Project 26 : Hybrid Adder

A Comprehensive Study of Advanced Digital Circuits

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

Contents

1 Project Overview

Hybrid Adders are digital circuits that combine different adder designs, like Ripple Carry Adders (RCA), Carry Look-Ahead Adders (CLA), and others, to optimize performance, area, or power consumption for specific bit-widths. By leveraging the strengths of multiple adder architectures, hybrid adders can achieve a balance between speed, complexity, and energy efficiency.

2 Hybrid Adder

2.1 Description

An 8-bit hybrid adder combines the strengths of multiple adder architectures, such as ripple carry adders (RCA) and carry look-ahead adders (CLA), to balance speed and area efficiency.

12-bit Hybrid Adders combine adders like RCA for lower significant bits and CLA for higher bits to minimize delay while keeping the design efficient. This balance is crucial for achieving faster addition while avoiding complex, power-hungry designs.

16-bit Hybrid Adders often use a mix of RCAs for the least significant bits and CLA or other faster adder architectures for the more significant bits to handle longer propagation delays effectively. These are designed to balance speed and area in more complex digital systems like processors.

2.2 Concepts of Hybrid Adder:

1. 8-bit Hybrid Adder

An 8-bit hybrid adder typically combines simpler and faster structures in a way that achieves an optimized performance. Here's how it might work:

- **Ripple Carry Adder for Lower Bits (4-bit RCA):** A ripple-carry adder can be used for the least significant bits (LSBs) where speed is less critical.
- **Carry Lookahead Adder for Upper Bits (4-bit CLA):** A carry lookahead adder is used for the most significant bits (MSBs) to minimize the delay caused by carry propagation.

The advantage here is that you benefit from the simplicity and small area of the RCA for lower bits while improving the speed with CLA for higher bits.

Characteristics:

- Architecture: Typically combines RCA for lower bits and CLA for higher bits.
- Speed: Faster than pure RCA due to the CLA speeding up the carry propagation.

2. 12-bit Hybrid Adder

For larger word sizes like 12 bits, the design has to balance speed and power efficiency. A 12-bit hybrid adder might be a combination of:

- Carry Look-Ahead Adder (CLA) for fast carry computation across groups of bits.
- Ripple Carry Adder (RCA) for low-area overhead in the lower bits where speed might not be as critical.

Designers often partition the 12-bit addition into smaller chunks, such as using a CLA for the higher significant bits (to optimize for speed) and RCA for the least significant bits.

Characteristics:

- Speed improvement: CLAs reduce delay in higher-order bits.
- Moderate area and power: Use of CLA might increase complexity and power, but this is mitigated by using RCA in lower bits.

- Flexible design: Can optimize performance for specific applications like digital signal processing (DSP) or control units.

3. 16-bit Hybrid Adder

A 16-bit hybrid adder typically leverages hierarchical or segmented designs to achieve high speed while managing area and power. A common approach involves:

- Carry Select Adders (CSAs) for fast operations by precomputing multiple scenarios.
- Carry Look-Ahead Adder (CLA) for speed improvements in large sections.
- Ripple Carry Adder (RCA) for simplicity and reduced hardware costs in lower bits.

For instance, the 16-bit adder can be split into two 8-bit adders (or four 4-bit adders). Each 8-bit adder can be a hybrid adder itself using CLA or CSA for high-speed carries, and RCAs for lower bits.

Characteristics:

- High speed: The combination of fast adders like CSA and CLA ensures quick carry propagation.
- Area overhead: Higher complexity means a larger area compared to smaller word sizes.
- Power efficiency: A balance between power consumption and performance is critical, especially in large-scale circuits like ALUs (Arithmetic Logic Units).

3 8-bit Hybrid Adder

3.1 Explanation

The 8-bit hybrid adder takes advantage of the strengths of both the Weinberger and Han-Carlson designs.

- **Lower Bits with Weinberger:** In this hybrid architecture, the lower significant bits (e.g., the least 4 or 5 bits) are computed using the Weinberger adder. These lower bits are typically where carry propagation delay is most critical. The Weinberger adder's efficient carry computation helps reduce the overall delay for these bits.
- **Higher Bits with Han-Carlson:** For the more significant bits (e.g., the upper 3 or 4 bits), we use the Han-Carlson adder. This approach reduces the area overhead by taking advantage of the Han-Carlson adder's balanced depth and area efficiency. Since the carries for higher bits are less critical in terms of speed, the Han-Carlson adder provides an optimal trade-off.

3.2 Working of the 8-bit Hybrid Adder

- **Input Signals:** Two 8-bit binary numbers A and B are provided as inputs.
- **Segmentation:** The bits are divided into groups (e.g., first 4 bits handled by the Wienerberger and the last 4 bits by the Han-Carlson).
- **Addition Process:**

Wienerberger Segment: The first 4 bits are processed using the Wienerberger adder. It calculates the generate (G) and propagate (P) signals, allowing for quick carry generation and propagation.

Han-Carlson Segment: The last 4 bits are processed using the Han-Carlson adder. It calculates potential sums based on the carry from the Wienerberger segment.

- **Final Output:** The outputs from both adders are combined to produce the final sum, and the carry-out is determined by the output of the most significant bit's adder.

3.3 RTL Code

Listing 1: 8-Bit Hybrid Adder

```
1
2
3  module hybrid8 (
4      input  logic [7:0] A, B,
5      input  logic       Cin,
6      output logic [7:0] Sum,
7      output logic       Cout
8  );
9      logic [3:0] Sum_HanCarlson, Sum_Weinberger;
10     logic Cout_HanCarlson, Cout_Weinberger;
11
12     // 4-bit Han-Carlson Adder
13     logic [3:0] G_HC, P_HC, C_HC;
14     assign G_HC = A[3:0] & B[3:0];     // Generate
15     assign P_HC = A[3:0] ^ B[3:0];     // Propagate
16
17     assign C_HC[0] = Cin;
18     assign C_HC[1] = G_HC[0]  (P_HC[0] & C_HC[0]);
19     assign C_HC[2] = G_HC[1]  (P_HC[1] & G_HC[0])  (P_HC[1] & P_HC[0]
           & C_HC[0]);
20     assign C_HC[3] = G_HC[2]  (P_HC[2] & G_HC[1])  (P_HC[2] & P_HC[1]
           & G_HC[0])  (P_HC[2] & P_HC[1] & P_HC[0] & C_HC[0]);
21
22     assign Sum_HanCarlson = P_HC ^ C_HC[3:0];
23     assign Cout_HanCarlson = G_HC[3]  (P_HC[3] & G_HC[2])  (P_HC[3] &
           P_HC[2] & G_HC[1])  (P_HC[3] & P_HC[2] & P_HC[1] & G_HC[0])
           (P_HC[3] & P_HC[2] & P_HC[1] & P_HC[0] & C_HC[0]);
24
25     // 4-bit Weinberger Adder
26     logic [3:0] G_W, P_W, C_W;
27     assign G_W = A[7:4] & B[7:4];     // Generate
28     assign P_W = A[7:4] ^ B[7:4];     // Propagate
29
30     assign C_W[0] = Cout_HanCarlson;
31     assign C_W[1] = G_W[0]  (P_W[0] & C_W[0]);
32     assign C_W[2] = G_W[1]  (P_W[1] & G_W[0])  (P_W[1] & P_W[0] &
           C_W[0]);
33     assign C_W[3] = G_W[2]  (P_W[2] & G_W[1])  (P_W[2] & P_W[1] &
           G_W[0])  (P_W[2] & P_W[1] & P_W[0] & C_W[0]);
34
35     assign Sum_Weinberger = P_W ^ C_W[3:0];
36     assign Cout_Weinberger = G_W[3]  (P_W[3] & G_W[2])  (P_W[3] &
           P_W[2] & G_W[1])  (P_W[3] & P_W[2] & P_W[1] & G_W[0])  (P_W[3]
           & P_W[2] & P_W[1] & P_W[0] & C_W[0]);
37
38     // 8-bit Hybrid Adder Outputs
39     assign Sum = {Sum_Weinberger, Sum_HanCarlson};
40     assign Cout = Cout_Weinberger;
41 endmodule
```

