-level programming language. The string manipulation abilities of this software allows for a simple text file containing every combination to be converted into an accurate test bench with inputs and output verification.

The next contribution is the analysis of the output data to predict the behavior of larger scale

model and implementations. Due to limitations, only a small model is created, simulated, and analyzed. Because of this, the larger sized division cores' timing, power consumption, and cell area are estimated. These predictions are used to justify and conclude the usefulness of this algorithm as a pipe-lined division core.

All contributions are completed and analyzed together to prove whether or not this algorithm could be successful if implemented in an actual processor. This research will provide possible choices for the algorithm and lead to an actual digital design. Simulations prove the accuracy of the design. Lastly, the analysis of the output data will lead to the conclusion and verdict about the successful implementation of the algorithm.

1.3 Organization

The organization of this graduate paper is as follows: Chapter 2 will discuss the background and preliminaries related to the proposed design's evaluation. Chapter 4 proposes the architectures for the Vedic division algorithm. Chapter 5 presents the simulation and attribute results. Further developments to the design with predictions to large-scale models of the system are given in Chapter 6 and the graduate paper is concluded in Chapter 7.

Chapter 2

Preliminaries and Background Research

Division algorithms from all around the world and history were considered as a possibility for this paper. In addition, research was done to help complete the task. The research conducted is presented below.

2.1 Pipelined System

A pipelined system is defined as a form of parallelism within a single core. It provides a faster throughput than in a sequential system where one manipulation finishes before another one is started[1]. The pipelined component of the designed division core will be that it computes multiple elements at a time. However, it can only take in one input and output one value at a time.

2.2 Vedic Division Algorithm

The chosen division algorithm used to compute both the divisor and remainder is based on Vedic mathematics. Vedic mathematics are the mathematical functions used in the Vedic period (1500-500 BCE) in India[2]. There is evidence that numbers as large as 10^{12} were being used at this

Divisor			Dividend			
1	2	5	2	3	7	1
	-2	-5		-4	-10	
					2	5
			2	-1	-1	6
			20-1-1=18		125-10+6=121	

Figure 2.1: Paravartya Sutra in the decimal number system

time in history[3]. Arithmetic used in this period was not for computation in processors, it was for rituals. One ritual required a fire-altar to be created that has a square base with five layers of bricks and 21 bricks in each layer. They would have to divide one square into three equal parts and that part into seven smaller parts[4]. This may seem simple for today's algebra, but at the time, it was revolutionary.

The chosen algorithm is a special case of synthetic division called the Paravartya Sutra[5]. Synthetic division is a short cut method for dividing a polynomial by a linear divisor[6]. Fortunately, the algorithm is compatible with pipelining. One element is computed at a time and is used in the computation of the next element. Figure 2.1 shows an example in the decimal number system where the number 2371 divided by 125.

Essentially, the most significant bit/digit (MSB) of the divisor is dropped and the rest of the digits are negated. The negated part is multiplied by the MSB of the dividend and added to the next set of MSBs. Then the negated part is multiplied by the total number in the second MSB of the dividend. This last step repeats until it fills the length of the dividend. The length of the negated part of the divisor determines the number of digits that correspond to the remainder (R). The other digits represent the quotient (Q).

All the digits are summed and result the quotient and remainder. Because the remainder is purely negative, 1 must be subtracted from the quotient. To determine the positive remainder, the negative remainder must be subtracted from the original divisor and the positive digits of the

Divisor				Dividend								
1	0	0	1	1	0	0	0	0	1	1	0	1
	0	0	-1		0	0	-1					
						0	0	0				
							0	0	0			
								0	0	1		
									0	0	0	
										0	0	-1
				1	0	0	-1	0	1	10	0	0
				Q=100001-10=11101						R=1000		

Figure 2.2: Paravartya Sutra in the binary number system

Divisor				Dividend								
1	1	0	0	1	1	0	0	0	1	0	0	1
	-1	0	-1		-1	0	0					
						0	0	0				
							0	0	0			
								0	0	0		
									0	0	0	
										-1	0	0
				1	0	0	0	0	1	-1	0	1
				Q=100001-1=100000						R=1100-100+1=1001		

Figure 2.3: Paravartya Sutra in the binary number system

remainder are added (*i.e.* $125-10+6=121$) [7].

This algorithm is very confusing at first with the cases for negatives and positives, but the simplicity and ability to be a pipelined system makes it worthwhile to implement. An example of the algorithm in the binary number system is shown in figure 2.2 where 269 divided by 9 is calculated.

Another example is outlined in figure 2.3 where the division of 12 into 393 occurs. In this case, the first bit of the remainder is negative. To calculate the remainder, all negative bits are subtracted from the divisor and the positive bits are added. Also, an additional 1 is subtracted from the quotient because of the negative sign.

2.3 Similar Architectures

Similar designs have been made based on the Vedic division algorithm.

In one design, the dividend is a 16-bit binary number and the divisor was 8-bits. The delay is only about 10.5ns and only consumes about 24μW with a layout area of about 10.25mm^2[8].

Another design shows different computation speeds for different size dividends and divisors. The 16-bit system's longest output time is only 3.9μs compared to the worst case of 49.290μs for the non-restore or normal division algorithm[9].

2.4 Verilog

Verilog is one of the most popular hardware description languages (HDLs) with the others being SystemVerilog and VHDL. Verilog is used to create the proposed design in this paper. HDL is a language that supports the early conceptual stages of design with its behavioral and structural abstractions[10]. Verilog contains hierarchical constructs that allow the designer to control the complexity of the design. Verilog was first created in 1983-84 as a proprietary verification and simulation product by Phil Moorby. Eventually, synthesis and timing analyzer tools were added to the language. Input specification for logic and behavioral synthesis tools were also added later. Verilog was standardized by IEEE in 1995 (IEEE Standard: 1364-1995) and revisited again in 2001 (IEEE Standard: 1364-2001) and 2005 (IEEE Standard: 1364-2005)[11].

Verilog can be used to design and test register transistor level (RTL) circuit. The test is called a test-bench. The most common test-benches simulate different inputs into the design and checks the outputs for correctness. Different simulation software packages allow the user to view internal signals as well as the input and output signals. A good test-bench checks many different input values especially at all the extreme inputs.

Verilog is used to design the division module and all of its components.

2.5 Python

Python is a widely used high-level programming language. It was first implemented in 1989 [12] with the design philosophy of emphasizing on code readability[13]. This is shown in the code by the white-space and indentation instead of using brackets (that are used in C) or keywords (such as 'begin' and 'end' in Verilog). One of the other unique traits about Python is that type constraints, such as “int” or “string”, are not checked at compile time because they are assigned as needed.

A Python 2 script is used to create a test bench by reading a simplified text file. Python 2.0 was released on October 16th, 2000[14]. The script converts simple division value, such as $15/4$, into an input to the division module. The same generated testbench also checks for the output of the division statement x clock cycles later where x is the number of bits of the division module.

2.6 Simulation and Analysis Software

The design is both simulated and analyzed in this paper for correctness and operation.

The software used for simulation is Incisive Suite which includes SimVision (64) 15.10s-014[15]. SimVision is a graphical debugging environment developed and distributed by Cadence Design Systems. The software lets the user view individual waveforms of internal signals in the system throughout the entire run-time. This software is very useful in determining whether signals are correct or incorrect at any given time in the design.

For logic synthesis, DC Ultra is used. DC Ultra RTL synthesis is a software tool that computes timing, area, power, and test optimization concurrently and is developed and distributed by Synopsys. Synopsys has documented that the software gives results that correlate to physical implementations within 10% error[16]. This software will be used to calculate the timing, cell area, and power consumption of the design proposed by this graduate paper.

Chapter 3

Binary Division Algorithm Implementation

The preliminaries for the Vedic division algorithm are presented in this chapter. The different components of the algorithm are separated into each individual step and the final step. The overall design of this system is summarized in figure 3.1.

Figure 3.1 shows 6 different steps that are executed in the algorithm. The first step is to read in the input. The second, third, and fourth are to manipulate the dividend, carry, and sign bit using the single manipulation algorithm. Step 5 is the execution of the final manipulation algorithm and the final step, step 6, is to output the quotient and remainder.

The movement of data from each step is shown in figure 3.2. The figure shows the flow of the four main registers, which are the divisor, dividend, sign, and carry, throughout the design. The divisor remains the same and only the dividend, carry, and sign values change. The only other registers that get assigned values are the quotient and remainder registers at the end. It is also seen that the system can easily be used as a pipeline. Steps 2 through 5 represent a different clock cycle in the pipeline with continuous inputs from dividend_in and divisor_in and an output sent to quotient_out and remainder_out every clock cycle.

