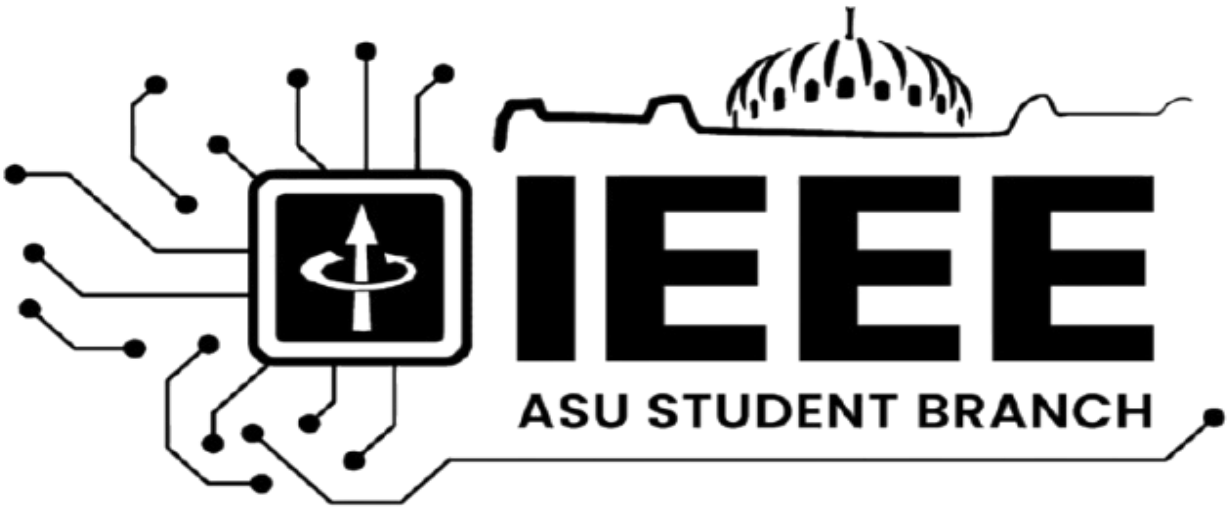
Author: Mustafa EL-Sherif



Generic Carry Look Ahead

DUT:

Sum Block:

```
module Sum_blk #(
    parameter N = 4
) (
    input  logic [N - 1 : 0]    p_i   ,
    input  logic [N - 1 : 0]    cin   ,

    output logic [N - 1 : 0]    sum_o
);

    assign sum_o = p_i ^ cin ;
endmodule
```

G/P Generator:

```
module gp_gen #(
    parameter DATA_WIDTH = 4
)(
    input  logic [DATA_WIDTH-1:0] x_i   ,
    input  logic [DATA_WIDTH-1:0] y_i   ,

    output logic [DATA_WIDTH-1:0] g_o   ,
    output logic [DATA_WIDTH-1:0] p_o
);

    // Generate signals
    assign g_o = x_i & y_i ;

    // Propagate signals
    assign p_o = x_i ^ y_i ;
endmodule
```

4-Bit CLA:

```
module cla_4bit (
    input  logic cin    ,
    input  logic g0_i   ,
    input  logic p0_i   ,
    input  logic g1_i   ,
    input  logic p1_i   ,
    input  logic g2_i   ,
    input  logic p2_i   ,
    input  logic g3_i   ,
    input  logic p3_i   ,
    output logic c1_o   ,
    output logic c2_o   ,
    output logic c3_o   ,
    output logic c4_o   ,
    output logic p_o    ,
    output logic g_o
);

    // Carry-out calculations
    assign c1_o = g0_i | (p0_i &
cin)
        ;

    assign c2_o = g1_i | (p1_i & g0_i) | (p1_i & p0_i &
cin)                                                    ;

    assign c3_o = g2_i | (p2_i & g1_i) | (p2_i & p1_i & g0_i) | (p2_i & p1_i & p0_i &
cin)                                    ;

    assign c4_o = g3_i | (p3_i & g2_i) | (p3_i & p2_i & g1_i) | (p3_i & p2_i & p1_i &
g0_i) | (p3_i & p2_i & p1_i & p0_i & cin) ;

    // Group Propagate and Generate
    assign p_o = p3_i & p2_i & p1_i &
p0_i                                            ;

    assign g_o = g3_i | (p3_i & g2_i) | (p3_i & p2_i & g1_i) | (p3_i & p2_i & p1_i &
g0_i)  ;

endmodule
```

