Project 8: 16-bit Prefix Adder (Kogge-Stone Adder)

A Comprehensive Study of Advanced Digital Circuits

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

Contents

1 Introduction

Prefix addition is a method used to efficiently perform binary addition by precomputing partial sums and carries, leveraging parallelism to reduce the overall computation time. This approach is fundamental in designing high-speed adders such as the Carry-Lookahead Adder (CLA), Brent-Kung Adder, and Kogge-Stone Adder.

2 Key Concepts

1. **Generate and Propagate:** Each bit in the binary addition process generates and propagates carry information.
   - **Generate (G):** A bit pair generates a carry if both bits are 1.
   - **Propagate (P):** A bit pair propagates a carry if at least one bit is 1.
2. **Carry Computation:** The carry for each bit position is computed using the generate and propagate signals. This computation can be performed in parallel for multiple bit positions.

3 Steps in Prefix Addition

1. **Preprocessing:** Compute the generate (G) and propagate (P) signals for each bit position.

$$G_i = A_i \cdot B_i$$
$$P_i = A_i + B_i$$

2. **Prefix Computation:** Compute the carry signals using a prefix tree structure.
   - The carry for each bit position is determined by combining the generate and propagate signals from previous bit positions.
   - This can be visualized as a tree where each level reduces the number of operations by combining results from the previous level.
3. **Postprocessing:** Compute the final sum for each bit position.

$$S_i = P_i \oplus C_{i-1}$$

   where C_{i-1} is the carry from the previous bit position.

4 Types of Prefix Adders

1. **Carry-Lookahead Adder (CLA):** Uses the generate and propagate signals to compute carries in logarithmic time.
2. **Brent-Kung Adder:** A tree structure that balances the trade-off between speed and hardware complexity.
3. **Kogge-Stone Adder:** A highly parallel adder that provides fast addition with minimal delay at the cost of increased hardware complexity.

5 Example: Kogge-Stone Adder

The Kogge-Stone Adder is one of the fastest adders, known for its minimal depth and maximum parallelism. Here is a simplified explanation of its operation:

1. **Initialization:** Compute the generate and propagate signals for each bit.

$$G_i = A_i \cdot B_i$$
$$P_i = A_i + B_i$$

2. **Prefix Tree Computation:** Use a tree structure to compute the carry signals.

$$G_{i:j} = G_i + (P_i \cdot G_{i-1:j})$$
$$P_{i:j} = P_i \cdot P_{i-1:j}$$

   At each level of the tree, combine generate and propagate signals from previous levels.

3. **Sum Computation:** Compute the final sum bits using the propagate and carry signals.

$$S_i = P_i \oplus C_{i-1}$$

The Kogge-Stone Adder reduces the carry computation to logarithmic time, significantly speeding up the addition process compared to traditional adders.

