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 & SERVICES INDIA PVT LTD

A Practice School-I Station of

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI

(JULY,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

(JULY,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 & SERVICES INDIA 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: 16th July 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 Goldschmidt Algorithm written in Verilog HDL, focusing on Optimization too.

Abstract: This project focuses on implementing a hardware-efficient, signed and pipelined version of division algorithm using Verilog on an FPGA, with particular emphasis on the Goldschmidt method. Given that the Intel CycloneV are 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. Our versions include signed and pipelined Q4.4 and pipelined Q4.12.

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

TABLE OF CONTENTS

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 his 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 CycloneV 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.

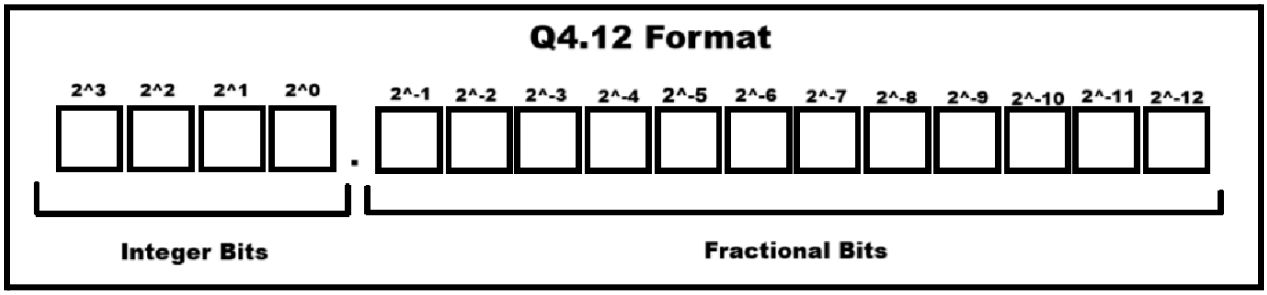
Q-POINT NOTATION

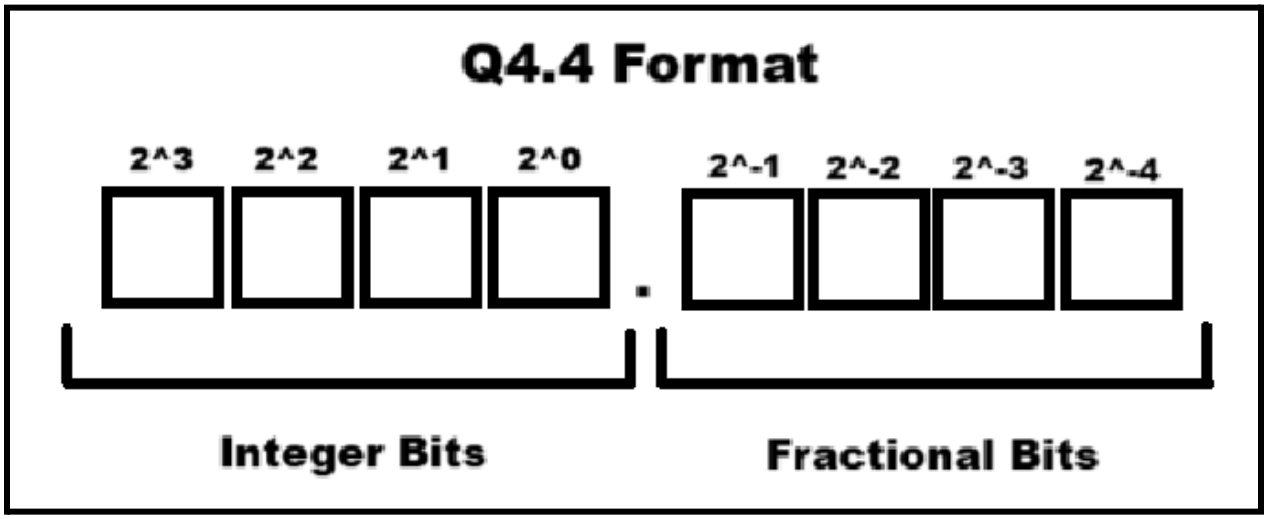
In digital systems, the Q-Point notation is widely used to represent **fixed point** numbers. Fixed point numbers allocate a fixed number of bits for the integer and fractional part, unlike floating point where the range is wide due to the presence of an exponent and mantissa. The Q-point notation provides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

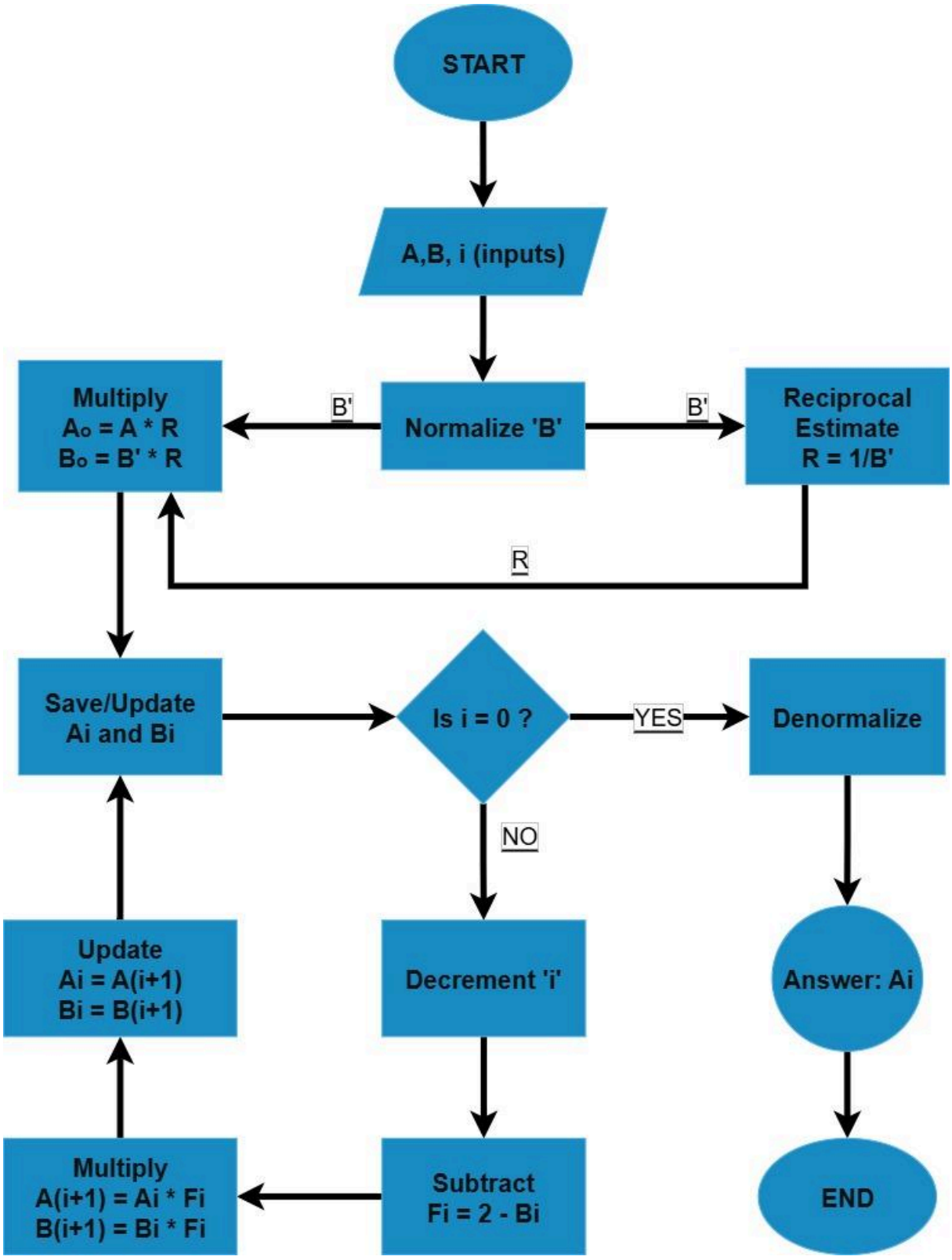
Priority Encoder for Shift Calculation:

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.

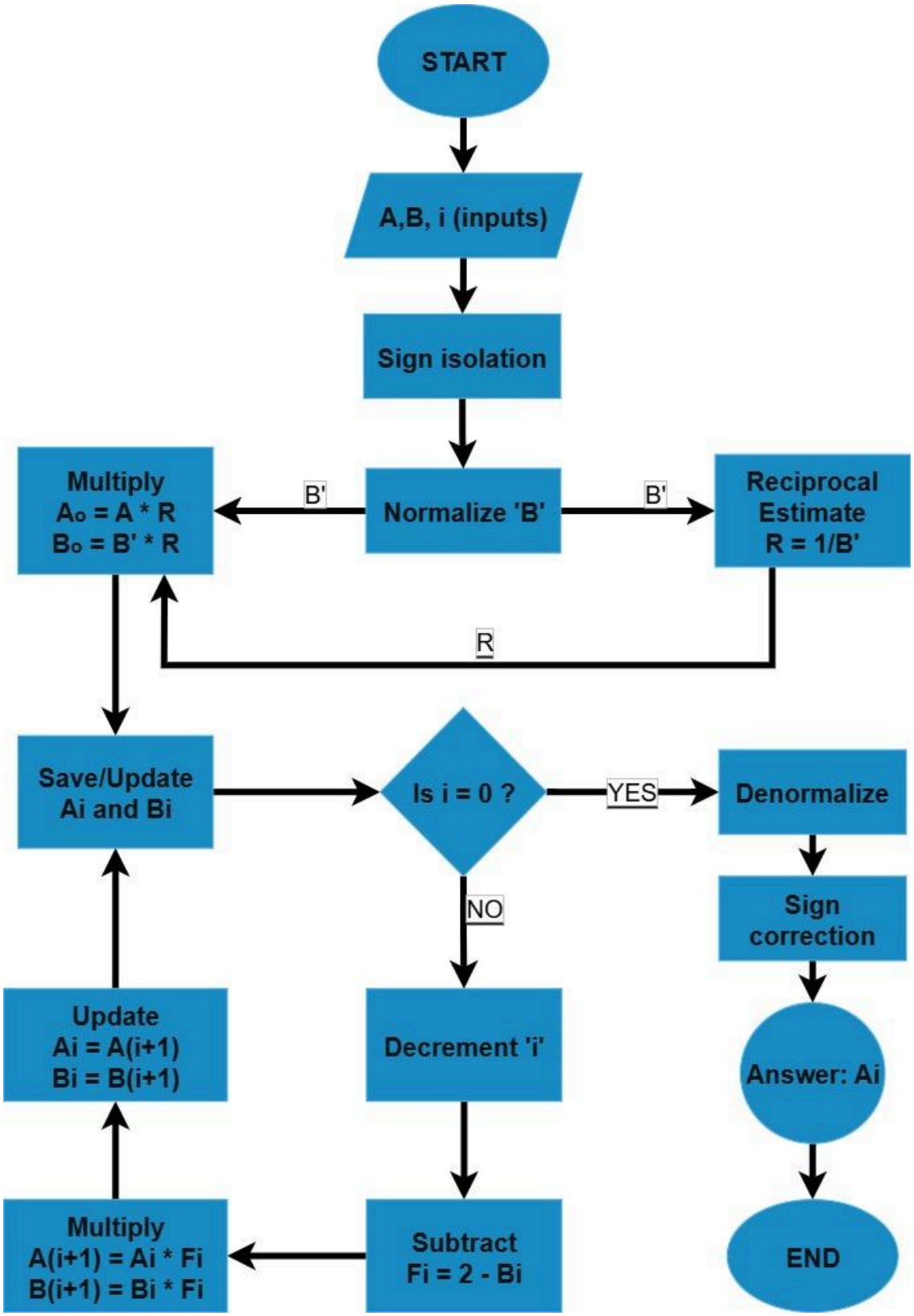
LUT Indexing for Reciprocal Estimation:

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 (UNSIGNED)

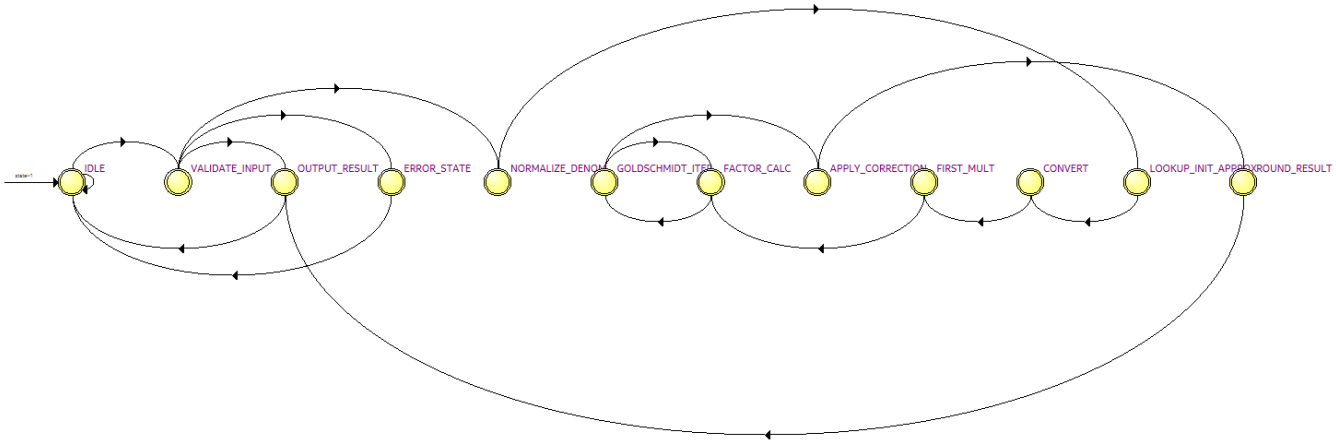


BLOCK DIAGRAM (SIGNED)



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)

	Name	ERROR_STATE	FACTOR_CALC	OUTPUT_RESULT	ROUND_RESULT	APPLY_CORRECTION	GOLDSCHMIDT_ITER	FIRST_MULT	CONVERT	LOOKUP_INIT_APPROX	NORMALIZE_DENOM	VALIDATE_INPUT	IDLE
1	IDLE	0	0	0	0	0	0	0	0	0	0	0	0
2	VALIDATE_INPUT	0	0	0	0	0	0	0	0	0	0	1	1
3	NORMALIZE_DENOM	0	0	0	0	0	0	0	0	0	1	0	1
4	LOOKUP_INIT_APPROX	0	0	0	0	0	0	0	0	1	0	0	1
5	CONVERT	0	0	0	0	0	0	0	1	0	0	0	1
6	FIRST_MULT	0	0	0	0	0	0	1	0	0	0	0	1
7	GOLDSCHMIDT_ITER	0	0	0	0	0	1	0	0	0	0	0	1
8	APPLY_CORRECTION	0	0	0	0	1	0	0	0	0	0	0	1
9	ROUND_RESULT	0	0	0	1	0	0	0	0	0	0	0	1
10	OUTPUT_RESULT	0	0	1	0	0	0	0	0	0	0	0	1
11	FACTOR_CALC	0	1	0	0	0	0	0	0	0	0	0	1
12	ERROR_STATE	1	0	0	0	0	0	0	0	0	0	0	1

STATE TABLE (Transitions):

	Source State	Destination State
1	APPLY_CORRECTION	ROUND_RESULT
2	CONVERT	FIRST_MULT
3	ERROR_STATE	IDLE
4	FACTOR_CALC	GOLDSCHMIDT_ITER
5	FIRST_MULT	FACTOR_CALC
6	GOLDSCHMIDT_ITER	APPLY_CORRECTION
7	GOLDSCHMIDT_ITER	FACTOR_CALC
8	IDLE	IDLE
9	IDLE	VALIDATE_INPUT
10	LOOKUP_INIT_APPROX	CONVERT
11	NORMALIZE_DENOM	LOOKUP_INIT_APPROX
12	OUTPUT_RESULT	IDLE
13	ROUND_RESULT	OUTPUT_RESULT
14	VALIDATE_INPUT	ERROR_STATE
15	VALIDATE_INPUT	NORMALIZE_DENOM
16	VALIDATE_INPUT	OUTPUT_RESULT

SIGNED IMPLEMENTATION OF Q4.4:

Once the unsigned Q4.4 implementation was verified, we shifted focus to handling signed numbers. This required not just updating the data types to signed type but also ensuring that the sign bit was preserved and correctly applied throughout all arithmetic operations. The trickiest part was making sure the result had the correct sign, especially when both the numerator and denominator could independently be positive or negative. This was handled by computing the sign as the XOR of the MSBs of the input operands and then applying it right before the final output stage.

We also had to revisit the rounding logic to make sure it worked correctly in both positive and negative cases. A simple truncation would introduce bias in negative values, so we added conditional logic to round towards zero or away from zero depending on the sign and the discarded bits. Overflow handling was updated to clamp the final result within the signed Q4.4 range, preventing wrap-around. We tested this version extensively using modified testbenches that included both positive and negative inputs and validated them against expected behavior.

PIPELINING : THE RATIONALE AND OUR FINDINGS

Pipelining is generally used to split long computations into smaller, manageable stages,across multiple clock cycles. The idea is to increase the maximum frequency (Fmax) the circuit can safely operate at, by ensuring no single stage has a long critical path that would otherwise limit performance.

In our case, the goal was to maximize Fmax, so we implemented partial pipelining by isolating key stages of computation, particularly the long multiplications inside the FIRST_MULT and GOLDSCHMIDT_ITER states. We introduced internal counters (fmult_count, iter_mult_count) to break down these multiplications across cycles without disrupting the FSM logic. This allowed us to track and complete each operation over multiple clock edges while preserving the state transitions.

However, after completing timing analysis using Quartus, we found that our design was already well-balanced. There were no dominant computational bottlenecks in the critical path, and in fact, the .rpt report showed that the **path with the worst slack involved a logic element with zero computation which means that the delay wasn't due to arithmetic complexity but more due to LUT access** and routing overheads. These lookups were inherent to the algorithm and couldn't be optimized away. Because of this, **pipelining did not significantly benefit us, and in fact, slightly reduced our Fmax**

due to the additional flip-flops and registers that were added. Still, the exercise helped us better understand the structure and bottlenecks in our datapath and gave us a clearer picture of where future optimization efforts could be directed.

```
// Pipelining
reg [1:0] fmult_count;
reg [1:0] iter_mult_count;


FIRST_MULT: begin
     if (fmult_count == 2'd2)
       next_state = FACTOR_CALC;
     else
       next_state = FIRST_MULT;  // Stay in state until done
end
FACTOR_CALC: begin
       ext_state = GOLDSCHMIDT_ITER;
end
GOLDSCHMIDT_ITER: begin
      if (iter_mult_count == 2'd2) begin
         if (iteration_counter == MAX_ITERATIONS - 1)
              next_state = APPLY_CORRECTION;
         else
              next_state = FACTOR_CALC;  // Go to FACTOR_CALC for next iteration
         end else begin
              ext_state = GOLDSCHMIDT_ITER;  // Wait until all 3 pipeline steps done
      end
end
```

```
FIRST_MULT: begin
        case (fmult_count)
        2'd0: begin
                mul_temp_64 <= num_q8_24 * factor_q8_24;
                fmult_count <= 2'd1;
        end
        2'd1: begin
                num_q8_24 <= mul_temp_64[55:24];
                mul_temp_64 <= denom_q8_24 * factor_q8_24;
                fmult_count <= 2'd2;
        end
        2'd2: begin
                denom_q8_24 <= mul_temp_64[55:24];
                fmult_count <= 2'd0;
        end
        endcase
end
GOLDSCHMIDT_ITER: begin
        case (iter_mult_count)
        2'd0: begin
                mul_temp_64 <= num_q8_24 * factor_q8_24;
                iter_mult_count <= 2'd1;
        end
        2'd1: begin
                num_q8_24 <= mul_temp_64[55:24];
                mul_temp_64 <= denom_q8_24 * factor_q8_24;
                iter_mult_count <= 2'd2;
        end
        2'd2: begin
                denom_q8_24 <= mul_temp_64[55:24];
                iter_mult_count <= 2'd0;
                iteration_counter <= iteration_counter + 1;
        end
        endcase
end
```

In our Q4.12 implementation, we introduced pipelining specifically within the multiplication-heavy stages to improve performance and **reduce critical path delays**. This was done in a non-intrusive way by keeping the FSM states the same and simply adding cycle-by-cycle control logic within each state. For both the `FIRST_MULT` and `GOLDSCHMIDT_ITER` states, we used small counters to split the long multiplications into two clock cycles. The FSM combinational logic was updated to stay in the same state until all multiplication steps were completed, and then proceed to the next logical state. This allowed us to reduce the combinational delay within a single cycle, which is typically what limits the maximum clock frequency. The sequential logic within each state handled the actual multiplication pipeline, tracking which stage of the multiplication

was being performed and storing the partial results accordingly. We did not alter the FSM state structure itself, so the logic remained easy to follow. This form of partial pipelining helped us test whether arithmetic stages were the bottleneck, but as mentioned earlier, our final timing analysis showed the delays were mainly due to LUT access and not arithmetic itself.

CODE

The Verilog HDL implementations of the Goldschmidt division algorithm across multiple fixed-point formats are included in the appendices of this report. Specifically, the base versions: Q4.4 unsigned, Q4.4 signed, and Q4.12 unsigned, along with their respective testbenches, are organized into separate appendices for clarity. The pipelined versions of these implementations, which introduce staged processing, are discussed in detail in the Pipelining section of the report.

