

IMPLEMENTATION AND PHYSICAL DESIGN OF 8/4-BIT SIGNED DIVIDER

*A Dissertation submitted in partial fulfilment of the requirements for the
award of the Degree of*

BACHELOR OF ENGINEERING

IN

ELECTRONICS AND COMMUNICATION ENGINEERING

SHAIK MUSHARRAF ALI	1604-18-735-311
MOHAMMED ABDUR RAHMAN	1604-18-735-119
SYED MOIZUDDIN	1604-18-735-112



Department of Electronics and Communication Engineering

Muffakham Jah College of Engineering and Technology

Banjara Hills, Hyderabad-500 034

(Affiliated to Osmania University)

2022

IMPLEMENTATION AND PHYSICAL DESIGN OF 8/4-BIT SIGNED DIVIDER

*A Dissertation submitted in partial fulfilment of the requirements for the
award of the Degree of*

BACHELOR OF ENGINEERING IN
ELECTRONICS AND COMMUNICATION ENGINEERING BY

MOHAMMED ABDUR RAHMAN	1604-18-735-119
SHAIK MUSHARRAF ALI	1604-18-735-112
SYED MOIZ UDDIN	1604-18-735-311

Under the guidance of
MOHD ZAKIR HUSSAIN
Assistant Professor, Department of ECE, MJCET



Department of Electronics and Communication Engineering

Muffakham Jah College of Engineering and Technology

Banjara Hills, Hyderabad-500 034

(Affiliated to Osmania University)

May 2022



**MUFFAKHAM JAH
COLLEGE OF ENGINEERING & TECHNOLOGY**

(Est. by Sultan-Ul-Uloom Education Society in 1980)

(Affiliated to Osmania University, Hyderabad)

Approved by the AICTE & Accredited by NBA

CERTIFICATE

This is to certify that the dissertation titled '**Implementation and Physical Design of 8/4-bit Signed Divider**' submitted by **Mohammed Abdur Rahman (Roll No: 1604-18-735-119)**, **Shaik Musharraf Ali (Roll No: 1604-18-735-311)**, **Syed Moiz Uddin (Roll No: 1604-18-735-112)** in partial fulfilment of the requirements for the award of the Degree of **Bachelor of Engineering**, is a bonafide record of work carried out by them under my guidance and supervision during the academic year 2021-2022.

The results embodied in this thesis have not been submitted to any University or Institute for the award of any Degree or Diploma.

Mohd Zakir Hussain

Project supervisor

Asst. Professor, ECED, MJCET,

Hyderabad

Dr. Mohammed Arifuddin Sohel

Professor and Head

Dept. of ECE

MJCET, Hyderabad

INTERNAL EXAMINER

EXTERNAL EXAMINER

DECLARATION

We hereby declare that the work presented in this dissertation entitled '**Implementation and physical design of 8/4-bit signed divider**' submitted in partial fulfilment of the requirement for the award of the Degree of **Bachelor of Engineering**, in the Department of Electronics and Communication Engineering, Muffakham Jah College of Engineering and Technology, Hyderabad is an authentic record of our own work carried out from November 2021 to April 2022 under the guidance and supervision of **Mohd Zakir Hussain** , Professor, Department of ECE, MJCET.

We have not submitted the matter embodied in this dissertation for the award of any other degree or diploma.

Mohammed Abdur Rahman

Shaik Musharraf ali

Syed Moiz Uddin

ACKNOWLEDGEMENT

At this moment of accomplishment, we are presenting our work with great pride and pleasure, we would like to express our sincere gratitude to all those who helped us in the successful completion of our venture.

We are very much indebted and express our deep sense of gratitude to **Mohd Zakir Hussain(Professor ECED, MJCET)**, our project supervisor, who took keen interest throughout the course of the project. Our zeal was kept alive throughout the project and he provided the much-needed guidance and unflinching support to complete the project. Without his constant encouragement and guidance this dissertation work could not have taken the present shape.

We sincerely thank the Project Coordinators: Dr. Kaleem Fatima, Professor, ECE Department, MJCET, and Mr. J. K. Nag, Associate Professor, Department of ECE for the in-depth and tough appraisals conducted by them and for their timely and qualitative suggestions that truly helped in improving the quality of the project work. More importantly the thesis writing guidelines provided by them was a blessing in disguise to write the thesis in proper structure and format overcoming major drawbacks.

We would like to thank **Dr. Mohammed Arifuddin Sohel**, Professor and Head, Department of Electronics and Communication Engineering, for his endless support and guidance throughout the course. All his valuable suggestions and feedback has helped us a lot to mould our self. He has also been a great source of inspiration in terms of work and studies.

We also extend our deepest sense of gratitude and thanks towards our Advisor

cum Director, **Dr. Basheer Ahmed, and Dr. Mahipal Singh Rawat**, Principal MJCET, for their constant support and encouragement.

We also thank other faculty members of the Department of Electronics and Communication Engineering for their valuable suggestions in the project reviews and for their invaluable support and guidance throughout the engineering career. We also thank the entire non-teaching staff of the Department of Electronics and Communication Engineering for their help and support during the course of the project and throughout the engineering career.

We would also like to thank our parents and friends for their over whelming and whole hearted encouragement and support without which this would not have been successful.

Above all we thank Almighty for constantly motivating us with His blessings, and giving us courage at each stride to step forward with confidence and self –belief.

Mohammed Abdur Rahman

Shaik Musharraf ali

Syed Moiz Uddin

TABLE OFContents

CHAPTER 1 INTRODUCTION.....	13
INTRODUCTION:	13
1.1 SCOPE INVLSI :	13
1.2 NECESSITY FOR MODIFICATION IN ALOGORITHM.....	14
CHAPTER 2.....	22
PERFORMANCE ANALYSIS OF VARIOUS DIVISION ALGORITHMS FOR LARGE NUMBERS	22
2.1 INTRODUCTION:	22
2.2 Novel Low power and high-speed array divider in 65 nm Technology	29
2.3 TRANSLATION OF DIVISION ALGORITHM INTO VERILOG HDL.....	32
2.4 RESULTS AND DISCUSSIONS:.....	37
CHAPTER 3.....	43
DESIGN METHODOLOGY AND REQUIRED TOOLS FOR IMPLEMENTATION OF OUR PROPOSED DESIGN	44
3.1 INTRODUCTION	44
3.2 THE PROPOSED ALGORITHM FOR SIGNED DIVIDER :	44
3.3 OPENSOURCE EDA TOOLS REQUIRED FOR IMPLEMENTATION OF REQUIRED DESIGN MODULE ARE:.....	46
Floorplan Commands:.....	48
Placement Commands	48
PDN Generation Commands	48
Routing Commands	48
Magic Commands	49
Klayout Commands	49

3.4 OUR PROPOSED VERILOG CODE FOR REQUIRED MODULE :	49
3.5 RESULTS OF PROPOSED VERILOG MODULE AND TESTBENCH:.....	53
CHAPTER 4.....	63
EXPERIMENT MODULES.....	63
4.1 INTRODUCTION:	63
CHAPTER 5.....	72
DISCUSSION AND CONCLUSION.....	72

LIST OF FIGURES

Figure1. 1.....	18
Figure1. 2.....	19
Figure1. 3.....	20
Figure2. 1.....	25
Figure2. 2.....	26
Figure2. 3.....	27
Figure2. 4.....	34
Figure2. 5.....	35
Figure2. 6.....	36
Figure2. 7.....	39
Figure2. 8.....	41
Figure2. 9.....	43
Figure 3. 1.....	44
FIGURE 3. 2.....	46
FIGURE 3. 3.....	53
FIGURE 3. 4.....	54
FIGURE 3. 5.....	55
FIGURE 3. 6.....	56
FIGURE 3. 7.....	57
FIGURE 3. 8.....	57
FIGURE 3. 9.....	58
FIGURE 3. 10.....	59
FIGURE 3. 11.....	59
FIGURE 3. 12.....	60
FIGURE 3. 13.....	61
FIGURE 3. 14.....	61
FIGURE 3. 15.....	62
FIGURE 4. 1.....	64
FIGURE 4. 2.....	67
FIGURE 4. 3.....	68
FIGURE 4. 4.....	71
FIGURE 5. 1.....	73

LIST OF TABLES

TABLE2. 1.....	24
TABLE2. 2.....	28
TABLE2. 3.....	28
TABLE2. 4 optimum value of 8-bit adder.....	31
TABLE2. 5 Optimum value of 8-bit subtractor	31
TABLE2. 6.....	37
TABLE2. 7FPGA Implementation for 16-bit Divider.	38

ABSTRACT

VLSI is one of the most widely used technologies for manufacturing microchip processors, integrated circuits (IC), and component designing. It was initially designed to support hundreds of thousands of transistor gates on a microchip which is now exceeded several billion per the data of 2012. All of these transistors are remarkably integrated and embedded within a microchip that has shrunk over time but still can hold enormous amounts of transistors.

The first 1-megabyte RAM was built on top of VLSI design principles and included more than one million transistors on its microchip die.

This project focuses on implementing a signed binary divider using Verilog and performing a physical design process i.e., register transfer level (RTL) to graphic design systemII (GDSII) on 180nm and 45nm technology nodes to analyze different parameters such as delay, area, power, and bandwidth. It also extends the unsigned divider to the signed divider and hence increases the range of division.

We implemented and designed 8/4-bit signed divider in the Verilog code by using open-source EDA (electronic design automation) tools

FRONT END

Firstly we perform functional simulation on the Verilog module and testbench of the required design, to verify whether the module and testbench are functional correct or not. It is done through the opensource eda tools called iverilog and check the output wave form of the simulation through gtkwave which is also an open source tool used as wave-viewer to view the output waves. Then in the next step we perform synthesis (it is like compiler) for required Verilog module along with the skywater130 liberty file which is an opensource liberty file. The main tool which is used for synthesis is yosys (opensource eda). Synthesis is a very important process for designing as it is used to convert RTL into simple logic gates, then mapping those gates to actual technology-dependent logic gates available in the technology libraries. After synthesis we get netlist file which contains information regarding the logical connectivity of all standard cells and macros.

BACK END

We perform backend process step by step through ubuntu which also an opensource eda tools, in backend we first perform synthesis again to get time level simulation by using Verilog module of the design, netlist file and along with the liberty file skywater 130 what we have done previous in synthesis to get gate. Then go for floorplan, placement and global routing of the standard cell by using openlane comments step by step from netlist file, after that we go for magic and klayout to get the view of the layout of the design and GDS 2 file which we get at the output of the layout which later used by the foundry people for fabrication.

By using all the opensource eda we got the report of area, power and delay of the required module along with the layout design and gds2 file.

CHAPTER 1

INTRODUCTION

1.1INTRODUCTION:

VLSI, Very Large-Scale Integration technology is used primarily for the IC design in the Semiconductor Industry which is part of the global electronics industry. So, VLSI always depends on the trends of the electronics industry. Currently, the business of electronics industry is driven mostly by the growth of mobile devices, automotive sector, cloud servers and other upcoming fields like IOT, Internet Of Things. So, VLSI which is part of the electronics field is also growing based on the developments in the electronics industry. For example, IoT is growing in a big way through mobile app-based services like Uber/Ola, Industrial automation, Smart Home and Smart Cities. So IOT demands new wireless and wired protocols to realise the connected devices. Protocols like Bluetooth, WIFI, 5G, Zigbee, Thread, etc will evolve and therefore we chip designers will create new IPs, Chips and SoCs to support the new protocols. Also, the growth of Artificial Intelligence demands new kinds of powerful processors and memories for the high-speed execution. The upcoming fields like self-driven cars demand the growth of artificial intelligence, especially machine learning. So, VLSI will also grow to support upcoming technologies like artificial intelligence.

In the Semiconductor industry, there are so many fields and specialisations like System Design, TLM [Transaction Level Modelling], IP- RTL Design, Design Verification, Emulation & Acceleration, FPGA prototyping, Verification IP, Synthesis, DFT, Timing Verification, Physical Design and Verification, Silicon Testing, Silicon Validation, PCB Design, ASIC Library Design, SOC-RTL design and verification, Analog Mixed Signal Verification, Analog IP design, Custom ASIC Design, etc.

1.2SCOPE IN VLSI:

If you are fresher, you can start off your journey with Digital Design and HDL coding and then choose either front-end or back-end ASIC design, based on your interest. Also while working in the industry, you will grow specialised in a particular domain like processor[Intel/AMD], memory, automotive [Infineon/NXP], wireless[Broadcom/Qualcomm], mobileSOC[Samsung/Apple/Nvidia]orIOT[Intel/Google], based on the company/opportunity.

1.3 NECESSITYFOR MODIFICATION IN ALOGORITHM

In recent decade arithmetic logic unit has change to give better result specially in division unit it is implementing various other division algorithm to produce fast and accurate response for complex division operation as per the rapid changing necessities of the modern-day industries

The old convention division algorithm of old computer has many issues not only with the output of the computation but also with design parameters like their area, timing and power consumption in performing the iteration process

Nowaday as per the growing necessity of the arithmetic problem solving which required typical computer to perform complex division operation, to solve this complex division operation the ALU is called hundreds of time per/ sec as per the instruction of the user sec to perform the computation here our old tradition old division algorithm is not enough to solve them accurately and give the result of multiple instructions immediately to the required a division algorithm which

So, in order to get accurate result with precision along with significant design parameter we design this division withnon-restoring algorithm which is very easy to implement as it required only adder and subtractor to get a signed binary divider which consumes only 31% and 33% of delay and power along with 95% area-efficiency.

1.4 AIM

Internet of Things (IoT) makes limitation on power consumption for sensors and portable devices. Thus, exact computation for division processes is not allowed due to the huge amount of power consumption. To address these limitations, a low power division computing units is required. The main aim of this thesis is to implement a 8/4-bit signed divider with ideal specifications for design, parameters like lessarea, less power consumption and less amount of time delay at micro meter/square by using opensource electronic design automation tools.

The objective of the project is to ensure power- hungry division operations to a simple addition and subtraction process.

To overcome the above-mentioned issues related to EDA and PDKs

We are going to use

Open-source EDA Tools (free, no NDA) and Open PDKs (free, no NDA)

Our Design point of view to achieve: -

1. Less power consumption
2. Less area consumption
3. Efficiency (frequency)

*NDA=Non-Disclosure Agreement.

1.5 MOTIVATION

In VLSI Design along with the ALU there is a huge requirement for dividers with less area and less power consumption for division computing applications,

Hence, we decided to opt for this project and implement and design an ideal divider which would be useful for further division applications apart from this we studied COA subject which taught us different ALU operations, different multiplier and divider algorithms in which we learned about the working, usage and advantages the dividers and also how they are utilized in the ALU unit.

1.6 PROBLEM STATEMENT

Internet of Things (IoT) makes limitation on power consumption for sensors and portable devices. Thus, exact computation for division processes is not allowed due to the huge amount of power consumption. To address these limitations, a low power division computing units is required.

In a typical program, an arithmetic unit produces thousands of division computations per second. To compute and produce correct results, the ALU 's division algorithms must be as effective as possible. Existing dividers are good enough but they are used to perform ordinary unsigned division which result in the following cons: -

1. More power consumption
2. Large area
3. Less performance (Frequency)

4. More Latency

Usage of commercial electronic design automation (EDA) tools is not possible by an individual and also for start-up firms because of the following: -

Purchase cost, Maintenance, License renewal, NDA issues

Commercial process design kits (PDKs) are also expensive which individual or startup companies find it difficult to procure.

1.7 Design Methodology

Open Lane is an automated RTL to GDSII flow or method based on several components which including Open ROAD, Yosys, Magic, Netgen, CVC, SPEF-Extractor, CU-GR, Klayout and a number of custom scripts for design exploration and optimization. The flow performs full ASIC implementation steps from RTL (register transfer level) to all the way down to GDSII (graphic design system).

1.8 Algorithm

- 1) First based on sign bit of dividend and divisor, positive values of dividend and divisor is processed
- 2) These positive values are then going to non-restoring divider logic.
- 3) Quotient and remainder generated from divider logic are still incorrect
- 4) Quotient and the remainder is calculated with the help of sign bits of dividend and divisor.

As an example, take a divisor as $(5)_{10}$ or $(0101)_2$ and dividend as $(26)_{10}$ or $(0001\ 1010)_2$. As described in the algorithm. (Note: initial value of quotient is zero).

- 1) First, the value of $dividend(p)$ and $divisor(p)$ is selected based on dividend and divisor sign bit. In this example, both the values are positive, hence there will be no change in the dividend and divisor's value. Moreover, the sign bit is stored as 0 for both.
- 2) Now, subtract the top four MSB from the divisor. Which is $0001 - 0101$.
- 3) The subtraction result is negative. Now the quotient is left-shifted by 1, so there will be no change in the quotient. The count is decreased by 1.
- 4) Now the dividend is left-shifted by 1, therefore the new dividend is $0011\ 0100$.

