Digital Adder Family: From Basic Gates to 64-bit Carry-Lookahead

Julia E. Benítez

11/2025

Contents

1 Introduction

This work explores the implementation of the most fundamental unit in combinational logic design: the adder, tracing its journey from basic Boolean equations through various architectural implementations at both gate and transistor levels.

Time efficiency stands as a critical factor in digital design, forming the central focus of this investigation, particularly how Boolean logic translates into real circuit behavior through CMOS technology. What began as a simple Verilog learning exercise evolved through multiple stages into a exploration of CMOS physics analyzed in LTspice, revealing the significant gap between digital abstraction and physical reality.

The path from Verilog simulation to CMOS implementation uncovered fundamental questions: How do idealized gate delays compare to actual propagation times? What optimization benefits survive the transition from Boolean algebra to physical transistors? And how do parasitic effects impact circuit behavior in ways invisible to digital simulation?

This project addresses these questions through development of adder architectures, experimental characterization of CMOS gates and comparative analysis of timing performance across different levels of abstraction.

2 Theoretical Background

2.1 Binary Addition Fundamentals

A bit, the fundamental unit of binary data, is a boolean value represented as either 0 or 1. The addition of two single bits follows four basic rules:

$$\begin{aligned} 0 + 0 &= 0 \\ 0 + 1 &= 1 \\ 1 + 0 &= 1 \\ 1 + 1 &= 0 \end{aligned}$$

A particular case occurs when both bits are 1 , it produces a sum of 0 and generates a **carry-out** of 1. This carry must be propagated to the next significant bit position.

Figure 1 demonstrates this process by adding the two-bit numbers $A = 01$ and $B = 11$ using the traditional columnar method:

$$\begin{array}{rr} \text{Carry} & 1\,1 \\ A & 0\,1 \\ + \quad B & 1\,1 \\ \hline \text{Sum} & 10\,0 \end{array}$$

Figure 1: Binary addition example showing carry propagation

The addition proceeds from right to left (Least Significant Bit to Most Significant Bit) for the bit 0 (LSB): $1 + 1 = 0$ with carry-out of 1, then for bit 1: $0 + 1 + 1$ (carry-in) $= 0$ with carry-out of 1, and for bit 2: $0 + 0 + 1$ (carry-in) $= 1$

Note that adding two 2-bit numbers produces a 3-bit result due to the final carry-out. This demonstrates that the result's width can exceed that of the operands. (This concept will later be discussed as overflow).

As operand size increases, the sequential propagation of carries from LSB to MSB becomes the dominant factor in adder performance. This **carry-propagation delay** is a critical timing constraint in arithmetic logic unit design and motivates various adder optimization techniques.

2.2 Full Adder Design

2.2.1 Half-Adder

The Half-Adder is a combinational logic circuit that implements the addition of two single bits. Based on the fundamental rules of binary addition, its behavior is defined by the following truth table (Figure 2):

A	**B**	**Sum (S)**	**Carry (C)**
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 2: Half-Adder truth table

The Boolean equations for the Sum (S) and Carry (C) outputs are derived using a Karnaugh Map (K-Map). The notation used for Boolean operations is as follows: the + symbol represents the OR operation, · (or its omission) represents the AND operation, and ⊕ represents the XOR operation.

$$
\begin{aligned}
S &= A'B + AB' = A \oplus B \\
C &= AB
\end{aligned}
$$

These equations can be implemented using logic gates. Figure 3 shows the direct implementation using fundamental gates (AND, OR, NOT), while Figure 4 shows the more common and efficient implementation that utilizes an XOR gate for the sum output.

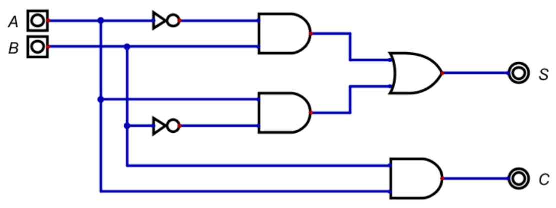


Figure 3: Half-Adder implementation with basic gates

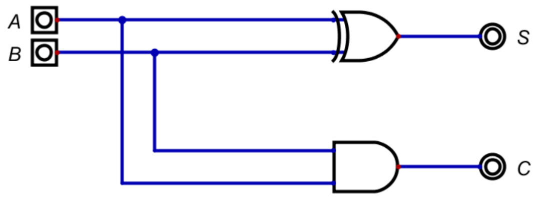


Figure 4: Half-Adder implementation with an XOR gate

2.2.2 Full Adder

The Half-Adder has a significant limitation: it cannot account for an incoming carry from a previous addition. To create an adder capable of chaining together for multi-bit operations, a third input called the **carry-in** (C_{in}) is introduced. This circuit is known as a **Full Adder**(FA).

The Full Adder adds three single-bit inputs—A, B, and C_{in}—and produces a Sum (S) and a **carry-out** (C_{out}). Its complete functionality is described by the following truth table:

The Boolean equations for the Sum and Carry-out are derived from this table. For the Sum output:

$$
\begin{aligned}
S &= A'B'C_{in} + A'BC'_{in} + AB'C'_{in} + ABC_{in} \\
S &= C_{in} \oplus (A \oplus B)
\end{aligned}
$$

A	**B**	C_{in}	**Sum (S)**	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 5: Full-Adder truth table

The Carry-out equation can also be minimized.

$$C_{out} = BC_{in} + AC_{in} + AB$$

While it can be expressed in a minimal sum-of-products form, it is often more efficient to implement it using the same XOR term shared with the Sum logic, reducing overall circuit complexity:

$$\begin{aligned} C_{out} &= AB + A'BC_{in} + AB'C_{in} = AB + C_{in}(A'B + AB') \\ C_{out} &= AB + C_{in}(A \oplus B) \end{aligned}$$

The key insight in this implementation is the shared term $(A \oplus B)$ between both the Sum and Carry-out equations. It reduces the number of logic gates required. Figure 6 shows the gate-level implementation of the Full Adder based on these optimized equations.

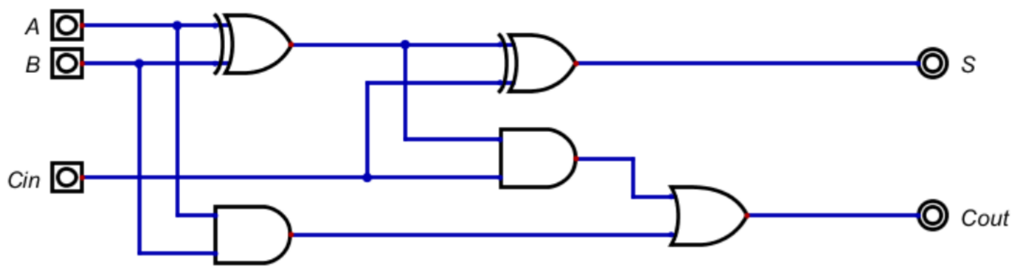


Figure 6: Full-Adder implementation using shared XOR logic

2.3 Ripple-Carry Adder

The Full-Adder can be extended to perform multi-bit addition through an iterative approach. A straightforward design that mimics traditional manual addition chains the carry-out of one Full-Adder to the carry-in of the next. In this configuration, the carry signal propagates sequentially through the entire chain. This design is known as a Ripple-Carry Adder (RCA), as illustrated in Figure 7.

The operands are distributed across the single-bit inputs of each Full-Adder module, beginning with the Least Significant Bit (LSB) at positions A_0, B_0. The individual sum outputs are concatenated to form the complete output word S, with the final carry-out from the last adder included to create an N+1-bit result (assuming unsigned operands).

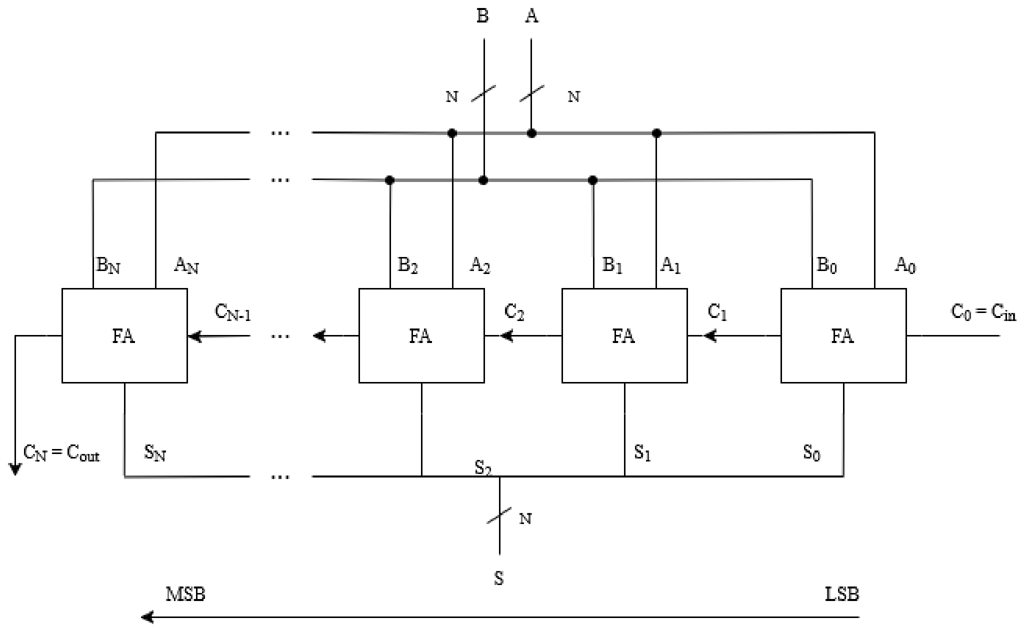


Figure 7: N-bit Ripple Carry Adder architecture

Propagation delay is a critical consideration in this design. Since each sum output depends on its carry-in, which must propagate through the entire preceding chain, the worst-case delay occurs when a carry ripples through all stages. Examining the Full-Adder implementation in Figure 6, the critical path for carry propagation passes through approximately three gate delays per stage. This results in an overall time complexity of $O(N)$, meaning the addition time increases linearly with operand size.

While the RCA provides a simple solution for small operand widths (where propagation delay remains acceptable), its linear timing characteristic becomes prohibitive for larger bit widths. This limitation motivates the development of alternative adder architectures that employ parallel computation to overcome the carry propagation bottleneck.

2.4 Carry-Lookahead Adder (CLA)

2.4.1 Single-Level CLA

To overcome the linear propagation delay of the Ripple-Carry Adder, parallel computation introduces a new architecture: the Carry-Lookahead Adder (CLA). The CLA employs a separate module that computes all carry signals simultaneously using two auxiliary signals derived from each Full-Adder stage.

Building upon the Full-Adder carry equation, we express C_{out} in terms of two new variables: P (Propagate) and G (Generate).

$$C_{out} = AB + C_{in}(A \oplus B)$$
$$C_{out} = G + C_{in}P$$

where $G = AB$ represents a generated carry, and $P = A \oplus B$ represents a propagated carry. For multiple stages, the general form for the carry at position $i + 1$ is:

$$C_{i+1} = G_i + P_iC_i \tag{1}$$

By recursively applying Equation 1, we can express all carries in terms of the independent Generate (G_i) and Propagate (P_i) signals, along with the initial carry-in C_0:

$$\begin{aligned}
C_1 &= G_0 + P_0C_0 \\
C_2 &= G_1 + P_1C_1 = G_1 + P_1G_0 + P_1P_0C_0 \\
C_3 &= G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \\
C_4 &= G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0
\end{aligned}$$

The complexity of the carry logic expands with each subsequent stage, both in the number of terms and the fan-in requirements of the gates. This growth imposes a practical limit on the size of a single-level carry generator. Figure 8 shows a 4-bit CLA implementation where four Full-Adders are fed with pre-computed carry signals, with C_4 serving as the final carry-out.

