MANIPAL INSTITUTE
OF TECHNOLOGY
MANIPAL
A Constituent Institution of Manipal University

Design & Simulation of a **32-bit Brent Kung Adder**

Yashvardhan Singh
230959136 | A2-46

Avyukth Dinesh
230959138 | A2-47

Manipal Institute of Technology
Department of Electronics and Communication Engineering
BTech in Electronics Engineering (VLSI Design and Technology)

VLSI Design Lab Mini-Project

Manipal, Karnataka, India

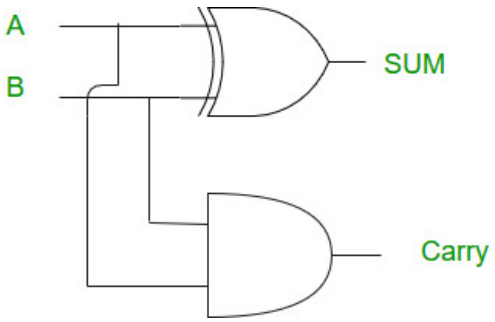
[0] Table of Contents

[1] Introduction

- Adders are fundamental components in digital circuits, serving as the building blocks for arithmetic operations essential to computing systems. They are combinational logic circuits that perform binary addition, producing a sum and a carry output based on the inputs provided. Over time, various types of adders have been developed to address the growing demands for speed, efficiency, and scalability in digital systems.

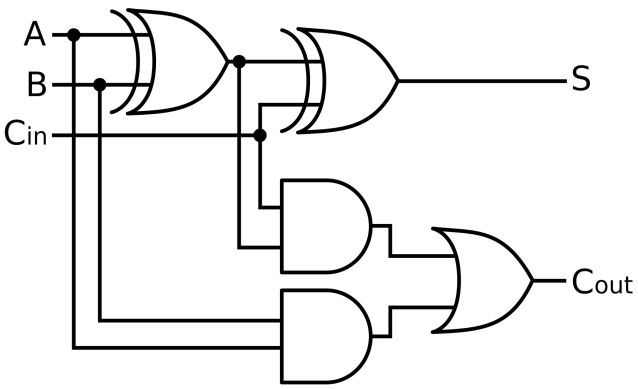
Half Adder

- The half adder is the simplest form of an adder, capable of adding two single-bit binary numbers. It produces two outputs: the sum and the carry. While it is a critical building block for more complex adders, its limitation lies in its inability to account for carry inputs from previous stages.



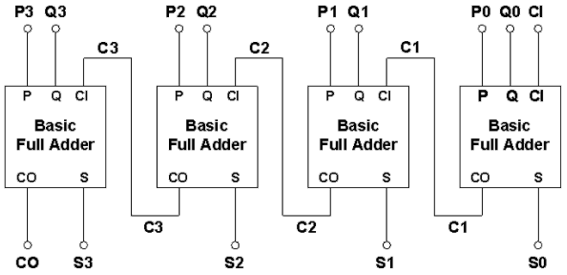
Full Adder

- To overcome the limitations of the half adder, the full adder was introduced. It can add three 1-bit binary numbers—two operands and a carry input—producing a sum and a carry output. Full adders are often cascaded to form multi-bit adders for larger binary numbers.



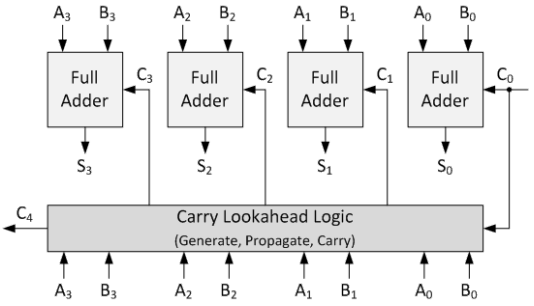
Binary Parallel Adders / Ripple Carry Adders

- When multiple full adders are connected in parallel, they form a binary parallel adder capable of adding multi-bit binary numbers simultaneously. However, this design introduces a significant limitation: carry propagation delay. In ripple-carry adders, each stage must wait for the carry from the previous stage, resulting in slower performance as the number of bits increases.



Carry Lookahead Adders

- To address the delay caused by ripple-carry propagation, the carry lookahead adder was developed. This design uses additional logic to compute carries in advance, significantly reducing delay and improving speed. Despite its advantages, carry lookahead adders require more complex hardware, which can become inefficient for very large bit-widths.



The Need for Faster Adders

- As digital systems evolved, so did the demand for faster and more efficient arithmetic operations. Applications such as high-speed processors, signal processing units, and real-time systems required adders with minimal delay and optimized resource usage. While carry lookahead adders improved performance significantly, their complexity and fan-out issues posed challenges for scalability.

Brent-Kung Adders

- The Brent-Kung adder emerged as a solution to these challenges. It is a type of parallel prefix adder that optimizes carry propagation using a tree-like structure with logarithmic depth. This design reduces computational delay and fan-out compared to traditional adders like ripple-carry or carry lookahead adders. By balancing speed and hardware complexity, Brent-Kung adders are well-suited for high-speed applications requiring efficient arithmetic operations.
- In this project, we focus on designing and simulating a 32-bit Brent-Kung adder using Verilog HDL and Cadence NCLaunch to demonstrate its advantages in terms of speed, scalability, and efficiency.

[2] Brief Description - Brent Kung Adder

32-bit Brent-Kung Adder is a logarithmic adder which does the computation of the carry faster than ripple carry adder because of the way it is designed in a tree like fashion to compute carries at each stage.

Designed by Richard P. Brent and H.T. Kung in 1982, the Brent-Kung Adder (BKA) is a well-known parallel prefix adder that provides an optimal number of stages from input to all outputs while minimizing wiring complexity.

BKA occupies less area than other adders, such as the Sparse Kogge Stone Adder (SKA), Kogge-Stone adder (KSA), and Spanning tree adder. The BKA also uses a limited number of propagating and generating cells, further contributing to its efficiency.

3 stages of Brent-Kung Adders

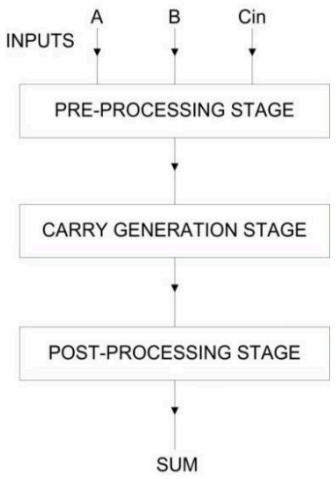
The Brent-Kung adder typically operates in three stages:

1. Pre-processing Stage: Computes Generate (G) and Propagate (P) signals from the input bits. This can also be understood as a half adder circuit. These signals are defined as:

$$Gi = Ai \cdot Bi \qquad and \qquad Pi = Ai \oplus Bi$$

2. Prefix Carry Tree Stage (Carry Generation Stage): Takes the outputs of the pre-processing stage to generate carry signals.This stage contains complex logic cells, including Black cells and Gray cells, which compute the signals. Black cells compute Gi:j and Pi:j as defined in equations such as: Gi:j= Gi:k+ Pi:k Gk-1:j and Pi:j = Pi:kPk-1:j
Gray cells compute Gi:j

