Project 54: Integer Divider

A Comprehensive Study of Advanced Digital Circuits

By: Abhishek Sharma , Ayush Jain , Gati Goyal, Nikunj Agrawal

Documentation Specialist: Dhruv Patel & Nandini Maheshwari

Contents

1 Project Overview

An integer divider is a digital circuit that calculates the quotient and remainder of a division operation between two integers, the dividend and divisor. This operation is more complex than basic addition or multiplication, as it often involves iterative steps and conditional logic to ensure accuracy. Integer division techniques can be either restoring or non-restoring, with restoring division involving a corrective step after each negative result, while non-restoring division skips this, sometimes enhancing efficiency. Advanced designs, like pipelined or parallel dividers, allow simultaneous processing of bits, significantly speeding up the division process, which is essential in applications like CPUs, digital signal processing, and fields requiring precise integer calculations.

2 Integer Divider

2.1 Key Concept of Integer Divider

The key concept of an integer divider is to iteratively compute the quotient and remainder of two integers (the dividend and divisor) by shifting and comparing bit values, much like manual long division. The process involves examining each bit of the dividend to see if the divisor "fits" within that part, updating the quotient and remainder step-by-step:

- **Shift and Compare:**
  Division is achieved by shifting bits of the dividend and comparing them to the divisor. This shifting aligns each bit of the dividend with the divisor to determine whether it can be divided, helping to build the quotient bit by bit.

- **Subtract and Restore:**
  After a shift, the divisor is subtracted from the dividend (or the current remainder). In restoring division, if this subtraction results in a negative, the previous remainder is restored by adding the divisor back. In non-restoring division, this restoring step is omitted, simplifying the process and sometimes improving speed.

- **Iterative Quotient Building:**
  The quotient is built gradually as the divider moves through each bit of the dividend, with successful subtractions setting the current quotient bit to 1 and unsuccessful subtractions leaving it at 0.

- **Final Remainder:**
  Once all bits are processed, the remaining value becomes the remainder, which is what's left of the dividend that could not be divided by the divisor.

2.2 Working of Integer Divider

The working of an integer divider involves a sequence of steps to iteratively determine the quotient and remainder when dividing two integers (the dividend and the divisor). Here's a simplified explanation of how a typical integer divider operates:

1. **Initialize:**
   The dividend and divisor are placed into registers, and the quotient and remainder registers are initialized. The initial remainder is usually set to zero.

2. **Bit-by-Bit Comparison:**
   Division is performed similarly to manual long division, where each bit of the dividend is compared against the divisor to determine if the divisor fits into that portion of the dividend. This process often uses shift operations and subtraction to achieve the result.

3. **Subtraction and Shifting:**
   The most common methods—restoring and non-restoring division—operate by shifting bits and subtracting the divisor from the dividend:

   Restoring Division: After each subtraction, if the result is negative, the operation "restores" the previous value by adding the divisor back and shifting left. This keeps the remainder positive while determining the quotient.
   Non-Restoring Division: Here, instead of restoring the value, the operation continues by shifting and adjusting the quotient depending on whether the previous subtraction was positive or negative. This can lead to faster division as it eliminates the need for restoring steps.

4. **Accumulating Quotient and Remainder:**
   Each successful subtraction updates the quotient bit, gradually building the final result bit-by-bit. The remaining value after all steps becomes the remainder.

5. **End Condition:**
   The process continues until all bits of the dividend have been processed. The quotient now holds the result of the integer division, and any leftover amount is stored as the remainder.

6. **Sign Adjustment (for Signed Division):**
   If either of the original inputs (dividend or divisor) was negative, the quotient and remainder are adjusted to ensure they reflect the correct signs based on the rules of signed arithmetic.

2.3 RTL Code

Listing 1: Integer Divider

```

module IntegerDivider #(parameter WIDTH = 8) (
    input logic [WIDTH-1:0] dividend,
    input logic [WIDTH-1:0] divisor,
    output logic [WIDTH-1:0] quotient,
    output logic [WIDTH-1:0] remainder
);
    always_comb begin
        if (divisor != 0) begin
            quotient = dividend / divisor;
            remainder = dividend % divisor;
        end else begin
            quotient = '0; // undefined when divisor is 0
            remainder = dividend;
        end
    end
endmodule
```