Top:

```
module CLA_top #(
    parameter int DATA_WIDTH = 64  // Width of the adder in bits (default 64-bit)
)(
    input  logic [DATA_WIDTH-1:0] A,    // First operand input
    input  logic [DATA_WIDTH-1:0] B,    // Second operand input
    input  logic                  cin,  // Carry input
    output logic [DATA_WIDTH-1:0] sum,  // Sum output
    output logic                  cout  // Carry output
);

    // Calculate number of 4-bit groups needed (e.g., 64 bits / 4 = 16 groups)
    localparam int NUM_GROUPS = DATA_WIDTH / 4;
    // Calculate tree depth: log4(NUM_GROUPS) = log2(NUM_GROUPS)/2
    // Each level combines 4 groups from previous level into 1 group
    localparam int NUM_LEVELS = ($clog2(DATA_WIDTH)/2);

    // Bit-level signals from initial GP generation
    logic [DATA_WIDTH - 1 : 0] g_vector;  // Generate: G[i] = A[i] & B[i] (both bits are 1)
    logic [DATA_WIDTH - 1 : 0] p_vector;  // Propagate: P[i] = A[i] ^ B[i] (carry will propagate
through)

    // Hierarchical group-level GP signals
    // [NUM_LEVELS:0] represents levels 0 through NUM_LEVELS
    // Level 0: 4-bit groups, Level 1: 16-bit groups, Level 2: 64-bit groups, etc.
    logic [NUM_LEVELS : 0] g_group [NUM_GROUPS - 1 : 0];  // Group generate signals
    logic [NUM_LEVELS : 0] p_group [NUM_GROUPS - 1 : 0];  // Group propagate signals

    // Carry chain: carry_vector[i] is the carry into bit position i
    logic [DATA_WIDTH:0] carry_vector;

    // Generate bit-level G and P for all bit positions
    // G[i] = A[i] & B[i]: this position generates a carry
    // P[i] = A[i] ^ B[i]: this position will propagate an incoming carry
    gp_gen #(
        .DATA_WIDTH(DATA_WIDTH)
    ) GP_GENERATOR (
        .x_i (A)        ,
        .y_i (B)        ,
        .g_o  (g_vector)   ,
        .p_o  (p_vector)
    );

    // Initialize carry chain with input carry
    assign carry_vector[0] = cin;
```

```
    // ========== LEVEL 0: Process individual bits into 4-bit groups =========
    // For 64-bit: creates 16 groups, each handling 4 consecutive bits
    generate
        for (genvar i = 0 ; i < NUM_GROUPS ; i++) begin
            localparam int OFFSET = i * 4;  // Starting bit index: 0, 4, 8, 12, ...
            logic c1, c2, c3, c4;  // Intermediate carries within this 4-bit group

            // Each cla_4bit computes:
            // - Individual carries c1, c2, c3, c4 for bits within the group
            // - Group-level G and P that represent the entire 4-bit block
            // Group G = G3 | (P3&G2) | (P3&P2&G1) | (P3&P2&P1&G0)
            // Group P = P3 & P2 & P1 & P0
            cla_4bit gp4 (
                .cin(carry_vector[OFFSET]),  // Carry into this 4-bit group
                .g0_i(g_vector[OFFSET]),      // Bit 0 generate
                .p0_i(p_vector[OFFSET]),      // Bit 0 propagate
                .g1_i(g_vector[OFFSET + 1]),  // Bit 1 generate
                .p1_i(p_vector[OFFSET + 1]),  // Bit 1 propagate
                .g2_i(g_vector[OFFSET + 2]),  // Bit 2 generate
                .p2_i(p_vector[OFFSET + 2]),  // Bit 2 propagate
                .g3_i(g_vector[OFFSET + 3]),  // Bit 3 generate
                .p3_i(p_vector[OFFSET + 3]),  // Bit 3 propagate
                .c1_o(c1),  // Carry out of bit 0: C1 = G0 | (P0&Cin)
                .c2_o(c2),  // Carry out of bit 1: C2 = G1 | (P1&G0) | (P1&P0&Cin)
                .c3_o(c3),  // Carry out of bit 2
                .c4_o(c4),  // Carry out of bit 3 (= carry out of entire 4-bit group)
                .p_o(p_group[0][i]),  // Level 0 group propagate for group i
                .g_o(g_group[0][i])   // Level 0 group generate for group i
            );

            // Store computed carries back into the carry vector for sum calculation
            assign carry_vector[OFFSET + 1] = c1;
            assign carry_vector[OFFSET + 2] = c2;
            assign carry_vector[OFFSET + 3] = c3;
