Project 13: Serial Adder

A Comprehensive Study of Advanced Digital Circuits

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

Contents

1 Introduction

The **Serial Adder** In the realm of digital electronics, the ability to perform arithmetic operations is fundamental to a wide range of applications, from basic calculators to complex computational systems. The Serial Adder is a digital circuit designed to add two binary numbers sequentially, one bit at a time. Unlike its parallel counterpart, which processes all bits simultaneously, the Serial Adder operates in a more resource-efficient manner, making it ideal for specific applications where simplicity and reduced hardware requirements are paramount.

2 Background

The Serial Adder is a fundamental digital circuit used for performing binary addition one bit at a time, in contrast to Parallel Adders which process multiple bits simultaneously. It operates using shift registers to store the binary numbers, a full adder to perform the addition, and a D flip-flop to propagate the carry-out signal. The Serial Adder's advantages include simplicity, reduced component count, lower power consumption, and flexibility in bit width, making it suitable for applications such as embedded systems, digital signal processing, and educational tools, although it is limited by slower speed compared to Parallel Adders. Understanding the operation and design of Serial Adders is crucial for grasping the fundamentals of digital electronics and binary arithmetic.

3 Structure and Operation

Structure and Operation of a Serial Adder: A Serial Adder is a digital circuit designed to perform binary addition sequentially, processing one bit at a time. Its structure typically consists of the following key components: Shift Registers: Two shift registers (often labeled A and B) are used to store the binary numbers being added. Each register holds one operand, allowing for the serial input of bits. As the addition progresses, bits are shifted out of these registers one at a time. Full Adder: The core of the Serial Adder is a single full adder, which takes three inputs: two bits from the shift registers (one from each register) and a carry-in bit from the previous addition. The full adder produces a sum bit and a carry-out bit. D Flip-Flop: A D flip-flop is employed to store the carry-out bit generated by the full adder. This carry-out is then used as the carry-in for the next pair of bits to be added. The flip-flop ensures that the carry is correctly propagated through the addition process.

Operation: The operation of a Serial Adder involves several steps: Initialization: The shift registers are loaded with the binary numbers to be added. Initially, the D flip-flop is cleared, meaning there is no carry present. Bit Addition: On each clock cycle, the following occurs: The least significant bits (LSBs) of the two shift registers are fed into the full adder. The full adder computes the sum of these bits along with the carry-in from the D flip-flop. It produces a sum bit and a carry-out bit. The sum bit is shifted into a separate shift register (or back into one of the original registers), while the carry-out is stored in the D flip-flop. Shifting: After each addition, the shift registers are shifted to the right, moving the next significant bits into position for the next addition. This process continues until all bits have been processed. Completion: Once all bits from both operands have been added, the final sum is available in the output shift register, and the carry-out indicates any overflow.

4 Implementation in SystemVerilog

The following RTL code implements the Serial Adder in SystemVerilog:

```
module SerialAdder (
    input logic clk,         // Clock signal
    input logic rst_n,      // Active low reset
    input logic a_bit,      // Input bit from operand A
    input logic b_bit,      // Input bit from operand B
    input logic start,      // Start signal
    output logic sum_bit,   // Output sum bit
    output logic carry_out,  // Carry out bit
```

```
    output logic done       // Done signal
);

    // State encoding
    typedef enum logic [1:0] {
        IDLE = 2'b00,
        ADD = 2'b01,
        DONE = 2'b10
    } state_t;

    state_t current_state, next_state;
    logic carry;

    // Sequential logic for state transitions
    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n)
            current_state <= IDLE;
        else
            current_state <= next_state;
    end

    // Combinational logic for next state and output logic
    always_comb begin
        next_state = current_state;
        sum_bit = 0;
        carry_out = 0;
        done = 0;

        case (current_state)
            IDLE: begin
                if (start) begin
                    next_state = ADD;
                end
            end
            ADD: begin
                {carry_out, sum_bit} = a_bit + b_bit + carry;
                carry = carry_out; // Update carry for the next addition
                next_state = DONE;
            end
            DONE: begin
                done = 1;
                next_state = IDLE; // Return to IDLE after done
            end
        endcase
    end
endmodule
```