- Q4.12 Unsigned : Appendix A
- Q4.4 Signed : Appendix B
- Q4.4 Unsigned : Appendix C
- Testbenches : Appendix D

TESTBENCH OUTPUTS

```
# run -all
# Test 1: 2.0/1.0 = 2000 (expected: 2000)
# Test 2: 1.0/2.0 = 0800 (expected: 0800)
# Test 3: 2.0/2.0 = 1000 (expected: 1000)
# Test 4: ERROR - Division by zero
# Test 5: 2.0/7.0 = 0492 (expected: 0492)
# Test 6: 2.75/1.25 = 2333 (expected: 2333)
# ** Note: $finish    : goldschmidt_divider_tb.v(107)
#    Time: 775 ns  Iteration: 0  Instance: /goldschmidt_divider_tb
```

Q4.12 (Unsigned)

```
# run -all
# Test 1: 2.0/1.0 = 20 (expected: 20)
# Test 2: 1.0/2.0 = 08 (expected: 08)
# Test 3: 2.0/2.0 = 10 (expected: 10)
# Test 4: ERROR - Division by zero
# Test 5: 2.0/7.0 = 05 (expected: ~05)
# Test 6: 2.75/1.25 = 23 (expected: ~23)
# ** Note: $finish    : Gold_schmidt_Q44.v(111)
#    Time: 715 ns  Iteration: 0  Instance: /goldschmidt_divider_q4_4_tb
```

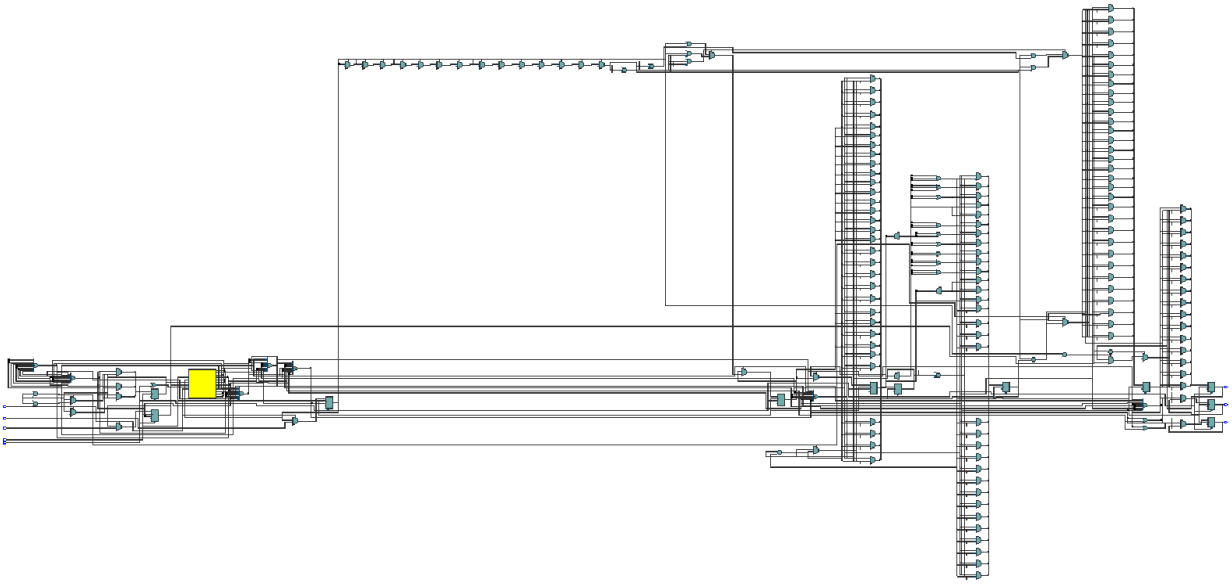
Q4.4 (Unsigned)

```
run 10000ns
Unsigned Test 1: 2.0 / 1.0 = 20 (Expected: 20)
Unsigned Test 2: 1.0 / 2.0 = 08 (Expected: 08)
Unsigned Test 3: 2.0 / 2.0 = 10 (Expected: 10)
Unsigned Test 4: ERROR - Division by zero
Unsigned Test 5: 2.0 / 7.0 = 05 (Expected: ~05)
Unsigned Test 6: 2.75 / 1.25 = 23 (Expected: ~23)
Signed Test 1: -8 / 2 = c0 (Expected: C0)
Signed Test 2: -2 / -2 = 10 (Expected: 10)
Signed Test 3: -1 / 2 = f8 (Expected: F8)
Signed Test 4: -1 / -2 = 08 (Expected: 08)
Signed Test 5: 2 / -1 = e0 (Expected: E0)
** Note: $finish    : goldschmidt_divider_tb.v(95)
   Time: 1395 ns  Iteration: 0  Instance: /goldschmidt_divider_q4_4_tb
```

Q4.4 (Signed)

ANALYSIS AND ELABORATION

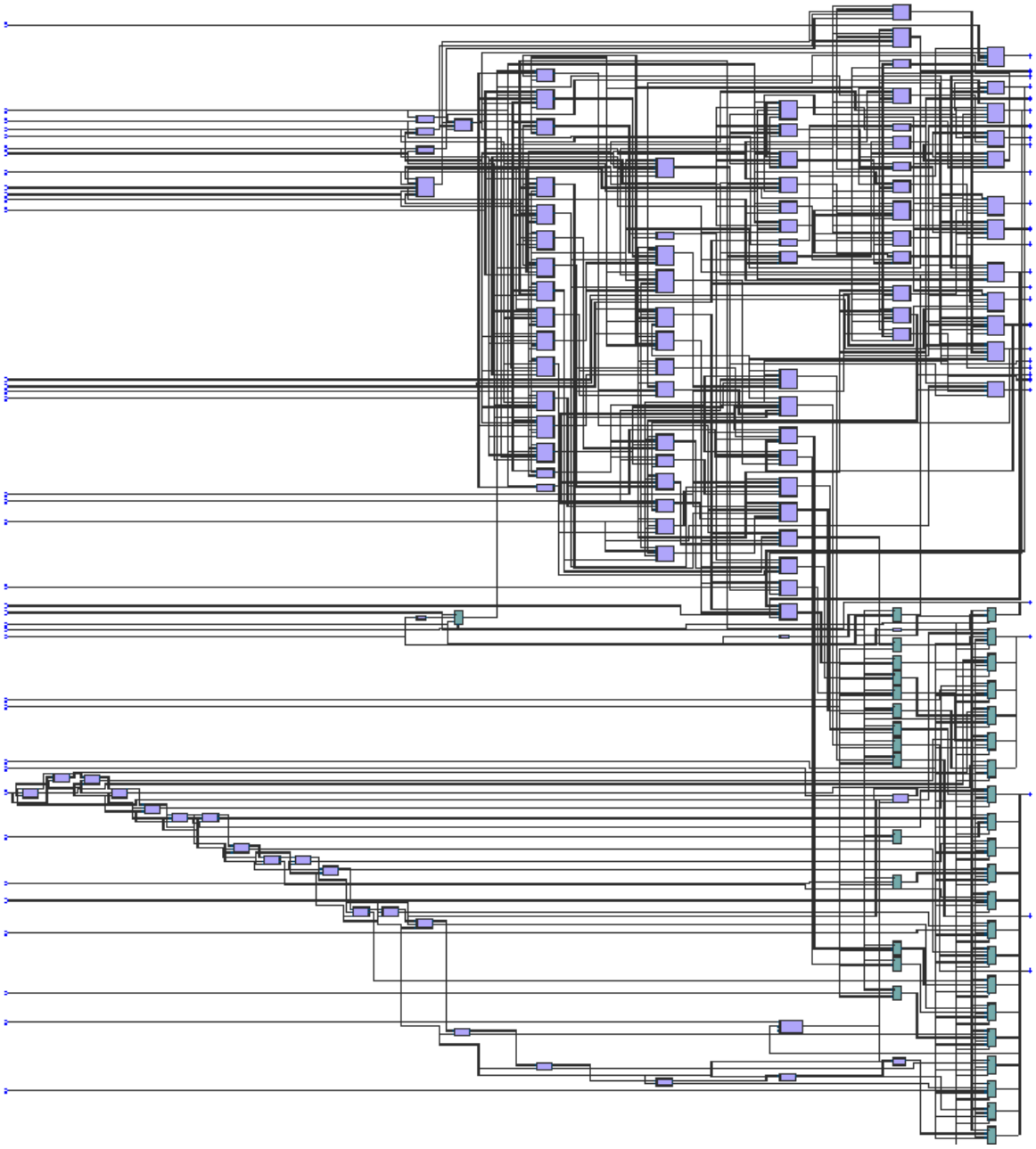
This is the first step before synthesis. It parses through our Verilog/VHDL code and checks syntax and semantics. It builds a hierarchical internal representation of given hardware design. It generates a **RTL (Register transfer level) Viewer** showing how our logic is structured before synthesis.



RTL viewer for Q4.12

ANALYSIS AND SYNTHESIS

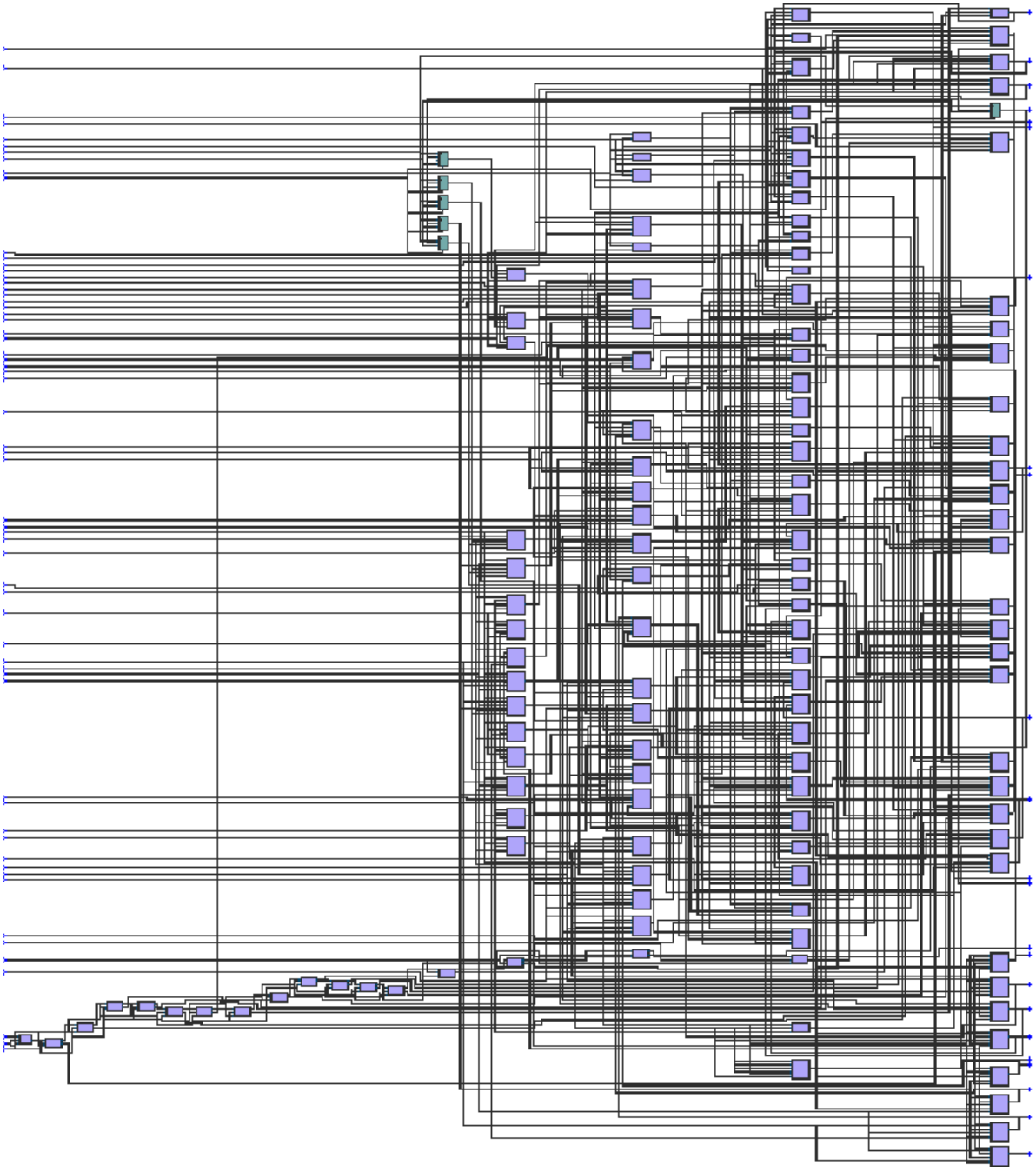
It converts RTL logic to technology-mapped gates (e.g., AND, OR, MUX, FFs), maps your logic into FPGA primitives (LUTs, DSPs, etc.) and creates the **Technology Map Viewer (Post-Mapping)**.



Technology viewer post mapping (Q4.12)

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)

ASSEMBLER

Takes the output from the fitter and generates the **.sof** (SRAM Object File) or **.pof** for programming which was uploaded into the remote FPGA on Labsland.

