A REPORT

ON

# **FPGA Implementation of Goldschmidt Division Algorithm**

BY

Names of the students ID.No.

Pavithra Ramesh 2023AAPS0362H

Pritham Kumar Jena 2023A8PS1308H

Harikriti Murali 2023AAPS0265H

AT

INTEL SOLUTIONS AND SERVICES PVT LTD

A Practice School-I Station of

**BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI**

**(JUNE,2025)**

A REPORT

ON

# **FPGA Implementation of Goldschmidt Division Algorithm**

BY

Names of the students ID.No.s Disciplines

Pavithra Ramesh 2023AAPS0362H Electronics & Communication Engg

Pritham Kumar Jena 2023A8PS1308H Electronics & Instrumentation Engg

Harikriti Murali 2023AAPS0265H Electronics & Communication Engg

Prepared in partial fulfillment of the

Practice School-I Course Nos.

BITS C221/BITS C231/BITS C241

AT

INTEL SOLUTIONS AND SERVICES PVT LTD

A Practice School-I Station of

**BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI**

**(JUNE,2025)**

**ACKNOWLEDGEMENT**

We would like to express our sincere gratitude to **Mr.Pawan Sharma** and **Mr. Padmanaban**, for their valuable guidance, support, and encouragement throughout this work. Their insights and constructive feedback helped us refine our approach and improve the quality of this report.

I am also thankful to **INTEL SOLUTIONS AND SERVICES PVT LTD (UNNATI PROGRAM)** for providing the necessary resources and a conducive environment to carry out this project.

I would also like to acknowledge the support of my teammates for their constant motivation and assistance during challenging phases of the project.

**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI**

**(RAJASTHAN)**

**Practice School Division**

**Station:** Online **Centre:** INTEL SOLUTIONS AND SERVICES PVT LTD

**Duration:** 2 months **Date of start:** 26th May 2025

**Date of submission:** 27th June 2025

**Title of project:** FPGA Implementation of Goldschmidt Division Algorithm

**ID.No.s/ Names/ Discipline of the students:**

Pavithra Ramesh 2023AAPS0362H Electronics & Communication Engg

Pritham Kumar Jena 2023A8PS1308H Electronics & Instrumentation Engg

Harikriti Murali 2023AAPS0265H Electronics & Communication Engg

**Name and designation of the expert:** Mr. Padmanaban

**Name of the PS faculty:** Mr. Pawan Sharma

**Key words: FPGA, Fast division Algorithm : Goldschmidt, Verilog, FSM**

**Project areas: FPGA implementation of the Verilog code of Goldschmidt Algorithm, focusing on Optimization too.**

**Abstract:** This project focuses on implementing a hardware-efficient division algorithm using Verilog on an FPGA, with particular emphasis on the Goldschmidt method. Given that the Intel MAX10 is optimized for multiplication rather than division, we selected Goldschmidt’s algorithm due to its compatibility with DSP blocks and iterative refinement through multiplication. Our design was developed and synthesized using Quartus Prime Lite and validated through functional simulation in QuestaSim.

**Signature of Student Signature of PS Faculty Signature of Scientist**

**Date Date Date**

**TABLE OF CONTENTS**

* **INTRODUCTION**
* **GOLDSCHMIDT ALGORITHM**
* **Q-POINT NOTATION**
* **NORMALIZATION AND RECIPROCAL ESTIMATE**
* **BLOCK DIAGRAM**
* **FINITE STATE MACHINE**
* **STATE TABLE**
* **CODE**
* **TESTBENCH OUTPUT FOR Q4.12 & Q4.4**
* **ANALYSIS & ELABORATION**
* **ANALYSIS & SYNTHESIS**
* **FITTER**
* **ASSEMBLER**
* **POWER ANALYSIS**
* **COMPARISON**
* **CONCLUSION**
* **REFERENCES**

**INTRODUCTION**

When it comes to hardware implementation of arithmetic operations, division has consistently stood out as one of the more challenging computations. While addition, subtraction, and multiplication can be implemented efficiently using relatively straightforward circuits, division typically requires more complex logic and a deeper understanding of iterative approximation techniques. In software programming, the division operator (/) is a simple, one-step instruction. However, in hardware, and more specifically on FPGAs (Field Programmable Gate Arrays), this operation can be viewed with much more scrutiny. Our project began with a literature review of various division algorithms in order to understand how they work at the hardware level and how feasible they are for implementation on an FPGA. Aside from this, we also looked into and understood the fixed point arithmetics and number system. Hence considering many factors, we decided to go ahead with This review, which set the foundation for our design approach and also helped us understand the subsequent course of action too (like pipelining to help use the parallel processing nature of the algorithm better) for an improved efficiency, and optimized resource utilization.

Our first task was to explore the landscape of division algorithms, starting from the more traditional ones like Restoring Division and Non-Restoring Division, and moving toward more advanced techniques such as SRT, Goldschmidt, and Newton-Raphson. Each of these algorithms offers its own trade-offs in terms of speed, accuracy, hardware complexity, and suitability for fixed-point or floating-point implementations. Restoring division, for example, is a digit recurrence algorithm that performs successive subtractions and checks for underflow in each step. If an intermediate result goes negative, the algorithm restores the previous value by adding back the divisor. Although simple to understand and implement, this restoring step introduces inefficiency and makes it less desirable for high-speed applications.

Non-restoring division improves upon this by removing the explicit restoration step. Instead of backtracking, it adjusts the result in the following iteration, which allows for a more streamlined process. This small change makes a noticeable difference in performance and hardware resource usage. Then we have the SRT algorithm. This algorithm allows for faster, carry-free computations and simultaneous generation of quotient digits. However, it also comes with increased complexity, including the need for lookup tables and additional control logic, which can make it harder to implement, especially in resource-constrained environments.

As we moved further into our review, the Goldschmidt algorithm became particularly interesting. Unlike digit recurrence algorithms, Goldschmidt is based on convergence and transforms the division operation into a sequence of multiplications and additions. The basic idea is to multiply both the numerator and denominator by a factor that drives the denominator toward 1, thus refining the quotient over several iterations. What makes this algorithm especially appealing for FPGAs is its compatibility with parallel processing and its reliance on multiplication, an operation that is very efficient on modern FPGA architectures due to the presence of dedicated DSP blocks. These blocks are optimized for fast arithmetic operations and can handle multiplication, addition, and subtraction with high throughput. Goldschmidt's approach aligns well with this hardware setup, making it a strong candidate for our implementation.

We also took time to reflect on the relevance of these algorithms in the context of FPGA-based digital design. Modern FPGAs, such as the Intel MAX10, include a wide range of built-in components like LUTs (Look-Up Tables), flip-flops, I/O blocks, and most importantly for us, DSP slices. These DSP units are built specifically for high-speed arithmetic, particularly multiplication. This makes algorithms like Goldschmidt and Newton-Raphson, which reformulate division into multiplication, significantly more efficient on FPGAs than traditional digit recurrence methods. Division, unlike multiplication, is not natively supported by these blocks.

This leads to an important insight about the Verilog programming language. While it allows the use of the division operator (/) during simulation, this operator is not synthesizable in a straightforward way. During synthesis, most FPGA tools either reject the division operator or convert it into a slow iterative process that consumes large amounts of logic and clock cycles. It is also vendor-dependent. Some synthesis tools may map it to proprietary IP cores or create elaborate state machines that are inefficient in terms of both timing and area. When Pawan sir had asked us this question, we looked it up and that made us further understand the importance of this project topic

To bring our design from theory into practice, we used a standard FPGA development workflow. Our primary development environment was Intel Quartus Prime Lite Edition, which provided the tools necessary for writing Verilog code, synthesizing it into a hardware bitstream, and uploading it to the FPGA. The FPGA platform we used was the MAX10 development board. For simulation and verification, we made use of QuestaSim, a simulation tool used to write and run testbenches. QuestaSim allowed us to validate our design before committing it to hardware, saving time and helping us catch logical errors early in the design cycle. Writing proper testbenches was an essential part of the process, as it enabled us to check whether the intermediate values in our division algorithm were converging as expected and that the final output matched the required accuracy, this further helped us catch errors in our code, helping in effective debugging.

**GOLDSCHMIDT ALGORITHM**

The goldschmidt division algorithm is an iterative algorithm that performs division by performing a series of multiplications. Division being a costly operation, this makes the algorithm comparatively more efficient.