Figure 3.1: Overall Algorithm

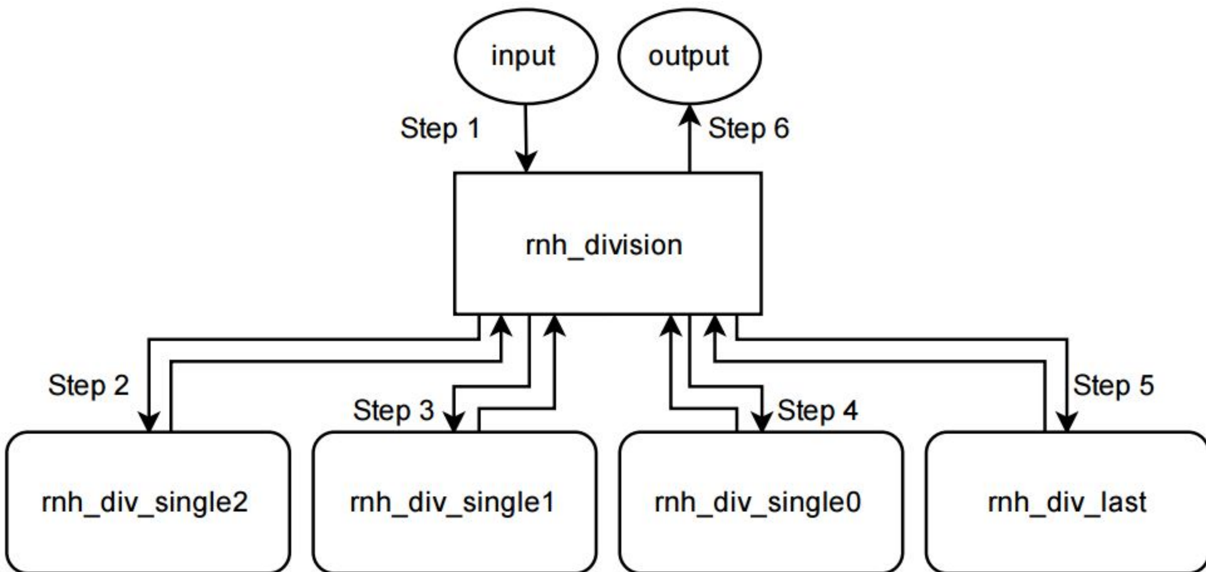
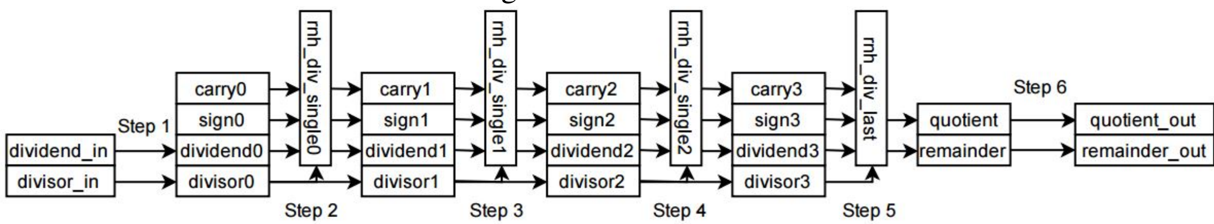


Figure 3.2: Data flow



3.1 Single Manipulation Algorithm

This section focuses only on the single step algorithm that is taken $N-1$ times where N is the number of bits. This algorithm gets more complicated as the number of bits increases, but the main algorithm stays roughly the same throughout. The modules that use this algorithm are named “rnh_div_single0”, “rnh_div_single1”, and “rnh_div_single2” with all three referenced as “rnh_div_singleX” in this paper.

The algorithm is broken down into three separate parts that generates three separate values. These values are for each characteristic word, updated dividend, carry, and sign. Each part is broken up into separate expressions for each individual bit of the characteristic word. Luckily, each expression is the same for each bit of the same characteristic word.

The expressions for these characteristic words are created from truth tables that are generated using different scenarios of current and previous bits using an online solver [17]. The conditional bits are the divisor, most significant dividend, carry, and sign bits for the operation. Current dividend, current carry, and current sign bits are used in the expressions to evaluate the characteristic words. In the previously described algorithm addition and subtraction occur. For these algorithms, it is simplified to a Boolean logic statement for each individual bit. Two logic statements are created for each output because of the two different conditions from the sign bit, negative or positive. This bit will determine if the operation being emulated is addition or subtraction.

The first value, the updated dividend, starts off as the original dividend value given to the division core. The dividend will determine when any manipulation will occur. When the manipulation occurs, the dividend value is adjusted and a new updated dividend is created. The new updated dividend is used to determine if and when another manipulation occurs. This process then repeats itself until all manipulations have occurred.

The number of bits being manipulated is determined by which manipulation step the current

dividend is at. The number of bits corresponds to $-M+4$ where M is the number of the current step.

The equations to calculate the new dividend, shown as D_{new}, is:

$$D_{new} = \bar{V} * D_{current} + V * \overline{D_{current}} + V * D_{current} * C_{current} \quad (3.1)$$

This equation does not change based on the sign of the manipulation bit.

In the equations, D_{new} is the new assigned value to the dividend bit, C_{new} is the new assigned value to the carry bit, S_{new} is the new assigned value to the sign bit, V is the divisor bit that is manipulating the value, $D_{current}$ is the current dividend bit, $C_{current}$ is the current carry value, and $S_{current}$ is the current sign value. Multiplication signs, $*$, are equivalent to a logic AND and addition signs, $+$, are equivalent to a logic OR. The line seen above some of the variables signifies a negation of that variable.

The second value calculated is the carry characteristic word. The carry bits correspond to each dividend bit directly. Because of this, the same corresponding carry bits change when the dividend bits change.

The carry bit works as the carry out when the sum of the dividend is greater than 1. This bit is important because it allows the dividend bit to be larger than 1 which is necessary for many cases. The current manipulation bit's sign determines which of the two equations are used to calculate the new bits.

The equations to calculate the new carry bit follows.

If the manipulation sign bit is positive:

$$C_{new} = \overline{V} * C_{current} + C_{current} * D_{current} + C_{current} * S_{Current} + V * D_{current} S_{current} + C_{current} V * \overline{D_{current}} \quad (3.2)$$

If the manipulation sign bit is negative:

$$C_{new} = C_{current} + V * D_{current} * \overline{S_{current}} + V * C_{current} \quad (3.3)$$

The sign characteristic word is the third and final value that is calculated. Simply put, it shows whether the dividend value at the shared bit location is negative or positive. For simplicity, 0 corresponds to positive and 1 corresponds to negative.

The sign bit is very important because it determines whether the divisor is added or subtracted from the current dividend. In other words, it determines which equation will occur. The current manipulation bit's sign determines which of the two equations are used to calculate the new bits.

The equations to calculate the new sign bit follows.

If the manipulation sign bit is positive:

$$S_{new} = S_{current} + (V * \overline{C_{current}} * \overline{D_{current}} + C_{current}) \quad (3.4)$$

If the manipulation sign bit is negative:

$$S_{new} = S_{current} * V + S_{current} * C_{current} \quad (3.5)$$

The only way each of these manipulation occurs is if the most significant dividend bit or carry bit for the operation is 1. If the bit is 0, then no addition or subtraction will occur which leads to no manipulation to the dividend, carry, and sign values.

These manipulations will occur for every bit in the dividend except the least significant bit (LSB). After the last single manipulation finishes the final manipulation will occur.

3.2 Final Manipulation Algorithm

This section focuses on the algorithm used to generate the final quotient and remainder based on the output from the previous single steps. This module is named “rnh_div_last” within the design and this paper.

The divisor is compared to the final calculated dividend to determine which digits represent the remainder and which represent the quotient. The number of remainder digits correspond to $I-1$ where I is the number of digits in the divisor.

The first bit evaluated is the MSB sign bit of the remainder. The MSB for the divisor is the most significant digit that has value (dividend is 1 or carry is 1). If the sign bit is 1, the remainder’s MSB dividend bit and carry bit are negative. If this is the case, the negative remainder digits are subtracted from the original divisor and the positive digits are added. If the bit is 0, the remainder’s MSB dividend bit and carry bit are positive. This simplifies the process, by only requiring the negative remainder digits to be subtracted from the positive digits. If any carry bits are set, they will be added or subtracted to one bit shifted to the left.

The formula for the remainder for each case follows, but unlike the above equations, the addition symbol, $+$, actually means addition.