5 Test Bench

The following test bench verifies the functionality of the Serial Adder:

```
 module SerialAdder_tb;

    logic clk;
    logic rst_n;
    logic a_bit;
    logic b_bit;
```

```
logic start;
logic sum_bit;
logic carry_out;
logic done;

// Instantiate the Serial Adder
SerialAdder dut (
    .clk(clk),
    .rst_n(rst_n),
    .a_bit(a_bit),
    .b_bit(b_bit),
    .start(start),
    .sum_bit(sum_bit),
    .carry_out(carry_out),
    .done(done)
);

// Clock generation
initial begin
    clk = 0;
    forever #5 clk = ~clk; // 10 time units period
end

// Test sequence
initial begin
    // Initialize signals
    rst_n = 0;
    a_bit = 0;
    b_bit = 0;
    start = 0;

    // Reset the system
    #10;
    rst_n = 1;

    // Test case 1: Add 1 + 1
    a_bit = 1; b_bit = 1; start = 1; #10;
    start = 0; #10; // Wait for the done signal
    wait(done);
    $display("Test 1: 1 + 1 = %b, Carry = %b", sum_bit, carry_out);

    // Test case 2: Add 0 + 1
    a_bit = 0; b_bit = 1; start = 1; #10;
    start = 0; #10; // Wait for the done signal
    wait(done);
    $display("Test 2: 0 + 1 = %b, Carry = %b", sum_bit, carry_out);

    // Test case 3: Add 1 + 0
    a_bit = 1; b_bit = 0; start = 1; #10;
    start = 0; #10; // Wait for the done signal
    wait(done);
    $display("Test 3: 1 + 0 = %b, Carry = %b", sum_bit, carry_out);

    // Test case 4: Add 1 + 0 with carry
    a_bit = 1; b_bit = 1; start = 1; #10;
    start = 0; #10; // Wait for the done signal
    wait(done);
    $display("Test 4: 1 + 1 = %b, Carry = %b", sum_bit, carry_out);
```

```
        // Finish simulation
        $finish;
    end
endmodule
```

