Semicustom Frontend VLSI Design and Analysis of a 32-bit Brent-Kung Adder in Cadence Suite

Yashvardhan Singh
Department of Electronics and Communication Engineering, MIT, Manipal
Email: yvs.373@gmail.com

***Abstract*—Adders are fundamental components in digital circuits, playing a crucial role in arithmetic operations within computing systems and many other applications. This paper focuses on the design and simulation of a 32-bit Brent-Kung parallel prefix adder, which is recognized for its efficient carry propagation and logarithmic delay characteristics. The Brent-Kung architecture balances computational speed and hardware complexity, making it suitable for high-speed digital applications. The design is implemented using Verilog HDL and simulated using Cadence Design Suite tools, including NCLaunch and Genus, to evaluate its performance in terms of scalability, speed, and functional working. Comparative analysis with traditional adder architectures highlights the advantages of the Brent-Kung adder for modern digital systems.**

Keywords: Brent-Kung Adder; Parallel Prefix Adder; Verilog; Digital Circuits; Semicustom VLSI Design; Cadence Design Suite

I. Introduction

Adders are logic circuits serving as building blocks for arithmetic operations in digital 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. Several prominent adder architectures are discussed below.

A. 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.

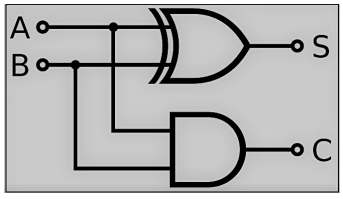

Fig. 1. Half-Adder Logic Gate Implementation

B. 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.

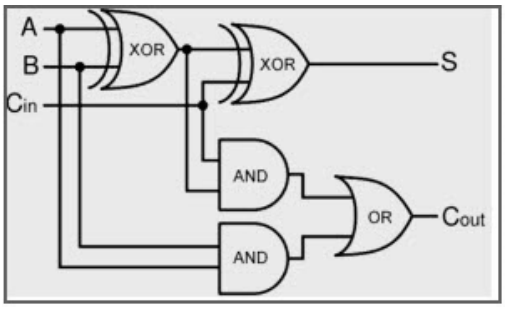

Fig. 2. Full-Adder Logic Gate Implementation

C. 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.

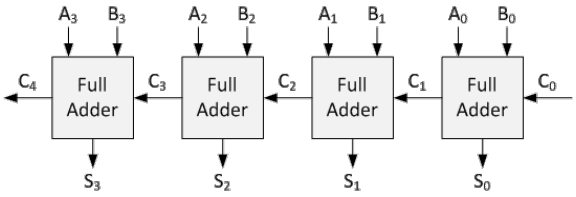

Fig. 3. Ripple Carry Adder

D. Carry Look Ahead 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.

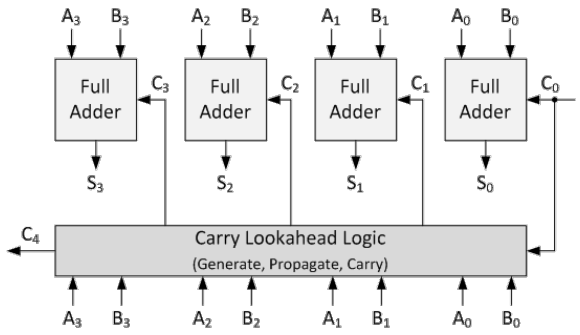

Fig. 4. Carry Look Ahead Adder

E. 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. Although carry look-forward adders improved performance significantly, their complexity and fan-out issues posed challenges for scalability.

F. Brent-Kung Adder

The 32-bit Brent-Kung Adder is a parallel prefix adder designed with a logarithmic-depth tree structure, enabling faster carry computation compared to ripple-carry adders. 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.

This paper focuses 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.

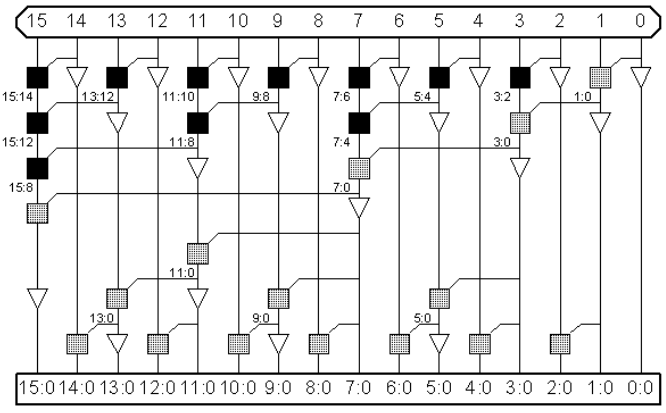

Fig. 5. Brent Kung Adder (16 bit)

II. Brief Description - Brent-Kung Adder

The 32-bit Brent-Kung Adder is a logarithmic-depth parallel prefix adder designed to compute carries faster than ripple-carry adders due to its tree-like structure.

Designed by Richard P. Brent and H.T. Kung in 1982 [1], 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 [17].