The main idea of the algorithm is to gradually adjust the denominator towards “1”, while simultaneously adjusting the numerator to approach the final quotient. This strategy simplifies the problem, as once the denominator is approximately one, the numerator itself becomes the quotient.

The steps involved in goldschmidt division algorithm are as follows:

1. **Normalizing** the denominator to bring it within a specific range (typically [0.5,1)).
2. Estimating the **reciprocal** of the normalized denominator using a lookup table.
3. **Multiplying** both the numerator and the denominator with this reciprocal estimate.
4. Further **refining** of the answer is done by repeatedly multiplying the numerator and denominator with a correction factor (This is the main iterative process of the goldschmidt division algorithm).
5. After the iterative process is done, we perform a **denormalization** operation to correct our answer.
6. The **final answer** is stored in the numerator.

The repeated multiplications with the correction factor make this an iterative algorithm and thus higher the number of iterations we perform, the more accurate our final answer will be. Although from our tests, usually 2 to 4 iterations are needed to achieve sufficient accuracy.

Due to the presence of repeated multiplications and subtractions only, the algorithm is especially well suited for parallel processing and pipelined hardware architectures. This can make this algorithm achieve even higher throughput and lower latency. Also due to the absence of actual division units the area and power taken is comparatively lower.

However, goldschmidt division also comes with a few challenges. It requires accurate normalization logic to perform shift operations correctly. Additionally, the reciprocal estimate plays a crucial role in the convergence rate of the algorithm. A poorly chosen reciprocal estimate may come with the need to perform a higher number of iterations. Despite these concerns, the goldschmidt division algorithm remains one of the most widely used approaches for hardware-based division due to its speed, simplicity in arithmetic operations, and excellent compatibility with pipelined processing.

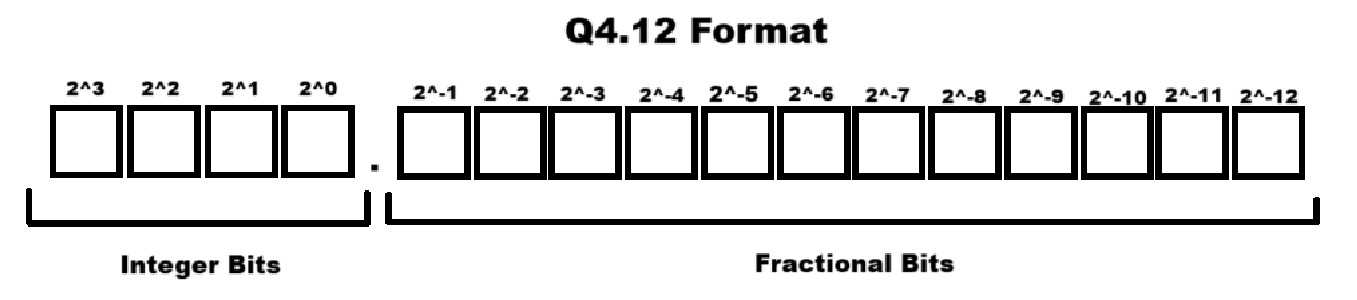
**Q-POINT NOTATION**

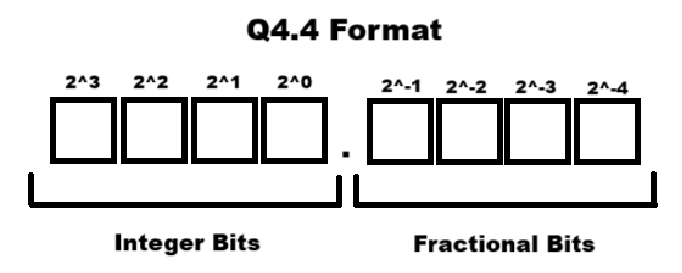
In digital systems, the Q-Point notation is widely used to represent **fixed point** numbers. Fixed point numbers allocate a fixed number of bits for the integer and fractional part, unlike floating point where the range is wide due to the presence of an exponent and mantissa. The Q-point notation provides a very convenient and standardized way to represent this allocation.

The notation **Qm.n** is used to represent fixed point numbers. Here, “m” represents the number of bits in the integer part (including sign bit if signed) and “n” represents the number of bits in the fractional part.

The key idea in Q-point notation is that all numbers are stored as integers in the hardware. To get the actual real number from the integer, we need to divide the integer by 2^n. This can be achieved by a right shifting operation.

We performed the goldschmidt division using two formats, unsigned Q4.4 and unsigned Q4.12 and then compared them. The Q4.12 format provides a range of **0 to 15.99975586** for unsigned numbers and the Q4.4 format has a range of **0 to 15.9375**.

****



**Operations involving Q-Point format in goldschmidt algorithm:**

* **Addition/Subtraction:** These operations are valid if the Qm.n formats of the operands match precisely. If possible the formats can be matched by using appropriate bit shifting operations.
* **Multiplication:** The result of multiplying two Qm.n numbers is wider. For example, two Q4.12 numbers multiplied by each other give a Q8.24 number. Similarly, Q4.4 multiplication results in a Q8.8 number. To return back to the original format, the result must be right shifted by ‘n’ bits. (For eg, 12 for Q4.12)
  + **Rounding to Nearest:** Simple shifting right truncates the result, which can introduce small errors. To perform rounding to the nearest value instead of truncating, you add half of the amount you are about to shift away. That value is 2^(n-1), where n is the number of bits you are shifting.

The biggest advantage of Fixed point Qm.n notation is that it provides predictable precision and also good performance. Fixed point representation maintains a constant resolution throughout their range. This format can also be used for lower latency and smaller hardware setups, making it ideal for any power/area constraints. It allows designers to control precision and range explicitly, which is especially useful in resource-constrained environments such as FPGAs, microcontrollers, and DSPs.

Like anything, this format also comes with a few trade-offs. The limited bit width can lead to **quantization errors**, **overflow**, or **loss of precision** if not managed carefully. Choosing the right balance between number of integer bits (n) and fractional bits (m) is crucial for accuracy and range.

**NORMALIZATION AND RECIPROCAL ESTIMATE**

The main idea is that before estimating the reciprocal, we normalize the denominator to bring it into the range [0.5, 1) in Q4.12/Q4.4 format. This makes it easier to find a reciprocal estimate using a simple lookup table. Without normalization, the input could be in a much wider range, which would require a much larger lookup table.

**How do we normalize?**

We shift the most significant '1' in the denominator to the 11th bit position, which is the bit just before the decimal point in Q4.12 format. (Similarly we move to the 3rd bit position in Q4.4 format). This ensures that the first 4 bits (the integer part) become 0, and the rest (the fractional part) contain the actual value. As a result, the number falls within the [0.5, 1) range.

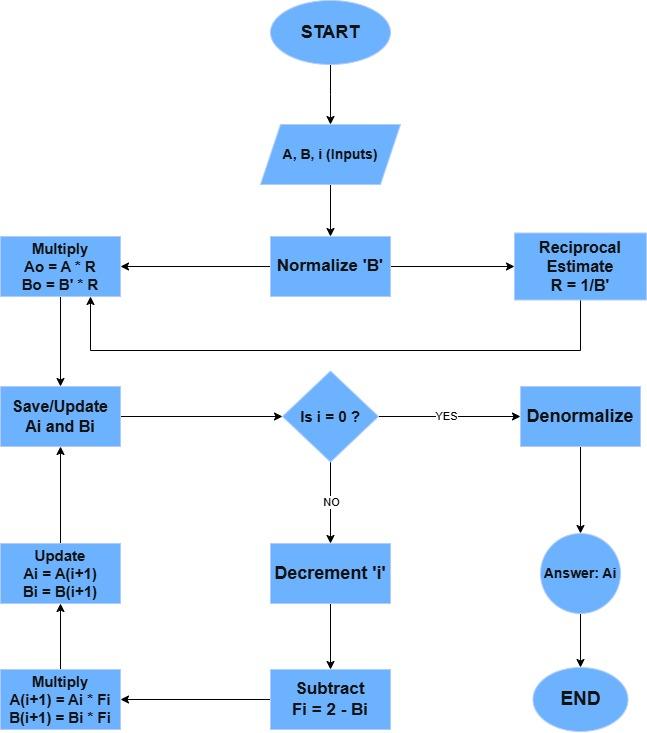
**Priority Encoder for Shift Calculation:**

To find how much to shift, we use a priority encoder that checks the bits from MSB to LSB and returns the position of the first ‘1’. Since it prioritizes higher bits, it's called a "priority" encoder.