3.4 Testbench

Listing 2: 8-Bit Hybrid Adder

```
1
2  module hybrid8tb1;
3      logic [7:0] A, B;    // Inputs
4      logic Cin;           // Carry-in
5      logic [7:0] Sum;     // Sum output
6      logic Cout;          // Carry-out
7
8      // Instantiate the 8-bit Hybrid Adder
9      hybrid8 a (
10         .A(A),
11         .B(B),
12         .Cin(Cin),
13         .Sum(Sum),
14         .Cout(Cout)
15     );
16
17     initial begin
18         // Initialize inputs
19         A = 8'b00000000;
20         B = 8'b00000000;
21         Cin = 1'b0;
22
23         // Apply test cases with delays
24         #10 A = 8'b00001111; B = 8'b00001111; Cin = 1'b0;  // Case 1:
               Adding two small numbers
25         #10 A = 8'b11110000; B = 8'b00001111; Cin = 1'b1;  // Case 2:
               Adding a large and small number with carry-in
26         #10 A = 8'b10101010; B = 8'b01010101; Cin = 1'b0;  // Case 3:
               Alternating bits
27         #10 A = 8'b11111111; B = 8'b00000001; Cin = 1'b0;  // Case 4:
               Maximum value addition without carry
28         #10 A = 8'b11111111; B = 8'b11111111; Cin = 1'b1;  // Case 5:
               Overflow case
29
30         // End the simulation after the test cases
31         #10 $stop;
32     end
33
34     // Monitor the signals
35     initial begin
36         $monitor("At time %0d: A = %b, B = %b, Cin = %b, Sum = %b,
               Cout = %b",
37                  $time, A, B, Cin, Sum, Cout);
38     end
39 endmodule
```

3.5 Simulation

Figure 1: Simulation of 8-Bit Hybrid Adder

3.6 Schematic

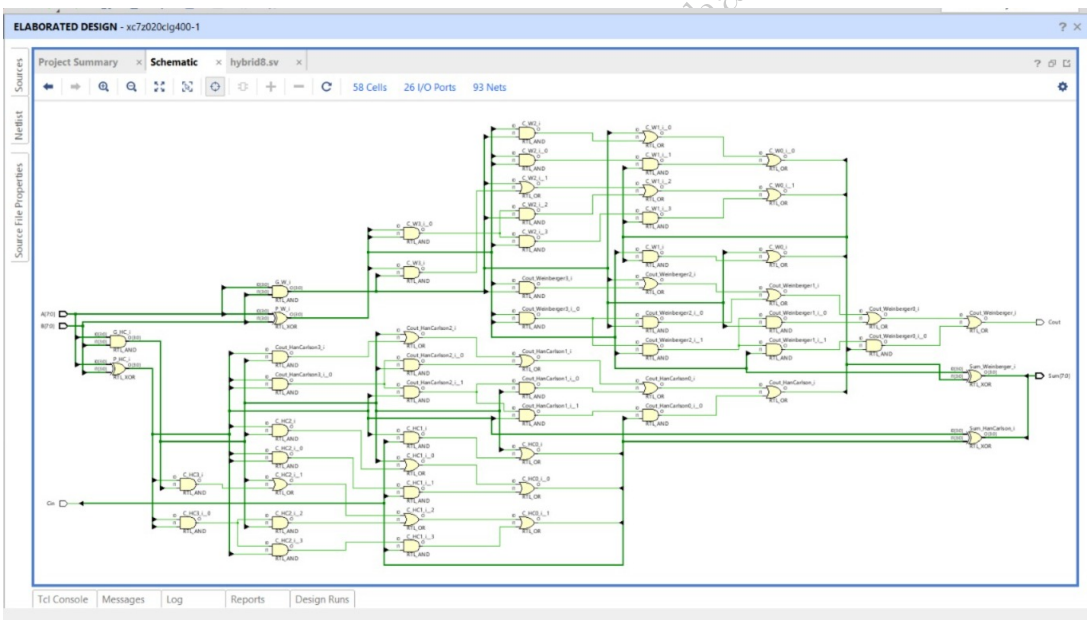

Figure 2: Schematic of 8-Bit Hybrid Adder

3.7 Synthesis Design

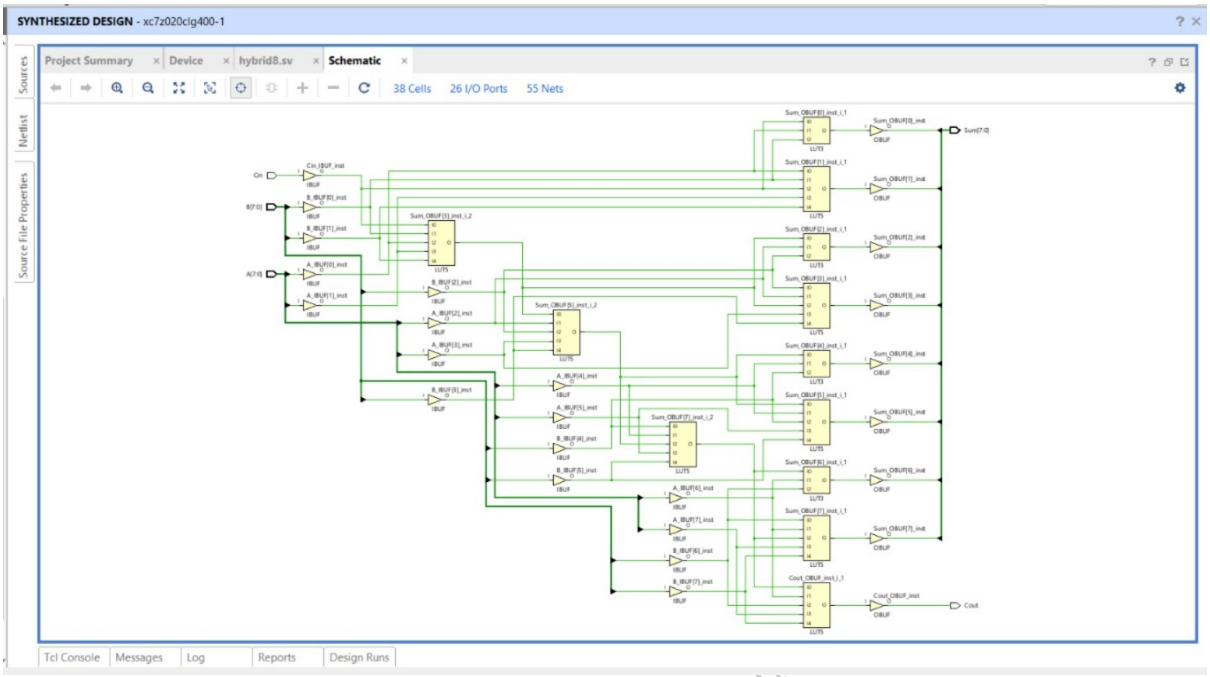

Figure 3: Synthesis Design of 8-Bit Hybrid Adder

4 12-Bit Hybrid Adder

4.1 Explanation

Wienberger Adder:

- A carry-save adder that can efficiently handle multiple inputs by reducing the carry propagation. It typically uses three inputs and two outputs (sum and carry).
- In a hybrid configuration, two Wienberger adders can be employed to sum the first 6 bits of each input number, producing two 6-bit outputs (sum and carry).

Han-Carlson Adder:

- A type of carry-lookahead adder designed for faster addition by pre-computing carry bits. It can manage larger bit-width inputs effectively by distributing carry calculations.
- In this setup, two Han-Carlson adders are used to handle the outputs of the Wienberger adders and the next set of bits.

4.2 Working of a 12-bit Hybrid Adder

Input Segmentation:

Divide the 12-bit inputs into segments. For instance, you can divide them as follows:

- Bits 11-8: Processed by the Ling adder
- Bits 7-4: Processed by the Wienberger adder
- Bits 3-0: Processed by the Han-Carlson adder

Addition Process:

- **MSBs (Ling Adder):**

The Ling adder computes the sum of the first four bits (bits 11 to 8) and generates a carry-out.

- **Middle Bits (Wienberger Adder):**

The Wienberger adder takes the next four bits (bits 7 to 4) and incorporates the carry from the Ling adder, calculating its own carry-out.

- **LSBs (Han-Carlson Adder):**

The Han-Carlson adder computes the final four bits (bits 3 to 0), taking into account the carry from the Wienerberger adder.

Final Carry Propagation:

After calculating the sums for each segment, the carries are propagated through the segments, ensuring that each adder's carry-out is accounted for in the next segment's input.

- **Output:**

The final output will be the combined result of the three adders, providing the 12-bit sum of the two inputs A and B.

4.3 RTL Code

Listing 3: 12-Bit Hybrid Adder

```
1
2  module hybrid12(
3      input  logic [11:0] A, B,
4      input  logic        Cin,
5      output logic [11:0] Sum,
6      output logic        Cout
7  );
8      logic [3:0] Sum_Ling, Sum_Weinberger, Sum_HanCarlson;
9      logic Cout_Ling, Cout_Weinberger, Cout_HanCarlson;
10
11     // 4-bit Ling Adder
12     logic [3:0] G_Ling, P_Ling, C_Ling;
13     assign G_Ling = A[3:0] & B[3:0];  // Generate
14     assign P_Ling = A[3:0] ^ B[3:0];  // Propagate
15
16     assign C_Ling[0] = Cin;
17     assign C_Ling[1] = G_Ling[0]  (P_Ling[0] & C_Ling[0]);
18     assign C_Ling[2] = G_Ling[1]  (P_Ling[1] & G_Ling[0])  (P_Ling[1]
           & P_Ling[0] & C_Ling[0]);
19     assign C_Ling[3] = G_Ling[2]  (P_Ling[2] & G_Ling[1])  (P_Ling[2]
           & P_Ling[1] & G_Ling[0])  (P_Ling[2] & P_Ling[1] & P_Ling[0] &
           C_Ling[0]);
20
21     assign Sum_L
22 [1:41 PM, 10/11/2024] Abhishek: module hybrid16 (
23     input  logic [15:0] A, B,
24     input  logic        Cin,
25     output logic [15:0] Sum,
26     output logic        Cout
27 );
28
29     logic [3:0] Sum_HanCarlson1, Sum_Weinberger1, Sum_Weinberger2,
           Sum_HanCarlson2;
```

```
30      logic Cout_HanCarlson1, Cout_Weinberger1, Cout_Weinberger2,
            Cout_HanCarlson2;
31
32      // 4-bit Han-Carlson Adder with Binary to Excess-1 Converter (BEC)
33      logic [3:0] G_HC1, P_HC1, C_HC1;
34      assign G_HC1 = A[3:0] & B[3:0];  // Generate
35      assign P_HC1 = A[3:0] ^ B[3:0];  // Propagate
36
37      assign C_HC1[0] = Cin;
38      assign C_HC1[1] = G_HC1[0]  (P_HC1[0] & C_HC1[0]);
39      assign C_HC1[2] = G_HC1[1]  (P_HC1[1] & G_HC1[0])  (P_HC1[1] &
            P_HC1[0] & C_HC1[0]);
40      assign C_HC1[3] = G_HC1[2]  (P_HC1[2] & G_HC1[1])  (P_HC1[2] &
            P_HC1[1] & G_HC1[0])  (P_HC1[2] & P_HC1[1] & P_HC1[0] &
            C_HC1[0]);
41
42      assign Sum_HanCarlson1 = P_HC1 ^ C_HC1[3:0];
43      assign Cout_HanCarlson1 = G_HC1[3]  (P_HC1[3] & G_HC1[2])
            (P_HC1[3] & P_HC1[2] & G_HC1[1])  (P_HC1[3] & P_HC1[2] &
            P_HC1[1] & G_HC1[0])  (P_HC1[3] & P_HC1[2] & P_HC1[1] &
            P_HC1[0] & C_HC1[0]);
44
45      // BEC for the Han-Carlson result (4-bit Binary to Excess-1
            Converter)
46      function automatic logic [3:0] BEC (input logic [3:0] in);
47          return in + 1;
48      endfunction
49
50      // 4-bit Weinberger Adder 1
51      logic [3:0] G_W1, P_W1, C_W1;
52      assign G_W1 = A[7:4] & B[7:4];   // Generate
53      assign P_W1 = A[7:4] ^ B[7:4];   // Propagate
54
55      assign C_W1[0] = Cout_HanCarlson1;
56      assign C_W1[1] = G_W1[0]  (P_W1[0] & C_W1[0]);
57      assign C_W1[2] = G_W1[1]  (P_W1[1] & G_W1[0])  (P_W1[1] & P_W1[0]
            & C_W1[0]);
58      assign C_W1[3] = G_W1[2]  (P_W1[2] & G_W1[1])  (P_W1[2] & P_W1[1]
            & G_W1[0])  (P_W1[2] & P_W1[1] & P_W1[0] & C_W1[0]);
59
60      assign Sum_Weinberger1 = P_W1 ^ C_W1[3:0];
61      assign Cout_Weinberger1 = G_W1[3]  (P_W1[3] & G_W1[2])  (P_W1[3] &
            P_W1[2] & G_W1[1])  (P_W1[3] & P_W1[2] & P_W1[1] & G_W1[0])
            (P_W1[3] & P_W1[2] & P_W1[1] & P_W1[0] & C_W1[0]);
62
63      // 4-bit Weinberger Adder 2
64      logic [3:0] G_W2, P_W2, C_W2;
65      assign G_W2 = A[11:8] & B[11:8]; // Generate
66      assign P_W2 = A[11:8] ^ B[11:8]; // Propagate
67
68      assign C_W2[0] = Cout_Weinberger1;
69      assign C_W2[1] = G_W2[0]  (P_W2[0] & C_W2[0]);
70      assign C_W2[2] = G_W2[1]  (P_W2[1] & G_W2[0])  (P_W2[1] & P_W2[0]
            & C_W2[0]);
71      assign C_W2[3] = G_W2[2]  (P_W2[2] & G_W2[1])  (P_W2[2] & P_W2[1]
            & G_W2[0])  (P_W2[2] & P_W2[1] & P_W2[0] & C_W2[0]);
72
73      assign Sum_Weinberger2 = P_W2 ^ C_W2[3:0];
```

```
74      assign Cout_Weinberger2 = G_W2[3] | (P_W2[3] & G_W2[2]) | (P_W2[3] &
            P_W2[2] & G_W2[1]) | (P_W2[3] & P_W2[2] & P_W2[1] & G_W2[0])
            | (P_W2[3] & P_W2[2] & P_W2[1] & P_W2[0] & C_W2[0]);
75
76      // 4-bit Han-Carlson Adder (Final)
77      logic [3:0] G_HC2, P_HC2, C_HC2;
78      assign G_HC2 = A[15:12] & B[15:12]; // Generate
79      assign P_HC2 = A[15:12] ^ B[15:12]; // Propagate
80
81      assign C_HC2[0] = Cout_Weinberger2;
82      assign C_HC2[1] = G_HC2[0] | (P_HC2[0] & C_HC2[0]);
83      assign C_HC2[2] = G_HC2[1] | (P_HC2[1] & G_HC2[0]) | (P_HC2[1] &
            P_HC2[0] & C_HC2[0]);
84      assign C_HC2[3] = G_HC2[2] | (P_HC2[2] & G_HC2[1]) | (P_HC2[2] &
            P_HC2[1] & G_HC2[0]) | (P_HC2[2] & P_HC2[1] & P_HC2[0] &
            C_HC2[0]);
85
86      assign Sum_HanCarlson2 = P_HC2 ^ C_HC2[3:0];
87      assign Cout_HanCarlson2 = G_HC2[3] | (P_HC2[3] & G_HC2[2]) |
            (P_HC2[3] & P_HC2[2] & G_HC2[1]) | (P_HC2[3] & P_HC2[2] &
            P_HC2[1] & G_HC2[0]) | (P_HC2[3] & P_HC2[2] & P_HC2[1] &
            P_HC2[0] & C_HC2[0]);
88
89      // 16-bit Hybrid Adder Outputs
90      assign Sum = {Sum_HanCarlson2, Sum_Weinberger2, Sum_Weinberger1,
            BEC(Sum_HanCarlson1)};
91      assign Cout = Cout_HanCarlson2;
92
93  endmodule
```