2.4 Testbench

Listing 2: Integer Divider

```

`timescale 1ns/1ps

module IntegerDivider_tb;
    parameter WIDTH = 8;
    logic [WIDTH-1:0] dividend, divisor;
    logic [WIDTH-1:0] quotient, remainder;

    // Instantiate the Integer Divider
```

```
    IntegerDivider #(.WIDTH(WIDTH)) uut (
        .dividend(dividend),
        .divisor(divisor),
        .quotient(quotient),
        .remainder(remainder)
    );

    initial begin
        // Test case 1: Simple division
        dividend = 20; divisor = 3;
        #10;
        $display("Dividend=%0d, Divisor=%0d, Quotient=%0d,
            Remainder=%0d", dividend, divisor, quotient, remainder);

        // Test case 2: Exact division
        dividend = 15; divisor = 5;
        #10;
        $display("Dividend=%0d, Divisor=%0d, Quotient=%0d,
            Remainder=%0d", dividend, divisor, quotient, remainder);

        // Test case 3: Division by zero
        dividend = 15; divisor = 0;
        #10;
        $display("Dividend=%0d, Divisor=%0d, Quotient=%0d,
            Remainder=%0d", dividend, divisor, quotient, remainder);

        $finish;
    end
endmodule
```

3 Results

3.1 Simulation

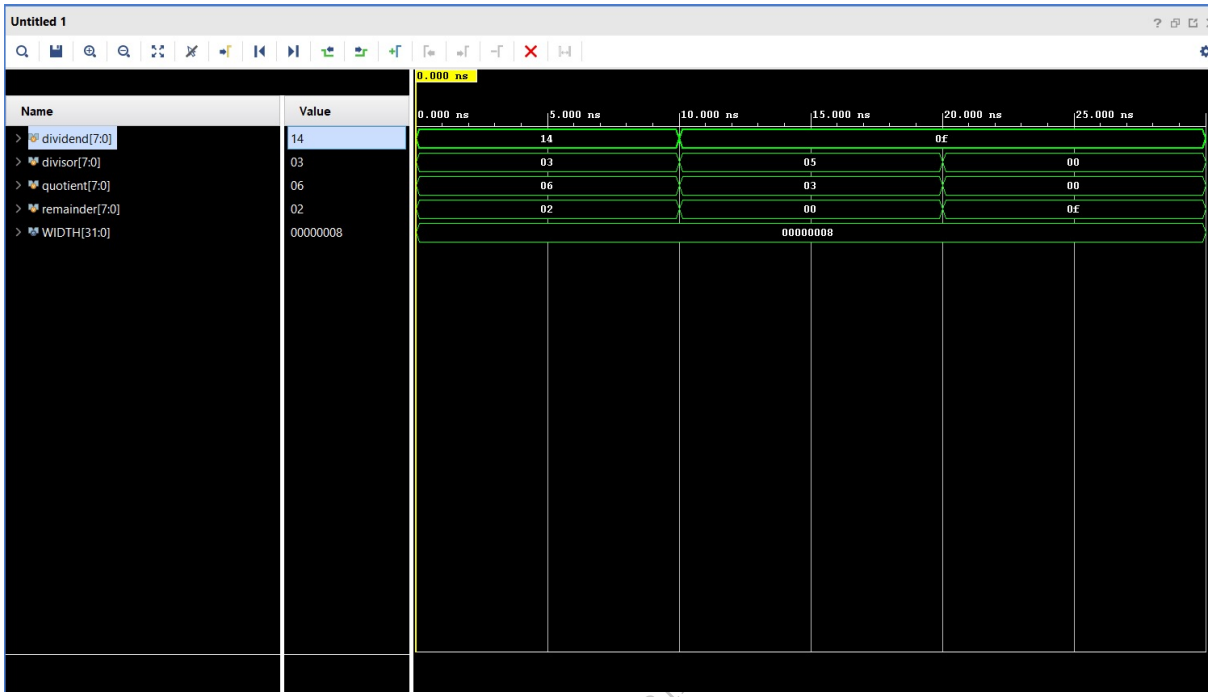

Figure 1: Simulation of Integer Divider

3.2 Schematic

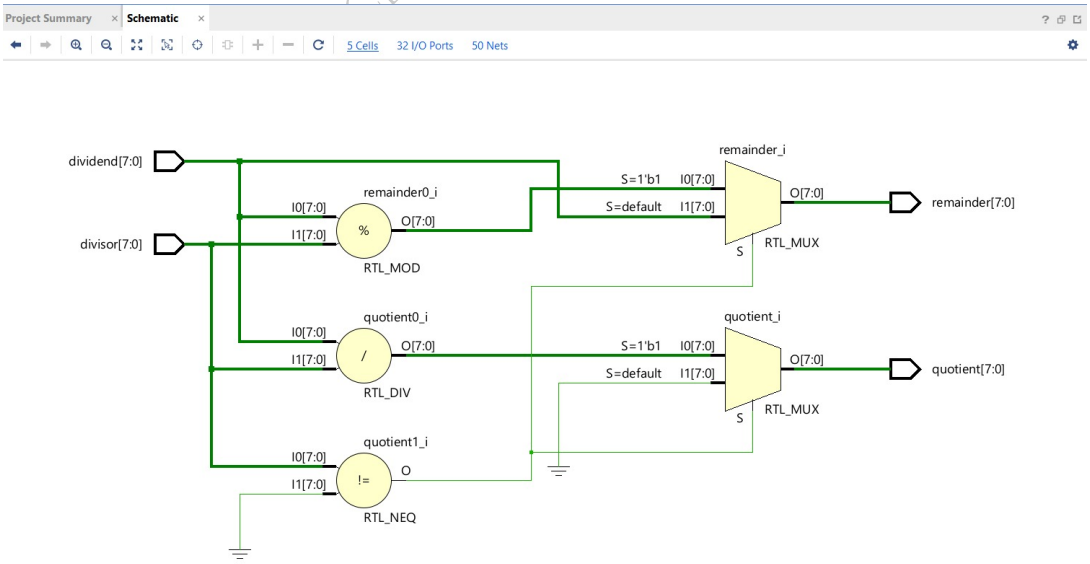

Figure 2: Schematic of Integer Divider

3.3 Synthesis Design

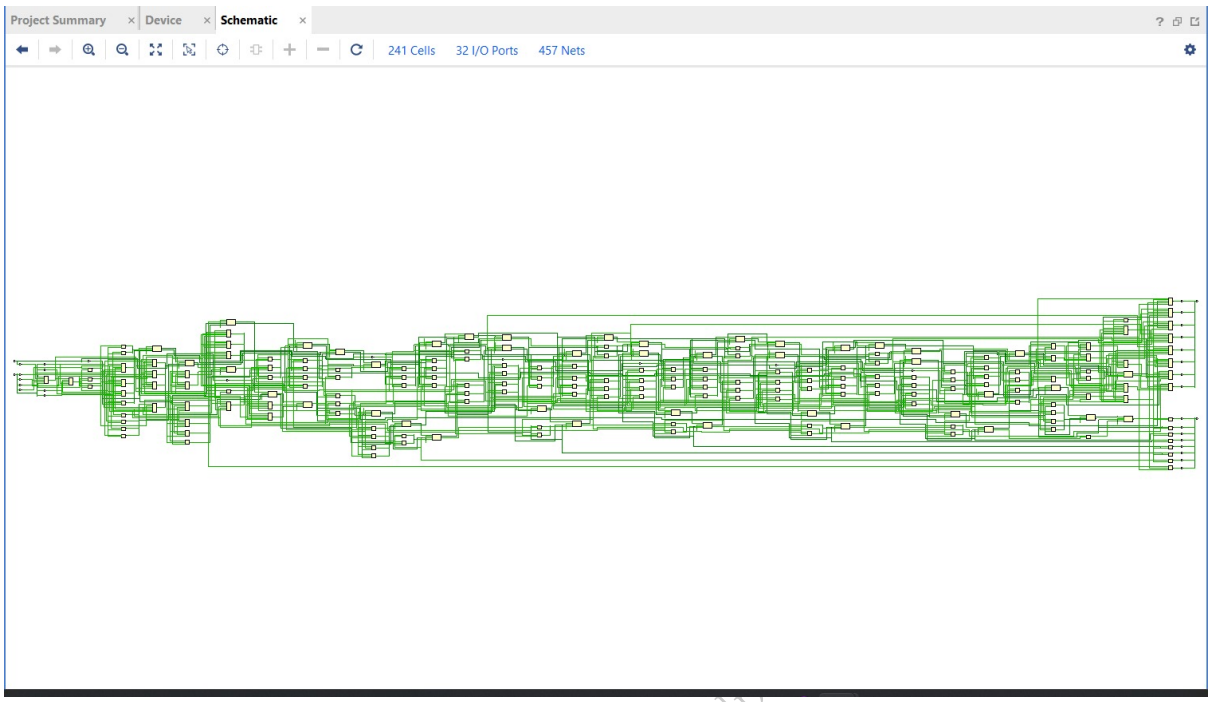

Figure 3: Synthesis Design of Integer Divider

4 Advantages of Integer Divider

- **Essential for Complex Arithmetic Operations:**
  Integer division enables essential calculations in applications like signal processing, cryptography, and computer graphics, where division is often necessary for accurate computation.

- **Precision in Whole-Number Calculations:**
  Unlike floating-point division, integer division provides exact whole-number results when inputs are integers, which is beneficial in scenarios that do not require fractional precision, reducing potential rounding errors.

- **Deterministic Performance:**
  Integer division follows predictable, set steps, making it well-suited for real-time systems that require reliable, consistent execution times without variability.