**LUT Indexing for Reciprocal Estimation:**

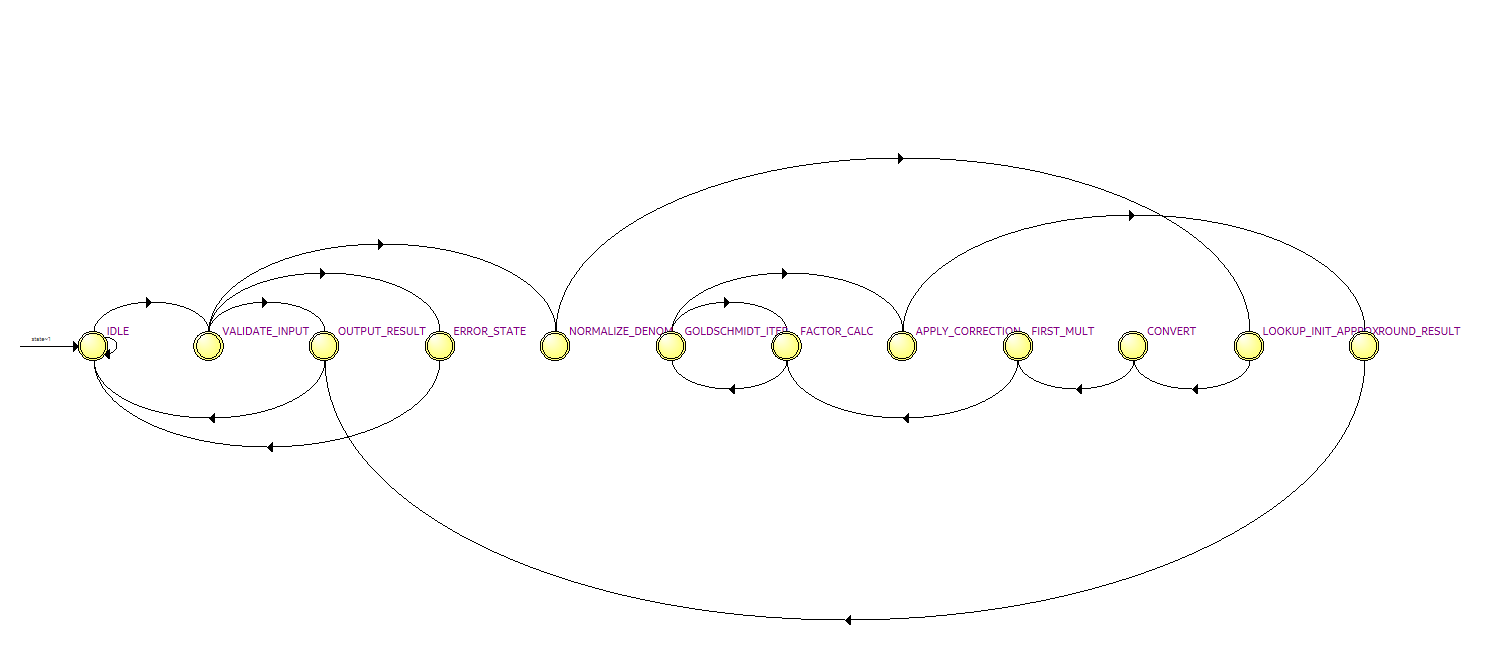
After normalization, we take the top 4 bits of the fractional part to index into a small lookup table. We convert the top 4 bits to an equivalent 3 bit as the MSB of the 4 bits is always 1 due to our normalization operation. This gives us a lookup table with 8 entries (2^3 = 8). This table stores precomputed reciprocals for values spaced evenly within the [0.5, 1) range, allowing us to quickly get a rough estimate of the reciprocal.

**BLOCK DIAGRAM**

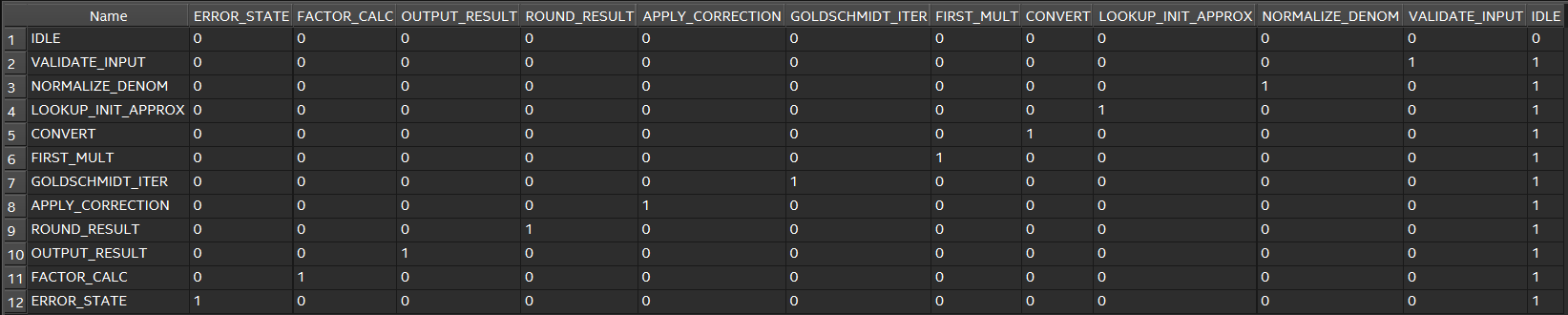
****

**FINITE STATE MACHINE**

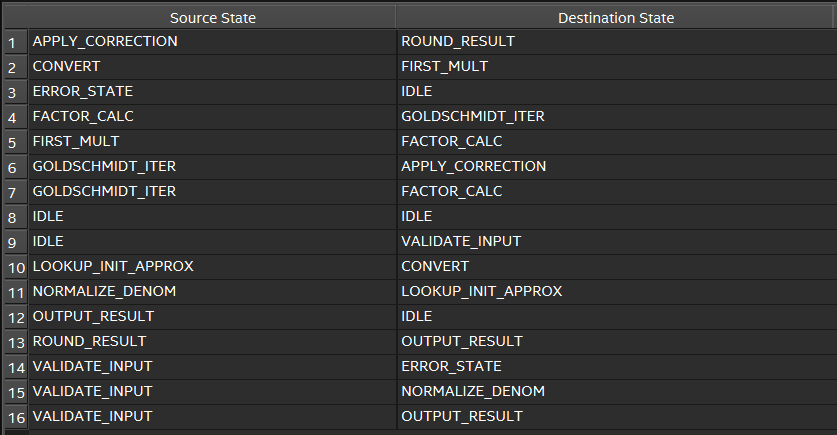
* **IDLE:** Waits for the start signal to begin the division process
* **VALIDATE\_INPUT:** Checks for special cases (den=0, num=den, num=0, den=1)
* **NORMALIZE\_DENOM:** Calculates how many shifts are needed and performs normalization of denominator
* **LOOKUP\_INIT\_APPROX:** Fetch the reciprocal estimate of the denominator from the lookup table.
* **CONVERT:** Convert inputs to Q8.24 for Q4.12 and Q8.8 for Q4.4
* **FIRST\_MULT:** Performs first multiplication with reciprocal estimate. (Needed to avoid clock lag)
* **FACTOR\_CALC:** Calculates the refinement factor: Fi = 2 - current\_denominator
* **GOLDSCHMIDT\_ITER:** Iteratively refines the numerator and denominator using refinement factor.
* **APPLY\_CORRECTION:** Denormalization of the numerator with the same shift as denominator.
* **ROUND\_RESULT:** Round result back to Q4.12 from Q8.24 / Q4.4 from Q8.8
* **OUTPUT\_RESULT:** Set valid signal to HIGH
* **ERROR\_STATE:** Set error signal to HIGH