Resource Summary:

	Resource	Usage	%
1	Logic utilization (ALMs needed / total ALMs on device)	326 / 32,070	1 %
2	▾ ALMs needed [=A-B+C]	326	
1	▾ [A] ALMs used in final placement [=a+b+c+d]	329 / 32,070	1 %
1	[a] ALMs used for LUT logic and registers	69	
2	[b] ALMs used for LUT logic	220	
3	[c] ALMs used for registers	40	
4	[d] ALMs used for memory (up to half of total ALMs)	0	
2	[B] Estimate of ALMs recoverable by dense packing	17 / 32,070	< 1 %
3	▾ [C] Estimate of ALMs unavailable [=a+b+c+d]	14 / 32,070	< 1 %
1	[a] Due to location constrained logic	0	
2	[b] Due to LAB-wide signal conflicts	0	
3	[c] Due to LAB input limits	14	
4	[d] Due to virtual I/Os	0	
3			
4	Difficulty packing design	Low	
5			
6	▾ Total LABs: partially or completely used	42 / 3,207	1 %
1	-- Logic LABs	42	
2	-- Memory LABs (up to half of total LABs)	0	
7			

8	▾ Combinational ALUT usage for logic	481	
1	-- 7 input functions	2	
2	-- 6 input functions	177	
3	-- 5 input functions	84	
4	-- 4 input functions	80	
5	-- <=3 input functions	138	
9	Combinational ALUT usage for route-throughs	31	
10			
11	▾ Dedicated logic registers	248	
1	▾ -- By type:		
1	-- Primary logic registers	218 / 64,140	< 1 %
2	-- Secondary logic registers	30 / 64,140	< 1 %
2	▾ -- By function:		
1	-- Design implementation registers	218	
2	-- Routing optimization registers	30	
12			
13	Virtual pins	0	
14	▾ I/O pins	53 / 457	12 %
1	-- Clock pins	2 / 8	25 %
2	-- Dedicated input pins	0 / 21	0 %
15			

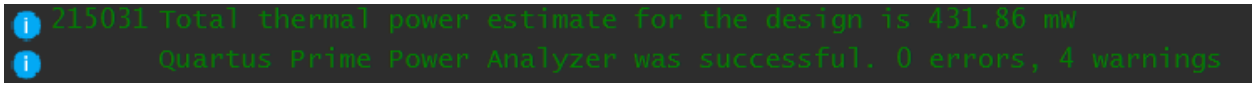
16	▾ Hard processor system peripheral utilization		
1	-- Boot from FPGA	0 / 1 (0 %)	
2	-- Clock resets	0 / 1 (0 %)	
3	-- Cross trigger	0 / 1 (0 %)	
4	-- S2F AXI	0 / 1 (0 %)	
5	-- F2S AXI	0 / 1 (0 %)	
6	-- AXI Lightweight	0 / 1 (0 %)	
7	-- SDRAM	0 / 1 (0 %)	
8	-- Interrupts	0 / 1 (0 %)	
9	-- JTAG	0 / 1 (0 %)	
10	-- Loan I/O	0 / 1 (0 %)	
11	-- MPU event standby	0 / 1 (0 %)	
12	-- MPU general purpose	0 / 1 (0 %)	
13	-- STM event	0 / 1 (0 %)	
14	-- TPIU trace	0 / 1 (0 %)	
15	-- DMA	0 / 1 (0 %)	
16	-- CAN	0 / 2 (0 %)	
17	-- EMAC	0 / 2 (0 %)	
18	-- I2C	0 / 4 (0 %)	
19	-- NAND Flash	0 / 1 (0 %)	
20	-- QSPI	0 / 1 (0 %)	
21	-- SDMMC	0 / 1 (0 %)	
22	-- SPI Master	0 / 2 (0 %)	
23	-- SPI Slave	0 / 2 (0 %)	

23	-- SPI Slave	0 / 2 (0 %)	
24	-- UART	0 / 2 (0 %)	
25	-- USB	0 / 2 (0 %)	
17			
18	M10K blocks	0 / 397	0 %
19	Total MLAB memory bits	0	
20	Total block memory bits	0 / 4,065,280	0 %
21	Total block memory implementation bits	0 / 4,065,280	0 %
22			
23	Total DSP Blocks	6 / 87	7 %
24			
25	Fractional PLLs	0 / 6	0 %
26	▾ Global signals	2	
1	-- Global clocks	2 / 16	13 %
2	-- Quadrant clocks	0 / 66	0 %
3	-- Horizontal periphery clocks	0 / 18	0 %
27	SERDES Transmitters	0 / 100	0 %
28	SERDES Receivers	0 / 100	0 %
29	JTAGs	0 / 1	0 %
30	ASMI blocks	0 / 1	0 %
31	CRC blocks	0 / 1	0 %
32	Remote update blocks	0 / 1	0 %
33	Oscillator blocks	0 / 1	0 %
34	Impedance control blocks	0 / 4	0 %

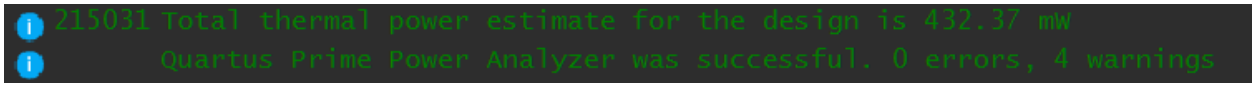
34	Impedance control blocks	0 / 4	0 %
35	Hard Memory Controllers	0 / 2	0 %
36	Average interconnect usage (total/H/V)	0.4% / 0.3% / 0.5%	
37	Peak interconnect usage (total/H/V)	11.7% / 10.8% / 14.9%	
38	Maximum fan-out	250	
39	Highest non-global fan-out	84	
40	Total fan-out	3645	
41	Average fan-out	4.17	

POWER ANALYSER

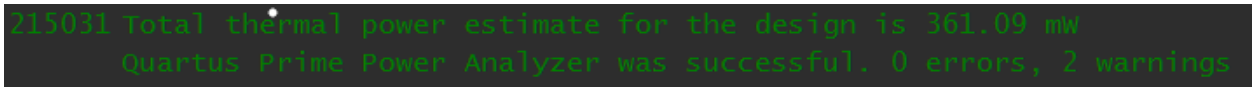
Estimates power consumption based on: Post-fitting netlist, Timing information and signal activity (from default or VCD file). The Power Analyser tool in Quartus Prime is used.



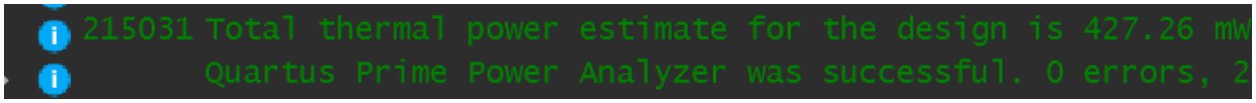
Results from Power Analyser (Q4.12 - Unsigned - Unpipelined)



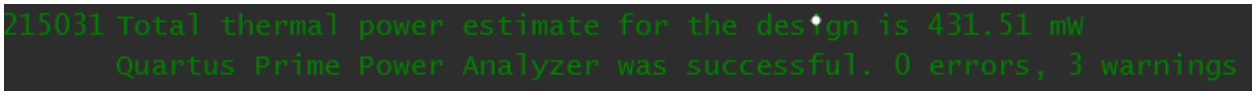
Results from Power Analyser (Q4.12 - Unsigned - Pipelined)



Results from Power Analyser (Q4.4- Unsigned)



Results from Power Analyser (Q4.4 - Signed)



Results from Power Analyser (Q4.4 - Signed and Pipelined)

TIMING ANALYSIS

Static Timing Analysis (STA) is a method used to evaluate the timing performance of a digital circuit without requiring simulation or input vectors. It checks all possible timing paths in a design to ensure signals propagate within required time constraints, such as setup and hold times. In Quartus, the Timing Analyzer tool performs STA by analyzing the delays through logic elements, routing, and clock domains. It helps identify violations (like setup or hold failures), ensuring reliable operation across all conditions (process, voltage, temperature).

During static timing analysis, the Timing Analyzer checks every possible path between registers to ensure signals arrive within the required time windows. When these conditions aren't met, it reports timing violations. A setup violation occurs when data arrives too late at a flip-flop before the clock edge, potentially causing incorrect values to be latched. A hold violation, on the other hand, happens when data changes too soon after the clock edge, risking instability as the flip-flop may capture changing data. These violations highlight critical paths in the design that limit the maximum frequency (Fmax). Resolving all violations is essential to achieve timing closure and ensure that the circuit functions reliably under worst-case conditions.

In Quartus, the .sdc (Synopsys Design Constraints) file is used to define timing constraints for the Timing Analyzer during Static Timing Analysis (STA). It includes commands to specify clocks (create_clock), input/output delays, false paths, multicycle paths, and other timing exceptions. The .sdc file guides the Timing Analyzer on how to interpret timing requirements, ensuring accurate analysis of the circuit's performance. Writing a proper .sdc file is essential for achieving timing closure, optimizing critical paths, and ensuring the design functions reliably on hardware under all operating conditions.

.sdc File used:

```
# Define the main clock
create_clock -name clk -period <time in ns> [get_ports clk]
set_input_delay <time in ns> -clock clk [get_ports start]
set_input_delay <time in ns> -clock clk [get_ports {numerator[*]}]
set_input_delay<time in ns> -clock clk [get_ports {denominator[*]}]
#output delay for output signals
set_output_delay <time in ns> -clock clk [get_ports {quotient[*]}]
set_output_delay <time in ns> -clock clk [get_ports valid]
set_output_delay <time in ns> -clock clk [get_ports error]
# reset is asynchronous
set_false_path -from [get_ports rst_n]
```

Finding FMAX:

To determine the maximum operating frequency (Fmax) of our implementation, we used the Timing Analyzer in Quartus. We first added appropriate timing constraints in the .sdc file, specifying the expected clock period. Through several iterations, we adjusted these constraints and recompiled the design to observe how Quartus optimized the circuit and how the reported Fmax changed. During this process, we monitored for setup and hold

violations, which indicate that the design cannot reliably operate at the specified frequency. This trial-and-error approach helped us better understand the timing limitations of our design and guided us toward achieving timing closure. The final Fmax was obtained from the Fmax Summary under the worst-case timing model in the Timing Analyzer report.