4.4 Testbench

Listing 4: 12-Bit Hybrid Adder

```
1
2   module hybrid12tb;
3       logic [11:0] A, B;  // Inputs
4       logic Cin;          // Carry-in
5       logic [11:0] Sum;   // Sum output
6       logic Cout;         // Carry-out
7
8       // Instantiate the 12-bit Hybrid Adder
9       hybrid12 a (
10          .A(A),
11          .B(B),
12          .Cin(Cin),
13          .Sum(Sum),
14          .Cout(Cout)
15      );
16
17      initial begin
18          // Initialize inputs
19          A = 12'b000000000000;
20          B = 12'b000000000000;
21          Cin = 1'b0;
22
23          // Apply test cases with delays
```

```
24          #10 A = 12'b000011110000; B = 12'b000011110000; Cin = 1'b0;
                // Case 1: Small numbers
25          #10 A = 12'b111100001111; B = 12'b000011110000; Cin = 1'b1;
                // Case 2: Mixed values with carry-in
26          #10 A = 12'b101010101010; B = 12'b010101010101; Cin = 1'b0;
                // Case 3: Alternating bits
27          #10 A = 12'b111111111111; B = 12'b000000000001; Cin = 1'b0;
                // Case 4: Max value addition without carry
28          #10 A = 12'b111111111111; B = 12'b111111111111; Cin = 1'b1;
                // Case 5: Overflow case
29
30          // End simulation after the test cases
31          #10 $stop;
32      end
33
34      // Monitor signals
35      initial begin
36          $monitor("At time %0d: A = %b, B = %b, Cin = %b, Sum = %b,
                Cout = %b",
37                   $time, A, B, Cin, Sum, Cout);
38      end
39  endmodule
```