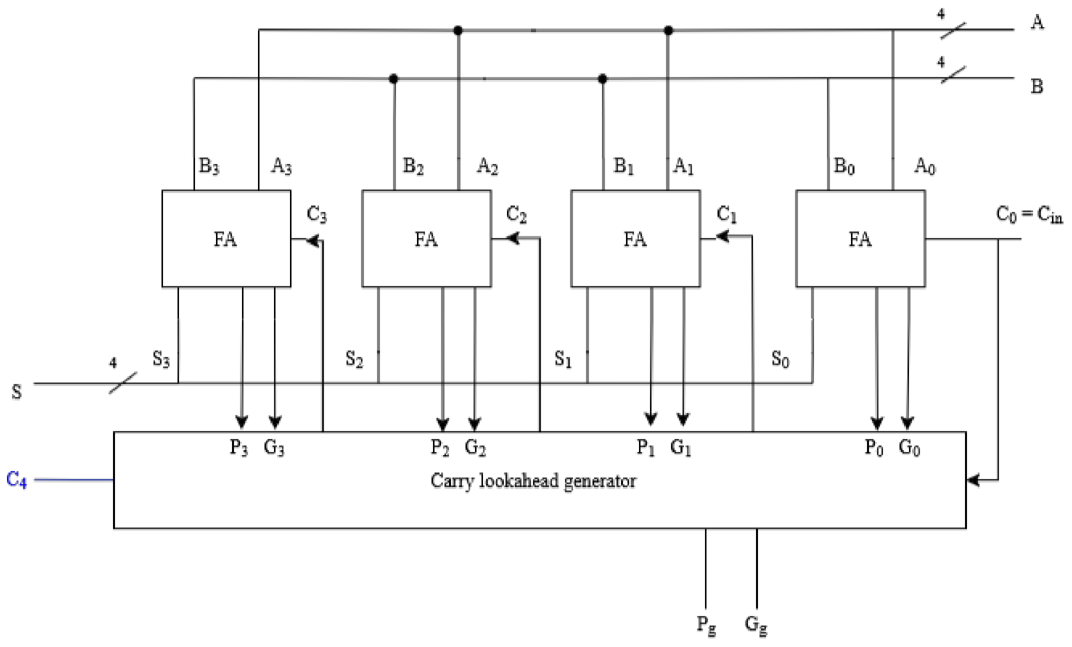


Figure 8: 4-bit Carry-Lookahead Adder architecture

While the critical path per stage remains approximately 3 gate delays, the key advantage is that all carries are computed in parallel. This means the propagation delay for S_1 through S_3 is nearly identical, resulting in constant time complexity $O(1)$ for the carry computation within the block. The CLA achieves significant speed improvement at the cost of increased circuit complexity—as more bits are added, the carry generation logic requires more gates with higher fan-in.

Finally total Sum is calculated using:

$$S_i = P_i \oplus C_i$$

To scale this approach efficiently, a hierarchical architecture using group propagate and generate signals becomes necessary.

2.4.2 64-bit Hierarchical Architecture

Constructing a 64-bit Carry-Lookahead Adder requires a hierarchical approach to manage circuit complexity. While various configurations are possible—such as chaining four 16-bit adders with mixed lookahead and ripple-carry propagation, this project employs a consistent three-level hierarchy using 4-bit CLA modules and 4-signal carry generators. This design enables module reuse while minimizing propagation delay.

The hierarchical approach introduces group propagate (P_g) and generate (G_g) signals for 4-bit blocks:

$$\begin{aligned} P_g &= P_3 \cdot P_2 \cdot P_1 \cdot P_0 \\ G_g &= G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 \end{aligned}$$

This necessitates a modification to the basic 4-bit CLA design (Figure 8): the explicit C_4 output is omitted since it can be derived as $C_4 = G_g + P_g \cdot C_0$.

The same carry generation principles scale to higher levels. Figure 10 shows how four 4-bit CLAs combine with a second-level carry generator to form a 16-bit CLA, using group signals as inputs. Similarly, Figure 9 demonstrates the complete 64-bit architecture, where four 16-bit CLAs feed into a third-level carry generator.

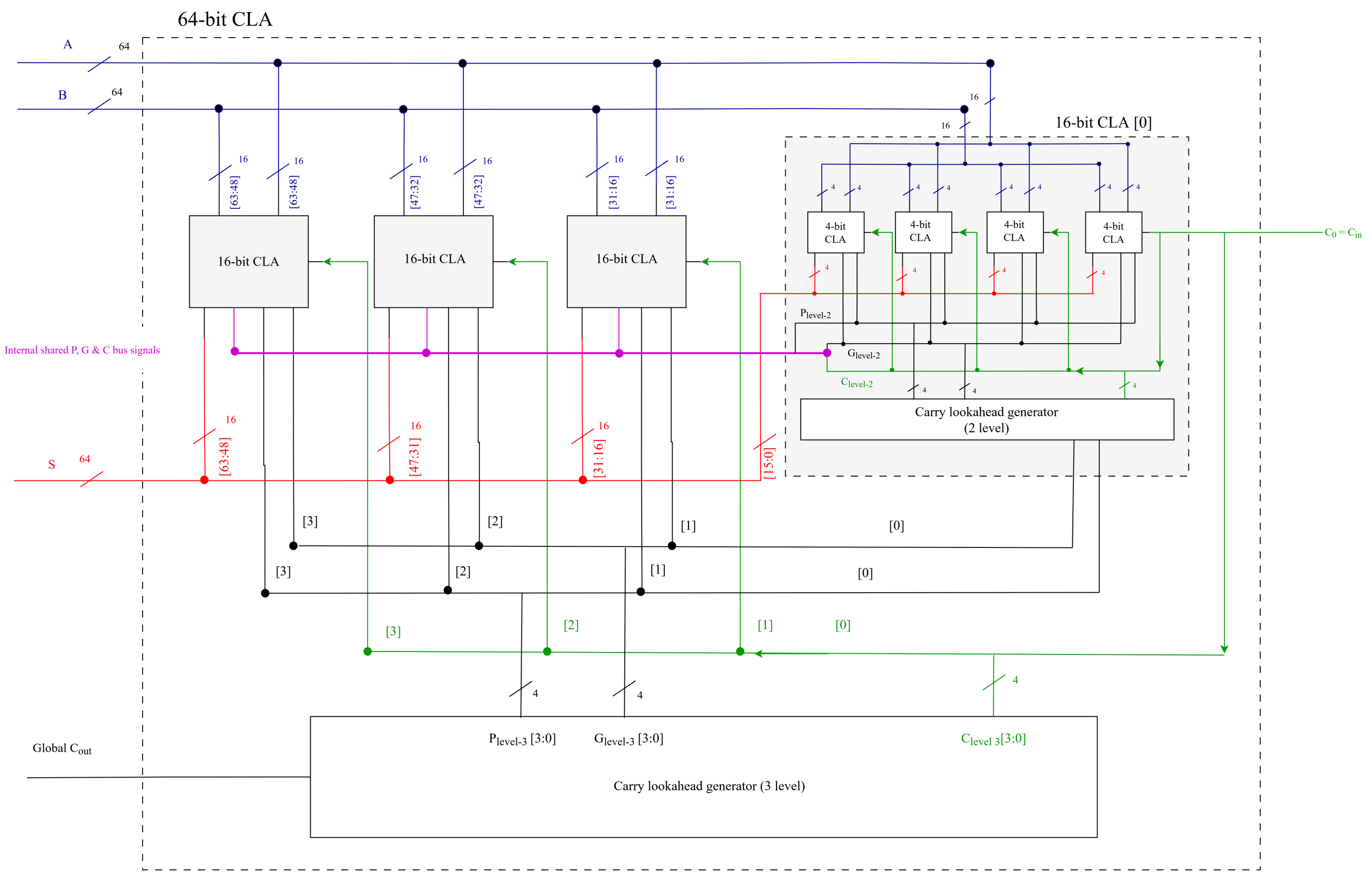


Figure 9: Complete 64-bit hierarchical CLA architecture

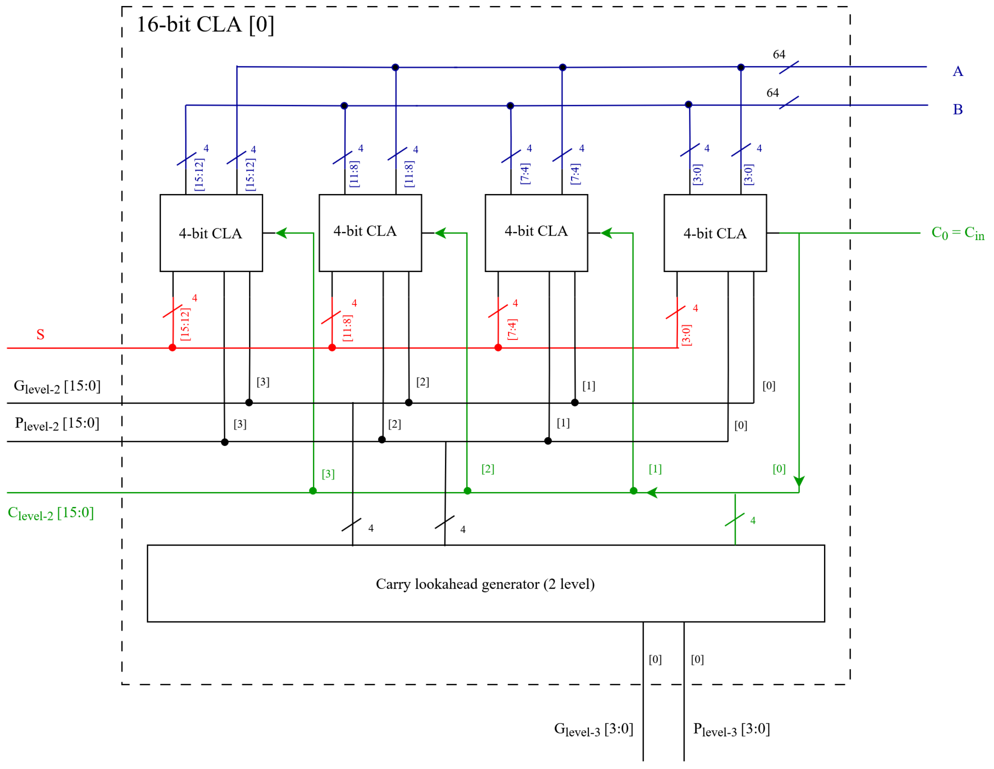


Figure 10: Internal structure of a 16-bit CLA block (detailed view)

The complete implementation comprises: 64 Full-Adder modules, 16 first-level carry generators (4-bit), 4 second-level carry generators (16-bit), 1 third-level carry generator (64-bit).

Signal widths follow the hierarchy: G_{level2}, P_{level2}, and C_{level2} are 16-bit buses shared across 16-bit modules, while level-3 signals are 4-bit wide. This will be further justified during the Verilog implementation.

2.4.3 Time Analysis

Assuming a delay of one time unit per logic gate, the carry propagation within each 4-bit CLA (Level-1) requires approximately three time units: one unit for (P_i) and (G_i), and two additional units for computing (G_g). The corresponding sum output requires one extra gate level, yielding a total of four time units for local sum computation.

For higher hierarchy levels, each 16-bit and 64-bit carry generator (Levels 2–3) adds about four time units due to additional group-propagate and group-generate logic. Then, the global carry output emerges after approximately ten time units, while the complete sum becomes valid after twelve time units.

This hierarchical delay has logarithmic scaling with operand width, following roughly $4 \cdot \log_4 N$ time units. For a 64-bit implementation, this yields 12 units compared to the 192 time units of a 64-bit Ripple-Carry Adder (assuming three gate delays per full adder $3 \cdot 64$). CLA is $16\times$ faster than a RCA.

2.5 CMOS Implementation

2.5.1 CMOS Digital Logic Circuits Basics

CMOS technology forms the foundation of modern digital design, using complementary pairs of PMOS and NMOS transistors to implement logic gates. In digital applications, MOS transistors operate primarily in the cutoff region (boolean 0) and triode region (boolean 1 at V_{DD}), enabling efficient switching behavior.

The fundamental structure of CMOS gates consists of two complementary networks: the Pull-Up Network (PUN) comprising PMOS transistors, and the Pull-Down Network (PDN) comprising NMOS transistors. The inverter, as the basic building block, demonstrates this complementary principle: when input is low, the PMOS conducts, pulling output to V_{DD}; when input is high, the NMOS conducts, pulling output to ground.

For complex gates, the PUN produces strong logic 1 states when the Boolean function $Y(In_1, In_2, ..., In_n)$ is true, while the PDN produces strong logic 0 states when the function is false. Basic operations map directly to transistor arrangements: OR functions use parallel paths, while AND functions use series connections. Most CMOS gates exhibit dual topology where parallel PMOS paths correspond to series NMOS paths and vice versa.