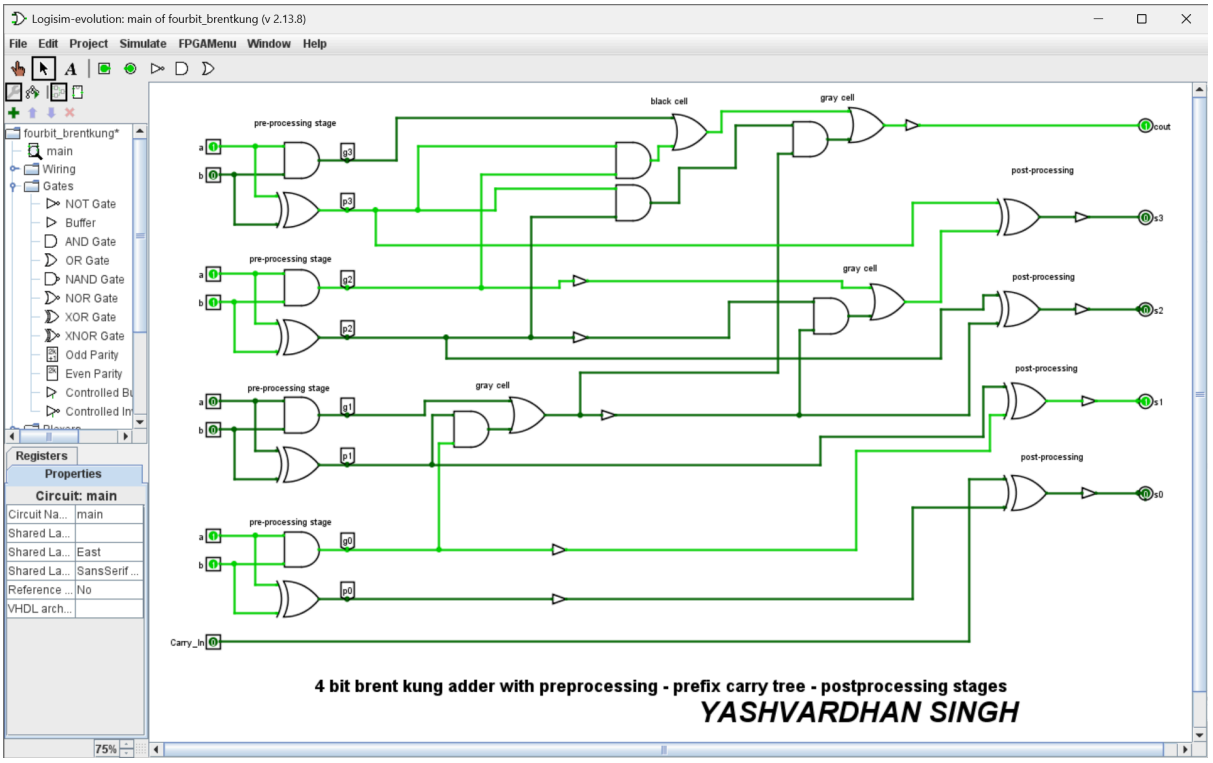
3. Post-processing Stage: Generates the final sum bits using the carry signals from the prefix carry tree stage and the propagate signals from the pre-processing stage.



Comparing Brent-Kung Adders with other adders

- Ripple-Carry Adder: High delay due to sequential carry propagation.
- Carry-Lookahead Adder: Faster but more complex in terms of hardware.
- Brent-Kung Adder: Balances speed and hardware complexity, making it ideal for high-speed applications.

To understand the gate level operation of the BKA, we designed a smaller scale 4 bit version of the adder in a logic analysis software Logisim Evolution, which gave us better understanding and helped us verify the accuracy of our design as well:

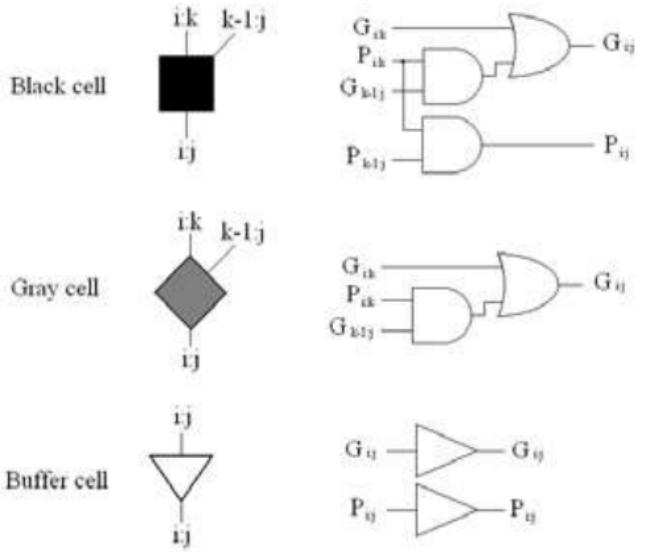


[3] Circuit Design

The BKA's circuital design is as follows:

3.1] Schematic Design

The circuit design for BKA consists of 3 main subcircuits, i.e. Black cell, Gray cell, and the buffer. We will model these modules separately in the Verilog code, and then call them as and when needed into the main module of the Brent Kung adder module.



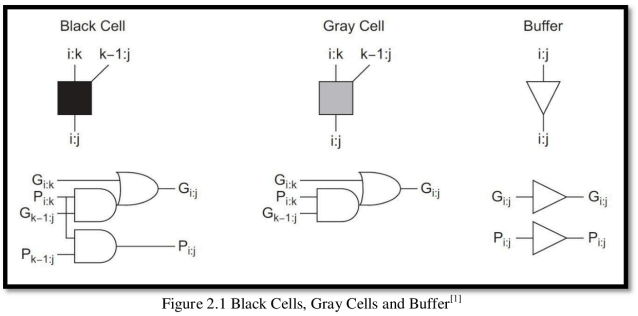


Figure 2.1 Black Cells, Gray Cells and Buffer[1]