If the remainder MSB sign bit is positive:

$$R = D[M_R:0] * \overline{S[M_R:0]} + \{S[M_R:0] * C[M_R:0] - S[M_R:0] * C[M_R:0], 0\} - \\ S[M_R-1:0] * D[M_R-1:0] \quad (3.6)$$

If the remainder MSB sign bit is negative:

$$R = D[N:0] - \{D[M_R:0] * S[M_R:0] - \{S[M_R:0] * C[M_R:0] - S[M_R:0] * C[M_R:0], 0\}\} + \overline{S[M_R:0]} * D[M_R:0] \tag{3.7}$$

In these equations, R is the remainder output, Q is the quotient output, N is the number of total bits, M_R is the most significant bit of the remainder, D is the entire dividend value, S is the entire sign value, and C is the entire carry value. The brackets after a value represent the bits that are used. For example, $D[M_R:0]$ means that the bits from the most significant bit of the remainder, M_R, down to 0, the least significant bit (LSB), are referenced. Whenever brackets are used, $\{\}$, the bits separated by commas are concatenated together in order of their appearance in the formula.

The final output evaluated is the quotient. The negative digits of the quotient are subtracted from the positive digits of the quotient. If the MSB of the remainder was determined to be negative, an additional subtraction of 1 occurs to account for the divisor subtraction. If any carry bits are set, they will be added or subtracted to one bit shifted to the left.

The formulas for the quotient follow.

If the remainder MSB sign bit is positive:

$$Q = D[N:M_R+1] * S[N:M_R+1] + \{\overline{S[N-1:M_R+1]} * C[N-1:M_R+1] - S[N-1:M_R+1] * C[N-1:M_R+1], 0\} - S[N-1:M_R+1] * C[N-1:M_R+1] \tag{3.8}$$

If the remainder MSB sign bit is negative:

$$\begin{aligned}
Q = D[N:M_R+1] * S[N:M_R+1] + \{\overline{S[N-1:M_R+1]} * C[N-1:M_R+1] - \\
S[N-1:M_R+1] * C[N-1:M_R+1], 0\} - \\
S[N-1:M_R+1] * C[N-1:M_R+1] - D[M_R] - C[M_R]
\end{aligned} \tag{3.9}$$

Once all the necessary equations are evaluated, the quotient and remainder are outputted.

Chapter 4

Simulation and Evaluation of the 4-bit Architecture

The section summarizes the simulation and evaluation of the proposed design as a 4-bit architecture. Simulations are completed to evaluate the correctness of the design and analysis is executed to measure values for timing, power consumption, and area usage.

In this paper, two software packages are used for simulation. These software packages are Cadence Incisive (64) and Synopsys DC Ultra. Incisive (64) is used to simulate and check the correctness of the overall design. DC Ultra is used for logic synthesis and also to calculate and report the final parameters of the design.

4.1 Simulation

To create the test-bench used in the simulation, a Python script is created. This Python script converts a simple division statement such as $15/4$ into an input that can be sent to division module. The same generated test-bench also checks for the output of the division statement x

clock cycles later where x is the number of bits of the division module (in this case, $x = 4$). If the output is no correct, an error statement is send to the compiler's output window.

All possible combinations up to 4-bits are tested in this simulation (*i.e.* $0/1$, $0/2$, ..., $0/15$, $1/1$, ...,$1/15$, ..., $15/1$, ..., $15/15$). No error statements are given which ensures correct implementation of the proposed design.

4.2 Timing Analysis

The timing is analyzed using a $50MHz$ ($20ns$ period) clock signal. It is necessary to solve for timing in every design because it can place limitations on the entire system. These limitations come from the delay in time it takes for the clock to reach different components in the design. The largest delay is the limiting factor of the entire design.

Using the simulation and analyze software, it is determined that the largest delay is $1.6529ns$. This is the needed width of the minimum clock period. This can be converted to a frequency using equation 4.1.

$$f = \frac{1}{T} \tag{4.1}$$

The necessary frequency is determined to be $403.332MHz$. This frequency is very good because it indicates that this division core will work with processor designs that use clock speeds up to this amount.

4.3 Power Consumption Analysis

In this section, the power consumption of the 4-bit design is analyzed and outputted. For the test, it is assumed the voltage on the system is 1V.

Table 4.1: Power consumption distribution

Power Group	Internal Power (μW)	Switching Power (μW)	Leakage Power (nW)	Total Power (μW)	% of Total
register	33.866	0.58848	16.7816	34.471e-02	86.%
combinational	3.2895	2.2737	50.5466	5.6138e-03	14.%
Total	37.156	2.8622	67.3282	40.085	100.%

Power consumption is one of the most important aspects when designing a system. The goal of every design is to make the this as minimal as possible. A low power consumption will allow batteries to last longer and will require less cooling. One of the biggest issues is leakage power which is power not consumed by the device to perform its necessary task.

The power consumption is outlined in table 4.1. The columns describe the individual powers that are consumed. Internal power is the energy needed to control each of the components such as registers or logic gates. Switching power is the power needed to switch a gate or register from one state to another. Leakage power is the amount of power that is wasted by the system when maintaining a certain value or when parts of the design is not being used. Total power is the sum of all the powers and the % of the total is the percentage of the sum amount of power consumed by the individual power group.

The two power groups are register and combinational which are represented in the rows. **Register** stands for the registers that consume power by storing and maintaining data every clock cycle so that it can be used in the next clock cycle. Pipelined systems use registers to store values at each step of the pipeline so other stages are not affected. **Combinational** stands for the combinational gates that compute a value, but do not store that value. The last row is the Total which represents the total of each column.

It is also seen that the total dynamic power usage is $40.0180\mu W$. Ninety-three percent of that is the cell's internal power at $37.1558\mu W$ and 7% is the net switching power at $2.8622\mu W$.

The dynamic power of each individual module of the overall division module is also deter-

Table 4.2: Power consumption by module

Module	Switching power (μW)	Internal power (μW)	Power Leakage (nW)	Total power (μW)	% of Total
rnh_div_last	1.28	6.46	40.073	7.78	19.4%
rnh_div_single2	0.248	6.31	5.734	6.56	16.4%
rnh_div_single1	0.417	6.53	7.473	6.95	17.3%
rnh_div_single0	0.540	6.55	8.571	7.10	17.7%
rnh_division	0.405	11.4	5.54	11.7	29.2%
Total	2.89	37.2	67.395	4.01	100.%

mined. This data can be found in table 4.2. Similar to the previous table, the columns describe the individual powers that are consumed.

Each row represents a different module in the design. rnh_div_last represents the last manipulation that outputs the quotient and remainder. The three modules named rnh_div_single0, rnh_div_single1, and rnh_div_single2, are each of the single modules that compute one of the 3 steps in the 4-bit design. There are 4 steps in a 5-bit design, 5 steps in a 6-bit design, and so on. The top-level design module is rnh_division. This module contains mostly registers and very few combinational logic gates. The last row, the Total row, represents the total of each column.

It can be seen in the table that most of the power consumption is found in the highest level of the design where all the values for sign, carry, dividend, and divisor for each step are stored. The next highest is rnh_div_last which account for the final manipulation that determines the quotient and remainder based on all other manipulations. This makes sense due to the amount of computing that occurs during this manipulation. However, 59.5% of the power leakage is from this module. This shows that some design changes may be needed to optimize the performance of the last algorithm.

Using the data for the three single modules, rnh_div_single0, rnh_div_single1, and rnh_div_single2, an equation can be derived to estimate the power usage of larger single modules. The line fit is selected to be logarithmic because the power consumption should increase as

Table 4.3: Cell area by module

Module	Combinational cell area (μm^2)	Non-combinational cell area (μm^2)	Total cell area (μm^2)	% of total
rnh_div_last	731.88	83.52	815.40	47.63%
rnh_div_single2	59.76	125.28	185.04	10.81%
rnh_div_single1	108.36	125.28	233.64	13.65%
rnh_div_single0	141.48	125.28	266.76	15.58%
rnh_division	2.16	208.80	210.96	12.32%
Total	1043.64	668.16	1711.80	100.%

the system becomes larger and approach an asymptotic value.

The logarithmic equation that is generated has an R^2 value of 0.9879 related to the original data. R^2 is coefficient of determination that indicates the proportion of variance in the dependent variable that is predictable from the independent variable[18]. In other words, the closer the R^2 value is to 1, the closer the line is to the data. This equation can be found below.

$$\frac{0.4951 * ln(x) + 6.5766}{1000} \tag{4.2}$$

4.4 Area Usage Analysis

When designing devices, cell area should be as small as possible. It costs more to manufacture large devices at both the device level and packaging level. Also, as the components of the device get smaller, the number of components that can be implemented in the device increases.