- 5) The same procedure is repeated. As the top 4 MSB bits of dividend (0011) is less than the divisor, we can shift the quotient, and dividend by 1. count is also decreased by 1.
- 6) The new dividend is 0110 1000. Now the subtraction of dividend top 4 MSB and divisor is positive. So the quotient will be updated to 0001.
- 7) Now, the top 4 MSB of dividend is replaced by subtraction result, which is 0001. Hence the updated dividend is 0001 1000. count is also decreased by 1, and the count is not zero, so left shift the dividend.
- 8) The new dividend is 0011 0000. As the top 4 MSB bits of dividend (0011) is less than the divisor, therefore, we can left shift the quotient and dividend by 1. count is also decreased by one, and dividend is left-shifted by 1.
- 9) The new dividend is 0110 0000. Now the subtraction of dividend top 4 MSB and divisor is positive. So, the quotient will be updated to 0101.
- 10) Now, the top 4 MSB of dividend is replaced by subtraction result, which is 0001. Hence the updated dividend is 0001 0000. count is also decreased by 1, and the count is zero now.
- 11) now results selected based on the value of sign bits of Dividend and divisor. As both Dividend and divisor are positive, the resultant quotient is 0101, and the remainder is 0001.

Flow chart for non-restoring and restoring algorithms is given below in Figure 1.1 and Figure 1.2 respectively.

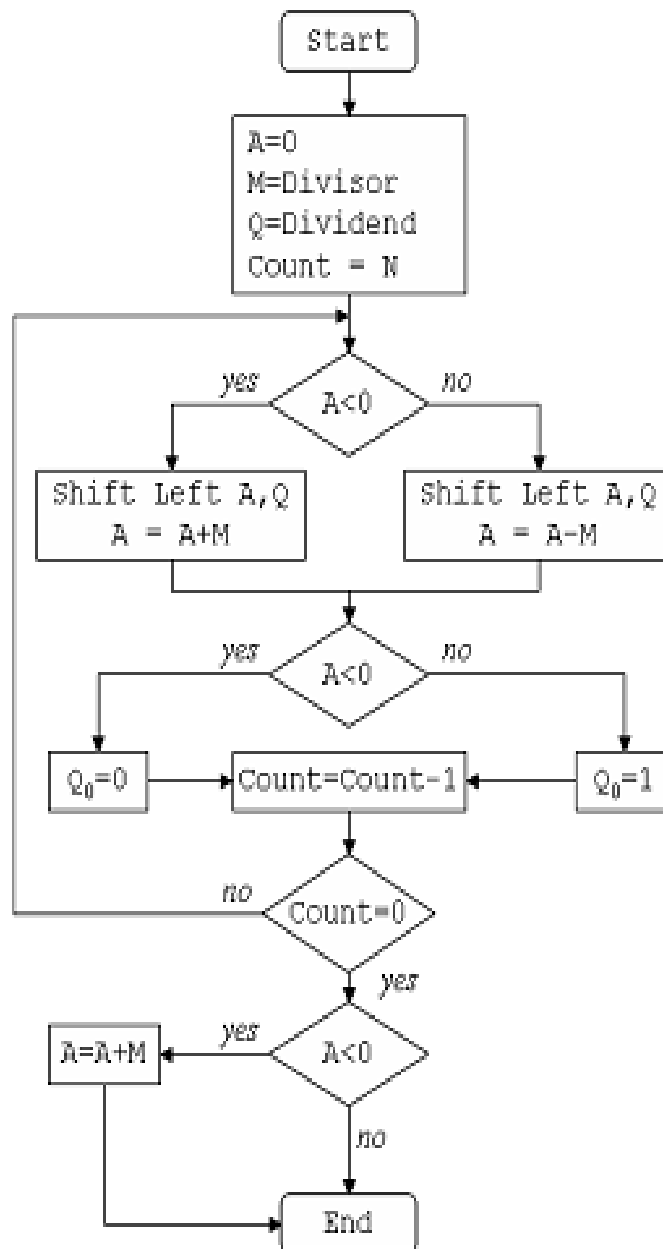


Figure1. 1

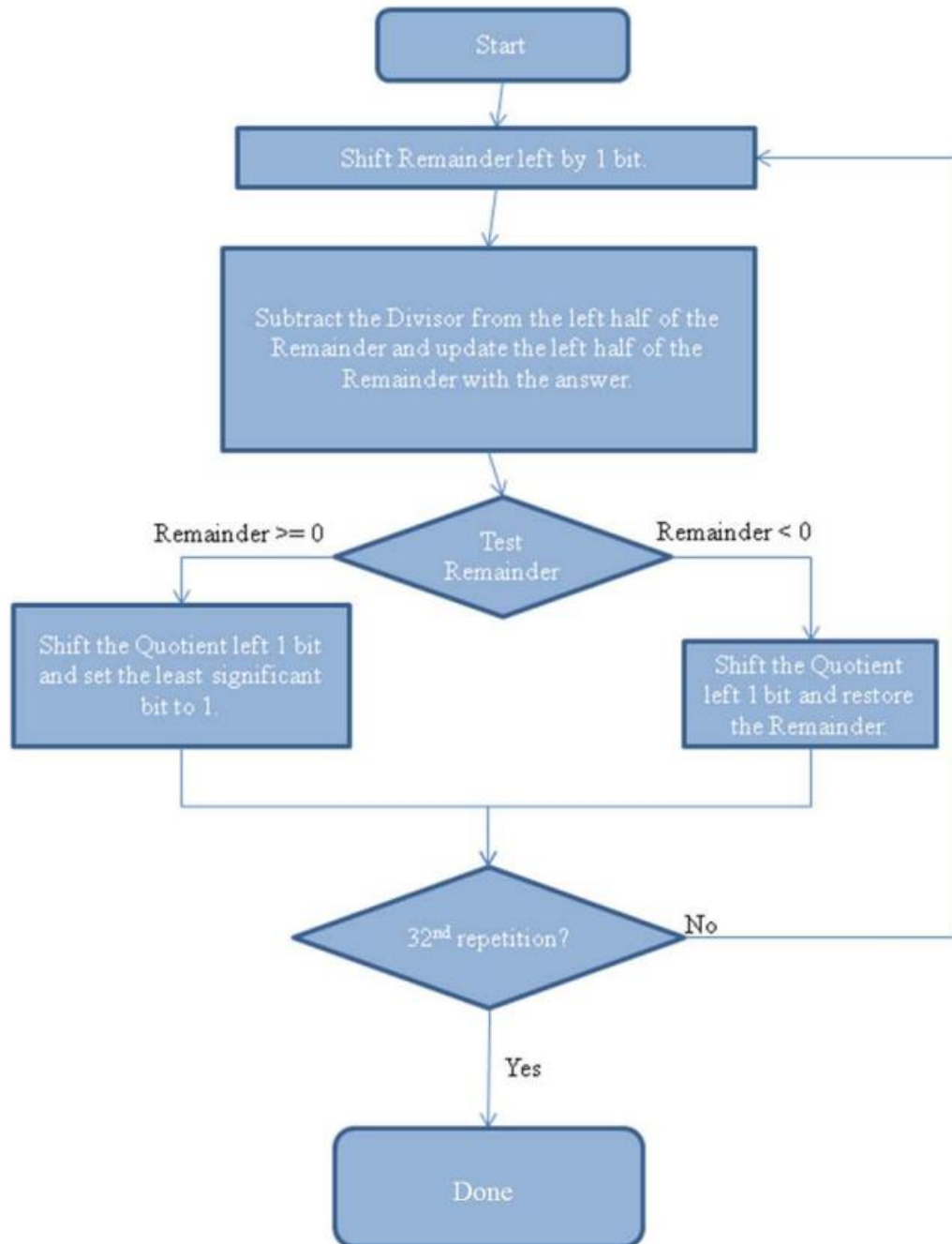


Figure1. 2

The RTL process using Openlane can be show below :-

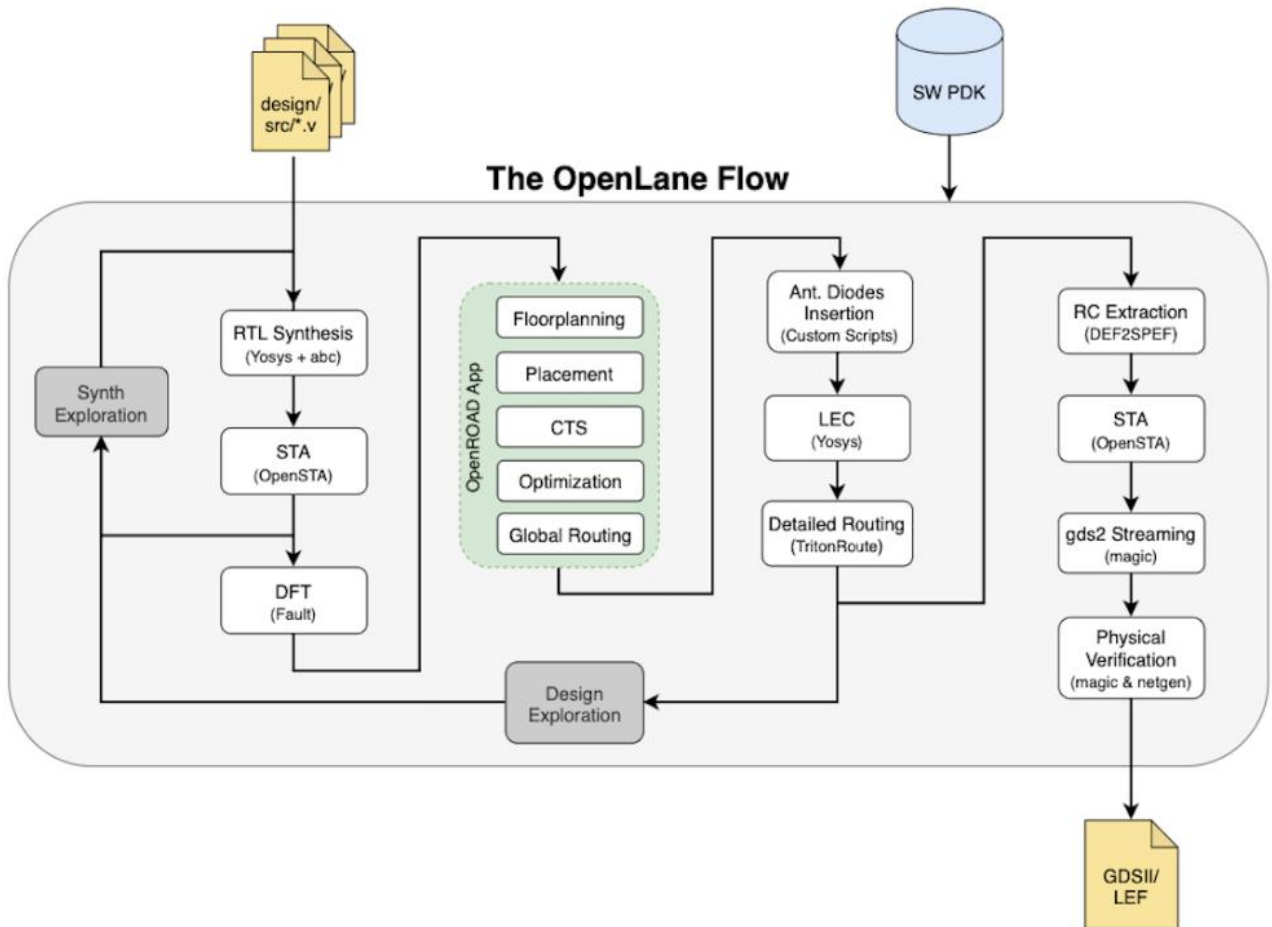


Figure1. 3

This flow is divided into frontend and backend flow during frontend by using iverilog for simulation and yosys(opensource eda tool) we perform simulation and synthesis first we perform simulation of the design Verilog code of the module and then we perform synthesis through which we get netlist file here our frontend is finish

Backend flow we did it through ubuntu(opensource eda tool) through several step one after the other first we placed our design module, testbench, skywater 130 liberty file along with other command related file in one folder by giving it some name with V extension then we make design flow in the docker of the ubuntu by using the design folder which we created through .v extension then we perform synthesis on it by giving random command to know the different design parameter of the

module such as power, area and sta report then we perform floorplan, placement, routing, magic layout and klayout by using similar design flow commands just like we use in synthesis to perform this step after magic layout we get the GDS2 file and through klayout we get information about the layout of the prepared design.

CHAPTER 2

PERFORMANCE ANALYSIS OF VARIOUS DIVISION ALGORITHMS FOR LARGE NUMBERS

2.1 INTRODUCTION:

This paper provides a detailed study on the algorithms used by an ALU to perform multiplication and division for large numbers, and recommends one algorithm that will give best performance for division and multiplication. The multiplication algorithms that are analysed are Pen and Paper algorithm, Booth's algorithm, and Divide and Conquer algorithm. The division algorithms that are analysed are Radix 2 restoring algorithm, Radix 2 non-restoring algorithm, and Radix 4 restoring algorithm. The algorithms are implemented using Verilog and the timing and area reports generated after synthesis is used to compare the algorithms. This paper concludes that out of the examined algorithms divide and conquer algorithm gives the best performance for multiplication, while Radix 4 restoring algorithm gives the best performance for division

This paper which exam each algorithm of both multiplier and divider suggest the algorithms which gives best result for multiplication and division it is possible by making use of multiplier algorithms such as Pen and Paper algorithm, Booth's algorithm, and Divide and Conquer algorithm. These algorithms will be able to multiply any 64-bit signed numbers in 2s complement form and provide a 128-bit result similarly The algorithms examined for division are Radix-2 Restoring Division algorithm, Radix-2 Non-Restoring Division algorithm, and Radix-4 Restoring Division algorithm. These algorithms will be able to divide two positive 64-bit number in 2s complement form

In this we are performing A 64-bit number was chosen for the computation because the latency difference to perform computation on smaller numbers between the different algorithms will be negligible. The criteria used to determine the best performance is the amount of time the algorithm takes to complete one division or multiplication operation, and the area the algorithm needs to implement on the circuit board without any use of pipelining.

Radix-2 Restoring Division

“Digital Recurrence algorithms use subtractive methods to calculate quotients one digit per iteration [3].” Restoring division algorithm is based on the digital recurrence algorithm that “...retire[s] a fixed number of quotient bits in every iteration [3].” Restoring division follows the same method as the pen and paper long division algorithm [2]. In the long division algorithm, the divisor is compared to the left digits of the dividend. If the divisor is bigger than the dividend numbers being compared, then a 0 is appended to the quotient and divisor is shifted to the right to compare with bigger dividend digits. If the divisor is smaller than the dividend, then the divisor being compared is subtracted from the dividend and the result is stored as remainder, while the number of times the divisor can go into the dividend is appended to the quotient. During the next loop, the dividend and the remainder need to be appended together to form the new dividend. This process is repeated until the dividend cannot be divided further by the divisor. The same process is applied in the Radix-2 restoring division algorithm. To decide whether the divisor is bigger than the dividend bits it is being compared to, it subtracts the divisor from the dividend bits and stores the result in remainder field [2]. If the divisor is bigger than the dividend bits, then the result will be negative. If the result is negative, then the remainder is wrong and it must be “restored” to the previous value and a 0 must be appended to the quotient before the divisor is shifted to the right (or dividend shifted to the left) and subtraction is tried again [2]. If the result is positive, then divisor

is bigger than the dividend bits being compared to and the result is valid. Therefore, a 1 is appended to the quotient to indicate that $\text{dividend_bits} - 1 * \text{divisor} = \text{remainder}$. This process is repeated until all bits of the dividend are evaluated.

Using restoring division algorithm in base 10 can be quite lengthy and repetitive since each increment in the quotient is followed by a multiplication and subtract and all nine digits may need to be tested before a restore takes place. In binary restore, however, there are only two choices and a shift can replace multiple iterations required otherwise. The following example uses restoring algorithm to find a solution to 61 divided by 10.

Figure 2.1 presents a detailed flowchart of radix-2 restoring algorithm for 32-bit division, which is further expanded to 64-bit when implemented in Verilog.

Example 3.1 – Restoring Divide example

Dividend z: 0011 1101 (61)

Quotient: 0110 (6)

Remainder: 0001 (1)

Iteration	P	Q
1	Shift z left once: 0111 101 Subtract d from left half of z: 1111 1101 Result is negative. Restore to previous value: 0111 101	0
2	Shift z left once: 1111 01 Subtracting d from left half of z: 0101 01 Result is positive. New z is 0101 01	01
3	Shift z left once: 1010 1 Subtracting d from left half of z: 0000 1 Result is positive. New z is 0000 1	011
4	Shift z left once: 0001 Subtracting d from left half of z: 1 0111 Result is negative: Restore to previous value: 0001	0110

TABLE2. 1

Following the Radix-2 restoring division algorithm discussed above, the first step is to shift the quotient 1 bit to the left. Second, subtract the divisor from the left half of the dividend and update the left half of the quotient with the answer. This new dividend value now has the remainder and rest of the quotient appended together. To avoid destroying the initial dividend value, the dividend can be copied into the remainder register and use remainder register as the dividend value. If the new dividend is less than zero, then shift the quotient left 1 bit and restore the previous value of the dividend. Otherwise, shift the quotient left 1 bit and set the least significant bit to 1. Repeat for procedure 4 times to evaluate all bits of the dividend. At the end of the procedure, the quotient value will be the remainder.