2.5.2 CMOS Gate design

OR gate CMOS logic circuit design can be derived from the equation:

$$Y = A + B$$

To obtain the PUN expression, as previously stated Y must be true, but for this to happen A and B must be off. Therefore, instead of working with the OR gate directly, analysis is more easily done with the NOR gate:

$$Y = (A + B)'$$

Using De Morgan's laws:

$$Y = A'B'$$

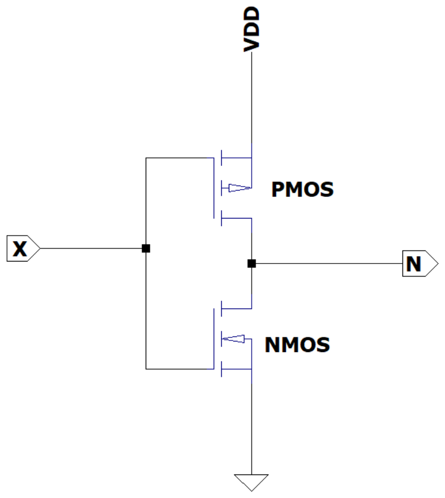


Figure 11: Basic inverter using PMOS and NMOS transistors

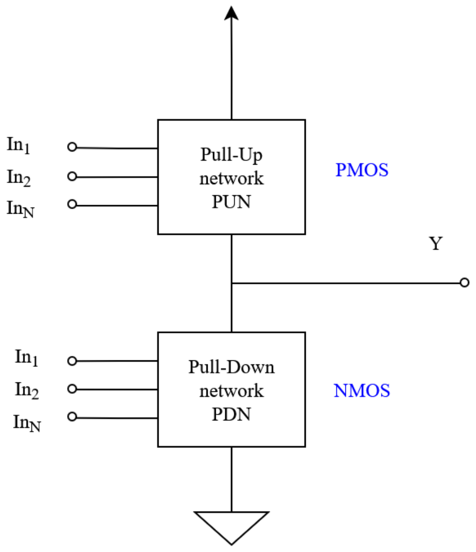


Figure 12: General structure of a CMOS logic-gate circuit

and equivalently for PDN:

$$Y' = ((A + B)')' = A + B$$

So we obtain a series arrangement for PUN and a parallel arrangement for PDN. Note that this network will produce NOR output; to obtain OR, it's necessary to use an additional inverter at the output.

Similarly for AND gate, NAND gate is built and then output is inverted to obtain AND. For PUN and PDN NAND gate:

$$Z = (AB)' = A' + B'$$

$$Z' = ((AB)')' = AB$$

Therefore the PUN will be a parallel arrangement while PDN will be a series arrangement. This idea can be expanded to create gates with more inputs; in a NAND gate, more series transistors can be added in the PDN, one per additional input (and equally parallel transistors in PUN).

Examining the XOR gate equation, note that outputs cannot be expressed as direct dual networks but some must be inverted for both PUN and PDN:

$$Z_2 = A'B + AB'$$

$$Z_2' = (A'B + AB')' = (A'B)'(AB')' = (A + B')(A' + B) = AB + A'B'$$

XOR will have 8 transistors considering that B' and A' are already given; if not, the circuit will have 4 additional transistors for A and B signal inverters. Note that XOR gate is not a dual network.

2.5.3 Full Adder CMOS Implementation

According to the full adder logic diagram and current CMOS gate design methodology, it would take 42 MOS transistors to build a gate-level implementation. However, new pairs of equations for Sum and Carry out can be derived for optimization.

According to the Carry out minimized equation:

$$C_{out} = BC_{in} + AC_{in} + AB = AB + C_{in}(B + A)$$

Similarly to AND/OR gates, operating with the inverse function is convenient. For the PUN:

$$NC_{out} = C'_{out} = (AB + C_{in}(B + A))' = (AB)'(C_{in}(B + A))' = (A' + B')(C'_{in} + (A + B)')$$

$$NC_{out} = (A' + B')(C'_{in} + A'B')$$

and for the PDN, NC'_{out} is the original function:

$$NC'_{out} = AB + C_{in}(B + A)$$

This reduces the total necessary MOS transistors from 36 (using gates) to only 12.

For the Sum equation, the terms can be factorized in a clever way:

$$S = A'B'C_{in} + A'BC'_{in} + AB'C'_{in} + ABC_{in}$$

$$S = ABC_{in} + (A+B+C_{in})(A'C'_{in} + B'C'_{in} + A'B') = ABC_{in} + (A+B+C_{in})(A'+B')(C'_{in} + A'B')$$

$$S = ABC_{in} + (A + B + C_{in})NC_{out}$$

Using this expression, PUN and PDN equations are built as was done for C_{out}:

$$NS = S' = (ABC_{in} + (A + B + C_{in})NC_{out})' = (A'B'C'_{in} + NC'_{out})(A' + B' + C'_{in})$$

$$NS' = ABC_{in} + (A + B + C_{in})NC_{out}$$

Notice that NC_{out} is reused. Sum uses 16 MOS transistors, making the total gate count 28. This design is known as Adder-28T.

All circuit models can be found in Section 3.3.

2.5.4 Propagation Delay Analysis

Feeding an inverter input with an ideal pulse results in a rounded output pulse, indicating finite transition times between high and low states. These delay times result from various capacitances in the circuit (MOSFET, wiring, and input capacitances). Two asymmetric delay times are defined for analysis as shown in Figure 13, where the "switching point" occurs at $V_{DD}/2$.

The inverter propagation delay is defined as:

$$t_p = \frac{1}{2}(t_{PHL} + t_{PLH})$$

Using the following equations, both delays can be found as functions of MOSFET channel width and length:

$$t_{PHL} = 0.69 R_N C \quad \text{with} \quad R_N = \frac{12.5}{(W/L)_n} \text{ k}\Omega$$

$$t_{PLH} = 0.69 R_P C \quad \text{with} \quad R_P = \frac{30}{(W/L)_p} \text{ k}\Omega$$

where C is the load capacitance.

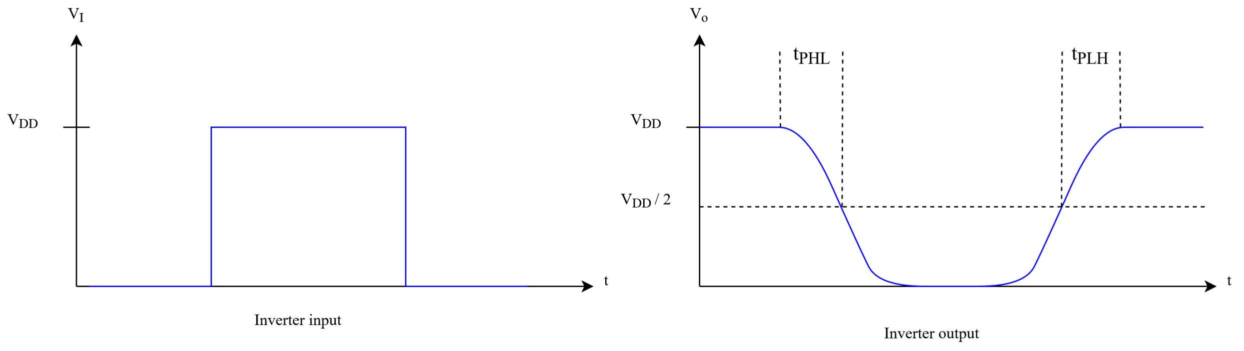


Figure 13: Input and output of inverter, delays are indicated

3 Design and Implementation

3.1 RTL Design in Verilog

3.1.1 Full Adder Implementation

The Full Adder module implements the single-bit addition logic derived in Section 2.2.2, with three input bits (x, y, z) and two outputs (sum s, carry c). The design reuses intermediate XOR operation to minimize gate count.

```verilog
/**
* Single-bit full adder module
*
* Outputs are produced according to the following boolean equations:
* s = x ^ y ^ z
* c = x & y | z & (x ^ y)
*/

module full_adder (
input x, y, z,
output s, c
);

        wire x_xor_y;
        assign x_xor_y = x ^ y;  // Intermediate signal: XOR of x and y

        assign s = z ^ x_xor_y;
        assign c = (x & y) | (z & x_xor_y);

endmodule
```

Listing 1: Single-bit Full Adder Implementation

This implementation directly corresponds to the equations presented in Section 2.2.2. Verilog assignments map directly to the logic gate arrangement shown in Figure 6, with approximately 3-gate propagation delay for critical paths.

Module serves as the fundamental building block for all subsequent adder implementations in the hierarchy.

3.1.2 Ripple-Carry Adder Implementation

A parameterized N-bit Ripple-Carry Adder is implemented by chaining N full adder modules, following the architecture shown in Figure 7. The design uses Full Adder module from Section 3.1.1.

```
/**
* N-bit Ripple Carry Adder (RCA)
*
* @param N     Bit width of the adder (default: 8)
*/

`include "full_adder.v"  // Include the full adder module

module ripple_carry_adder
#(parameter N = 8)    // Configurable bit width
(
input   [N-1:0] N1,    // First N-bit operand
input   [N-1:0] N2,    // Second N-bit operand
input           cin,   // Carry-in input
output  [N-1:0] St,    // N-bit sum output
output          cout   // Carry-out output
);

// Internal carry chain: carry[i] is the carry-out from bit position
    i
wire [N:0] carry;  // N+1 bits to include cin and cout

assign carry[0] = cin;  // First carry-in

// Generate chain of N full adders
genvar i;
generate
for (i = 0; i < N; i = i + 1) begin : FA_CHAIN
full_adder fa (
.x(N1[i]),
.y(N2[i]),
.z(carry[i]),   // Carry from previous stage
.s(St[i]),
.c(carry[i+1])  // Carry to next stage
);
end
endgenerate

assign cout = carry[N];  // Final carry-out

endmodule
```

Listing 2: N-bit Ripple Carry Adder Implementation

The carry chaining is accomplished using a `generate` block and an internal N+1-bit `carry` bus. This implementation exhibits a worst-case propagation delay of approximately $3N$ gate delays, resulting in $O(N)$ time complexity. While area-efficient and simple to implement, this linear scaling makes the RCA suitable only for small values of N or applications where speed is not critical.

3.1.3 4-bit CLA Block Design

This module implements the 4-bit Carry Lookahead Adder following the theoretical design established in Section 2.4.1. Unlike the Ripple-Carry approach, it computes all carry signals in parallel using propagate and generate logic, without explicitly instantiating Full Adder modules.

```
/**
* 4-bit Carry Lookahead Adder (CLA)
*
```

```
* Implements parallel carry computation using group P/G signals
* for hierarchical expansion in larger CLA architectures
*/

module carry_lookahead_4bit (
input  [3:0] A,        // 4-bit operand A
input  [3:0] B,        // 4-bit operand B
input        cin,      // Carry input
output [3:0] S,        // 4-bit sum output
output       Pg,       // Group Propagate (for hierarchical
   expansion)
output       Gg        // Group Generate (for hierarchical expansion
   )
);

wire [3:0] P, G;       // Individual bit propagate/generate signals
wire [3:0] C;          // Internal carry chain

// Bit-level propagate and generate
assign P = A ^ B;      // Propagate: A XOR B
assign G = A & B;      // Generate: A AND B

assign C[0] = cin;     // External carry-in

// Parallel carry computation
genvar i;
generate
for (i = 1; i < 4; i = i + 1) begin : carry_lookahead_chain
// Carry equation: C[i] = G[i-1] | (P[i-1] & C[i-1])
assign C[i] = G[i-1] | (P[i-1] & C[i-1]);
end
endgenerate

// Sum computation
assign S = P ^ C;

// Group signals for hierarchical expansion
assign Pg = &P;  // P[3] & P[2] & P[1] & P[0] - all bits propagate
assign Gg = G[3] | (P[3] & G[2]) | (P[3] & P[2] & G[1]) |
(P[3] & P[2] & P[1] & G[0]);  // Group generate

endmodule
```

Listing 3: 4-bit Carry Lookahead Adder Implementation

The design employs three internal buses: individual propagate (P) and generate (G) signals, and the computed carry chain (C). An architectural decision replaces the explicit carry-out with group propagate (Pg) and generate (Gg) signals, enabling efficient hierarchical composition in larger CLA structures. This modification aligns with the hierarchical equations defined in Section 2.4.2 and facilitates the 64-bit CLA implementation.

3.1.4 64-bit CLA Integration

This module performs 64-bit addition by instantiating sixteen 4-bit CLA modules in a hierarchical structure. The first level is constituted by 16-bit blocks, each containing four 4-bit CLA modules with a carry generator that processes group propagate (Pg) and generate (Gg) signals.The second level combines four 16-bit blocks using the same auxiliary module to form the complete 64-bit CLA, following diagram presented in figure 4.