The overall cell area is $1711.80\mu m^2$. The module by module breakdown can be found in table 4.3. The columns are broken down into 2 major category sets and 2 conclusion data sets. The 2 major category sets are **combinational** area and **non-combinational** area. Non-combinational area comes from sequential elements in the design such as registers. Combinational area comes from the logic gates. The last two columns, total area and % of total, represent the total area for that row and the percentage of the overall sum that total is.

The rows in this table are similar to the rows used in table 4.2.

Unlike power consumption where the top level adds the most to the overall number, rnh_div_last is takes up the most space. This is the last manipulation completed to output the quotient and remainder. The combinational cell area is almost 90% of its total area, which is understandable since there are many manipulations happening across all of the bits.

The same reasoning justifies the size of the non-combinational area of the top level design, rnh_division. This design contains almost no logic, instead it just stores values in registers for the entire system. It also explains why all the rnh_div_singleX designs have the same non-combinational cell area because they all have the same amount of register outputs.

The combinational logic decreases from rnh_div_single0 down to rnh_div_single2 due to fewer manipulations occuring. It can be seen that when more bits are being manipulated, such as in rnh_div_single0, the area increases to the step in a logarithmic trend. Using the data, it can be found that the increase follows the trend shown in equation 4.3 where x is the number of bits being manipulated. Similar to equation 4.2, the fit will be a logarithmic trend because the cell area will always increase with the overall size and approach an asymptote.

$$f(x) = 73.925 * ln(x) + 59.048 \tag{4.3}$$

This logarithmic equation fits the original data with an R^2 of 0.9983.

To make a complete estimation of the total area of the single modules, an expression must be made that includes both the combinational and non-combinational cell areas. Note that the non-combinational area must also be added for each single manipulation. The expression for this total cell area for the single modules can be found in equation 4.4 and simplified in equation 4.5 where$y$ is the total bit width. The value 10.44 in equation 4.4 comes from the constant size of a

register for one bit and the value 3 comes from the number of characteristic word registers.

$$\sum_{x=1}^{y} f(x) + y * 3 * 10.44 * (y-1) \tag{4.4}$$

$$\sum_{x=1}^{y} f(x) + 31.32(y^2 - y) \tag{4.5}$$

This equation can be used to estimate larger scale designs and what the cell areas could be.

Chapter 5

Future Work

This chapter outlines how to upgrade the design to perform larger and more complex implementations. It also outlines what could possibly be upgraded or optimized to result in a better and more efficient system.

5.1 Future and Complex Implementations

To implement the design proposed in larger and more complex systems, more components must be added to the design.

For each bit that is added to the design, a new single manipulation module must be created. Each new module must contain one more bit operation than the previous one. This means that the size of the single manipulation modules will increase exponentially. The estimated sizes and power consumption values are outlined in table 5.1 based on the data collected from the 4-bit design using equations 4.2 and 4.5. A plot of the areas can be found in figure 5.1 and the total area per bit is plotted in figure 5.2.

The columns of the table are set up with bit width being the main variable. Bit width is the

Table 5.1: Estimated single manipulation module values

Bit width #	# of Manipulations	Estimated cell area (μm^2)	Total cell area (μm^2)	Estimated power usage (μW)	Total power usage (μW)
4	3	140.26	685.44	7.12	20.6
5	4	161.53	1097.53	7.26	27.9
6	5	178.03	1588.76	7.37	35.3
7	6	191.50	2156.10	7.46	42.7
8	7	202.90	2797.48	7.54	50.3
16	15	259.24	10464.97	7.92	112.
32	31	312.91	38672.90	8.28	243.
64	63	365.33	144861.88	8.63	514.

Figure 5.1: Area estimation for rnh_div_singleX

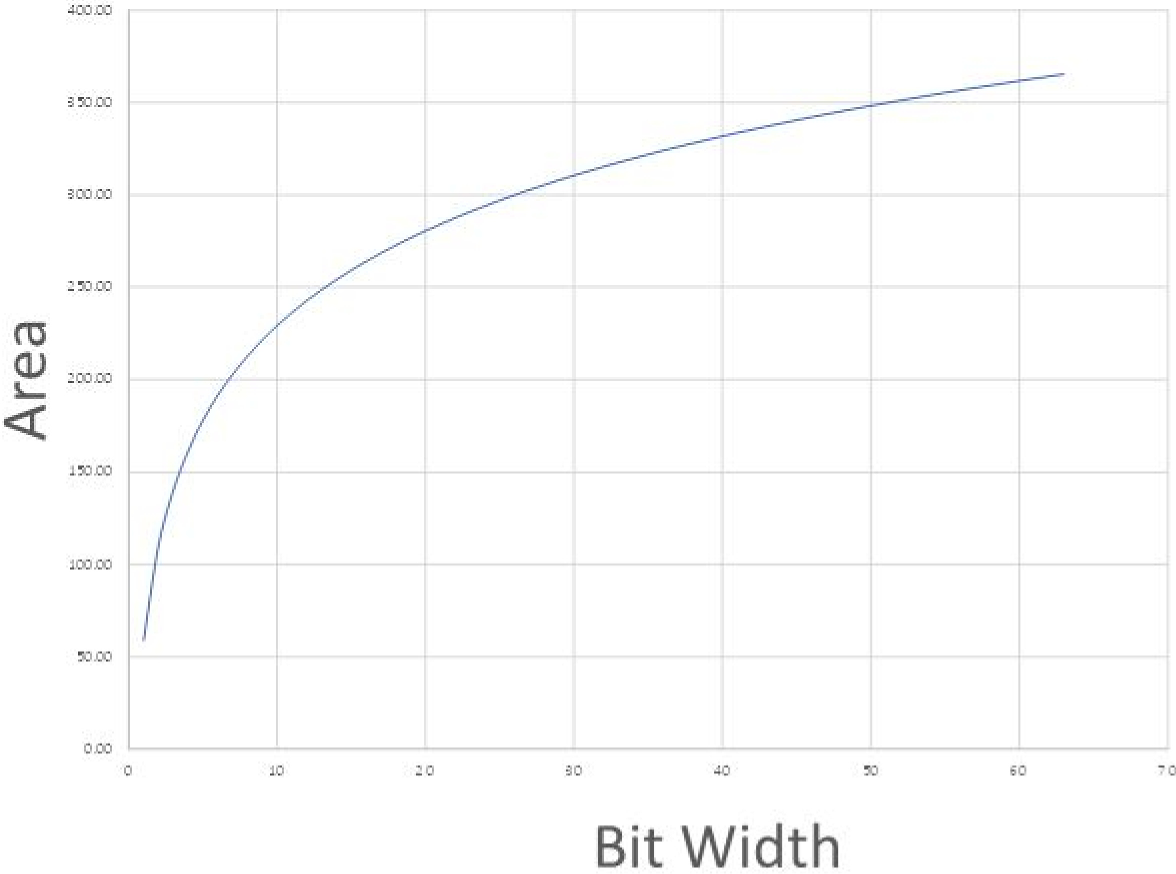
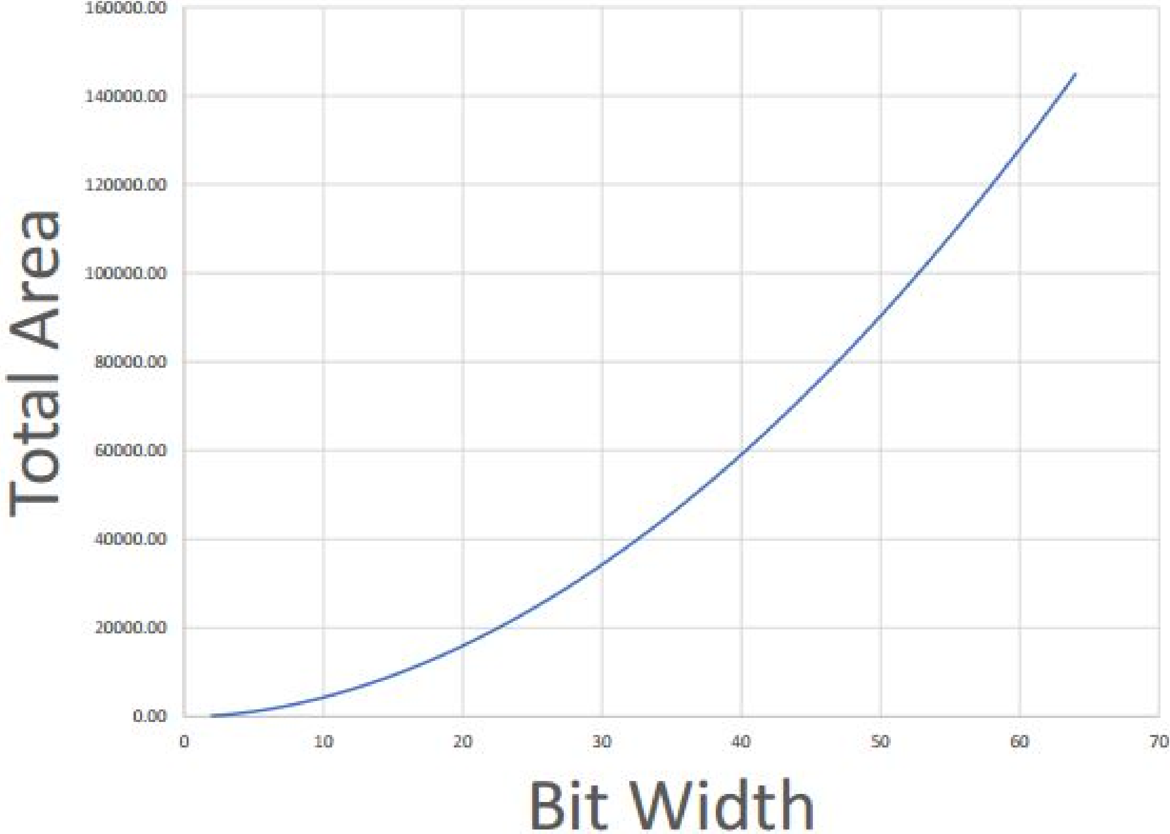