4.5 Simulation

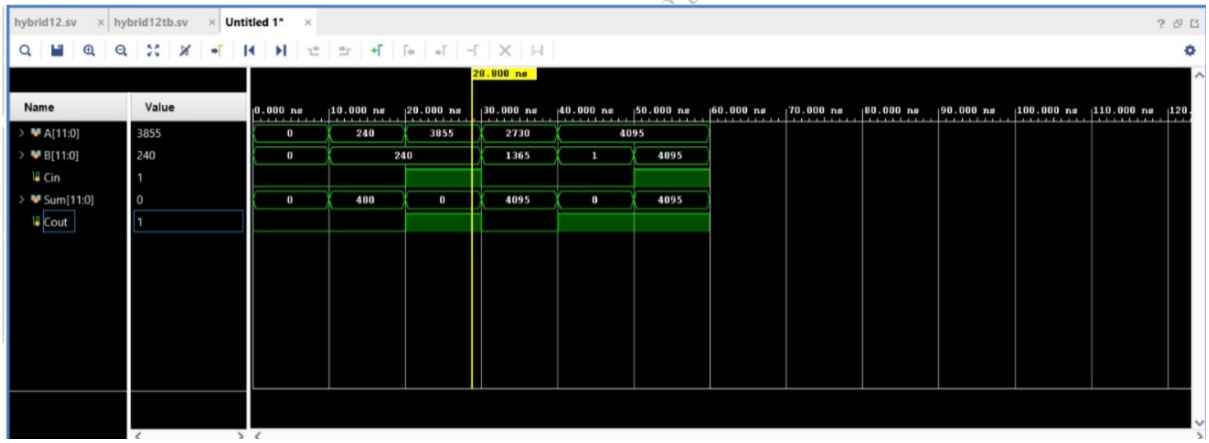

Figure 4: Simulation of 12-Bit Hybrid Adder

4.6 Schematic

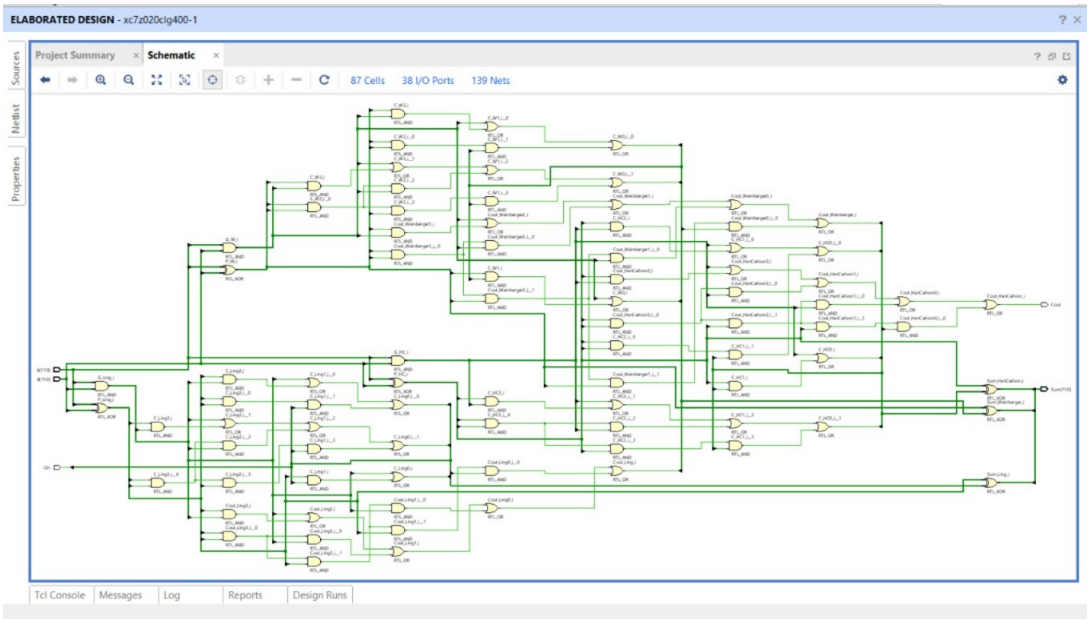

Figure 5: Schematic of 12-Bit Hybrid Adder

4.7 Synthesis Design

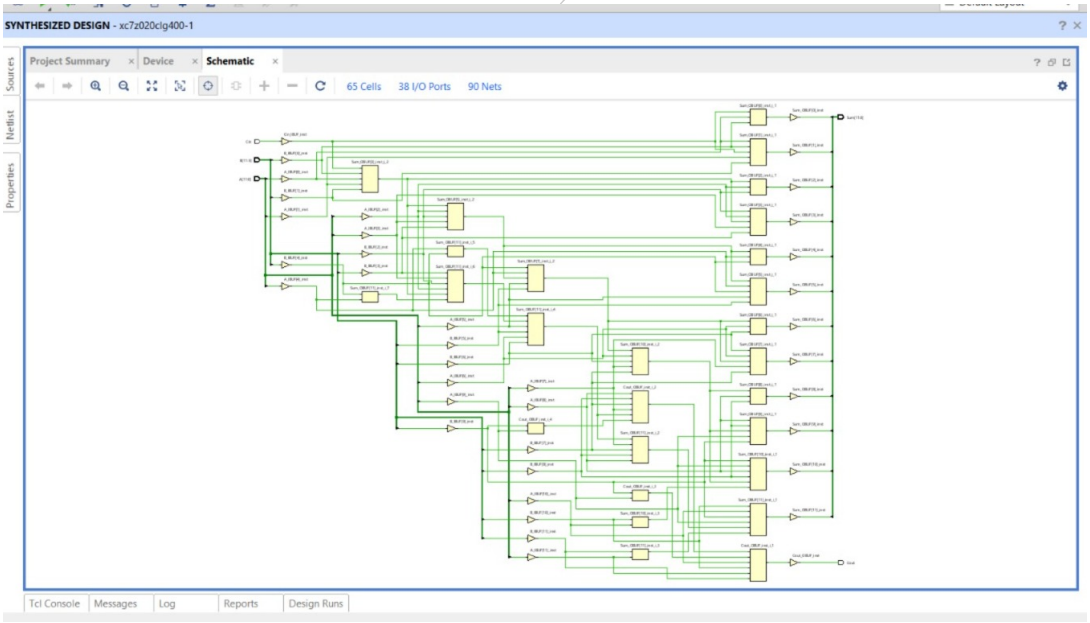

Figure 6: Synthesis Design of 12-Bit Hybrid Adder

5 16-Bit Hybrid Adder

5.1 Explanation

Wienerberger Adders: These are used for the higher bits due to their efficient carry propagation mechanism. Typically, two 4-bit Wienerberger adders are used:

- The first adder adds the two LSBs (A0 and B0) to produce a sum (S0) and a carry-out (C0).
- The second adder processes the next higher bits (A1 and B1) along with the carry from the first adder.

Han-Carlson Adders: These are used for the lower bits since they can handle smaller groups of bits efficiently:

- The first Han-Carlson adder takes A2 and B2 as inputs and adds them, considering the carry from the first Wienerberger adder.
- The second Han-Carlson adder adds A3 and B3, factoring in the carry from the second Wienerberger adder.

5.2 Working of a 16-bit Hybrid Adder

Wienerberger Addition:

- W1 computes the sum S1 and carry C1 from the first 8 bits.
- W2 computes the sum S2 and carry C2 from the next 8 bits.
- The carry C1 from W1 and C2 from W2 are propagated to the Han-Carlson adders.

Han-Carlson Addition:

- The outputs S1 and S2 from the Wienerberger adders are input to the two Han-Carlson adders, which compute their sums.
- The Han-Carlson adders will use the carry outputs from the Wienerberger adders as needed.

Final Output:

- The outputs from the Han-Carlson adders provide the final 16-bit sum, combining all segments and carries.