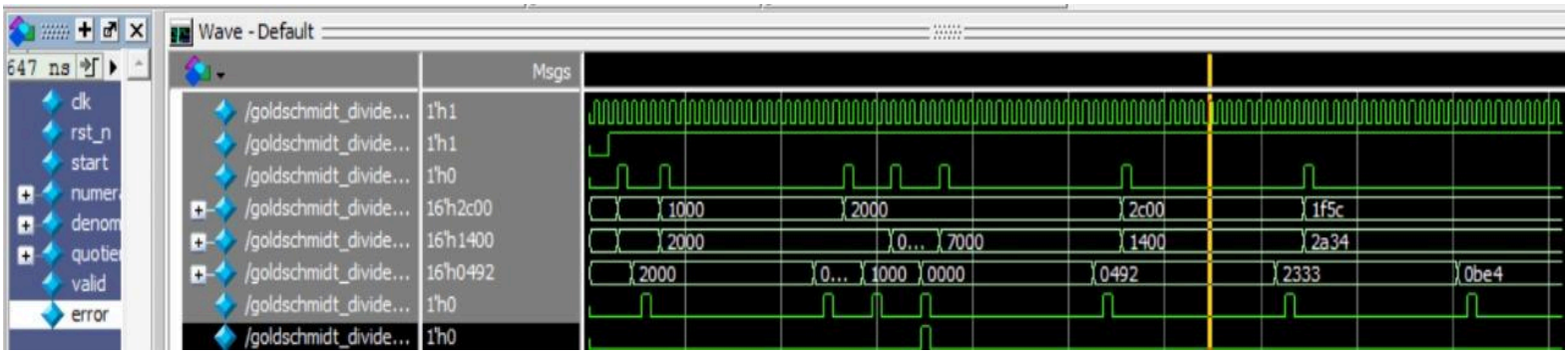
Unsigned Q4.4: 79.12 MHz

Signed Q4.4: 79.55 MHZ

Unsigned Q4.12: 70.09MHZ

Unsigned Pipelined Q4.12: 67MHZ

Signed Pipelined Q4.4: 79.24MHZ



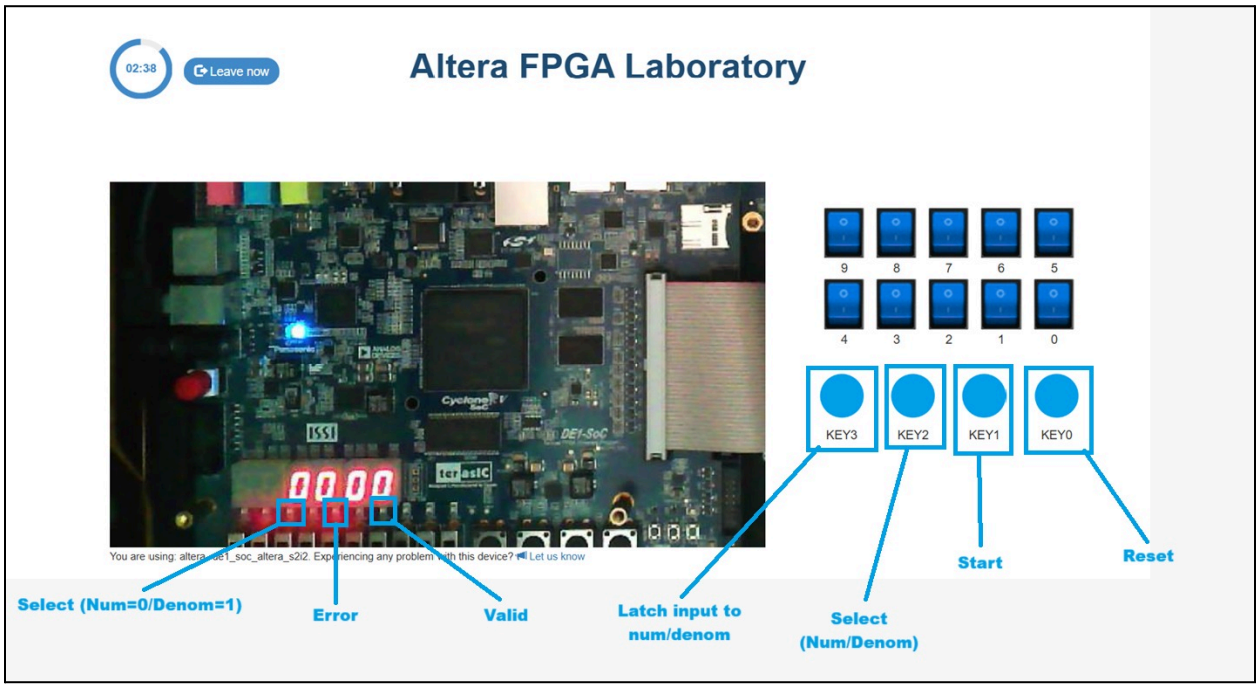
Waveform obtained for Q4.12 unsigned version

LABSLAND IMPLEMENTATION AND PIN PLANNING

Limitations:

Since we were remotely accessing the FPGA through the LabsLand platform, we were limited to using the available switches, push buttons, LEDs, and 7 segment displays on the De1-SoC to verify our verilog code. Tools like SignalTap, UART, and custom external peripherals were unavailable. Therefore, we designed a minimal but complete system that could be **fully operated using the board's physical inputs and outputs.**

LabsLand Setup:



To run and verify our Goldschmidt division implementation, we deployed our design on the Intel DE1-SoC FPGA board remotely using LabsLand. LabsLand provided a web interface that allowed us to interact with the actual board using its physical component. We used the following components:

- **Slide Switches (SW[9:0]):**
  Used to input 9-bit fixed-point values in Q4.12 format.
  - SW[9:6] → 4 bits of the integer part
  - SW[5:0] → Middle 3 bits of the fractional part
- **Push Buttons (KEY[3:0]):**
  Assigned for controlling the state machine and data input:
  - KEY[0] → Reset (active-low): resets internal registers and FSM
  - KEY[1] → Start pulse: triggers the division operation
  - KEY[2] → Select toggle: toggles between inputting numerator (0) and denominator (1)
  - KEY[3] → Set/Latch pulse: latches the current switch value into either numerator or denominator depending on the select state
- **Red LEDs (LEDR[9:0]):**
  Provided real-time visual feedback to monitor internal control and output signals:
  - LEDR[1] → High when a start pulse is registered
  - LEDR[3] → High when a valid output is generated
  - LEDR[5] → High when a division error occurs (e.g., divide by zero)
  - LEDR[7] → Shows the current select state (0 = numerator, 1 = denominator)
  - LEDR[9] → High when a value is latched from the switches
- **Seven-Segment Displays (HEX0 to HEX3):**
  Displayed the final 16-bit quotient in hexadecimal.

Pin Planning:

Once the logic was finalized, we performed pin planning in Intel Quartus Prime to map all signals to correct physical pins on the De1-SoC board. This step is essential as it ensures that logical ports are physically connected to the corresponding buttons, switches, LEDs, etc.

CLOCK_50	PIN_AF14
HEX0[6]	PIN_AH28
HEX0[5]	PIN_AG28
HEX0[4]	PIN_AF28
HEX0[3]	PIN_AG27
HEX0[2]	PIN_AE28
HEX0[1]	PIN_AE27
HEX0[0]	PIN_AE26
HEX1[6]	PIN_AD27
HEX1[5]	PIN_AF30
HEX1[4]	PIN_AF29
HEX1[3]	PIN_AG30
HEX1[2]	PIN_AH30
HEX1[1]	PIN_AH29
HEX1[0]	PIN_AJ29

HEX2[6]	PIN_AC30
HEX2[5]	PIN_AC29
HEX2[4]	PIN_AD30
HEX2[3]	PIN_AC28
HEX2[2]	PIN_AD29
HEX2[1]	PIN_AE29
HEX2[0]	PIN_AB23
HEX3[6]	PIN_AB22
HEX3[5]	PIN_AB25
HEX3[4]	PIN_AB28
HEX3[3]	PIN_AC25
HEX3[2]	PIN_AD25
HEX3[1]	PIN_AC27
HEX3[0]	PIN_AD26

LEDR[9]	PIN_Y21
LEDR[8]	PIN_W21
LEDR[7]	PIN_W20
LEDR[6]	PIN_Y19
LEDR[5]	PIN_W19
LEDR[4]	PIN_W17
LEDR[3]	PIN_V18
LEDR[2]	PIN_V17
LEDR[1]	PIN_W16
LEDR[0]	PIN_V16

SW[9]	PIN_AF16
SW[8]	PIN_AE16
SW[7]	PIN_AG16
SW[6]	PIN_AH17
SW[5]	PIN_AH18
SW[4]	PIN_AJ16
SW[3]	PIN_AJ17
SW[2]	PIN_AJ19
SW[1]	PIN_AK19
SW[0]	PIN_AK18
KEY[3]	PIN_Y18
KEY[2]	PIN_AD17
KEY[1]	PIN_Y17
KEY[0]	PIN_AC18

Top wrapper module:

To handle input formatting, button toggling, and output display, all with limited I/O we created a top-level wrapper module, top_goldschmidt_fpga, which connected the core goldschmidt module to physical inputs and outputs.

```
module top_goldschmidt_fpga (
    input wire [9:0] SW,
    input wire [3:0] KEY,
    output wire [9:0] LEDR,
    output wire [6:0] HEX0, HEX1, HEX2, HEX3,
    input wire CLOCK_50
);

wire clk = CLOCK_50;
wire rst_n = KEY[0]; // active-low reset
wire start_btn = ~KEY[1];
wire toggle_select_btn = ~KEY[2];
wire latch_btn = ~KEY[3];

reg select = 0;   // 0 = numerator, 1 = denominator
reg toggle_prev = 0;
reg latch_prev = 0;
reg start_prev = 0;

reg start = 0;
```

```
reg set_val = 0;

always @(posedge clk) begin
    // Toggle logic for select
    if (!rst_n) begin
        select <= 0;
    end else if (toggle_select_btn && !toggle_prev) begin
        select <= ~select;
    end
    toggle_prev <= toggle_select_btn;

    // Pulse logic for set_val
    if (!rst_n) begin
        set_val <= 0;
    end else begin
        set_val <= latch_btn && !latch_prev;
    end
    latch_prev <= latch_btn;

    // Pulse logic for start
    if (!rst_n) begin
        start <= 0;
    end else begin
        start <= start_btn && !start_prev;
    end
    start_prev <= start_btn;
end

// Inputs from SW
wire [15:0] user_input_q412 = {1'b0, SW[9:6], SW[5:0], 6'b0};

reg [15:0] numerator = 16'h0000;
reg [15:0] denominator = 16'h0000;

// Latch input into numerator or denominator
always @(posedge clk) begin
    if (!rst_n) begin
        numerator <= 16'h0000;
        denominator <= 16'h0000;
    end else if (set_val) begin
        if (select == 1'b0) begin
            numerator <= user_input_q412;
        end else begin
            denominator <= user_input_q412;
        end
    end
end

// --- Output
wire [15:0] quotient;
wire valid;
wire error;
```

```
reg latched_valid = 0;
reg latched_error = 0;

always @(posedge clk) begin
    if (!rst_n || start) begin
        latched_valid <= 0;
        latched_error <= 0;
    end else begin
        if (valid) latched_valid <= 1;
        if (error) latched_error <= 1;
    end
end

// Instantiate Goldschmidt divider
goldschmidt goldschmidt_inst (
    .clk(clk),
    .rst_n(rst_n),
    .start(start),
    .numerator(numerator),
    .denominator(denominator),
    .quotient(quotient),
    .valid(valid),
    .error(error)
);

// LED Display
assign LEDR[1] = start;
assign LEDR[3] = latched_valid;
assign LEDR[5] = latched_error;
assign LEDR[7] = select;   // 0 = numerator, 1 = denominator
assign LEDR[9] = set_val;
// assign LEDR[9:5] = quotient[15:11]; // Top 5 bits of output

// HEX Displays (show Q4.12 output)
seven_seg seg0(.bin(quotient[3:0]), .seg(HEX0));
seven_seg seg1(.bin(quotient[7:4]), .seg(HEX1));
seven_seg seg2(.bin(quotient[11:8]), .seg(HEX2));
seven_seg seg3(.bin(quotient[15:12]), .seg(HEX3));

endmodule
```

```
module seven_seg(
    input wire [3:0] bin,
    output reg [6:0] seg
);
// seg = {a, b, c, d, e, f, g}
always @(*) begin
    case (bin)
        4'h0: seg = 7'b1000000;
        4'h1: seg = 7'b1111001;
        4'h2: seg = 7'b0100100;
        4'h3: seg = 7'b0110000;
        4'h4: seg = 7'b0011001;
        4'h5: seg = 7'b0010010;
        4'h6: seg = 7'b0000010;
        4'h7: seg = 7'b1111000;
        4'h8: seg = 7'b0000000;
        4'h9: seg = 7'b0010000;
        4'hA: seg = 7'b0001000;
        4'hB: seg = 7'b0000011;
        4'hC: seg = 7'b1000110;
        4'hD: seg = 7'b0100001;
        4'hE: seg = 7'b0000110;
        4'hF: seg = 7'b0001110;
        default: seg = 7'b1111111;  // All segments off
    endcase
end
endmodule
```

LabsLand Video Demo:

https://drive.google.com/file/d/1BNh7L42GwVNGSPVOCyDumaqJh5-G90LX/view?usp=sharing

COMPARISON

In the beginning, the Q4.4 version worked well because it used fewer resources and less power. It was simpler and more efficient, especially for basic use cases. But since it had lower precision, we had to be a bit more careful with rounding to make sure the results stayed accurate. The Q4.12 version, on the other hand, gave much better accuracy but used more logic and DSP blocks, and it needed more power too, so clearly there was a trade-off between how precise we wanted the output and how much hardware we were willing to use.