6 RTL Code

Listing 1: Kogge Stone Adder RTL Code

```
module project8(
    input  logic [15:0] A,
    input  logic [15:0] B,
    input  logic        Cin,
    output logic [15:0] Sum,
    output logic        Cout
);

    logic [15:0] G, P;          // Generate and Propagate
    logic [15:0] G1, P1;        // First stage
    logic [15:0] G2, P2;        // Second stage
    logic [15:0] G3, P3;        // Third stage
    logic [15:0] C;             // Carry

    // Generate and Propagate signals
    assign G = A & B;
    assign P = A ^ B;

    // First stage
    assign G1[0] = G[0];
    assign P1[0] = P[0];
    assign G1[1] = G[1]  (P[1] & G[0]);
    assign P1[1] = P[1] & P[0];
    assign G1[2] = G[2]  (P[2] & G[1]);
    assign P1[2] = P[2] & P[1];
    assign G1[3] = G[3]  (P[3] & G[2]);
    assign P1[3] = P[3] & P[2];
    assign G1[4] = G[4]  (P[4] & G[3]);
    assign P1[4] = P[4] & P[3];
    assign G1[5] = G[5]  (P[5] & G[4]);
    assign P1[5] = P[5] & P[4];
    assign G1[6] = G[6]  (P[6] & G[5]);
    assign P1[6] = P[6] & P[5];
    assign G1[7] = G[7]  (P[7] & G[6]);
    assign P1[7] = P[7] & P[6];
```

```
    assign G1[8] = G[8]  (P[8] & G[7]);
    assign P1[8] = P[8] & P[7];
    assign G1[9] = G[9]  (P[9] & G[8]);
    assign P1[9] = P[9] & P[8];
    assign G1[10] = G[10]  (P[10] & G[9]);
    assign P1[10] = P[10] & P[9];
    assign G1[11] = G[11]  (P[11] & G[10]);
    assign P1[11] = P[11] & P[10];
    assign G1[12] = G[12]  (P[12] & G[11]);
    assign P1[12] = P[12] & P[11];
    assign G1[13] = G[13]  (P[13] & G[12]);
    assign P1[13] = P[13] & P[12];
    assign G1[14] = G[14]  (P[14] & G[13]);
    assign P1[14] = P[14] & P[13];
    assign G1[15] = G[15]  (P[15] & G[14]);
    assign P1[15] = P[15] & P[14];

    // Second stage
    assign G2[1:0] = G1[1:0];
    assign P2[1:0] = P1[1:0];
    assign G2[2] = G1[2];
    assign P2[2] = P1[2];
    assign G2[3] = G1[3]  (P1[3] & G1[1]);
    assign P2[3] = P1[3] & P1[2];
    assign G2[4] = G1[4];
    assign P2[4] = P1[4];
    assign G2[5] = G1[5]  (P1[5] & G1[3]);
    assign P2[5] = P1[5] & P1[4];
    assign G2[6] = G1[6]  (P1[6] & G1[4]);
    assign P2[6] = P1[6] & P1[5];
    assign G2[7] = G1[7]  (P1[7] & G1[5]);
    assign P2[7] = P1[7] & P1[6];
    assign G2[8] = G1[8];
    assign P2[8] = P1[8];
    assign G2[9] = G1[9]  (P1[9] & G1[7]);
    assign P2[9] = P1[9] & P1[8];
    assign G2[10] = G1[10]  (P1[10] & G1[8]);
    assign P2[10] = P1[10] & P1[9];
    assign G2[11] = G1[11]  (P1[11] & G1[9]);
    assign P2[11] = P1[11] & P1[10];
    assign G2[12] = G1[12]  (P1[12] & G1[10]);
    assign P2[12] = P1[12] & P1[11];
    assign G2[13] = G1[13]  (P1[13] & G1[11]);
    assign P2[13] = P1[13] & P1[12];
    assign G2[14] = G1[14]  (P1[14] & G1[12]);
    assign P2[14] = P1[14] & P1[13];
    assign G2[15] = G1[15]  (P1[15] & G1[13]);
    assign P2[15] = P1[15] & P1[14];

    // Third stage
    assign G3[3:0] = G2[3:0];
    assign P3[3:0] = P2[3:0];
    assign G3[4] = G2[4];
    assign P3[4] = P2[4];
    assign G3[5] = G2[5];
    assign P3[5] = P2[5];
    assign G3[6] = G2[6];
    assign P3[6] = P2[6];
```

```
    assign G3[7] = G2[7]  (P2[7] & G2[3]);
    assign P3[7] = P2[7] & P2[6];
    assign G3[8] = G2[8];
    assign P3[8] = P2[8];
    assign G3[9] = G2[9];
    assign P3[9] = P2[9];
    assign G3[10] = G2[10];
    assign P3[10] = P2[10];
    assign G3[11] = G2[11]  (P2[11] & G2[7]);
    assign P3[11] = P2[11] & P2[10];
    assign G3[12] = G2[12]  (P2[12] & G2[8]);
    assign P3[12] = P2[12] & P2[11];
    assign G3[13] = G2[13]  (P2[13] & G2[9]);
    assign P3[13] = P2[13] & P2[12];
    assign G3[14] = G2[14]  (P2[14] & G2[10]);
    assign P3[14] = P2[14] & P2[13];
    assign G3[15] = G2[15]  (P2[15] & G2[11]);
    assign P3[15] = P2[15] & P2[14];

    // Final stage (Carries)
    assign C[0] = Cin;
    assign C[1] = G[0]  (P[0] & Cin);
    assign C[2] = G1[1]  (P1[1] & Cin);
    assign C[3] = G2[3]  (P2[3] & Cin);
    assign C[4] = G3[3]  (P3[3] & Cin);
    assign C[5] = G3[4]  (P3[4] & C[1]);
    assign C[6] = G3[5]  (P3[5] & C[2]);
    assign C[7] = G3[6]  (P3[6] & C[3]);
    assign C[8] = G3[7]  (P3[7] & C[4]);
    assign C[9] = G3[8]  (P3[8] & C[5]);
    assign C[10] = G3[9]  (P3[9] & C[6]);
    assign C[11] = G3[10]  (P3[10] & C[7]);
    assign C[12] = G3[11]  (P3[11] & C[8]);
    assign C[13] = G3[12]  (P3[12] & C[9]);
    assign C[14] = G3[13]  (P3[13] & C[10]);
    assign C[15] = G3[14]  (P3[14] & C[11]);

    // Sum and Cout
    assign Sum = P ^ C;
    assign Cout = G3[15]  (P3[15] & C[12]);

endmodule
```