/* *

```
* 64-bit Carry Lookahead Adder (CLA)
*
*NOTE: This adder treats operands as UNSIGNED integers.
*/

`include "Adder4bitsCLA.v"

module add64CLA #(parameter N = 64)(
input  [N-1:0] A, B,
input  carry_in,
output [N-1:0] S,
output carry_out
);

localparam BLOCKS_16B = N / 16;
localparam BLOCKS_4B  = N / 4;

// 4-bit level signals (level 3)
wire [BLOCKS_4B-1:0] prop_4b_blocks, gen_4b_blocks;
wire [BLOCKS_4B-1:0] carry_4b_blocks;

// 16-bit level signals (level 2)
wire [BLOCKS_16B-1:0] prop_16b_blocks, gen_16b_blocks, carry_16b_blocks;

// Global level (64 bits) signals
wire prop_global_64b, gen_global_64b;

genvar i, j;
generate
for (j = 0; j < BLOCKS_16B; j = j + 1) begin : group16
// Instantiate four 4-bit CLAs for each 16-bit block
for (i = 0; i < 4; i = i + 1) begin : group4
carry_lookahead_4bit add4_inst (
.A   (A[((16*j)+(i+1)*4) -1 : (16*j)+(i*4)]),
.B   (B[((16*j)+(i+1)*4) -1 : (16*j)+(i*4)]),
.cin (carry_4b_blocks[(4*j)+i]),
.S   (S[((16*j)+(i+1)*4) -1 : (16*j)+(i*4)]),
.Pg  (prop_4b_blocks[(4*j)+i]),
.Gg  (gen_4b_blocks[(4*j)+i])
);
end

// Second level Carry Lookahead (16 bits)
// Generates carry signals for the 4-bit blocks within this 16-bit group
CLA4 cla16_inst (
.P   (prop_4b_blocks[(j+1)*4 - 1 : j*4]),
.G   (gen_4b_blocks[(j+1)*4 - 1 : j*4]),
.cin (carry_16b_blocks[j]),
.C   (carry_4b_blocks[(j+1)*4 - 1 : j*4]),
.Pg  (prop_16b_blocks[j]),
.Gg  (gen_16b_blocks[j])
);
end
endgenerate

// Third level Carry Lookahead (64 bits)
// Generates carry signals for the 16-bit blocks
CLA4 cla64_inst (
.P   (prop_16b_blocks),
.G   (gen_16b_blocks),
.cin (carry_in),
.C   (carry_16b_blocks),
.Pg  (prop_global_64b),
.Gg  (gen_global_64b)
);

// Final global carry output
assign carry_out = gen_global_64b | (prop_global_64b & carry_in);

endmodule

// Auxiliary carry generator module for 4 CLA modules
// Computes carry signals using group propagate and generate signals
module CLA4 (
input  [3:0] P, G,
input  cin,
output [3:0] C,
output Pg, Gg
);

assign C[0] = cin;

// Generate carry signals for positions 1-3
genvar i;
generate
for (i = 1; i < 4; i = i + 1) begin : carries_chain
assign C[i] = G[i-1] | (P[i-1] & C[i-1]);
end
endgenerate

// Group Propagate
assign Pg = &P;  // P[3] & P[2] & P[1] & P[0]

// Group Generate
assign Gg = G[3]
| (P[3] & G[2])
| (P[3] & P[2] & G[1])
| (P[3] & P[2] & P[1] & G[0]);
endmodule
```

Listing 4: 64-bit Full Carry Lookahead Adder Implementation

The auxiliary `CLA4` module implements the same carry generation logic as the 4-bit CLA but operates on group signals rather than individual bits. The global carry-out is computed from the third-level group signals using the fundamental carry equation $C_{out} = G + P \cdot C_{in}$.

A key implementation detail involves the optimized bus distribution across hierarchy levels. The nested `generate` blocks partition the 64-bit operands into appropriate slices: input operands are split into 4-bit chunks (`A[((16*j)+(i+1)*4)-1 : (16*j)+(i*4)]`), while propagate/generate signals are grouped into 4-bit buses (`prop_4b_blocks[(j+1)*4-1 : j*4]`) for second-level carry computation. This maintains consistent 4-bit granularity throughout the hierarchy while enabling efficient 16-bit and 64-bit aggregation.

3.2 Verification Methodology

3.2.1 Testbench Structure

All testbenches follow a consistent structure: registers drive module inputs, wires capture outputs, and waveform files enable visual analysis via GTKWave.

Input operands A and B are systematically varied to verify that output S and carry-out signals adhere to the binary addition rules established in Section 2.1.

For the Full Adder module, which has only 8 possible input combinations, visual inspection of waveforms suffices for verification. For larger modules where exhaustive visual verification becomes impractical, a comparative approach leverages Verilog's native '+' operator. This built-in operator handles multi-bit addition in a single line, automatically managing carry propagation and providing a reference result.

Since the sum of A + B may exceed the bit width of S due to carry-out, proper comparison requires concatenating S and Cout into an (N+1)-bit vector. This approach correctly handles unsigned overflow conditions.

All simulations use a timescale of 1ns/1ps. Although Verilog computes operations instantaneously without modeling propagation delay, this timescale allows clear observation of signal transitions.

Complete testbench code for all modules is provided in Appendix A.

3.2.2 Test Cases and Coverage

Verification strategies adapt to module complexity. Small modules undergo exhaustive testing, while larger modules use statistical sampling.

Full Adder: Exhaustive testing covers all 8 input combinations using a simple increment pattern:

```
repeat(7) begin
#10; // Wait 10ns to observe outputs
{A, B, cin} = {A, B, cin} + 3'b001; // Increment to next combination
end
```

Waveform inspection confirms S and Cout values against the truth table in Figure 5.

N-bit RCA: Exhaustive testing requires 2^{2N} input combinations, making it feasible only for small N. Automatic verification compares against the native '+' operator:

```
for (i = 0; i < (1 << N); i = i + 1) begin
for (j = 0; j < (1 << N); j = j + 1) begin
A = i;
B = j;
#5; // Propagation delay

expected = {1'b0, A} + {1'b0, B} + cin_t;

if ({cout_t, S} !== expected) begin
$display("MISMATCH: Time=%0t, A=%0d, B=%0d, cin=%b", $time, A, B,
    cin_t);
$display("       Expected: 0x%h (%0d)", expected, expected);
$display("   Got:       0x%h (%0d)", {cout_t, S}, {cout_t, S});
error_count = error_count + 1;
end
end
end
```

4-bit CLA: Testing all 256 input combinations requires deriving the carry-out from group signals, since the module outputs Pg and Gg instead of a direct carry-out:

```
        // Temporary carry-out calculation for verification
        wire calculated_carry_out;
        assign calculated_carry_out = Gg | (Pg & carry_in);
```

The verification then compares against the native '+' operator, with error counting and reporting.

64-bit CLA: Exhaustive testing being non-viable (2^{128} combinations), verification uses 50 random test cases:

```
        A = {$urandom, $urandom};
        B = {$urandom, $urandom};
        cin_t = $urandom % 2;  // Random carry input (0 or 1)
        #1; // Propagation delay
```

An enhanced terminal interface provides color-coded feedback, including overflow detection:

```
        else if (cout_t != 0) begin
        $display("%s[%0d] OVERFLOW%s | A=%d | B=%d | cin=%d | S=%d",
        yellow, i + 1, reset, A, B, cin_t, S);
        end
```

3.2.3 Waveform Analysis

GTKWave provides comprehensive visibility into both internal module signals and I/O ports. For functional verification, signals A, B, C_{in}, S, and C_{out} are of primary interest.

Decimal representation of A, B, and S facilitates quick verification of addition results. Figure 14 demonstrates an 8-bit RCA addition where A = 138 and B = 72 produce the expected sum S = 210, with $C_{out} = C_{in} = 0$, confirming correct operation.

Overflow conditions require careful interpretation. Figure 15 illustrates a case where A = 146 and B = 110 should sum to 256, but this value exceeds the 8-bit representation range. The sum output S = 0 might initially suggest an error. However, switching to binary representation (Figure 16) reveals the complete picture: $C_{out} = 1$ and S = 8'b00000000. The concatenated result $\{C_{out}, S\}$ = 9'b100000000= 256 confirms proper overflow handling. (Figures 14,15 and 16)

For the 64-bit CLA, waveform inspection becomes impractical due to the enormous value range (0 to $2^{64} - 1$) in decimal and the visual clutter of 64-bit binary values. These limitations reinforce the necessity of automated verification methodologies described in Section 3.3.2.

3.3 CMOS Circuit Design

3.3.1 Logic Gate Schematics

The CMOS gates derived in Section 2.5 were implemented in LTspice to validate functionality and characterize timing performance. Figure 18 shows the fundamental inverter, while Figures 19 and 20 implement 2-input OR and AND gates using NOR/NAND gates with output inverters. Figure 21 shows the 8-transistor XOR implementation, and Figure **??** demonstrates the 4-input NOR gate for fan-in delay analysis.

Two full adder implementations were constructed: Figure 23 uses discrete gates (42 transistors), while Figure 24 implements the optimized 28-transistor version derived through algebraic optimization.

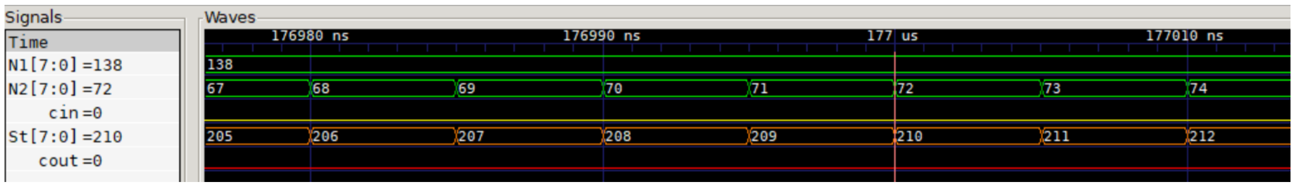


Figure 14: 8-bit RCA addition waveform (decimal representation)

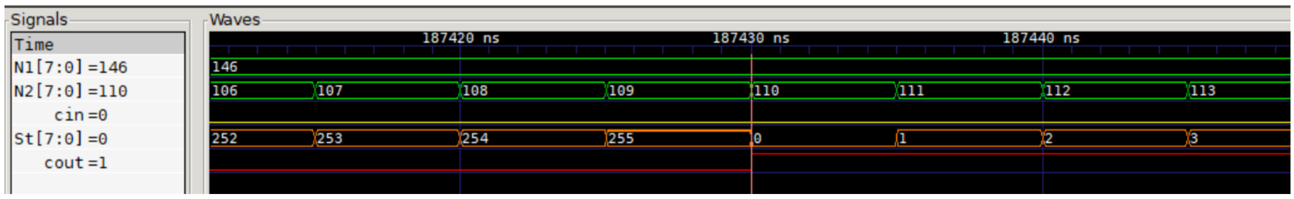


Figure 15: Overflow condition in decimal representation

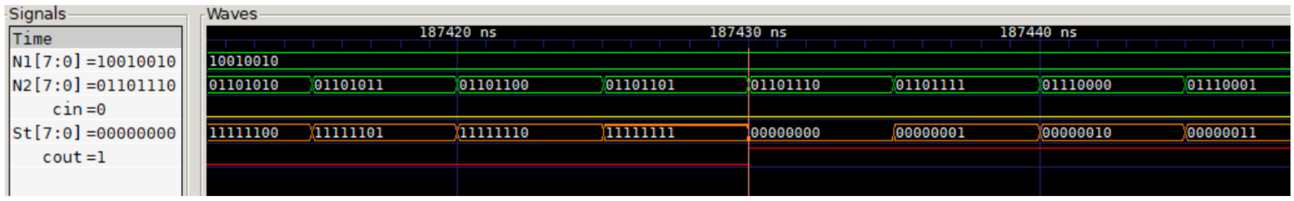


Figure 16: Overflow condition in binary representation revealing carry-out

3.3.2 SPICE Netlist Generation

All circuits were simulated using BSIM Level 1 models with technology parameters representative of a 0.18µm process. The transistor models capture essential characteristics including threshold voltages, transconductance, and parasitic capacitances:

```
.model PMOS_L PMOS (LEVEL=1 VTO=-0.6 KP=40u CGSO=0.2n CGDO=0.8n W=2u L=0.18u)
.model NMOS_L NMOS (LEVEL=1 VTO=0.6 KP=120u CGSO=0.2n CGDO=0.2n W=4u L=0.18u)
```

The NMOS-to-PMOS width ratio of 2:1 compensates for mobility differences, providing balanced rise/fall times where topology permits. Gate and drain capacitances were included to accurately model dynamic behavior.

3.3.3 Simulation Setup in LTspice

Each gate was characterized under loading conditions with $C_L = 50$ fF at the output. Input vectors A, B, C, and D (representing C_{in} for adders) provided comprehensive Boolean coverage through carefully timed pulses (Figure 17) with $V_{DD} = 1.8$ V, $t_{on} = 5$ ns, and transition times of 10-50 ps.