Radix-2 Restoring Division Timing Analysis

Division algorithms are more complex than the multiplication algorithms and therefore, require a much large clock cycle to complete one operation. Shown below is the partial timing analysis generated from Design Compiler. To complete one operation, this implementation of the radix-2 restoring division algorithm requires 324ns clock period. As seen in the multiplication algorithm, the division algorithm's loop iteration is independent on the previous loop iterations. Therefore, the next loop iteration cannot start

until the previous loop is complete. Due to this requirement, the path from start of loop to the end is serialized; hence a large clock period is required. The Verilog simulation of the Radix-2 restoring division algorithm tests special cases and random cases to validate the algorithm. Some of the special cases included division by 1 and division by itself. This simulation does not test division by 0 because the algorithm does not support this case. It is assumed the user will only send valid inputs to the algorithm. The result of various divisions simulated is shown below.

```
x / x: q = 0, r = 0
87 / 5: q = x, r = x
87 / 5: q = 17, r = 2
59 / 20: q = 17, r = 2
59 / 20: q = 2, r = 19
18446744073709551615 / 2: q = 9223372036854775807, r = 1
305419896 / 1: q = 9223372036854775807, r = 1
305419896 / 1: q = 305419896, r = 0
305419896 / 305419896: q = 1, r = 0
$finish at simulation time      26
      V C S  S i m u l a t i o n  R e p o r t
Time: 26
CPU Time:   0.010 seconds;   Data structure size:  0.0Mb
Tue Mar 23 07:40:02 2010
```

Figure2. 1

Report : timing

-path full

-delay max

-max_paths 1

Design : Restore

Version: Z-2007.03

Date : Mon Mar 22 11:58:27 2010

Operating Conditions: BCCOM Library: lsi_10k

Wire Load Model Mode: top

Startpoint: Divisor[1] (input port)

Endpoint: Remainder_reg[59] (rising edge-triggered flip-flop clocked by clk)

Path Group: clk

Path Type: max

Point	Incr	Path

clock (input port clock) (rise edge)	0.00	0.00
input external delay	0.00	0.00 f
Divisor[1] (in)	0.00	0.00 f
sub_41/B[1] (Restore_DW01_sub_63)	0.00	0.00 f
...		
sub_41_I64/DIFF[59] (Restore_DW01_sub_0)	0.00	323.75 f
Remainder_reg[59]/D (FD2S)	0.00	323.75 f
data arrival time		323.75
clock clk (rise edge)	325.00	325.00
clock network delay (ideal)	0.00	325.00

Figure2. 2

Remainder_reg[59]/CP (FD2S)	0.00	325.00 r
library setup time	-1.25	323.75
data required time		323.75
<hr/>		
data required time		323.75
data arrival time		-323.75
<hr/>		
slack (MET)		0.00

1

Figure2. 3

This report analysed three algorithms for multiplication and division for the best performance. The criteria to judge the performance was based on the amount of time it took for the algorithm to compute one result and the amount of area required to implement the algorithm in hardware. The multiplication algorithms that were studied include Pen and Paper algorithm, Booth's algorithm, and Divide and Conquer. The division algorithm that was analysed include Radix-2 Restoring algorithm, Radix-2 Non-Restoring algorithm, and Radix-4 Restoring algorithm. After thorough analyses of timing and area reports, Divide and Conquer far exceeded the performance when compared to other multiplication algorithm. In division algorithm comparisons, Radix-4 Restoring algorithm shows the best performance if large area is not a concern. If area needs to be minimized, Radix-2 Restoring algorithm seems to be a good compromise of speed versus area. The algorithms studied in this report can be further optimized to achieve better time and area. Many of these algorithms can be pipelined or run in a multi-cycle configuration. For example, the Pen and Paper would benefit tremendously if it was pipelined. Although, it would not decrease the amount of time it takes to generate one result, it would help increase the throughput of the algorithm. Booth's multiplication can benefit by running it in a multi-cycle configuration instead of running the whole algorithm in one clock cycle. In addition, other algorithms can be investigated for better speed and area. The Radix-4 algorithm can be taken a step further and converted into a 68SRT division algorithm [2]. Another division algorithm that can be investigated is the Newton-Raphson division algorithm, which is currently the fastest division algorithm. Each of the algorithms were simulated and analyzed for time and area constraints. Table 4.1 compares the timing and area requirements for three different multiplication algorithms.

Multiplication	Time	Area
Booth's Algorithm	170.0	145194.0
Divide and Conquer	36.0	127985.0
Pen and Paper	174.0	115135.0

TABLE2. 2

Pen and Paper algorithm takes the most amount of time to compute the multiplication results. Since it is the simplest algorithm out of the three analysed, it takes the least amount of area. However, due to its simplicity, it does the most amount of work which results in the high time to compute the result. The next fastest algorithm is Booth's multiplier. It avoids the addition at every step as in the pen and paper algorithm by using shifters, which results in a slight increase in speed, but the extra addition and subtraction logic needed require a larger area. Divide and conquer provides the best speed out of the three algorithms with its ability to perform several computations in parallel. Due to the parallel nature of the algorithm, it requires duplication of hardware that results in a large area size as well. Nonetheless, the timing benefits of Divide and Conquer algorithm outweigh the area disadvantages that come with the algorithm. Table 2.3 compares the time and area requirements for the three division algorithms.

Division	Time	Area
Radix-2 Restoring Division Algorithm	325.0	133688.0
Radix-4 Restoring Division Algorithm	210.0	181980.0
Radix-2 Non-Restoring Division	324.0	164602.0

TABLE2. 3

As seen in the Table 2.3, Radix-4 restoring divide algorithm provides the best performance in terms of time, however it also requires the most amount of area. Radix-2 restoring divide and Radix-2 non-restoring divide have similar timing requirement, which is expected since non-restoring divide is used

to avoid timing issues that can occur in restore divide and not to increase performance. The area requirement of Radix-2 non-restoring divide is larger than the Radix-2 restoring divide because non-restoring divide

algorithm requires an adder and a subtractor, which adds more hardware. Radix-4 division requires the most area because multiple test subtractions are implemented during each iteration and therefore the algorithm requires multiple subtraction units. Multiple comparisons take place to determine the best quotient value. However, it provides the optimal speed because it can compute 2 bits in one iteration, therefore, reducing the number of iterations used to compute the result. Since Radix-4 restoring algorithm

requires a large amount of area, Radix-4 algorithm would have the best performance if area is not a concern. If area needs to be minimized, then Radix-2 restoring division algorithm would be considered the best performance.

2.2 Novel Low power and high-speed array divider in 65 nm Technology

The power consumed in high performance microprocessors has increased to levels that impose a fundamental limitation to increasing performance and functionality [1]– [3]. If the current trend in increasing power continues, high performance microprocessors will soon consume thousands of watts. The power density of a high-performance microprocessor will exceed the power density levels encountered in typical rocket nozzles within the next decade [2]. The generation, distribution, and dissipation of power are at the forefront of current problems faced by the integrated circuit industry [1]– [5]. The application of aggressive circuit design techniques which only focus on enhancing circuit speed without considering power is no longer an acceptable approach in most high complexity digital systems. Dynamic switching power, the dominant component of the total power consumed in current CMOS technologies, is quadratically reduced by lowering the supply voltage. Lowering the supply voltage, however, degrades circuit speed due to reduced transistor currents. Threshold voltages are scaled to reduce the degradation in speed caused by supply voltage scaling while maintaining the dynamic power consumption within acceptable levels [1]– [5]. At reduced threshold voltages, however, subthreshold leakage currents increase exponentially. Energy efficient circuit techniques aimed at lowering leakage currents are, therefore, highly desirable. Domino logic circuit techniques are extensively applied in high performance microprocessors due to the superior speed and area

characteristics of domino CMOS circuits as compared to static CMOS circuits [7]– [8]. However, deep sub micrometre (DSM) domino logic circuits utilizing low power supply and threshold voltages have decreased noise margins [9]–[11]. As on-chip noise becomes more severe with technology scaling and increasing operating frequencies, error free operation of domino logic circuits has become a major challenge [9], [10], [11]. The focus of this paper is to implement various Reduced-swing domino logic circuit techniques which offer better speed, energy-efficiency and noise immunity in DSM technology. The organization of the paper is as follows. A brief review of the sources of power dissipation in CMOS circuits is provided in Section II. In Section III various Reduced-swing techniques in domino logic circuits for power reduction are proposed. In Section IV simulation and implementation results are presented. Finally, conclusions are presented in Section V.

METHOD

In this work, Shannon's and Mixed Shannon's logics have been proposed. The simulation results show that the proposed techniques offer low power, high speed and with high performance than the existing designs CMOS and CPL. Non-Restoring and Restoring array divider circuits have been designed using adder cell and subtractor cell respectively. The proposed Shannon adder cell consists of 16 transistors and mixed Shannon based adder cell consists of 12 transistors compared to CPL-28 transistors and CMOS-28 transistors. The two different (Non-Restoring and Restoring array) 7x4 bit divider circuits have been simulated by using Microwind 3.1 VLSI CAD tool. Various parameters such as propagation delay, power dissipation, PDP have been determined from array dividers layout of feature size 65nm technology. The divider circuits have been analyzed using BSIM 4 parameter analyzer. The proposed Shannon adder based non-Restoring array divider circuit has a reduced power dissipation of 82.77%, a reduced propagation delay of 44.12% and a reduced PDP of 90.37% compared with CMOS based array divider circuits due to lower critical path in the proposed adder cell. Similarly, Restoring array divider circuit has a reduced power dissipation of 66.09%, a reduced propagation delay of 28.88% and a reduced PDP of 75.98% compared with CMOS based array divider circuits.

Results and Discussion:

TECHNIQUES		POWER(μ Watts)	DELAY(ns)	POWER DELAY PRODUCT ¹⁵ Watt-Sec) (*10 ⁻
CMOS	65nm	427.25	0.347	148.25
CPL	65nm	589.82	0.412	243.005
MIXED-SHANNON	65nm	143.53	0.150	21.53
SHANNON	65nm	121.28	0.242	29.34

TABLE2. 4optimum value of 8-bit adder

TECHNIQUES		POWER(μ Watts)	DELAY(ns)	POWER DELAY PRODUCT Watt-Sec) (*10 ⁻¹⁵
CMOS	65nm	370.42	0.417	154.46
CPL	65nm	598.2	1.251	748.35
MIXED-SHANNON	65nm	159.61	0.205	32.72
SHANNON	65nm	63.81	0.233	14.868

TABLE2. 5Optimum value of 8-bit subtractor

2.3 TRANSLATION OF DIVISION ALGORITHM INTO VERILOG HDL

INTRODUCTION:

Division is the most complicated of all the elemental operations, whether to implement the algorithm in hardware or software. However, it has been shown that ignoring its implementation can result in significant system performance degradation for many applications [2]. There are number of binary division algorithm such as Multiplicative Algorithm, Approximation Algorithms, CORDIC Algorithm and Continued Product Algorithm.

In this paper, the Verilog HDL code for non-restoring algorithm is proposed. However, the size of bit is limited to 16 bit value for the input dividend and divisor. The non-restoring division gives the exact value of the quotient and remainder, besides the implementation required less hardware since the calculation only involves shifting process, arithmetic addition and subtraction

In recent years, many researchers have proposed different algorithms. They aimed to perform fast division different algorithms. They aimed to perform fast division operation and at the same time enhancing the performance Some of the proposed division algorithms to include

Restoring division:

Restoring division functions on fixed-point fractional numbers and depends on the following assumptions [6], $D < N$ and $0 < N$, $D < 1$.

The quotient digits Q are formed from the digit set $\{0, 1\}$. To begin the operation, the dividend and the divisor is broke into the right half of the $2n$ -bit A register and into the left half of the $2n$ -bit B register respectively. The divisor B is subtracted from the remainder register A . If the result of previous step is negative, set the quotient, $Q_0 = 0$ and restore the old remainder. This is the reason why this method is called restoring division. Else, set Q_0 to 1. In the next step, the divisor is shifted to the right, aligning the divisor with the dividend for the next iteration. Repeat the steps until there is no bit left

SRT division

The name of the SRT division stands for Dura W. Sweeney, James E. Robertson and Keith D. Tocher who proposed a fast algorithm for 2's complement numbers that use the technique of shifting over zeros for division. [7]. The basic algorithm for binary (radix 2) SRT division is initially by inserting dividend and divisor into A and B registers respectively. If register B has k leading zeros, shift all the registers (B and A) positions left k bits. Then, the following steps are repeated n times. If the top three bits of the A register are equal, shift the A registers one position left and set $Q_i = 0$. If the top three bits of the A register are unequal and negatives, shift the A registers one position left, set $Q_i = -1$ and add B to A. Otherwise, shift A one bit left, set $Q_i = 1$ and subtract B. After the steps are repeated by n times and the final remainder is negative, correct the remainders by adding B also correct the quotient by subtracting 1 from Q_i . Shift remainder k bits right.

NON-RESTORING DIVISION ALGORITHM

The non-restoring algorithm comes from restoring division and it calculates the remainder by successively subtracting the shifted divisor from the dividend until the remainder is in the appropriate range. The method Assume that we have dividend, D and divisor, X as an input data, quotient, Q as division result and R as remainder. The steps of the non-restoring algorithm are calculated as visible in Figure.

D=1010101 (85), X=0110 (6),

0 0 0 0 1 0 1 0 1 0 1 (D=85)	
0 1 1 0	D-X (step 1)
<u>1 0 1 0 1</u>	Negative, Q0=0
0 1 1 0	shift right X, ADD
<u>1 0 1 1 0</u>	Negative, Q1=0
0 1 1 0	shift right X, ADD
<u>1 1 0 0 1</u>	Negative, Q2=0
0 1 1 0	shift right X, ADD
<u>1 1 1 1 0</u>	Negative, Q3=0
0 1 1 0	shift right X, ADD
0 1 0 0 1	Positive, Q4=1
0 1 1 0	shift right X, SUBTRACT
0 0 1 1 0	Positive, Q5=1
0 1 1 0	shift right X, SUBTRACT
0 0 0 0 1	Positive, Q6=1
0 1 1 0	shift right X, SUBTRACT
<u>1 0 1 1</u>	Negative, Q7=0
0 1 1 0	ADD
<u>R= 0 0 0 1</u>	

The result is Q=00001110 (14), R=0001 (1)

Figure2. 4

As illustrated in Figure, the process starts by subtracting from the most significant bit of dividend with divisor. After making the subtraction process, bring down the next MSB of dividend and attached to the results from the first step. These steps are repeated until all the bits of dividend are calculated as well as the bits of quotient are determined. From the algorithm, it can be concluded that if the result of subtraction is negative, 0 is selected as the quotient Q. On the other hand, if the result of subtraction is

positive which gives 0 as a different, quotient, Q is selected as 1. In summary, Table-1 provides the steps to calculate the binary number for non-restoring division algorithm.

PROPOSED VERILOG HDL

This paper proposed a Verilog HDL coding of non-restoring division algorithm as shown in Figure-1. Theclk is the input clock signal, means that the process is begin to calculate the division operation in the first clock cycle and signal is ready when the iteration is done. However, the size of bit is limited to 16-bit value for the input dividend and divisor. In this implementation, the divisor is shifted to right by one bit until there is no bit left.