            assign carry_vector[OFFSET + 4] = c4;  // This becomes carry input for next group
        end
    endgenerate

    // ========== LEVELS 1 to NUM_LEVELS: Build hierarchical CLA tree =========
    // Each level combines 4 groups from the previous level into 1 parent group
    // Level 1: combines 4-bit groups into 16-bit groups
    // Level 2: combines 16-bit groups into 64-bit groups, etc.
    generate
        for (genvar j = 1 ; j <= NUM_LEVELS ; j++) begin : levels
            // Number of child groups from previous level
            // Level 1: NUM_GROUPS/4^0 = 16 (for 64-bit)
            // Level 2: NUM_GROUPS/4^1 = 4
            // Level 3: NUM_GROUPS/4^2 = 1
            localparam int NUM_CHILD = (NUM_GROUPS / 4**(j - 1));
```

```
        // Number of parent groups at this level
        // Level 1: NUM_GROUPS/4^1 = 4 (16-bit groups)
        // Level 2: NUM_GROUPS/4^2 = 1 (64-bit group)
        localparam int NUM_PARENT = (NUM_GROUPS / 4**(j));
        for (genvar k = 0; k < NUM_PARENT; k++) begin : parents
            // Index of first child group that feeds into this parent
            // Parent 0 uses children 0-3, Parent 1 uses children 4-7, etc.
            localparam int CHILD = k * 4;

            // Local GP signals for the 4 child groups
            logic g0, p0;  // Child group 0
            logic g1, p1;  // Child group 1
            logic g2, p2;  // Child group 2
            logic g3, p3;  // Child group 3

            // Fetch GP from child groups with bounds checking
            // If child doesn't exist (e.g., only 3 children), use default values
            // Default G=0 (no generate), P=1 (transparent propagate)
            assign g0 = (CHILD < NUM_CHILD)      ? g_group[j - 1][CHILD] : 1'b0;
            assign p0 = (CHILD < NUM_CHILD)      ? p_group[j - 1][CHILD] : 1'b1;
            assign g1 = (CHILD + 1 < NUM_CHILD)  ? g_group[j - 1][CHILD + 1] : 1'b0;
            assign p1 = (CHILD + 1 < NUM_CHILD)  ? p_group[j - 1][CHILD + 1] : 1'b1;
            assign g2 = (CHILD + 2 < NUM_CHILD)  ? g_group[j - 1][CHILD + 2] : 1'b0;
            assign p2 = (CHILD + 2 < NUM_CHILD)  ? p_group[j - 1][CHILD + 2] : 1'b1;
            assign g3 = (CHILD + 3 < NUM_CHILD)  ? g_group[j - 1][CHILD + 3] : 1'b0;
            assign p3 = (CHILD + 3 < NUM_CHILD)  ? p_group[j - 1][CHILD + 3] : 1'b1;

            // Combine 4 child groups into 1 parent group using CLA logic
            // Parent G = G3 | (P3&G2) | (P3&P2&G1) | (P3&P2&P1&G0)
            // Parent P = P3 & P2 & P1 & P0
            // No cin needed here - we're just combining GP signals, not computing carries
            cla_4bit COMB (
                .cin(1'b0),  // Cin=0 for GP combination (not computing actual carries)
                .g0_i(g0),
                .p0_i(p0),
                .g1_i(g1),
                .p1_i(p1),
                .g2_i(g2),
                .p2_i(p2),
                .g3_i(g3),
                .p3_i(p3),
                .c1_o(),     // Intermediate carries not needed - only final GP matters
                .c2_o(),
                .c3_o(),
                .c4_o(),
                .p_o(p_group[j][k]),  // Parent group propagate
                .g_o(g_group[j][k])   // Parent group generate
            );
        end
    end
endgenerate
```

```
    // Final carry out is the carry that would propagate beyond the last bit
    assign cout = carry_vector[DATA_WIDTH];

    // Compute final sum bits using: Sum[i] = P[i] ^ Carry[i]
    // P[i] = A[i] ^ B[i], so Sum[i] = (A[i] ^ B[i]) ^ Carry[i] = A[i] ^ B[i] ^ Carry[i]
    Sum_blk #(
        .N(DATA_WIDTH)
    ) SUM (
        .p_i   (p_vector),                  // Bit-level propagate (A ^ B)
        .cin   (carry_vector[DATA_WIDTH-1:0]),  // All carry signals (carry into each bit)
        .sum_o (sum)                        // Final sum = (A ^ B) ^ Carry
    );

endmodule
```