6 Simulation Results

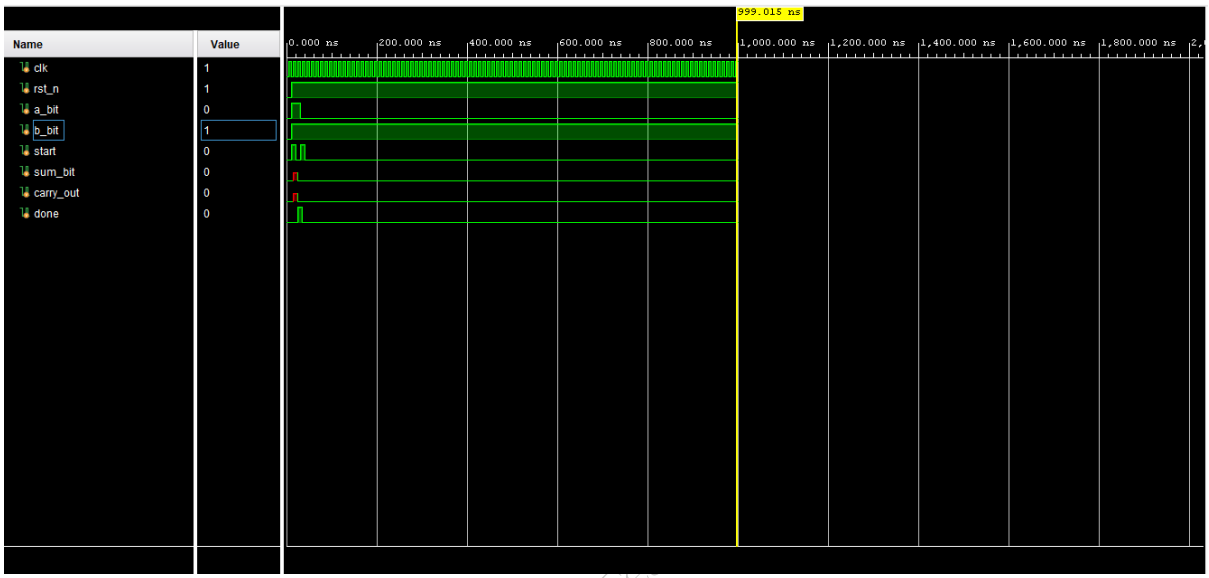

Figure 1: Simulation results of Serial Adder

7 Schematic

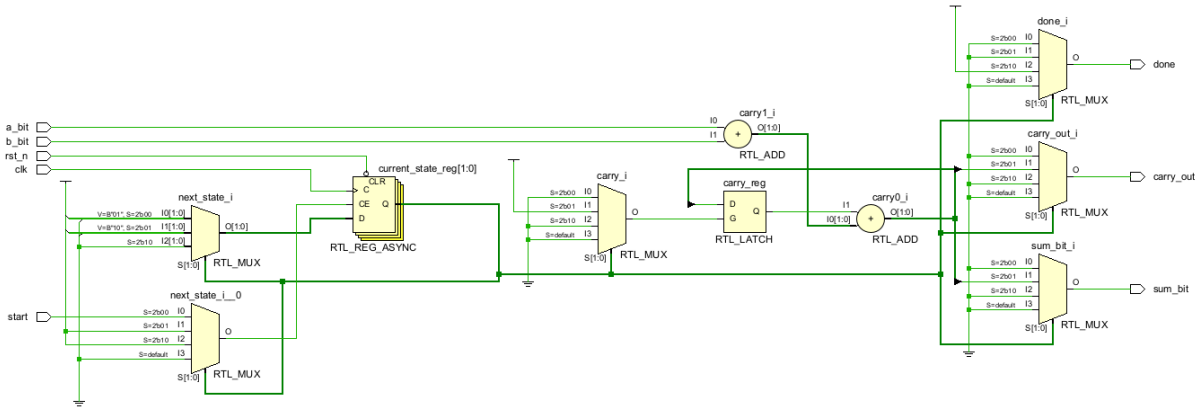

Figure 2: Schematic of Serial Adder

8 Synthesis Design

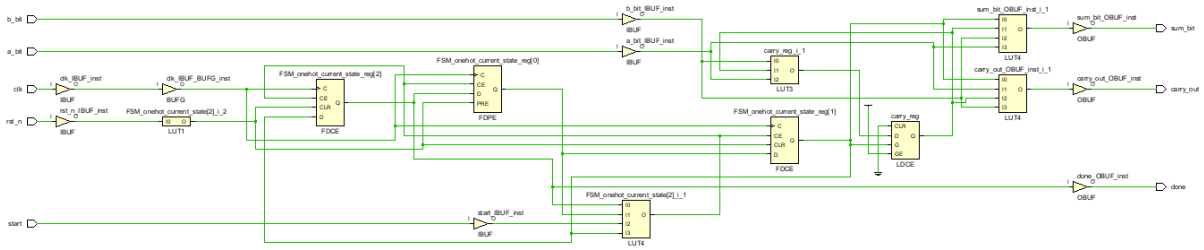

Figure 3: Synthesis of Serial Adder

9 Advantages and Disadvantages

9.1 Advantages

- **Fewer Components**: Serial Adders require significantly fewer components compared to Parallel Adders. A Serial Adder typically uses a single full adder and a few flip-flops, while a Parallel Adder requires multiple full adders equal to the number of bits being added .This simplicity can lead to lower costs and reduced power consumption in designs where component count is critical.
- **Reduced Complexity**: The design of a Serial Adder is generally simpler than that of a Parallel Adder. This can make it easier to implement and troubleshoot, especially in smaller or less complex digital systems.
- **Lower Propagation Delay**: In a Serial Adder, the addition is performed one bit at a time, which can lead to lower propagation delays for each individual bit. Since it processes bits sequentially, the carry propagation is managed in a straightforward manner, without the need for complex carry lookahead logic that is often required in Parallel Adders.
- **Power Efficiency**: Serial Adders can be more power-efficient, especially in battery-powered or low-power applications, since they utilize fewer active components and can operate at lower power levels compared to the multiple full adders used in Parallel Adders.

9.2 Disadvantages

- **Slower Speed**: Serial Adders can be more power-efficient, especially in battery-powered or low-power applications, since they utilize fewer active components and can operate at lower power levels compared to the multiple full adders used in Parallel Adders.
- **Inefficiency with Parallel Data**: Most computers use parallel data, so using a Serial Adder would require converting the data to serial format, performing the addition, and then converting the result back to parallel format. This process can be inefficient compared to using a Parallel Adder that can directly operate on the parallel data.

- **Increased Propagation Delay with Bit-Width**: While Serial Adders have lower propagation delays for individual bits, the total propagation delay increases linearly with the number of bits being added. This can be a significant disadvantage in high-speed applications where rapid processing of multiple bits is essential.

10 Conclusion

In conclusion, Serial Adders and Parallel Adders serve distinct purposes in digital arithmetic, each with its own set of advantages and disadvantages. Serial Adders are simpler, require fewer components, and consume less power, making them suitable for resource-constrained applications; however, they operate at significantly slower speeds since they process one bit at a time, leading to increased propagation delays with larger bit widths. In contrast, Parallel Adders excel in speed and efficiency by adding multiple bits simultaneously, making them ideal for high-performance applications, but they come with increased complexity, higher costs, and greater power consumption. Ultimately, the choice between a Serial Adder and a Parallel Adder depends on the specific requirements of the application, including speed, complexity, power consumption, and the nature of the data being processed.