In this paper, the Verilog HDL codes for division are generated and simulate using Xilinx ISE 14.4. The Verilog HDL code is broken down into modules which deal with the division of 16 bit dividend and 16 bit divisor. We chose the non-restoring algorithm because it is simple to implement since it requires basic adder or subtractor operation and the shifting process, which is to the left or right in each stage of calculation. Hence, it does not involve other hardware components such as multipliers and multiplexors. The top module connects all of the inputs and produces output as shown in Figure2.5

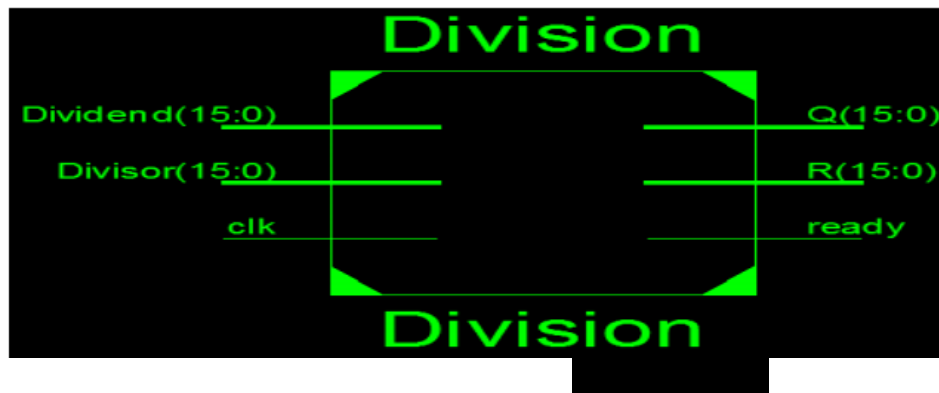


Figure2. 5

```

1  module Division(clk,ready,Dividend,Divisor,Q,R);
2
3      input      clk;
4      input [15:0] Dividend;
5      input [15:0] Divisor;
6
7      output[15:0] Q;
8      output[15:0] R;
9      output      ready;
10
11     reg [15:0]    Q;
12     reg [31:0]    Dividend_copy, Divisor_copy, diff;
13
14     wire [15:0]    remainder = Dividend_copy[15:0];
15
16     reg [31:0]     bit;
17
18     wire          ready = !bit;
19
20     initial bit = 0;
21
22     always @( posedge clk )
23
24         if( ready )
25             begin
26                 bit = 16;
27                 Q = 0;
28                 Dividend_copy = {16'd0,Dividend}; //Output Dividend_copy=31 bits
29                 Divisor_copy = {1'b0,Divisor,15'd0}; //Output Divisor_copy=31 bits
30             end
31         else
32             begin
33                 diff = Dividend_copy - Divisor_copy; //Difference is negative:
34                                                         //copy dividend and put 0 i
35                 Q = Q << 1; //Shift 1 bit
36
37                 if( !diff[31] )
38                     begin
39                         Dividend_copy = diff;
40                         Q[0] = 1'd1;
41                     end
42
43                 Divisor_copy = Divisor_copy >> 1;
44                 bit = bit - 1; //Check for count,
45                 //when bit = bit-1 then stop the loop
46
47             end
48
49     endmodule

```

Figure2. 6

2.4 RESULTS AND DISCUSSIONS:

Simulation

The simulation result of 16 bit fixed point division algorithm is shown in Figure-3. It is clearly shown that the system needs 17 clock cycles, so that the output for 16-bit input is in the ready state. READY state here means that the enumeration stage is completed. We consider the division of 16 bits two fixed point numbers, which are 32768 (1000000000000000) and 158 (0000000010011110) fed as the input dividend and divisor respectively. The desired output of quotient = 207 (0000000011001111) and remainder 62 (0000000000111110) is shown in Figure-4. This proved that the calculation executed by the Verilog HDL code is valid. The simulation performed the calculation through the iteration per bit value until all the bits for quotient and remainder is determined.

Synthesis

Verilog HDL Code for division is then synthesized using the XC6VLX240T device with the package of Vertex 6 FPGA family implementation and the implementation clock frequency is 10 MHz. The characteristic of the device [10] is listed in Table-2.6

Device	CLBs arrays (Total slices)	Maximum I/O
XC6VLX240	37,680	720

TABLE2. 6

The translation of the division algorithm is quite simple and the implementation is done using theXilinx Vertex-6. The FPGA implementation's result for the 16-bitnon-Restoring divider is shown in Table-.2.6

No. of slices No. of LUTs No. of bonded IOBs	No. of slices No. of LUTs No. of bonded IOBs	No. of slices No. of LUTs No. of bonded IOBs
95	166	160

TABLE2. 7FPGA Implementation for 16-bit Divider.

Computer Arithmetic and Verilog HDL Fundamentals

There are two operands in division; the dividend A and the divisor B that yield a quotient Q and a remainder R , as shown below. The remainder is smaller than the divisor and has the same sign as the dividend. Unlike multiplication, division is not always commutative; that is, $A/B \neq B/A$, except when $A = B$. Dividend ($2n$ bits) = Quotient (n bits), Remainder (n bits) Divisor (n bits) Division can be considered as the inverse of multiplication in which the dividend, divisor, and quotient correspond to the product, multiplicand, and multiplier, respectively. Division employs the same general principles as multiplication.

Multiplication is an add-shift operation, whereas division is a shift-subtract/add operation. In division, the result of each subtraction determines the next operation in the division sequence. There can be no overflow in multiplication because the product can never exceed $2n$ bits. In division, however, the dividend may be so large compared to the divisor that the value of the quotient exceeds n bits.

In decimal division, the divisor is compared to the current partial remainder or to the dividend initially to determine a number that is equal to or greater than the divisor. This is inherently a trial-and-error process. Binary division is simpler, because there are only two possible results: 1 or 0. This section will present these sequential shift-subtract/add restoring division and non-restoring division. An example will now be given that shows both decimal and binary division using unsigned operands of the same numerical values.

Figure 2.7 shows two unsigned division examples: Figure (a) displays the decimal example of dividing 327 by 14; Figure (b) displays the binary example of dividing 10100011 (327) by 1110 (14). This paper-and-pencil approach involves repeated shifting and subtraction or addition.

Restoring Division

Let A and B be the dividend and divisor, respectively, where

$$A = a_{2n-1} a_{2n-2} \dots a_n a_{n-1} \dots a_1 a_0$$

$$B = b_{n-1} b_{n-2} \dots b_1 b_0$$

The dividend is a $2n$ -bit positive integer and the divisor is an n -bit positive integer.

The quotient Q and remainder R are n -bit positive integers, where

$$Q = q_{n-1} q_{n-2} \dots q_1 q_0$$

$$R = r_{n-1} r_{n-2} \dots r_1 r_0$$

When division is implemented in hardware, the process is slightly different than previously described. The dividend is initially shifted left 1 bit position. Then the divisor is subtracted from the dividend. Subtraction is accomplished by adding the 2s complement of the divisor. If the carry-out of the subtract operation is 1, then a 1 is placed in the next lower-order bit position of the quotient; if the carry-out is 0, then a 0 is placed in the next lower-order bit position of the quotient. The concatenated partial remainder and dividend are then shifted left 1 bit position. Fixed-point binary restoring division requires one subtraction for each quotient bit. Figure 5.19 shows an example of the hardware algorithm for restoring division. Since there are 4 bits in the divisor, there are four cycles. Each cycle begins with a left shift operation. The low-order quotient bit is left blank after each left shift operation it will be set before the next left shift. Then the divisor is subtracted from the high-order half of the dividend. If the resulting difference is negative — indicated by a carry-out of 0 — then the low-order quotient bit q_0 is set to 0 and the previous partial remainder is restored. If the carry-out is 1, then q_0 is set to 1 and there is no restoration the partial remainder thus obtained is loaded into the high-order half of the dividend. This sequence repeats for all n bits of the divisor. If the sign of the remainder is different than the sign of the dividend, then the previous partial remainder is restored.

Overflow can occur in division if the value of the dividend and the value of the divisor are disproportionate, yielding a quotient that exceeds the range of the machine's word size. Since the dividend has double the number of bits of the divisor, *overflow* can occur when the high-order half of the dividend has a value that is greater than or equal to that of the divisor. If the high-order half of the dividend ($a_{2n-1} \dots a_n$) and the divisor ($b_{n-1} \dots b_0$) are equal, then the quotient will exceed n bits. If the value of the high-order half of the dividend is greater than the value of the divisor, then value of the quotient will be even greater. Overflow can be detected by subtracting the divisor from the high-order

half of the dividend before the first shift-subtract cycle. If the difference is positive, then an overflow has been detected; if the difference is negative, then the condition for overflow has not been met. Division by zero must also be avoided. This can be detected by the preceding method, since any high-order dividend bits will be greater than or equal to a divisor of all zeroes.

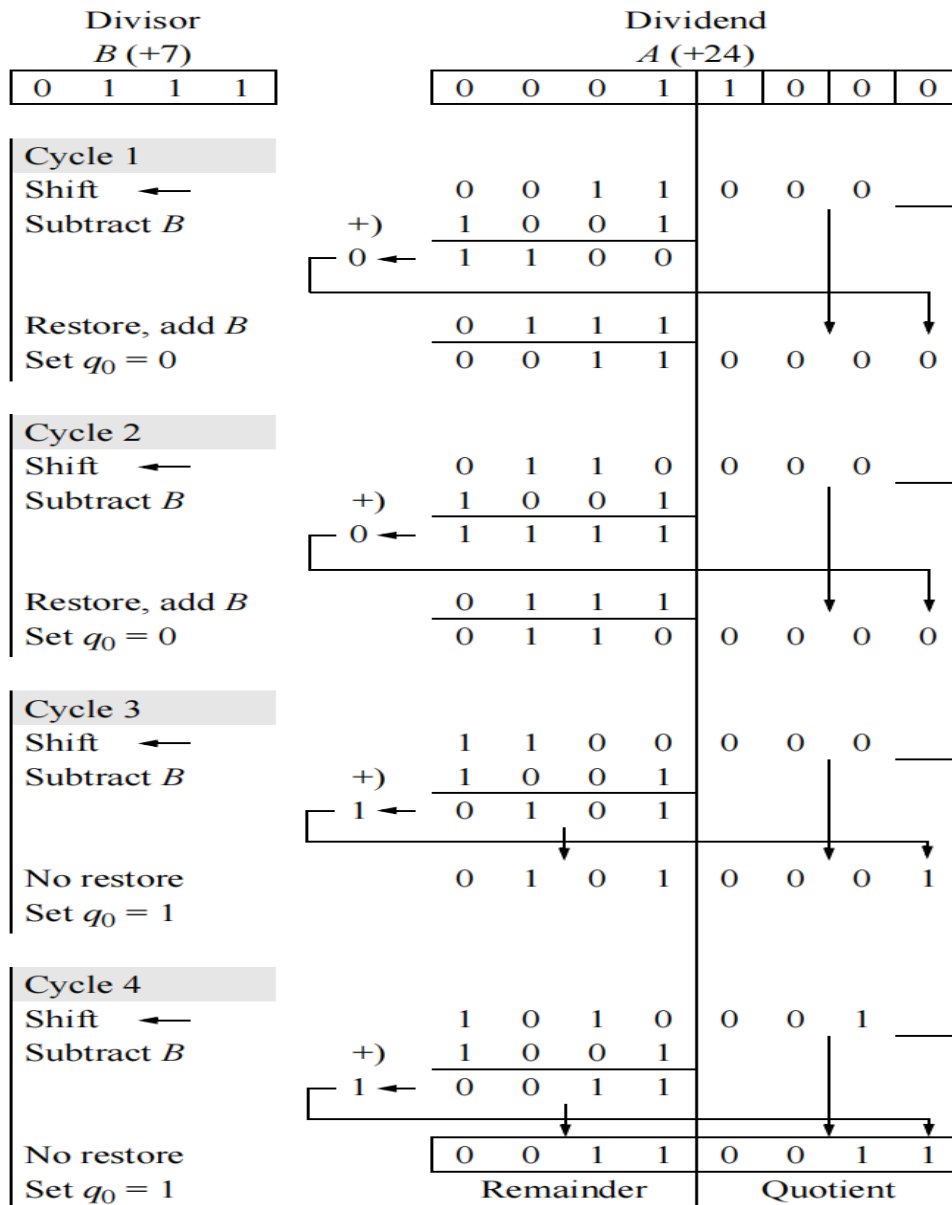


Figure2. 8

Non-restoring Division

The speed of the shift-subtract/add restoring division can be increased by redefining the algorithm. Instead of restoring the partial remainder to the previous partial remainder if the result of a subtraction is negative, the algorithm allows both positive and negative partial remainders to be used. That is, a negative partial remainder is used unchanged in the following cycle. Thus, the absolute value of the partial remainder is reduced every cycle by adding or subtracting the divisor from the partial remainder.

In restoring division, if the partial remainder is positive after a subtract operation, then the dividend is shifted left 1 bit position and the divisor is subtracted; that is, the operation is $2A - B$. If the partial remainder is negative after a subtract operation, then the partial remainder is restored by adding the divisor ($A + B$), then it is shifted left 1 bit position and the divisor B is subtracted. This is equivalent to $2A + B$, as shown below.

$$2(A + B) - B = 2A + B$$

Thus, in non-restoring division, only two operations are required: $2A - B$ and $2A + B$ for each cycle. The value of q_0 can be determined by the carry-out of the addition or subtraction operation as shown below.

The initial left shift of the dividend must be followed by a subtraction of the divisor in order to establish a starting point for the non-restoring procedure. Only the final partial remainder is restored to a positive value if it is negative. Thus, if the final value for q_0 is 0, then the previous partial remainder must be restored in order to obtain the correct remainder. Therefore, in non-restoring division, there are n or $n + 1$ shift-add/ subtract cycles. Figure 5.20 shows an example of the hardware algorithm for non-restoring division. As in the previous restoring division example, subtraction is done by adding the 2s complement of the divisor. In cycle 2, the operation is $2A + B$, because the previous partial remainder was negative. There is a final restore cycle, because the sign of the remainder is different than the sign of the quotient.

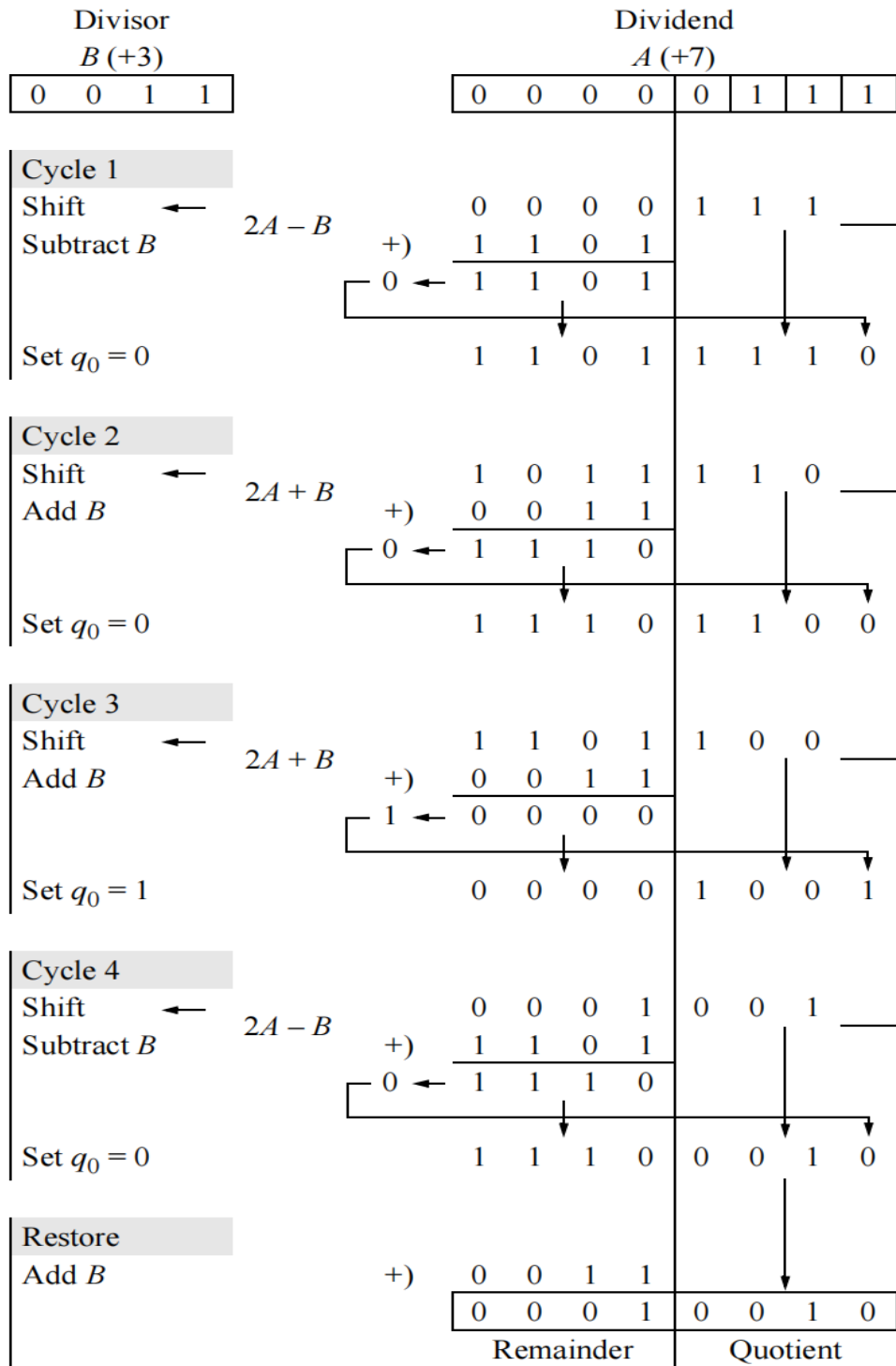


Figure2. 9

CHAPTER 3

DESIGN METHODOLOGY AND REQUIRED TOOLS FOR IMPLEMENTATION OF OUR PROPOSED DESIGN

3.1 INTRODUCTION

The design methodology for the implementation and physical design of 8/4 signed divider required algorithm which perform the division operation and produced result in signed bit then from using this proposed algorithm we design the Verilog module for our required design

3.2 THE PROPOSED ALGORITHM FOR SIGNED DIVIDER:

The proposed architecture for the signed binary divider uses the non-restoring divider logic as a baseblock. In this work the size of the dividend is 8-bit and the size of the divisor is 4-bit. The algorithm is demonstrated in Fig. 1. The steps of a block diagram are explained below. In Fig. 1 D_{n-1} to D_0 are bits of Dividend p in this paper it is given as D_7 to D_0 as we designed the divider for an 8-bit dividend

DIVIDEND AND DIVISORS			
Dividend	Divisor	Quotient	Remainder
Positive	Positive	Positive	Positive
Positive	Negative	Negative	Positive
Negative	Positive	Negative	Negative
Negative	Negative	Positive	Negative

Figure 3. 1

- 1) First based on sign bit of dividend and divisor, positive values of dividend and divisor is processed
- 2) These positive values are then going to non-restoring divider logic.
- 3) Quotient and reminder generated from divider logic are still incorrect
- 4) Quotient and the remainder is calculated with the help of sign bits of dividend and divisor. The Table 3.1 for the same is mentioned above.

