Project 4: 32 Bit CLA Using Block Carry Lookahead Generators

A Comprehensive Study of Advanced Digital Circuits

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

Contents

1 Project Overview

The focus of this project is the design and implementation of Block Carry Lookahead Generators (BCLGs) to enhance the performance of Carry Lookahead Adders (CLAs) for wider bit-width binary addition. This project addresses the challenges of scaling CLAs for larger bit-widths by employing a block-based approach to manage carry signals more efficiently and reduce propagation delays.

2 Block Carry Lookahead Generators

2.1 Description

In a CLA, carry signals are generated and propagated in parallel to reduce delay. For larger bit-width adders, directly implementing a single CLA can become complex and inefficient. Instead, the adder is divided into smaller blocks, each with its own carry lookahead logic. The Block Carry Lookahead Generator then manages the carry signals between these blocks.

2.2 Carry Lookahead Logic

Each block in a CLA has its own generate (G) and propagate (P) signals. The block generate (G) and propagate (P) signals for a block of bits are defined as:

$$G_{block} = G_i + P_i \cdot G_{i-1} + P_i \cdot P_{i-1} \cdot G_{i-2} + \ldots + P_i \cdot P_{i-1} \cdot \ldots \cdot P_0 \cdot G_0$$

$$P_{block} = P_i \cdot P_{i-1} \cdot \ldots \cdot P_0$$

Where G_i and P_i are the generate and propagate signals for bit i within the block. These block signals allow the calculation of carry signals for each block.

2.3 Block Carry Lookahead Generator Implementation

A Block CLG calculates the carry output for each block using the block generate and propagate signals. This can be represented as:

$$C_{i+1} = G_{block} + P_{block} \cdot C_i$$

Where C_i is the carry input to the block, and C_{i+1} is the carry output from the block.

3 Explantation

- **Generate Signals (G):** These are derived from each bit in the block and indicate if a carry will be generated at that bit.
- **Propagate Signals (P):** These indicate if a carry input will propagate through that bit.
- **Block Propagate Signal (P_block)** : $Thissignalindicatesifacarryinputtotheblockwillpropagatethroughallbitsinthebloc$

3.1 RTL Code

Listing 1: 32 Bit Carry Lookahead Adder

```
module cla04 (
    input  logic [3:0] A,
    input  logic [3:0] B,
    input  logic       Cin,
    output logic [3:0] Sum,
  //  output logic       Cout,
    output logic       gout, // Propagate
```

```
    output logic        pout  // Generate
);

    logic [3:0] G_i, P_i,Ci;

    assign G_i = A & B;  // Generate
    assign P_i = A ^ B;  // Propagate

   // assign C_i[0] = Cin;
   // assign C_i[1] = G_i[0] | (P_i[0] & Cin);
  //  assign C_i[2] = G_i[1] | (P_i[1] & C_i[1]);
   // assign C_i[3] = G_i[2] | (P_i[2] & C_i[2]);
   // assign Cout = G_i[3] | (P_i[3] & C_i[3]);
BCLG_16bit  a1(G_i, P_i, Cin, Ci, gout, pout);
    assign Sum = P_i ^ Ci;
  //  assign Cout=Ci[3];

  //  assign P = &P_i; // Overall block propagate
   // assign G = G_i[3] | (P_i[3] & G_i[2]) | (P_i[3] & P_i[2] &
       G_i[1]) | (P_i[3] & P_i[2] & P_i[1] & G_i[0]); // Overall block
       generate

endmodule

module BCLG_16bit (g, p, cin, cout, gout, pout);

  input [3:0] g, p;
  input cin;
  output [3:0] cout;
  output gout, pout;

  assign cout[0] = cin;
  assign cout[1] = g[0]  (p[0] & cin);
  assign cout[2] = g[1]  (p[1] & g[0])  (p[1] & p[0] & cin);
  assign cout[3] = g[2]  (p[2] & g[1])  (p[2] & p[1] & g[0])  (p[2] &
      p[1] & p[0] & cin);

  assign gout = g[3]  (p[3] & g[2])  (p[3] & p[2] & g[1])  (p[3] &
      p[2] & p[1] & g[0]);
  assign pout = p[0] & p[1] & p[2] & p[3];
endmodule

module CLA_16bit (
    input  logic [15:0] A,
    input  logic [15:0] B,
    input  logic        Cin,
    output logic [15:0] Sum,
    output logic       Cout,
    output logic gout,pout
);

    logic [3:0] P, G, C;
    logic [15:0] Sum0 ;

    cla04 cla0 (A[ 3: 0], B[ 3: 0], Cin, Sum0[3:0], G[0], P[0]);
    cla04 cla1 (A[ 7: 4], B[ 7: 4], C[1], Sum0[7:4], G[1], P[1]);
    cla04 cla2 (A[11: 8], B[11: 8], C[2], Sum0[11:8], G[2], P[2]);
    cla04 cla3 (A[15:12], B[15:12], C[3], Sum0[15:12], G[3], P[3]);
```

```
62
63      BCLG_16bit bclg (G, P, Cin, C,gout,pout);
64
65      assign Sum = {Sum0[15:12], Sum0[11:8], Sum0[7:4], Sum0[3:0]};
66     assign Cout = C[3];
67
68  endmodule
69
70  module cla32(
71      input  logic [31:0] A,
72      input  logic [31:0] B,
73      input  logic        Cin,
74      output logic [31:0] Sum,
75      output logic        Cout
76  );
77
78      logic [1:0]  C;
79      logic [15:0] Sum0, Sum1;
80
81      CLA_16bit cla0 (A[15:0], B[15:0], Cin, Sum0,C[0]);
82      CLA_16bit cla1 (A[31:16], B[31:16], C[0], Sum1,C[1]);
83
84
85
86      assign Sum = {Sum1, Sum0};
87      assign Cout = C[1];
88
89  endmodule
```