Verification:

Testbench:

```
module top_tb ();

    // Testbench parameter - can be overridden for different bit widths
    parameter DATA_WIDTH = 64;

    // DUT input signals
    logic [DATA_WIDTH-1:0] A;    // First operand for addition
    logic [DATA_WIDTH-1:0] B;    // Second operand for addition
    logic                  cin;  // Carry input

    // DUT output signals
    logic [DATA_WIDTH-1:0] sum;  // Sum result from CLA
    logic                  cout; // Carry output from CLA

    // Counters for test statistics
    int passed_tests = 0;  // Count of successful test cases
    int failed_tests = 0;  // Count of failed test cases

    // Expected result for comparison
    logic [DATA_WIDTH:0] expected_result;  // Full width to hold sum + carry
```

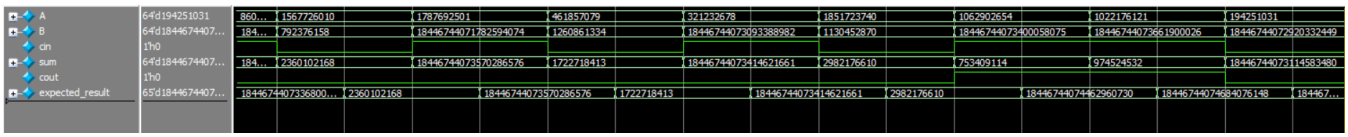
```
    // Instantiate the Device Under Test (DUT)
    CLA_top #(
        .DATA_WIDTH(DATA_WIDTH)  // Pass parameterized width to CLA
    ) DUT (
        .A   (A),    // Connect operand A
        .B   (B),    // Connect operand B
        .cin (cin),  // Connect carry input
        .sum (sum),  // Connect sum output
        .cout(cout)  // Connect carry output
    );




    // Main test sequence
    initial begin
        $display("======================================");
        $display("Starting CLA Testbench");
        $display("Data Width: %0d bits", DATA_WIDTH);
        $display("======================================\n");

        // ========== Test 1: Corner Cases ==========
        $display("Test 1: Corner Cases");

        // Test case 1a: All zeros (minimum values)
        A = 0; B = 0; cin = 0;
        #5;  // Wait for combinational logic to settle
        check_result("All zeros");
        #5;

        // Test case 1b: All ones (maximum values, will overflow)
        A = '1; B = '1; cin = 1;  // '1 creates all-ones pattern
        #5;
        check_result("All ones with carry");
        #5;

        // Test case 1c: Maximum value + 0
        A = '1; B = 0; cin = 0;
        #5;
        check_result("Max + 0");
        #5;

        // Test case 1d: Carry propagation test (alternating pattern)
        A = {DATA_WIDTH{1'b1}};  // All ones
        B = 1;                   // Add 1 to trigger full carry chain
        cin = 0;
        #5;
        check_result("Carry propagation chain");
        #5;
```

```
        // ========== Test 2: Power-of-2 Values ==========
        $display("\nTest 2: Power-of-2 Values");

        // Test addition of powers of 2 (single bit set)
        for (int i = 0; i < DATA_WIDTH; i++) begin
            A = (1 << i);  // 2^i
            B = 0;
            cin = 0;
            #5;
            check_result($sformatf("2^%0d", i));
            #5;
        end

        // ========== Test 3: Alternating Bit Patterns ==========
        $display("\nTest 3: Alternating Bit Patterns");

        // Pattern: 0x5555... (01010101...)
        A = {DATA_WIDTH{2'b01}};
        B = {DATA_WIDTH{2'b10}};  // 0xAAAA... (10101010...)
        cin = 0;
        #5;
        check_result("0x5555... + 0xAAAA...");
        #5;

        // Pattern: Same with carry
        A = {DATA_WIDTH{2'b01}};
        B = {DATA_WIDTH{2'b10}};
        cin = 1;
        #5;
        check_result("0x5555... + 0xAAAA... + 1");
        #5;

        // ========== Test 4: Sequential Values ==========
        $display("\nTest 4: Sequential Additions");

        // Test sequential numbers
        for (int i = 0; i < 20; i++) begin
            A = i;
            B = i + 1;
            cin = i % 2;  // Alternate carry input
            #5;
            check_result($sformatf("Sequential %0d", i));
            #5;
        end

        // ========== Test 5: Random Vectors ==========
        $display("\nTest 5: Random Test Vectors");

        // Comprehensive random testing
        repeat (1000) begin
            A = $random;    // Generate random value for A
```

```
            B = $random;    // Generate random value for B
            cin = $random;  // Random carry input (0 or 1)
            #5;

            // Verify result matches expected sum
            check_result("Random");

            #5;
        end

// ========== Test 6: Stress Test - Near Overflow ==========
        $display("\nTest 6: Near-Overflow Cases");

        // Test values near maximum
        for (int i = 0; i < 10; i++) begin
            A = ({DATA_WIDTH{1'b1}} - i);  // Max - i
            B = i;
            cin = $random % 2;
            #5;
            check_result($sformatf("Near-max %0d", i));
            #5;
        end

        // ========== Final Report ==========
        $display("\n=======================================");
        $display("Test Summary:");
        $display("  Passed: %0d", passed_tests);
        $display("  Failed: %0d", failed_tests);
        $display("  Total:  %0d", passed_tests + failed_tests);

        if (failed_tests == 0) begin
            $display("  Status: ALL TESTS PASSED ✓");
        end else begin
            $display("  Status: SOME TESTS FAILED ✗");
        end
        $display("=======================================\n");

        $stop;  // Stop simulation
    end
```

```
    // Task to check and report results
    // Compares DUT output against expected value and updates statistics
    task check_result(string test_name);
        // Calculate expected result: full width addition
        expected_result = A + B + cin;

        // Compare DUT output {cout, sum} with expected result
        if ({cout, sum} !== expected_result) begin
            // Mismatch detected - report error
            $error("[%s] FAIL: A=%0h, B=%0h, cin=%0b => {cout,sum}=%0h, expected=%0h",
                   test_name, A, B, cin, {cout, sum}, expected_result);
            failed_tests++;
        end else begin
            // Test passed
            $display("[%s] PASS: A=%0h + B=%0h + %0b = %0h",
                     test_name, A, B, cin, {cout, sum});
            passed_tests++;
        end
    endtask
endmodule
```