Figure 5.2: Total area estimation



first column. The values 4 through 8 are given to show consecutive numbers and the bit values 16, 32, and 64 are given as the common bit widths for integer operations. The second column is the number of manipulations that must occur before the quotient and remainder can be calculated. This value happens to be one less than the bit width number. The third column is the estimated combinational cell area for the largest rnh_div_singleX for the bit width. The next column is the estimated total cell area for all the rnh_div_singleX modules combined. Estimated power usage is the consumption from the largest rnh_div_singleX module for the given bit width. The final column is the total power consumption of all the rnh_div_singleX modules. It is estimated that this total power usage is about 50% of the overall total power consumption. For instance, the overall total power consumption for a 16-bit design is estimated to be about $224\mu W$. Similarly, the cell area will be about 40% of the total design's cell area leading to an estimated overall cell area of $26000\mu m^2$for a 16-bit design.

It can be seen from the estimated data that the total area for a bit width increases exponentially as expected. However, the estimated power consumption ends up being linear though it is expected to be exponential. This may be because the estimation of the total power usage does not include an increase for the additional bits stored in additional registers for each rnh_div_singleX due to limitations in the original testing. This would also increase linearly due as the rise of the bit number. This increase along to the estimated linear total power usage would combine to make an exponential increase.

It is expected that timing should remain roughly the same when the bit width increases. This is due to the all the modules being very close to the top level design. In other words, timing is expected to not be the limiting factor of larger designs.

In addition to the exponential findings, upgrades must be performed on the algorithm to account for some extra states that were not designed or tested for. The extra states result when the most significant dividend bit being acted upon has a carry bit value of 1. Due to this, the ex-

pressions to update the dividend, carry, and sign values change. In some cases, the dividend and carry value will also have to exceed the limitation of 2 bits, which leads to even more complexity. Because of this, the design will work best with smaller and simpler systems.

5.2 Optimizations

The proposed system works completely and efficiently, but there are some areas for improvement.

One area for improvement is the overall system evaluating every single manipulation module. This it isn't necessary. The current design requires every bit to be checked for a possible manipulation. A better design could can perform more efficiently by only checking for the most significant 1. It could also stop checking for manipulations after the entire divisor has been completely used. This could lead to less power consumption of the entire system and therefore optimizing the design. The downside of this optimization is more controls would needed to be implemented which result is higher cell area usage.

Due to the large amount of power loss from the module rnh_div_last, optimizations should be made to limit this. Combining logic together to limit the number of logic gates that are not used in every scenario will help bring the power loss down.

Another example of optimization is the final manipulation to solve for the quotient and remainder. The final manipulation could be completed in earlier steps, resulting in the throughput latency being one clock cycle less. However, this results in the algorithm not being straight forward.

Chapter 6

Conclusion

A model of a 4-bit division core based on an ancient Vedic mathematics operation is successfully designed and created. The core is modeled using Verilog and simulated using Cadence's Incisive Suite. The design is synthesized and analyzed using Synopsis's DC Ultra and used to estimate the operation larger models.

The design is successfully simulated with every possible input combination and checked for correct operation using a generated test-bench. The test-bench is generated using Python which allows for future test-benches to be created and implemented easily.

In timing analysis, the design has very little impact. The maximum allowable clock is found to be about $400MHz$ as opposed to the $50MHz$ clock used to analyze the design. This means that timing will not be a factor when implementing this design in larger models or in processors. With a clock cycle of $400MHz$, 16-clock cycles would take $40ns$. This means that the worst case for a 16-bit system would take about $40ns$ to compute the quotient and remainder. In comparison to a similar design based on the same algorithm, the worst case is $3.9\mu s$[9]. This shows current design may end up being more efficient when considering timing. However, more research and experimentation must be completed to conclude this.

The total power consumption is $40.085\mu W$ with $67.3282nW$ being consumed as leakage. It is estimated that in a 16-bit design, the total power consumption will increase exponentially towards $224\mu W$. After comparing this to a similar design showing only $24\mu W$[8], it is clear that this design needs to be reworked.

The cell area usage is $1711.80\mu m^2$. It is predicted that the area will increase exponentially and end up being $0.026mm^2$ for a 16-bit design. Comparing this to $10.25mm^2$ in a similar design[8], it can be seen that while power consumption is higher, the design takes up significantly less space.

From the comparisons, it can be seen that some improvements must be made in order to increase the ability of the design. Different components must be added to the current design to allow for the evaluation of all the combinations of inputs into the Vedic division algorithm. In addition, the power consumption must be minimized by modifying the design to not consume as much power when evaluating the quotient and remainder especially in the final algorithm implementation.

Overall, the possibility of this design to be implemented in a processor or other embedded systems is quite high. After some optimizations and upgrading, this design could fully function in many different sized bit systems by replacing older, slower, or less optimized division cores. The recommendation of this graduate paper is to continue research and development of this method so that it can be implemented in an actual physical design in the future.

References

[1] J. Squire. Cmsc 411 lecture 19, pipelining data forwarding.

[2] Kenneth Pletcher. *The History of India*. Britannica Educational Pub. in association with Rosen Educational Services, 2011.

[3] Gavin D. Flood. *The Blackwell companion to Hinduism*. Blackwell, 2005.

[4] Pierre-Sylvain Filliozat. Ancient sanskrit mathematics: An oral tradition and a written literature. *History of Science, History of Text Boston Studies in the Philosophy of Science*, 2004.

[5] Bharatikrshnatirtha and Vasudeva S. Agrawala. *Vedic mathematics, or, sixteen simple mathematical formulae from the Vedas (for one-line answers to all mathematical problems)*. Motilal Banarsidass, 1970.

[6] Lianghuo Fan. A generalization of synthetic division and a general theorem of division polynomials. *Mathematical Medley*, 30(1), Jun 2003.

[7] Swami Bharati Krishna Tirtha and Vasudeva Sharana. *Vedic Mathematics*. Motilal Banarsidass Publishers Private Limited, 2004.

[8] Prabir Saha, Arindam Banerjee, Partha Bhattacharyya, and Anup Dandapat. Vedic divider:

Novel architecture (asic) for high speed vlsi applications. *2011 International Symposium on Electronic System Design*, 2011.

[9] D. Sengupta, M. Sultana, and A. Chaudhuri. An algorithm facilitating fast bcd division on low end processors using ancient indian vedic mathematics sutras. *2012 International Conference on Communications, Devices and Intelligent Systems (CODIS)*, 2012.

[10] D. E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1991.

[11] Verilog. *Introduction to Verilog*, 2015.

[12] Guido van Rossum. The history of python, Jan 1970.

[13] Python. General python faq. 2017.

[14] Moshe Zadka and A M Kuchling. Whats new in python 2.0, Oct 2000.

[15] Cadence. *SimVision Debug*.

[16] Synopsis. *DC Ultra*, 2015.

[17] 32x8.com. Logic circuit simplification (sop and pos).

[18] Stat Trek. *Statistics and Probability Dictionary*. 2017.

