Development of Hardware Division Unit using Mental Arithmetic Algorithm

by

Muhammad Amirrullah bin Alwi
17005398

Dissertation submitted in partial fulfilment of
the requirement for the
Bachelor of Engineering (Hons)
(Electrical and Electronics)

JULY 2022

Universiti Teknologi PETRONAS
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

CERTIFICATION OF APPROVAL

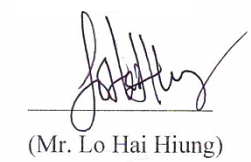
Development of Hardware Division Unit using Mental Arithmetic Algorithm

by

Muhammad Amirrullah bin Alwi
17005398

A project dissertation submitted to the
Electrical and Electronics Engineering Programme
Universiti Teknologi Petronas
In partial fulfilment of the requirement for the
BACHELOR OF ENGINEERING (Hons)
(ELECTRICAL and ELECTRONICS)

Approved by,



(Mr. Lo Hai Hiung)

Lo Hai Hiung
Lecturer
Electrical & Electronic Engineering
Universiti Teknologi PETRONAS
Bandar Seri Iskandar, 31750 Tronoh
Perak Darul Ridzuan, Malaysia

UNIVERSITI TEKNOLOGI PETRONAS

TRONOH, PERAK

July 2022

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.

_____

MUHAMMAD AMIRRULLAH BIN ALWI

ABSTRACT

This project is to propose a hardware division design that could possibly provide performance improvement in terms of clock cycle required to produce the output from any given input. This is by taking the advantage of the current technologies which is the same size of the silicon wafer now packs more transistor. This allow more and larger logic circuit to be implemented. Other than that, the proposed hardware design will be based on the mental arithmetic algorithms specifically using Vedic division algorithms. The algorithm used is an improved and optimised algorithm derive from Vedic division proposed by the previous work done in this fields. Hardware division design will then be implemented on hardware level into the FPGA development board. This will include the use of switch, push button, and 7-segment display to provide the input and display the result or output. Performance comparison is performed against restoring divider, which are based the utilisation and power consumptions. The overall performance is a bit behind compared to the restoring divider, however the simulation result and hardware implementation shows some potential from the mental arithmetic algorithm.

ACKNOWLEDGEMENT

First and foremost, praises and thanks to the God, the Almighty, for His shower of blessing and health throughout my final year project (FYP).

I also cannot express enough thanks and gratitude toward my FYP supervisor Mr. Lo Hai Hung for his invaluable patience and feedback. I could not have undertaken this journey without the help of my him, who generously provide knowledge and expertise.

I would like to express my utmost appreciation to both FYP 1 and FYP 2 coordinator, Dr. Azrina Abd Aziz, Dr. Norashikin Bt Yahya and also to all of the university personnel that are responsible in managing students' FYP. Their hard work allowed smooth experience for me before and during the FYP period.

Last but not least, a huge gratefulness from me for to my parents and family for their never ending mental and physical support on me in my entire live not to mention during my FYP.

To all person that I mention and not mention, directly or indirectly helping me. To their hard work and determination, I would like to thank them and my prayers for their good health and blessing.

Tables of Contents

LIST OF FIGURES

ABBREVIATION AND NOMENCLATURES

1. AI – Artificial Intelligence
2. ALU - Arithmetic Logic Unit
3. BCD - Binary Coded Decimal
4. CPU - Central Processing Unit
5. DSP - Digital Signal Processing
6. EDA - Electronic Design Automation
7. FF - Flip Flops
8. HDL - Hardware Design Language
9. IC - Integrated Circuit
10. IEEE - Institute of Electrical and Electronics Engineers
11. LUT - Lookup Table
12. SRT - Sweeney Robertson and Tocher
13. HDU - Hardware Division Unit
14. RTL - Register Transfer Level
15. FPGA - Field Programmable Gate Array
16. DFT - Discrete Fourier Transform
17. FFT - Fast Fourier Transform
18. MSB - Most Significant Bit
19. LSB - Least Significant Bit
20. ASM - Algorithm State Machine

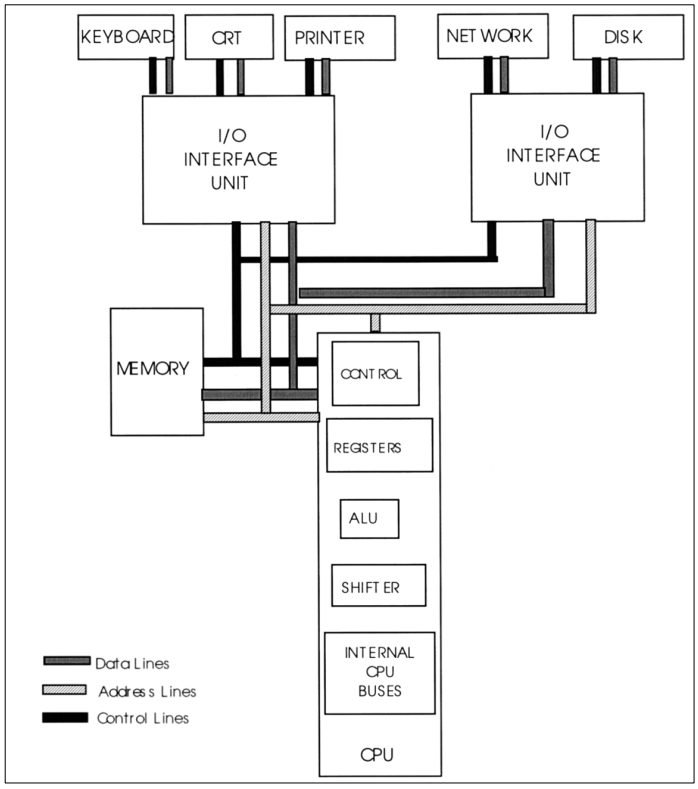
CHAPTER 1
INTRODUCTION

1. BACKGROUND STUDY

1.1. Computer Architecture

Computer has become one of the staples equipment to almost everyone either in form of a smart phones or a personal computer. It has become the definition of the modern world. Thanks to the computing power that we are currently have, it accelerates the next breakthrough in many fields. As mentioned by [1, p. 526] computer has become so advance that it is a part of our lives like our brains. The advancement in computer performance has been constantly improved by a staggering 25% each year [1, p. 526]. It can be said that, if the automobile industry had the same improvement as the computers, the car would only cost 10 cents and only took 5 minutes for a 4000 km trip [1, p. 527]. [2]

In general, there are 3 components in the computer system. These components are classified as processor, memory, and I/O [1, p. 531]. All of the components will communicate with each other to perform a particular task. The main brain for the computer system is the processor. Processor performed many tasks includes fetching, decoding, and executing [1, p. 532]. It consists of sequential and combinational logic circuit. The processor as a whole is composed of many components. The components include the control, arithmetic logic unit (ALU), registers, shifter, and internal busses. It responsible in coordinate the flow cycle of the whole system based on the programmed instruction. Modern processors contained more than 1 billion transistors [3], which is the building block for the logic circuit. It is a huge improvement comparing to the intel first-generation processor, which only have 3500 transistors.



[1]Figure 1. Interconnection between main computer components

Out of all processor components, it is considered that ALU is the center of the computer systems [1, p. 535]. ALU is a combinational circuit that will perform arithmetic operation such [4]as addition, subtraction, multiplication, division, etc. It also designed to perform logic operation such as “and” and “or” logic operation. The basic component circuit in ALU is hardware adder unit, which used to perform addition. It also can be manipulated in software level to perform other operation. However specific hardware unit can be designed to accelerate certain computation. As an example, hardware division unit can be design and implemented in hardware level in ALU. This in return will accelerate any process requiring division operation. In Intel 8008 processor only equipped with subtraction and addition hardware unit. Modern processor equipped with more complex ALU which also includes division & multiplication hardware unit.

Hardware division unit which used for processing division is important in digital signal processing (DSP), artificial intelligence (AI), rendering, graphic compression and etc

[5, p. 282]. Many algorithms have been developed and design to improve the hardware division unit design [2]. Division algorithm split into two categories which are slow and fast division. Slow division method like Restoring, non-Restoring and SRT produce one digit of the final quotient at each iteration. Meanwhile, fast division utilise multiplication with series expansion. The computation will converge quadratically towards the final quotient after each iteration [6]. The examples of fast division are Newton Raphson and Goldschmidt algorithms.

1.2. Mental Arithmetic

Mental arithmetic or calculation can be defined as performing calculation mentally without the need of any external tools or help such as calculator or paper [4, p. 2177]. Mental arithmetic usually involves the specific rules or strategies in breaking down a specific type of problems and solving it [4, p. 2177]. Mental arithmetic has been around in human history for so long. It assists human in solving everyday problems. People may decide to solve the problems mentally when it is more efficient comparing to other method of calculation. In general, mental arithmetic can be characterised by three main features [4, p. 2178]: (a) the strategy operates on numbers rather than on the whole digits, (b) the strategy developed with understanding of the underlying mathematical nature and relations, (c) the operation is creatively and flexibly based on the number's characteristic in the problems.

Vedic Mathematic is referring to the collection of sutras (techniques) that help to solve many mathematical problems mentally. Vedic math consists of 16 sutras with 13 sub-sutras. The introduced sutras cover many mathematical problems such as simple arithmetic for multiplication, division, algebra, geometry, calculus and etc [7]. the sutra helps solving complicated and huge numbers easily. Specifically for division problems, there are many sutras developed such as Nikhilam (all from 9 last from 10), Paravartya (transpose and apply), Anurupyena (proportion) and etc.

2. Problem Statement

1. With the abundance of transistors in IC design, there is a need for more complex arithmetic unit design to take advantage of the availability of transistor to help improve the performance of CPU design. This is an attempt at looking for possible performance improvement in terms of clock cycle in hardware division unit design.

2. There are many methods introduced to improve the hardware division unit design. However, the design of the hardware is still significantly bulky, require significantly more cycle and power compared to other basic operation. This project is to study the feasibility of mental arithmetic in designing hardware division unit.

3. Objective

1. To design and develop a hardware division unit using mental arithmetic algorithm.
2. To compare the developed hardware design with any available technology.
3. To develop a possible RTL model IP for hardware division unit.

4. Scope of Work

1. Literature review on previous attempt in improving hardware division unit using Vedic Sutras mental arithmetic algorithm.
2. Implementing algorithm on hardware level using FPGA development board.
3. Design simulation and perform comparative study.

CHAPTER 2
LITERATURE REVIEW

1. Division Operation Based on Vedic Mathematics

Author: Suyash Toro, Avinash Patil, Y.V Chavan, S.C Patil, D S Bormane, Susha Wadar

Summary: This paper works on new hardware division implementation. Comparing to the conventional method, this paper discusses the feasibility in implementing mental arithmetic algorithm on the hardware level. The paper discusses a few of the Vedic math algorithm in division. The author implements Nikhilam and Anurupyena sutras on FPGA.

Comparative study then carried out between the proposed design and the conventional algorithm design. The study shows the result of the new hardware design for both Nikhilam and Anurupyena algorithm. Comparing Anurupyena with the conventional algorithm. The proposed design utilises less lookup table (LUT) space by 30% compared to the conventional design.

Takeaway: The paper discusses the steps on how to perform Nikhilam and Paravartya algorithm. However, the paper does not discuss Anurupyena methods even though the algorithm is implemented in their study. The paper also does not mention whether the algorithm is implemented in binary or performed using binary coded decimal (BCD). In the algorithm discussion section, example was done in decimal number for all algorithms. Moreover, Nikhilam method required a few levels of recursive operation depend on the divisor and dividend as shown in the calculation example in appendix 1.1.

2. Survey on Implementation of IEEE754 Floating Point Number Division using Vedic Techniques

Author: Rajani M, S Sridevi.

Summary: Divider design for arithmetic logic unit is more critical as it consume more area and power compared to other arithmetic operations. Efficient and compact divider design is always a challenging task. Vedic mathematic provides algorithms to simplify the division operations.

There are 5 sutras can be applied for decimal division. However, only 3 sutras are suitable for binary division. These sutras also can be applied to ieee754 floating point standards. The 3 sutras that are suitable are Nikhilam, Dwajanka, and Paravartya. Operation delay is simulated to compare conventional method with Nikhilam and Paravartya method. Nikhilam algorithm have 35.787ns delay while Paravartya algorithm is 14.453ns delay. Non-Restoring algorithm produce 17.911ns delay.

From the simulated result, Vedic divider is faster by 19% compared to the conventional method. Static timing analysis is performed using 32nm standard libraries comparing Paravartya algorithm and conventional algorithm. Vedic divider consumed ~109W less power and ~7 times faster than the conventional divider. Vedic divider also occupied ~13 times less space than the conventional method. In term of resource utilization, Vedic divider improve by 12% and have more package density compared to the conventional method.

Takeaway: Even though, all sutras initially applied and works with base 10(decimals) numbers. 3 of the sutras can be easily applied to perform division in base 2(binary). Thus, the hardware does not require and BCD to process any mathematical operation.

1. Nikhilam:

1. Numbers of Dividend bits – numbers of divisor bits, will produce number of quotient bits.

2. The remaining bits is for the remainder.

3. Get the deficit by subtracting divisor from 1bits higher than the divisor value i.e., divisor = 111, to get the deficit: 1000 – 101 = 011.

4. Dividend and the deficit will be used to calculate the quotient and remainder.

5. Nikhilam usually required a few iterations before it came out with the final answer.

Nikhilam step required a separate process for quotient and remainder. As an example, if the remainder region is the last 4 bit, the summation of value in the last 4-bit region cannot be carry into the quotient region even when it exceeds 4 bits like the example in appendix 1.2. Then the remainder will repeat the same operation as before. If the remainder have the same number of bits as the divisor, it needs to perform subtraction operation. Thus, it seems that binary with Nikhilam algorithm required memory space the quotient and remainder.

2. Paravartya

1. Subtract transposed bit of divisor from the dividend.
2. Performed column and row operation for the final product.

Example of calculation using this algorithm is shown in appendix 1.3. The main challenge for this method is dealing with signed value and performing column operation after each iteration right before proceeding to the next iteration.