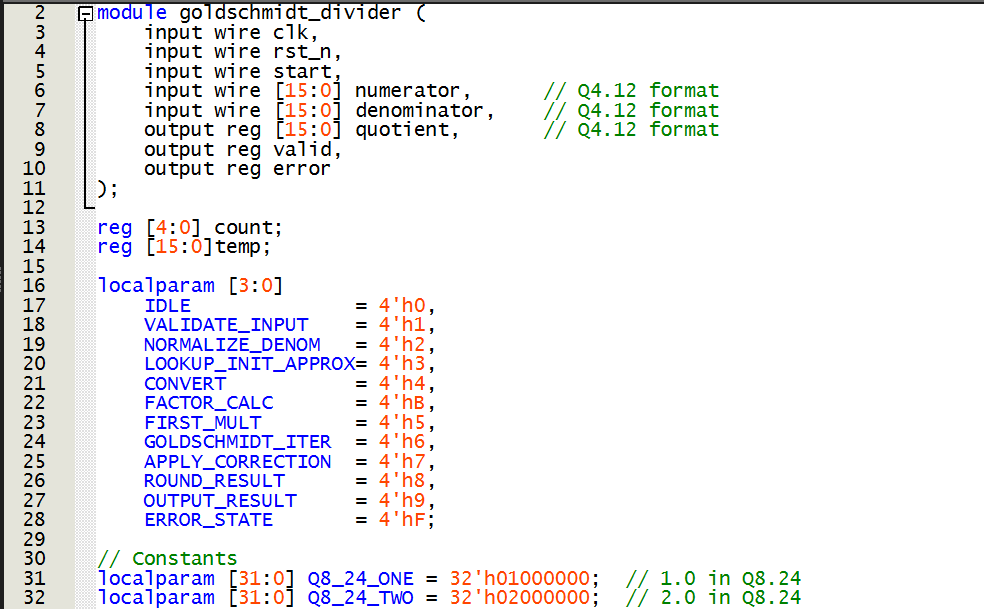
**STATE TABLE (Encoding)**

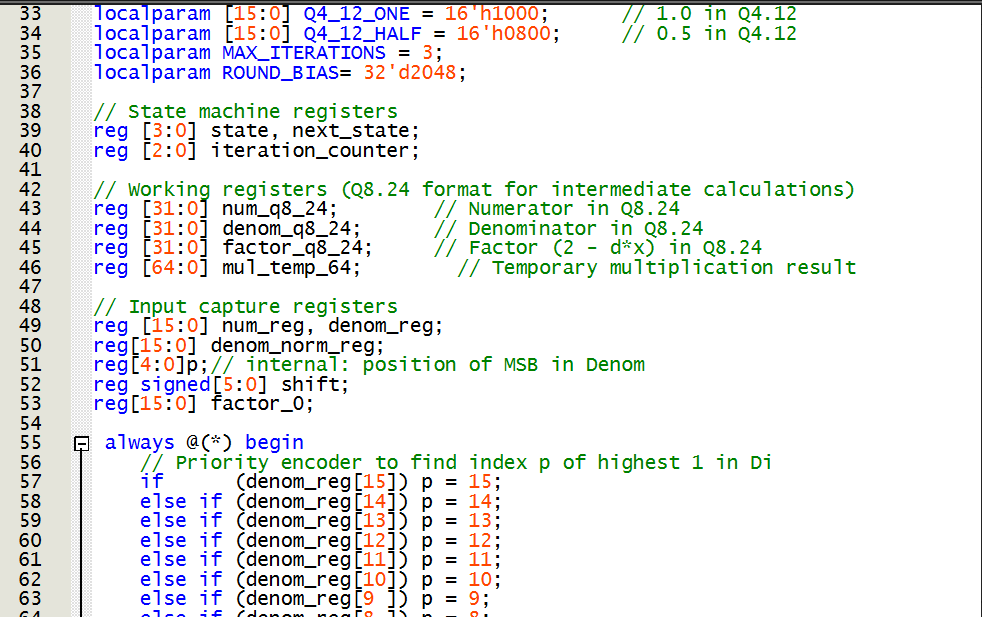
****

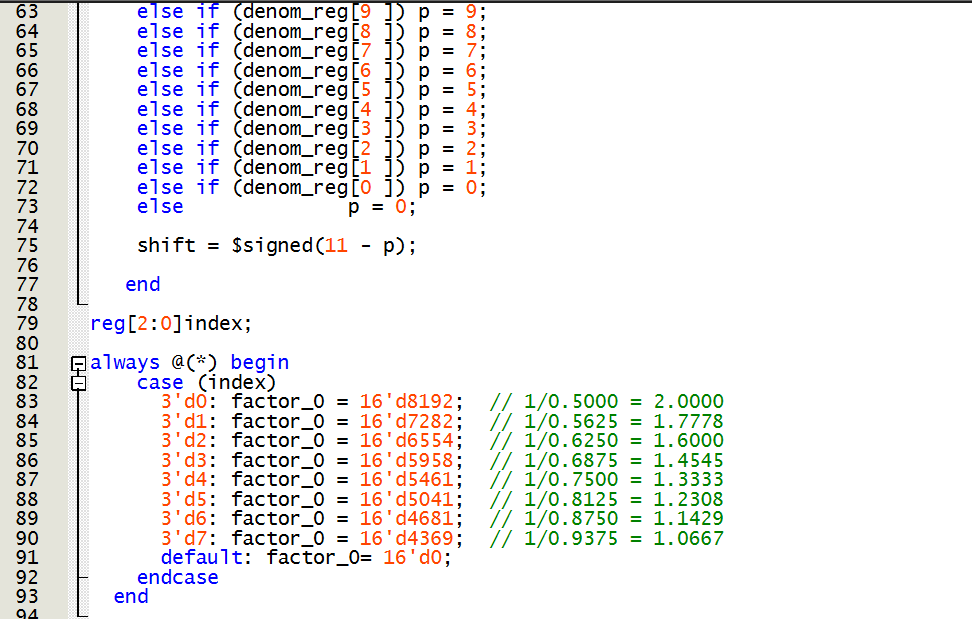
**STATE TABLE (Transitions):**

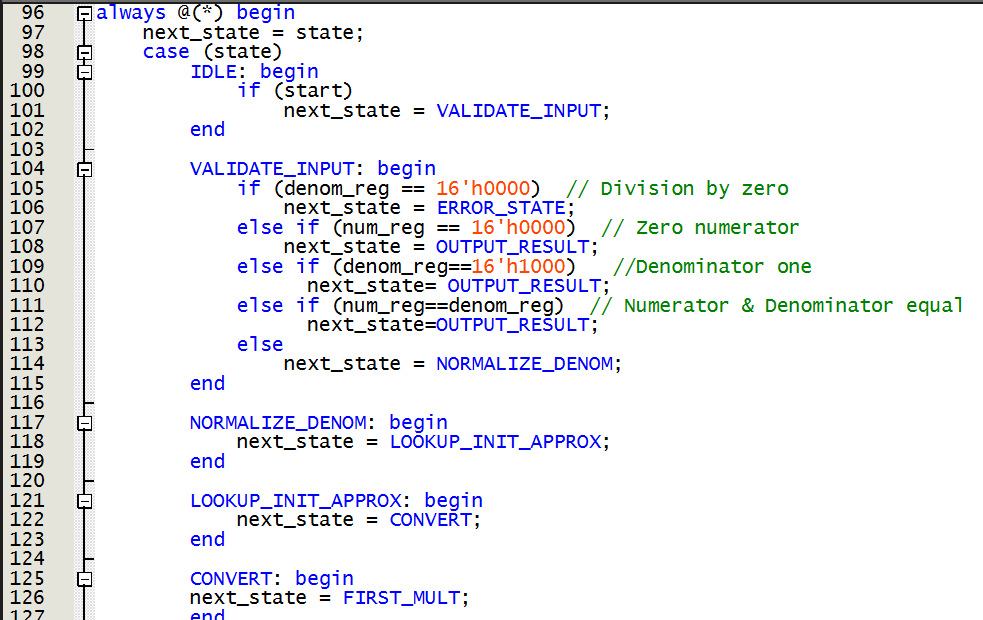


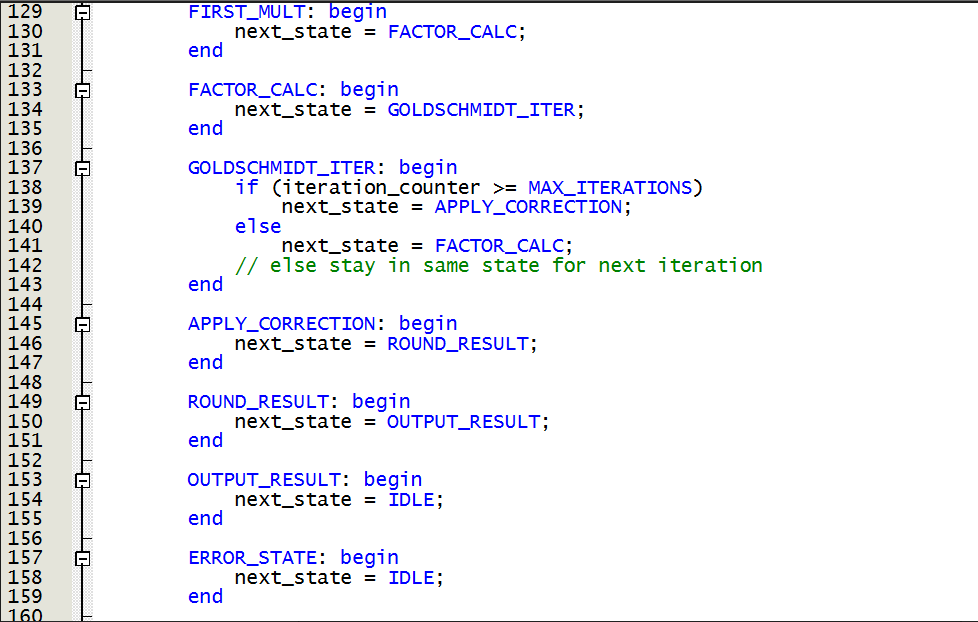
**CODE**

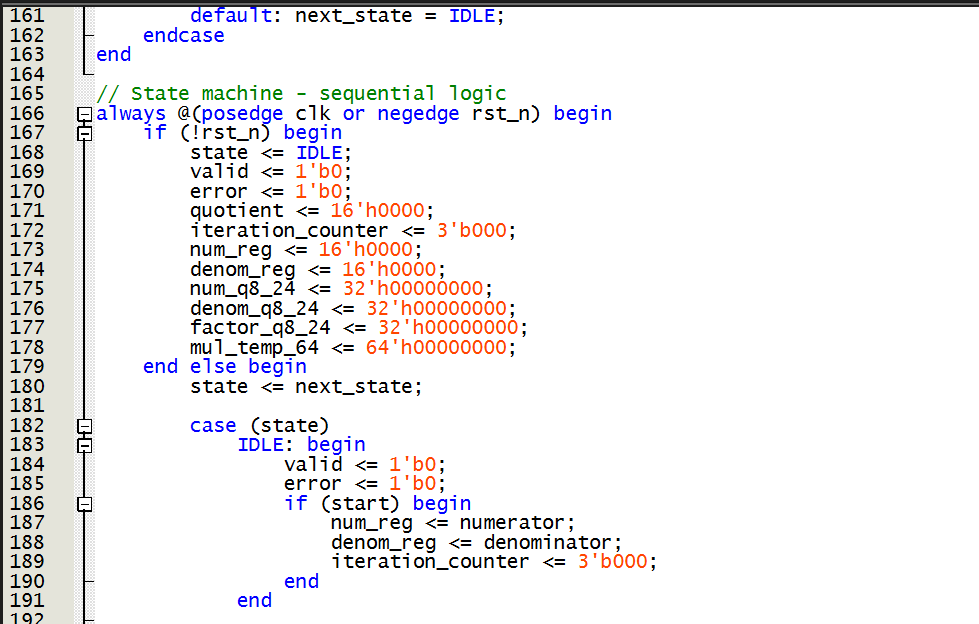
****

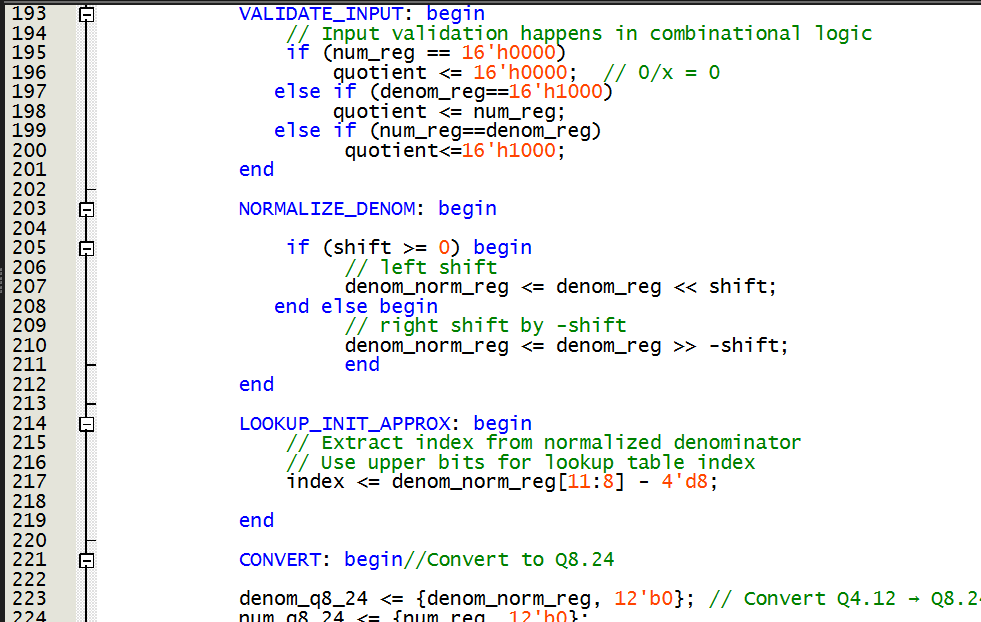