Appendix I

Verilog Design Code

```
1  module rnh_division (reset, clk, divisor_in, dividend_in,
     quotient_out, remainder_out, scan_in0, scan_en, test_mode,
     scan_out0);
2  input
3      reset,                      // system reset
4      clk;                        // system clock
5  input
6      scan_in0,                   // test scan mode data input
7      scan_en,                    // test scan mode enable
8      test_mode;                  // test mode select
9  output
10     scan_out0;  // test scan mode data outputmodule
         rnh_division
11
12     input [3:0] divisor_in, dividend_in;
```

```
13
14      output [3:0] quotient_out, remainder_out;
15      wire [3:0] divisor_in, dividend_in;
16
17      wire [3:0] quotient_out, remainder_out;
18
19  //———— internal siganls ——————————//
20      reg [3:0] divisor0; // 4 4-bit registers for divisor
21      reg [3:0] dividend0; // 4 4-bit registers for dividend
22
23      wire [3:0] dividend_out [3:0]; // 4 4-bit wires for
          dividend outputs
24      wire [3:0] carry_out [2:0];
25      wire [3:0] sign_out [2:0];
26      reg [3:0] divisor1, divisor2, divisor3;
27      wire [3:0] dividend1, dividend2, dividend3;
28      wire [3:0] carry1, carry2, carry3;
29      wire [3:0] sign1, sign2, sign3;
30
31  //—————— connected modules ————————————//
32  rnh_div_single0 my_div0 (clk, reset, divisor0, dividend0, 4'
      b0000, 4'b0000, dividend1, carry1, sign1,
33              scan_in0, scan_en, test_mode, scan_out0);
34  rnh_div_single1 my_div1 (clk, reset, divisor1, dividend1,
      carry1, sign1, dividend2, carry2, sign2,
```

```
35              scan_in0 , scan_en , test_mode , scan_out0 ) ;
36 rnh_div_single2 my_div2 (clk , reset , divisor2 , dividend2 ,
       carry2 , sign2 , dividend3 , carry3 , sign3 ,
37              scan_in0 , scan_en , test_mode , scan_out0 ) ;
38 rnh_div_last my_last (clk , reset , divisor3 , dividend3 , carry3 ,
       sign3 , quotient_out , remainder_out ,
39              scan_in0 , scan_en , test_mode , scan_out0 ) ;
40 //---------- module begin ------------------//
41
42 always@(posedge clk or posedge reset) begin
43     if (reset == 1'b1) begin
44         divisor0 = 4'b0000; divisor1 = 4'b0000; divisor2 = 4'
             b0000; divisor3 = 4'b0000;
45         dividend0 = 4'b0000;
46     end
47     else begin
48       divisor3 = divisor2;
49       divisor2 = divisor1;
50       divisor1 = divisor0;
51       divisor0 = divisor_in;
52       dividend0 = dividend_in;
53     end
54 end
55 endmodule
56
```

```
57 module rnh_div_single0 (clk, reset, divisor, dividend, carry,
     sign, dividend_out, carry_out, sign_out, scan_in0, scan_en,
     test_mode, scan_out0);
58      input clk, reset;
59      input [3:0] divisor, dividend, carry, sign;
60      output [3:0] dividend_out, carry_out, sign_out;
61 input
62      scan_in0,                    // test scan mode data input
63      scan_en,                     // test scan mode enable
64      test_mode;                   // test mode select
65 output
66      scan_out0;   // test scan mode data outputmodule
         rnh_division
67
68      wire [1:0] bit_num;
69      wire [2:0] temp_div;
70
71      reg [3:0] dividend_out, carry_out, sign_out;
72
73 rnh_divisor_detect  my_detect   (divisor, bit_num, temp_div,
     scan_in0, scan_en, test_mode, scan_out0, reset, clk);
74      always@(posedge clk or posedge reset) begin
75    if (reset == 1'b1) begin
76      dividend_out = 4'b0000;
77      carry_out = 4'b0000;
```

```
78        sign_out = 4'b0000;
79     end
80     else begin
81        sign_out = sign;
82        carry_out = carry;
83        dividend_out = dividend;
84        if (bit_num <= 3) begin
85             casex({carry[3],dividend[3],sign[3]})
86                  3'bx11, 3'b1x1: begin
87             if (bit_num > 0 ) begin
88                       if (dividend[3] == 1'b1)
89             dividend_out[2] = (~(temp_div[bit_num-1]) & dividend
                   [2])|(temp_div[bit_num-1] & ~(dividend[2]) &
                   dividend[3]);
90                       carry_out[2] = carry[2] | ~(sign[2]) &
                            temp_div[bit_num-1] & dividend[2] | (
                            carry[3] & temp_div[bit_num-1]);
91                       sign_out[2] = (carry[2] & sign[2]) | (sign
                            [2] & ~(temp_div[bit_num-1]));
92             end
93             if (bit_num > 1) begin
94                       if (dividend[3] == 1'b1)
95             dividend_out[1] = (~(temp_div[bit_num-2]) & dividend
                   [1])|(temp_div[bit_num-2] & ~(dividend[1]) &
                   dividend[3]);
```

```
96                      carry_out[1] = carry[1] | ~(sign[1]) &
                            temp_div[bit_num-2] & dividend[1] | (
                            carry[2] & temp_div[bit_num-2]);
97                      sign_out[1] = (carry[1] & sign[1]) | (sign
                            [1] & ~(temp_div[bit_num-2]));
98          end
99
100         if (bit_num > 2) begin
101                     if (dividend[3] == 1'b1)
102         dividend_out[0] = (~(temp_div[bit_num-3]) & dividend
                [0])|(temp_div[bit_num-3] & ~(dividend[0]) &
                dividend[3]);
103                     carry_out[0] = carry[1] | ~(sign[0]) &
                            temp_div[bit_num-3] & dividend[0] | (
                            carry[3] & temp_div[bit_num-3]);
104                     sign_out[0] = (carry[0] & sign[0]) | (sign
                            [0] & ~(temp_div[bit_num-3]));
105         end
106           end
107           3'bx10, 3'b1x0 : begin
108         if (bit_num > 0) begin
109                     if (dividend[3] == 1'b1)
110         dividend_out[2] = (~(temp_div[bit_num-1]) & dividend
                [2])|(temp_div[bit_num-1] & ~(dividend[2]) &
                dividend[3]);
```

```
111                     carry_out[2] = (carry[2] & ~temp_div[
                            bit_num-1]) | (carry[2] & dividend[2]) |
                            (sign[2] & carry[2]) |
112             (sign[2] & temp_div[bit_num-1] & dividend[2]) | (
                    carry[3] & temp_div[bit_num-1]);
113                     sign_out[2] = sign[2] | (~carry[2] &
                            temp_div[bit_num-1] & ~dividend[2]);
114                 end
115           if (bit_num > 1) begin
116                     if (dividend[3] == 1'b1)
117           dividend_out[1] = (~(temp_div[bit_num-2]) & dividend
                 [1])|(temp_div[bit_num-2] & ~(dividend[1]) &
                 dividend[3]);
118                     carry_out[1] = (carry[1] & ~temp_div[
                            bit_num-2]) | (carry[1] & dividend[1]) |
                            (sign[1] & carry[1]) |
119             (sign[1] & temp_div[bit_num-2] & dividend[1]) | (
                    carry[3] & temp_div[bit_num-2]);
120                     sign_out[1] = sign[1] | (~carry[1] &
                            temp_div[bit_num-2] & ~dividend[1]);
121           end
122                 if (bit_num > 2) begin
123                     if (dividend[3] == 1'b1)
124           dividend_out[0] = (~(temp_div[bit_num-3]) & dividend
                 [0])|(temp_div[bit_num-3] & ~(dividend[0]) &
```

```
            dividend [3]);
125                       carry_out [0] = (carry [0] & ~temp_div [
                              bit_num −3]) | (carry [0] & dividend [0]) |
                              (sign [0] & carry [0]) |
126                 (sign [0] & temp_div [bit_num −3] & dividend [0]) | (
                       carry [3] & temp_div [bit_num −3]);
127                       sign_out [0] = sign [0] | (~carry [0] &
                              temp_div [bit_num −3] & ~dividend [0]);
128               end
129                 end
130                 default : begin
131             dividend_out [2:0] = dividend [2:0];
132             sign_out = sign;
133             carry_out = carry;
134         end
135             endcase
136             dividend_out [3] = dividend [3];
137       end
138     end
139       end
140 endmodule
141 module rnh_div_single1 (clk, reset, divisor, dividend, carry,
       sign, dividend_out, carry_out, sign_out, scan_in0, scan_en,
       test_mode, scan_out0);
142       input clk, reset;
```

```
143     input [3:0] divisor, dividend, carry, sign;
144     output [3:0] dividend_out, carry_out, sign_out;
145 input
146     scan_in0,               // test scan mode data input
147     scan_en,                // test scan mode enable
148     test_mode;              // test mode select
149 output
150     scan_out0;              // test scan mode data
           outputmodule rnh_division
151
152     wire [1:0] bit_num;
153     wire [2:0] temp_div;
154
155     reg [3:0] dividend_out, carry_out, sign_out;
156
157 rnh_divisor_detect  my_detect  (divisor, bit_num, temp_div,
       scan_in0, scan_en, test_mode, scan_out0, reset, clk);
158     always@(posedge clk or posedge reset) begin
159   if (reset == 1'b1) begin
160     dividend_out = 4'b0000;
161     carry_out = 4'b0000;
162     sign_out = 4'b0000;
163   end
164   else begin
165     sign_out = sign;
```

```
166     carry_out = carry;
167     dividend_out = dividend;
168     if (bit_num <= 2) begin
169         casex({carry[2],dividend[2],sign[2]})
170       3'bx11, 3'b1x1: begin
171           if (bit_num > 0) begin
172                   if (dividend[2] == 1'b1)
173           dividend_out[1] = (~temp_div[bit_num-1] & dividend
              [1]) | (temp_div[bit_num-1] & ~dividend[1] &
              dividend[2]);
174                   carry_out[1] = carry[1] | ~(sign[1]) &
                        temp_div[bit_num-1] & dividend[1] | (
                        carry[2] & temp_div[bit_num-1]);
175                   sign_out[1] = (sign[1] & ~(temp_div[bit_num
                        -1])) | (sign[1] & carry[1]);
176           end
177
178           if (bit_num > 1) begin
179                   if (dividend[2] == 1'b1)
180           dividend_out[0] = (~temp_div[bit_num-2] & dividend
              [0]) | (temp_div[bit_num-2] & ~dividend[0] &
              dividend[2]);
181                   carry_out[0] = carry[0] | ~(sign[0]) &
                        temp_div[bit_num-2] & dividend[0] | (
                        carry[2] & temp_div[bit_num-2]);