3. Low Power Divider Using Vedic Mathematics

Author: Dalal Rutwik Kishor, V.S Kanchana Bhaaskaran

Summary: Divider hardware are essential in advance and high speed digital signal processing technology. It is widely used in radar technology, communication, and industrial control systems. This paper proposed a fast, low power and cost-effective architecture of a divider using the ancient Indian Vedic division algorithm. Division algorithm is developed to ease the computational difficulties for division problems.

The Restoring, non-Restoring and SRT (Sweeney, Robertson and Tocher) division algorithms are the widely used algorithm. Division algorithms can be divided into slow and fast division. The slow division produces a single bit of final quotient per iteration like algorithm that apply digit recurrence. The digit recurrence method is based on successive subtraction technique to obtain fixed number of quotient bits per iteration. This method used smaller area however increase the latency of the process. Fast division type are Goldschmidt algorithm and Newton-Rapson method. This operation is achieved by multiplication operation, this technique offers low latency and high precision however consume large silicon area.

This paper introduces Vedic mathematics to carry out mathematical computations. When comparing conventional and Vedic divider, power dissipation on Vedic divider is 52.93% less than the conventional divider. Logic cells used to construct Vedic divisor is 24 compared to 53 for conventional divisor. Vedic divider proposed also does not used any comparator and multiplexer which reduce the circuit complexity. The delay in the Vedic divider is 34% lower than the conventional divider.

Takeaway: Using Paravartya method required dealing with signed value. This paper introduced crumb method derive from the Paravartya algorithm to simplify the algorithm when dealing with sign in binary. It is by introducing another bit to each bit to represent the bit sign. Thus, this architecture does not require to deal with any sign

value which ease the total operation. However, it requires (1) encoding operation each divisor and dividend bit into two bits. Then, (2) division operation which by successive partial multiplication and addiction. lastly, (3) it required decoding operation, so the output value is usable and in binary representation

4. Design and Synthesis of Goldschmidt Algorithm based Floating Point Divider on FPGA

Author: Naginder Singh, Trailokya Nath Sasamal

Summary: This paper presents design for a single precision floating point division based on Goldschmidt algorithm. The Goldschmidt algorithm is implemented using 32-bit floating-point multiplier and 32-bit floating-point subtraction modules. The algorithm uses an iterative process in which the denominator gets scaled to one to get the final quotient. The floating-point multiplier is design using Vedic multiplication instead of conventional multiplication algorithm. The result show improvement in computational speed. The paper concludes that using the Vedic multiplication in the design improve the performance.

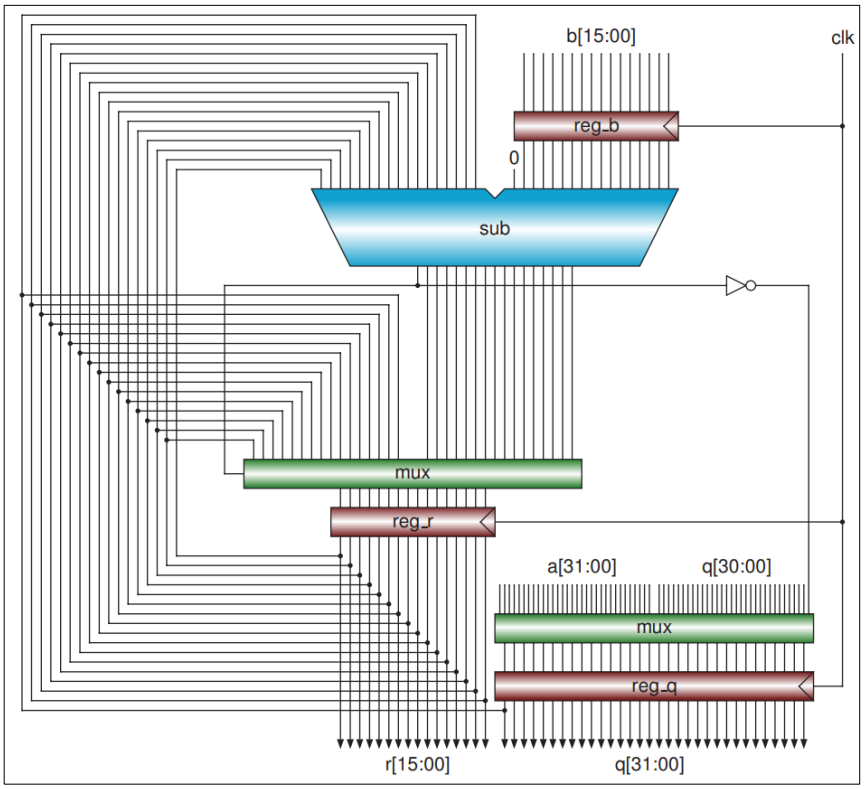
Takeaway: The simulation result shows the proposed hardware design latency is 75ns compared to two reference design which took 175.49ns and 130.8ns in performing the operation. The power consumption come with this hardware design is 0.037w compared to 0.05w for the reference design. It shows that, with just optimizing the floating-point multiplier with Vedic math, the power consumption and latency time are reduce by 26% and 42.6% respectively. Thus, a promising improvement could be made in the hardware division unit design when implementing Vedic sutras that specifically design to tackle division problems.

5. Computer Principles and Design in Verilog HDL

Author: Yamin Li

Summary: The book describes computer principles, computer designs, and how to use Verilog HDL (hardware description language) to implement the designs. Besides the source codes of Verilog HDL, the simulation waveforms are also included in the book for easy understanding of the designs.

Takeaway: Computer processor work in binary number. The binary number representation will be either in unsigned number or 2's complement which representing signed integer. The conventional circuit design for addition, subtraction, multiplication and also division are discussed and can be taken for comparative study. It also an important learning material to refer when designing hardware division unit. The book provides many binary divisions algorithms design and code such as Restoring, non-Restoring, and Goldschmidt algorithm. Code and design are shown in appendix 1.4.



[8]Figure 2. Schematic diagram of a Restoring divider

6. Integer Multiplication in time O (n log n)

Author: David Harvery, Joris van der Hoeven

Summary: The paper presents an algorithm that compute the product of two long n-bit integer in *O* (*n log n*) time. The analysis take place in the multitape Turing machine, with integer encoded in the usual binary representation. The proposed algorithm is using "Gaussian resampling" method which reduce the integer multiplication problems into a set of multidimensional discrete Fourier transform (DFT). All of the dimensions are in the power of two. The proposed algorithms are inspired by the observation that a particular multivariate polynomial right admit certain efficient multiplication algorithms.

Takeaway: The problems discuss in this paper is on how to optimise the multiplication algorithm which help reducing the time required to calculate the multiplication. For decade, mathematicians are trying to improve the time required for multiplication problem. The idea first discovers by Karatsuba which works an algorithm that required $\sim n^{1.58}$ time to solve the multiplication. The work on improving algorithm that use less time again done in 1963, 1966 by various mathematician.

After the introduction of the Fast Fourier Transform (FFT) in 1968, new work has been proposed by Schönhage and Strassen in 1971 which multiplication algorithm in time *O* (*n log n log log n*) using FFT. Schönhage and Strassen also proposed a conjecture that the best possible time can be achieved for integer multiplication is in time *O* (n *log n*). Fast forward to modern day, if the conjecture is correct, the proposed technique in this paper is the best algorithm that can be used on integer multiplication to improve the calculation time.

7. 64 Bit Divider using Vedic Mathematics

Author: Aditi Tadas and Dinesh Rotake

Summary: Vedic sutras i.e Nikhilam, Dhwajank and Paravartya is developed to work with base 10. Thus, it suits base 10 (decimal) calculation more than in another base. Current world computer and processor are utilizing binary numbers to process and communicate. Thus, it is a necessary to develop a mental arithmetic algorithm that is suitable with base 2 (binary). A new method was proposed in this paper which optimized binary division. The proposed method was derived from Vedic Sutra(s). The new method and 3 other Sutras are then analysed in terms of their memory consumption, power, time, and design complexity. It shows that the new proposed method derived from sutras required less operation and more efficient in terms of hardware usage.

Takeaway: The most interesting things from this paper are the proposed algorithms. The algorithm proposed is optimised to work in binary operation and simplified to reduce the design complexity as compared to other Vedic division algorithms. One of the most interesting features offered by this algorithm is the algorithm only require row processing. Thus, at each iteration there is no need to re-evaluate and perform column wise subtraction operation like Paravartya method does. This method will be discussed more in the next chapter

CHAPTER 3
METHODOLOGY

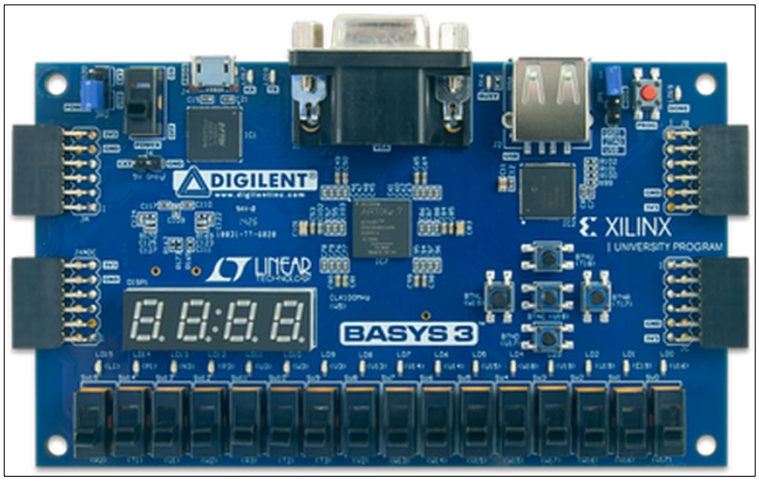
1. TOOL

1.1. Electronic Design Automation (EDA)

The EDA tool that are going to be used for designing, simulating, and synthesising the hardware division unit is Vivado 2022 design suite by Xilinx. The hardware description language used is Verilog.

1.2. FPGA Development Board

The FPGA development board chosen is Basys 3 by Digilent. The board is based on Xilinx Artic 7 FPGA chip. It contains 33280 logic cells in 5200 slices. The board equipped with 16 toggle switch and LED, and four 7-segment display.



[9]Figure 3. Basys 3 FPGA development board

2. ALGORITHM

The algorithm proposed in this project is based on work done by Aditi Tadas and Prof. Dinesh Rotake in their paper titled “64 Bit Divider using Vedic Mathematics”. One of the features of the proposed algorithm is the algorithms does not require any comparator or complementing circuit. Other than that, the algorithms only will be performing row operation. Thus, no column operation required such as performing subtraction operation in column wise before the operation can proceed to the next iteration. The algorithm is developed based on Vedic math. It is also has been improvised and optimised to work in binary and the operation does not require any multiple level of recursive iteration like Nikhilam method. This is because the final remaining from this proposed algorithm is the remainder, however in Nikhilam method, it is possible that the produced remaining value at the end of the process is not the final remainder. Thus, in order to produce the final remainder, the remaining values are required to go through the algorithm process again until it produces the final value.

Algorithm proposed involve repeatedly subtraction using shifted divisor. Thus, it reduces the number of iterations to the final results. The algorithm as suggested by [8] will require more power for memory read and write cycles for the shift counter. The algorithm flow chart is shown in figure 4. The calculation example is shown in figure 5 and appendix 2.1.

Steps:

1. Block shifting divisor to 1 bit position less than the most significant bit (MSB).
2. Store shift counter in the memory in form of power of 2.
3. Subtract the shifted divisor from dividend.
4. Repeat step 1 to 3 for each new value from the division until dividend is less than divisor.
5. The remaining value will be the remainder.
6. Total up all the shift counter from the memory to get the final Quotient.