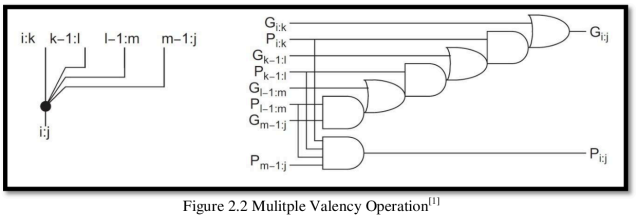
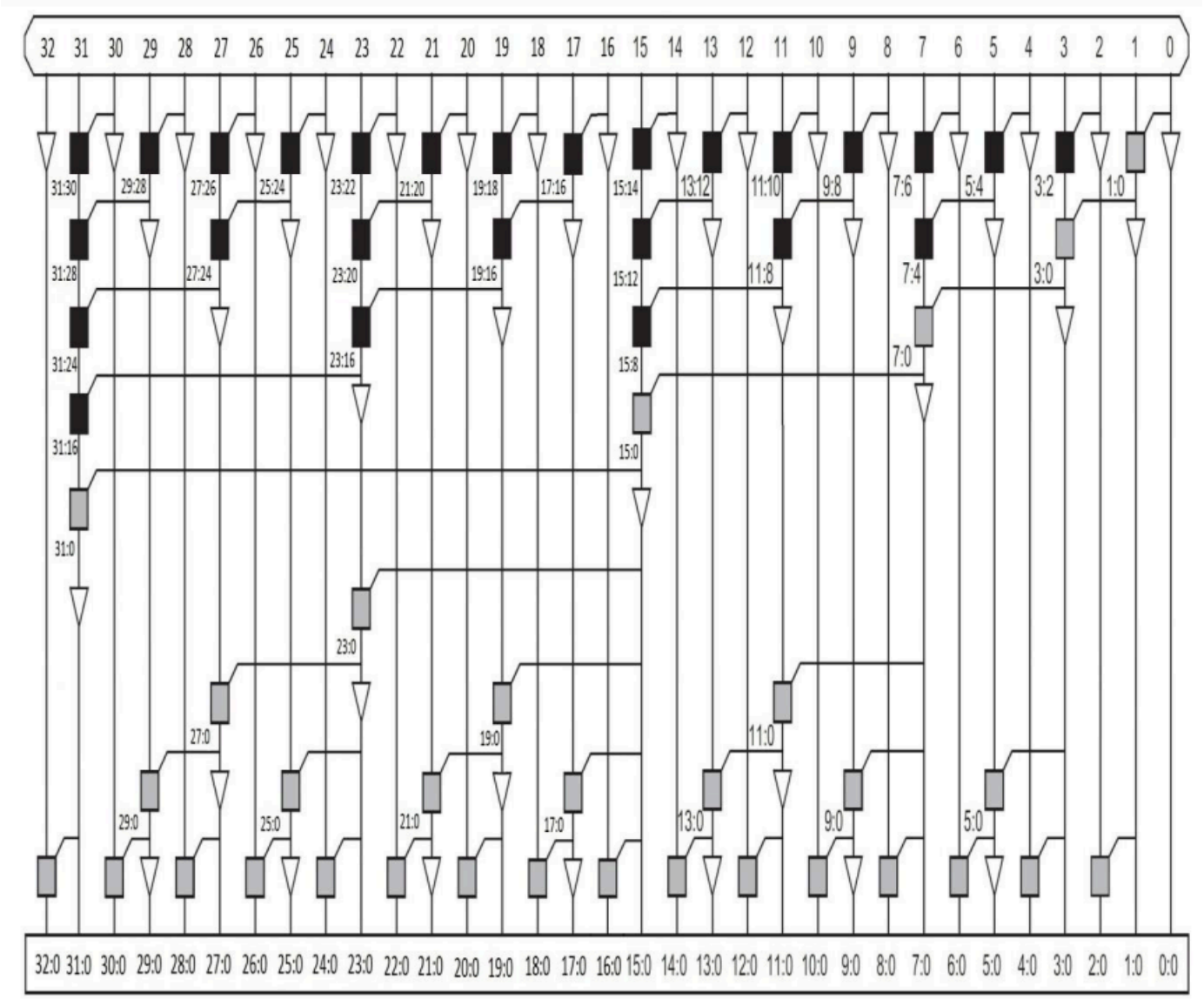


Figure 2.2 Mulitple Valency Operation[1]

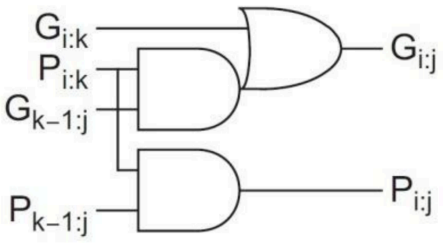


3.2] Verilog Implementation

Sub-circuits Verilog Modules:

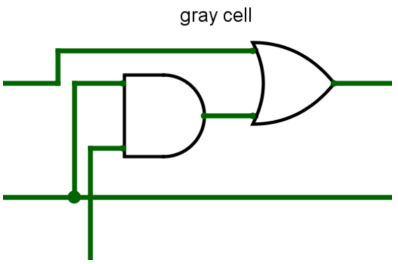
3.2.1] Black Cell Module

```
module blackcell(input gik,pik,gkj,pkj, output gij,
pij);
wire a;
assign a=pik & gkj;
assign gij=gjk + a;
assign pij=pik & pkj;
endmodule
```



3.2.2] Gray Cell Module

```
module graycell( input gi,pi,gi1, output c);
wire b;
assign b=pi&gi1;
assign c=gi|b;
endmodule
```

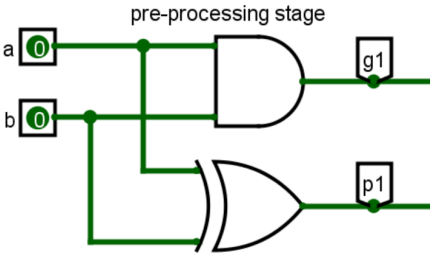


3.2.3] White Cell Module

```
module whitecell( input a, output y);
assign y=a;
endmodule
```

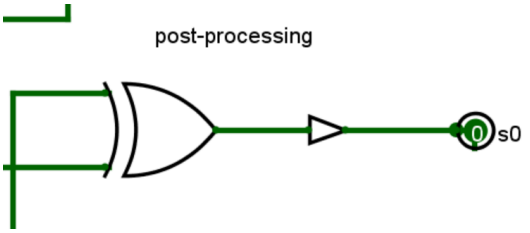
3.2.4] Pre-Processing Module

```
module preprocess( input a,b, output g,p);
assign g=a&b;
assign p=a^b;
endmodule
```



3.2.5] Post-Processing Module

```
module postprocess( input c,p, output s);
wire f;
assign f=c^p;
buf b1 #10 (s,f);
endmodule
```



Brent Kung Adder Verilog Module:

3.2.6] BK Adder Module

```
module brent_kung_adder(
```

```
    input [31:0] A, B,
    input Ci,
    output [31:0] S,
    output Co
);
    wire [31:0] P1, G1;  // First order propagate and generate
    wire [32:0] C;       // Carry signals, including C[0] for Ci
    wire [15:0] G2, P2;
    wire [7:0] G3, P3;
    wire [3:0] G4, P4;
    wire [1:0] G5, P5;
    wire G6, P6;

    // Pre-processing (Generate & Propagate)
    genvar i;
    generate
        for (i = 0; i < 32; i = i + 1) begin: preprocessing_stage
            preprocess pp (A[i], B[i], G1[i], P1[i]);
        end
    endgenerate

    // Prefix tree for carry computation
    generate
        for (i = 0; i < 16; i = i + 1) begin: second_stage
            blackcell bc (G1[2*i+1], P1[2*i+1], G1[2*i], P1[2*i], G2[i], P2[i]);
        end
        for (i = 0; i < 8; i = i + 1) begin: third_stage
            blackcell bc (G2[2*i+1], P2[2*i+1], G2[2*i], P2[2*i], G3[i], P3[i]);
        end
        for (i = 0; i < 4; i = i + 1) begin: fourth_stage
            blackcell bc (G3[2*i+1], P3[2*i+1], G3[2*i], P3[2*i], G4[i], P4[i]);
        end
        for (i = 0; i < 2; i = i + 1) begin: fifth_stage
            blackcell bc (G4[2*i+1], P4[2*i+1], G4[2*i], P4[2*i], G5[i], P5[i]);
        end
```

```
    endgenerate

    // Last level black cell
    blackcell bc_last (G5[1], P5[1], G5[0], P5[0], G6, P6);

    // Compute carries
    assign C[0] = Ci;  // Initialize C[0] with Carry-in
    assign C[1] = G1[0] | (P1[0] & C[0]);
    assign C[2] = G2[0] | (P2[0] & C[0]);
    assign C[4] = G3[0] | (P3[0] & C[0]);
    assign C[8] = G4[0] | (P4[0] & C[0]);
    assign C[16] = G5[0] | (P5[0] & C[0]);
    assign C[32] = G6 | (P6 & C[0]);

    generate
        for (i = 3; i < 32; i = i + 1) begin: carry_stage
            if (i != 4 && i != 8 && i != 16) begin
                graycell gc (G1[i-1], P1[i-1], C[i-1], C[i]);
            end
        end
    endgenerate

    // Post-processing (Sum computation)
    generate
        for (i = 0; i < 32; i = i + 1) begin: postprocessing_stage
            postprocess pp (C[i], P1[i], S[i]);
        end
    endgenerate

    assign Co = C[32];

endmodule
```

Testbench Verilog Module for verification:

3.2.7] Testbench Module

```
`timescale 1ns / 1ps

module tb_brent_kung_adder;
    reg [31:0] A, B;
    reg Ci;
    wire [31:0] S;
    wire Co;

    // Instantiate Brent-Kung Adder
    brent_kung_adder uut (
        .A(A), .B(B), .Ci(Ci),
        .S(S), .Co(Co)
    );

    initial begin
        // EPWave dump setup
        $dumpfile("waveform.vcd");
        $dumpvars(0, tb_brent_kung_adder);

        // Initialize inputs
        A = 0; B = 0; Ci = 0;
        #10; // Wait for 10 ns before starting tests

        // Test Case 1: Basic Addition
        A = 32'b00000000000000000000000000000001; // A = 1
        B = 32'b00000000000000000000000000000001; // B = 1
        Ci = 0;
        #20; // Increased delay

        // Test Case 2: Adding large numbers (Overflow case)
        A = 32'b11111111111111111111111111111111; // A = 4294967295
        B = 32'b00000000000000000000000000000001; // B = 1
        Ci = 0;
        #20;
```

```
        // Test Case 3: Carry Propagation (Adding two large positive numbers)
        A = 32'b10000000000000000000000000000000; // A = 2147483648
        B = 32'b10000000000000000000000000000000; // B = 2147483648
        Ci = 0;
        #20;

        // Test Case 4: Zero Addition
        A = 32'b00000000000000000000000000000000; // A = 0
        B = 32'b00000000000000000000000000000000; // B = 0
        Ci = 0;
        #20;

        // Test Case 5: Random Addition
        A = 32'b00000000000000000000000000001010; // A = 10
        B = 32'b00000000000000000000000000010100; // B = 20
        Ci = 0;
        #20;

        // Test Case 6: Carry-in Effect
        A = 32'b00000000000000000000000000001111; // A = 15
        B = 32'b00000000000000000000000000000001; // B = 1
        Ci = 1;
        #20;

        // Finish Simulation
        $finish;
    end

    initial begin
        $monitor("Time = %t | A = %b | B = %b | Cin = %b | S = %b | Cout = %b",
                 $time, A, B, Ci, S, Co);
    end
endmodule
```

3.3] Simulation Setup

The initial testing and verification of the code and testbench were done with the help of online in-browser simulator tool EDAplayground. We have used the IcarusVerilog 12.0 simulator alongside the -Wall -g2012 compiler.
Outputs were seen in 2 places, the first one being the $monitor results in the log file and then we saw waveform outputs using the EPWave option in the tool.

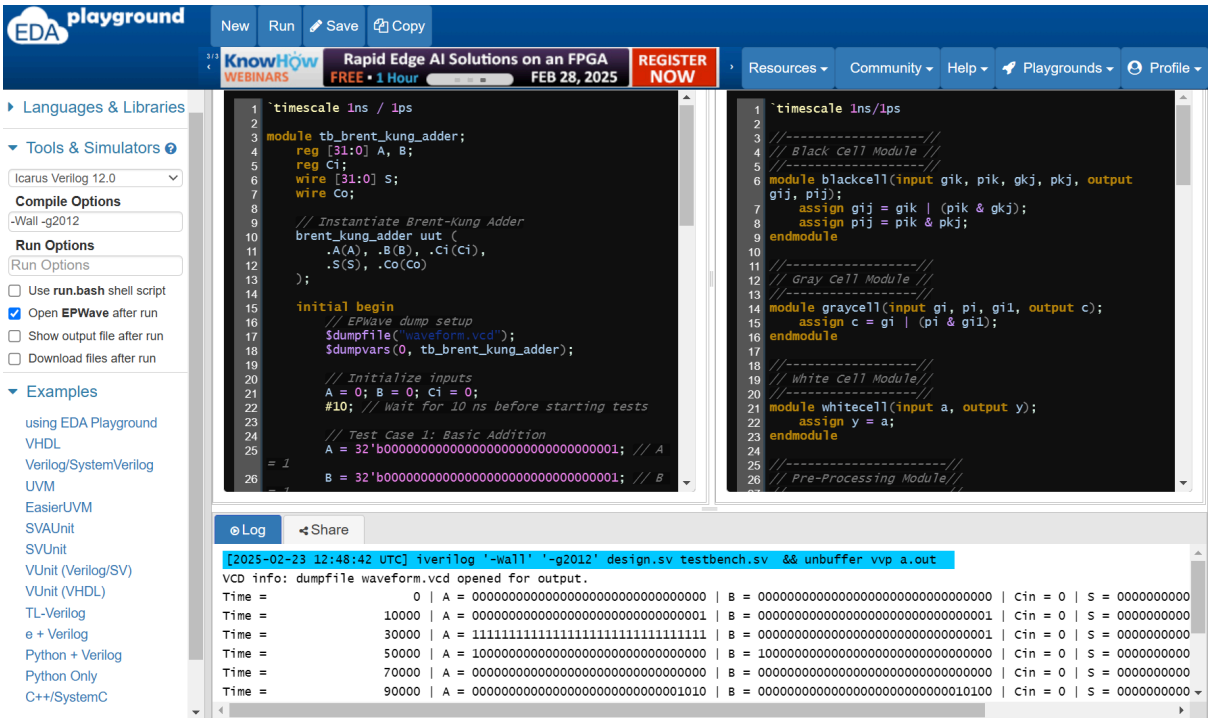
The entirety of the circuit has been described in Verilog HDL and also tested and verified in the Cadence tool using Cadence NCLaunch.