After our midsem, we added support for signed numbers and also tried pipelining. The signed Q4.4 version actually gave a slightly better max frequency compared to the unsigned one, it went up from 79.12 MHz to 79.55 MHz. The unsigned Q4.12 version ran at 70.09 MHz, but once we pipelined it to split up the multiplication across cycles, the frequency dropped a bit to 67 MHz. That was because of the extra flip-flops and registers we added, which slightly slowed things down. In the case of signed pipelined Q4.4, we still got a solid 79.24 MHz, which showed that pipelining didn't make a huge difference when the design was already well-balanced.

Overall, we realised that improving performance isn't just about making operations faster. It also depends on how the design is routed and how well it is balanced. The timing analysis made that clearer to us, and these results helped us understand where pipelining is useful and where it might not give us much gain.

CONCLUSION

Over the course of this project, we moved beyond just the theoretical understanding of Goldschmidt's algorithm to focus on the actual hardware realization using Verilog. We implemented and verified both unsigned and signed fixed-point pipelined versions in Q4.4 format, as well as a pipelined Q4.12 version to improve throughput. Rather than stopping at functional correctness, we paid particular attention to rounding behavior and overflow handling, especially in the signed implementation, where these aspects required careful design.

Floating-point support was initially considered, but given the time and complexity constraints, we decided to treat it as a potential future direction. Another significant aspect of the project was the effort we put into timing analysis. We explored the paths thoroughly, ensured that there were no violations in our design, and tried optimized critical paths to operate reliably at the desired clock frequency.

As part of the final deployment, we used LabsLand to carry out hardware-level testing and verified the correctness of outputs through physical interfacing. This helped reinforce our understanding of how a design that looks good on simulation may still need attention when mapped onto actual hardware.

Overall, this project helped us bridge the gap between algorithm-level thinking and real-world hardware constraints. It gave us a better appreciation of what it takes to design for area, power, and performance simultaneously, and we hope to build on this foundation in future work. We are grateful to Intel’s UNNATI program for giving us this opportunity.

Appendix A (Unsigned Q4.12 Unpipelined)

```
module goldschmidt_divider_412_unpip (
    input wire clk,
    input wire rst_n,
    input wire start,
    input wire [15:0] numerator,      // Q4.12 format
    input wire [15:0] denominator,    // Q4.12 format
    output reg [15:0] quotient,       // Q4.12 format
    output reg valid,
    output reg error
);

// Alternative synthesizable CLZ function (add this before the module)
reg [4:0] count;
reg [15:0]temp;

localparam [3:0]
    IDLE              = 4'h0,
    VALIDATE_INPUT    = 4'h1,
    NORMALIZE_DENOM   = 4'h2,
    LOOKUP_INIT_APPROX= 4'h3,
    CONVERT           = 4'h4,
    WAIT_MULT_INPUTS  = 4'hA,
    FACTOR_CALC       = 4'hB,
    FIRST_MULT        = 4'h5,
    GOLDSCHMIDT_ITER  = 4'h6,
    APPLY_CORRECTION  = 4'h7,
    ROUND_RESULT      = 4'h8,
    OUTPUT_RESULT     = 4'h9,
    ERROR_STATE       = 4'hF;

// Constants
localparam [31:0] Q8_24_ONE = 32'h01000000;  // 1.0 in Q8.24
localparam [31:0] Q8_24_TWO = 32'h02000000;  // 2.0 in Q8.24
localparam [15:0] Q4_12_ONE = 16'h1000;      // 1.0 in Q4.12
localparam [15:0] Q4_12_HALF = 16'h0800;     // 0.5 in Q4.12
localparam MAX_ITERATIONS = 3;

// State machine registers
reg [3:0] state, next_state;
reg [2:0] iteration_counter;
reg [4:0] shift_amount;

// Working registers (Q8.24 format for intermediate calculations)
reg [31:0] num_q8_24;        // Numerator in Q8.24
reg [31:0] denom_q8_24;      // Denominator in Q8.24
reg [31:0] factor_q8_24;     // Factor (2 - d*x) in Q8.24
reg [64:0] mul_temp_64;        // Temporary multiplication result

// Input capture registers
```

```
reg [15:0] num_reg, denom_reg;
reg[15:0] denom_norm_reg;
reg[4:0]p;// internal: position of MSB in Denom
reg signed[5:0] shift;
reg[15:0] factor_0;

 always @(*) begin
    // Priority encoder to find index p of highest 1 in Di
    if      (denom_reg[15]) p = 15;
    else if (denom_reg[14]) p = 14;
    else if (denom_reg[13]) p = 13;
    else if (denom_reg[12]) p = 12;
    else if (denom_reg[11]) p = 11;
    else if (denom_reg[10]) p = 10;
    else if (denom_reg[9 ]) p = 9;
    else if (denom_reg[8 ]) p = 8;
    else if (denom_reg[7 ]) p = 7;
    else if (denom_reg[6 ]) p = 6;
    else if (denom_reg[5 ]) p = 5;
    else if (denom_reg[4 ]) p = 4;
    else if (denom_reg[3 ]) p = 3;
    else if (denom_reg[2 ]) p = 2;
    else if (denom_reg[1 ]) p = 1;
    else if (denom_reg[0 ]) p = 0;
    else              p = 0;

        shift = 11 - $signed(p);

       end

// Lookup table for initial approximation (256 entries)
// Pre-computed 1/x values for x in range [0.5, 1.0) in Q8.24 format
//reg [31:0] lookup_table [0:255];
reg[2:0]index;

always @(*) begin
    case (index)
      3'd0: factor_0 = 16'd8192;  // 1/0.5000 = 2.0000
      3'd1: factor_0 = 16'd7282;  // 1/0.5625 = 1.7778
      3'd2: factor_0 = 16'd6554;  // 1/0.6250 = 1.6000
      3'd3: factor_0 = 16'd5958;  // 1/0.6875 = 1.4545
      3'd4: factor_0 = 16'd5461;  // 1/0.7500 = 1.3333
      3'd5: factor_0 = 16'd5041;  // 1/0.8125 = 1.2308
      3'd6: factor_0 = 16'd4681;  // 1/0.8750 = 1.1429
      3'd7: factor_0 = 16'd4369;  // 1/0.9375 = 1.0667
      default: factor_0= 16'd0;
    endcase
  end

// State machine - combinational next state logic
always @(*) begin
    next_state = state;
```

```
    case (state)
        IDLE: begin
            if (start)
                next_state = VALIDATE_INPUT;
        end

        VALIDATE_INPUT: begin
            if (denom_reg == 16'h0000)  // Division by zero
                next_state = ERROR_STATE;
            else if (num_reg == 16'h0000)  // Zero numerator
                next_state = OUTPUT_RESULT;
                            else if (denom_reg==16'h1000)
                                            next_state= OUTPUT_RESULT;
            else
                next_state = NORMALIZE_DENOM;
        end

        NORMALIZE_DENOM: begin
            next_state = LOOKUP_INIT_APPROX;
        end

        LOOKUP_INIT_APPROX: begin
            next_state = CONVERT;
        end

        CONVERT: begin
        next_state = WAIT_MULT_INPUTS; // Wait one cycle for assignments to settle
                 end

                 WAIT_MULT_INPUTS: begin
                 next_state = FIRST_MULT; // Now do multiplication
                 end

                 FIRST_MULT: begin
            next_state = FACTOR_CALC;
        end

                 FACTOR_CALC: begin
                             next_state = GOLDSCHMIDT_ITER;
                 end

        GOLDSCHMIDT_ITER: begin
            if (iteration_counter >= MAX_ITERATIONS)
                next_state = APPLY_CORRECTION;
                            else
                                    next_state = FACTOR_CALC;
            // else stay in same state for next iteration
        end

        APPLY_CORRECTION: begin
            next_state = ROUND_RESULT;
        end
```

```
        ROUND_RESULT: begin
            next_state = OUTPUT_RESULT;
        end

        OUTPUT_RESULT: begin
            next_state = IDLE;
        end

        ERROR_STATE: begin
            next_state = IDLE;
        end

        default: next_state = IDLE;
    endcase
end

// State machine - sequential logic
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        state <= IDLE;
        valid <= 1'b0;
        error <= 1'b0;
        quotient <= 16'h0000;
        iteration_counter <= 3'b000;
        shift_amount <= 5'b00000;
        num_reg <= 16'h0000;
        denom_reg <= 16'h0000;
        num_q8_24 <= 32'h00000000;
        denom_q8_24 <= 32'h00000000;
        factor_q8_24 <= 32'h00000000;
        mul_temp_64 <= 64'h00000000;
    end else begin
        state <= next_state;

        case (state)
            IDLE: begin
                valid <= 1'b0;
                error <= 1'b0;
                if (start) begin
                    num_reg <= numerator;
                    denom_reg <= denominator;
                    iteration_counter <= 3'b000;
                end
            end
            VALIDATE_INPUT: begin
                // Input validation happens in combinational logic
                if (num_reg == 16'h0000)
                    quotient <= 16'h0000;  // 0/x = 0
                 else if (denom_reg==16'h1000)
                      quotient <= num_reg;
            end
```

```
            NORMALIZE_DENOM: begin
                if (shift >= 0) begin
                      // left shift
                      denom_norm_reg <= denom_reg << shift;
                 end else begin
                      // right shift by -shift
                      denom_norm_reg <= denom_reg >> -shift;
                 end
            end
            LOOKUP_INIT_APPROX: begin
                            $display("Normalized denominator: %h",denom_norm_reg);
                // Extract index from normalized denominator
                // Use upper bits for lookup table index
                index <= denom_norm_reg[11:8] - 4'd8;
            end

            CONVERT: begin//Convert to Q8.24
            denom_q8_24 <= {denom_norm_reg, 12'b0}; // Convert Q4.12 → Q8.24
            num_q8_24 <= {num_reg, 12'b0};
            factor_q8_24 <= factor_0 << 12;  // Q4.12 to Q8.24
            $display("Initial reciprocal: %h",factor_0);
            end

            FIRST_MULT: begin
            $display("CONVERT: num_q8_24 = %h,denom_q8_24= %h factor_q8_24 = %h",
num_q8_24,denom_q8_24, factor_q8_24);
            mul_temp_64 = (num_q8_24 * factor_q8_24);
            num_q8_24 <= mul_temp_64[55:24];

            // d = d * delta
             mul_temp_64 = (denom_q8_24 * factor_q8_24);
            denom_q8_24 <= mul_temp_64[55:24];
            end

               FACTOR_CALC: begin
                      factor_q8_24 <= Q8_24_TWO - denom_q8_24;
               end

            GOLDSCHMIDT_ITER: begin
                if (iteration_counter < MAX_ITERATIONS) begin
                    // Goldschmidt iteration:
                    // factor = 2 - denominator * current_approximation
                    // numerator = numerator * factor
                    // denominator = denominator * factor
                    // Calculate 2 - d*x (factor)

                    // factor_q8_24 <= Q8_24_TWO - denom_q8_24;

                      mul_temp_64 = (num_q8_24 * factor_q8_24);
                      num_q8_24 <= mul_temp_64[55:24];

                    // d = d * delta
```

```
                    mul_temp_64 = (denom_q8_24 * factor_q8_24);
                    denom_q8_24 <= mul_temp_64[55:24];

                iteration_counter <= iteration_counter + 1;
                  $display("ITER%d: denom = %h, num = %h, factor = %h", iteration_counter,
denom_q8_24, num_q8_24, factor_q8_24);
            end
        end

        APPLY_CORRECTION: begin
            // Apply shift correction based on normalization
            if (shift>=0) begin
                // MSB set indicates right shift was applied during normalization
                // Need to shift result left to compensate
                num_q8_24 <= num_q8_24 << (shift);
            end else begin
                // Left shift was applied during normalization
                // Need to shift result right to compensate
                num_q8_24 <= num_q8_24 >> (-shift);
            end
            // If shift_amount == 0, no correction needed
        end

        ROUND_RESULT: begin
            // Round down Q8.24 to Q4.12 (truncate lower 12 bits)
            quotient <= num_q8_24[27:12];  // Extract Q4.12 portion
              $display("ROUND_RESULT: num_q8_24 = %h, quotient = %h", num_q8_24, num_q8_24[27:12]);
        end

        OUTPUT_RESULT: begin
            valid <= 1'b1;
            error <= 1'b0;
        end

        ERROR_STATE: begin
            valid <= 1'b1;
            error <= 1'b1;
            quotient <= 16'h0000;
        end
    endcase
  end
end
endmodule
```