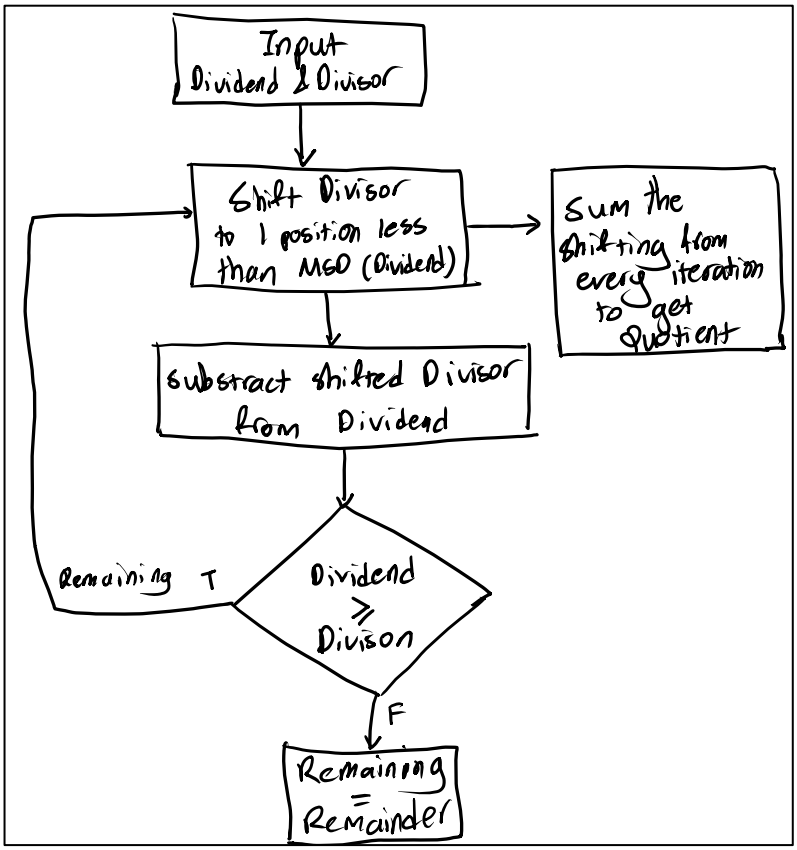


Figure 4. Proposed algorithm flow chart

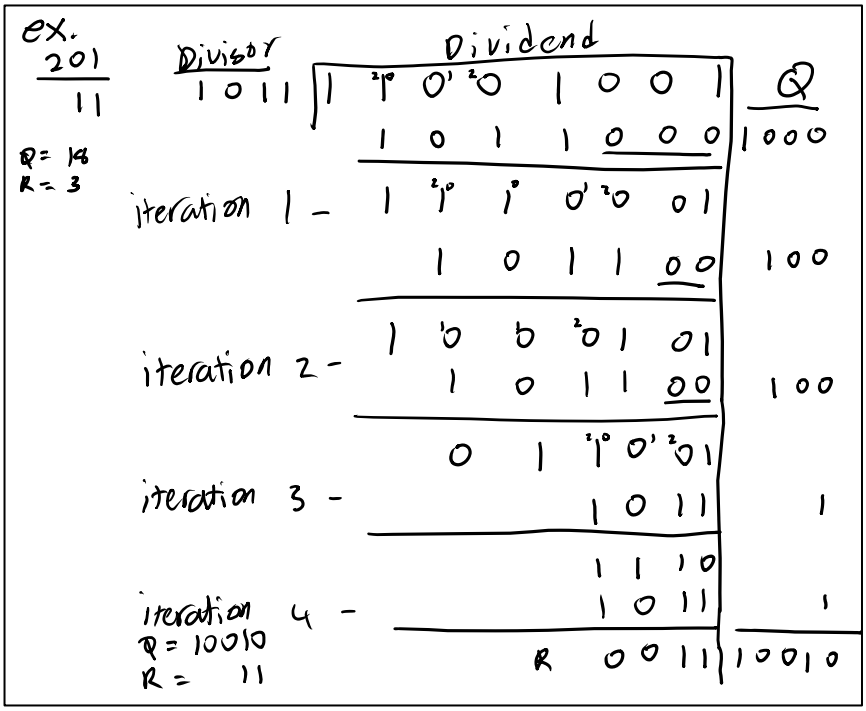


Figure 5. Solution example using the proposed algorithm

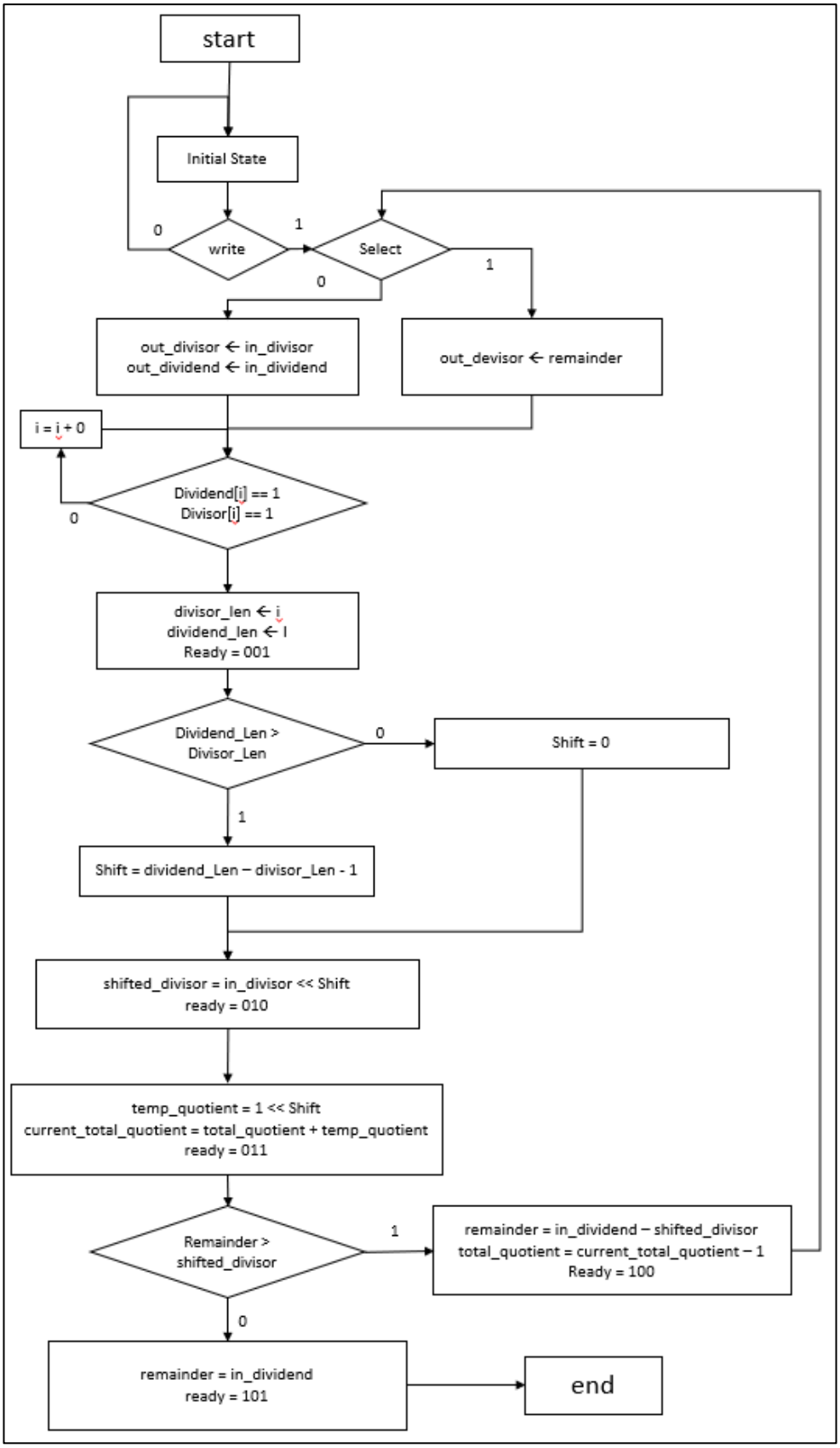


Figure 6 Algorithm State Machine Chart

Figure 6 shows the algorithm state machine (ASM) chart. The algorithms are separated into 4 stages and one controller module. The first stage is to store the input into the register and calculate the input dividend and divisor bit length. The next stage is performing bit shifting to the divisor until it is 1 bit less than the dividend except when then dividend length is the same as the divisor. To determine how many bits to shift, the system will subtract the dividend length with the divisor length plus one (shift = dividend length – divisor length -1). The third stage is to compute the current quotient of the current iteration and compute the total quotient. The last stage will subtract the shifted divisor from the dividend and calculate the current total quotient of the operations. Then, the last stage will return the new dividend and total quotient value. The return dividend value will then be updated into the dividend register.

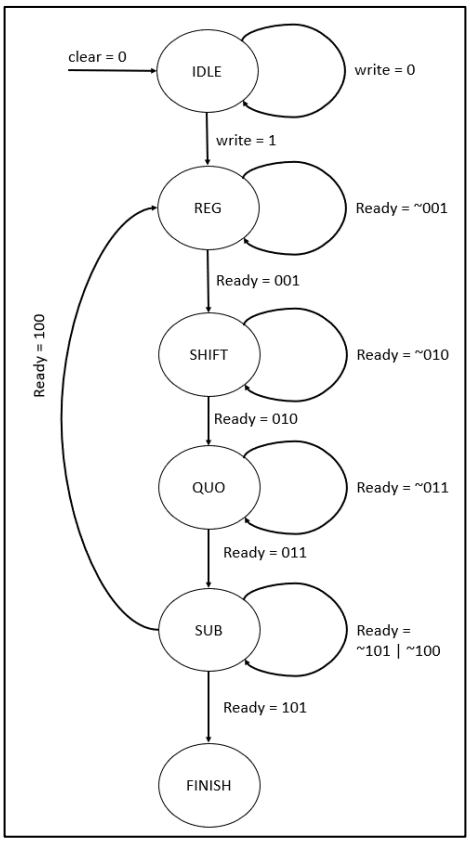


Figure 7 Finite State Machine

Figure above shows the finite state machine proposed for the hardware design. It is to control the sequential logic flow of the hardware module designed. Here proposed 6 states. 4 of the states is the state for each module operations which includes store value in register (REG state), block shifting the divisor (SHIFT state), calculate quotient

(QUO state) and subtract shifted divisor from the dividend (SUB) state. Other 2 state is for idle and finish calculation state.

The system calculation is performed iteratively until it achieved the final output. Each stage function takes one clock cycle thus for 1 iteration it took 4 clock cycle theoretically. If the input is 8-bit dividend and 4-bit divisor, the worst-case scenario the computation took 60 cycles while the best-case scenario it took only 4 cycles theoretically excluding the overhead cycle. If the time period is 10ns, theoretically it will take between 40 to 600ns minimum to complete the operation.

For performance comparison, the mental arithmetic divider algorithm will be compared against the Restoring divider algorithm. This performance comparison parameter includes resource utilization, power consumptions and computation clock cycles.

The design later will be synthesis into the FPGA board. For the input, 16-toggle button will be used to insert the dividend and the divisor in binary value. To display the output, 4 7-segment display will be used to display the quotient and the remainder in decimal number representation. The study will only be performing division on unsigned integer which means the divisor and the dividend will be positive integer. The main initial target is to develop 8-bit dividend and 4-bit divisor hardware division unit.

CHAPTER 4
RESULT AND DISCUSSION

1. HARDWARE SCHEMATIC

1.1 Mental Arithmetic Divider

Figure below shows the generated schematic from the post-implementation by the Xilinx hardware development tool, Vivado. Here it shows in detail the connection and flow of each, and every component required. On the far left is the shifter module, next to it, is the quotient module, then the register, controller and the last one is the subtractor.

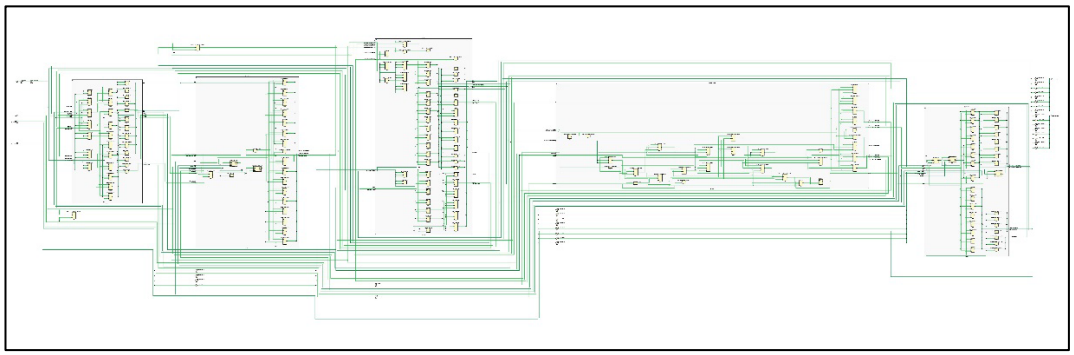


Figure 8 Generated post-implementation schematic

Below is the similar hardware schematic as figure above however simplified for easy user views. More details hardware schematic for each module can be view in appendix 3.

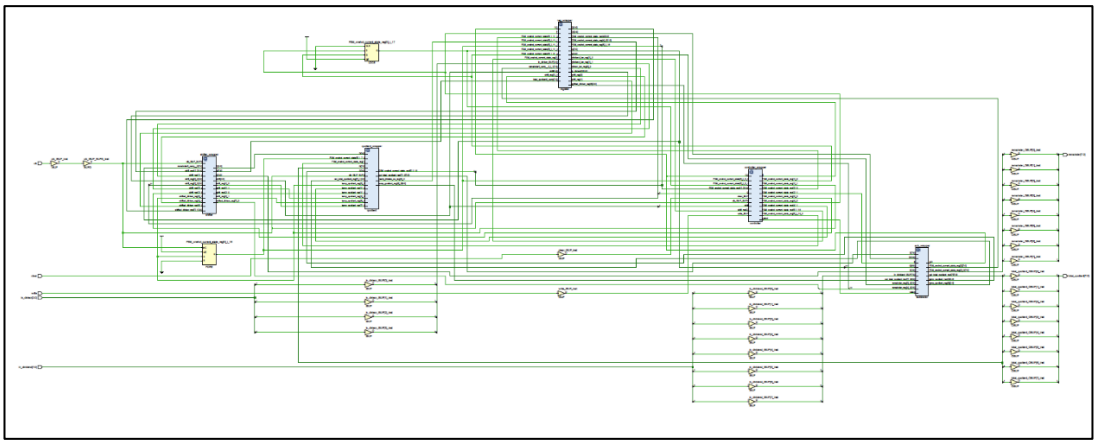


Figure 9 Simplified hardware schematic

The input is 8 toggle switches for dividend, 4 toggle switches for the divisor, 1 toggle switch for the clear input and lastly 1 push button for write input. The output will display the remainder and the total quotient from the calculation. 16 Led is used to display the output. 8 for the quotient and 8 for remainder. Below is the input and output map of each input and output on the FPGA board.