3.2 Testbench

Listing 2: 32 bit Carry Lookahead Adder Testbench

```
1  module adder32bit_tb(
2
3      );
4
5
6      // Declare inputs as reg and outputs as wire
7      logic [31:0] A;
8      logic [31:0] B;
9      logic Cin;
10    logic [31:0] Sum;
11     logic Cout;
12
13     // Instantiate the bclg32 module
14     bclg32 uut (
15         .A(A),
16         .B(B),
17         .Cin(Cin),
18         .Sum(Sum),
19         .Cout(Cout)
20     );
21
22     // Testbench procedure
23     initial begin
24         // Initialize inputs
25         A = 32'h00000000;
```

```
        B = 32'h00000000;
        Cin = 1'b0;
        #10;  // Wait for 10 time units

        // Test Case 1: Add zero to zero
        A = 32'h00000000;
        B = 32'h00000000;
        Cin = 1'b0;
        #10;
        $display("TC1: A = %h, B = %h, Cin = %b | Sum = %h, Cout =
            %b", A, B, Cin, Sum, Cout);

        // Test Case 2: Add two small numbers
        A = 32'h0000000F;
        B = 32'h00000001;
        Cin = 1'b0;
        #10;
        $display("TC2: A = %h, B = %h, Cin = %b | Sum = %h, Cout =
            %b", A, B, Cin, Sum, Cout);

        // Test Case 3: Add two large numbers with carry in
        A = 32'hFFFFFFFF;
        B = 32'h00000001;
        Cin = 1'b1;
        #10;
        $display("TC3: A = %h, B = %h, Cin = %b | Sum = %h, Cout =
            %b", A, B, Cin, Sum, Cout);

        // Test Case 4: Add large and small number with carry in
        A = 32'h12345678;
        B = 32'h87654321;
        Cin = 1'b1;
        #10;
        $display("TC4: A = %h, B = %h, Cin = %b | Sum = %h, Cout =
            %b", A, B, Cin, Sum, Cout);

        // Test Case 5: Add random values
        A = 32'hABCDEF01;
        B = 32'h12345678;
        Cin = 1'b0;
        #10;
        $display("TC5: A = %h, B = %h, Cin = %b | Sum = %h, Cout =
            %b", A, B, Cin, Sum, Cout);

        // Test Case 6: Overflow check
        A = 32'h80000000;
        B = 32'h80000000;
        Cin = 1'b0;
        #10;
        $display("TC6: A = %h, B = %h, Cin = %b | Sum = %h, Cout =
            %b", A, B, Cin, Sum, Cout);

        // Finish simulation
        $finish;
    end
```

```
endmodule
```

Design Choice: Combination of 4-bit and 16-bit CLAs

In the example provided, a combination of 4-bit CLAs and 16-bit CLAs was used to illustrate a practical approach to designing a 32-bit adder with Block Carry Lookahead Generators (BCLGs). Here's why this combination might be used:

1. Modular Design with Hierarchical Structure

- **4-bit CLAs:** By breaking down the adder into smaller 4-bit CLA blocks, you create a modular and manageable design. Each 4-bit CLA handles a small section of the addition, making the design easier to debug and verify.
- **16-bit CLAs:** These are used to combine the smaller 4-bit CLAs into larger, more efficient sections. This approach leverages the benefits of having fewer, larger blocks to manage carry propagation, thereby reducing the overall complexity compared to using many smaller blocks.