Appendix B (Signed Q4.4 Unpipelined)

```
module goldschmidt_signed_44_unpip (
    input wire clk,
    input wire rst_n,
    input wire start,
    input wire signed [7:0] numerator,      // Q4.4 signed
    input wire signed [7:0] denominator,    // Q4.4 signed
    output reg signed [7:0] quotient,       // Q4.4 signed
    output reg valid,
    output reg error
);

    reg [4:0] count;
    reg [7:0] temp;

    localparam [3:0]
    IDLE              = 4'h0,
    VALIDATE_INPUT    = 4'h1,
    NORMALIZE_DENOM   = 4'h2,
    LOOKUP_INIT_APPROX= 4'h3,
    CONVERT           = 4'h4,
    FACTOR_CALC       = 4'hB,
    FIRST_MULT        = 4'h5,
    GOLDSCHMIDT_ITER  = 4'h6,
    APPLY_CORRECTION  = 4'h7,
    ROUND_RESULT      = 4'h8,
    OUTPUT_RESULT     = 4'h9,
    ERROR_STATE       = 4'hF;

    // Q-format constants
    localparam signed [15:0] Q8_8_ONE = 16'sh0100;
    localparam signed [15:0] Q8_8_TWO = 16'sh0200;
    localparam signed [7:0]  Q4_4_ONE = 8'sh10;
    localparam signed [7:0]  Q4_4_HALF = 8'sh08;
    localparam signed [7:0]  Q4_4_MAX = 8'sh7F;  // +127 in Q4.4
    localparam signed [7:0]  Q4_4_MIN = -8'sh80; // -128 in Q4.4
    localparam MAX_ITERATIONS = 3;

    reg [3:0] state, next_state;
    reg [2:0] iteration_counter;

    // Declare as signed for arithmetic correctness
    reg signed [15:0] num_q8_8;
    reg signed [15:0] denom_q8_8;
    reg signed [15:0] factor_q8_8;
    reg signed [31:0] mul_temp_32;

    reg signed [7:0] num_reg, denom_reg;
    reg signed [7:0] denom_norm_reg;
    reg [4:0] p;
```

```
    reg signed [5:0] shift;
    reg [7:0] factor_0;
    reg [2:0] index;

    reg result_sign;

    // Bit scan for normalization
    always @(*) begin
        if      (denom_reg[7]) p = 7;
        else if (denom_reg[6]) p = 6;
        else if (denom_reg[5]) p = 5;
        else if (denom_reg[4]) p = 4;
        else if (denom_reg[3]) p = 3;
        else if (denom_reg[2]) p = 2;
        else if (denom_reg[1]) p = 1;
        else if (denom_reg[0]) p = 0;
        else                   p = 0;

        shift = $signed(3 - p);
    end

    // Lookup table for initial approximation
    always @(*) begin
        case (index)
            3'd0: factor_0 = 8'd32;
            3'd1: factor_0 = 8'd28;
            3'd2: factor_0 = 8'd26;
            3'd3: factor_0 = 8'd23;
            3'd4: factor_0 = 8'd21;
            3'd5: factor_0 = 8'd20;
            3'd6: factor_0 = 8'd18;
            3'd7: factor_0 = 8'd17;
            default: factor_0 = 8'd0;
        endcase
    end

    // FSM: Next State
    always @(*) begin
        next_state = state;
        case (state)
            IDLE: if (start) next_state = VALIDATE_INPUT;
            VALIDATE_INPUT: begin
                if (denom_reg == 0)
                    next_state = ERROR_STATE;
                else if (num_reg == 0 || denom_reg == Q4_4_ONE || num_reg == denom_reg)
                    next_state = OUTPUT_RESULT;
                else
                    next_state = NORMALIZE_DENOM;
            end
            NORMALIZE_DENOM:     next_state = LOOKUP_INIT_APPROX;
            LOOKUP_INIT_APPROX:  next_state = CONVERT;
            CONVERT:             next_state = FIRST_MULT;
```

```
            FIRST_MULT:          next_state = FACTOR_CALC;
            FACTOR_CALC:         next_state = GOLDSCHMIDT_ITER;
            GOLDSCHMIDT_ITER:    next_state = (iteration_counter >= MAX_ITERATIONS) ?
APPLY_CORRECTION : FACTOR_CALC;
            APPLY_CORRECTION:    next_state = ROUND_RESULT;
            ROUND_RESULT:        next_state = OUTPUT_RESULT;
            OUTPUT_RESULT,
            ERROR_STATE:         next_state = IDLE;
            default:             next_state = IDLE;
        endcase
    end
    // FSM: Sequential
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            state <= IDLE;
            valid <= 0;
            error <= 0;
            quotient <= 0;
            iteration_counter <= 0;
            num_reg <= 0;
            denom_reg <= 0;
            num_q8_8 <= 0;
            denom_q8_8 <= 0;
            factor_q8_8 <= 0;
        end else begin
            state <= next_state;
            case (state)
                IDLE: begin
                    valid <= 0;
                    error <= 0;
                    if (start) begin
                        result_sign <= numerator[7] ^ denominator[7];  // XOR: if signs differ,
result is negative
                        num_reg <= (numerator[7]) ? -numerator : numerator;
                        denom_reg <= (denominator[7]) ? -denominator : denominator;
                        iteration_counter <= 0;
                    end
                end
                VALIDATE_INPUT: begin
                    if (num_reg == 0)
                        quotient <= 0;
                    else if (denom_reg == Q4_4_ONE)
                        quotient <= result_sign ? -num_reg : num_reg;
                    else if (num_reg == denom_reg)
                        quotient <= result_sign ? -Q4_4_ONE : Q4_4_ONE;
                end
                NORMALIZE_DENOM: begin
                    denom_norm_reg <= (shift >= 0) ? (denom_reg <<< shift) : (denom_reg >>> -shift);
                end
                LOOKUP_INIT_APPROX: begin
                    index <= denom_norm_reg[3:1];
                end
```

```
                CONVERT: begin
                    denom_q8_8 <= {denom_norm_reg, 4'b0};
                    num_q8_8   <= {num_reg, 4'b0};
                    factor_q8_8 <= factor_0 <<< 4;
                end
                FIRST_MULT: begin
                    mul_temp_32 = num_q8_8 * factor_q8_8;
                    num_q8_8    <= mul_temp_32[23:8];
                    mul_temp_32 = denom_q8_8 * factor_q8_8;
                    denom_q8_8  <= mul_temp_32[23:8];
                end
                FACTOR_CALC: begin
                    factor_q8_8 <= Q8_8_TWO - denom_q8_8;
                end
                GOLDSCHMIDT_ITER: begin
                    if (iteration_counter < MAX_ITERATIONS) begin
                        mul_temp_32 = num_q8_8 * factor_q8_8;
                        num_q8_8    <= mul_temp_32[23:8];
                        mul_temp_32 = denom_q8_8 * factor_q8_8;
                        denom_q8_8  <= mul_temp_32[23:8];
                        iteration_counter <= iteration_counter + 1;
                    end
                end
                APPLY_CORRECTION: begin
                    num_q8_8 <= (shift >= 0) ? (num_q8_8 <<< shift) : (num_q8_8 >>> -shift);
                end
                ROUND_RESULT: begin
                    // Round off
                    reg signed [7:0] rounded;
                    if (num_q8_8[3])
                        rounded = num_q8_8[11:4] + 1;
                    else
                        rounded = num_q8_8[11:4];
                    // Apply sign
                    quotient <= (result_sign) ? -rounded : rounded;
                    // Overflow clamp
                    if (quotient > Q4_4_MAX) quotient <= Q4_4_MAX;
                    else if (quotient < Q4_4_MIN) quotient <= Q4_4_MIN;
                end
                OUTPUT_RESULT: begin
                    valid <= 1;
                    error <= 0;
                end
                ERROR_STATE: begin
                    valid <= 1;
                    error <= 1;
                    quotient <= 0;
                end
            endcase
        end
    end
endmodule
```