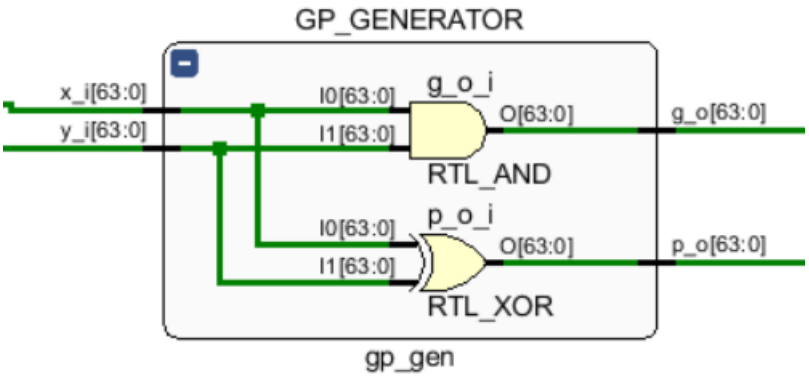
Waveform:

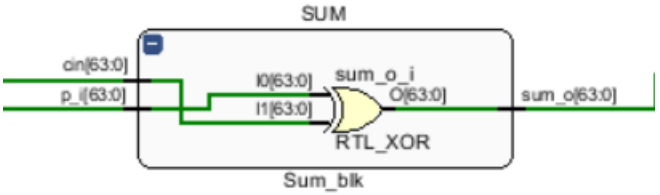


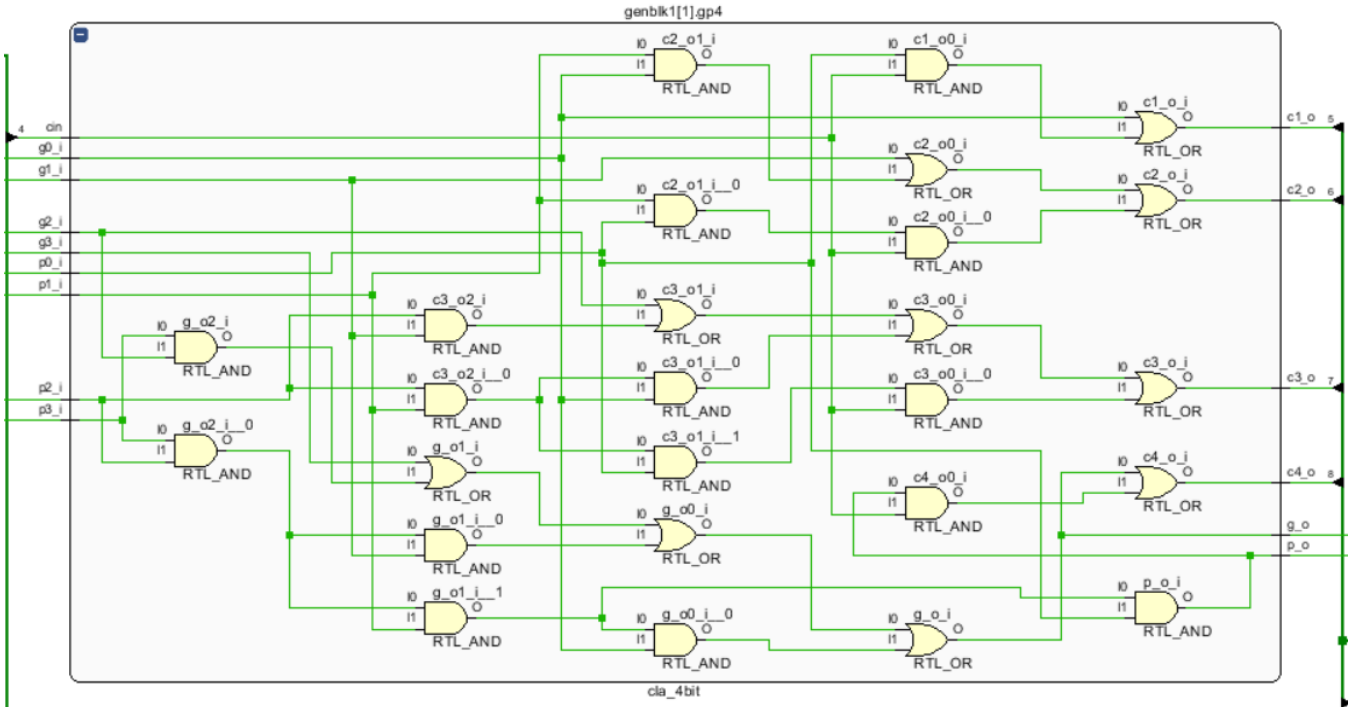

```
# ========================================
# Test Summary:
#   Passed: 1100
#   Failed: 0
#   Total:  1100
#   Status: ALL TESTS PASSED â
# ========================================
#
# ** Note: $finish    : CLA_tb.sv(166)
#    Time: 11 us  Iteration: 0  Instance: /top_tb
# 1
# Break in Module top_tb at CLA_tb.sv line 166
```