5.3 RTL Code

Listing 5: 16-Bit Hybrid Adder

```
1
2
3  module hybrid16 (
4      input  logic [15:0] A, B,
5      input  logic        Cin,
6      output logic [15:0] Sum,
7      output logic        Cout
8  );
9
10     logic [3:0] Sum_HanCarlson1, Sum_Weinberger1, Sum_Weinberger2,
           Sum_HanCarlson2;
11     logic Cout_HanCarlson1, Cout_Weinberger1, Cout_Weinberger2,
           Cout_HanCarlson2;
```

```
12
13      // 4-bit Han-Carlson Adder with Binary to Excess-1 Converter (BEC)
14      logic [3:0] G_HC1, P_HC1, C_HC1;
15      assign G_HC1 = A[3:0] & B[3:0];  // Generate
16      assign P_HC1 = A[3:0] ^ B[3:0];  // Propagate
17
18      assign C_HC1[0] = Cin;
19      assign C_HC1[1] = G_HC1[0]  (P_HC1[0] & C_HC1[0]);
20      assign C_HC1[2] = G_HC1[1]  (P_HC1[1] & G_HC1[0])  (P_HC1[1] &
           P_HC1[0] & C_HC1[0]);
21      assign C_HC1[3] = G_HC1[2]  (P_HC1[2] & G_HC1[1])  (P_HC1[2] &
           P_HC1[1] & G_HC1[0])  (P_HC1[2] & P_HC1[1] & P_HC1[0] &
           C_HC1[0]);
22
23      assign Sum_HanCarlson1 = P_HC1 ^ C_HC1[3:0];
24      assign Cout_HanCarlson1 = G_HC1[3]  (P_HC1[3] & G_HC1[2])
           (P_HC1[3] & P_HC1[2] & G_HC1[1])  (P_HC1[3] & P_HC1[2] &
           P_HC1[1] & G_HC1[0])  (P_HC1[3] & P_HC1[2] & P_HC1[1] &
           P_HC1[0] & C_HC1[0]);
25
26      // BEC for the Han-Carlson result (4-bit Binary to Excess-1
           Converter)
27      function automatic logic [3:0] BEC (input logic [3:0] in);
28          return in + 1;
29      endfunction
30
31      // 4-bit Weinberger Adder 1
32      logic [3:0] G_W1, P_W1, C_W1;
33      assign G_W1 = A[7:4] & B[7:4];   // Generate
34      assign P_W1 = A[7:4] ^ B[7:4];   // Propagate
35
36      assign C_W1[0] = Cout_HanCarlson1;
37      assign C_W1[1] = G_W1[0]  (P_W1[0] & C_W1[0]);
38      assign C_W1[2] = G_W1[1]  (P_W1[1] & G_W1[0])  (P_W1[1] & P_W1[0]
           & C_W1[0]);
39      assign C_W1[3] = G_W1[2]  (P_W1[2] & G_W1[1])  (P_W1[2] & P_W1[1]
           & G_W1[0])  (P_W1[2] & P_W1[1] & P_W1[0] & C_W1[0]);
40
41      assign Sum_Weinberger1 = P_W1 ^ C_W1[3:0];
42      assign Cout_Weinberger1 = G_W1[3]  (P_W1[3] & G_W1[2])  (P_W1[3] &
           P_W1[2] & G_W1[1])  (P_W1[3] & P_W1[2] & P_W1[1] & G_W1[0])
           (P_W1[3] & P_W1[2] & P_W1[1] & P_W1[0] & C_W1[0]);
43
44      // 4-bit Weinberger Adder 2
45      logic [3:0] G_W2, P_W2, C_W2;
46      assign G_W2 = A[11:8] & B[11:8]; // Generate
47      assign P_W2 = A[11:8] ^ B[11:8]; // Propagate
48
49      assign C_W2[0] = Cout_Weinberger1;
50      assign C_W2[1] = G_W2[0]  (P_W2[0] & C_W2[0]);
51      assign C_W2[2] = G_W2[1]  (P_W2[1] & G_W2[0])  (P_W2[1] & P_W2[0]
           & C_W2[0]);
52      assign C_W2[3] = G_W2[2]  (P_W2[2] & G_W2[1])  (P_W2[2] & P_W2[1]
           & G_W2[0])  (P_W2[2] & P_W2[1] & P_W2[0] & C_W2[0]);
53
54      assign Sum_Weinberger2 = P_W2 ^ C_W2[3:0];
55      assign Cout_Weinberger2 = G_W2[3]  (P_W2[3] & G_W2[2])  (P_W2[3] &
           P_W2[2] & G_W2[1])  (P_W2[3] & P_W2[2] & P_W2[1] & G_W2[0])
```

```
        (P_W2[3] & P_W2[2] & P_W2[1] & P_W2[0] & C_W2[0]);

    // 4-bit Han-Carlson Adder (Final)
    logic [3:0] G_HC2, P_HC2, C_HC2;
    assign G_HC2 = A[15:12] & B[15:12]; // Generate
    assign P_HC2 = A[15:12] ^ B[15:12]; // Propagate

    assign C_HC2[0] = Cout_Weinberger2;
    assign C_HC2[1] = G_HC2[0]  (P_HC2[0] & C_HC2[0]);
    assign C_HC2[2] = G_HC2[1]  (P_HC2[1] & G_HC2[0])  (P_HC2[1] &
        P_HC2[0] & C_HC2[0]);
    assign C_HC2[3] = G_HC2[2]  (P_HC2[2] & G_HC2[1])  (P_HC2[2] &
        P_HC2[1] & G_HC2[0])  (P_HC2[2] & P_HC2[1] & P_HC2[0] &
        C_HC2[0]);

    assign Sum_HanCarlson2 = P_HC2 ^ C_HC2[3:0];
    assign Cout_HanCarlson2 = G_HC2[3]  (P_HC2[3] & G_HC2[2])
        (P_HC2[3] & P_HC2[2] & G_HC2[1])  (P_HC2[3] & P_HC2[2] &
        P_HC2[1] & G_HC2[0])  (P_HC2[3] & P_HC2[2] & P_HC2[1] &
        P_HC2[0] & C_HC2[0]);

    // 16-bit Hybrid Adder Outputs
    assign Sum = {Sum_HanCarlson2, Sum_Weinberger2, Sum_Weinberger1,
        BEC(Sum_HanCarlson1)};
    assign Cout = Cout_HanCarlson2;

endmodule
```


5.4 Testbench

Listing 6: 16-Bit Hybrid Adder

```

module hybrid16tb();

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

    // Instantiate the 16-bit Hybrid Adder
    hybrid16 a(
        .A(A),
        .B(B),
        .Cin(Cin),
        .Sum(Sum),
        .Cout(Cout)
    );

    // Test stimulus
    initial begin
        // Monitor output signals
        $monitor("Time = %0t, A = %b, B = %b, Cin = %b -> Sum = %b,
            Cout = %b",
                 $time, A, B, Cin, Sum, Cout);

```

```
25          // Test cases
26          A = 16'b0000000000000001; B = 16'b0000000000000001; Cin =
                1'b0; #10; // Test Case 1
27          A = 16'b1111000011110000; B = 16'b0000111100001111; Cin =
                1'b0; #10; // Test Case 2
28          A = 16'b1111111111111111; B = 16'b0000000000000001; Cin =
                1'b0; #10; // Test Case 3
29          A = 16'b0101010101010101; B = 16'b1010101010101010; Cin =
                1'b0; #10; // Test Case 4
30          A = 16'b1111000011110000; B = 16'b1111000011110000; Cin =
                1'b0; #10; // Test Case 5
31          A = 16'b0000000000000000; B = 16'b0000000000000000; Cin =
                1'b1; #10; // Test Case 6
32          A = 16'b1111111111111111; B = 16'b1111111111111111; Cin =
                1'b1; #10; // Test Case 7
33
34          // Finish simulation
35          $stop;
36      end
37
38  endmodule
```