Propagation delays were measured automatically using LTspice's `.meas` directives, capturing transitions at the $V_{DD}/2$ switching point. For example, the XOR gate measurement:

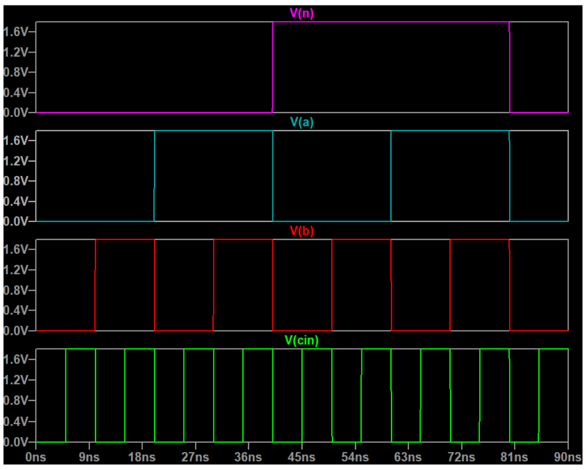


Figure 17: Input pulse configuration for testing

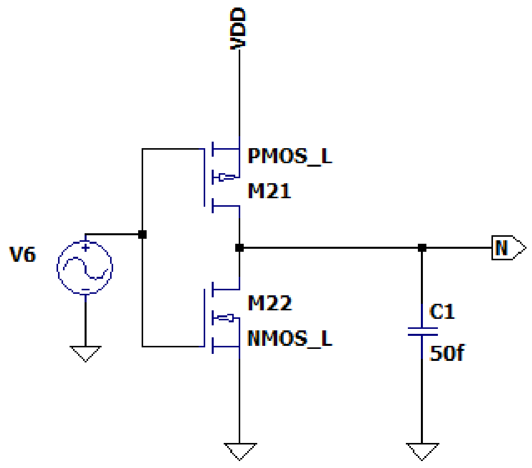


Figure 18: CMOS inverter implementation

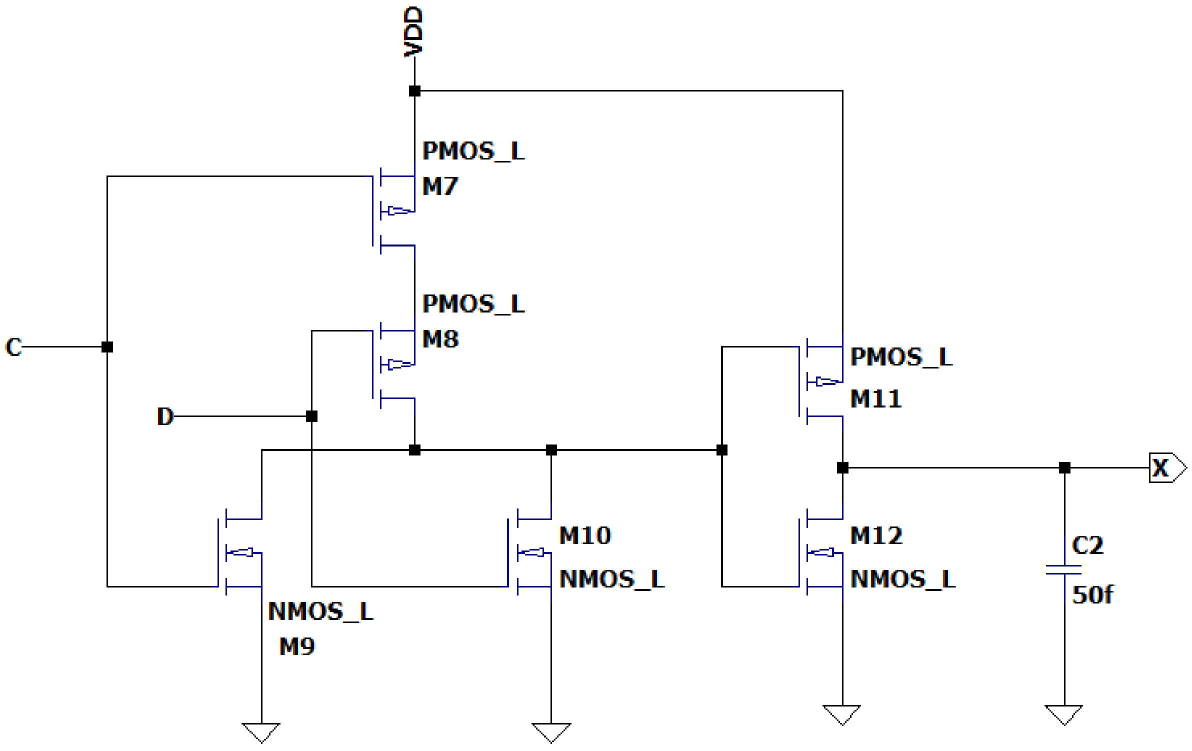


Figure 19: 2-input OR gate using NOR and inverter

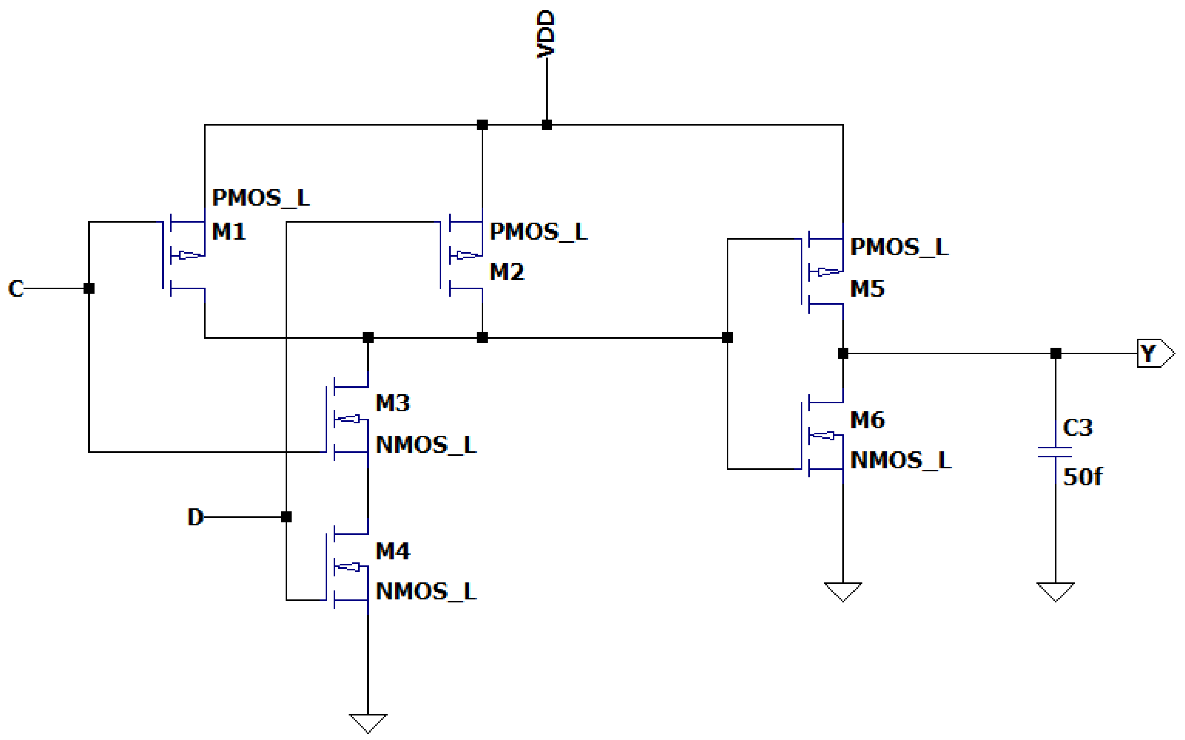
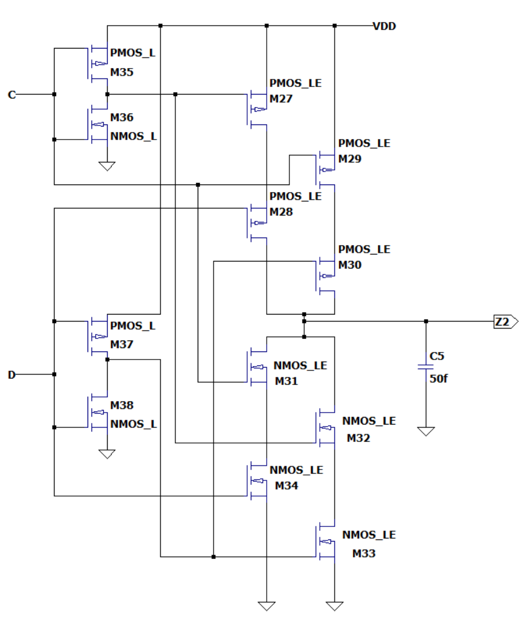


Figure 20: 2-input AND gate using NAND and inverter



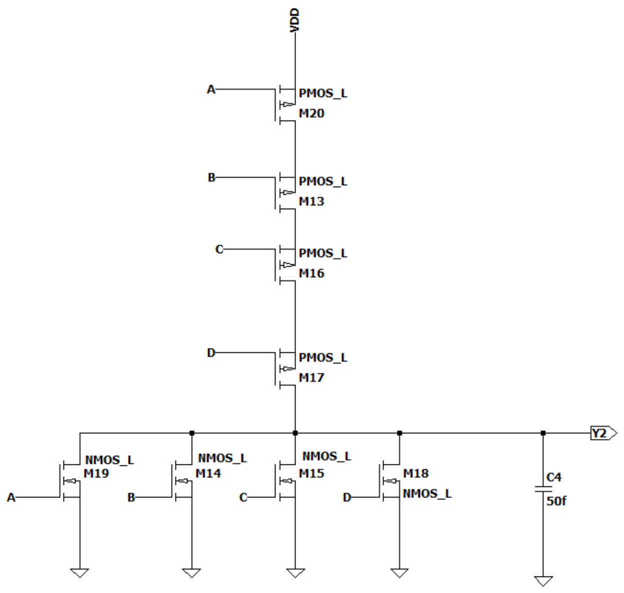


Figure 21: 12-transistor XOR gate implementation

Figure 22: 4-input NOR gate for fan-in delay analysis

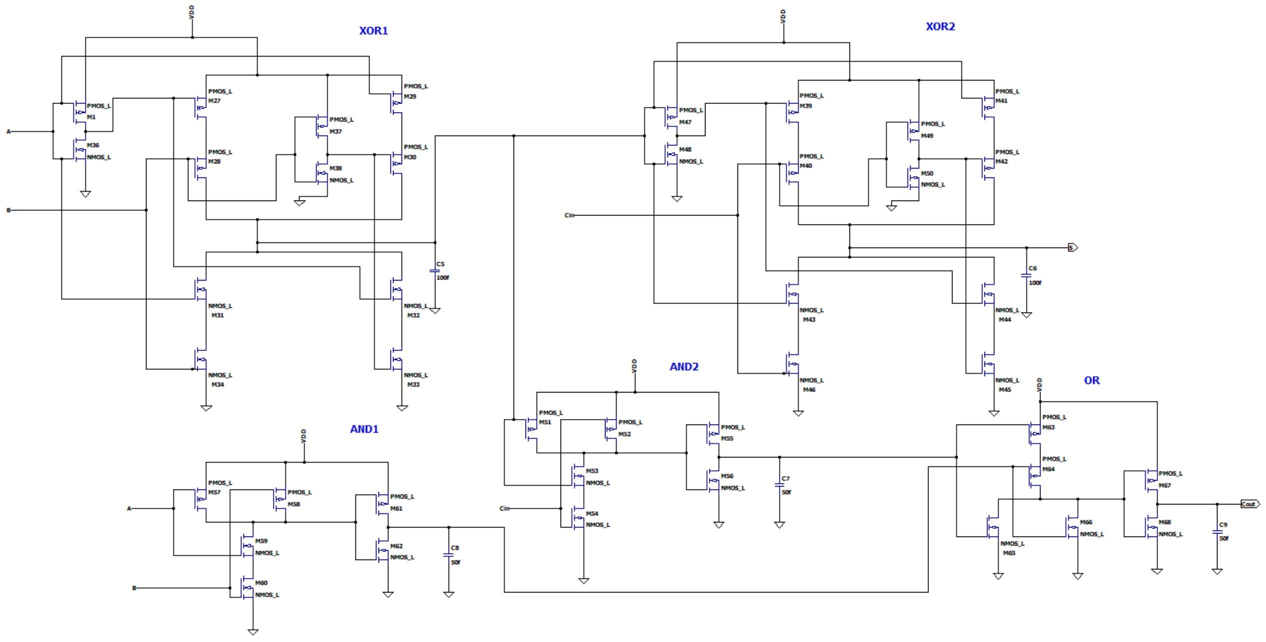


Figure 23: FA using CMOS gates (42 transistors)