[4] Simulation & Analysis

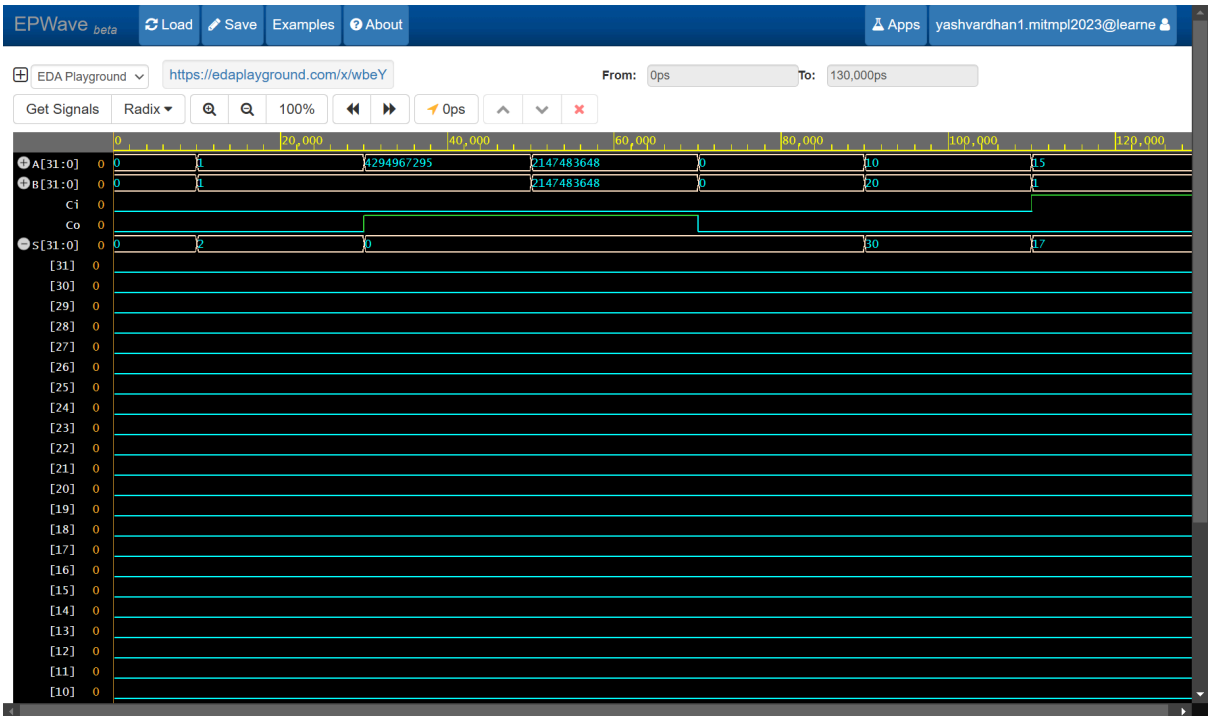
4.1] Initial EDA Playground simulation results:

Initial testing and verification were done on EDAplayground using IcarusVerilog 12.0 with -Wall -g2012. Outputs were observed via $monitor logs and EPWave waveforms.

EDA Playground simulator snapshot:



EPWave simulator snapshot:



4.2] Initial EDA Playground simulation detailed analysis:

At 0 ps (Initial state):

A = 00000000000000000000000000000000 (0 in decimal)

B = 00000000000000000000000000000000 (0 in decimal)

Cin = 0

S = 00000000000000000000000000000000

Cout = 0

This result is correct. Adding 0 and 0 with no carry in results in 0 with no carry out.

At 10000 ps:

A = 00000000000000000000000000000001 (1 in decimal)

B = 00000000000000000000000000000001 (1 in decimal)

Cin = 0

S = 00000000000000000000000000000010 (2 in decimal)

Cout = 0

This result is correct. Adding 1 and 1 results in 2, which is correctly represented in binary.

At 30000 ps:

A = 11111111111111111111111111111111 (4,294,967,295 in decimal)

B = 00000000000000000000000000000001 (1 in decimal)

Cin = 0

S = 00000000000000000000000000000000

Cout = 1

This result is correct. Adding 4,294,967,295 and 1 in a 32-bit system causes an overflow, resulting in all zeros with a carry out of 1.

At 50000 ps

:A = 10000000000000000000000000000000 (2,147,483,648 in decimal)

B = 10000000000000000000000000000000 (2,147,483,648 in decimal)

Cin = 0

S = 00000000000000000000000000000000

Cout = 1

This result is correct. Adding two numbers, each 2,147,483,648, in a 32-bit system causes an overflow. The sum (4,294,967,296) exceeds the maximum representable value, resulting in all zeros with a carry out of 1.

At 70000 ps:

A = 00000000000000000000000000000000 (0 in decimal)

B = 00000000000000000000000000000000 (0 in decimal)

Cin = 0

S = 00000000000000000000000000000000

Cout = 0

This result is correct. Adding 0 and 0 with no carry in results in 0 with no carry out.

At 90000 ps:

A = 00000000000000000000000000001010 (10 in decimal)

B = 00000000000000000000000000010100 (20 in decimal)

Cin = 0

S = 00000000000000000000000000011110 (30 in decimal)

Cout = 0

This result is correct. Adding 10 and 20 results in 30, which is correctly represented in binary.

At 110000 ps:

A = 00000000000000000000000000001111 (15 in decimal)

B = 00000000000000000000000000000001 (1 in decimal)