5.5 Simulation

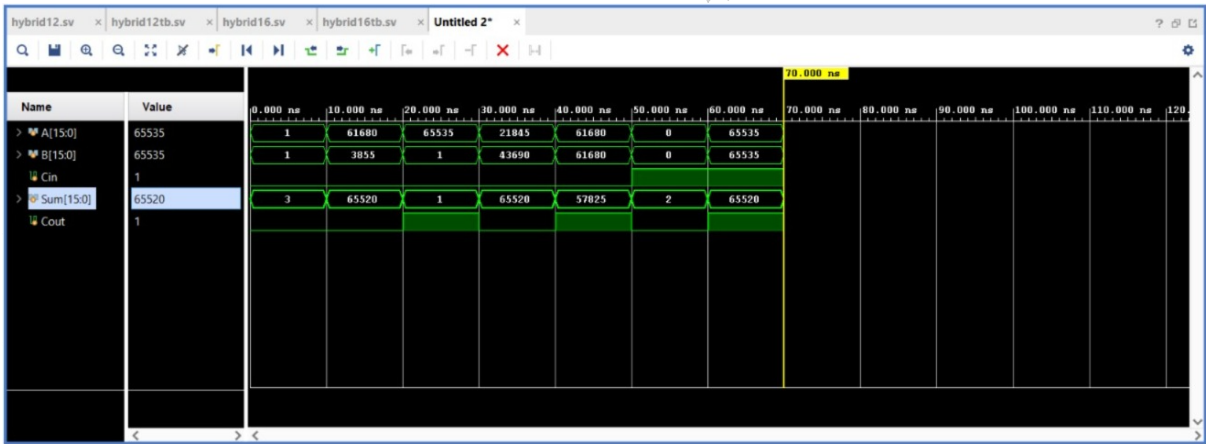

Figure 7: Simulation of 16-Bit Hybrid Adder

5.6 Schematic

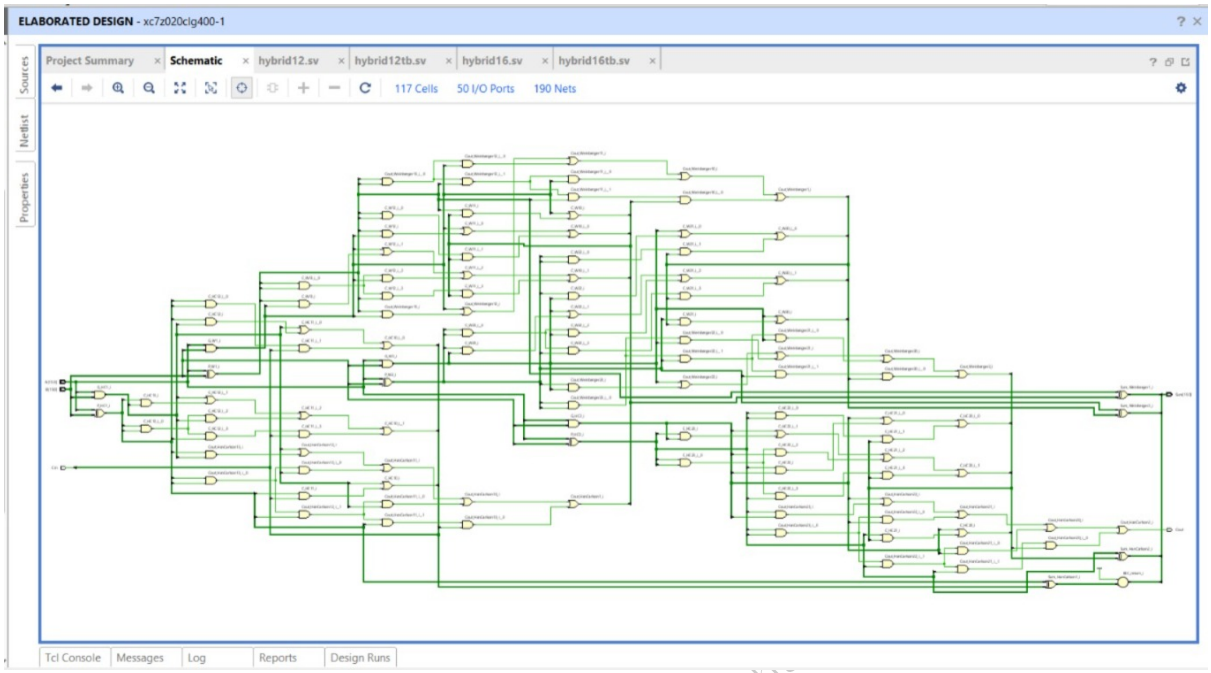

Figure 8: Schematic of 16-Bit Hybrid Adder

5.7 Synthesis Design

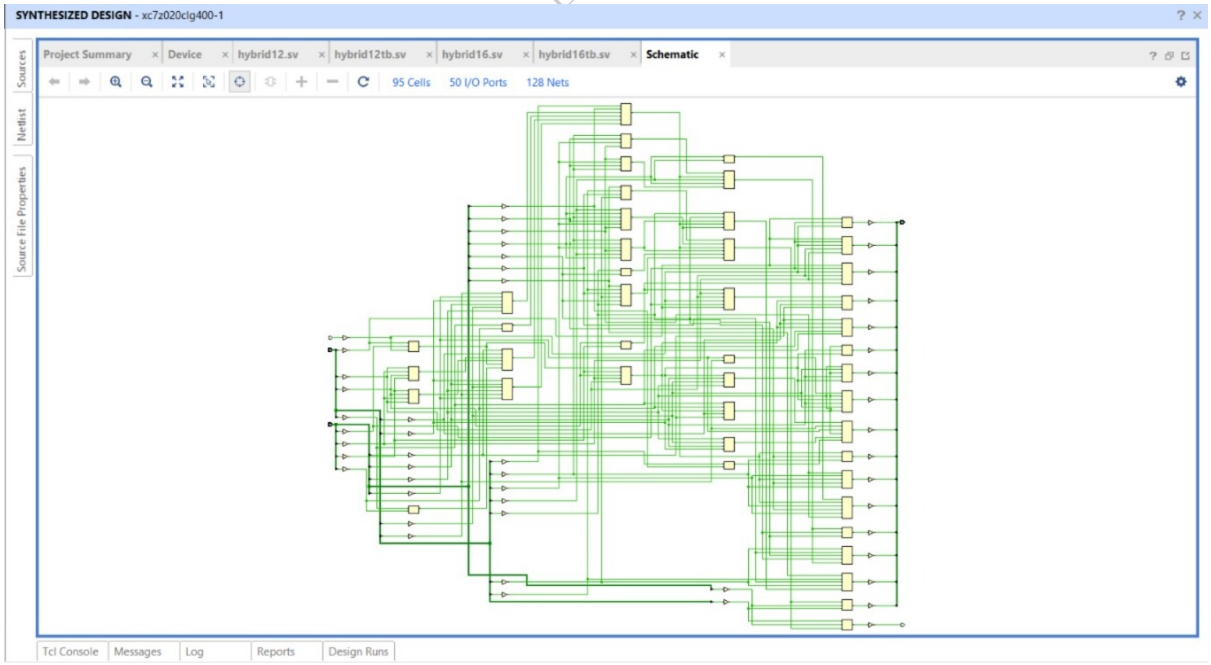

Figure 9: Synthesis Design of 16-Bit Hybrid Adder

6 Comparison between 8-bit,12-bit, and 16-bit Hybrid Adders

Aspect	8-bit Hybrid Adder (Weinberger & Han-Carlson)	12-bit Hybrid Adder (Ling, Weinberger, & Han-Carlson)	16-bit Hybrid Adder (Two Weinberger & Two Han-Carlson)
Bit-width	8-bit	12-bit	16-bit
Adders Used	1 Weinberger, 1 Han-Carlson	1 Ling, 1 Weinberger, 1 Han-Carlson	2 Weinberger, 2 Han-Carlson
Stages	2 stages (Weinberger → Han-Carlson)	3 stages (Ling → Weinberger → Han-Carlson)	3 stages (Weinberger → Weinberger → Han-Carlson)
Carry Propagation Delay	Moderate (due to smaller size and fewer stages)	Low (Ling adder reduces critical path delay)	Moderate (due to larger size, but parallel computation)
Speed	Faster than traditional adders	Very fast (Ling adder further speeds up carry computation)	Fast (larger size balanced by parallel computation)
Complexity (Hardware Area)	Moderate	High (due to the combination of Ling, Weinberger, and Han-Carlson adders)	Higher complexity (due to larger bit-width and more adders)
Use Case	Suitable for small to medium arithmetic tasks	Optimized for high-speed arithmetic in moderate bit-width tasks	Suitable for larger, high-speed arithmetic applications
Advantages	Balanced between performance and area	Reduced delay due to Ling adder, highly efficient for 12-bit	Parallelism reduces overall delay, good for larger inputs
Challenges	Balancing delay between the stages	More complex to implement due to extra Ling adder	More hardware area due to double adders
Carry Calculation Method	Carry-save followed by carry-lookahead	Pre-calculated carry (Ling) followed by carry-save and carry-lookahead	Carry-save followed by parallel carry-lookahead

Table 1: Comparison of Different Hybrid Adders

7 Conclusion

In this series of RTL projects, we have successfully implemented 26 different types of adders, exploring both basic and advanced arithmetic circuits. Each design presents unique characteristics in terms of performance, area, and complexity, providing valuable insights into the trade-offs involved in digital design. From simple Half and Full Adders to more complex structures like Kogge-Stone, Han-Carlson, and Modular Adders, the breadth of adders we explored reflects the diversity of techniques used in modern computing systems to optimize speed and power consumption.

The completion of this phase of the project allowed us to not only gain a deep understanding of the theoretical aspects of each design but also engage with practical implementation challenges, such as handling carry propagation, optimizing for speed (as seen in CLA and KSA), and reducing gate count (Brent-Kung and Ladner-Fischer Adders).

This milestone contributes to the broader goal of completing 108 RTL designs, as we now have a comprehensive library of adder architectures, ranging from traditional designs to cutting-edge hybrid and pipelined approaches. These implementations will serve as foundational building blocks for more complex computational circuits as we move forward in our project.

8 FAQs

1. What is a hybrid adder?

Answer: A hybrid adder combines different types of adders to optimize performance, area, and power efficiency. By using various adder architectures, such as carry-lookahead adders (CLA), carry-save adders (CSA), and parallel-prefix adders (PPA), hybrid adders achieve a balance between speed and complexity. They are used in scenarios where a single adder type cannot meet the requirements of large bit-width arithmetic operations.

2. How does an 8-bit hybrid adder made of a Wienerberger and Han-Carlson adder work?

Answer: In an 8-bit hybrid adder:

The Wienerberger adder handles the initial bit additions with carry-save techniques, which minimizes carry propagation.

The result is then processed by a Han-Carlson adder, which efficiently computes the final carry bits using a parallel prefix approach. This combination reduces carry propagation delay while maintaining a balanced performance for small to medium-sized additions.

3. Why is the Ling adder used in the 12-bit hybrid adder?