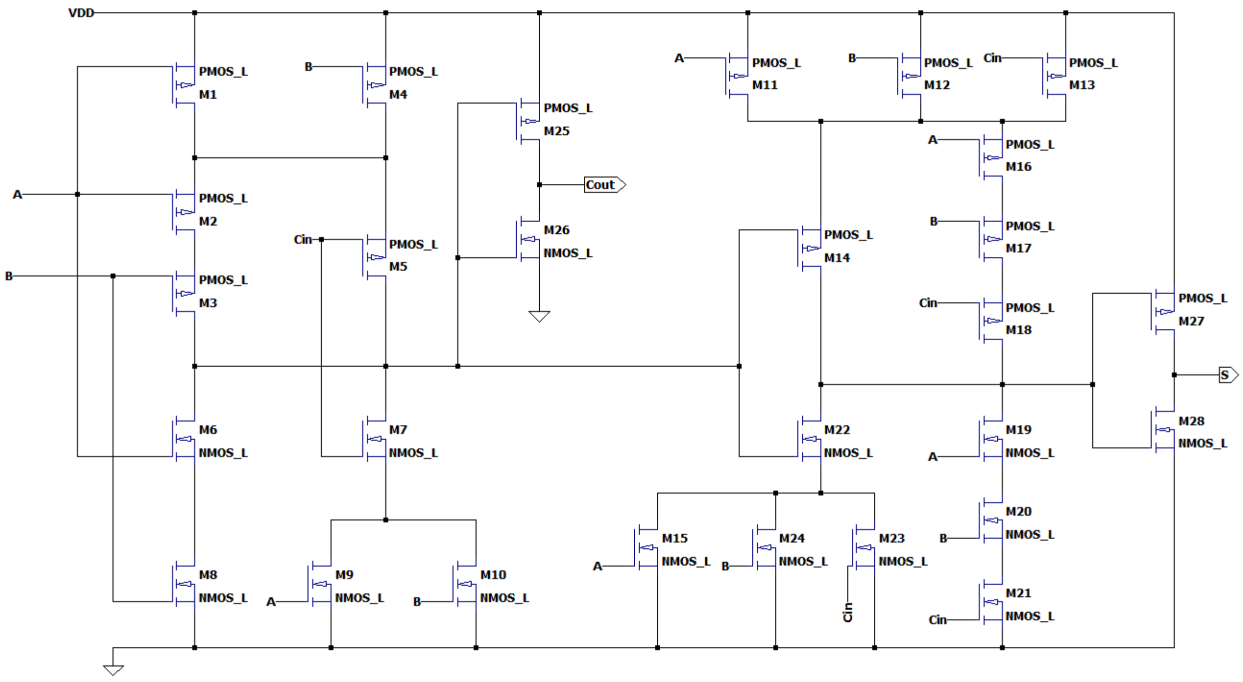


Figure 24: Optimized 28-transistor FA implementation

```
.meas TRAN T_PLH_XOR2 TRIG V(D)=0.9 RISE=1 TARG V(Z2)=0.9 RISE=1
.meas TRAN T_PHL_XOR2 TRIG V(D)=0.9 RISE=2 TARG V(Z2)=0.9 FALL=1
.meas tran tp_XOR2 param='(T_PLH_XOR2 + T_PHL_XOR2)/2'
```

This automated approach eliminated cursor placement errors and ensured consistent, precise timing measurements across all gate configurations.

4 Results and Analysis

4.1 Functional Verification

4.1.1 Full Adder Operation

Waveform analysis in GTKWave confirms the Full Adder's functional correctness across all eight possible input combinations. As shown in Figure 25, input transitions (green) occur every 10ns, while outputs (yellow) respond according to the truth table defined in Figure 5.

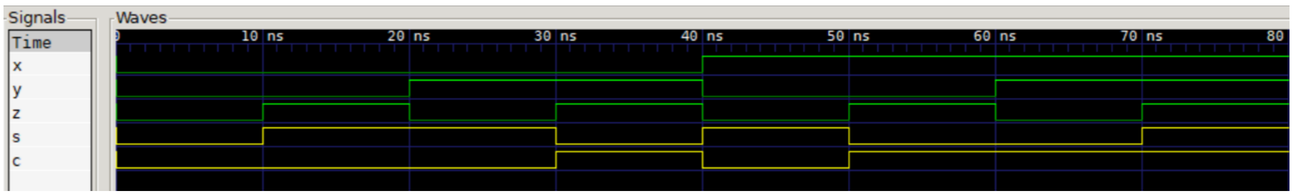
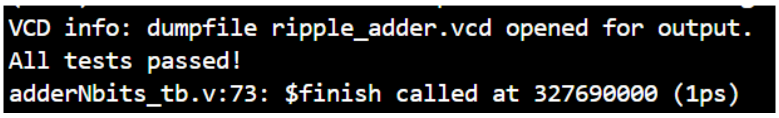


Figure 25: Full Adder waveform verification showing input combinations and corresponding outputs

The waveform demonstrates exact correspondence between simulated behavior and the theoretical truth table used for the module's design. Each input combination produces the expected sum and carry outputs, validating the Full Adder's implementation.

4.1.2 Ripple-Carry Adder Verification

The Ripple-Carry Adder verification used a parameter value of N=8, enabling testing of all 65,536 (2^{2N}) possible input combinations. Execution of the compiled .vvp testbench file automatically verified each case against Verilog's native addition operator.



```
VCD info: dumpfile ripple_adder.vcd opened for output.
All tests passed!
adderNbits_tb.v:73: $finish called at 327690000 (1ps)
```

Figure 26: Terminal output confirming successful verification of all 65,536 test cases

As shown in Figure 26, the terminal reports "All tests passed!" with zero mismatches.

Waveform analysis in GTKWave confirms proper operation, though it reveals an important characteristic of behavioral simulation: all 65,536 test cases execute with simultaneous output transitions. Figure 27 demonstrates that even worst-case carry propagation scenarios (such as A = 8'b00000001, B = 8'b11111111) show instantaneous C_{out} result.

While this verification confirms arithmetic correctness, the theoretical $O(N)$ propagation delay characteristic of the RCA architecture is not observable in this simulation. Verilog's zero-delay model computes all results within the same simulation timestep, masking the sequential carry ripple that would occur in physical implementation.

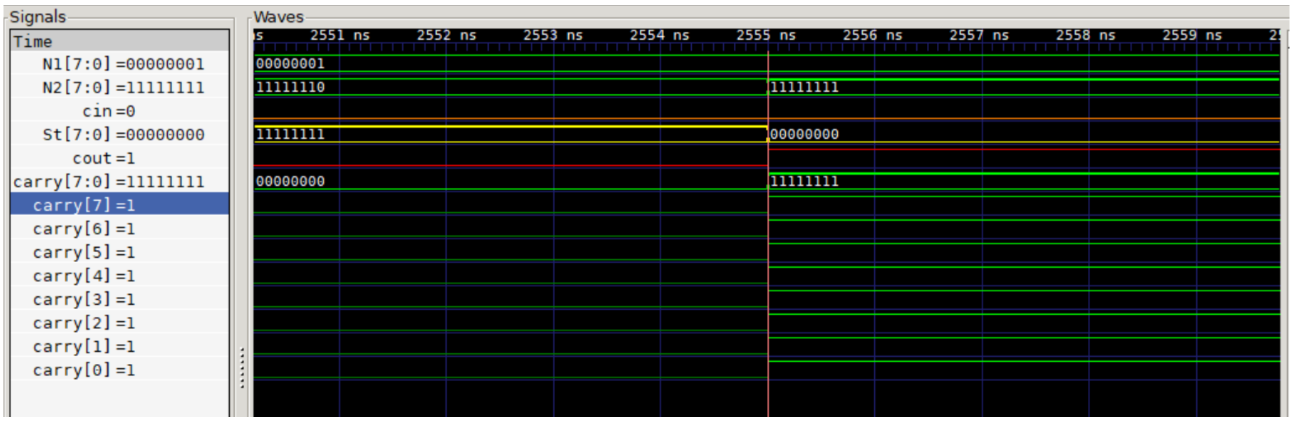


Figure 27: RCA waveform showing simultaneous output transitions across all test cases due to zero-delay simulation

The verification methodology provides complete functional validation, while timing performance is analytically derived from the theoretical gate-level analysis presented in Section 2.3.

4.1.3 64-bit CLA Comprehensive Testing

The 64-bit Carry Lookahead Adder was tested using the random sampling methodology described in Section 3.3.2. A total of 50 test cases were executed, combining random 64-bit operands with random carry-in values.

As shown in Figures 28 and 29, the color-coded terminal interface provides visual feedback:Green entries indicate successful addition without overflow, yellow entries flag unsigned overflow conditions where $C_{out} = 1$, the absence of red entries confirms zero functional mismatches.

All 50 test cases passed verification, with 28 cases (56 % of the sample) triggering overflow detection. In overflow conditions, the correct result is obtained by considering the concatenated value $\{C_{out}, S\}$, as demonstrated in the waveform analysis of Section 3.2.3.

4.2 Performance Analysis

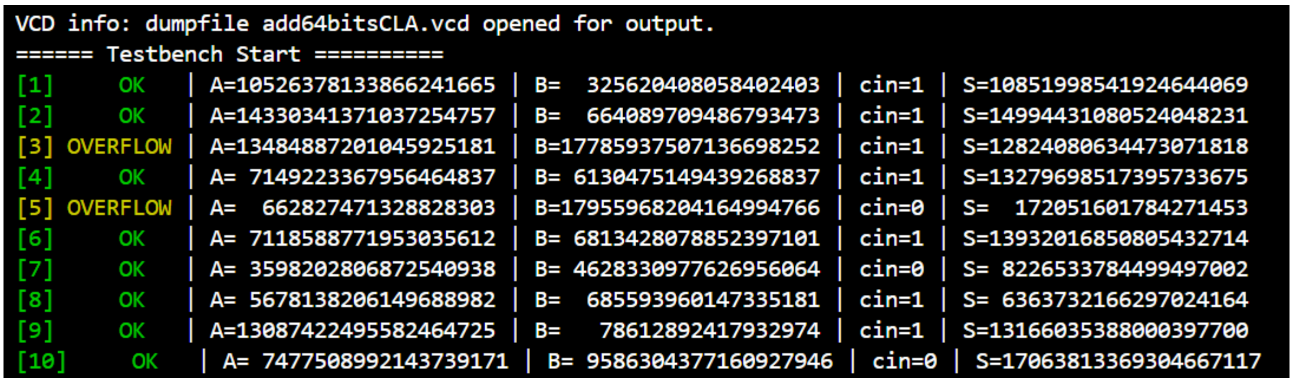
4.2.1 Theoretical Performance Comparison

Given the limitations of behavioral Verilog simulation for timing and power analysis, this evaluation focuses on theoretical performance metrics derived from gate-level analysis.