```

```
182                     sign_out[0] = (sign[0] & ~(temp_div[bit_num
                          -2])) + (sign[0] & carry[0]);
183          end
184            end
185            3'bx10, 3'b1x0 : begin
186          if (bit_num > 0) begin
187                     if (dividend[2] == 1'b1)
188          dividend_out[1] = (~(temp_div[bit_num-1]) & dividend
               [1])|(temp_div[bit_num-1] & ~(dividend[1]) &
               dividend[2]);
189                     carry_out[1] = (carry[1] & ~temp_div[
                          bit_num-1]) | (carry[1] & dividend[1]) |
                          (sign[1] & carry[1]) |
190            (sign[1] & temp_div[bit_num-1] & dividend[1]) | (
                 carry[2] & temp_div[bit_num-1]);
191                     sign_out[1] = sign[1] | (~carry[1] &
                          temp_div[bit_num-1] & ~dividend[1]);
192          end
193                 if (bit_num > 1) begin
194                     if (dividend[2] == 1'b1)
195          dividend_out[0] = (~(temp_div[bit_num-2]) & dividend
               [0])|(temp_div[bit_num-2] & ~(dividend[0]) &
               dividend[2]);
196                     carry_out[0] = (carry[0] & ~temp_div[
                          bit_num-2]) | (carry[0] & dividend[0]) |
```

```
                         (sign[0] & carry[0]) |
197            (sign[0] & temp_div[bit_num-2] & dividend[0]) | (
                  carry[2] & temp_div[bit_num-2]);
198                       sign_out[0] = sign[0] | (~carry[0] &
                              temp_div[bit_num-2] & ~dividend[0]);
199          end
200            end
201            default : begin
202        dividend_out[1:0] = dividend[1:0];
203        sign_out = sign;
204        carry_out = carry;
205      end
206        endcase
207    end
208  end
209    end
210 endmodule
211 module rnh_div_single2 (clk, reset, divisor, dividend, carry,
      sign, dividend_out, carry_out, sign_out, scan_in0, scan_en,
      test_mode, scan_out0);
212    input clk, reset;
213    input [3:0] divisor, dividend, carry, sign;
214    output [3:0] dividend_out, carry_out, sign_out;
215 input
216    scan_in0,                    // test scan mode data input
```

```
217      scan_en,                    // test scan mode enable
218      test_mode;                  // test mode select
219  output
220      scan_out0;  // test scan mode data outputmodule
         rnh_division
221
222      wire [1:0] bit_num;
223      wire [2:0] temp_div;
224
225      reg [3:0] dividend_out, carry_out, sign_out;
226
227  rnh_divisor_detect  my_detect  (divisor, bit_num, temp_div,
       scan_in0, scan_en, test_mode, scan_out0, reset, clk);
228      always@(posedge clk or posedge reset) begin
229    if (reset == 1'b1) begin
230      dividend_out = 4'b0000;
231      carry_out = 4'b0000;
232      sign_out = 4'b0000;
233    end
234    else begin
235      sign_out = sign;
236      carry_out = carry;
237      dividend_out = dividend;
238      if (bit_num <= 1) begin
239          casex({carry[1],dividend[1],sign[1]})
```

```
240         3'bx11, 3'b1x1: begin
241             if (bit_num > 0) begin
242                 if (dividend[1] == 1'b1 | (sign[0] == 1'b1
                        & dividend[0] == 1'b1))
243             dividend_out[0] = (~(temp_div[bit_num-1]) & dividend
                [0]) | (temp_div[bit_num-1] & ~(dividend[0])) |
244                 (dividend[0] & temp_div[bit_num-1] & carry[1]);
245                 carry_out[0] = carry[0] | ~(sign[0]) &
                        temp_div[bit_num-1] & dividend[0] | (
                        carry[1] & temp_div[bit_num-1]);
246                 sign_out[0] = (sign[0] & ~(temp_div[bit_num
                        -1])) | (sign[0] & carry[0]);
247             end
248         end
249         3'bx10, 3'b1x0 : begin
250             if (bit_num > 0) begin
251                 if (dividend[1] == 1'b1 | (sign[0] == 1'b0
                        & dividend[0] == 1'b1))
252             dividend_out[0] = (~(temp_div[bit_num-1]) & dividend
                [0]) | (temp_div[bit_num-1] & ~(dividend[0])) |
253                 (dividend[0] & temp_div[bit_num-1] & carry[1]);
254                 carry_out[0] = (carry[0] & ~temp_div[
                        bit_num-1]) | (carry[0] & dividend[0]) |
                        (sign[0] & carry[0]) |
```

```
255                 (sign[0] & temp_div[bit_num-1] & dividend[0]) | (
                      carry[1] & temp_div[bit_num-1] & ~dividend[0]);
256                         sign_out[0] = sign[0] | (~carry[0] &
                              temp_div[bit_num-1] & ~dividend[0]) | (
                              carry[1]);
257                end
258                  end
259                  default : begin
260             dividend_out[0] = dividend[0];
261             sign_out = sign;
262             carry_out = carry;
263           end
264             endcase
265         end
266       end
267         end
268   endmodule
269
270   module rnh_div_last (clk, reset, divisor, dividend, carry, sign
        , quotient_out, remainder_out, scan_in0, scan_en, test_mode,
        scan_out0);
271         input clk, reset ;
272         input [3:0] divisor, dividend, carry, sign;
273         output [3:0] quotient_out, remainder_out;
274   input
```

```
275       scan_in0,                       // test scan mode data input
276       scan_en,                        // test scan mode enable
277       test_mode;                      // test mode select
278   output
279       scan_out0;  // test scan mode data outputmodule
            rnh_division
280       reg [3:0] quotient_out, remainder_out;
281       wire [1:0] bit_num;
282       wire [2:0] temp_div;
283       wire [1:0] rem_num;
284
285   rnh_divisor_detect  my_detect  (divisor, bit_num, temp_div,
        scan_in0, scan_en, test_mode, scan_out0, reset, clk);
286   rnh_remainder_detect my_rdetect (dividend, bit_num, rem_num,
        scan_in0, scan_en, test_mode, scan_out0, reset, clk);
287       always@(posedge clk or posedge reset) begin
288           if (reset == 1'b1) begin
289               quotient_out = 4'h0;
290               remainder_out = 4'h0;
291           end
292     else begin
293       case(bit_num)
294           3 : begin
295               if (sign[rem_num] == 1'b1) begin
```

```
296                 remainder_out = divisor[3:0] - {1'b0, ((3'b111
                      & dividend[2:0] & (sign[2:0])) - {(~(sign
                      [2:0]) & carry[2:0]) -
297                 (sign[2:0] & carry[2:0]),1'b0})}
298                   + (dividend[2:0] & ~sign[2:0]);
299           quotient_out = (1'b1 & dividend[3] & ~(sign[3])) +
                {(~(sign[2]) & carry[2]) - (sign[2] & carry[2]),1'
                b0}
300                                  - dividend[rem_num] - carry[2];
301             end
302             else begin
303                 remainder_out = (3'b111 & dividend[2:0] & ~(
                      sign[2:0])) + {(~(sign[2:0]) & carry[2:0]) -
304                                       (sign[2:0]& carry[2:0]),1'
                                            b0} - (sign[2:0] &
                                            dividend[2:0]);
305           quotient_out = (1'b1 & dividend[3] & ~(sign[3])) +
                {(~(sign[2]) & carry[2]) - (sign[2] & carry[2]),1'
                b0};
306             end
307         end
308         2 : begin
309
310             if (sign[rem_num] == 1'b1) begin
```

```
311                     remainder_out = divisor[2:0] - {1'b0, ( (2'b11
                          & dividend[1:0] & (sign[1:0]) ) - {(~(sign
                          [1:0]) & carry[1:0])-
312                     (sign[1:0] & carry[1:0]),1'b0}) }
313                       + (dividend[1:0] & ~sign[1:0]);
314                 quotient_out = (2'b11 & dividend[3:2] & ~(sign[3:2]))
                      + (~(sign[2]) & carry[2]) - (sign[2] & carry[2])
315                                     - (sign[2] & dividend[2]) -
                                           dividend[rem_num] - carry[1];
316                 end
317                 else begin
318                     remainder_out = (2'b11 & dividend[1:0] & ~(sign
                          [1:0])) + {(~(sign[1:0]) & carry[1:0]) -
319                                         (sign[1:0] & carry[1:0]),1'
                                            b0} - (sign[0] & dividend
                                            [0]);
320                 quotient_out = (2'b11& dividend[3:2] & ~(sign[3:2]))
                      + {(~(sign[2]) & carry[2]) - (sign[2] & carry[2])
                      ,1'b0}
321                                     - (sign[2] & dividend[2]);
322                 end
323             end
324             1 : begin
325                 if (sign[rem_num] == 1'b1) begin
```

```
326                 remainder_out = divisor[1:0] - {1'b0, ((1'b1 &
                        dividend[0] & sign[0])- {(~(sign[0]) & carry
                        [0])-(sign[0] & carry[0]),1'b0})}
327                 + (dividend[0] & ~sign[0]);
328             quotient_out = (3'b111 & dividend[3:1] & ~(sign[3:1])
                  ) + {(~(sign[2:1]) & carry[2:1]) - (sign[2:1] &
                  carry[2:1]),1'b0}
329                                         - (sign[2:1] & dividend[2:1])-
                                                dividend[rem_num] - carry[0];
330             end
331             else begin
332                 remainder_out = (1'b1 & dividend[0] & ~(sign
                        [0])) + {(~(sign[0]) & carry[0]) -
333                                                 (sign[0]& carry[0]),1'b0};
334             quotient_out = (3'b111 & dividend[3:1] & ~(sign[3:1])
                  ) + {(~(sign[2:1]) & carry[2:1]) - (sign[2:1] &
                  carry[2:1]),1'b0}
335                                         - (sign[2:1] & dividend[2:1]);
336             end
337         end
338         default : begin
339             quotient_out = dividend[3:0];
340             remainder_out = 4'h0;
341         end
342     endcase
```

```
343         if (remainder_out[3:0] != 0 && remainder_out[3:0] ==
               divisor[3:0]) begin
344           quotient_out = quotient_out + 1;
345           remainder_out = 4'h0;
346         end
347     end
348         end
349 endmodule
```