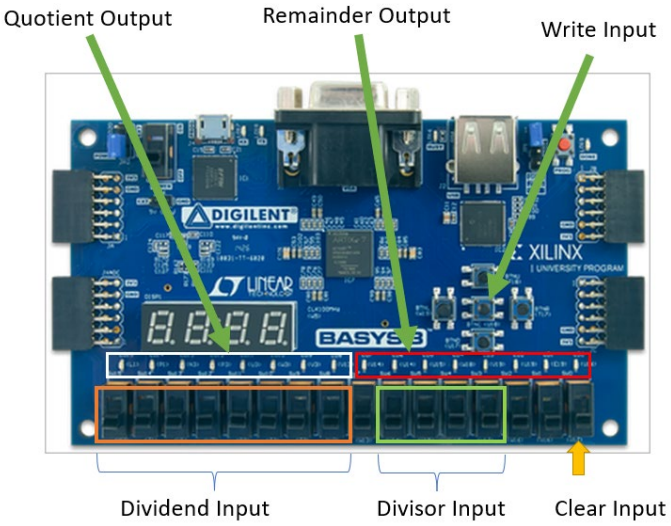


Figure 10 Hardware IO Constraints

1.2 Restoring Divider

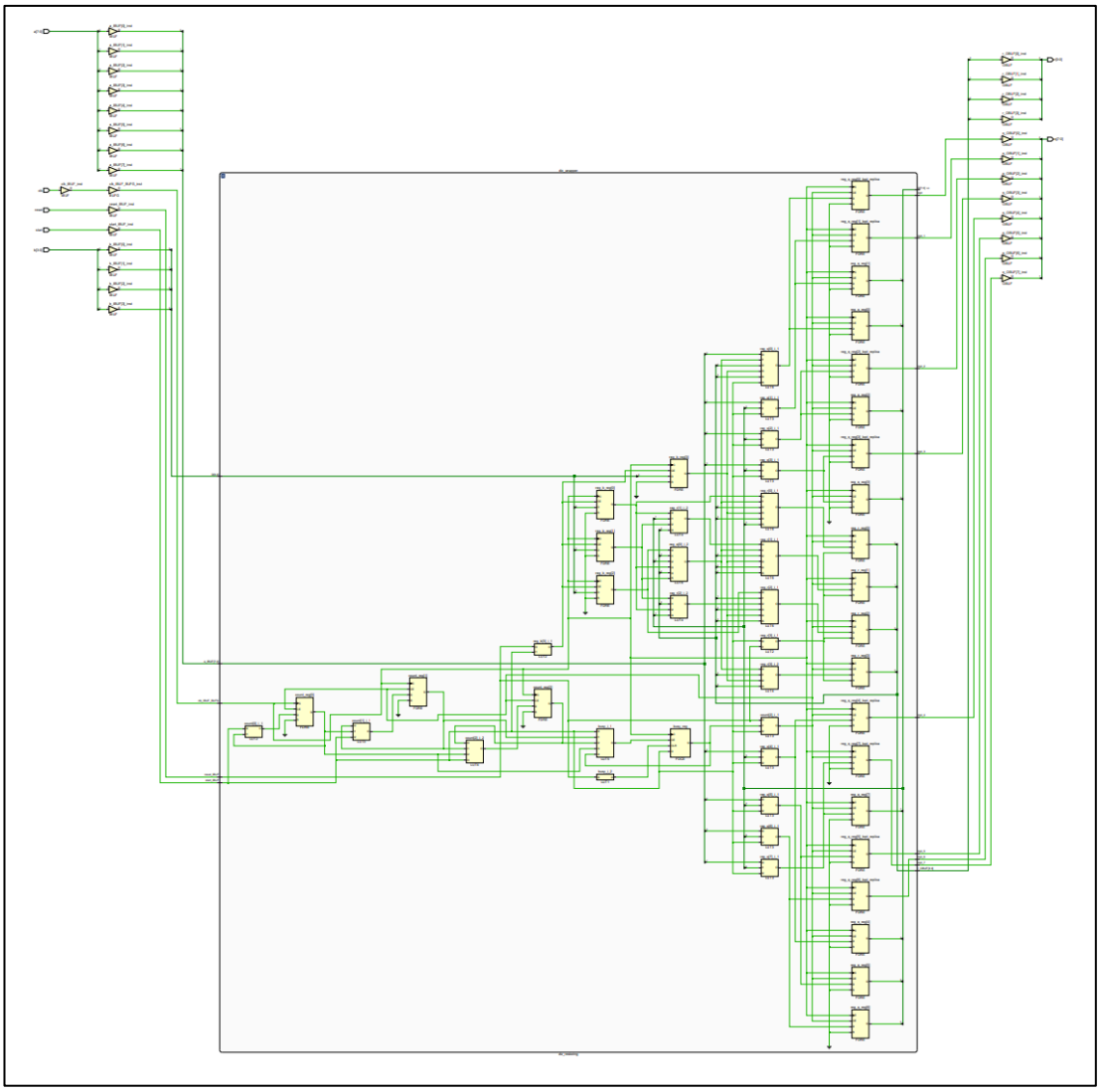


Figure 11 Restoring Divider Schematic

Above shows the generated schematic from the RTL code for the Restoring divider. The design is retrieved from [8] with some modification from 32-Bbit to 8-bit hardware division unit. The inputs are 8-bit dividend, 4-bit divisor, reset and start. The output is 8-bit quotient and 4-bit remainder. This Restoring divider will be use as a reference in analysing mental arithmetic divider performance.

2. SIMULATION

2.1. Mental Arithmetic Divider

Simulation testbench was set up to perform the behavioural simulation to analyse the workflow and evaluate the generated output based on the given input.

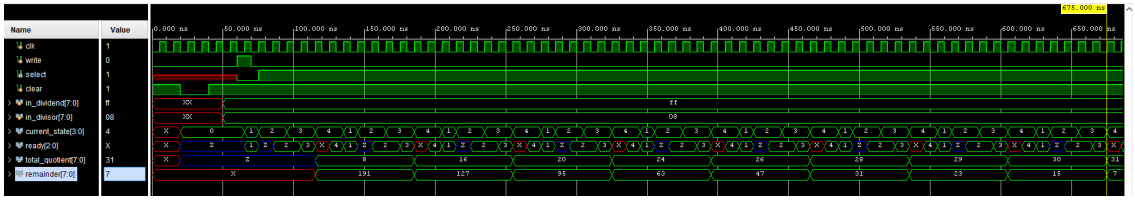


Figure 12 Behavioural Simulation Result

Figure above shows the simulation result of the generated hardware module. The clear input is active low, used to reset all the stored value. Hardware will start computing when write signal is high. The system will sequentially move from one state to another state which at the same time enable the hardware module accordingly. Given the input is 255 dividend and 8 divisors, the hardware starts the computation at time 70 nano second and complete the computation at time 680 ns which took about 61 clock cycles to complete. The output from the simulation is 31 quotient and 7 remainder which is as intended. Moreover, upon close inspection on the behavioural simulation, it is shown that the hardware module is working properly and as intended.

2.2. Restoring Divider

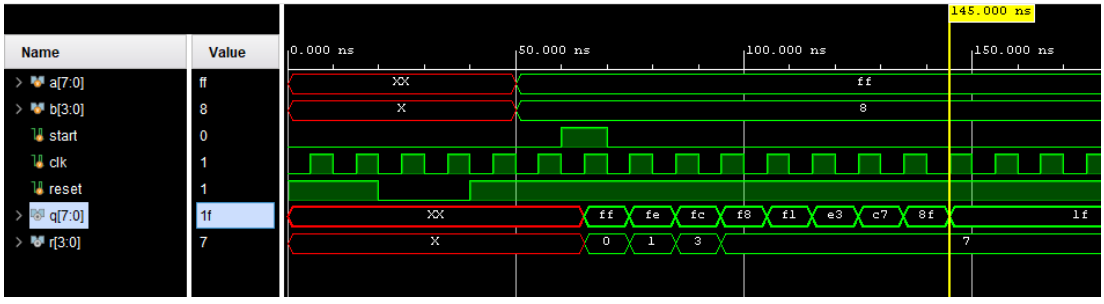


Figure 13 Restoring division simulation

Above shows the simulation result of the Restoring division operation. The input is 255 dividend and 8 divisors. the computation is complete after 8 clock cycles, which took about 80ns.

3. FPGA IMPLEMENTATION

The RTL code of mental arithmetic divider then is synthesized into the FPGA board using the provided hardware development tools (Vivado). The FPGA memory then

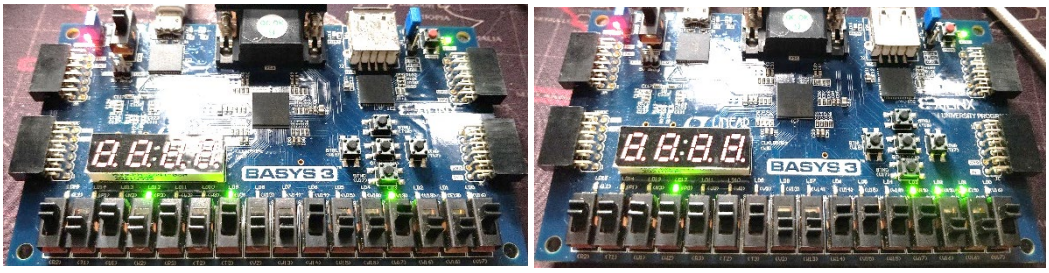


Figure 14 FPGA performing division

been programmed with the generated binary file. Figure above shows a few of the division calculation that was performed using the FPGA board. The first picture in the figure above shows 170 divided by 9 which produce 18 as the quotient with 8 remainder. While the second picture shows 254 divided by 15 which produce 16 with remainder of 14. The output from the FPGA can be analyse from the LEDs that light up. The first 8 LEDs starting from the left is showing the quotient output with remaining 8 LEDs is for Remainder. The most significant bit is on the left. For both calculations the output produced are correct and as intended. More of sample calculation can be seen in appendix 4.

4. IMPLEMENTATION REPORT AND IP

4.1. Mental Arithmetic Divider

From the generated design based on the RTL code the resource utilisation can be evaluated. Below is the hardware utilisation report, which shows number of look up tables (LUT), flip flops (FF) and input/output (IO). The required LUTs are 80 with 76 FF to construct the designed hardware

Summary

Resource	Utilization	Available	Utilization %
LUT	80	20800	0.38
FF	76	41600	0.18
IO	31	106	29.25

Figure 15 Utilisation Report

Timing analysis below shows that the generated hardware has worst negative slack of 3.02ns, worst hold slack of 0.185ns and worst pulse width slack of 4.5ns. negative slack shows by how much the timing requirement is miss. However, the timing constrain was not set.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3.020 ns	Worst Hold Slack (WHS):	0.185 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	73	Total Number of Endpoints:	73	Total Number of Endpoints:	39

All user specified timing constraints are met.

Figure 16 Timing Report

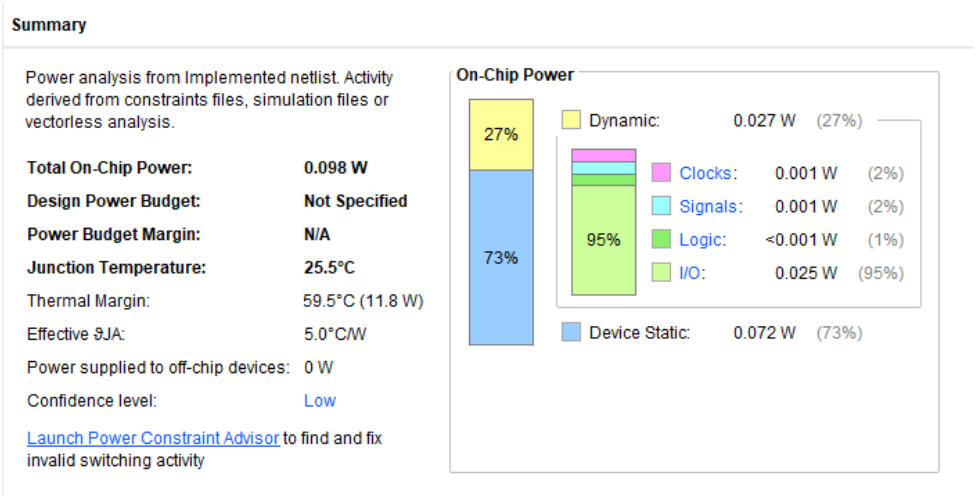


Figure 17 Power Summary Report

Based on the default device configuration the power analysis is shown in the figure above. The power consumption estimated on the hardware design is 27mW. It also can be seen that the most power consumed is from the I/O which is all the switches, button and LEDs. Thus, the hardware power consumption excluding the physical I/O is less than 3mW. Device static power is ignored in this case.

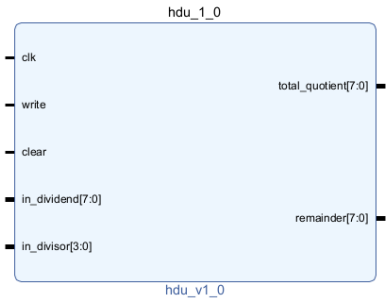


Figure 18 Generated IP

Figure above shows the generated IP block based on the RTL code designed earlier. The IP block is generated using Vivado. It has 3 logic input, 2 logic vector input, and 2 logic vector output.

4.2. Restoring Divider

Summary

Resource	Utilization	Available	Utilization %
LUT	15	20800	0.07
FF	28	41600	0.07
IO	27	106	25.47

Figure 19 Hardware utilisation

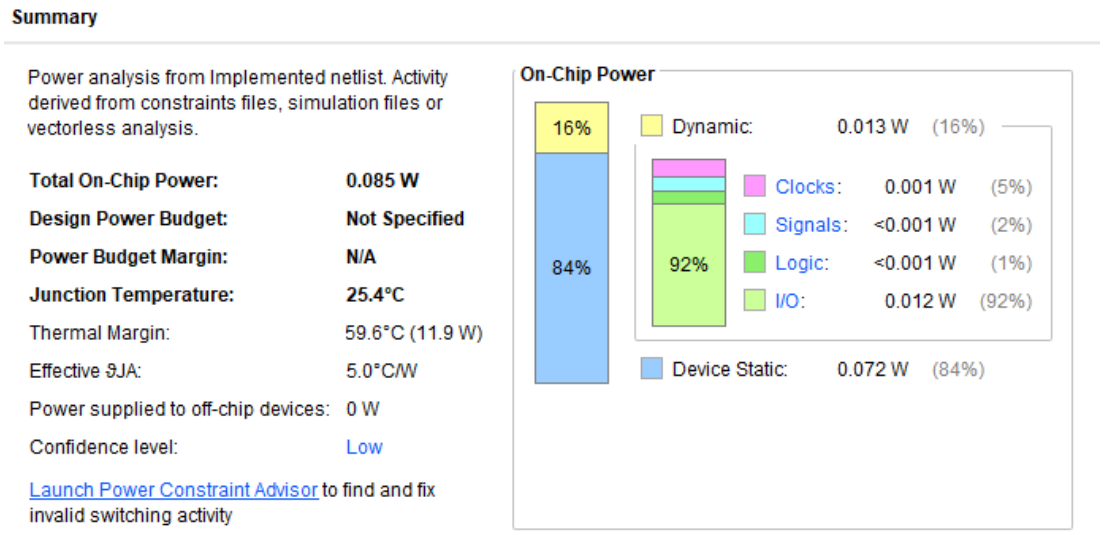


Figure 20 Power utilisation report

From figure 19 shows the utilisation of the LUT and FF which are only 15 and 28 respectively. The power utilisation of the hardware design is about 3 mW when excluding the IO power consumptions.

CHAPTER 5
CONCLUSION AND RECOMMENDATION

1. Conclusion

The mental arithmetic divider is successful designed and synthesize into the FPGA board. However, based on the result gather from the simulation and reports the Restoring divider has much lower resource utilisation which about 5 times smaller on LUTs and 2.5 times fewer flip flops required. The power consumption is almost similar however Restoring divider consume slightly less power than the mental arithmetic divider. When comparing the clock cycle required to perform the calculation which here refer to the clock cycle from write input until the outputs are produced, the restoring divider require less clock cycles which is only 8 cycles for every computation while Mental arithmetic divider required between 4 to 60 cycles.