****

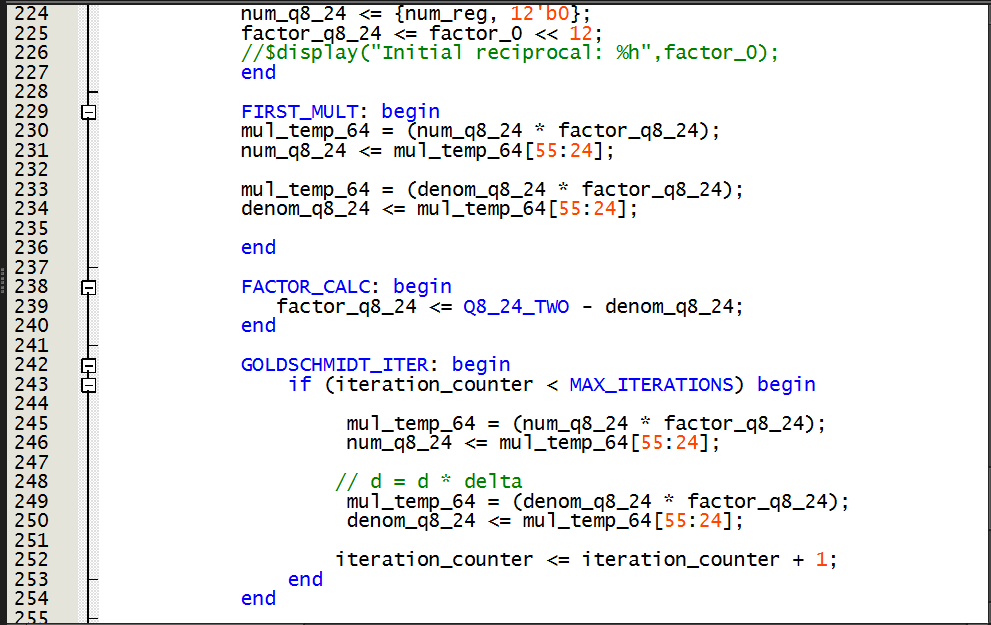
****

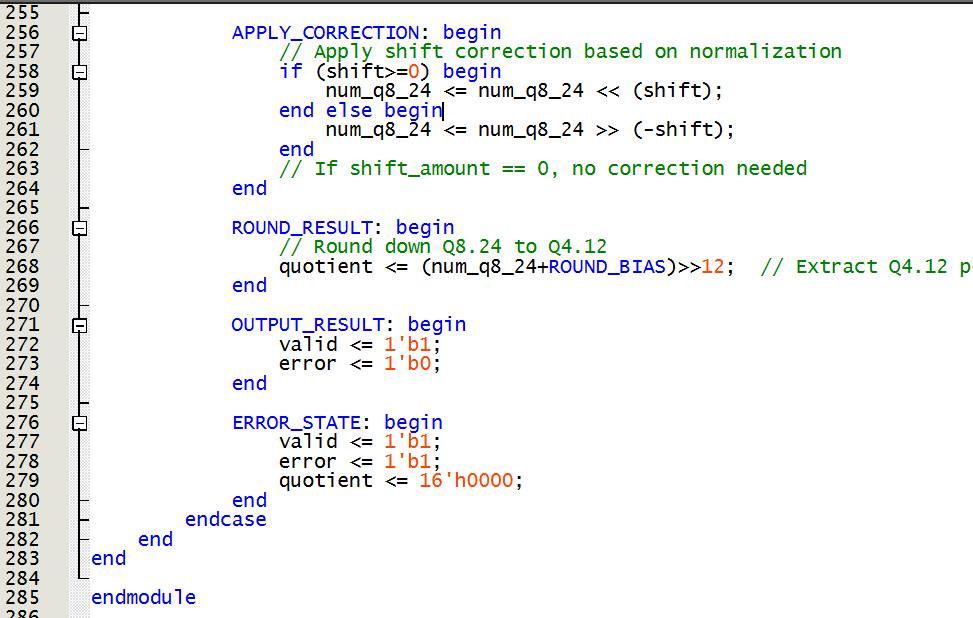
****

****

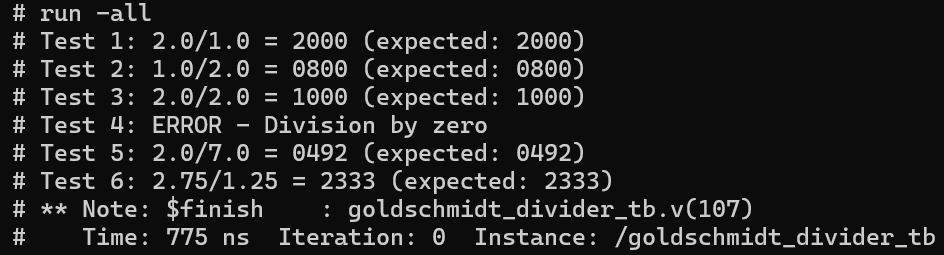
****

****

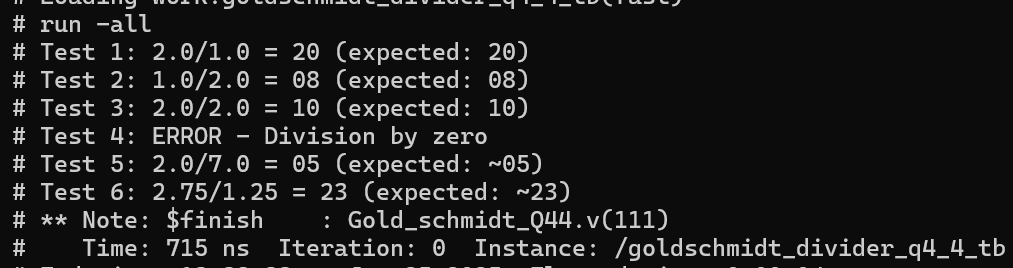
****

****

**TESTBENCH OUTPUT FOR Q4.12**

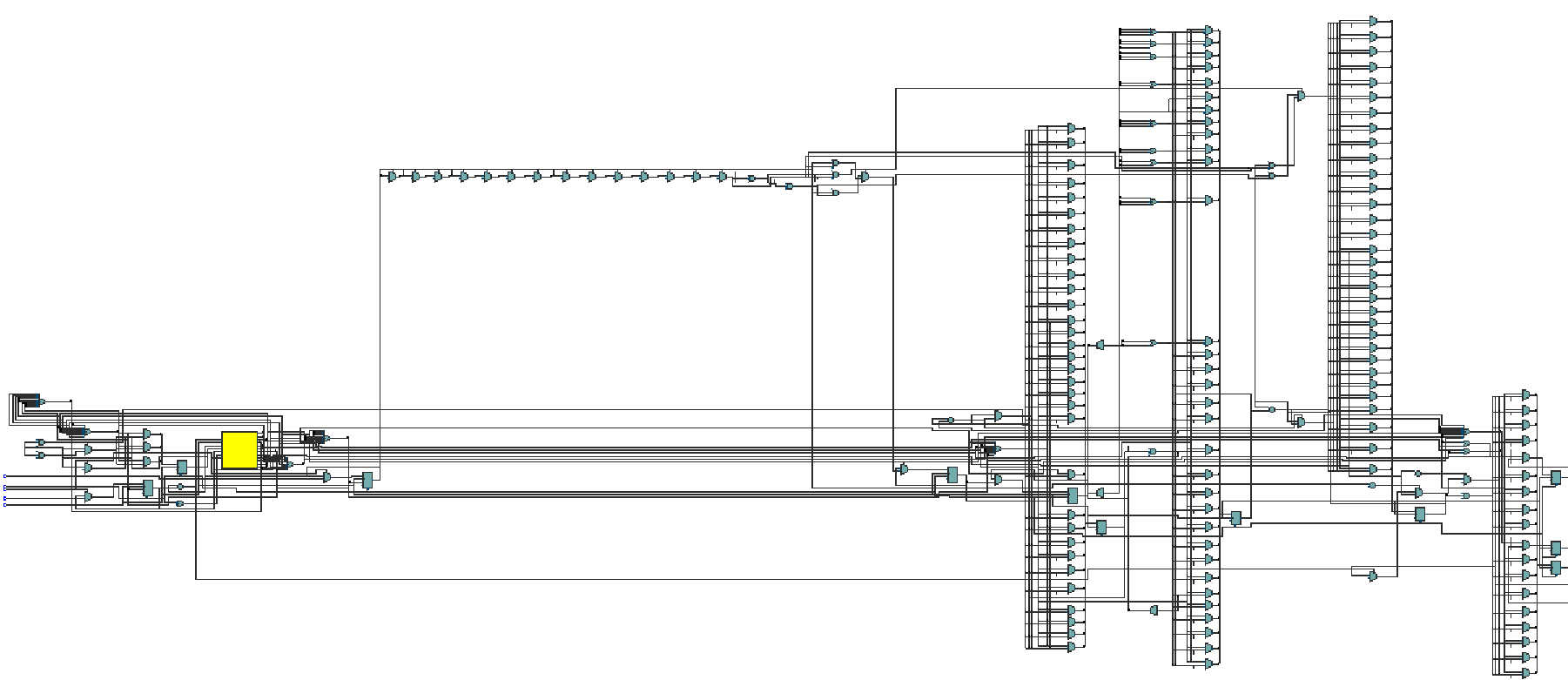
****

**TESTBENCH OUTPUT FOR Q4.4**

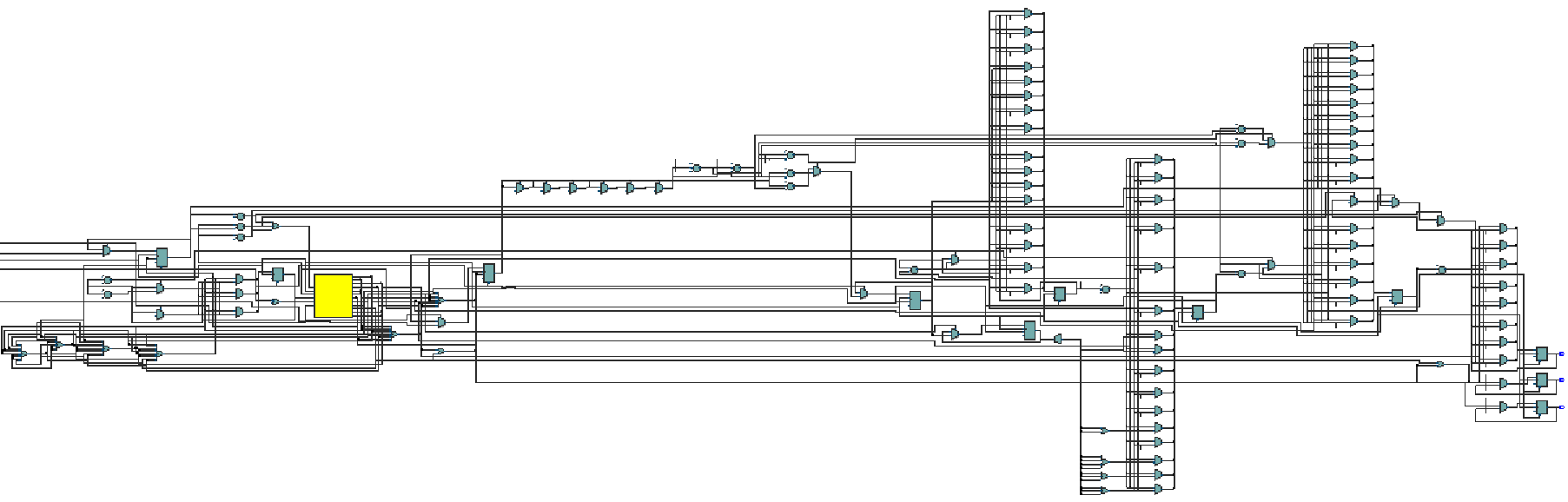
****

**ANALYSIS AND ELABORATION**

This is the first step before synthesis. It parses through our Verilog/VHDL code and checks syntax and semantics. It builds a hierarchical internal representation of given hardware design. It generates a **RTL (Register transfer level)** **Viewer** showing how our logic is structured before synthesis.