2. Performance Optimization

- **4-bit CLAs:** While suitable for handling small sections of addition, a single 4-bit CLA is limited in terms of its ability to handle larger bit widths efficiently on its own. It simplifies the carry propagation within its 4-bit section but might introduce more blocks and additional complexity when scaling up.
- **16-bit CLAs:** By using 16-bit CLAs, you consolidate the carry lookahead logic for a larger block, which can significantly reduce the carry propagation delay compared to using many 4-bit CLAs. This can improve overall performance, especially for larger bit-width adders.

3. Integration with Block Carry Lookahead Generators (BCLGs)

- **4-bit CLAs and BCLG:** When using 4-bit CLAs, a BCLG can be employed to manage carry propagation between these smaller blocks. This setup allows efficient handling of carry signals between 4-bit sections, optimizing performance and scalability.
- **16-bit CLAs:** For a more streamlined approach, a 16-bit CLA can be used to handle larger sections of addition directly, potentially reducing the need for extensive BCLG integration. This simplifies the design and can be more efficient for larger bit widths.

4. Design Complexity and Verification

- **4-bit CLAs:** Smaller blocks like 4-bit CLAs are easier to design, verify, and debug individually. They allow for incremental design and testing, which can simplify the development process.
- **16-bit CLAs:** By using fewer, larger blocks, you reduce the overall number of modules in the design. This can simplify the overall structure and reduce the complexity of integrating and verifying the design as a whole.

5. Practical Design Trade-offs

- **Scalability:** Using 4-bit CLAs allows for easier scaling by adding more blocks, while 16-bit CLAs offer better performance for larger sections but require more complex integration.
- **Efficiency:** Combining 4-bit and 16-bit CLAs allows for a balanced approach where the modular design benefits of 4-bit blocks are utilized, while the performance benefits of larger 16-bit blocks are also realized.

Summary

The combination of 4-bit and 16-bit CLAs in the design provides a balance between modularity, performance, and complexity. Using 4-bit CLAs allows for a modular and scalable approach, while 16-bit CLAs streamline the design for better performance in larger sections. This approach leverages the strengths of both techniques to create an efficient and manageable 32-bit adder design.