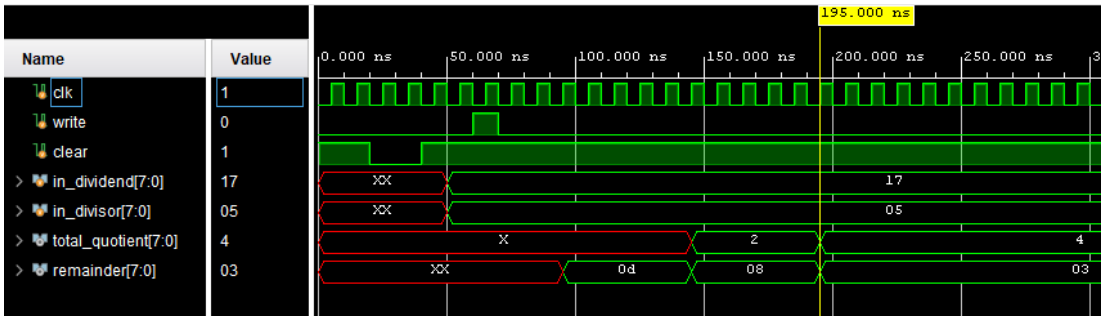


Figure 21 Mental Arithmetic Divider

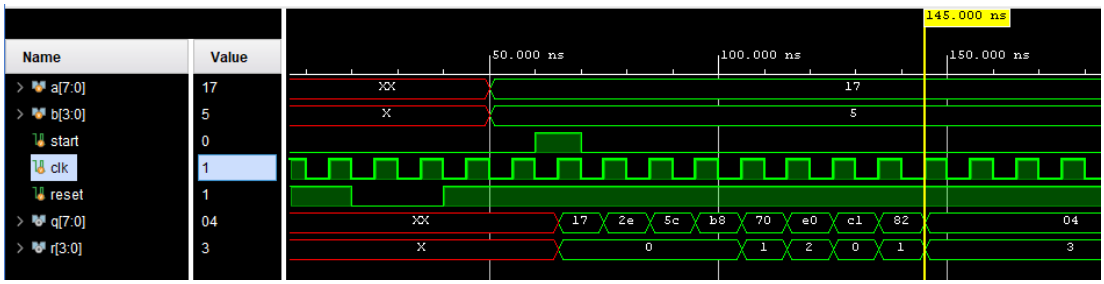


Figure 22 Restoring Divider

The clock cycle cost on the restoring algorithm is fix, whether the computation is simple or complex as shown in figure 22 and figure 13. Moreover, when the Restoring algorithm are modified to compute more larger number such as for 16, 32 or 64 bits, the clock cycle required will also increase linearly. On the other side, the mental arithmetic algorithm uses less clock cycle when the computation is simpler as shown in figure 21 compared to the figure 12. Because of this, the clock cycle required is unpredictable when using the mental arithmetic algorithm. This is not great for and not desired in computation as there is more uncertainty in the computation. However, this project shows the feasibility of mental arithmetic algorithm in solving division problems. The source code are provided in appendix 5.

2. Recommendation

There are many improvements that the designed mental arithmetic required this includes both further upgrades or updates and optimisation in the RTL design.

1. Optimise the RTL Design.

   The current design implemented is in 4 stages which it took 4 clock cycles to complete one iteration. Take as example in stage number 2 and 3, it could be executed in one single cycle. However, during this design exploration when synthesize into the FPGA, certain computation was unable to obtain the correct quotient this occur on any set of numbers that the last iteration quotient produced is not (binary) 1 which either (binary) 10,100, or 1000. Further understanding and observation are required to improve further the RTL design and reduce the stage number. This will allow this mental arithmetic algorithm to be more competitive against the restoring divider in terms of clock cycles and resource utilisation.

   Moreover, algorithm improvement also required to ensure the algorithm is more predictable and certain which in this case, the required clock cycles to complete the computation.

2. Design for scalability.

   The current design is using static comparators for all cases to determine the dividend and divisor length. This is because to allowed easy synthesize on the FPGAs. However, dividend and divisor bit length also can be determined by looping and checking the which is the largest input bit. The design was simulated and works in the simulation. This is one alternative method that was been tested which could allowed easy scalability to implement the algorithm for 16, 32 or 64 bits. Unlike the current design which required all cases to be programmed manually in the design.

3. Modified algorithm for floating point calculation.

   Current RTL design only able to perform calculation on integer numbers only. Further design exploration can be made to verify the potential of mental arithmetic divider in solving floating points problems.

BIBLIOGRAPHY

[1] J. Grimes, "Computer Architecture," *Encylopedia of Physical Science and Technology (Third Edition),* pp. 525-542, 2002.

[2] D. R. Kishor and V. S. K. Bhaaskaran, "Low power divider using vedic mathematics," *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI),* pp. 575-570, 2014.

[3] P. Alcorn, "AMD Ryzen 9 3900X and Ryzen 7 3700X Review: Zen 2 and 7nm Unleashed," November 2019. [Online]. Available: https://www.tomshardware.com/reviews/ryzen-9-3900x-7-3700x-review,6214.html.

[4] L. Verschaffel, J. Torbeyns and B. De Smedt, "Mental Arithmetic," *Encyclopedia of the Sciences of Learning,* 2012.

[5] S. S. Kuldeep and K. S. Ashwani, "A Low Power 16 Bit Vedic Divider for High Speed VLSI Applications," *An International Journal of Engineering Sciences,* vol. 17, pp. 282-288, January 2016.

[6] P. Saha, D. Kumar, P. Bhattacharyya and A. Dandapat, "Vedic division methodology for high-speed very large scale integration applications," *The Journal of Engineering,* 2014.

[7] R. Bhangale, "mathlearners," February 2013. [Online]. Available: mathlearners.com.

[8] A. Tadas and D. Rotake, "64 Bit Divider using Vedic Math," in *2015 International Conference on Smart Technologies and Management for Computing, Communicaton, Controls, Energy and Materials (ICSTM)*, Chennai, 2015.

[9] Y. Li, "Binary Division Algorithms," in *Computer Principles and Design in Verilog HDL*, John Wiley & Sons, 2015, pp. 84-95.

[10] N. Signh and T. N. Sasamal, "Design and Synthesis of Goldschmidt Algorithm based Floating Point divider on FPGA," in *2016 International Conference on Communication and Signal Processing (ICCSP)*, Melmaruvathur, 2016.

[11] S. Toro, A. Patil, Y. V. Chavan, S. C. Patil, D. S. Bormane and S. Wadar, "Division operation based on Vedic mathematics," *2016 IEEE International Conference on*

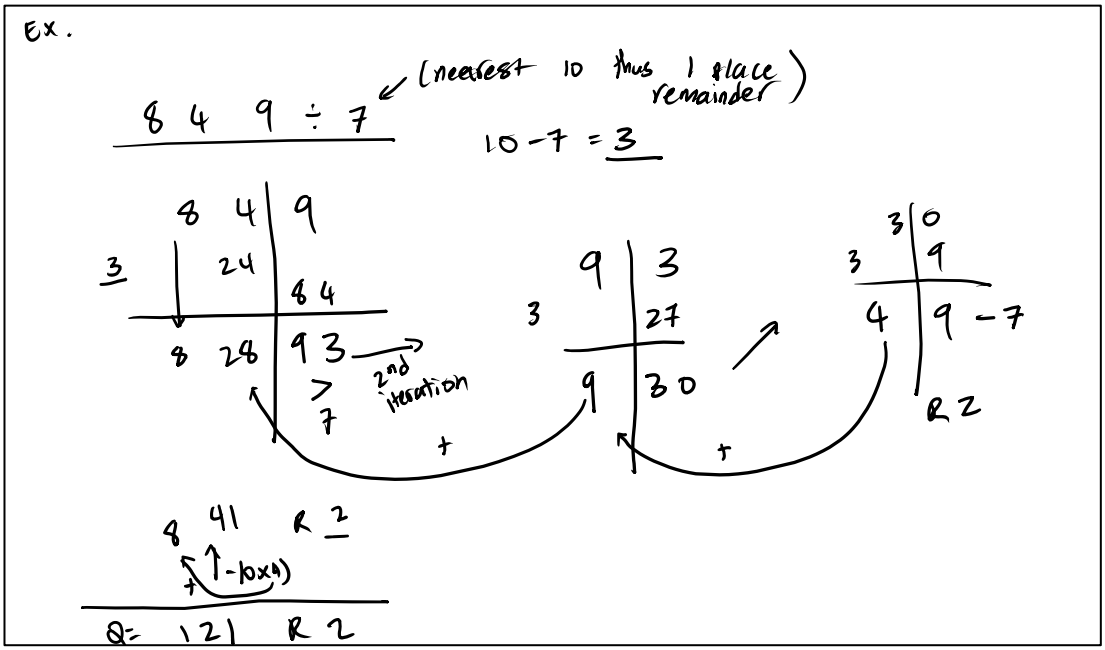
Advances in Electronics, Communication and Computer Technology (ICAECCT), pp. 450-454, 2016.

[12] R. M and S. Sridevi, "Survey on Implementation of IEEE754 Floating Point Number Division using Vedic Techniques," *International Journal of Engineering Development and Research (IJEDR),* vol. 3, no. 3, 2015.

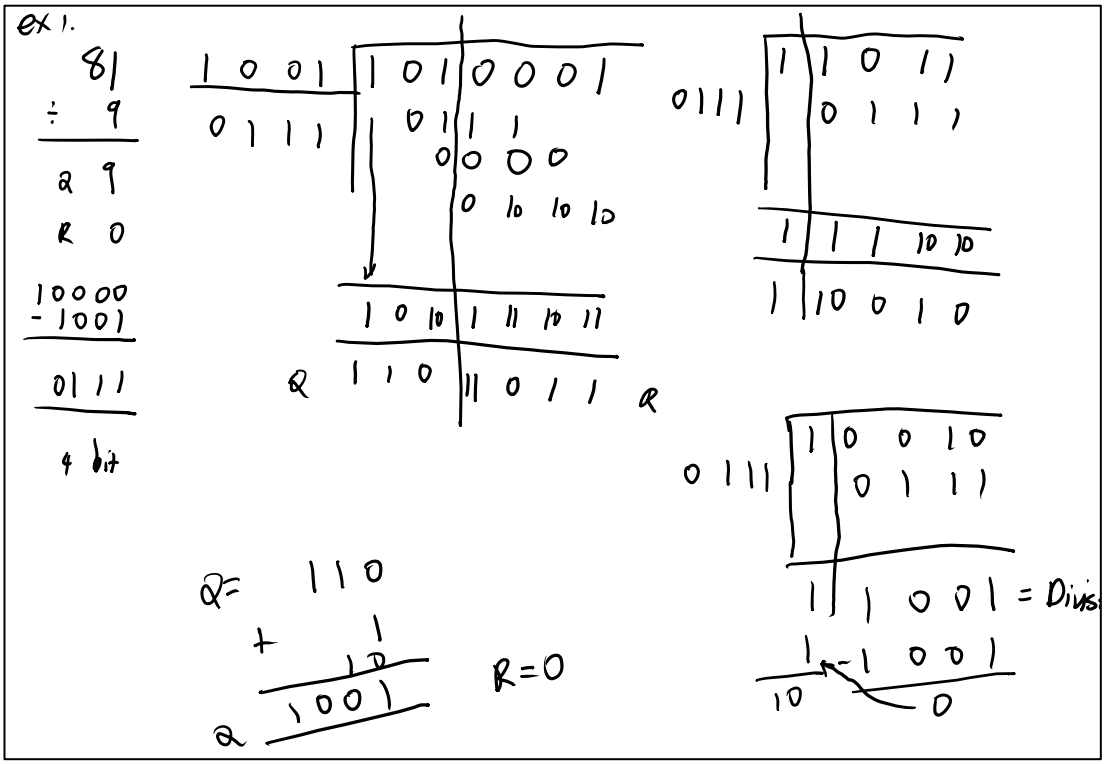
[13] h. David and H. v. d. John, "Integer multiplication in time O (n log n)," *Annals of Mathematics,* vol. 193, no. 2, pp. 563-617, 2021.