As an example, take a divisor as $(5)_{10}$ or $(0101)_2$ and dividend as $(26)_{10}$ or $(0001\ 1010)_2$. As described in the algorithm. (Note: initial value of quotient is zero).

1) First, the value of $dividend(p)$ and $divisor(p)$ is selected based on dividend and divisor sign bit. In this example, both the values are positive, hence there will be no change in the dividend and divisor's value. Moreover, the sign bit is stored as 0 for both.

2) Now, subtract the top four MSB from the divisor. Which is $0001 - 0101$.

3) The subtraction result is negative. Now the quotient is left-shifted by 1, so there will be no change in the quotient. The count is decreased by 1.

4) Now the dividend is left-shifted by 1, therefore the new dividend is $0011\ 0100$.

5) The same procedure is repeated. As the top 4 MSB bits of dividend (0011) is less than the divisor, we can shift the quotient, and dividend by 1. count is also decreased by 1.

6) The new dividend is $0110\ 1000$. Now the subtraction of dividend top 4 MSB and divisor is positive. So the quotient will be updated to 0001 .

7) Now, the top 4 MSB of dividend is replaced by subtraction result, which is 0001 . Hence the updated dividend is $0001\ 1000$. count is also decreased by 1, and the count is not zero, so left shift the dividend.

8) The new dividend is $0011\ 0000$. As the top 4 MSB bits of dividend (0011) is less than the divisor, therefore, we can left shift the quotient and dividend by 1. count is also decreased by one, and dividend is left-shifted by 1.

9) The new dividend is $0110\ 0000$. Now the subtraction of dividend top 4 MSB and divisor is positive. So, the quotient will be updated to 0101 .

10) Now, the top 4 MSB of dividend is replaced by subtraction result, which is 0001 . Hence the updated dividend is $0001\ 0000$. count is also decreased by 1, and the count is zero now.

11) now results selected based on the value of sign bits of Dividend and divisor. As both Dividend and divisor are positive, the resultant quotient is 0101 , and the remainder is 0001 .

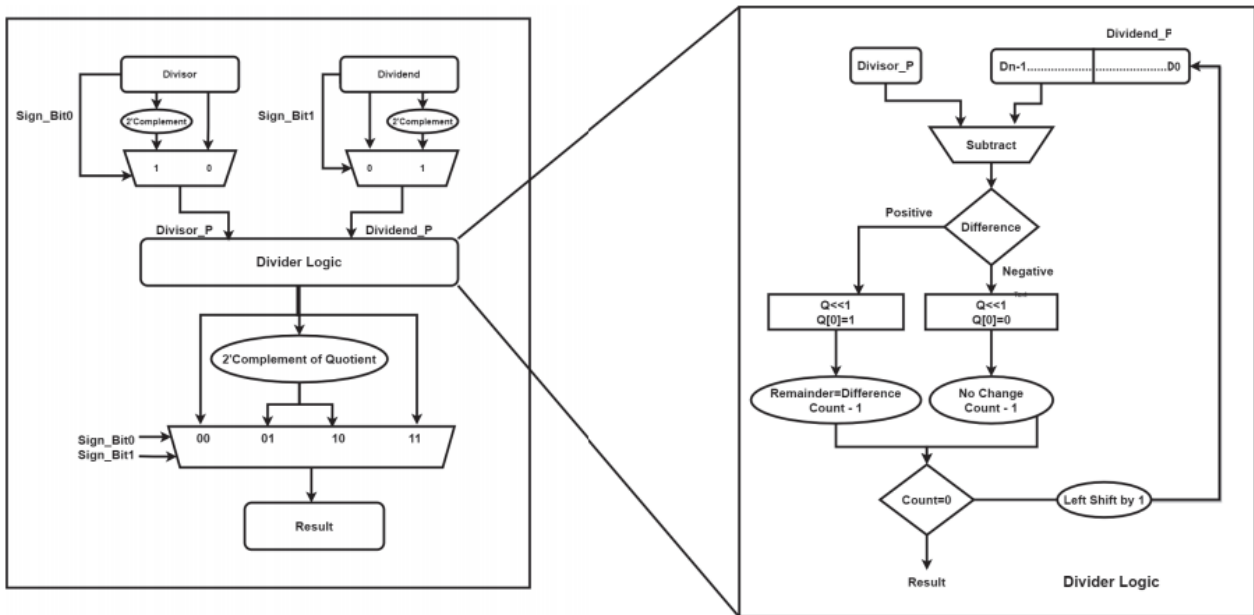


FIGURE 3. 2

Flow chart of proposed algorithm

3.3 OPENSOURCE EDA TOOLS REQUIRED FOR IMPLEMENTATION OF REQUIRED DESIGN MODULE ARE:

I-VERILOG FOR SIMULATION:

In simulation we are verifying the design Verilog code for proposed signed divider is correct or not according to the function term of the logic gate for this we using I verillog(opensource eda) through visual studio code which act as IDE(interactive development environment) to provide command to I verillog tool.

GTKWAVE:It is used to show as the output wave of simulation and other graph form

YOSYS(For synthesis):Yosys is use for synthesis of the Verilog code to map the logical gate of the design to the technical liberty cells.

For backend design:

We have used ubuntu version 25.0 which is operating system run through virtual box machine in ubuntu we had download git which support open lane tool for performing backend designing like time level simulation, synthesis for getting chip area, power and STA report then we perform floorplan, placement, routing, magic and k-layout

COMMANDS USED FOR: -

FRONT END:-

1)SIMULATION:-

1. iverilog "module file name(.v)"
2. vvp "tab a" for a.out file

2) SYNTHESIS:-

1. yosys >invoke yosys
2. Read_liberty sky130_fd_sc_hd__tt_025C_1v80.lib
3. Read_ Verilog "module name"
4. synth -top "module name"
5. ABC -liberty sky130_fd_sc_hd__tt_025C_1v80.lib
6. show > show file
7. write Verilog -noattr netlist. v > netlist file
8. exit > yosys exit
9. iverilog netlist sky130_fd_sc_hd.vtestbench

3)GTKWAVE:-

1. gtkwave "module name.vcd"

4)COMMAND FOR FRONTEND & BACKEND IN UBUNTU:-

1. ./flow.tcl -interactive
2. Package required open lane
3. Prep -design -overwrite "design name" -tag run1

BACK END:-

1) SYNTHESIS COMMANDS:-

- 1.run synthesis
2. run_yosys
3. run_synth_exploration

2) FLOORPLAN COMMANDS:-

- 1.run_floorplan
- 2.chip_floorplan
- 3.init_floorplan

3) PLACEMENT COMMANDS:-

- 1.global_placement_or
2. global_placement
3. random_global_placement
4. detailed_placement

4) PDN GENERATION COMMANDS:-

- 1.gen_pdn

5) ROUTING COMMANDS:-

- 1.global_routing
2. detailed_routing
3. run_routing

6) MAGIC COMMANDS:-

- 1.run_magic
2. run_magic_drc

7) KLAYOUT COMMANDS:-

- 1.run_klayout
- 2.run_klayout_drc

3.4 OUR PROPOSED VERILOG CODE FOR REQUIREDMODULE:

Verilog Module:

```
module div1 (opnds, quot, rem, ovfl);  
input [11:0] opnds; //dividend 8 bits; divisor 4 bits  
output [3:0] quot, rem;  
output ovfl;  
wire [11:0] opnds;  
reg [3:0] quot, rem;  
reg ovfl;  
//check for overflow  
always @ (opnds)  
begin  
if (opnds[11:8] >= opnds[3:0])  
ovfl = 1'b1;  
else  
ovfl = 1'b0;  
end  
always @ (opnds)  
begin  
case (opnds)
```

```

//dvdnd = +6;dvsr = +1;quot = 6;rem = 0
12'b000001100001 :begin
quot= 4'b0110;
rem = 4'b0000;
end
//dvdnd = +7;dvsr = +2;quot = 3;rem = 1
12'b000001110010 :begin
quot= 4'b0011;
rem = 4'b0001;
end
//dvdnd = +17;dvsr = +3;quot = 5;rem = 2
12'b000100010011 :begin
quot= 4'b0101;
rem = 4'b0010;
end
//dvdnd = +41;dvsr = +3;quot = 13;rem = 2
12'b001010010011 :begin
quot= 4'b1101;
rem = 4'b0010;
end
//dvdnd = +51;dvsr = +8;quot = 6;rem = 3
12'b001100111000 :begin
quot= 4'b0110;
rem = 4'b0011;
end
//dvdnd = +72;dvsr = +5;quot = 14;rem = 2
12'b010010000101 :begin
quot= 4'b1110;
rem = 4'b0010;
end
//dvdnd = +76;dvsr = +6;quot = 12;rem = 4

```

```

12'b010011000110 :begin
quot= 4'b1100;
rem = 4'b0100;
end
//dvdnd = +110;dvsr = +7;quot = 15;rem = 5
12'b011011100111 :begin
quot= 4'b1111;
rem = 4'b0101;
end
//dvdnd = +97;dvsr = +5;quot = 19;rem = 2
//overflow occurs
12'b011000010101 :begin
quot= 4'bxxxx;
rem = 4'bxxxx;
end
//dvdnd = +70;dvsr = +4;quot = 17;rem = 2
//overflow occurs
12'b010001100100 :begin
quot= 4'bxxxx;
rem = 4'bxxxx;
end
default :begin
quot= 4'b0000;
rem = 4'b0000;
end
endcase
end
endmodule

```

Testbench:

```

module div1_tb;

```

```

reg [11:0] opnds;
wire [3:0] quot, rem;
wire ovfl;
//display variables
initial
$monitor ("opnds= %b, quot = %b, rem = %b, ovfl = %b", opnds, quot, rem, ovfl);
//apply stimulus
initial
begin
//dvdnd = +6;dvsr = +1;quot = 6;rem = 0
#0 opnds = 12'b0000001100001;
//dvdnd = +7;dvsr = +2;quot = 3;rem = 1
#10 opnds = 12'b0000001110010;
//dvdnd = +17;dvsr = +3;quot = 5;rem = 2
#10 opnds = 12'b000100010011;
//dvdnd = +41;dvsr = +3;quot = 13;rem = 2
#10 opnds = 12'b001010010011;
//dvdnd = +51;dvsr = +8;quot = 6;rem = 3
#10 opnds = 12'b001100111000;
//dvdnd = +72;dvsr = +5;quot = 14;rem = 2
#10 opnds = 12'b010010000101;
//dvdnd = +76;dvsr = +6;quot = 12;rem = 4
#10 opnds = 12'b010011000110;
//dvdnd = +110;dvsr = +7;quot = 15;rem = 5
#10 opnds = 12'b011011100111;
//dvdnd = +97;dvsr = +5;quot = 19;rem = 2
//overflow occurs
#10 opnds = 12'b011000010101;
//dvdnd = +70;dvsr = +4;quot = 17;rem = 2
//overflow occurs
#10 opnds = 12'b010001100100;

```

```

#10 $stop;
end

//instantiate the module into the test bench
srt_div_case2 inst1 (
.opnds(opnds),
.quot(quot),
.rem(rem),
.ovfl(ovfl));
Endmodule

```

3.5 RESULTS OF PROPOSED VERILOG MODULE AND TESTBENCH:

Simulation result:

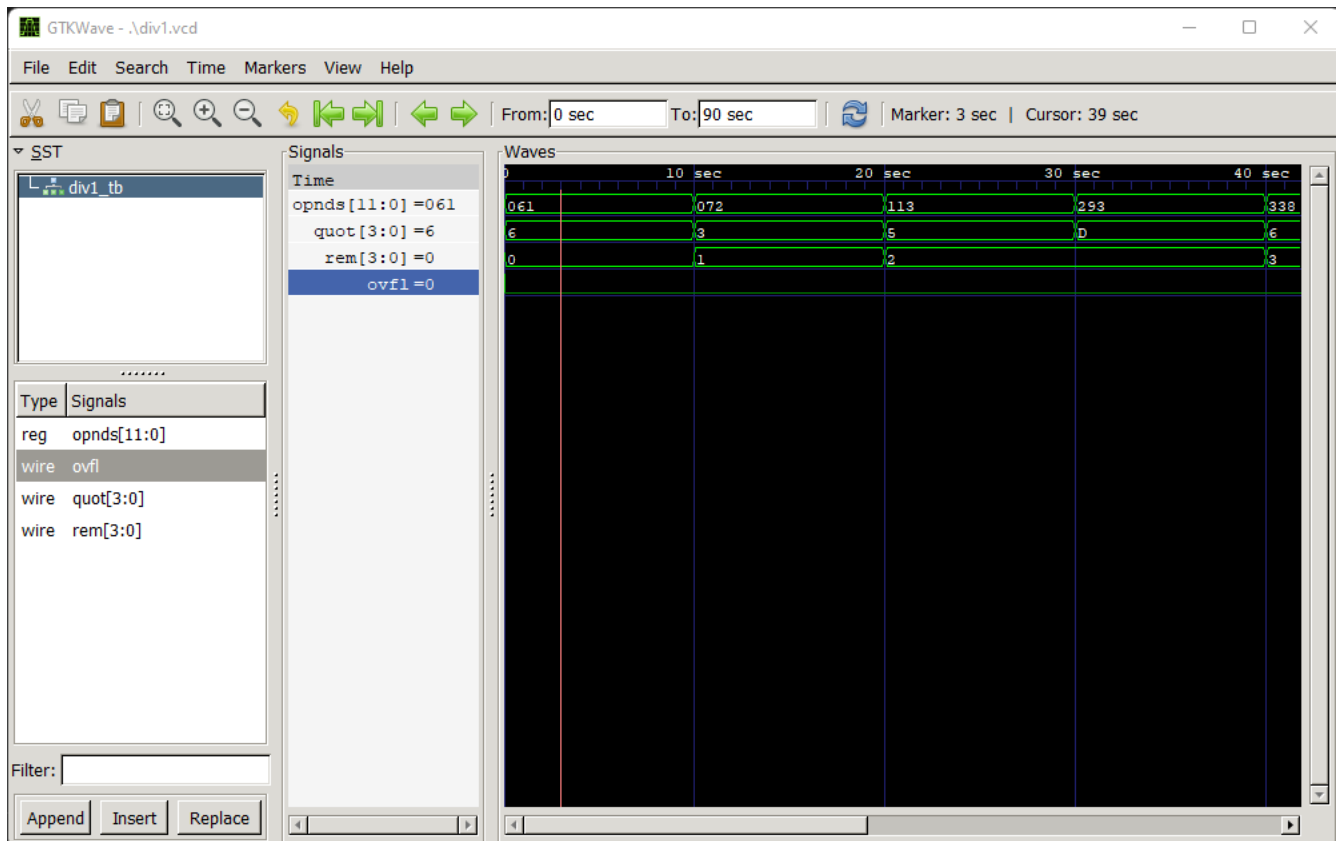


FIGURE 3. 3

SYNTHESIS: Synthesis is a process of converting the Verilog code into a logic gate-level netlist. The synthesis tool requires two input files. First is the technology library file(.lib), which contains standardcells. The second file is a constraint file(.sdc), including timing, loading, and optimization algorithm for logic optimization. Open-source tool yosys is used for generating synthesizable Verilog code which is necessary for timing analysis in the back end flow.