Cin = 1

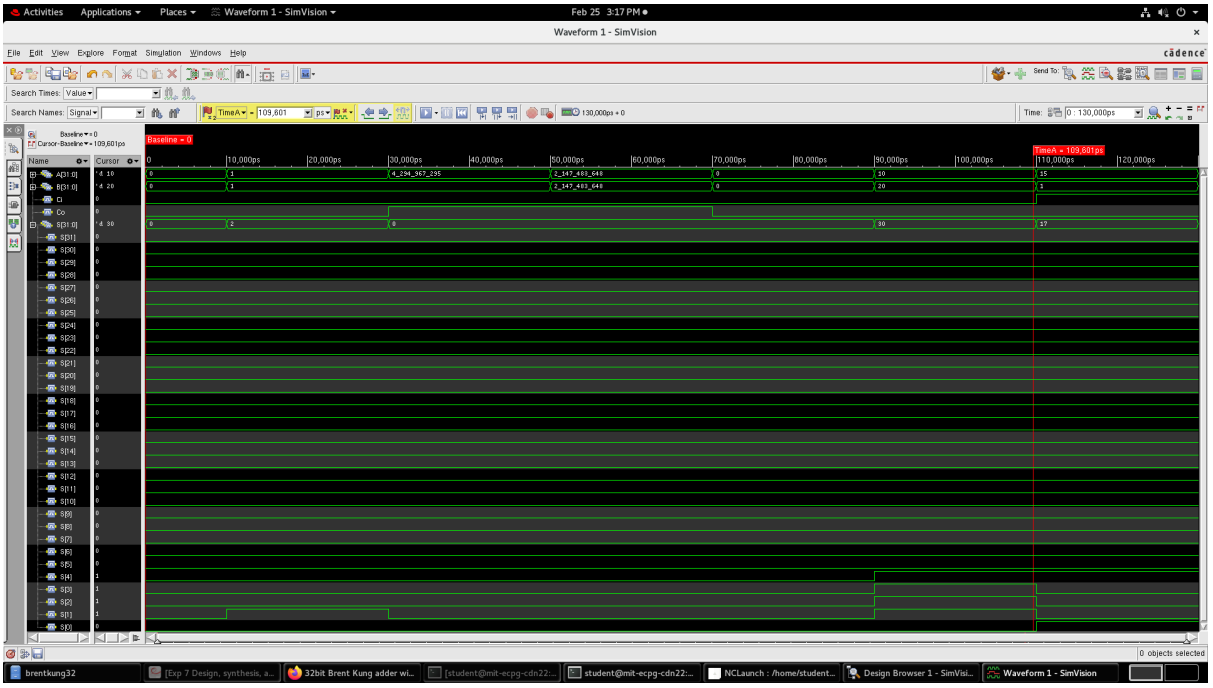
S = 00000000000000000000000000010001 (17 in decimal)

Cout = 0

This result is correct. Adding 15 and 1, plus a carry-in of 1, results in 17, which is correctly represented in binary.

All test cases demonstrate that the Brent-Kung adder is functioning correctly, handling various scenarios including basic addition, overflow conditions, and carry-in situations.

4.3] Cadence NCLaunch simulation results:



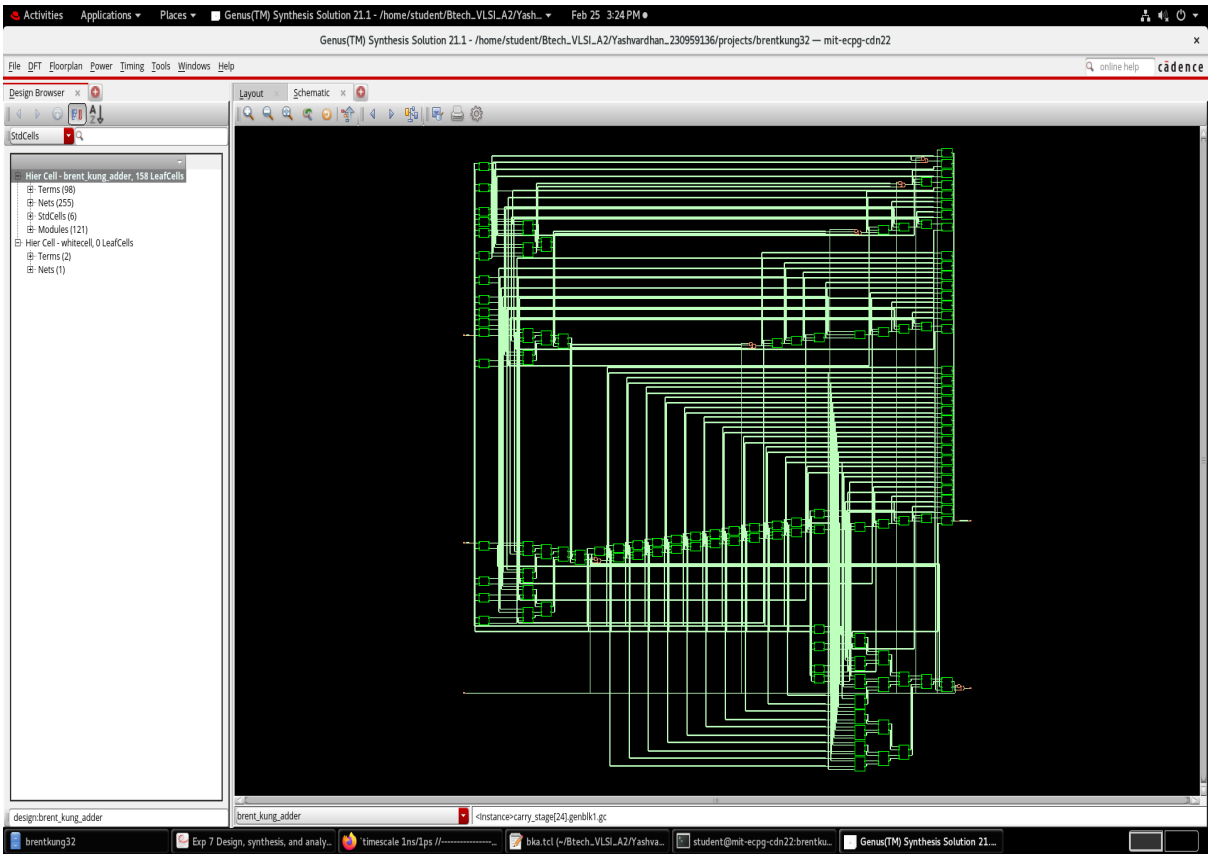
Testing and verification was also done on Cadence NCLaunch software, which gave the same outputs as the EDA Playground simulation. This adds to the accuracy of our previous results, which was discussed in great detail in the previous section.
This completes the Functional Analysis of our design, in the following section we will explore performance in terms of parameters like Power, Area and Time.

4.4] Synthesis & PAT Reports - using Cadence Genus

PAT refers to Power, Area and Time analysis which was done with the help of the Genus 21.14 tool after synthesizing the design. The Verilog code was used from our previous section and using a .tcl file we invoked unconstrained .sdc conditions to get the timing report, because the design we are working with is combination in nature.

The Synthesized design can be viewed below:

4.4.1] Synthesized Design:



4.4.2] The Power Report:

```
Instance: /brent_kung_adder
Power Unit: W
PDB Frames: /stim#0/frame#0
-------------------------------------------------------------------------------
  Category          Leakage      Internal     Switching         Total     Row%
-------------------------------------------------------------------------------
    memory      0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00    0.00%
  register      0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00    0.00%
     latch      0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00    0.00%
     logic      8.63267e-06   2.60288e-05   8.65628e-06   4.33177e-05  100.00%
      bbox      0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00    0.00%
     clock      0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00    0.00%
       pad      0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00    0.00%
        pm      0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00    0.00%
-------------------------------------------------------------------------------
  Subtotal      8.63267e-06   2.60288e-05   8.65628e-06   4.33177e-05  100.00%
Percentage            19.93%        60.09%        19.98%       100.00%  100.00%
-------------------------------------------------------------------------------
```