APPENDICES

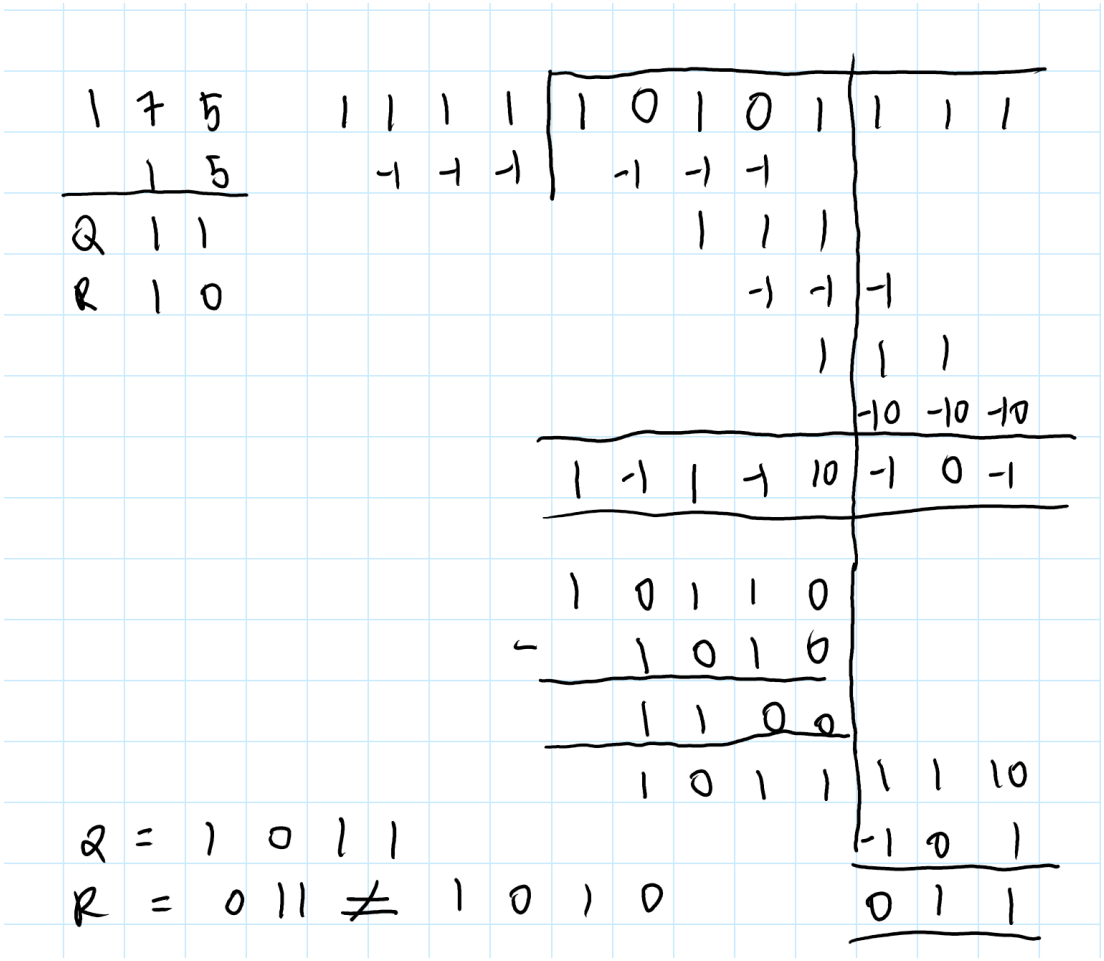
Appendix 1.1



Appendix 1.2



Appendix 1.3

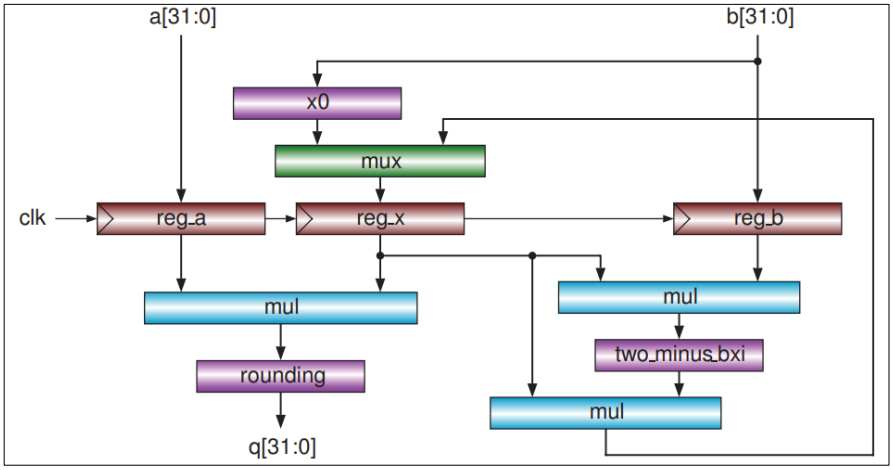


Appendix 1.4

Verilog code for Restoring divider

```
module div_restoring (a,b,start,clk,clrn,q,r,busy,ready,count);
    input  [31:0] a;                                          // dividend
    input  [15:0] b;                                          // divisor
    input         start;                                      // start
    input         clk, clrn;                                  // clk,reset
    output [31:0] q;                                          // quotient
    output [15:0] r;                                          // remainder
    output reg    busy;                                       // busy
    output reg    ready;                                      // ready
    output  [4:0] count;                                      // counter
    reg    [31:0] reg_q;
    reg    [15:0] reg_r;
    reg    [15:0] reg_b;
    reg     [4:0] count;
    wire   [16:0] sub_out = {reg_r,reg_q[31]} - {1'b0,reg_b};  // sub
    wire   [15:0] mux_out = sub_out[16]?                      // restoring
                  {reg_r[14:0],reg_q[31]} : sub_out[15:0];    // or not
    assign q = reg_q;
    assign r = reg_r;
    always @ (posedge clk or negedge clrn) begin
        if (!clrn) begin
            busy  <= 0;
            ready <= 0;
        end else begin
            if (start) begin
                reg_q <= a;                                   // load a
                reg_b <= b;                                   // load b
                reg_r <= 0;
                busy  <= 1;
                ready <= 0;
                count <= 0;
            end else if (busy) begin
                reg_q <= {reg_q[30:0],~sub_out[16]};          // << 1
                reg_r <= mux_out;
                count <= count + 5'b1;                        // counter++
                if (count == 5'h1f) begin                     // finished
                    busy  <= 0;
                    ready <= 1;                               // q,r ready
                end
            end
        end
    end
endmodule
```

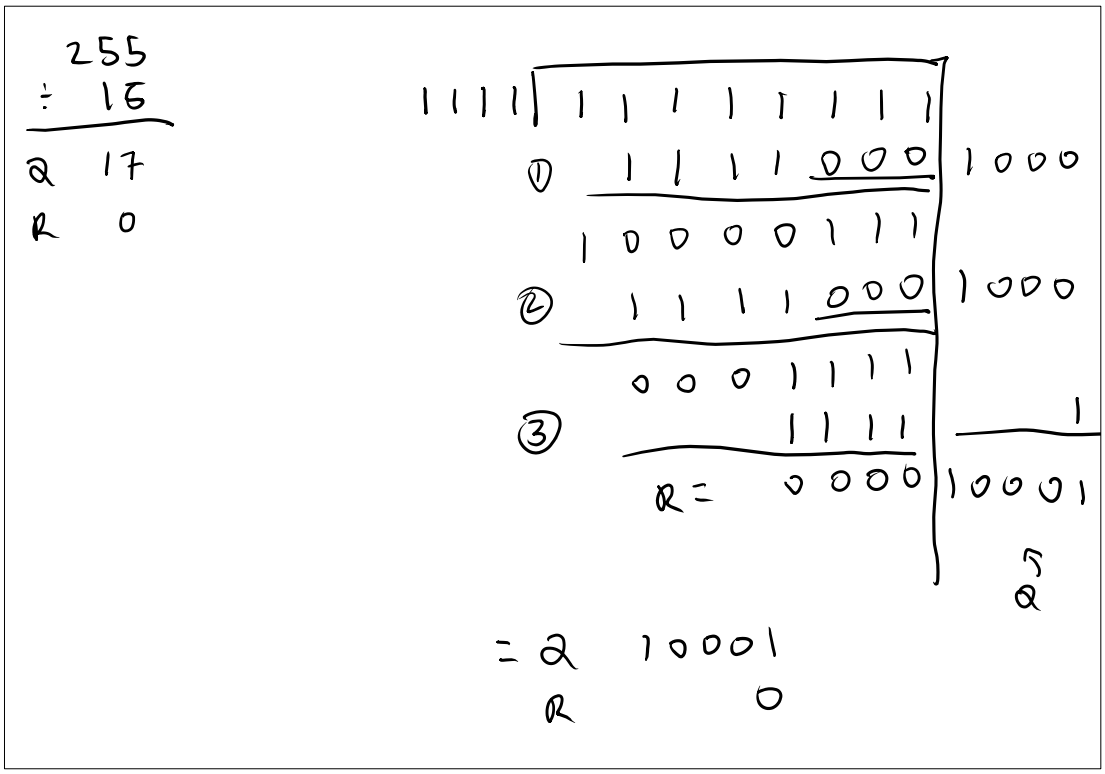
Newton-Raphson

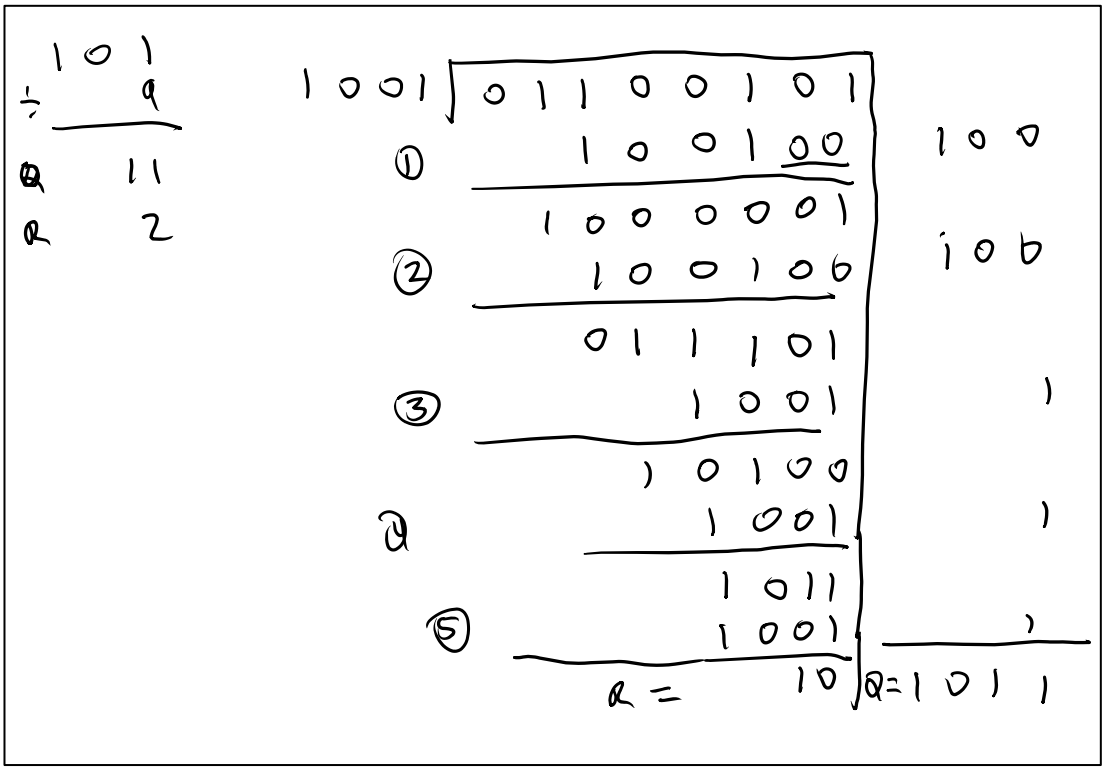



```
module newton (a,b,start,clk,clrn,q,busy,ready,count);
    input  [31:0] a;                                  // dividend: .1xxx...x
    input  [31:0] b;                                  // divisor:  .1xxx...x
    input         start;                              // start
    input         clk, clrn;                          // clock and reset
    output [31:0] q;                                  // quotient: x.xxx...x
    output reg    busy;                               // busy
    output reg    ready;                              // ready
    output  [1:0] count;                              // counter
    reg    [33:0] reg_x;                              // xx.xxxxx...xx
    reg    [31:0] reg_a;                              //   .1xxxx...xx
    reg    [31:0] reg_b;                              //   .1xxxx...xx
    reg     [1:0] count;
```

```
// x_{i+1} = x_i * (2 - x_i * b)
wire   [65:0] axi =  reg_x * reg_a;                   //  xx.xxxxx...x
wire   [65:0] bxi =  reg_x * reg_b;                   //  xx.xxxxx...x
wire   [33:0] b34 = ~bxi[64:31] + 1'b1;               //   x.xxxxx...x
wire   [67:0] x68 =  reg_x * b34;                     // xxx.xxxxx...x
wire    [7:0] x0  =  rom(b[30:27]);
assign        q   =  axi[64:33] + |axi[32:30];        // rounding up
always @ (posedge clk or negedge clrn) begin
    if (!clrn) begin
        busy  <= 0;
        ready <= 0;
    end else begin
        if (start) begin
            reg_a <= a;                               //   .1xxxx...x
            reg_b <= b;                               //   .1xxxx...x
            reg_x <= {2'b1,x0,24'b0}                  // 01.xxxx0...0
            busy  <= 1;
            ready <= 0;
            count <= 0;
        end else begin
            reg_x <= x68[66:33];                      // xx.xxxxx...x
            count <= count + 2'b1;                    // count++
            if (count == 2'h2) begin                  // 3 iterations
                busy  <= 0;
                ready <= 1;                           // q is ready
            end
        end
    end
```

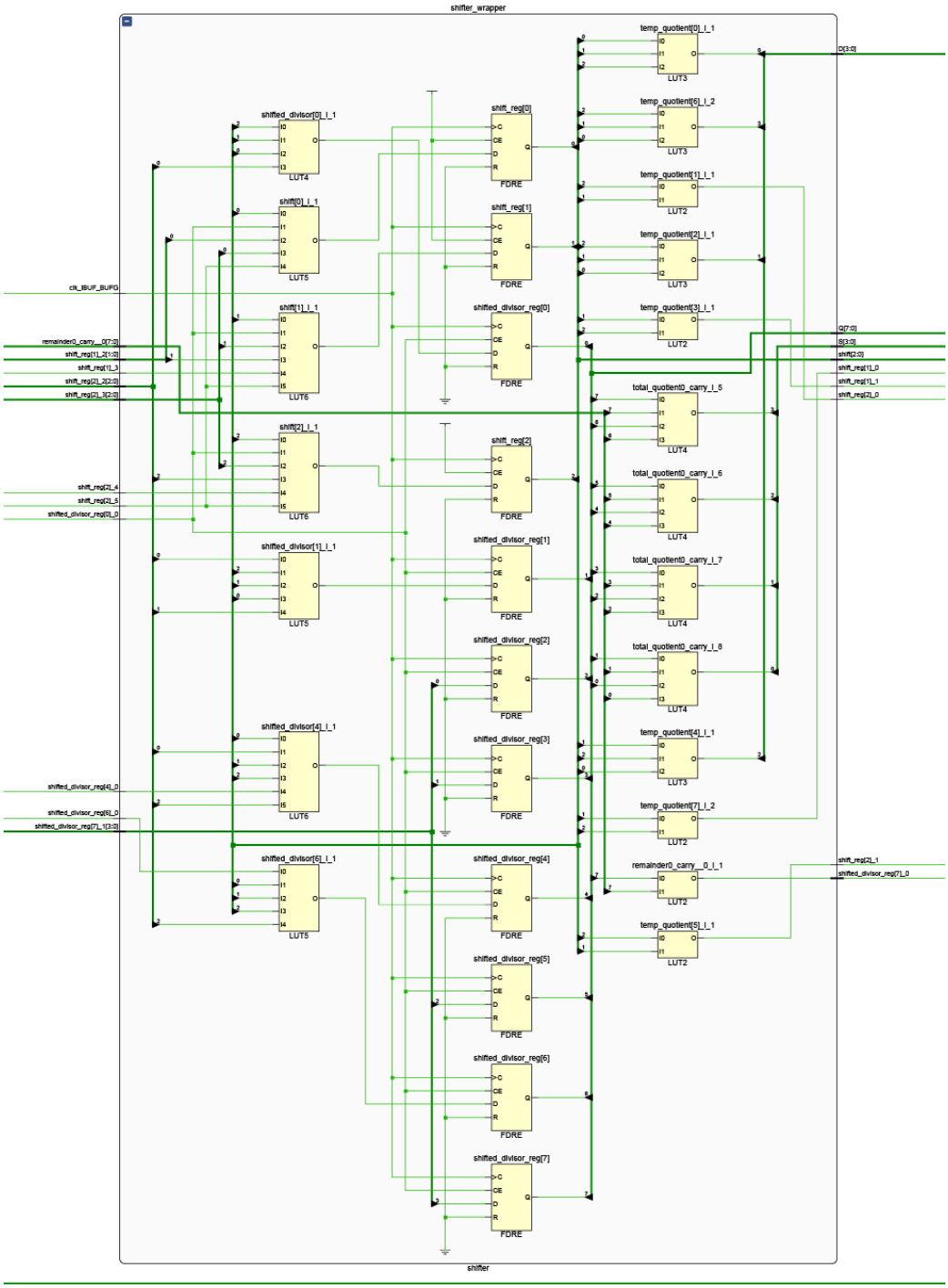
Appendix 2.1


255
÷ 16
Q 17
R 0
1111
1 1 1 1 1 1 1 1
① 1 1 1 1 0 0 0 0
1 0 0 0 0 1 1 1
② 1 1 1 1 0 0 0 0
0 0 0 1 1 1 1
③ 1 1 1 1
R = 0 0 0 0
1000
1000
1
10001
Q
= 2 10001
R 0