Synthesis using yosys:

```
yosys> read_verilog divph.v
1. Executing Verilog-2005 frontend: divph.v
Parsing Verilog input from `divph.v' to AST representation.
Generating RTLIL representation for module `div1'.
Note: Assuming pure combinatorial block at divph.v:12 in
compliance with IEC 62142(E):2005 / IEEE Std. 1364.1(E):2002. Recommending
use of @* instead of @(...) for better match of synthesis and simulation.
divph.v:14: Warning: Range [11:4] select out of bounds on signal `dividend': Setting 4 MSB bits to undef.
Note: Assuming pure combinatorial block at divph.v:19 in
compliance with IEC 62142(E):2005 / IEEE Std. 1364.1(E):2002. Recommending
use of @* instead of @(...) for better match of synthesis and simulation.
Successfully finished Verilog frontend.
```

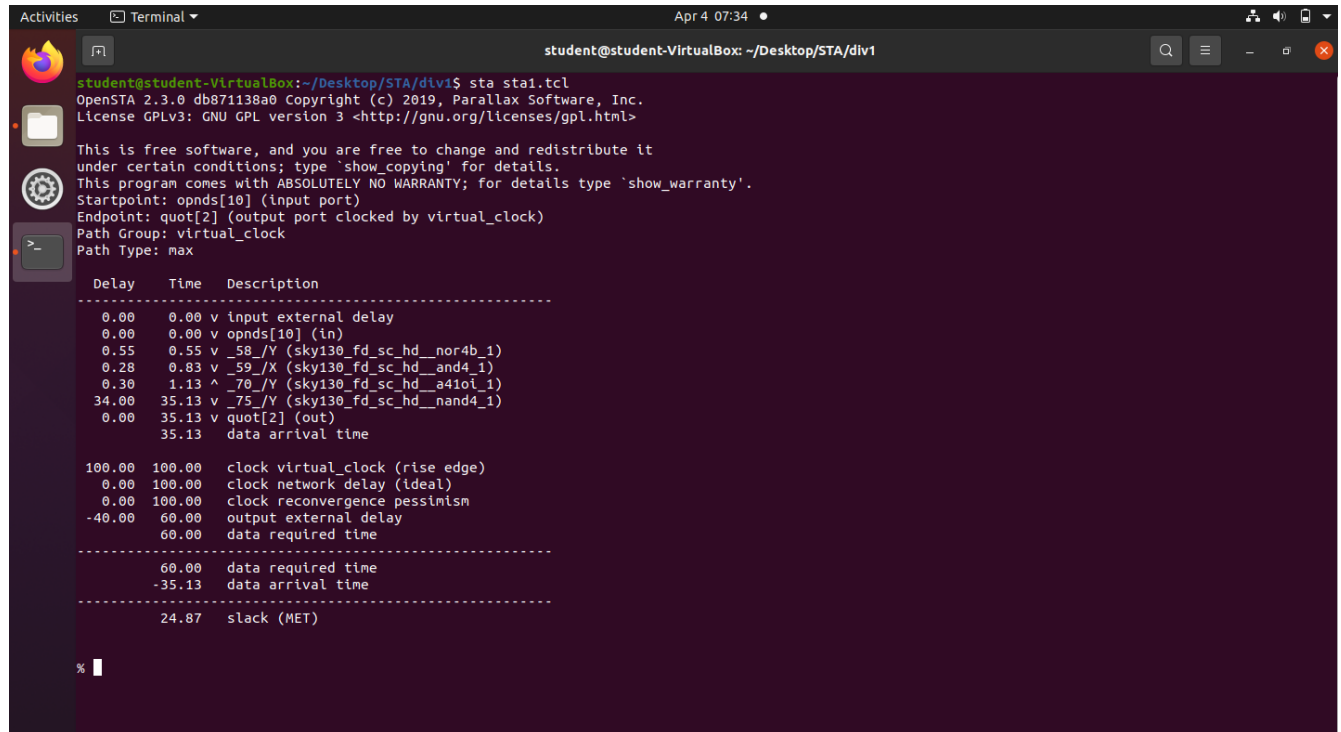
FIGURE 3. 4

9. Printing statistics.

```
=== div1 ===  
  
Number of wires:                141  
Number of wire bits:            158  
Number of public wires:         4  
Number of public wire bits:     21  
Number of memories:             0  
Number of memory bits:          0  
Number of processes:            0  
Number of cells:                43  
  sky130_fd_sc_hd__a31oi_1      1  
  sky130_fd_sc_hd__a41oi_1      2  
  sky130_fd_sc_hd__and3b_1       1  
  sky130_fd_sc_hd__clkinv_1      4  
  sky130_fd_sc_hd__lpflow_isobufsrc_1  3  
  sky130_fd_sc_hd__maj3_1        3  
  sky130_fd_sc_hd__nand2_1       4  
  sky130_fd_sc_hd__nand2b_1      5  
  sky130_fd_sc_hd__nand3_1       2  
  sky130_fd_sc_hd__nand3b_1      1  
  sky130_fd_sc_hd__nand4_1       1  
  sky130_fd_sc_hd__nand4b_1      3  
  sky130_fd_sc_hd__nand4bb_1     1  
  sky130_fd_sc_hd__nor2_1        2  
  sky130_fd_sc_hd__nor3_1        1  
  sky130_fd_sc_hd__nor4_1        3  
  sky130_fd_sc_hd__nor4b_1       1  
  sky130_fd_sc_hd__o211ai_1      1  
  sky130_fd_sc_hd__o311ai_0      1  
  sky130_fd_sc_hd__or4_1         2  
  sky130_fd_sc_hd__or4b_1        1  
  
Chip area for module '\div1': 285.273600
```

FIGURE 3. 5

STATIC TIMING REPORT:-



```
student@student-VirtualBox: ~/Desktop/STA/div1
student@student-VirtualBox:~/Desktop/STA/div1$ sta sta1.tcl
OpenSTA 2.3.0 db871138a0 Copyright (c) 2019, Parallax Software, Inc.
License GPLv3: GNU GPL version 3 <http://gnu.org/licenses/gpl.html>

This is free software, and you are free to change and redistribute it
under certain conditions; type 'show_copying' for details.
This program comes with ABSOLUTELY NO WARRANTY; for details type 'show_warranty'.
Startpoint: opnds[10] (input port)
Endpoint: quot[2] (output port clocked by virtual_clock)
Path Group: virtual_clock
Path Type: max

-----
Delay    Time    Description
-----
0.00     0.00    v input external delay
0.00     0.00    v opnds[10] (in)
0.55     0.55    v _58_/Y (sky130_fd_sc_hd__nor4b_1)
0.28     0.83    v _59_/X (sky130_fd_sc_hd__and4_1)
0.30     1.13    ^ _70_/Y (sky130_fd_sc_hd__a4ioi_1)
34.00    35.13    v _75_/Y (sky130_fd_sc_hd__nand4_1)
0.00     35.13    v quot[2] (out)
0.00     35.13    data arrival time

-----
100.00   100.00   clock virtual_clock (rise edge)
0.00     100.00   clock network delay (ideal)
0.00     100.00   clock reconvergence pessimism
-40.00    60.00   output external delay
0.00     60.00   data required time

-----
0.00     60.00   data required time
-35.13    24.87   data arrival time
-----
24.87    24.87   slack (MET)

% █
```

FIGURE 3. 6

This is the approximate chip area 440 for required design module which show the primitive logic gate (wires and reg) mapped with technology nodes in the liberty file. The chip area is not the core area where floorplan, placement and routing take place.

BACKEND:

POWER REPORT IN BACKEND SYNTHESIS:


```
report_power
```

Group	Internal Power	Switching Power	Leakage Power	Total Power	
Sequential	0.00e+00	0.00e+00	0.00e+00	0.00e+00	0.0%
Combinational	4.03e-06	6.85e-06	2.32e-10	1.09e-05	100.0%
Macro	0.00e+00	0.00e+00	0.00e+00	0.00e+00	0.0%
Pad	0.00e+00	0.00e+00	0.00e+00	0.00e+00	0.0%
Total	4.03e-06	6.85e-06	2.32e-10	1.09e-05	100.0%
	37.1%	62.9%	0.0%		

```
power_report_end
area_report
```

```
report_design_area
```

FIGURE 3. 7

PLACEMENT REPORT IN BACKEND SYNTHESIS:

```
[INFO ODB-0223] Created 13 technology layers
[INFO ODB-0224] Created 25 technology vias
[INFO ODB-0225] Created 441 library cells
[INFO ODB-0226] Finished LEF file: /openlane/designs/div1/runs/try4/tmp/merged_unpadded.lef
[INFO ODB-0127] Reading DEF file: /openlane/designs/div1/runs/try4/tmp/placement/15-resizer.def
[INFO ODB-0128] Design: div1
[INFO ODB-0130] Created 23 pins.
[INFO ODB-0131] Created 111 components and 680 component-terminals.
[INFO ODB-0132] Created 2 special nets and 420 connections.
[INFO ODB-0133] Created 91 nets and 259 connections.
[INFO ODB-0134] Finished DEF file: /openlane/designs/div1/runs/try4/tmp/placement/15-resizer.def
Design name: div1
Core Area Boundaries: 5520 10880 34960 38080
Number of instances 111
Placed 79 instances
```

FIGURE 3. 8

Placement: The location of every component on the die can be determined by Placement by considering the timing data and length of interconnects. To optimize the placement step, it is divided into 4 phases: 1. Pre-placement optimization,

2. In placement optimization

3. post-placement optimization (PPO) before clock tree synthesis (CTS)

4. post-placement optimization after CT

ROUTING REPORT IN BACKEND SYNTHESIS:

```
[INFO GRT-0017] Processing 5 blockages on layer met5.

[INFO GRT-0053] Routing resources analysis:
Layer      Routing      Original      Derated      Resource
Direction  Resources    Resources    Reduction (%)
-----
li1         Vertical      525           159          69.71%
net1        Horizontal   700           487          30.43%
net2        Vertical     525           522           0.57%
net3        Horizontal   350           292          16.57%
net4        Vertical     210           208           0.95%
net5        Horizontal    70            52          25.71%
-----

[INFO GRT-0191] Wirelength: 310, Wirelength1: 0
[INFO GRT-0192] Number of segments: 156
[INFO GRT-0193] Number of shifts: 0
[INFO GRT-0097] First L Route.
[INFO GRT-0191] Wirelength: 310, Wirelength1: 310
[INFO GRT-0192] Number of segments: 156
[INFO GRT-0193] Number of shifts: 0
[INFO GRT-0135] Overflow report.
[INFO GRT-0136] Total hCap           : 831
[INFO GRT-0137] Total vCap           : 889
[INFO GRT-0138] Total usage           : 310
[INFO GRT-0139] Max H overflow        : 0
[INFO GRT-0140] Max V overflow        : 0
[INFO GRT-0141] Max overflow          : 0
[INFO GRT-0142] Number of overflow edges : 0
[INFO GRT-0143] H   overflow          : 0
[INFO GRT-0144] V   overflow          : 0
[INFO GRT-0145] Final overflow        : 0
```

FIGURE 3. 9

The goal of routing is to decide the interconnect paths, including macro pins and standard cells. Routing makes all the connections described in the netlist too effectively without violating setup and holding time constraints.

```

[INFO DRT-0198] Complete detail routing.
Total wire length = 2556 um.
Total wire length on LAYER li1 = 1 um.
Total wire length on LAYER met1 = 983 um.
Total wire length on LAYER met2 = 1072 um.
Total wire length on LAYER met3 = 354 um.
Total wire length on LAYER met4 = 144 um.
Total wire length on LAYER met5 = 0 um.
Total number of vias = 532.
Up-via summary (total 532):.

-----
FR_MASTERSLICE      0
    li1             204
    met1            244
    met2             65
    met3             19
    met4              0
-----
                        532

[INFO DRT-0267] cpu time = 00:00:09, elapsed time = 00:00:18, memory = 112.04 (MB), peak = 450.88 (MB)

```

FIGURE 3. 10

Routing: Routing aims to decide the interconnects paths, which includes macro pins & standard cells. Routing makes all the connections described in the netlist file too effectively without violating setup and holding time constraints. The area taken by the path during routing is 531 micro meter square in our required design

In the figure 3.10 it shows the routing report for various layer in different direction with their reduced area in percentage.

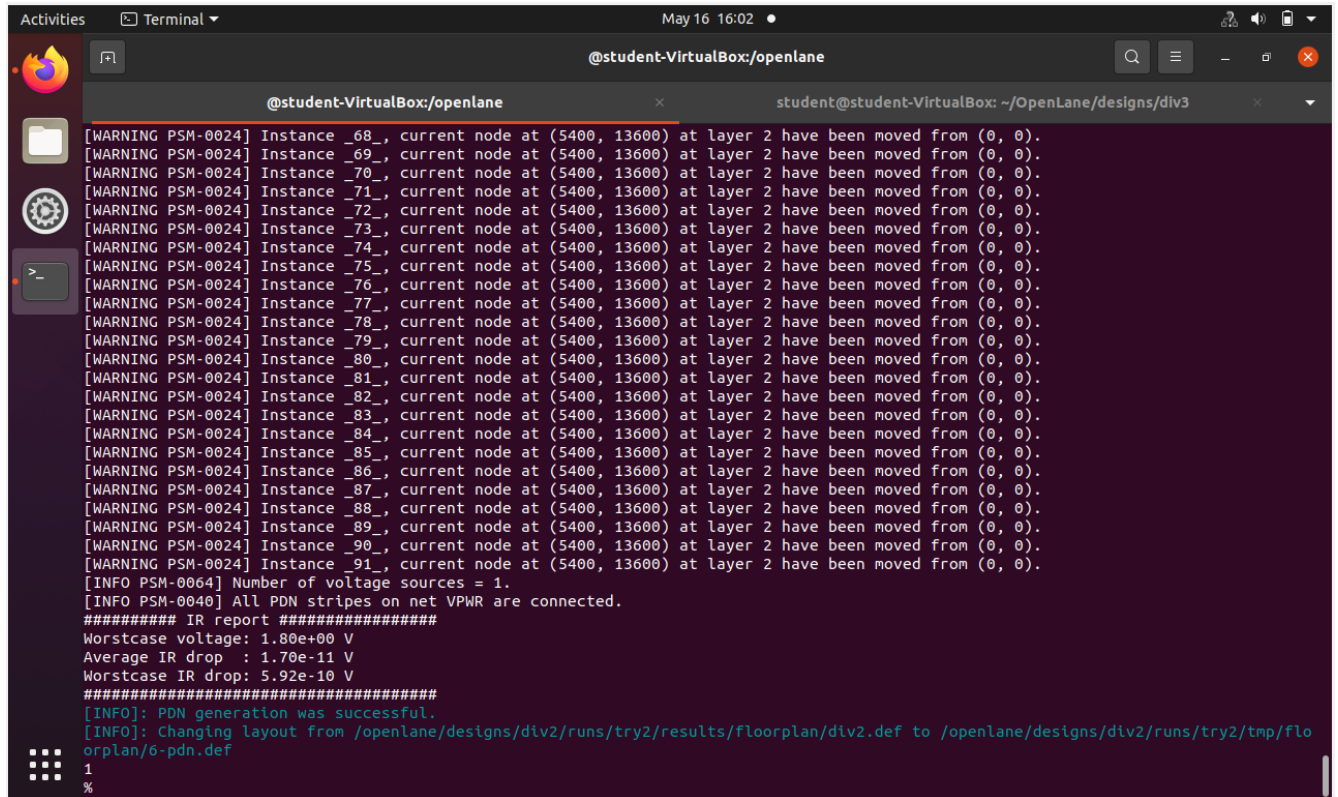
FLOORPLAN RESULT IN BACKEND SYNTHESIS:

```

#####
# Design Rules
#####
set_max_fanout 5.0000 [current_design]
[INFO IFP-0001] Added 6 rows of 37 sites.
[INFO] Extracting DIE_AREA and CORE_AREA from the floorplan
[INFO] Floorplanned on a die area of 0.0 0.0 28.37 39.09 (microns). Saving to /openlane/designs/div3/runs/try4/reports/floorplan/7-initial_fp_die_area.rpt.
[INFO] Floorplanned on a core area of 5.52 10.88 22.54 27.2 (microns). Saving to /openlane/designs/div3/runs/try4/reports/floorplan/7-initial_fp_core_area.rpt.
[INFO]: Core area width: 17.02
[INFO]: Core area height: 16.32
[INFO]: Final Vertical PDN Offset: 2.8366666666666664
[INFO]: Final Horizontal PDN Offset: 2.72
[INFO]: Final Vertical PDN Pitch: 5.673333333333333
[INFO]: Final Horizontal PDN Pitch: 5.44
[INFO]: Changing layout from /openlane/designs/div3/runs/try4/results/floorplan/div3.def to /openlane/designs/div3/runs/try4/tmp/floorplan/7-initial_fp.def
/openlane/designs/div3/runs/try4/tmp/floorplan/7-initial_fp.sdc

```

FIGURE 3. 11



The screenshot shows a terminal window titled "@student-VirtualBox:/openlane" with a search bar and window controls. The terminal output displays a series of warnings for instances 68 through 91, indicating node movements at layer 2. It then shows information about voltage sources and an IR report. The IR report includes worstcase voltage (1.80e+00 V), average IR drop (1.70e-11 V), and worstcase IR drop (5.92e-10 V). The terminal concludes with a successful PDN generation message and a layout change command.

```
@student-VirtualBox:/openlane
x
student@student-VirtualBox: ~/OpenLane/designs/div3
x
v
[WARNING PSM-0024] Instance _68_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _69_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _70_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _71_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _72_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _73_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _74_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _75_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _76_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _77_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _78_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _79_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _80_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _81_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _82_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _83_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _84_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _85_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _86_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _87_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _88_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _89_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _90_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[WARNING PSM-0024] Instance _91_, current node at (5400, 13600) at layer 2 have been moved from (0, 0).
[INFO PSM-0064] Number of voltage sources = 1.
[INFO PSM-0040] All PDN stripes on net VPWR are connected.
##### IR report #####
Worstcase voltage: 1.80e+00 V
Average IR drop : 1.70e-11 V
Worstcase IR drop: 5.92e-10 V
#####
[INFO]: PDN generation was successful.
[INFO]: Changing layout from /openlane/designs/div2/runs/try2/results/floorplan/div2.def to /openlane/designs/div2/runs/try2/tmp/floorplan/6-pdn.def
1
%
```

FIGURE 3. 12

MAGIC AND KLAYOUT RESULTS IN BACKEND SYNTHESIS:

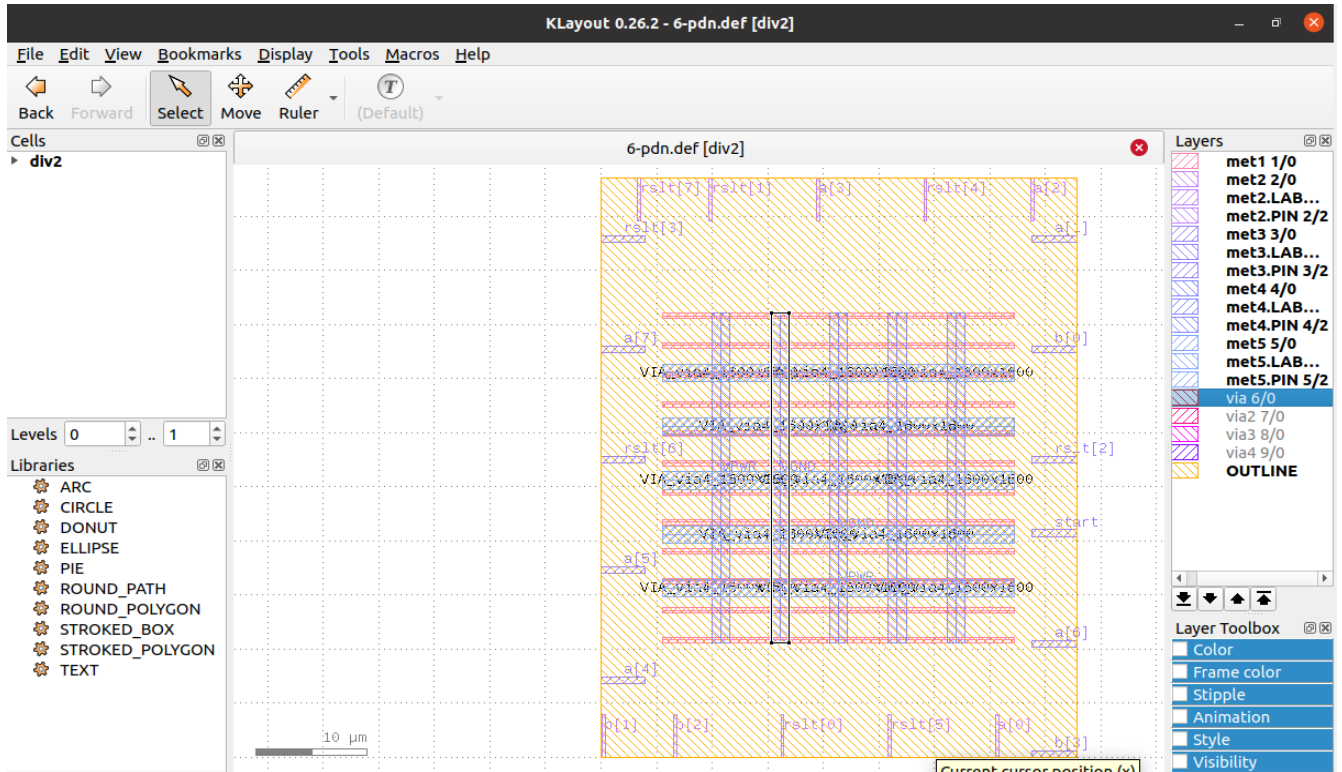


FIGURE 3. 13

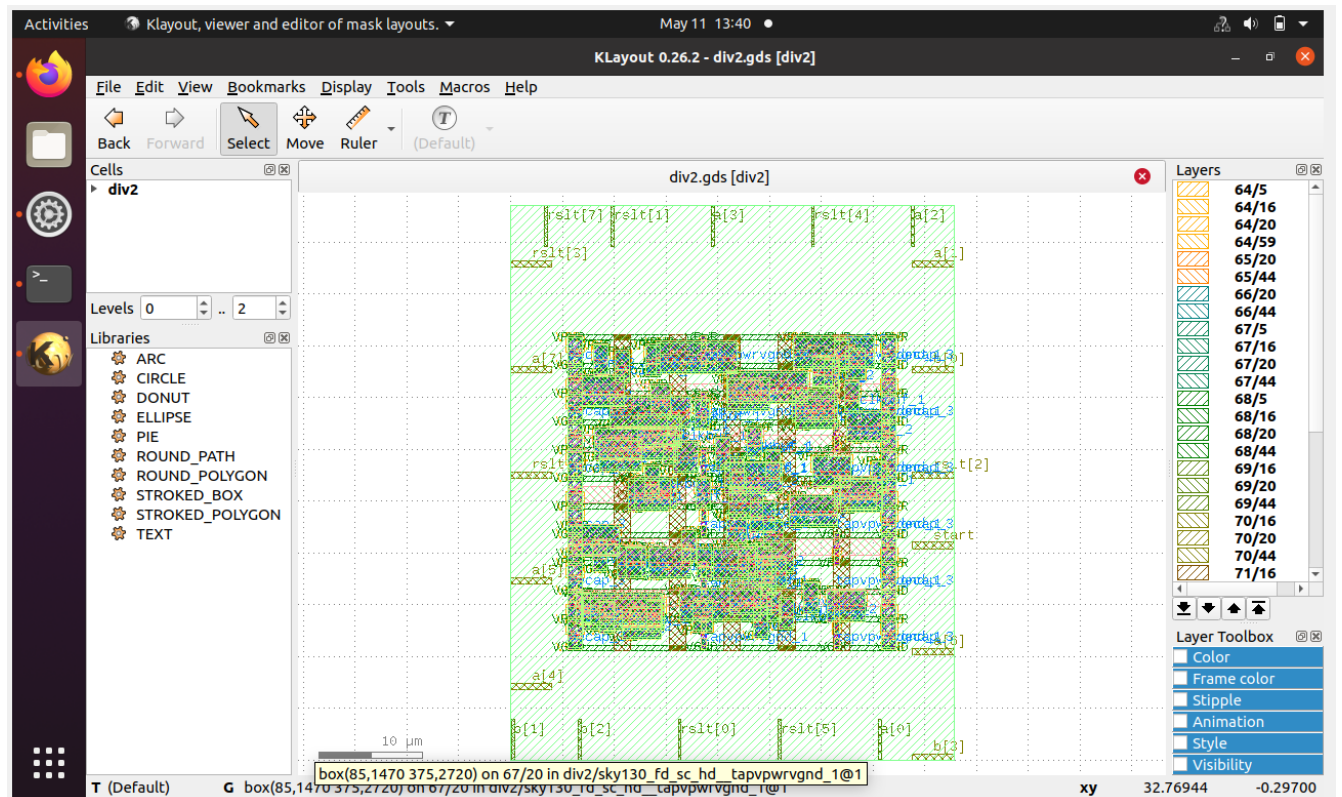


FIGURE 3. 14

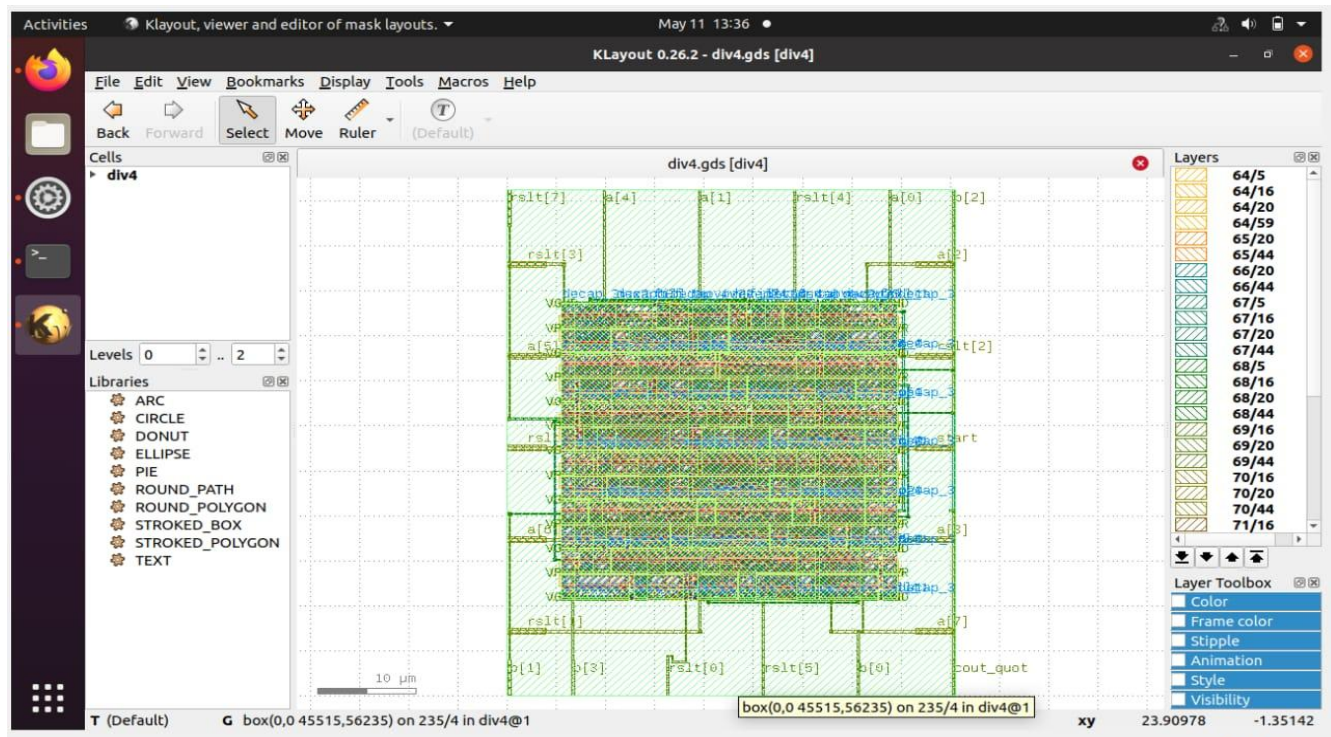


FIGURE 3. 15

CHAPTER 4

EXPERIMENT MODULES

4.1 INTRODUCTION:

In this chapter we are going to talk about the different divider modules for different sign bit which are similar to our original proposed 8/4 bit signed divider, we have done experiment on 16,64 and we are going to explain their simulation and output result.

16 BIT DIVIDER VERILOG MODULE BY USING NON-RESTORING ALGORITHM:

```
module bit44(clk, ready, dividend, divisor,Q,R );
input clk;
input[15:0] dividend;
input [15:0] divisor;
output[15:0] Q;
output[15:0] R;
output ready;
reg[15:0]Q;
reg[31:0] dividend_copy,divisor_copy,diff;
wire[15:0] remainder=dividend_copy[15:0];
reg[31:0] bit;
wire ready=!bit;
initial bit=0;
always @(posedge clk)
if(ready)
begin
bit=16;
Q=0;
dividend_copy={ 16'd0,dividend};
```

```

divisor_copy={1'b0,divisor,15'd0};
end
else
begin
iff=dividend_copy -divisor_copy;
Q=Q<<1;
if( !diff[31])
begin
dividend_copy=diff;
Q[0] =1'd1;
end
divisor_copy=divisor_copy>>1;
bit=bit-1;
end
endmodule

```

SIMULATION RESULT: -

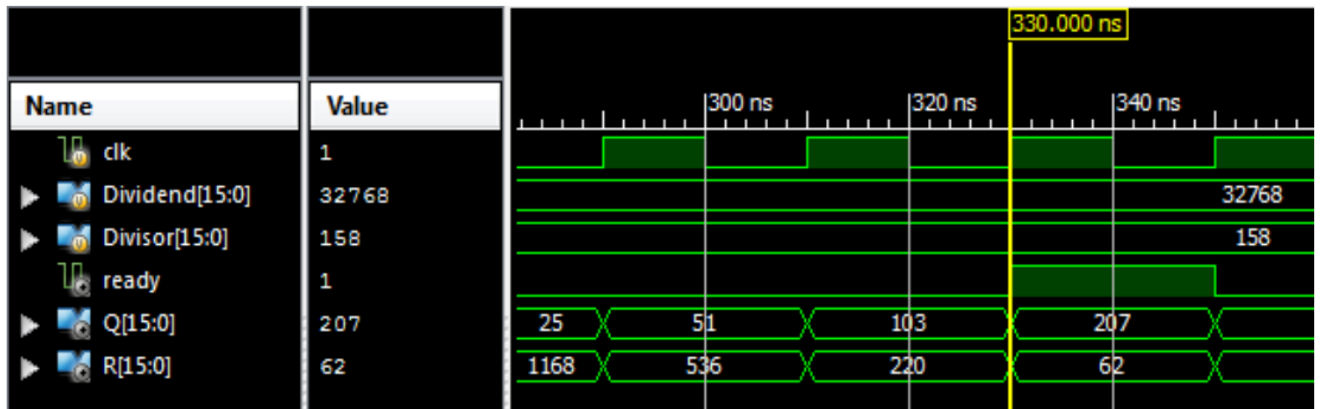


FIGURE 4. 1

8 BIT DIVIDER VERILOG MODULE BY USING RESTORING ALGORITHM:

Verilog module code:

```

module div_restoring (a, b, start, rslt);

```



```

input [7:0] a;
input [3:0] b;
input start;
output [7:0] rslt;
wire [3:0] b_bar;
//define internal registers
reg [3:0] b_neg;
reg [7:0] rslt;
reg [3:0] count;
assign b_bar = ~b;
always @ (b_bar)
b_neg = b_bar + 1;
always @ (posedge start)
begin
rslt = a;
count = 4'b0100;
if ((a!=0) && (b!=0)) //if a or b = 0, exit
begin
rslt = rslt<< 1;
rslt = {(rslt[7:4] + b_neg), rslt[3:0]};
if (rslt[7] == 1)
begin
rslt = {(rslt[7:4] + b), rslt[3:1], 1'b0};
count = count - 1;
end
else
begin
rslt = {rslt[7:1], 1'b1};
count = count - 1;
end
end
end

```

```
end
```

```
endmodule
```

TESTBENCH:

```
module div_restoring_tb;
```

```
reg [7:0] a;
```

```
reg [3:0] b;
```

```
reg start;
```

```
wire [7:0] rslt;
```

```
initial //display variables
```

```
$monitor ("a = %b, b = %b, quot = %b, rem = %b",
```

```
a, b, rslt[3:0], rslt[7:4]);
```

```
initial //apply input vectors
```

```
begin
```

```
#0 start = 1'b0;
```

```
a = 8'b0000_1101; b = 4'b0101;
```

```
#10 start = 1'b1;
```

```
#10 start = 1'b0;
```

```
#10 a = 8'b0011_1100; b = 4'b0111;
```

```
#10 start = 1'b1;
```

```
#10 start = 1'b0;
```

```
#10 a = 8'b0101_0010; b = 4'b0110;
```

```
#10 start = 1'b1;
```

```
#10 start = 1'b0;
```

```
#10 a = 8'b0011_1000; b = 4'b0111;
```

```
#10 start = 1'b1;
```

```
#10 start = 1'b0;
```

```
#10 a = 8'b0110_0100; b = 4'b0111;
```

```
#10 start = 1'b1;
```

```
#10 start = 1'b0;
```

```
#10 a = 8'b0110_1110; b = 4'b0111;
```

```

#10 start = 1'b1;
#10 start = 1'b0;
#10 $stop;
end
div_restoring inst1 ( //instantiate the module
.a(a),
.b(b),
.start(start),
.rslt(rslt)
);
endmodule

```

SIMULATION RESULT:

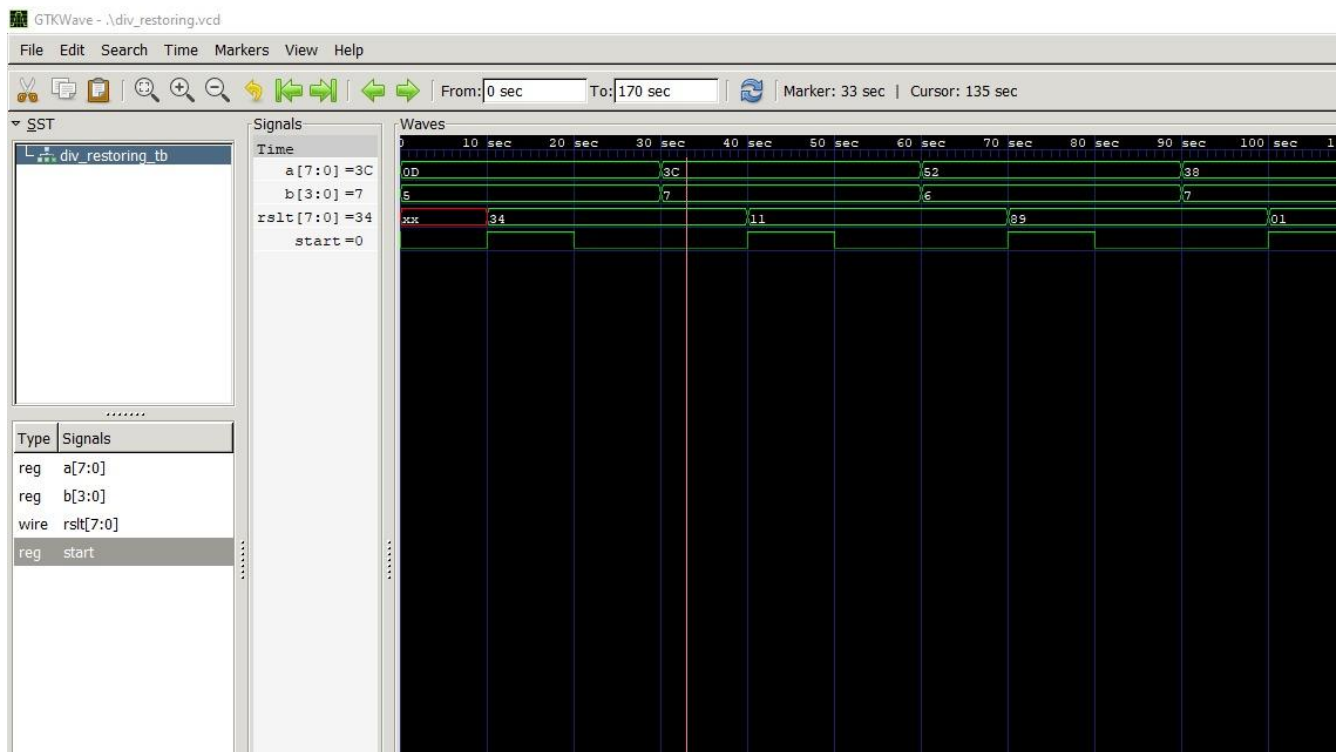


FIGURE 4. 2

SYNTHESIS RESULT:

```
8. Printing statistics.

=== div_restoring ===

Number of wires:          90
Number of wire bits:      114
Number of public wires:   4
Number of public wire bits: 21
Number of memories:       0
Number of memory bits:    0
Number of processes:      0
Number of cells:          43
    sky130_fd_sc_hd__a21oi_1      4
    sky130_fd_sc_hd__a31oi_1      1
    sky130_fd_sc_hd__clkinv_1     4
    sky130_fd_sc_hd__dfxtp_1      8
    sky130_fd_sc_hd__lpflow_isobufsrc_1  2
    sky130_fd_sc_hd__maj3_1       2
    sky130_fd_sc_hd__mux2_1       1
    sky130_fd_sc_hd__mux2i_1      1
    sky130_fd_sc_hd__nand2_1      2
    sky130_fd_sc_hd__nor2_1      4
    sky130_fd_sc_hd__nor4_1      3
    sky130_fd_sc_hd__o21a_1       1
    sky130_fd_sc_hd__o21ai_0      2
    sky130_fd_sc_hd__o22ai_1      1
    sky130_fd_sc_hd__or3_1        1
    sky130_fd_sc_hd__xnor2_1      5
    sky130_fd_sc_hd__xor2_1       1

Chip area for module '\div_restoring': 379.113600
```

FIGURE 4. 3

64 BIT DIVIDER VERILOG MODULE BY USING RESTORING ALGORITHM:

```
module Divide(
    input clk,
    input reset,
    input start,
    input [31:0] A,
```

```

input [31:0] B,
output [31:0] D,
output [31:0] R,
outputok, // =1 when ready to get the result
output errs
);
reg    active; // True if the divider is running
reg [4:0]  cycle; // Number of cycles to go
reg [31:0] result; // Begin with A, end with D
reg [31:0] denom; // B
reg [31:0] work; // Running R
// Calculate the current digit
wire [32:0] sub = { work[30:0], result[31] } - denom;
    assign err = !B;
// Send the results to our master
assign D = result;
assign R = work;
assign ok = ~active;
always @(posedge clk,posedge reset) begin
    if (reset) begin
        active <= 0;
        cycle <= 0;
        result <= 0;
    denom<= 0;
        work <= 0;
    end
    else if(start) begin
        if (active) begin
            // Run an iteration of the divide.
            if (sub[32] == 0) begin
                work <= sub[31:0];

```

```

    result <= {result[30:0], 1'b1};
end
else begin
    work <= {work[30:0], result[31]};
    result <= {result[30:0], 1'b0};
end
if (cycle == 0) begin
    active <= 0;
end
cycle <= cycle - 5'd1;
end
else begin
    // Set up for an unsigned divide.
    cycle <= 5'd31;
    result <= A;
denom<= B;
    work <= 32'b0;
    active <= 1;
end
end
end
endmodule

```

SIMULATION RESULT:-



FIGURE 4. 4

CHAPTER 5

DISCUSSION AND CONCLUSION

We have successfully implemented RTL to GDSII physical design flow for the proposed signed binary divider. Different design parameters such as area, power, delay and time has been obtained. This work has been Performed using the qflow tool, which comprises various open-source tools. At 45nm proposed design is 10.6% power efficient, 95% area-efficient, and 26.74% timing efficient compared to design at 180nm node. Also, the proposed algorithm consumes low power and area. So, it can be used for very low-power integer division applications.

Our goal is to achieve implementation and physical design of a divider that can be efficient and it can consume less area, uses less power and less delay for performing crucial and complex operations in high level and low-level circuits which are widely used in IOT applications, Microprocessors, Microcontrollers and other Industry based applications.

We have implemented 4-6 different division modules wherein they consist of different algorithms and they are restoring algorithm and non-restoring algorithm and each of these modules are used in specific applications depending upon the application requirement/specifications. The aim is to focus on the simplicity of the algorithm which is easy to translate into Verilog codes. This is because the non-restoring method only involved the basic addition and subtraction and shifting process, which is either left or right. Hence, this method can be implemented for any value of binary numbers for division operations. These proposed modules have different outcomes i.e.; some have less latency but large power consumption. and some have more latency and less power usage so based on the requirement of applications different modules are used.

The obtained GDS-2 File is shown below

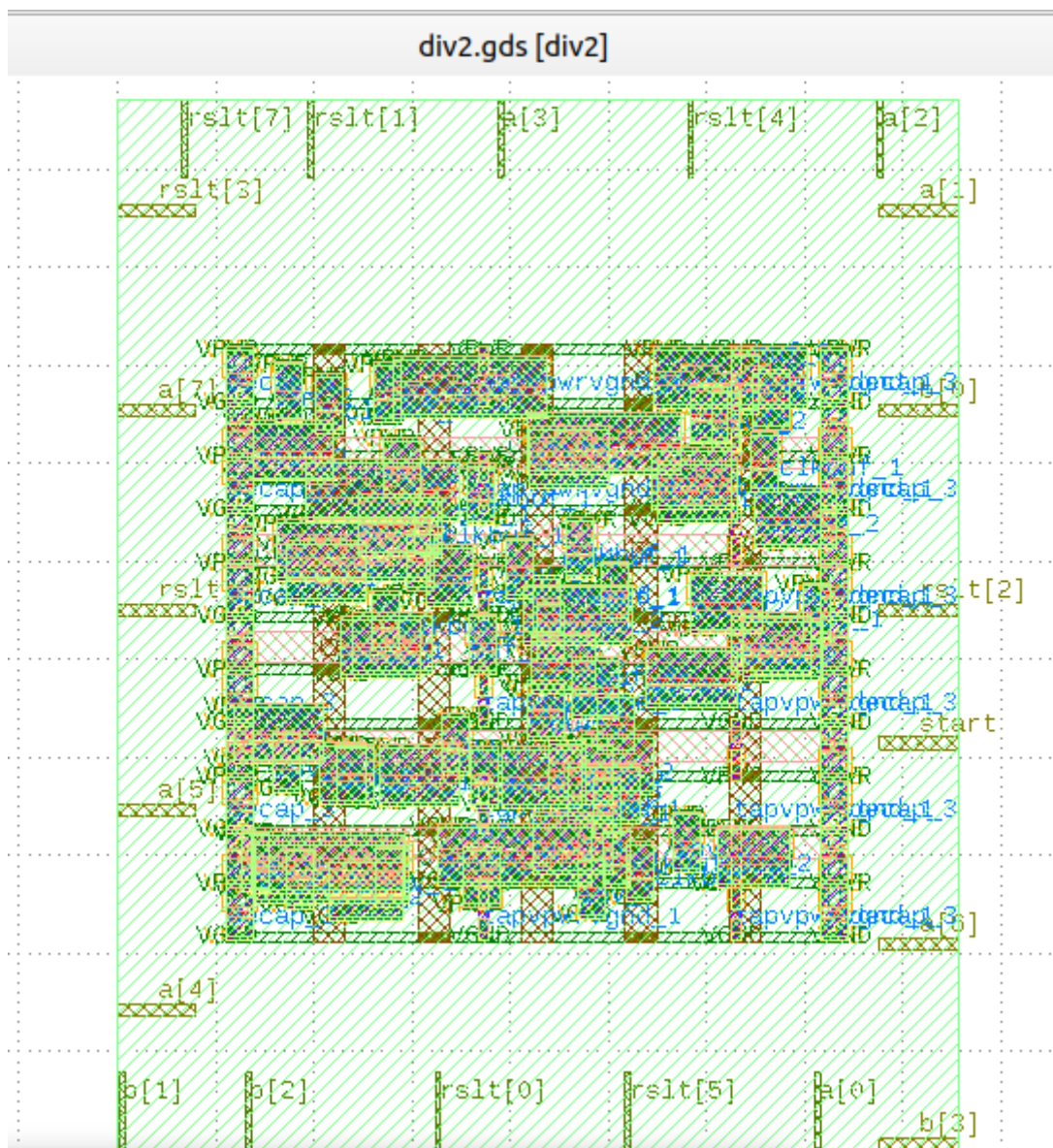
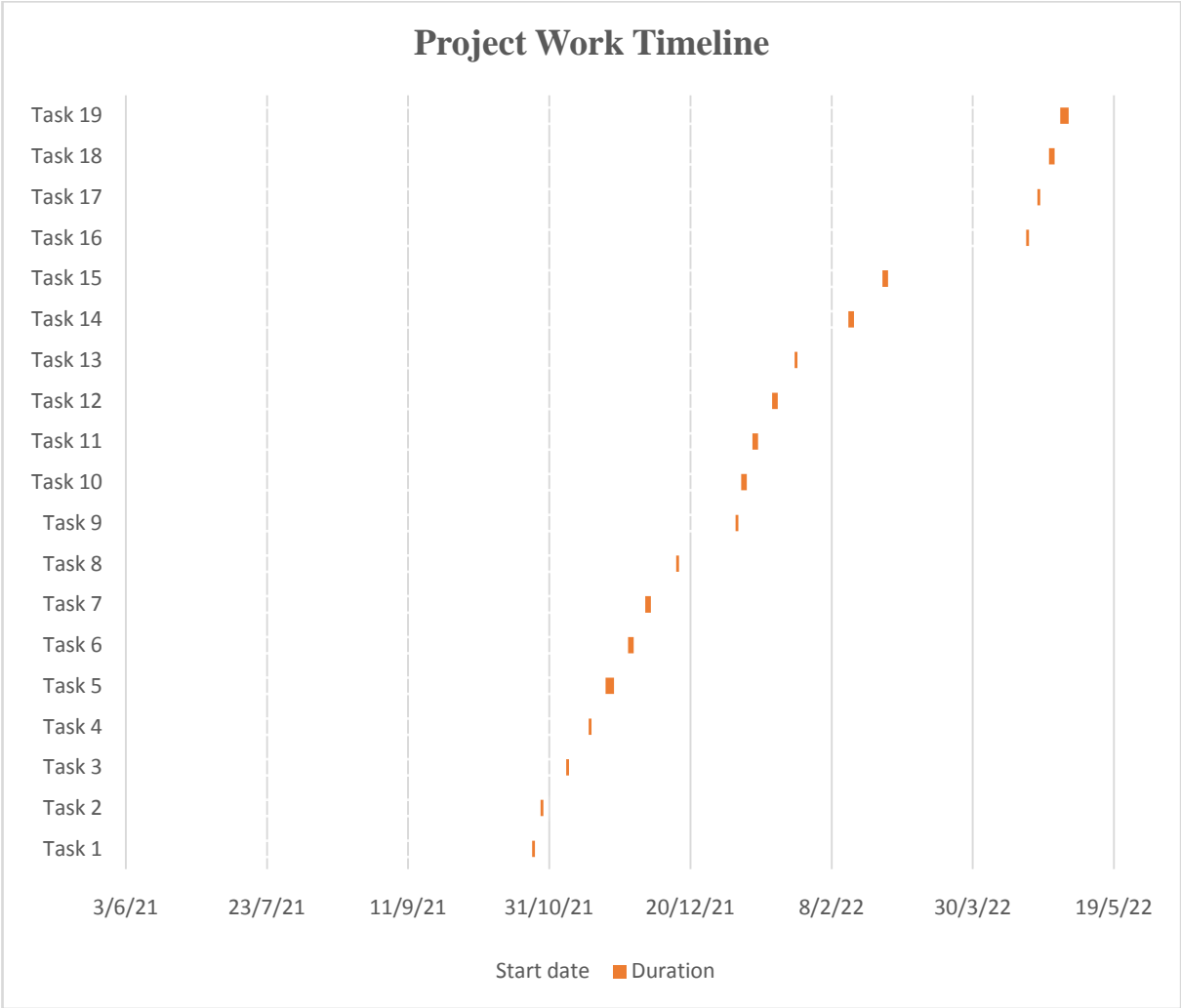


FIGURE 5. 1

APPENDIX -1

GANTT CHART:



APPENDIX-2

APPLICATION OF DIFFERENT BIT DIVIDERS

S.NO	DIVIDER	APPLICATION
1.	64 BIT DIVIDER	MICROCONTROLLER AND MICROPROCESSOR
2.	32 BIT DIVIDER	ALU UNIT
3.	8 BIT DIVIDER	IMAGE RPOCESSING

REFERENCES

- [1] H. Kaur,” Performance Analysis of Various Multiplication and Division Algorithms For Large Numbers”
- [2] Y.Yusmardiah, et al., ”Translation of Division Algorithm Into Verilog HDL,” *ARPJ Journal of Engineering and Applied Sciences*, vol. 12, pp. 3214–3217, 2006.
- [3] T. Aoki, K. Nakazawa, and T. Higuchi,” High-radix parallel VLSI dividers without using quotient digit selection tables,” in *Proc. 30th IEEE International Symposium on Multiple-Valued Logic (ISMVL 2000)*, pp.345-352, 2000.
- [4] K. Reddy, et al., ”Design of approximate dividers for error-tolerant applications,” in *Proc. 61st International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 496-499, 2018.
- [5] Gaalswyk, F. Matthew, and J. Stine,” A Low-Power Recurrence-Based Radix 4 Divider Using Signed-Digit Addition,”in *proc. IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 391-396, 2019.
- [6] R. Zendegani, et al.,” SEERAD: A high speed yet energy-efficient rounding-based approximate divider,” in *proc. Design, Automation & Test in Europe Conference & Exhibition*, pp. 1481-1484, 2016.
- [7] R. Vemula, et. al.,” Design and Implementation of 64-Bit Divider Using 45nm CMOS Technology,” *International Journal of Pure and Applied Mathematics*, 2018.
- [8] S. Panda, and A. Sahu,” A novel Vedic divider architecture with reduced delay for VLSI applications,” *International Journal of Computer Applications*, 2015.
- [9] P. Groeneveld,” Physical design challenges for billion transistor chips,” in *proc. IEEE International Conference on Computer Design: VLSI in Computers and Processors. IEEE*, pp. 78-83, 2002.
- [10] A. Saleh, et al.,” High speed digital CMOS divide-by-N frequency

divider,” *IEEE International Symposium on Circuits and Systems*, pp. 592-595, 2008.

[11] M. Basha, et al.,” Novel Low Power and High-speed array divider in 65 nm Technology,” *International Journal of Advances in Science and Technology*, pp.44-56.

[12] C. Senthilpari, et al.,” Lower delay and area efficient non-restoring array divider by using Shannon based adder technique,” *in proc. IEEE International Conference on Semiconductor Electronics*, pp. 140-144, 2010.

[13] S. Venkatachalam, et al.,” Design of approximate restoring dividers,” *IEEE International Symposium on Circuits and Systems*, pp. 1-5, 2019.

[14] J. Jung, et al.,” OpenDesignflow database: the infrastructure for VLSI design and design automation research,” *in proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, 2016.

[15] K. Ghosh, and A. Ghosh,” Technology mediated tutorial on RISC-V CPU core implementation and sign-off using revolutionary EDA management system (EMS)—VSDFLOW,” *in proc. China Semiconductor Technology International Conference (CSTIC)*, IEEE, pp.1-3, 2018.

[16] J. Dean, D. Patterson, and C. Young,” A new golden age in computer architecture: Empowering the machine-learning revolution,” *IEEE Micro*, vol. 38, no. 2, pp. 21-29, 2018.

[17] W. Lee,” Verilog coding for logic synthesis,” *Wiley-Interscience*, 2003.

[18] J. Hennessy, and D. Patterson,” Computer architecture: a quantitative approach,” *Elsevier*, 2011.

[19] G. Khosrow,” Physical design essentials,” *Springer Science+ Business Media, LLC*, 2007.

[20] A. Kahng, et al.,” VLSI physical design: from graph partitioning to timing closure,” *Springer Science & Business Media*, 2011.