Appendix II

Python Test Bench Generator

```
1  # creates testbench file to test a 4-bit divisor
2
3  filename = "rnh_division_tb_txt.txt"
4  destfile = "rnh_division_test.v"
5  headfile = "rnh_division_tb_head.v"
6  footfile = "rnh_division_tb_foot.v"
7  NCfile = "rnh_division_tb_NC.txt"
8
9  error = 0
10
11 temp = open(NCfile,'w')
12 try:
13     with open(filename) as lines:
14         for line in lines:
15             temp.write(line.replace("%","\n%").split("%")[0])
```

```
16 except:
17     print("File %s not found\n" % filename);
18     error = error + 1
19
20 temp.close();
21
22 txt = open(NCfile,'r')
23 content = txt.readlines()
24
25 try:
26     try:
27       fileStart = content.index(".file_start\n") + 1
28     except:
29   fileStart = content.index(".file_start") + 1
30 except:
31     error = error + 1
32     print(".file_start NOT FOUND\n")
33
34 try:
35     try:
36         fileEnd = content.index(".file_end\n") - 1
37     except:
38         fileEnd = content.index(".file_end") - 1
39 except:
40     error = error + 1
```

```
41      print(".file_end NOT FOUND\n")
42
43  divisor = []
44  dividend = []
45  quotient = []
46  remainder = []
47
48  j = 0
49
50  for i in range(fileStart,fileEnd):
51      line = content[i]
52      if '/' in line:
53          sym_loc = line.index('/')
54          if int(line[0:sym_loc]) > 15 :
55              print("Dividend at line %s, %s, is too large" % (i
                    +2,line[0:sym_loc]))
56              dividend.append('0')
57              error += 1
58          else :
59              dividend.append(line[0:sym_loc])
60
61          if int(line[sym_loc+1:len(line)-1]) > 15 :
62              print("Divisor at line %s, %s, is too large" % (i
                    +2,line[sym_loc+1:len(line)-1]))
63              divisor.append('0')
```

```
64                  error += 1
65              else :
66                  divisor.append(line[sym_loc+1:len(line)-2])
67
68              if divisor[j] == '0' :
69                  quotient.append('0')
70                  remainder.append('0')
71
72              else :
73                  quotient.append(int(int(dividend[j])/int(divisor[j
                    ])))
74                  remainder.append(int(dividend[j])%int(divisor[j]))
75              j += 1
76
77  output = open(destfile,'w')
78
79  with open(headfile) as lines:
80      for line in lines:
81          output.write(line)
82
83  for i in range(0,j+4):
84      if (i >= 4 and i >= j) :
85          output.write("apply_test_vector(0, 0, 4'h0, 4'h0, 4'h%s
                , 4'h%s); // Na/Na; %s/%s = %s R:%s \n\t" % \
```

```
86                          ('{0:0x}'.format(int(quotient[i-4])),'{0:0
                              x}'.format(int(remainder[i-4])),\
87                          dividend[i-4],divisor[i-4],quotient[i-4],
                              remainder[i-4]))
88      elif (i < 4 and i >= j) :
89          output.write("apply_test_vector(0, 0, 4'h0, 4'h0, 4'h0,
              4'h0); // Na/Na; Na/NA = Na R:Na \n\t")
90      elif i >= 4 :
91          output.write("apply_test_vector(0, 0, 4'h%s, 4'h%s, 4'h
              %s, 4'h%s); // %s/%s; %s/%s = %s R:%s \n\t" % \
92                          ('{0:0x}'.format(int(dividend[i])),'{0:0x}
                              '.format(int(divisor[i]))\
93                          ,'{0:0x}'.format(int(quotient[i-4])),'
                              {0:0x}'.format(int(remainder[i-4])),\
94                          dividend[i],divisor[i],dividend[i-4],
                              divisor[i-4],quotient[i-4],remainder[i
                              -4]))
95
96      else :
97          output.write("apply_test_vector(0, 0, 4'h%s, 4'h%s, 4'
              h0, 4'h0); // %s/%s; NA/Na = Na R:Na \n\t" % \
98                          ('{0:0x}'.format(int(dividend[i])),'{0:0x}
                              '.format(int(divisor[i])),\
99                          dividend[i],divisor[i]))
100
```

```
101 output.write("\nend\n\n")
102
103 with open(footfile) as lines:
104     for line in lines:
105         output.write(line)
106
107 output.close()
108
109 if error == 0:
110     print("\nExport to %s was successful!\n" % (destfile))
111 else:
112     print("\nExport to %s was UNSUCCESSFUL with %d errors!\n" %
        (destfile, error))
113
114 import time
115 raw_input("Press Enter to exit ;)\n")
```