6.1 Testbench

Listing 2: Kogge Stone Adder Testbench

```
module project8_tb;

    logic [15:0] A, B;
    logic        Cin;
    logic [15:0] Sum;
    logic        Cout;

    // Instantiate the Kogge-Stone Adder
    project8 uut (
        .A(A),
        .B(B),
```

```verilog
        .Cin(Cin),
        .Sum(Sum),
        .Cout(Cout)
    );

    // Test cases
    initial begin
        // Initialize inputs
        A = 16'h0000; B = 16'h0000; Cin = 1'b0;
        #10;  // Wait for 10 time units

        // Test case 1
        A = 16'h1234; B = 16'h5678; Cin = 1'b0;
        #10;
        $display("A=%h, B=%h, Cin=%b -> Sum=%h, Cout=%b", A, B, Cin,
           Sum, Cout);

        // Test case 2
        A = 16'hAAAA; B = 16'h5555; Cin = 1'b1;
        #10;
        $display("A=%h, B=%h, Cin=%b -> Sum=%h, Cout=%b", A, B, Cin,
           Sum, Cout);

        // Test case 3
        A = 16'hFFFF; B = 16'h0001; Cin = 1'b0;
        #10;
        $display("A=%h, B=%h, Cin=%b -> Sum=%h, Cout=%b", A, B, Cin,
           Sum, Cout);

        // Test case 4
        A = 16'hFFFF; B = 16'hFFFF; Cin = 1'b1;
        #10;
        $display("A=%h, B=%h, Cin=%b -> Sum=%h, Cout=%b", A, B, Cin,
           Sum, Cout);

        // Test case 5
        A = 16'h8000; B = 16'h8000; Cin = 1'b0;
        #10;
        $display("A=%h, B=%h, Cin=%b -> Sum=%h, Cout=%b", A, B, Cin,
           Sum, Cout);

        $finish; // End simulation
    end

endmodule
```

7 Simulation Results

8 Schematic

9 Synthesis Design

10 Advantages

- **Speed:** Parallel computation of carries significantly reduces addition time.