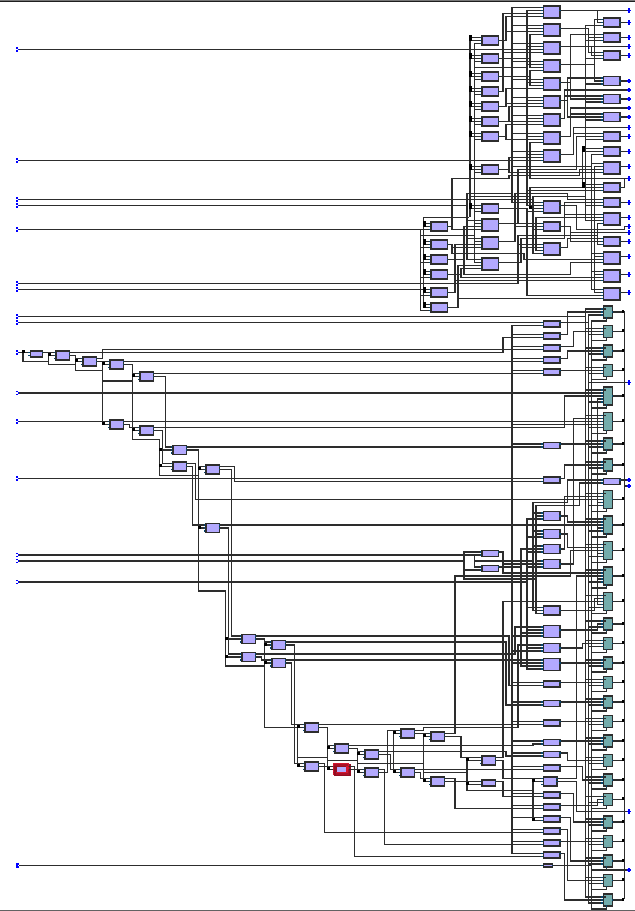
RTL viewer for Q4.12



RTL viewer for Q4.4

**ANALYSIS AND SYNTHESIS**

Itconverts RTL logic to technology-mapped gates (e.g., AND, OR, MUX, FFs), maps your logic into FPGA primitives (LUTs, DSPs, etc.) and Creates the **Technology Map Viewer (Post-Mapping)**.



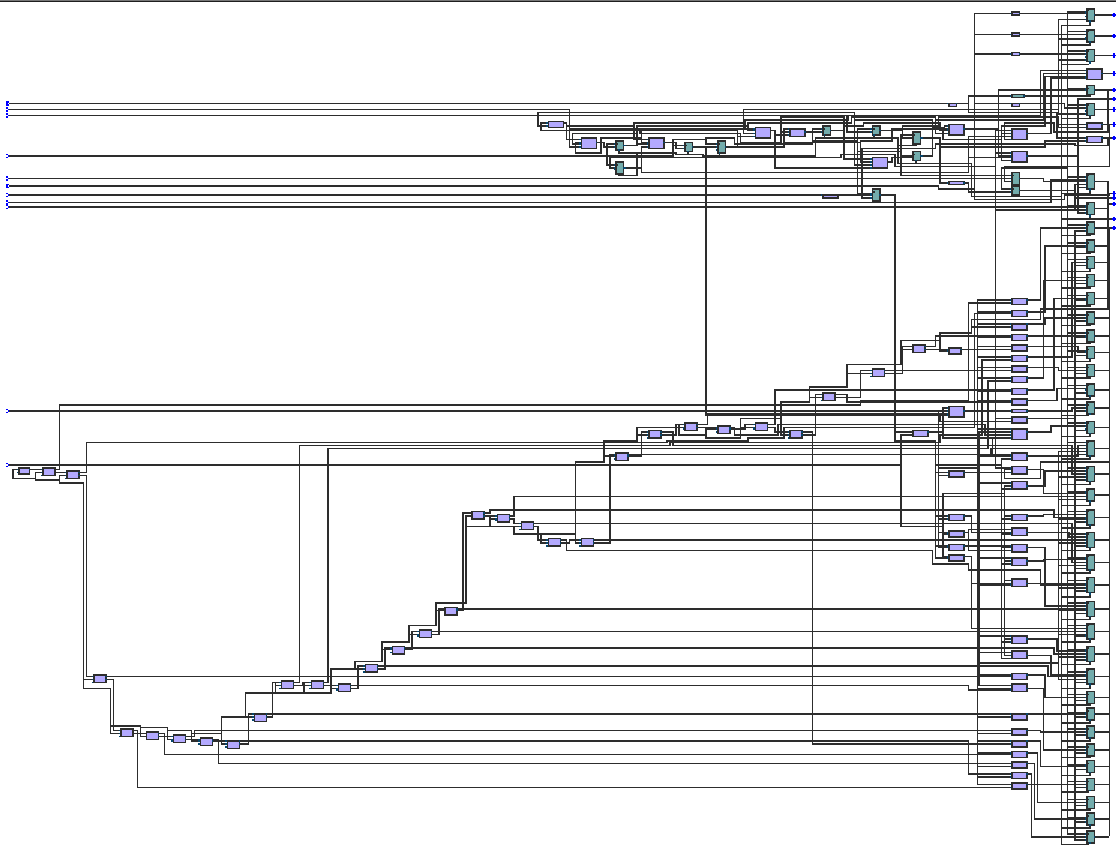
Technology viewer post mapping (Q4.12)



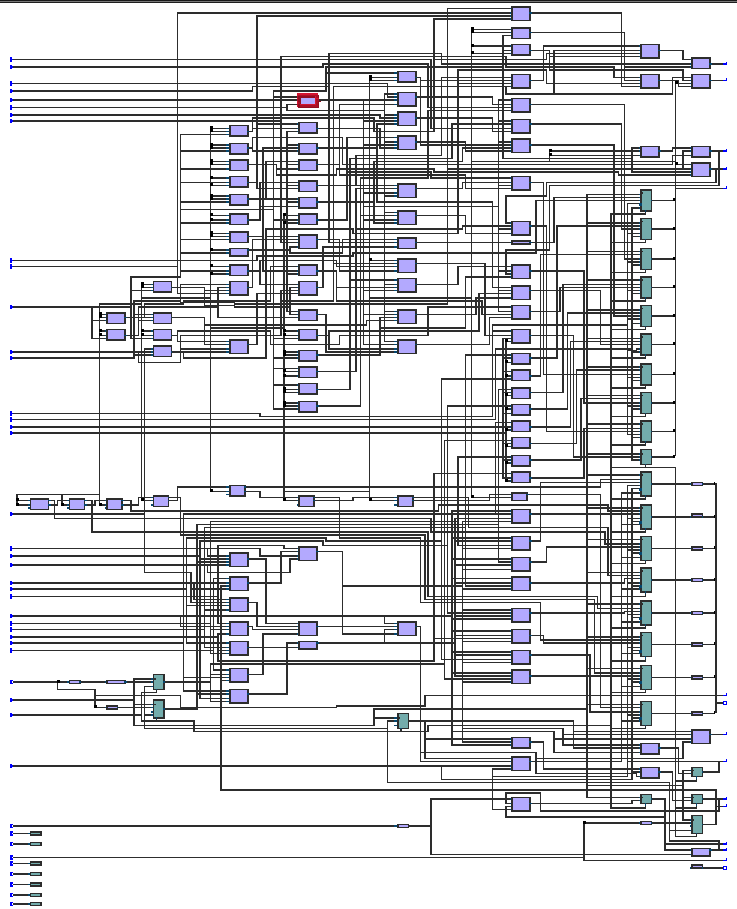
Technology viewer post mapping (Q4.4)

**FITTER (PLACE & ROUTE)**

It assigns logic blocks to actual physical FPGA resources and routes the signals between them. It generates the **Technology Viewer (Post-Fitting)** and **Resource Utilization Summary**.