- **Cost-Effective for Fixed-Point Applications:**
  For systems requiring fixed-point arithmetic, integer dividers offer a simpler and more cost-effective solution than floating-point units, saving on hardware complexity and power.

- **Optimized Designs Available:**
  Variants like non-restoring and pipelined dividers allow for optimized, faster division operations, improving efficiency in CPUs and other high-speed applications without sacrificing precision for integer-based tasks.

5 Disadvantages of Integer Divider

- **Complexity and Power Consumption:**
  Division is a more complex operation than addition or multiplication, often requiring iterative or conditional steps, which leads to higher power consumption and greater circuit complexity.

- **Slower Processing Speed:**
  Integer division generally takes more time to complete compared to other arithmetic operations. This slower speed can bottleneck processing, especially in real-time or high-performance applications.

- **Higher Hardware Resource Usage:**
  Building efficient integer dividers, especially parallel or pipelined ones, requires more transistors and hardware resources, increasing both the cost and size of the circuit.

- **Precision Limitations:**
  Integer dividers work with whole numbers only, leading to rounding or truncation errors. This can reduce accuracy, particularly in applications requiring high precision or fractional results.

- **Error Propagation in Algorithms:**
  Due to rounding limitations, repeated use of integer division in complex algorithms can cause small errors to accumulate, potentially leading to significant inaccuracies.

6 Applications of Integer Divider

- **Processors and CPUs:**
  Used for arithmetic operations in ALUs, enabling efficient handling of division in calculations essential to general computing, floating-point arithmetic, and system operations.

- **Digital Signal Processing (DSP):**
  Integral in algorithms for filtering, audio and image processing, where division helps normalize values, perform Fourier transforms, and calculate ratios in real-time.

- **Cryptography:**
  Necessary for modular arithmetic in encryption and decryption algorithms, particularly in public-key cryptography methods like RSA, where large integers are divided to generate keys and encrypt data.

- **Graphics and Gaming:**
  Used for real-time scaling, transformations, and normalization of values in rendering graphics, enabling smooth animations and accurate coordinate mapping.

- **Control Systems and Robotics:**
  Helps in precision control by normalizing sensor data, calculating ratios for feedback loops, and performing real-time adjustments in automation tasks.

- **Financial and Scientific Calculations:**
  Used in algorithms requiring accurate fixed-point arithmetic, such as tax computations, interest calculations, and various scientific modeling processes.

7 Summary

An integer divider is a circuit that performs division on two integers, outputting a quotient and remainder. Techniques like restoring and non-restoring division manage iterative steps differently to improve accuracy or speed. Advanced designs, like pipelined or parallel dividers, increase processing speed and are crucial in applications requiring precise calculations.

8 FAQs

1. What is an integer divider?
An integer divider is a digital circuit that performs integer division, calculating the quotient and remainder when dividing two integers (the dividend and the divisor).

2. How does an integer divider work?
An integer divider works by iteratively comparing and subtracting the divisor from parts of the dividend. It shifts bits to align the divisor with portions of the dividend, gradually building the quotient and remainder.

3. What are the main methods used in integer division circuits?
There are two main methods:

Restoring Division: After each subtraction, if the result is negative, the remainder is "restored" by adding the divisor back.
Non-Restoring Division: This method skips the restoring step, adjusting the quotient based on the sign of each subtraction, which can increase speed.

4. What are the advantages of using integer dividers in digital systems?
Integer dividers are crucial for accurate, whole-number calculations in applications like signal processing, cryptography, and computer graphics. They provide deterministic, exact results for integer-based operations without the rounding issues common with floating-point division.

5. What are the disadvantages of integer dividers?
Integer division circuits are more complex, use more hardware resources, consume more power, and generally operate slower than other arithmetic circuits. They can also cause rounding issues since they do not handle fractional results.

6. Where are integer dividers commonly used?
Integer dividers are used in CPUs for arithmetic logic operations, digital signal processing, cryptographic algorithms, real-time control systems, and applications requiring precise integer calculations.

7. What's the difference between integer division and floating-point division?
Integer division only calculates whole-number quotients and discards fractions, while floating-point division can handle fractional results and is generally more complex and resource-intensive.

8. Can integer dividers handle negative numbers?
Yes, integer dividers can handle signed numbers. After performing division, they adjust the quotient and remainder based on the original signs of the dividend and divisor.

9. What's the purpose of the remainder in integer division?
The remainder is the portion of the dividend that is left after dividing out as many multiples of the divisor as possible. It represents what is left "undivided" and is often crucial in applications like modular arithmetic and cryptography.

10. Why are integer dividers slower than adders or multipliers?
Integer division requires multiple steps, including bit shifting and conditional subtraction, which are more complex than the straightforward operations in addition or multiplication. This iterative process

increases the computation time.