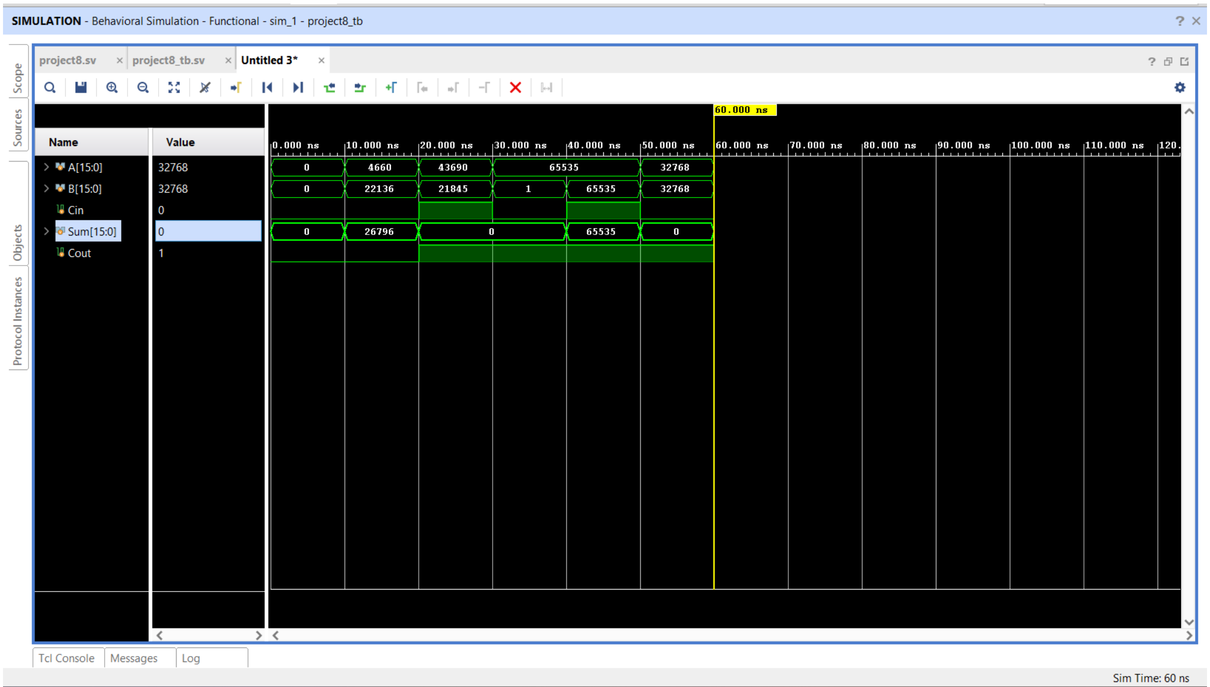

Figure 1: Simulation results of Kogge Stone Adder

- **Scalability:** Suitable for large bit-width additions due to its logarithmic time complexity.

11 Disadvantages

- **Hardware Complexity:** Increased number of logic gates and interconnections.
- **Power Consumption:** Higher power consumption due to the increased hardware complexity.

12 Applications

- **High-Speed Processors:** Used in arithmetic logic units (ALUs) and central processing units (CPUs) where fast addition is critical.
- **Digital Signal Processing (DSP):** Employed in DSP applications requiring rapid arithmetic operations.

13 Conclusion

Prefix adders provide a significant speed advantage in binary addition by leveraging parallelism and precomputing partial results. Their ability to quickly compute carries makes them essential in high-performance computing applications, despite the trade-off in hardware complexity and power consumption.