3.3 Simulation Results

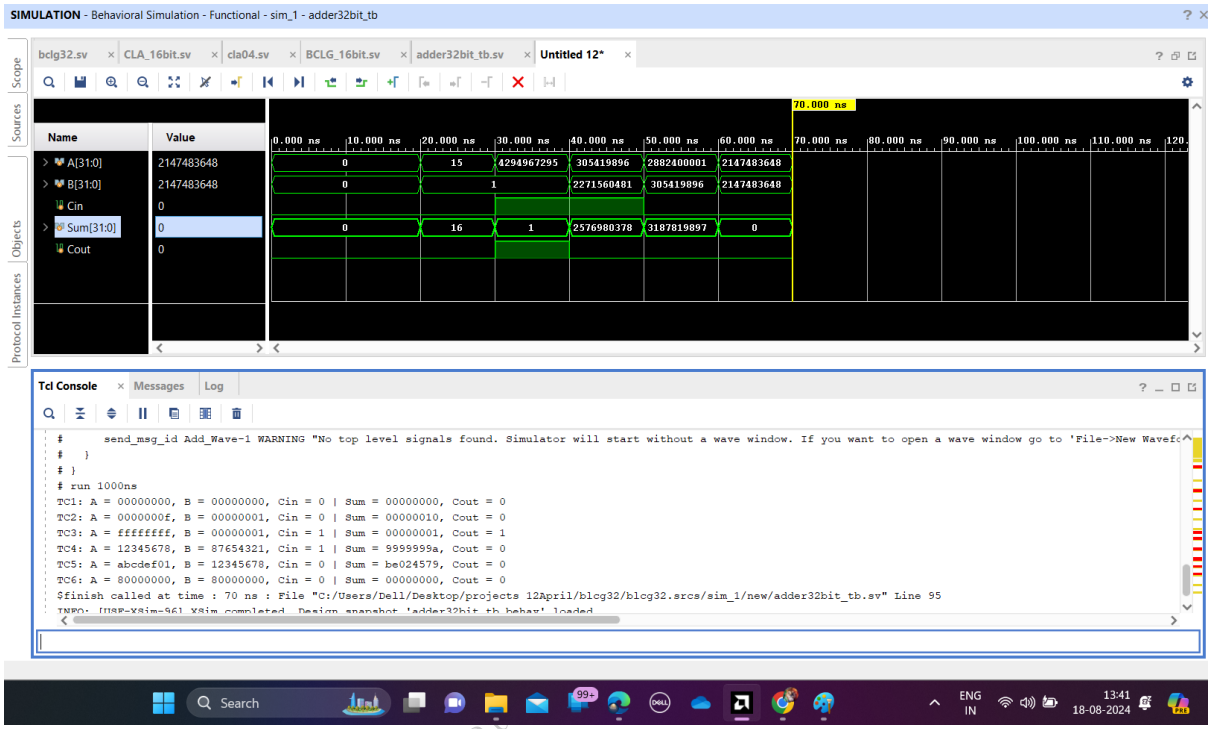

Figure 1: Simulation results of 32 bit adder

3.4 Schematic

3.5 Synthesis Design

3.6 Advantages

- **Speed:** Significantly reduces the carry propagation delay, allowing for faster arithmetic operations.
- **Scalability:** Makes it easier to implement wider bit-width adders by managing carry signals efficiently.

3.7 Disadvantages

- **Complexity:** Increases the design complexity due to additional logic for block carry lookahead generation.
- **Area and Power:** Requires more transistors, increasing the chip area and power consumption.

3.8 Applications

- **Large-scale Integrated Circuits:** Effective in implementing wide bit-width adders in complex digital systems.

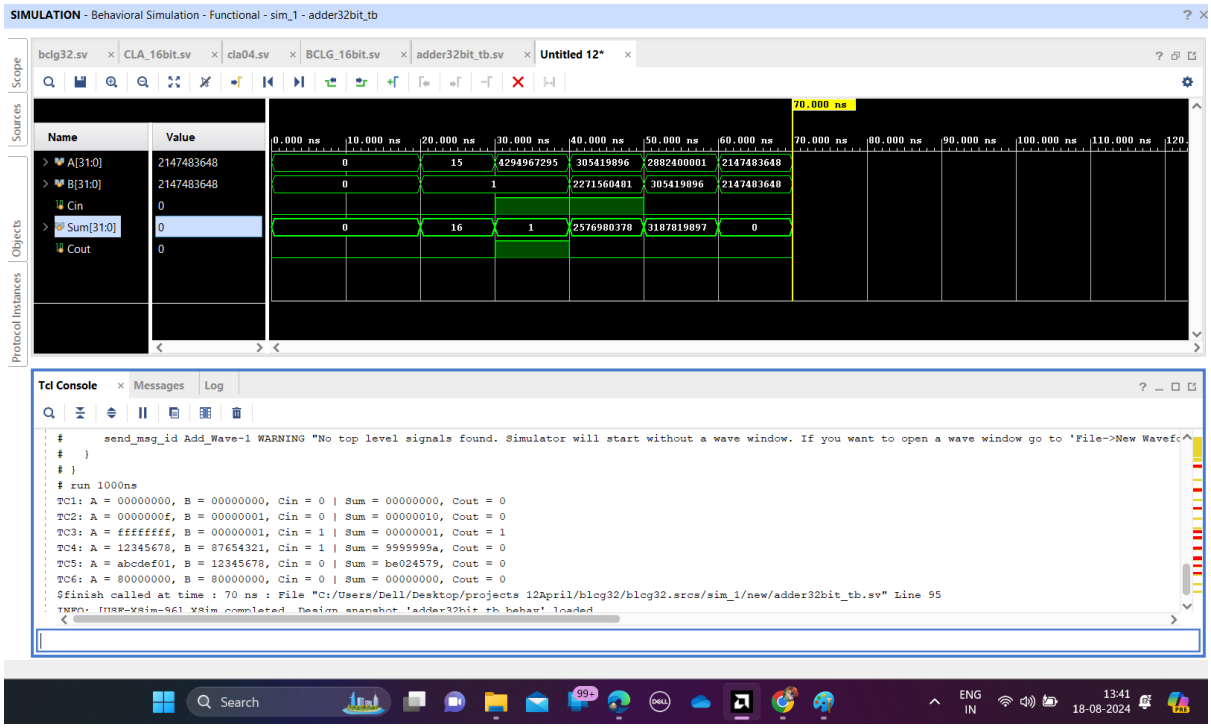

Figure 2: Schematic of 32 bit adder

- **High-speed Arithmetic Units**: Used in processors and digital signal processors (DSPs) where fast arithmetic operations are critical.

Carry Lookahead Adder (CLA Block Carry Lookahead Generator)

Bit Width Limitations

Standard CLA: Traditional CLAs can efficiently handle bit widths up to around 16-32 bits. Beyond this range, the complexity of the logic for generate and propagate signals increases, making the design and verification more challenging.