4.4.3] The **Area** Report:

```
====================================================================
  Generated by:           Genus(TM) Synthesis Solution 21.14-s082_1
  Generated on:           Feb 25 2025  04:16:46 pm
  Module:                 brent_kung_adder
  Operating conditions:   slow (balanced_tree)
  Wireload mode:          enclosed
  Area mode:              timing library
====================================================================
```

Instance	Module	Cell Count	Cell Area	Net Area	Total Area	Wireload
brent_kung_adder		158	1223.907	0.000	1223.907	<none> (D)
bc_last	blackcell_31	2	11.354	0.000	11.354	<none> (D)
carry_stage[3].genblk1.gc	graycell	1	6.812	0.000	6.812	<none> (D)
carry_stage[5].genblk1.gc	graycell_85	1	6.812	0.000	6.812	<none> (D)
carry_stage[6].genblk1.gc	graycell_84	1	6.812	0.000	6.812	<none> (D)
carry_stage[7].genblk1.gc	graycell_83	1	6.812	0.000	6.812	<none> (D)
carry_stage[9].genblk1.gc	graycell_82	1	6.812	0.000	6.812	<none> (D)
carry_stage[10].genblk1.gc	graycell_81	1	6.812	0.000	6.812	<none> (D)
carry_stage[11].genblk1.gc	graycell_80	1	6.812	0.000	6.812	<none> (D)
carry_stage[12].genblk1.gc	graycell_79	1	6.812	0.000	6.812	<none> (D)
carry_stage[13].genblk1.gc	graycell_78	1	6.812	0.000	6.812	<none> (D)
carry_stage[14].genblk1.gc	graycell_77	1	6.812	0.000	6.812	<none> (D)
carry_stage[15].genblk1.gc	graycell_76	1	6.812	0.000	6.812	<none> (D)
carry_stage[17].genblk1.gc	graycell_75	1	6.812	0.000	6.812	<none> (D)
carry_stage[18].genblk1.gc	graycell_74	1	6.812	0.000	6.812	<none> (D)
carry_stage[19].genblk1.gc	graycell_73	1	6.812	0.000	6.812	<none> (D)
carry_stage[20].genblk1.gc	graycell_72	1	6.812	0.000	6.812	<none> (D)
carry_stage[21].genblk1.gc	graycell_71	1	6.812	0.000	6.812	<none> (D)
carry_stage[22].genblk1.gc	graycell_70	1	6.812	0.000	6.812	<none> (D)
carry_stage[23].genblk1.gc	graycell_69	1	6.812	0.000	6.812	<none> (D)
carry_stage[24].genblk1.gc	graycell_68	1	6.812	0.000	6.812	<none> (D)
carry_stage[25].genblk1.gc	graycell_67	1	6.812	0.000	6.812	<none> (D)
carry_stage[26].genblk1.gc	graycell_66	1	6.812	0.000	6.812	<none> (D)
carry_stage[27].genblk1.gc	graycell_65	1	6.812	0.000	6.812	<none> (D)
carry_stage[28].genblk1.gc	graycell_64	1	6.812	0.000	6.812	<none> (D)
carry_stage[29].genblk1.gc	graycell_63	1	6.812	0.000	6.812	<none> (D)

4.4.4] The **Timing** Report:

```
====================================================================
  Generated by:           Genus(TM) Synthesis Solution 21.14-s082_1
  Generated on:           Feb 25 2025  04:16:46 pm
  Module:                 brent_kung_adder
  Operating conditions:   slow (balanced_tree)
  Wireload mode:          enclosed
  Area mode:              timing library
====================================================================


Path 1: UNCONSTRAINED
     Startpoint: (R) B[1]
       Endpoint: (R) S[31]

                  Capture    Launch
      Drv Adjust:+       0         0

       Data Path:-    3780
```

# Timing Point	Flags	Arc	Edge	Cell	Fanout	Load (fF)	Trans (ps)	Delay (ps)	Arrival (ps)	Instance Location
B[1]	-	-	R	(arrival)	1	3.8	0	0	0	(-,-)
preprocessing_stage[1].pp/g26__7482/S	-	A->S	F	ADDHXL	3	5.2	94	207	207	(-,-)
second_stage[0].bc/g29__9315/Y	-	A1->Y	F	AO21X1	2	3.4	61	168	376	(-,-)
third_stage[0].bc/g29__1666/Y	-	A1->Y	F	AO21X1	2	3.4	61	160	536	(-,-)
fourth_stage[0].bc/g29__8428/Y	-	A1->Y	F	AO21X1	2	3.4	61	160	696	(-,-)
fifth_stage[0].bc/g29__2398/Y	-	A1->Y	F	AO21X1	2	3.4	61	160	856	(-,-)
g135__6131/Y	-	B0->Y	F	AO21X1	2	5.2	73	170	1026	(-,-)
carry_stage[17].genblk1.gc/g19__7098/Y	-	A1->Y	F	AO21X1	2	5.2	73	176	1201	(-,-)
carry_stage[18].genblk1.gc/g19__6131/Y	-	A1->Y	F	AO21X1	2	5.2	73	176	1377	(-,-)
carry_stage[19].genblk1.gc/g19__1881/Y	-	A1->Y	F	AO21X1	2	5.2	73	176	1553	(-,-)
carry_stage[20].genblk1.gc/g19__5115/Y	-	A1->Y	F	AO21X1	2	5.2	73	176	1728	(-,-)
carry_stage[21].genblk1.gc/g19__7482/Y	-	A1->Y	F	AO21X1	2	5.2	73	176	1904	(-,-)
carry_stage[22].genblk1.gc/g19__4733/Y	-	A1->Y	F	AO21X1	2	5.2	73	176	2080	(-,-)
carry_stage[23].genblk1.gc/g19__6161/Y	-	A1->Y	F	AO21X1	2	5.2	73	176	2255	(-,-)
carry_stage[24].genblk1.gc/g19__9315/Y	-	A1->Y	F	AO21X1	2	5.2	73	176	2431	(-,-)
carry_stage[25].genblk1.gc/g19__9945/Y	-	A1->Y	F	AO21X1	2	5.2	73	176	2607	(-,-)
carry_stage[26].genblk1.gc/g19__2883/Y	-	A1->Y	F	AO21X1	2	5.2	73	176	2782	(-,-)
carry_stage[27].genblk1.gc/g19__2346/Y	-	A1->Y	F	AO21X1	2	5.2	73	176	2958	(-,-)
carry_stage[28].genblk1.gc/g19__1666/Y	-	A1->Y	F	AO21X1	2	5.2	73	176	3134	(-,-)
carry_stage[29].genblk1.gc/g19__7410/Y	-	A1->Y	F	AO21X1	2	5.2	73	176	3310	(-,-)
carry_stage[30].genblk1.gc/g19__6417/Y	-	A1->Y	F	AO21X1	2	5.2	73	176	3485	(-,-)
carry_stage[31].genblk1.gc/g19__5477/Y	-	A1->Y	F	AO21X1	1	3.6	62	165	3650	(-,-)
postprocessing_stage[31].pp/g13__1881/Y	-	B->Y	R	CLKXOR2X1	1	0.0	37	130	3780	(-,-)
S[31]	-	-	R	(port)	-	-	-	0	3780	(-,-)