101
÷ 9
Q 11
R 2
1001
0 1 1 0 0 1 0 1
① 1 0 0 1 0 0
1 0 0 0 0 1
② 1 0 0 1 0 0
0 1 1 1 0 1
③ 1 0 0 1
1 0 1 0 0
④ 1 0 0 1
1 0 1 1
⑤ 1 0 0 1
R = 10
100
100
1
1
1
Q=1011

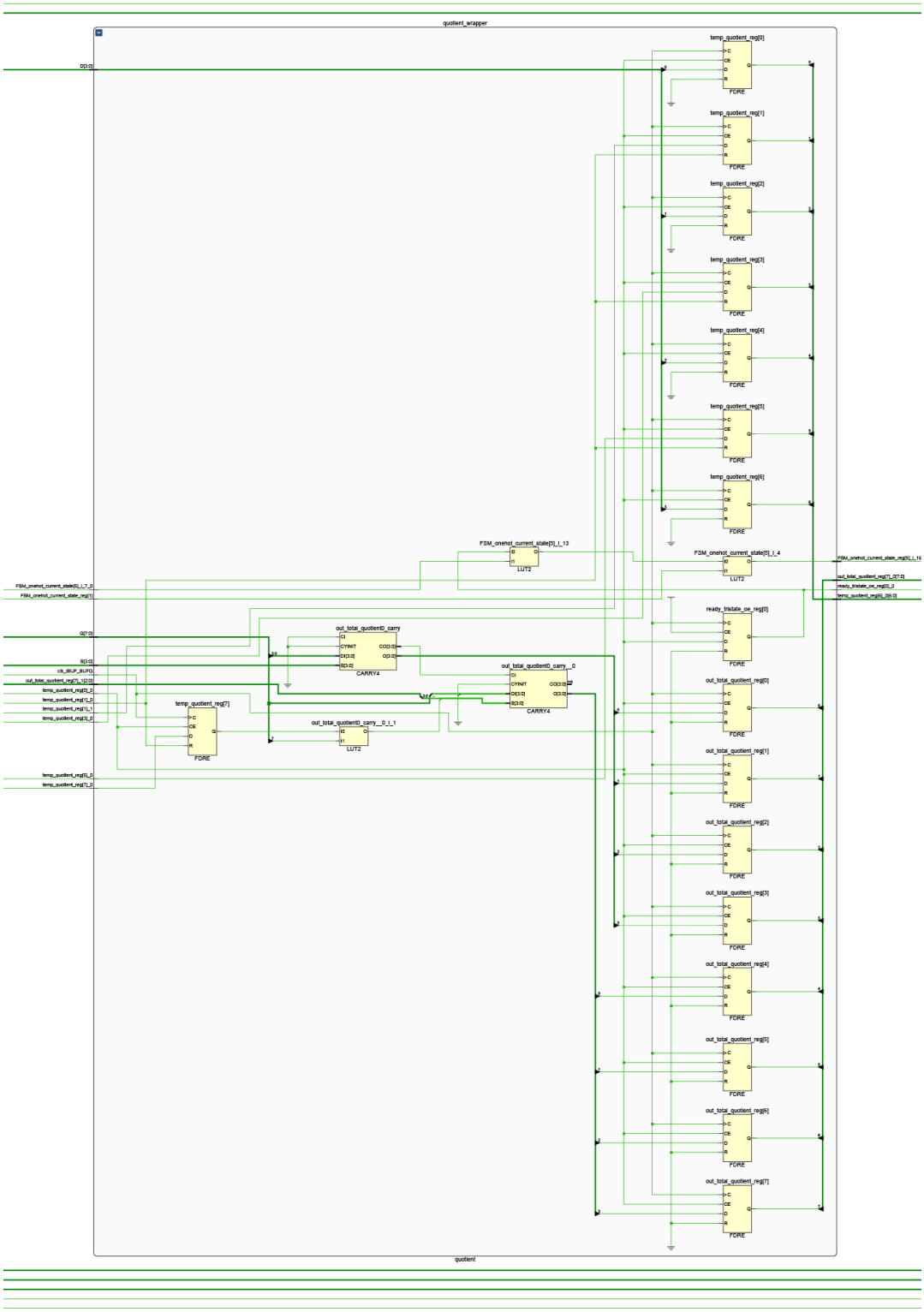
Appendix 3.1

Shifter Stage



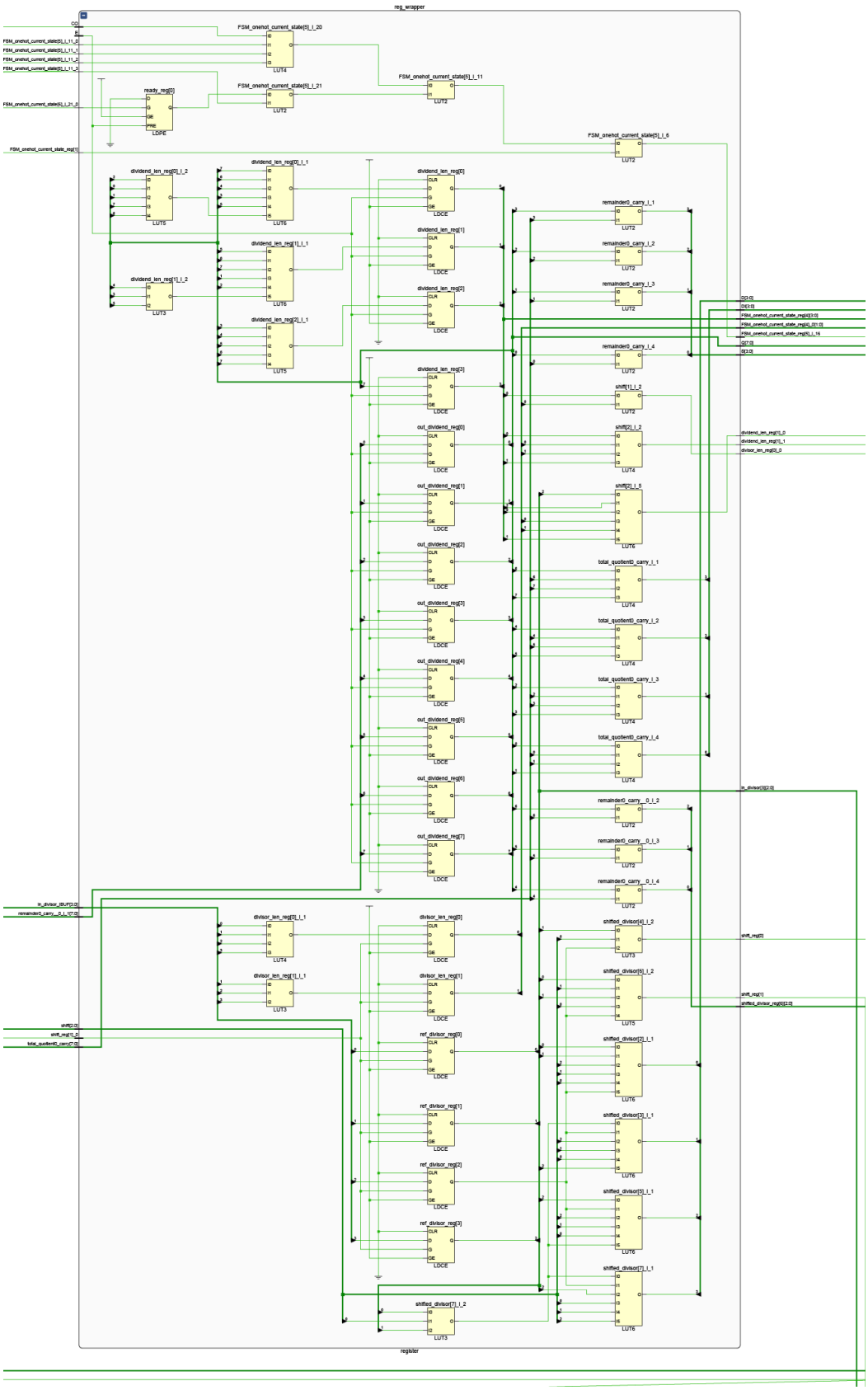
Appendix 3.2

Quotient Stage



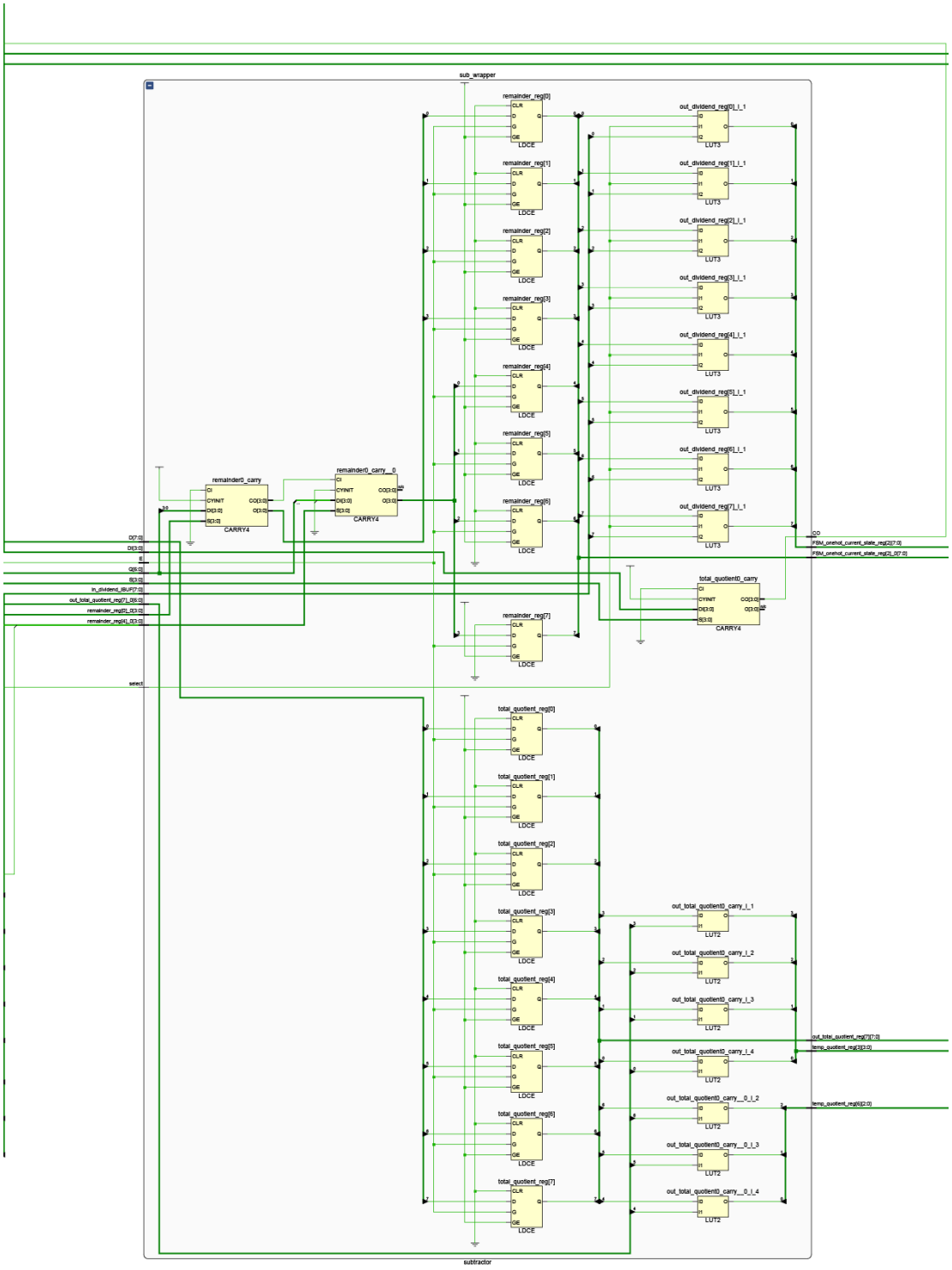
Appendix 3.3

Register Stage

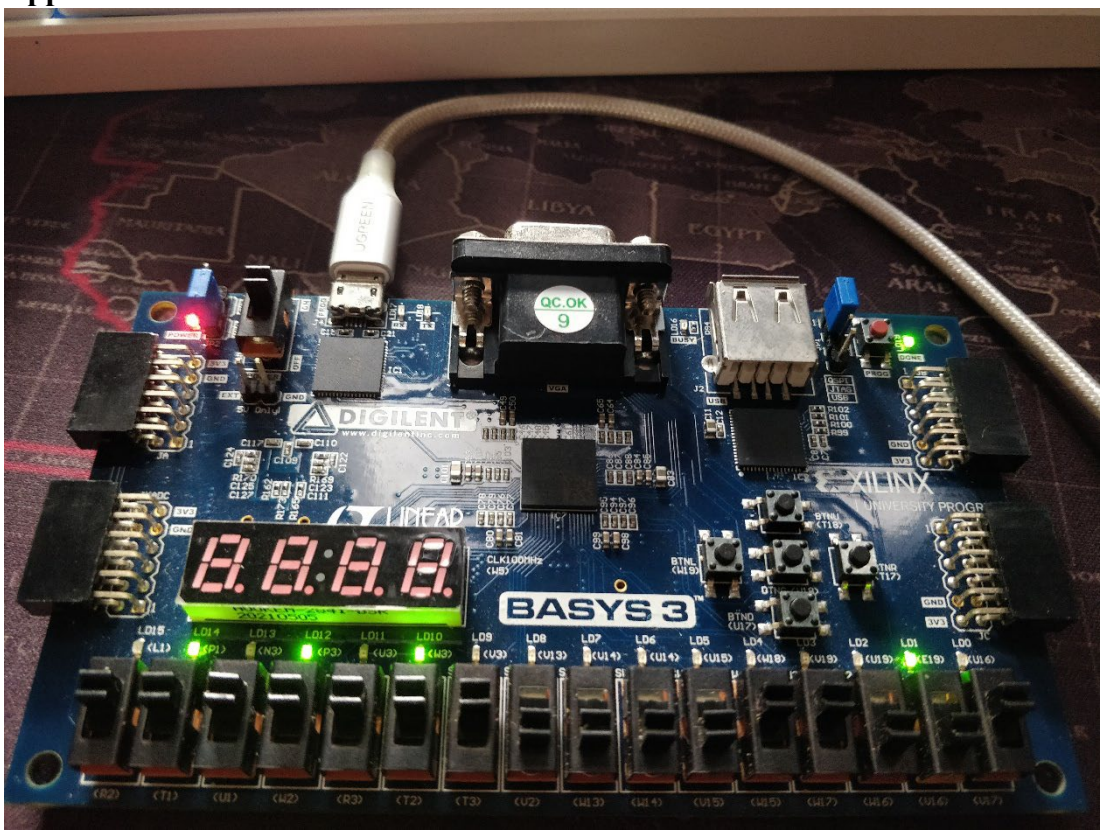


Appendix 3.4

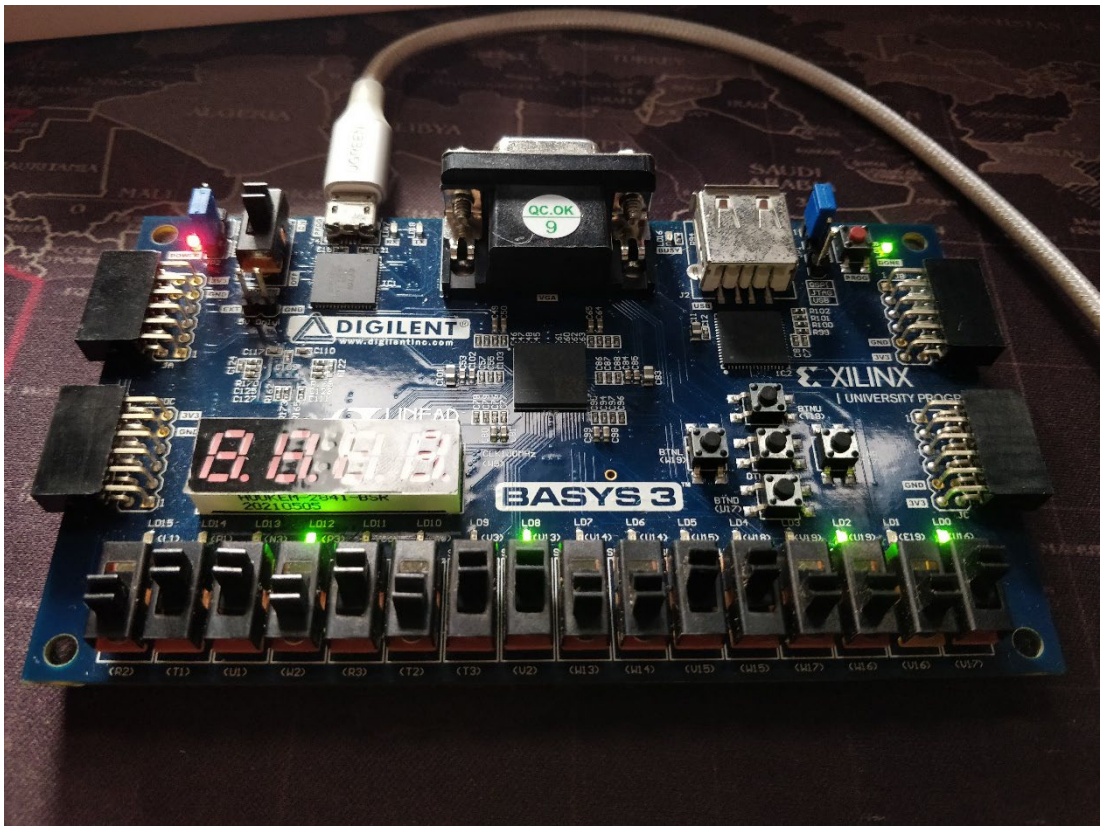
Subtractor stage



Appendix 4.1



255 / 3 = 84 r 2



107/6 = 17 r 5

Appendix 5

karakuve/Mental_ArithmeticHDU: Mental Arithmetic based Hardware Division Unit (github.com)

top.v

```
`timescale 1ns / 1ps

module top(
   input clk,
   input write,
   input clear,
   input [7:0] in_dividend,
   input [3:0] in_divisor,
   output [7:0] total_quotient,
   output [7:0] remainder
);

   wire select;
   wire [2:0] ready;
   wire [3:0] current_state;
   wire [7:0] out_dividend;
   wire [2:0] divisor_len;
   wire [3:0] dividend_len;
   wire [3:0] ref_divisor;
   wire [7:0] shifted_divisor;
   wire [7:0] current_total_quotient;
   wire [2:0] shift;

   controller controller_wrapper
   (
      .clk(clk),
      .clear(clear),
      .ready(ready),
      .write(write),
      .select(select),
      .current_state(current_state)
   );

   register reg_wrapper
   (
      .select(select),
      .current_state(current_state),
      .in_dividend(in_dividend),
      .in_divisor(in_divisor),
```

```
        .remainder(remainder),
        .ref_divisor(ref_divisor),
        .out_dividend(out_dividend),
        .dividend_len(dividend_len),
        .divisor_len(divisor_len),
        .ready(ready)
    );

    shifter shifter_wrapper
    (
        .clk(clk),
        .current_state(current_state),
        .in_divisor (ref_divisor),
        .divisor_len(divisor_len),
        .dividend_len(dividend_len),
        .shifted_divisor(shifted_divisor),
        .shift(shift),
        .ready(ready)

    );

    quotient quotient_wrapper
    (
        .clk(clk),
        .clear(clear),
        .current_state(current_state),
        .in_total_quotient(total_quotient),
        .out_total_quotient(current_total_quotient),
        .shift(shift),
        .ready(ready)
    );

    subtractor sub_wrapper
    (
        .current_state(current_state),
        .shifted_divisor(shifted_divisor),
        .in_dividend(out_dividend),
        .current_total_quotient(current_total_quotient),
        .total_quotient(total_quotient),
        .remainder(remainder),
        .clear(clear),
        .ready(ready)
    );
endmodule
```

controller.v

```
`timescale 1ns / 1ps

module controller(
   input clk,
   input clear,
   input write,
   input [2:0] ready,
   output reg select,
   output reg [3:0] current_state
);

   parameter
   IDLE = 4'b0000,
   REG =4'b0001,
   SHIFT = 4'b0010,
   QUO = 4'b0011,
   SUB = 4'b0100,
   TOTAL = 4'b1000;

   reg [3:0] next_state;

   always @ (posedge clk or negedge clear)
   begin
      if(!clear)
         begin
            current_state <= IDLE;
         end
      else
         current_state <= next_state;
   end

   always @ (current_state, ready, write, select)
   begin
      case (current_state)
         IDLE:
         begin

            if(write == 1'b1)
               begin
                  select <= 1'b0;
                  next_state <= REG;
               end
            else
```

```
            next_state <= IDLE;
        end

        REG:
        begin
            if(ready == 3'b001)
                next_state <= SHIFT;
            else
                next_state <= REG;
        end

        SHIFT:
        begin
            if(select ==1'b0)
                select <= 1'b1;
            if(ready == 3'b010)
                next_state <= QUO;
            else
                next_state <= SHIFT;
        end

        QUO:
        begin
            if(ready == 3'b011)
                next_state <= SUB;
            else
                next_state <= QUO;
        end

        SUB:
        begin
            if(ready == 3'b100)
                next_state <= REG;
            else if(ready == 3'b101)
                next_state <= TOTAL;
            else
                next_state <= SUB;
        end

        TOTAL:  next_state <= TOTAL;

        default:
        next_state  <= current_state;
    endcase
```

```
    end
endmodule
```

register.v

```
`timescale 1ns / 1ps

module register(
   input select,
   input [3:0] current_state,
   input [7:0] in_dividend,
   input [3:0] in_divisor,
   input [7:0] remainder,
   output reg [7:0] out_dividend,