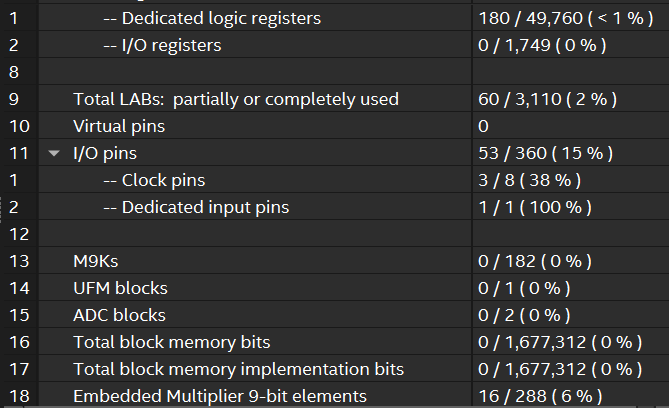
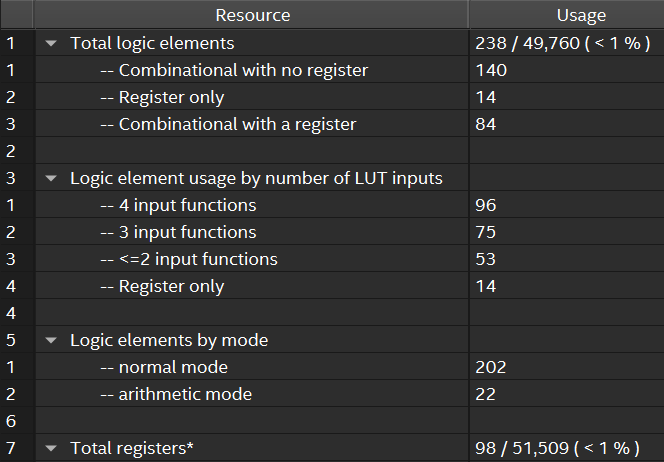
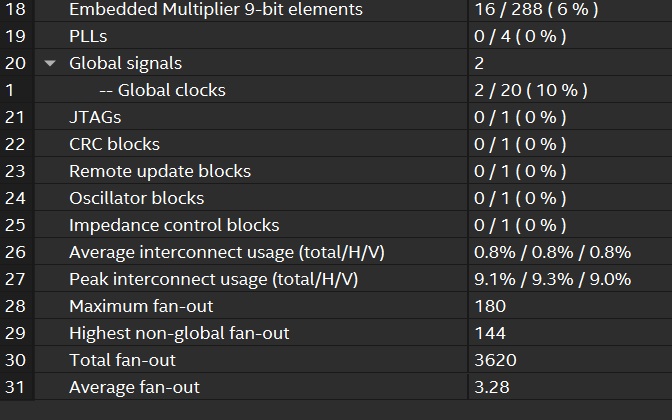
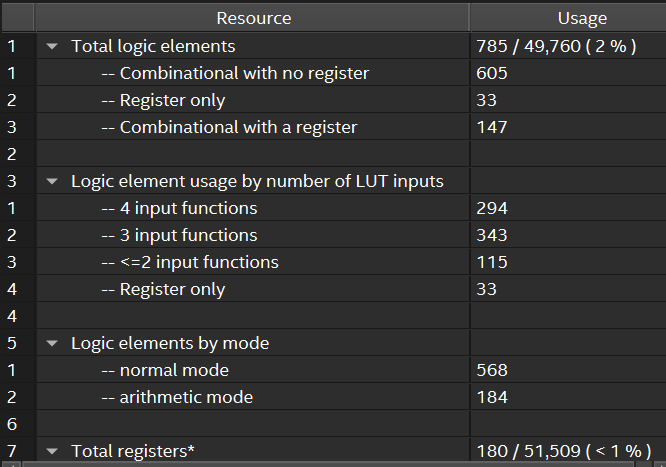
Technology viewer post fitting(Q4.12)

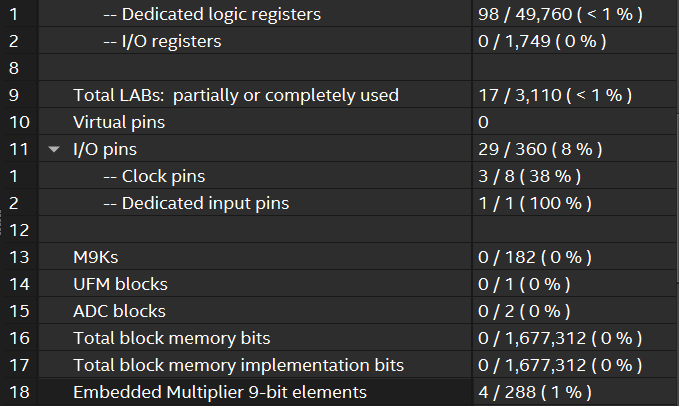
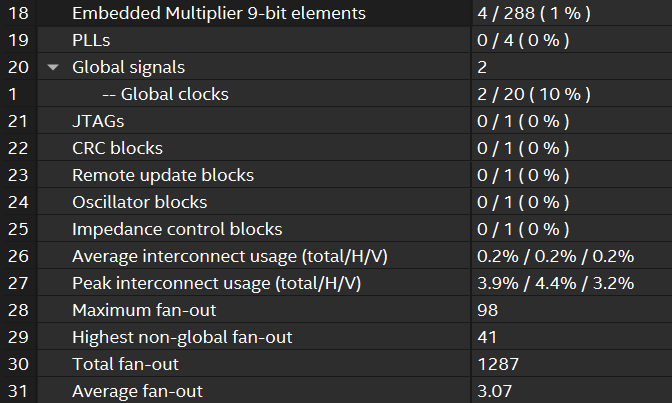


Technology viewer post fitting(Q4.4)

**ASSEMBLER**

Takes the output from the fitter and generates the **.sof** (SRAM Object File) or **.pof** for programming.





Fitter resource summary table (Q4.12) Fitter resource summary table (Q4.4)

**POWER ANALYSER**

Estimates power consumption based on: Post-fitting netlist, Timing information and signal activity (from default or VCD file). The Power Analyser tool in Quartus Prime is used.

  
Results from Power Analyser (Q4.12)



Results from Power Analyser (Q4.4)

**COMPARISON**

We implemented the Gold-Schmidt division algorithm on FPGA using two fixed-point formats: Q4.12 and Q4.4. The Q4.4 version used fewer resources like logic elements and DSPs, and also consumed less power, making it more efficient for simpler designs. However, it had lower precision, so rounding became more important to maintain accuracy. On the other hand, the Q4.12 version gave more accurate results but required more hardware and power. This shows a clear trade-off between precision and resource usage.

**CONCLUSION**

Overall, this phase of the project was focused heavily on understanding the theory behind the algorithms and going forward with the writing of the Verilog code and simulating the testbenches. After that, we prepared our initial area, power and performance report.The literature review gave us both depth and direction, and allowed us to appreciate the importance of choosing the right algorithm not just for mathematical correctness, but also for hardware efficiency. In the subsequent weeks, we hope to add pipelining for improved performance and also expand this to accommodate for signed numbers too. Another possible extension we have in mind is to use the ROMs on the FPGA itself for building the LUTs, rather than using a switch-case, which is what we have currently implemented. We believe implementing these improvements can lead us to have a more resource-efficient implementation in terms of area and power. We will also be trying to extend from just the Q4.4 and Q4.12 fixed point representation to floating point as well. We are grateful to the UNNATI program of INTEL for giving us this opportunity.

**REFERENCES**

* Division algorithms and implementations - S.F. Obermann and Michael J. Flynn  
  <https://www.researchgate.net/publication/3043862_Division_algorithms_and_implementations>
* Verilog HDL Basics - Altera  
  <https://youtu.be/aido2P7Edc4>
* Intel Quartus Prime Software User guides  
  <https://www.intel.com/content/www/us/en/support/programmable/support-resources/design-software/user-guides.html>
* Goldschmidt algorithm  
  <https://lauri.xn--vsandi-pxa.com/hdl/arithmetic/goldschmidt-division-algorithm.html>