Answer: The Ling adder is known for its efficient carry computation by simplifying the carry propagation logic. In a 12-bit hybrid adder, the Ling adder is used as the first stage to significantly reduce the critical path delay, making the overall adder faster. It is especially useful in moderate bit-width adders where speed is a priority. This is followed by a combination of a Wienerberger and Han-Carlson adder to finalize the sum.

4. What is the advantage of using two Weinberger and two Han-Carlson adders in a 16-bit hybrid adder?

Answer: A 16-bit hybrid adder uses two Weinberger adders in parallel to manage different parts of the 16-bit inputs efficiently. The output from these is then processed by two Han-Carlson adders to compute the final sum. This parallel approach improves performance by reducing the time needed for carry propagation, allowing for fast addition of larger bit-width numbers. The design helps balance speed and hardware complexity for high-speed applications.

5. What are the key differences between the 8-bit, 12-bit, and 16-bit hybrid adders?

Answer:

8-bit hybrid adder: Uses one Weinberger and one Han-Carlson adder. It is optimized for small bit-width arithmetic and offers a good balance between performance and area. 12-bit hybrid adder: Incorporates a Ling adder along with a Wienerberger and Han-Carlson adder. The Ling adder helps reduce the critical path delay, making this adder highly efficient for moderate bit-width operations. 16-bit hybrid adder: Uses two Weinberger and two Han-Carlson adders in parallel, allowing for faster processing of larger bit-width operations by minimizing carry propagation delay.

6. What is the role of carry propagation delay in hybrid adders?

Answer: Carry propagation delay refers to the time it takes for the carry signal to propagate through the adder stages. Minimizing this delay is crucial for fast arithmetic operations, especially in large bit-width adders. Hybrid adders use architectures like the Han-Carlson and Ling adders to precompute or parallelize the carry operations, significantly reducing the delay compared to traditional adders like ripple carry adders.

7. Why are hybrid adders used in high-performance arithmetic circuits?

Answer: Hybrid adders are used in high-performance circuits because they offer a balanced trade-off between speed, power, and area. By combining different types of adders, the hybrid design optimizes critical parameters such as carry propagation delay and hardware complexity. This makes them ideal for applications like digital signal processing (DSP), cryptography, and processors where fast and efficient arithmetic operations are essential.

8. What challenges are involved in designing hybrid adders?

Answer: The main challenges in designing hybrid adders include:

Balancing complexity and performance: As more adders are combined, the overall design becomes more complex, which could increase the hardware area and power consumption. Managing carry propagation: Properly managing carry propagation across different stages is critical to ensure that the performance gains from the hybrid design are realized. Optimizing for specific applications: Different hybrid adder designs are suited for different use cases. For instance, a 12-bit adder optimized for speed may not perform as well in low-power applications.

9. What factors influence the choice of adder types in a hybrid adder?

Answer: The choice of adder types in a hybrid adder depends on several factors, including:

Bit-width: Larger bit-width adders require architectures that minimize carry propagation delay, like parallel-prefix adders (e.g., Han-Carlson or Kogge-Stone).

Performance requirements: Adders like the Ling adder are used in scenarios where speed is critical. **Power and area constraints:** For low-power applications, adders with simpler structures, such as ripple-carry or carry-save, may be favored.

Application-specific needs: DSP and cryptographic algorithms may require specific adder types for optimal performance.

10. How does the combination of carry-save and carry-lookahead methods improve hybrid adders?

Answer: In hybrid adders, the carry-save method (used by adders like the Wienerberger) helps reduce the carry propagation delay by grouping and summing multiple inputs simultaneously without waiting for carry propagation. The carry-lookahead method (used by adders like the Han-Carlson or Ling) computes the carry in parallel, minimizing the carry delay further. Combining these two methods in a hybrid adder allows for fast and efficient addition, especially in larger bit-width designs.