Practical Use: For typical applications, CLAs are commonly used up to 16 bits. Designs for wider adders are often optimized for speed and area, leveraging the benefits of parallel computation.

Block Carry Lookahead Generators (BCLGs)

Bit Width Scalability

Wider Bit Widths: BCLGs are designed to handle much wider bit widths efficiently. They are particularly useful for bit widths beyond 32 bits.

Scalability: The block-based approach allows BCLGs to manage the carry lookahead logic in smaller, manageable sections, improving scalability and maintainability. BCLGs can be implemented for bit widths exceeding 64 bits, with practical implementations reaching 128 bits or more in advanced digital systems.

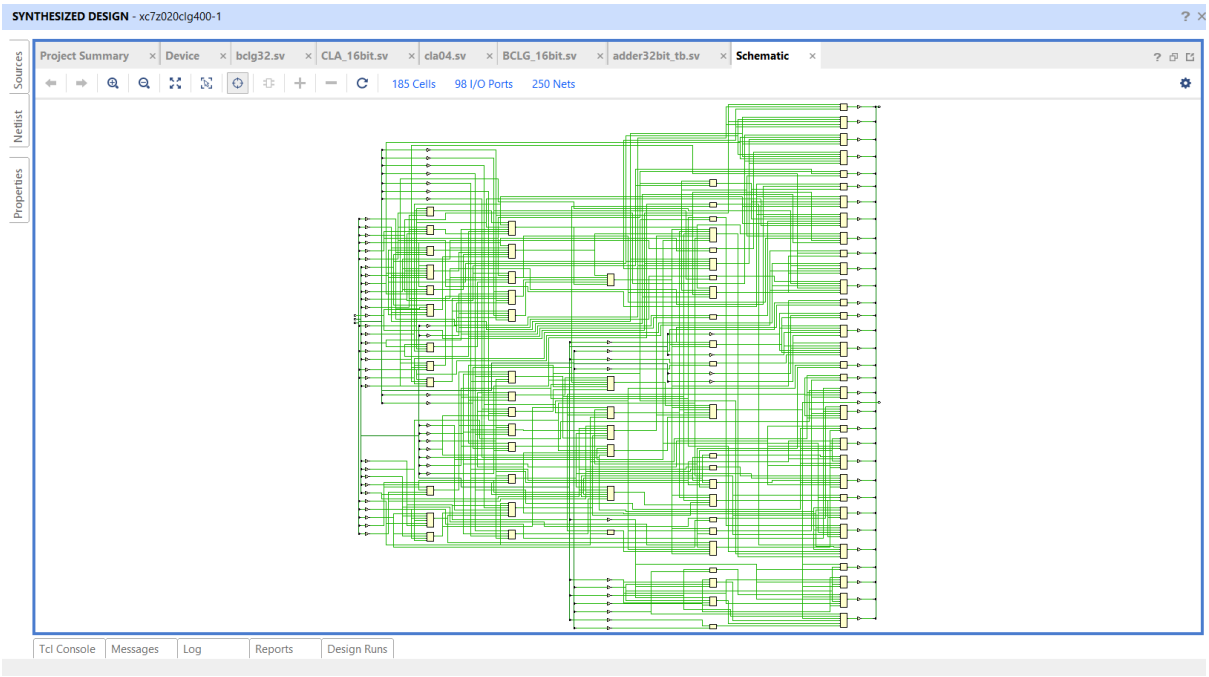

Figure 3: Synthesis Design of 32 bit adder

Practical Considerations

Design Complexity:

- **CLAs:** For bit widths up to 32 bits, standard CLA designs are generally straightforward and efficient.
- **BCLGs:** For wider widths, block-based designs are preferred. They simplify the carry computation process and reduce the overall delay.

Technology and Integration:

- **FPGA/ASIC Design:** In FPGA or ASIC implementations, the choice of CLA or BCLG depends on the technology's capability to handle large-scale logic. Modern FPGAs and ASICs are well-suited for designs up to 128 bits or more using BCLGs.

Performance and Trade-offs:

- **Speed vs. Complexity:** While CLAs are simpler, BCLGs offer better performance for very wide adders. The trade-off involves increased design complexity and area.

Example Use Cases

- **Up to 32 bits:** Standard CLA is usually sufficient.
- **64 bits and Beyond:** Block Carry Lookahead Generators are preferred, facilitating efficient carry computation for high-speed and wide bit-width adders.

Conclusion

- **Carry Lookahead Adders (CLA):** Effective up to 32 bits for most applications.
- **Block Carry Lookahead Generators (BCLG):** Scalable to 128 bits or more, suitable for high-performance and large-scale digital systems.