Table 1: Theoretical Performance Comparison of Adder Architectures

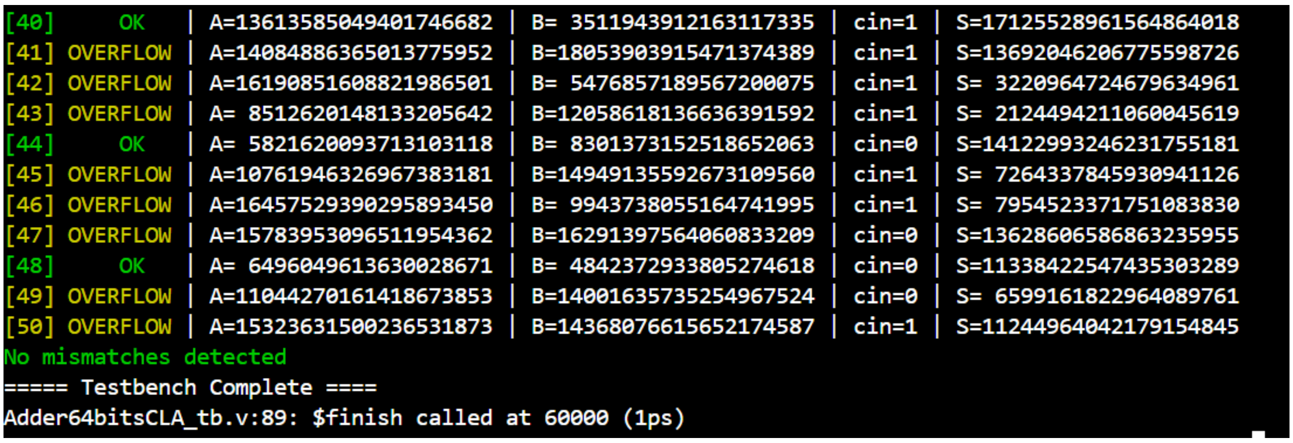
Architecture	Delay	Gate Count	Area-Delay Product
Full Adder (1-bit)	3 gate delays	5 gates	15
4-bit RCA	$3 \times 4 = 12$ gate delays	$4 \times 5 = 20$ gates	240
4-bit CLA	3 gate delays	26 gates	78
64-bit RCA	$3 \times 64 = 192$ gate delays	$64 \times 5 = 320$ gates	61,440
64-bit CLA	$4 \times \log_4 64 = 12$ gate delays	≈ 488 gates	5856

Simulation cannot capture actual propagation delays, as Verilog computes results instantaneously. The theoretical analysis from Section 2.4.3 shows the CLA achieving 16× theoretical speed improvement over RCA for 64-bit operations.



```
VCD info: dumpfile add64bitsCLA.vcd opened for output.
====== Testbench Start ==========
[1]      OK     | A=10526378133866241665 | B=  3256204080584O2403 | cin=1 | S=10851998541924644069
[2]      OK     | A=14330341371037254757 | B=  664089709486793473 | cin=1 | S=14994431080524048231
[3] OVERFLOW    | A=13484887201045925181 | B=17785937507136698252 | cin=1 | S=12824080634473071818
[4]      OK     | A= 7149223367956464837 | B= 6130475149439268837 | cin=1 | S=13279698517395733675
[5] OVERFLOW    | A=  662827471328828303 | B=17955968204164994766 | cin=0 | S=  172051601784271453
[6]      OK     | A= 7118588771953035612 | B= 6813428078852397101 | cin=1 | S=13932016850805432714
[7]      OK     | A= 3598202806872540938 | B= 4628330977626956064 | cin=0 | S= 8226533784499497002
[8]      OK     | A= 5678138206149688982 | B=  685593960147335181 | cin=1 | S= 6363732166297024164
[9]      OK     | A=13087422495582464725 | B=   78612892417932974 | cin=1 | S=13166035388000397700
[10]      OK     | A= 7477508992143739171 | B= 9586304377160927946 | cin=0 | S=17063813369304667117
```

Figure 28: First 10 test cases executed for 64-bit CLA verification



```
[40]     OK     | A=13613585049401746682 | B= 3511943912163117335 | cin=1 | S=17125528961564864018
[41] OVERFLOW   | A=14084886365013775952 | B=18053903915471374389 | cin=1 | S=13692046206775598726
[42] OVERFLOW   | A=16190851608821986501 | B= 5476857189567200075 | cin=1 | S= 3220964724679634961
[43] OVERFLOW   | A= 8512620148133205642 | B=12058618136636391592 | cin=1 | S= 2124494211060045619
[44]     OK     | A= 5821620093713103118 | B= 8301373152518652063 | cin=0 | S=14122993246231755181
[45] OVERFLOW   | A=10761946326967383181 | B=14949135592673109560 | cin=1 | S= 7264337845930941126
[46] OVERFLOW   | A=16457529390295893450 | B= 9943738055164741995 | cin=1 | S= 7954523371751083830
[47] OVERFLOW   | A=15783953096511954362 | B=16291397564060833209 | cin=0 | S=13628606586863235955
[48]     OK     | A= 6496049613630028671 | B= 4842372933805274618 | cin=0 | S=11338422547435303289
[49] OVERFLOW   | A=11044270161418673853 | B=14001635735254967524 | cin=0 | S= 6599161822964089761
[50] OVERFLOW   | A=15323631500236531873 | B=14368076615652174587 | cin=1 | S=11244964042179154845
No mismatches detected
===== Testbench Complete ====
Adder64bitsCLA_tb.v:89: $finish called at 60000 (1ps)
```

Figure 29: Final 10 test cases and verification summary confirming zero mismatches

Table 1 shows an idealized comparison of ripple-carry and carry look-ahead adders. Each logic gate (regardless of fan-in or type) is assumed to have equal delay and area. This abstraction highlights the architectural complexity: the ripple-carry adder grows linearly with word size O(N) while the CLA grows logarithmically O(log N). However, this comparison does not reflect physical realities,A 4-input AND or OR gate, for instance, is slower and larger than a 2-input one, and XOR gates are especially expensive in transistor count.

Thus, while the CLA appears to have low theoretical delay, in practice its complex carry-generation logic and high fan-in gates make it slower and more area-consuming than this ideal model suggests.

Despite these practical considerations, the theoretical analysis validates the CLA's architectural superiority for large operand sizes, particularly in applications where speed outweighs area constraints.

4.3 CMOS Simulation Results

4.3.1 Gate-Level Timing Characteristics

While Verilog simulations typically assume uniform gate delays, physical CMOS implementations exhibit significant timing variations due to capacitive effects and transistor arrangements. Table 2 summarizes measured propagation delays for various gates.

Gate	t_{PLH} **(ps)**	t_{PHL} **(ps)**	t_p **(ps)**
NOT	175.4	40.6	108.0
AND	185.0	97.6	141.3
NAND	138.0	57.9	97.9
OR	174.3	216.7	195.5
NOR4	750.1	30.7	390.4
NOR4 (Sized)	207.3	30.7	119.0
XOR	352.0	29.6	190.8
OR4	246.7	394.5	320.6
AND4	291.6	126.2	208.9

Table 2: Propagation delay times measured for different logic gates

The RC model explains these timing asymmetries. For the inverter (Figure 30), $R_P \approx 2.7\ \text{k}\Omega$ and $R_N \approx 0.56\ \text{k}\Omega$ result in $t_{PLH} > t_{PHL}$. Series configurations dramatically increase delays, as evidenced by NOR4's 750 ps rise time due to four series PMOS transistors. Proper transistor sizing improves NOR4 performance by 3.6×, demonstrating the critical importance of device optimization.

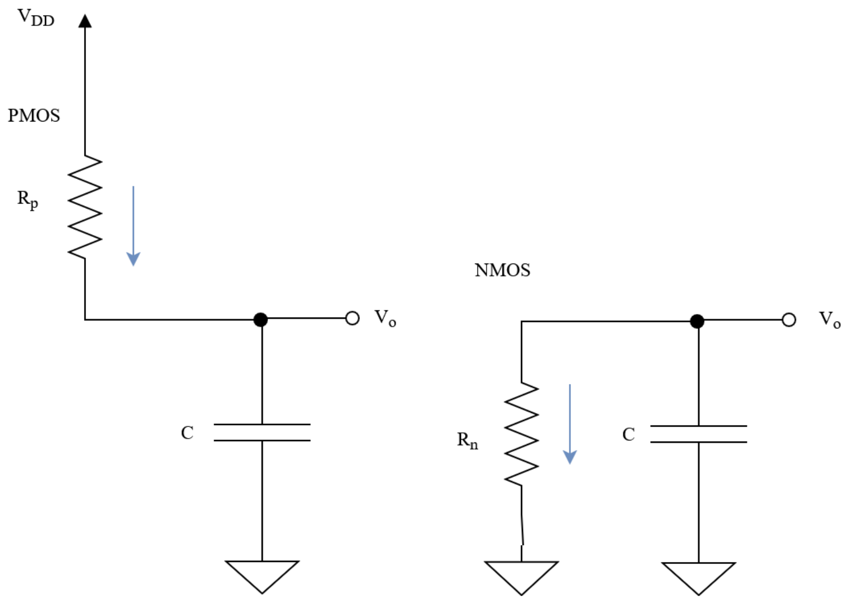


Figure 30: RC model for inverter explaining timing asymmetries

Complex gates like full adders exhibit path-dependent delays due to multiple parallel paths with different series depths. Table 3 compares gate-level and optimized 28-T implementations.

Module	t_{PLH} **(ps)**	t_{PHL} **(ps)**	t_p **(ps)**
Adder (full gates)			
S	632.5	120.0	376.2
Cout	413.7	301.8	357.8
Adder 28-T			
S (worst case)	422.1	223.5	322.8
S (best case)	338.6	203.7	271.1
Cout (worst case)	351.5	238.3	294.9
Cout (best case)	347.7	175.1	261.4

Table 3: Propagation delays for adder modules showing path-dependent timing

The 28-T optimization provides 20% speed improvement and 30% area reduction. More importantly, the significant path-dependent variations (83 ps for Sum, 4 ps for Cout) have crucial implications for adder architectures.

With this information, more precise timing analysis for larger adder modules can be performed. For example, a 4-bit ripple-carry adder using our characterized gates has a critical-path delay of:

$$t_{ripple} = 4 \times (t_{p,XOR} + t_{p,AND} + t_{p,OR}) = 4 \times (190.8 + 141.3 + 195.5) \approx 2.11 \text{ ns}$$

Using the optimized Adder-28T implementation:

$$t_{ripple28T} = 4 \times t_{p,FA28T} = 4 \times 294.9 \approx 1.18 \text{ ns}$$

For a 4-bit Carry Lookahead Adder using discrete gates:

$$t_{CLA} = t_{p,XOR} + t_{p,AND4} + t_{p,OR4} = 190.8 + 208.9 + 320.6 \approx 0.72 \text{ ns}$$

While the gate-level CLA shows a significant 300% speed improvement over the gate-level ripple-carry, this advantage reduces to approximately 160% when comparing against the transistor-optimized Adder-28T. However, this comparison is somewhat unfair, as the CLA equations could also be implemented directly at the CMOS level for additional performance gains.

4.3.2 Parasitic Impact Analysis

Parasitic effects manifest as glitches and timing variations that extend beyond simple RC models, revealing the complex dynamic behavior of CMOS circuits.

AND gates exhibited significant 0.4V glitches during specific input transitions (C↑, D↓) due to timing mismatches in series NMOS chains, as shown in Figure 31. The glitch occurred when the slow turn-off of one series NMOS created a temporary conduction path during input transitions. This was effectively eliminated by optimizing input slope timing, reducing D's fall time from 50 ps to 10 ps (Figure 32).

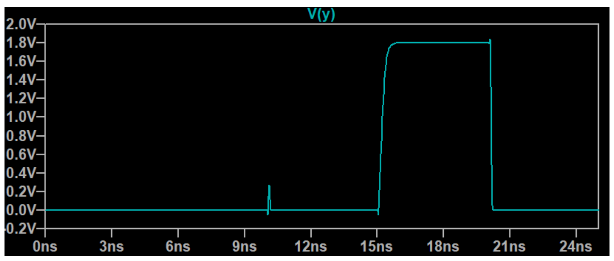


Figure 31: AND gate glitch due to series NMOS timing mismatch

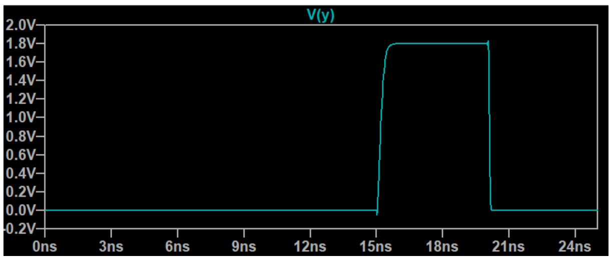


Figure 32: Glitch elimination through input slope optimization

XOR gates displayed similar but more complex glitching behavior. A negative spike (1.799V → 1.76V) occurred during (C↑, D↓) transitions, accompanied by a 0.08V positive spike during (C↓, D↓) transitions (Figure 33). Input slope optimization reduced the first glitch but amplified the second to 0.37V (Figure 34), demonstrating the trade-offs in glitch mitigation strategies.

Complex gates exhibited significant path-dependent timing variations exceeding 80 ps depending on input patterns, as evidenced in the Adder-28T waveforms (Figure 35). These variations arise from different conduction paths through series/parallel transistor networks having distinct RC characteristics.

These parasitic effects necessitate careful timing analysis, hazard detection, and input slope control in production designs to ensure signal integrity and reliable operation.