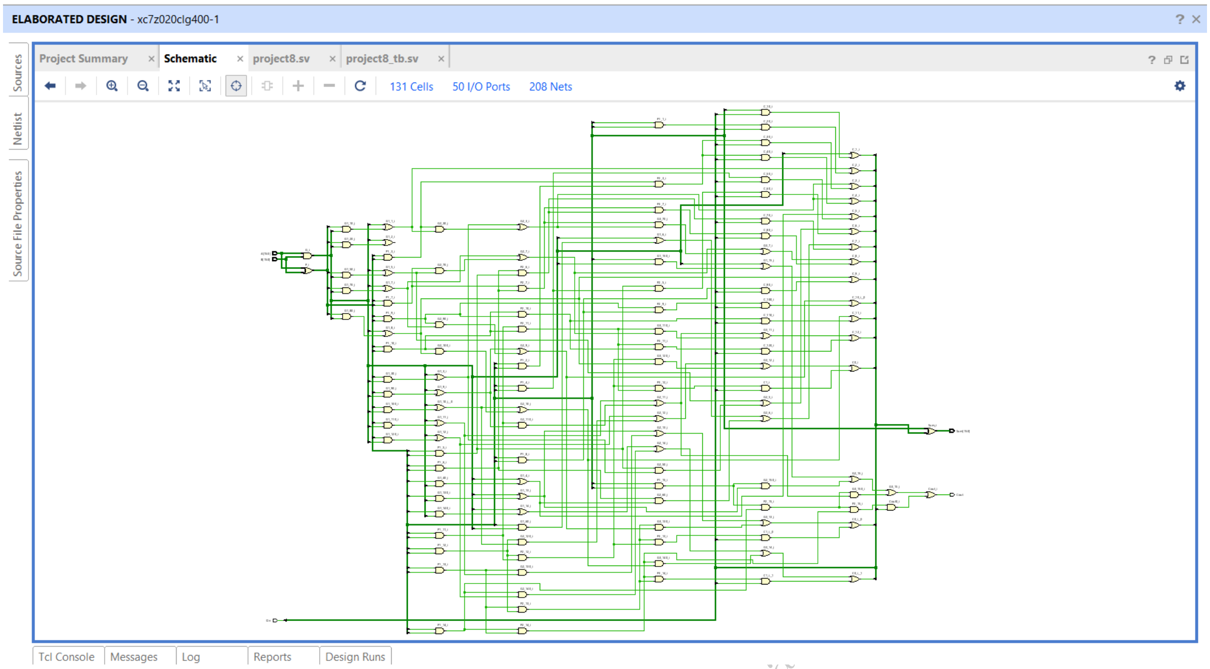

Figure 2: Schematic of Kogge Stone Adder

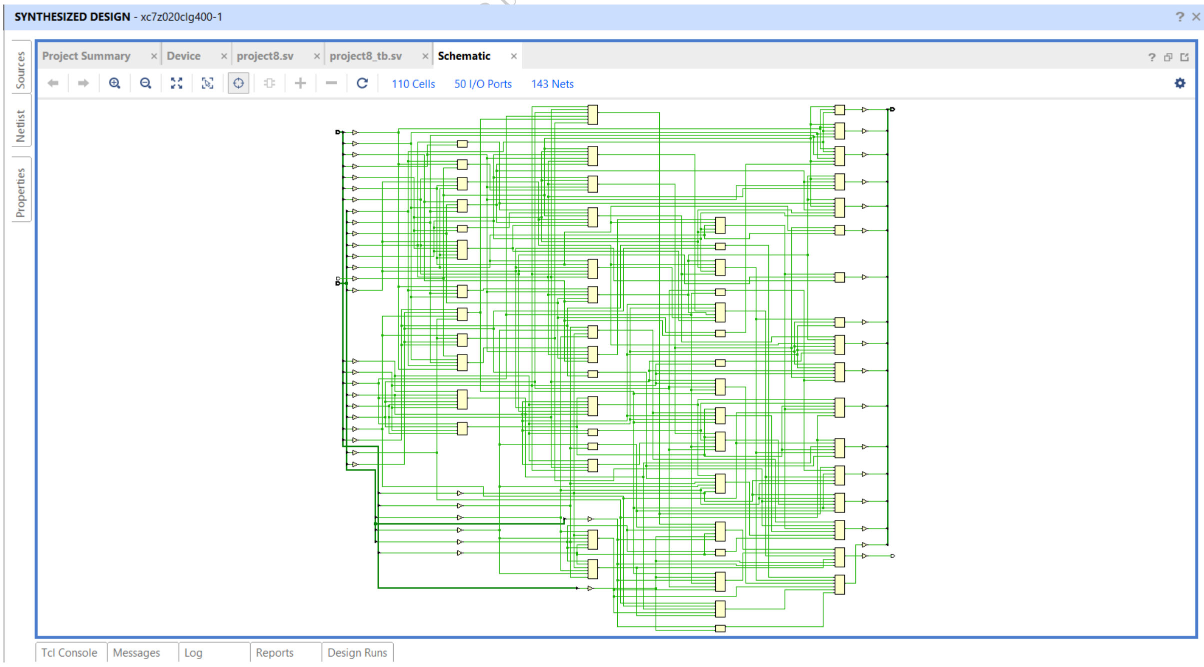

Figure 3: Synthesis Design of RCA