   output reg [3:0] ref_divisor,
   output reg [3:0] dividend_len,
   output reg [2:0] divisor_len,
   output reg [2:0] ready
);

   always @(current_state or in_dividend or select or in_divisor or remainder or
out_dividend )
   begin
      if(current_state == 3'b0001)
         begin
            if (select == 1'b0)
               begin

                  out_dividend <= in_dividend;
                  ref_divisor <= in_divisor ;

                  if(in_divisor[3]==1'b1)
                     divisor_len <= 3'b100;
                  else if(in_divisor[2] == 1'b1)
                     divisor_len <= 3'b011;
                  else if(in_divisor[1] == 1'b1)
                     divisor_len <= 3'b010;
                  else if(in_divisor[0] == 1'b1)
                     divisor_len <= 3'b001;
                  else
                     divisor_len <= 3'b000;
               end
            else if (select == 1'b1)
               begin
                  out_dividend <= remainder;
               end


            if(out_dividend[7] == 1'b1)
```

```
            dividend_len <= 4'b1000;
          else if (out_dividend[6] == 1'b1)
            dividend_len  <= 4'b0111;
          else if (out_dividend[5] == 1'b1)
            dividend_len  <= 4'b0110;
          else if (out_dividend[4] == 1'b1)
            dividend_len <= 4'b0101;
          else if (out_dividend[3] == 1'b1)
            dividend_len <= 4'b0100;
          else if (out_dividend[2] == 1'b1)
            dividend_len  <= 4'b0011;
          else if (out_dividend[1] == 1'b1)
            dividend_len  <= 4'b0010;
          else if (out_dividend[0] == 1'b1)
            dividend_len <= 4'b0001;
          else
            dividend_len <= 4'b0000;

          ready <= 3'b001;

        end
      else
        ready =3'bzzz;
    end
endmodule
```

shifter.v

```
`timescale 1ns / 1ps

module shifter(
   input clk,
   input [3:0] current_state,
   input [7:0] in_divisor,
   input [3:0] dividend_len,
   input [2:0] divisor_len,
   output reg [2:0] shift,
   output reg [7:0] shifted_divisor,
   output reg [2:0] ready
);

   reg [7:0] temp_quotient;
   always @ (posedge clk)
   begin
      if (current_state == 4'b0010)
         begin
            if (dividend_len > divisor_len)
               begin
                  shift <= dividend_len - divisor_len - 1;

               end
            else
               begin
                  shift <= 0;
               end

            if (shift>0)
               begin
                  shifted_divisor <= in_divisor << shift;
               end
            else
               shifted_divisor <= in_divisor ;

            if (shift >= 0 && shifted_divisor >= 0 )
               ready <= 3'b010;
         end
      else
         ready <= 3'bzzz;
   end
endmodule
```

quotient.v

```
`timescale 1ns / 1ps

module quotient(
   input clk,
   input clear,
   input [3:0] current_state,
   input [7:0] in_total_quotient,
   input [2:0] shift,
   output reg [7:0] out_total_quotient,
   output reg [2:0] ready
);

   reg [7:0] temp_quotient;

   always @ (posedge clk)
   begin
      if(!clear)
         out_total_quotient <= 8'bzzzzzzzz;

      if (current_state == 4'b0011)
         begin
            if (shift>=0)
            begin
               temp_quotient <= 1'b1 << shift;
            end

            if(in_total_quotient > 0)
               out_total_quotient <= in_total_quotient + temp_quotient;
            else
               out_total_quotient <= temp_quotient ;

            ready <=3'b011;
         end
      else
         ready <= 3'bzzz;

   end
endmodule
```

subtractor.v

```
`timescale 1ns / 1ps

module subtractor(
   input clear,
   input [3:0] current_state,
   input [7:0] shifted_divisor,
   input [7:0] in_dividend,
   input [7:0] current_total_quotient,
   output reg [7:0] total_quotient,
   output reg [7:0] remainder,
   output reg [2:0] ready
);

   always @(current_state or in_dividend or shifted_divisor or current_total_quotient or
clear)
   begin

      if (!clear)
         total_quotient <= 8'bzzzzzzzz;

      if(current_state == 4'b0100)
         begin


            if (in_dividend >= shifted_divisor)
               begin
                  total_quotient  <= current_total_quotient;
                  remainder <= in_dividend - shifted_divisor;
                  ready <= 3'b100;
               end
            else
               begin
                  remainder <= in_dividend ;
                  ready <= 3'b101;
               end

         end
      else
         ready <= 3'bzzz;
   end
endmodule
```