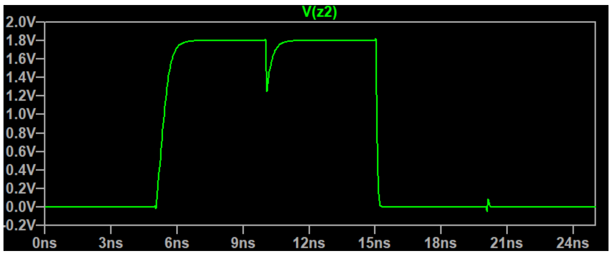


Figure 33: XOR gate glitches showing multiple parasitic effects

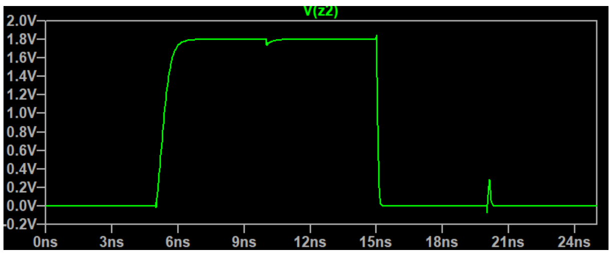


Figure 34: Trade-off in XOR glitch mitigation

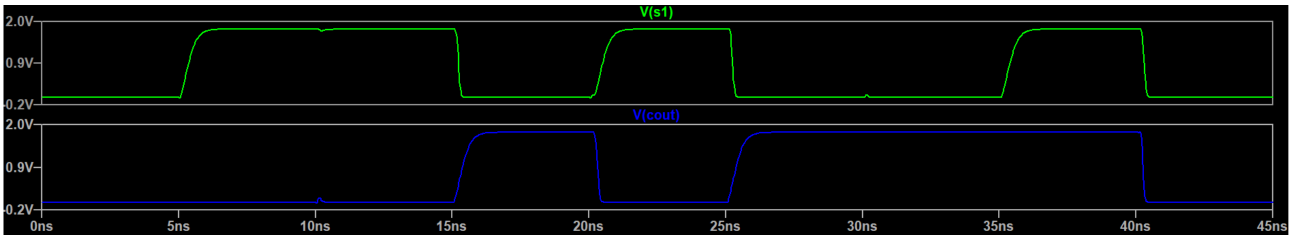


Figure 35: Path-dependent timing variations in Adder-28T

4.3.3 Layout Considerations

Simulations assume ideal wiring; real layouts introduce additional RC delays

The characterized gate delays provide a foundation for place-and-route tools to optimize timing closure in full-chip implementations.

5 Conclusions and Future Work

5.1 Conclusions

The experimental results reveal a substantial divergence between Verilog abstraction and CMOS physical implementation. While Verilog provides a simplified approach to logic design that accelerates the development of complex modules and enables efficient arithmetic and conceptual validation, the underlying CMOS reality presents physical behaviors that significantly impact circuit performance. The investigation uncovered timing asymmetries and non-instantaneous state transitions inherent to CMOS gates, where even identical fan-in/fan-out configurations exhibit propagation delays that become important in complex modules.

Direct gate-level transistor implementation proves inefficient compared to equation-driven optimization at the transistor level. The Adder-28T module demonstrates this principle, achieving approximately 30% area reduction and 20% delay improvement through Boolean optimization. LTspice simulations further revealed transient spikes and glitches during finite-slope input transitions, highlighting the necessity of careful input slope control in physical implementations. Transistor sizing emerged as a powerful optimization technique, where adjusted width-length ratios reduce resistance in series and parallel paths, enhancing current flow and reducing critical timing delays.

5.2 Limitations

This study focused primarily on RCA and CLA architectures, leaving alternative designs such as Carry-Skip Adders, Linear Carry-Select Adders, and Square-Root Carry-Select Adders unexplored. The MOS transistor models employed incorporated minimal parameters for parasitic effects, prioritizing timing delay analysis over physical phenomenon characterization. This limited approach, while sufficient for initial timing investigations, necessarily omitted certain second-order effects that would become relevant in production-grade designs. The analysis also maintained simplified loading conditions that may not fully represent the complex interconnect parasitics encountered in actual integrated circuit layouts.

5.3 Future Improvements

The full adder architecture can be extended into an adder-subtractor module by implementing a control signal M connected to the global carry-in and XOR gates at each bit of the second operand input. When M equals 0, the second operand remains unchanged and the module functions as a conventional adder; when M equals 1, the second operand becomes complemented and the module operates as a subtractor. This modification necessitates rethinking overflow detection since the MSB transitions to representing signed magnitude rather than an additional numeric digit. Future work could also incorporate more physically accurate MOS transistor models, which would capture additional parasitic effects including detailed short-channel behaviors, quantum mechanical effects, and advanced leakage current mechanisms. Further investigation could explore power consumption analysis, temperature variation effects, and process corner simulations to develop a more comprehensive understanding of real-world CMOS implementation challenges.

6 References

References

[1] A. Sedra, K. Smith, *Microelectronic Circuits*, 8th ed., Oxford University Press, 2019.

[2] M. Morris Mano, M. D. Ciletti, *Digital Design*, 6th ed., Pearson, 2018.

[3] S. Vidhyadharan, *VLSI Design Notes*, Lecture Notes

Software Tools

- **Digital by hneemann**: Logic design and simulation
- **Draw.io**: System schematics and diagrams
- **TEXstudio**: Document preparation
- **Icarus Verilog + GTKWave**: Verilog simulation
- **LTspice**: CMOS circuit analysis

Appendix A: Full Testbenches codes

```verilog

/**
* Testbench for the full_adder module.
*
* Verifies by testing all 8 possible input combinations (2^3).
* Output waveforms can be viewed in GTKWave for visual inspection.
*
*/
`timescale 1ns/1ps

module full_adder_tb();

reg  A, B, cin;  // Inputs to the Device Under Test (DUT)
wire S, cout;     // Outputs from the DUT

// Instantiate the DUT
full_adder DUT (
.x(A),
.y(B),
.z(cin),
.s(S),
.c(cout)
);

initial begin
// Initialize simulation and create waveform file
$dumpfile("full_adder.vcd");
$dumpvars(0, full_adder_tb);

// Start with all inputs at 0
{A, B, cin} = 3'b000;

// Test all 8 input combinations (0 to 7)
repeat(7) begin
#10; // Wait for 10ns to observe outputs
{A, B, cin} = {A, B, cin} + 3'b001; // Increment to next combination
end

#10 $finish;
end

endmodule
```

Listing 5: Single-bit Full Adder testbench

```verilog

/**
* Testbench for the N-bit Ripple Carry Adder (RCA)
*
* Verifies the RCA functionality by testing all possible input
* combinations for a given bit width N. Includes automatic result
   checking.
*
* Tests all 2^(2N) input combinations for operands A and B
* Compares RCA results against Verilog's native + operator
*
* @note For large N (>8), simulation time grows exponentially. Use
   small N for quick verification.
```

```
*
* Output waveforms can be viewed in GTKWave for visual inspection.
*/

`timescale 1ns/1ps

module ripple_adder_tb #(parameter N = 8) ();

reg  [N-1:0] A, B;         // Input operands to DUT
reg          cin_t;
wire [N-1:0] S;            // Sum output from DUT
wire         cout_t;

// Verification signals
reg  [N:0]   expected;     // Reference result using Verilog's native
    + operator
integer      error_count; // Tracks verification errors
integer i, j;

// Device Under Test instantiation
ripple_carry_adder #(.N(N)) DUT (
.N1(A),
.N2(B),
.cin(cin_t),
.St(S),
.cout(cout_t)
);

initial begin
// Initialize simulation
$dumpfile("ripple_adder.vcd");
$dumpvars(0, ripple_adder_tb);

error_count = 0;
cin_t = 1'b0;  // Start with carry-in = 0

for (i = 0; i < (1 << N); i = i + 1) begin
for (j = 0; j < (1 << N); j = j + 1) begin
A = i;
B = j;
#5; // Wait for propagation

// Calculate expected result using + operator
expected = A + B + cin_t;

// Compare DUT output with expected result
if ({cout_t, S} !== expected) begin
$display("MISMATCH: Time=%0t, A=%0d, B=%0d, cin=%b", $time, A, B,
   cin_t);
$display("       Expected: 0x%h (%0d)", expected, expected);
$display("       Got:      0x%h (%0d)", {cout_t, S}, {cout_t, S});
error_count = error_count + 1;

end
end
end

if (error_count == 0) begin
$display("All tests passed!");
end
```

```
else begin
$display(" %0d mismatches detected", error_count);
end

#10 $finish;
end

endmodule
```

Listing 6: N-bit Ripple Carry Adder testbench

```

/*
* Testbech for 4-bits Carry lookahead adder.
* Verifies all posible combinations for 4 bits operators A and B.
* Includes automatic result checking.
* Compares CLA results against Verilog's native + operator
*
* Output waveforms can be viewed in GTKWave for visual inspection.
*/


`timescale 1ns/1ps

module carry_lookahead_4bit_tb # (parameter N = 4) ();
reg [N-1:0] A, B;  // Input operands to test
reg          carry_in;                // Carry-in signal
wire [N-1:0] sum_out;                 // Sum output from CLA
reg [N:0]    expected;      // Expected result for verification
wire         group_generate, group_propagate;  // Group signals
integer i-out calculation for verification
wire calculated_carry_out;
assign calculated_carry_out = group_generate | (group_propagate &
    carry_in);

carry_lookahead_4bit DUT (
.A(A),
.B(B),
.cin(carry_in), , j, error_count;          // Loop counters and error
     tracker

// Temporary carry
.S(sum_out),
.Gg(group_generate),
.Pg(group_propagate)
);

initial
begin
error_count = 0;    // Initialize error counter
carry_in = 0;       // Start with carry-in = 0
$dumpfile("carry_lookahead_4bit.vcd");
$dumpvars(0, carry_lookahead_4bit_tb);


for(j = 0; j < (1 << N); j = j + 1) begin
for(i = 0; i < (1 << N); i = i + 1) begin
A = i;
B = j;
#5;

```

```
        // Calculate expected result using Verilog addition
        expected= A + B + carry_in;

        // Compare CLA output with expected result
        if ({calculated_carry_out, sum_out} !== expected) begin
        $display("Error: A=%d, B=%d, Expected=0x%h, Got=0x%h",
        A, B, expected, {calculated_carry_out, sum_out});
        error_count = error_count + 1;
        end
        end
        end

        // Test result summary
        if (error_count == 0)
        $display("All tests passed");
        else
        $display("mismatches found", error_count);

        #10 $finish;
        end

        endmodule
```

Listing 7: 4-bit Carry Lookahead Adder testbech

```
        `timescale 1ns/1ps

        /*
        * Testbench for 64-bit Carry Lookahead Adder (CLA)
        *
        * Tests the CLA implementation by comparing against Verilog's native
            addition.
        * Since testing of all 2^128 combinations is impossible, this
           testbench
        * uses 50 random input combinations to verify functionality.
        *
        *
        * - Color-coded terminal output for easy result interpretation
        * - Overflow detection and reporting
        * - Mismatch detection between CLA and native addition
        *
        * Output waveforms can be viewed in GTKWave for visual inspection.
        *
        * NOTE: Input operands are treated as unsigned for proper overflow
           detection.
        */

        module add64bitsCLA_tb #(parameter N = 64) ();

        // Testbench signals
        reg [N-1:0] A, B;
        reg cin_t;
        wire [N-1:0] S;
        reg [N:0] expected;  // N+1 bits to accommodate carry out
        wire cout_t;
        integer i, errors = 0;


        add64CLA DUT (.A(A), .B(B), .carry_in(cin_t), .S(S), .carry_out(
           cout_t));


        string green  = "\033[32m";
        string red    = "\033[31m";
        string yellow = "\033[33m";
        string reset  = "\033[0m";

        initial begin
        // Initialize inputs
        cin_t = 0;
        A = 0;
        B = 0;

        $dumpfile("add64bitsCLA.vcd");
        $dumpvars(0, add64bitsCLA_tb);

        $display("====== Testbench Start ==========");

        // Test 50 random number combinations + random input carry
        for (i = 0; i < 50; i = i + 1) begin

        A = {$urandom, $urandom};
        B = {$urandom, $urandom};
        cin_t = $urandom % 2;  // Random carry input (0 or 1)
        #1;


        // Use N+1 bits to properly handle carry out
        expected = {1'b0, A} + {1'b0, B} + cin_t;


        if ({cout_t, S} !== expected) begin
        $display("%s[%0d] MISMATCH%s | A=%d | B=%d | Exp=%d | Got=%d",
        red, i + 1, reset, A, B, expected, {cout_t, S});
        errors = errors + 1;
        end
        // Check for overflow
        else if (cout_t != 0) begin
        $display("%s[%0d] OVERFLOW%s | A=%d | B=%d | cin=%d | S=%d",
        yellow, i + 1, reset, A, B, cin_t, S);
        end

        else begin
        $display("%s[%0d]     OK%s   | A=%d | B=%d | cin=%d | S=%d",
        green, i + 1, reset, A, B, cin_t, S);
        end
        end


        if (errors == 0)
        $display("%sNo mismatches detected%s", green, reset);
        else
        $display("%s%d mismatches detected%s", red, errors, reset);

        $display("===== Testbench Complete ====");


        #10 $finish;
        end

        endmodule
```

Listing 8: 64-bit Full Carry Lookahead Adder testbench