Appendix C (Unsigned Q4.4 Unpipelined)

```
module goldschmidt_unsigned_44_unpip (
    input wire clk,
    input wire rst_n,
    input wire start,
    input wire [7:0] numerator,
    input wire [7:0] denominator,
    output reg [7:0] quotient,
    output reg valid,
    output reg error
);

// State and control registers
reg [4:0] count;
reg [7:0] temp;

localparam [3:0]
    IDLE              = 4'h0,
    VALIDATE_INPUT    = 4'h1,
    NORMALIZE_DENOM   = 4'h2,
    LOOKUP_INIT_APPROX= 4'h3,
    CONVERT           = 4'h4,
    FACTOR_CALC       = 4'hB,
    FIRST_MULT        = 4'h5,
    GOLDSCHMIDT_ITER  = 4'h6,
    APPLY_CORRECTION  = 4'h7,
    ROUND_RESULT      = 4'h8,
    OUTPUT_RESULT     = 4'h9,
    ERROR_STATE       = 4'hF;

// Fixed-point constants
localparam [15:0]
    Q8_8_ONE = 16'h0100,
    Q8_8_TWO = 16'h0200;

localparam MAX_ITERATIONS = 2;
localparam ROUND_BIAS = 16'd8;

// Internal state registers
reg [3:0] state, next_state;
reg [2:0] iteration_counter;

reg [15:0] num_q8_8;
reg [15:0] denom_q8_8;
reg [15:0] factor_q8_8;
reg [31:0] mul_temp_32;

reg [7:0] num_reg, denom_reg;
reg [7:0] denom_norm_reg;
reg [4:0] p;
```

```
reg signed [5:0] shift;
reg [7:0] factor_0;

// Priority encoder for normalization shift
always @(*) begin
    if      (denom_reg[7]) p = 7;
    else if (denom_reg[6]) p = 6;
    else if (denom_reg[5]) p = 5;
    else if (denom_reg[4]) p = 4;
    else if (denom_reg[3]) p = 3;
    else if (denom_reg[2]) p = 2;
    else if (denom_reg[1]) p = 1;
    else if (denom_reg[0]) p = 0;
    else                   p = 0;

    shift = $signed(3 - p);
end

// Lookup table for initial approximation factors
reg [2:0] index;
always @(*) begin
    case (index)
        3'd0: factor_0 = 8'd32;
        3'd1: factor_0 = 8'd28;
        3'd2: factor_0 = 8'd26;
        3'd3: factor_0 = 8'd23;
        3'd4: factor_0 = 8'd21;
        3'd5: factor_0 = 8'd20;
        3'd6: factor_0 = 8'd18;
        3'd7: factor_0 = 8'd17;
        default: factor_0 = 8'd0;
    endcase
end

// FSM next state logic
always @(*) begin
    next_state = state;
    case (state)
        IDLE:
            if (start)
                next_state = VALIDATE_INPUT;

        VALIDATE_INPUT: begin
            if (denom_reg == 8'h00)
                next_state = ERROR_STATE;
            else if (num_reg == 8'h00 || denom_reg == Q4_4_ONE || num_reg == denom_reg)
                next_state = OUTPUT_RESULT;
            else
                next_state = NORMALIZE_DENOM;
        end
        NORMALIZE_DENOM: next_state = LOOKUP_INIT_APPROX;
        LOOKUP_INIT_APPROX: next_state = CONVERT;
```

```
        CONVERT: next_state = FIRST_MULT;
        FIRST_MULT: next_state = FACTOR_CALC;
        FACTOR_CALC: next_state = GOLDSCHMIDT_ITER;
        GOLDSCHMIDT_ITER: next_state = (iteration_counter >= MAX_ITERATIONS) ? APPLY_CORRECTION :
FACTOR_CALC;
        APPLY_CORRECTION: next_state = ROUND_RESULT;
        ROUND_RESULT: next_state = OUTPUT_RESULT;
        OUTPUT_RESULT, ERROR_STATE: next_state = IDLE;
        default: next_state = IDLE;
    endcase
end

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        state             <= IDLE;
        valid             <= 0;
        error             <= 0;
        quotient          <= 8'h00;
        iteration_counter <= 0;
        num_reg           <= 8'h00;
        denom_reg         <= 8'h00;
        num_q8_8          <= 16'h0000;
        denom_q8_8        <= 16'h0000;
        factor_q8_8       <= 16'h0000;
    end else begin
        state <= next_state;

case (state)
        IDLE: begin
            valid <= 0;
            error <= 0;
            if (start) begin
                num_reg <= numerator;
                denom_reg <= denominator;
                iteration_counter <= 0;
            end
        end
        VALIDATE_INPUT: begin
            if (num_reg == 8'h00) quotient <= 8'h00;
            else if (denom_reg == Q4_4_ONE) quotient <= num_reg;
            else if (num_reg == denom_reg) quotient <= Q4_4_ONE;
        end
        NORMALIZE_DENOM: begin
            denom_norm_reg <= (shift >= 0) ? (denom_reg << shift) : (denom_reg >> -shift);
        end
        LOOKUP_INIT_APPROX: begin
            index <= denom_norm_reg[3:1];
        end
        CONVERT: begin
            denom_q8_8 <= {denom_norm_reg, 4'b0};
            num_q8_8 <= {num_reg, 4'b0};
            factor_q8_8 <= factor_0 << 4;
```

```
        end
        FIRST_MULT: begin
            mul_temp_32 = num_q8_8 * factor_q8_8;
            num_q8_8 <= mul_temp_32[23:8];
            mul_temp_32 = denom_q8_8 * factor_q8_8;
            denom_q8_8 <= mul_temp_32[23:8];
        end
        FACTOR_CALC: begin
            factor_q8_8 <= Q8_8_TWO - denom_q8_8;
        end
        GOLDSCHMIDT_ITER: begin
            if (iteration_counter < MAX_ITERATIONS) begin
                mul_temp_32 = num_q8_8 * factor_q8_8;
                num_q8_8 <= mul_temp_32[23:8];
                mul_temp_32 = denom_q8_8 * factor_q8_8;
                denom_q8_8 <= mul_temp_32[23:8];
                iteration_counter <= iteration_counter + 1;
            end
        end
        APPLY_CORRECTION: begin
            num_q8_8 <= (shift >= 0) ? (num_q8_8 << shift) : (num_q8_8 >> -shift);
        end
        ROUND_RESULT: begin
            quotient <= (num_q8_8+ROUND_BIAS)>>4;
        end
        OUTPUT_RESULT: begin
            valid <= 1;
            error <= 0;
        end
        ERROR_STATE: begin
            valid <= 1;
            error <= 1;
            quotient <= 8'h00;
        end
    endcase
end
```

Appendix D (Testbenches)

```
module goldschmidt_divider_tb;
    reg clk, rst_n, start;
    reg [15:0] numerator, denominator;
    wire [15:0] quotient;
    wire valid, error;
    // Clock generation
    always #5 clk = ~clk;
    goldschmidt_divider dut (
        .clk(clk),
        .rst_n(rst_n),
        .start(start),
        .numerator(numerator),
        .denominator(denominator),
        .quotient(quotient),
        .valid(valid),
        .error(error)
    );
    initial begin
        clk = 0;
        rst_n = 0;
        start = 0;
        numerator = 0;
        denominator = 0;
        #20 rst_n = 1;  // Reset
        // Test case 1: 2.0 / 1.0 = 2.0
        #10;
        numerator = 16'h2000;    // 2.0 in Q4.12
        denominator = 16'h1000;  // 1.0 in Q4.12
        start = 1;
        #10 start = 0;
        // Wait for completion
        wait(valid);
              if (error)
                  $display("Test 1: ERROR - Division by zero");
               else
                  $display("Test 1: 2.0/1.0 = %h (expected: 2000)", quotient);
        // Test case 2: 1.0 / 2.0 = 0.5
        #20;
        numerator = 16'h1000;    // 1.0 in Q4.12
        denominator = 16'h2000;  // 2.0 in Q4.12
        start = 1;
        #10 start = 0;
        wait(valid);
              if (error)
                  $display("Test 2: ERROR - Division by zero");
               else
                  $display("Test 2: 1.0/2.0 = %h (expected: 0800)", quotient);
       #20;
        numerator = 16'h2000;    // 2.0 in Q4.12
```

```
        denominator = 16'h2000;  // 2.0 in Q4.12
        start = 1;
        #10 start = 0;
        wait(valid);
             if (error)
                $display("Test 3: ERROR - Division by zero");
             else
                $display("Test 3: 2.0/2.0 = %h (expected: 1000)", quotient);
       #20;
        numerator = 16'h2000;    // 2.0 in Q4.12
        denominator = 16'h0000;  // 0.0 in Q4.12
        start = 1;
        #10 start = 0;
        wait(valid);
             if (error)
                $display("Test 4: ERROR - Division by zero");
             else
                $display("Test 4: 2.0/0.0 = %h (expected: error)", quotient);
        #20;
        numerator = 16'h2000;    // 2.0 in Q4.12
        denominator = 16'h7000;  // 7.0 in Q4.12
       start = 1;
        #10 start = 0;
        wait(valid);
            if (error)
                $display("Test 5: ERROR - Division by zero");
            else
                $display("Test 5: 2.0/7.0 = %h (expected: 0492)", quotient);
       #20;
        numerator = 16'h2C00;    // 2.75 in Q4.12
        denominator = 16'h1400;  // 1.25 in Q4.12
       start = 1;
        #10 start = 0;
        wait(valid);
              if (error)
                $display("Test 6: ERROR - Division by zero");
             else
                $display("Test 6: 2.75/1.25 = %h (expected: 2333)", quotient);
        #100 $finish;
    end
endmodule
```

```
module goldschmidt_divider_q4_4_tb;

reg clk, rst_n, start;
reg signed [7:0] numerator, denominator;
wire signed [7:0] quotient;
wire valid, error;

// Clock generation
always #5 clk = ~clk;

// DUT
goldschmidt_divider_q4_4 dut (
    .clk(clk),
    .rst_n(rst_n),
    .start(start),
    .numerator(numerator),
    .denominator(denominator),
    .quotient(quotient),
    .valid(valid),
    .error(error)
);
initial begin
    clk = 0;
    rst_n = 0;
    start = 0;
    numerator = 0;
    denominator = 0;
    // Reset
    #20 rst_n = 1;
    // ===== UNSIGNED TEST CASES =====
    // 2.0 / 1.0 = 2.0 → 0x20
    #10; numerator = 8'h20; denominator = 8'h10; start = 1; #10 start = 0;
    wait(valid);
    $display("Unsigned Test 1: 2.0 / 1.0 = %h (Expected: 20)", quotient);

    // 1.0 / 2.0 = 0.5 → 0x08
    #20; numerator = 8'h10; denominator = 8'h20; start = 1; #10 start = 0;
    wait(valid);
    $display("Unsigned Test 2: 1.0 / 2.0 = %h (Expected: 08)", quotient);

    // 2.0 / 2.0 = 1.0 → 0x10
    #20; numerator = 8'h20; denominator = 8'h20; start = 1; #10 start = 0;
    wait(valid);
    $display("Unsigned Test 3: 2.0 / 2.0 = %h (Expected: 10)", quotient);

    // 2.0 / 0.0 = error
    #20; numerator = 8'h20; denominator = 8'h00; start = 1; #10 start = 0;
    wait(valid);
    if (error)
        $display("Unsigned Test 4: ERROR - Division by zero");
    else
        $display("Unsigned Test 4: 2.0 / 0.0 = %h (Expected: error)", quotient);
```

```
    // 2.0 / 7.0 ≈ 0.2857 → ~0x05
    #20; numerator = 8'h20; denominator = 8'h70; start = 1; #10 start = 0;
    wait(valid);
    $display("Unsigned Test 5: 2.0 / 7.0 = %h (Expected: ~05)", quotient);

    // 2.75 / 1.25 ≈ 2.2 → ~0x23
    #20; numerator = 8'h2C; denominator = 8'h14; start = 1; #10 start = 0;
    wait(valid);
    $display("Unsigned Test 6: 2.75 / 1.25 = %h (Expected: ~23)", quotient);

    // ===== SIGNED TEST CASES =====

    // -8 / 2 = -4 → -4.0 = 0xC0
    #20; numerator = -8'sd8 << 4; denominator = 8'sd2 << 4; start = 1; #10 start = 0;
    wait(valid);
    $display("Signed Test 1: -8 / 2 = %h (Expected: C0)", quotient);

    // -2 / -2 = 1 → 0x10
    #20; numerator = -8'sd2 << 4; denominator = -8'sd2 << 4; start = 1; #10 start = 0;
    wait(valid);
    $display("Signed Test 2: -2 / -2 = %h (Expected: 10)", quotient);

    // -1 / 2 = -0.5 → 0xF8
    #20; numerator = -8'sd1 << 4; denominator = 8'sd2 << 4; start = 1; #10 start = 0;
    wait(valid);
    $display("Signed Test 3: -1 / 2 = %h (Expected: F8)", quotient);

    // -1 / -2 = +0.5 → 0x08
    #20; numerator = -8'sd1 << 4; denominator = -8'sd2 << 4; start = 1; #10 start = 0;
    wait(valid);
    $display("Signed Test 4: -1 / -2 = %h (Expected: 08)", quotient);

    // 2 / -1 = -2.0 → 0xE0
    #20; numerator = 8'sd2 << 4; denominator = -8'sd1 << 4; start = 1; #10 start = 0;
    wait(valid);
    $display("Signed Test 5: 2 / -1 = %h (Expected: E0)", quotient);

    #100 $finish;
end

endmodule
```

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
- DE1 SOC Documentation
  http://www.ee.ic.ac.uk/pcheung/teaching/ee2_digital/de1-soc_user_manual.pdf#page=4.15