4.4.5] Compiled Data of PAT Reports:

- ***Total Power Consumption: 43.32 µW***
- ***Total Cell Area: 1223.91 µm²***
- ***Critical Path Delay: 3.78 ns***

4.4.6] Comparison with some existing designs:

Adder Type	Delay (ns)	Bit Width	Technology Node	Source Citation
Brent-Kung (This Project)	**3.78**	32-bit	Not specified	Our Synthesis Report
Ripple Carry (RCA)	**38.66**	32-bit	180nm	M. Y. Divya and B. K. L. Aruna, "32-Bit Adders using Different Full Adders," VRSEC, India.
Carry Look-Ahead (CLA)	**9.33**	32-bit	45nm	L. G. R. et al., "Design of 32-Bit Adder Using Carry Look-Ahead Adder," *SIT, India* .
Kogge-Stone (KSA)	**6.7**	16-bit	90nm	A. Chaturvedi & V. Gupta, "Review on Ripple Carry and Carry Look-Ahead Adders," *TIT, Bhopal, India* .
Modified KSA	**3.5**	32-bit	45nm	A. Chaturvedi & V. Gupta, "Review on Ripple Carry and Carry Look-Ahead Adders," TIT, Bhopal, India.
Han-Carlson (HCA)	**3.129**	16-bit	65nm	V. Sidharthan & M. Prasannakumar, "Comparative Analysis of Adders: Parallel-Prefix Adder for Area, Delay, and Power,"
Pass Transistor Full Adder	**7.04**	32-bit	180nm	Sidharthan, V., & Prasannakumar, M. "Comparative Analysis of Adders: Parallel-Prefix Adder for Area, Delay, and Power (

The evaluation of various adder architectures based on Power, Area, and Timing (PAT) parameters demonstrates that the Brent-Kung adder synthesized in this project achieves a critical path delay of **3.78 ns**, positioning it among the fastest adder designs compared to conventional approaches. While not the absolute fastest, it significantly outperforms traditional adders like Ripple Carry Adders (RCA), which exhibit excessive delay (38.66 ns) due to their linear carry propagation.

Compared to the Kogge-Stone (KSA) and Modified KSA adders, which achieve delays of 6.7 ns (16-bit) and 3.5 ns (32-bit) respectively, our Brent-Kung adder maintains competitive timing with a more area-efficient structure. The Han-Carlson adder, which is another parallel-prefix adder, achieves a slightly lower delay (3.129 ns for 16-bit), but its area and power overhead may limit scalability.

In terms of total power consumption, our design maintains a reasonable **43.32 µW**, making it a balanced choice when considering both performance and energy efficiency. The total cell area of **1223.91 µm²** also confirms that Brent-Kung offers a good trade-off between complexity and speed, unlike the Kogge-Stone adder, which, while fast, suffers from high area and power requirements due to its dense interconnect structure.

Thus, the results validate the Brent-Kung adder as a super-fast, efficient adder architecture, particularly suitable for high-speed applications where delay optimization is critical while keeping area constraints in check. The structured prefix computation reduces carry propagation time without introducing excessive power or area overhead, making it a viable choice for VLSI-based arithmetic units, DSP processors, and energy-efficient computing architectures.

[5] Advantages & Disadvantages

5.1 Advantages

1. High speed due to logarithmic delay.
2. Reduced fan-out compared to other parallel prefix adders.
3. Scalable design suitable for various bit-widths.

5.2 Disadvantages

1. Slightly higher hardware complexity than ripple-carry adders.
2. Requires careful optimization for power efficiency in large designs.

[6] Conclusion

The implementation and simulation of a 32-bit Brent-Kung adder demonstrate the effectiveness of parallel prefix adders in modern digital circuit design. This project successfully showcases the adder's ability to balance high-speed performance with reasonable hardware complexity, making it an excellent choice for various applications in digital systems.

Key achievements of this project include:

1. Successful design and simulation of a 32-bit Brent-Kung adder using Verilog HDL

2. Verification of the adder's functionality through comprehensive test cases

3. Demonstration of the adder's ability to handle various scenarios, including basic addition, overflow conditions, and carry-in situations

Potential future work scope: Extending this brent kung design for greater bit-widths, working with other fast adder designs like kogge-stone, han-carlson, sklansky, wallace tree, Knowles, etc.

The Codes and Files shown in this project can be accessed on GitHub:
https://github.com/yashv373/32-Bit-Brent-Kung-Adder/tree/main

[7] FAQs

Q1: What makes the Brent-Kung Adder suitable for high-speed applications?
A: Its logarithmic delay ensures faster carry propagation compared to sequential adders like ripple-carry adders.

Q2: Why use Verilog HDL for this project?

A: Verilog allows precise modeling of digital circuits and is widely supported by simulation tools like Cadence NCLaunch, Icarus Verilog, EDA Playground.

Q3: How does the Brent-Kung Adder handle scalability?
A: Its hierarchical structure ensures efficient operation even as the bit-width increases.

Q4: Why choose Brent-Kung adder as a project?
A: It's very relevant in modern computing applications, while being easier in terms of complexity compared to other Fast adders like the Kogge-Stone or Han-Carlson adder. It is very scalable as well.

[8] References:

For #1 - introduction:
1. https://testbook.com/digital-electronics/adders

For #2 - brief description:
1. https://www.ijariit.com/manuscripts/v4i3/V4I3-1383.pdf
2. https://github.com/faizaan22/32-bit-Brent-Kung-Adder/blob/main/README.md

For #3 -

Main resource for understanding schematics and internal structure: *ANAS ZAINAL ABIDIN et al: 4-BIT BRENT KUNG PARALLEL PREFIX ADDER SIMULATION STUDY USING . . IJSSST, Vol. 13, No. 3A ISSN: 1473- 51 804x online, 1473-8031 print 4-bit Brent Kung Parallel Prefix Adder Simulation Study Using Silvaco EDA Tools Anas Zainal Abidin, Syed Abdul Mutalib Al Junid, Khairul Khaizi Mohd Sharif, Zulkifli Othman, Muhammad Adib Haron Faculty of Electrical Engineering Universiti Teknologi Mara Shah Alam, 40450, Selangor, Malaysia e-mail: samaljunid@salam.uitm.edu.my*

1. https://www.ijvdcs.org/uploads/423615IJVDCS11719-175.pdf
2. https://www.ripublication.com/acst17/acstv10n10_14.pdf
3. https://onlinelibrary.wiley.com/doi/10.5402/2012/253742
4. Indonesian Journal of Electrical Engineering and Computer Science Vol. 29, No. 3, March 2023, pp. 1345~1354 ISSN: 2502-4752, DOI: 10.11591/ijeecs.v29.i3.pp1345-1354
5. https://www.semanticscholar.org/paper/Implementation-of-32-Bit-Brent-Kung-Adder-Using-Gundi/abbb8432b1cd28aee84eada6702536a3b3cf1305

End of Document