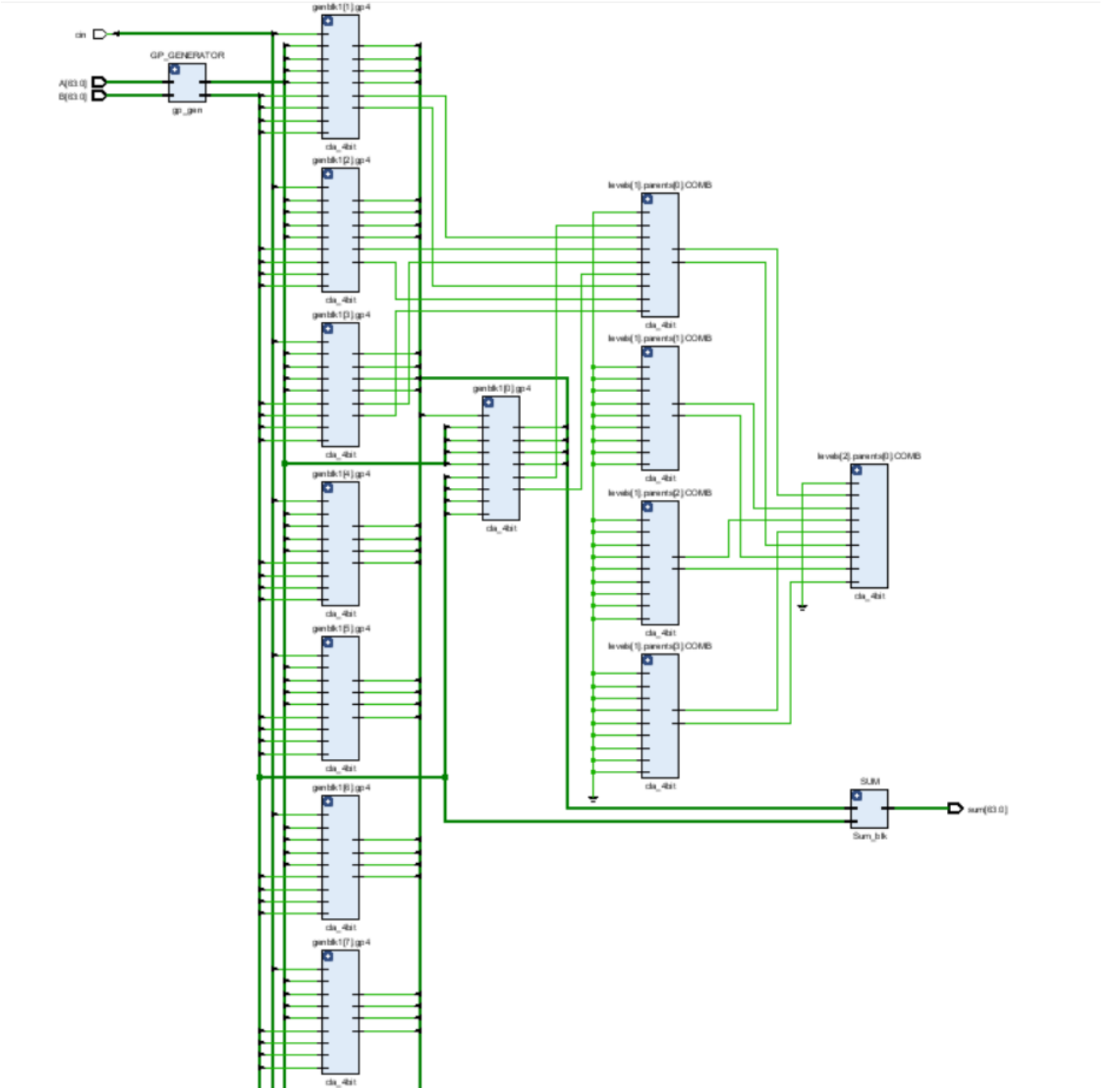
Vivado Results:

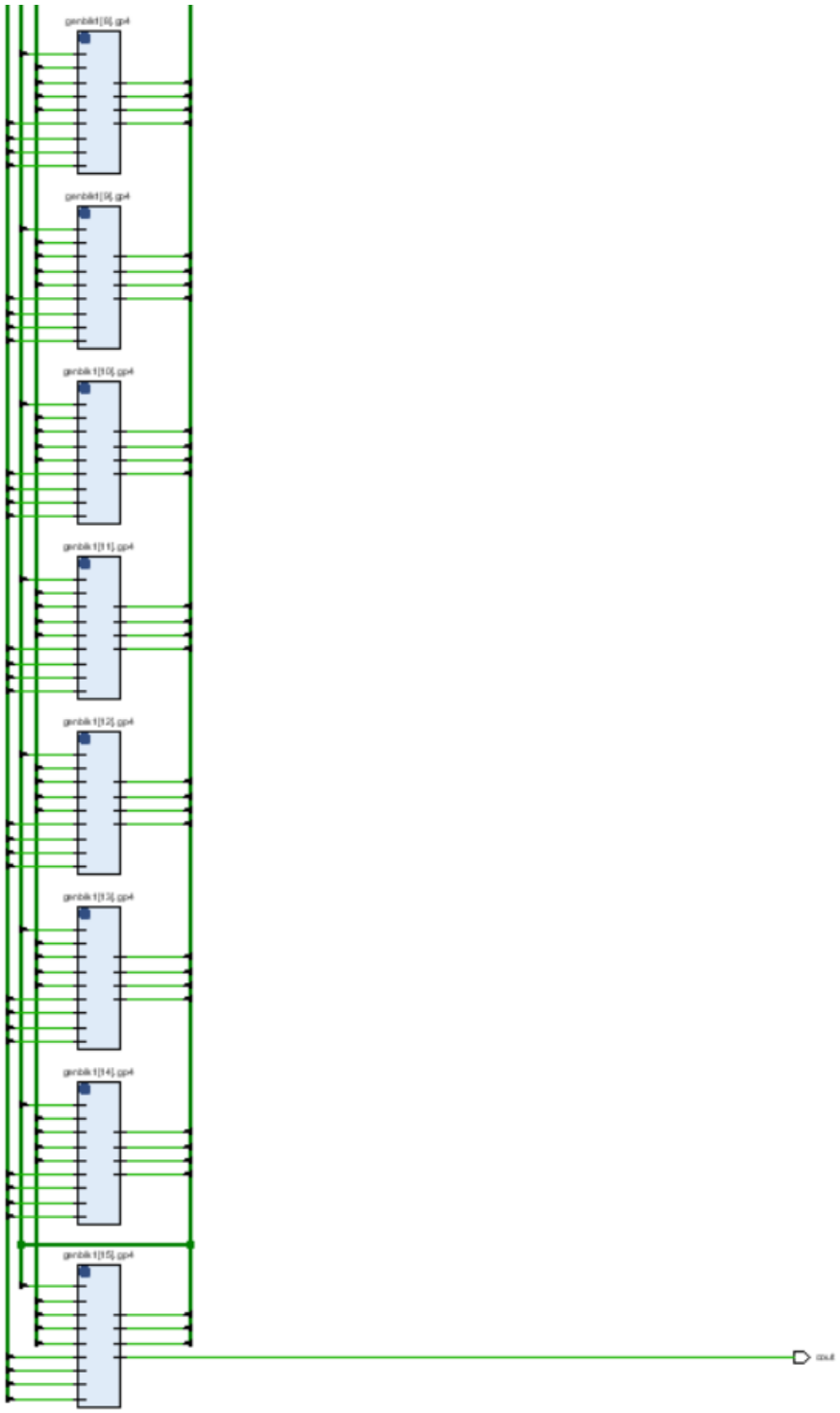
Elaboration:



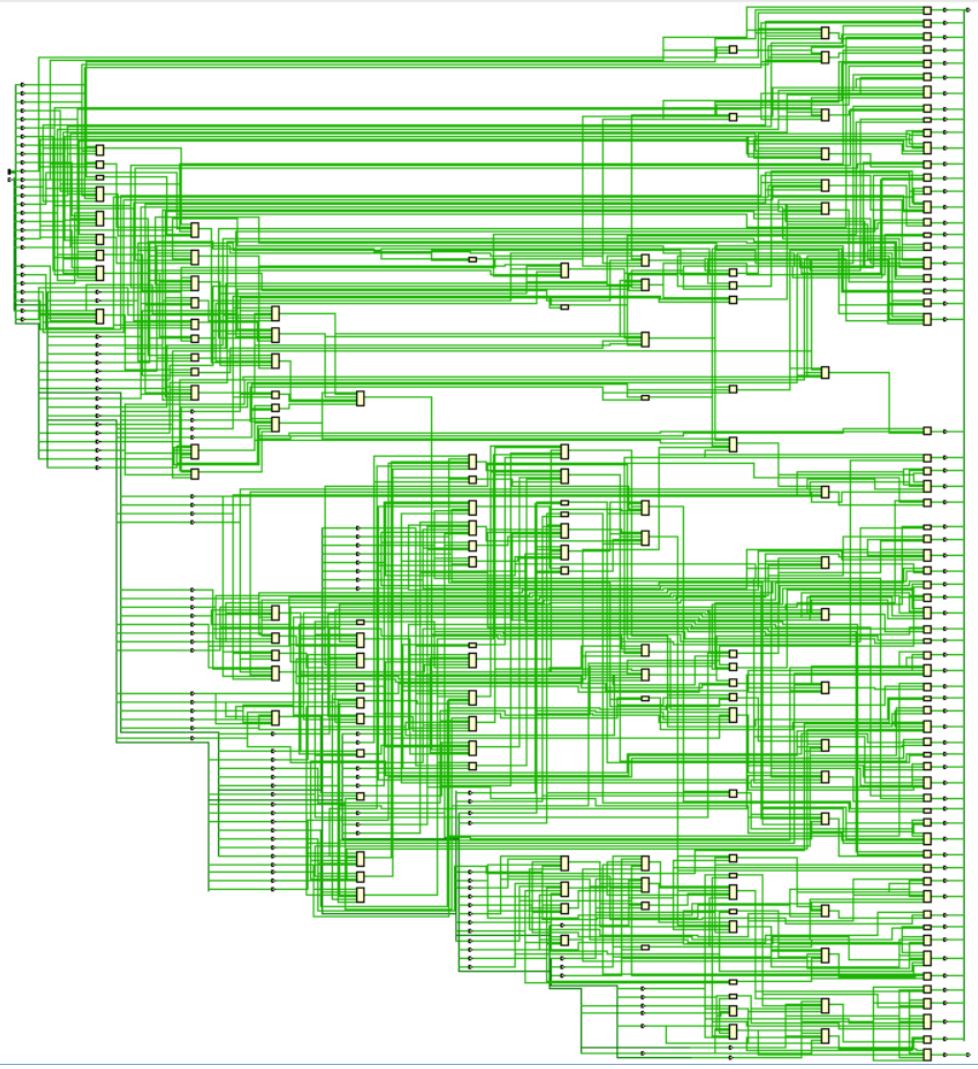





cin
GP_GENERATOR
A[63:0]
B[63:0]
gp_gen
genblk1[1].gp4
genblk1[2].gp4
genblk1[3].gp4
genblk1[4].gp4
genblk1[5].gp4
genblk1[6].gp4
genblk1[7].gp4
genblk1[0].gp4
cla_4bit
levels[1].parents[0].COMB
levels[1].parents[1].COMB
levels[1].parents[2].COMB
levels[1].parents[3].COMB
levels[2].parents[0].COMB
SUM
Sum_blk
sum[63:0]



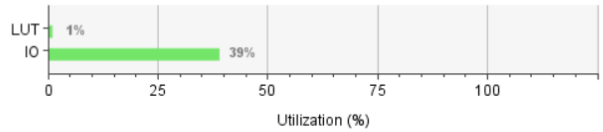
Synthesis:



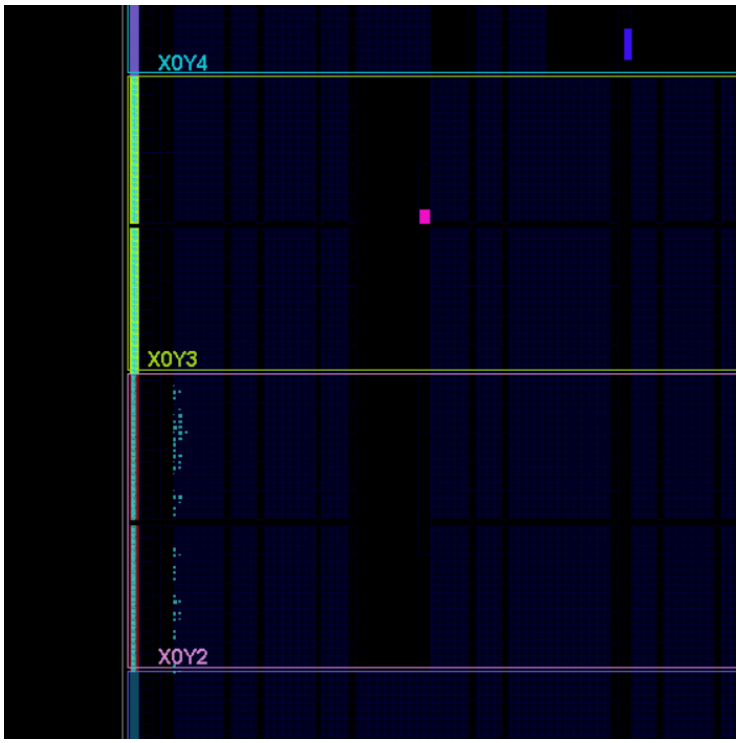
Synthesis Utilization Analysis:

Summary

Resource	Utilization	Available	Utilization %
LUT	137	134600	0.10
IO	194	500	38.80



Implementation:

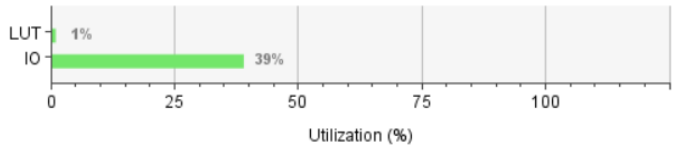


Implementation Utilization Analysis:

Name	Slice LUTs (134600)	Slice (33650)	LUT as Logic (134600)	Bonded IOB (500)
CLA_top	136	49	136	194

Summary

Resource	Utilization	Available	Utilization %
LUT	136	134600	0.10
IO	194	500	38.80



Limitations:

- Any value of the parameter “DATA_WIDTH” must be a logarithmic scale of 4 (e.g. 4, 16, 64, 256, … etc.) otherwise the design will not work as expected.

Contacts:

- All code sources, with the latest updates, and analysis are provided in my GitHub Mustafa11005/CLA_64_Bit
- Email: elsherifmustafa04@gmail.com
- LinkedIn: Mustafa El-Sherif | LinkedIn

Acknowledgments

[1] Computer arithmetic _ algorithms and hardware designs -- Parhami, Behrooz -- Oxford series in electrical and computer engineering, 2nd (Chapter 6)