A. Stages of Brent-Kung Adders

- **Pre-processing Stage:** Computes Generate (G) and Propagate (P) signals:

$$G_i = A_i \cdot B_i, \quad P_i = A_i \oplus B_i \tag{1}$$

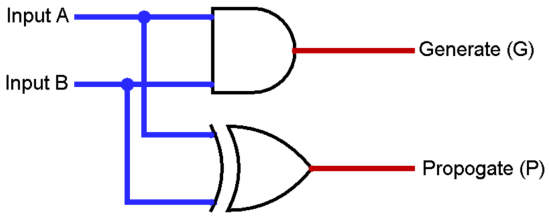

Fig. 6. Pre Processing Logic Circuit

- **Prefix Carry Tree Stage:** Uses Black and Gray cells to compute carry signals:

$$G_{i:j} = G_{i:k} + P_{i:k}G_{k-1:j}, \quad P_{i:j} = P_{i:k}P_{k-1:j} \tag{2}$$

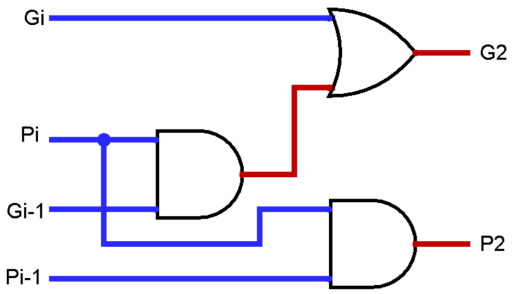

Fig. 7. Black Cell

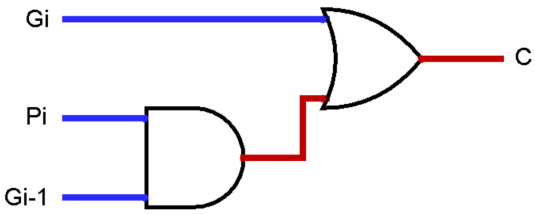

Fig. 8. Gray Cell

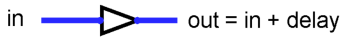

Fig. 9. White Cell

- **Post-processing Stage:** Generates sum bits using computed carry signals.

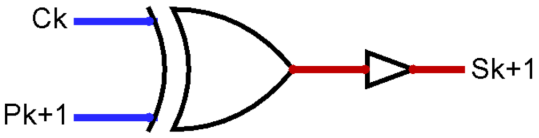

Fig. 10. Post Processing Cell

In the pre-processing stage, each pair of input bits A_i and B_i is processed to generate two signals: the Generate (G) and Propagate (P) signals.

The Carry Prefix Tree, an intermediate stage, is crucial as it efficiently computes carry signals using a structured arrangement of logic cells called Black cells and Gray cells. The data flow in this stage follows a tree-like hierarchical structure. The propagate and generate signals from the pre-processing stage serve as inputs to these cells.

This hierarchical arrangement significantly reduces carry propagation delay by computing carries in parallel rather than sequentially.

In the post-processing stage, the sum bits are computed by combining the propagate signals from the pre-processing stage with carry outputs from the prefix carry tree stage. The flow of data through the adder is aptly visualized in the flowchart shown in Fig. 11.

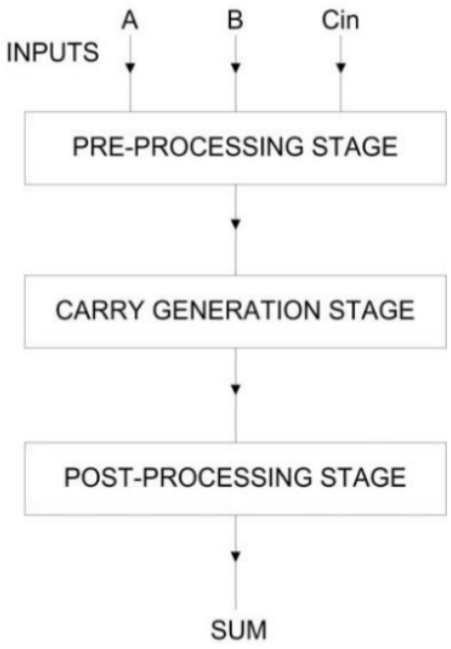

Fig. 11. Flow of data within BK Adder

B. Comparing the Brent-Kung adder 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:** Faster than RCA and CLA, and balances speed and hardware complexity, making it ideal for high-speed applications.
- *To gain a deeper understanding of the gate-level operations and verify design principles, a small-scale 4-bit Brent-Kung adder was implemented using Logisim Evolution software. This preliminary design served as a valuable learning tool and provided a clear visualization of the adder's logic structure before scaling up to the full 32-bit RTL implementation.*

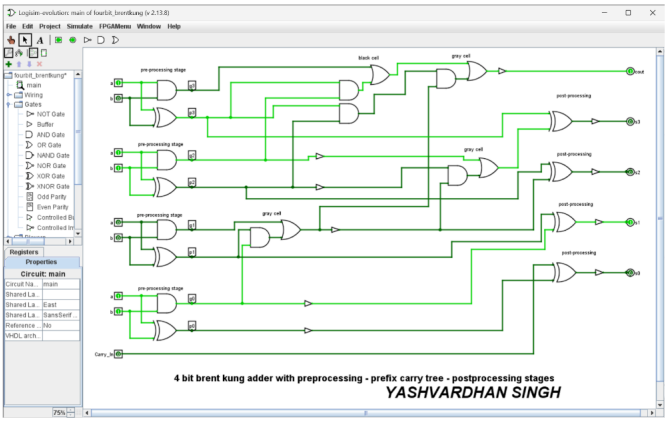

Fig. 12. Functional Analysis of 4-bit BKA in Logisim

III. Circuit Design

A. Schematic Design

The BKA consists of three main sub-circuits: Black cell, Gray cell, and White Cell(Buffer), as discussed previously. These are modeled separately as modules in Verilog HDL and integrated into the main/top BKA module. A schematic view for the same can be seen in Fig. 13. [7]

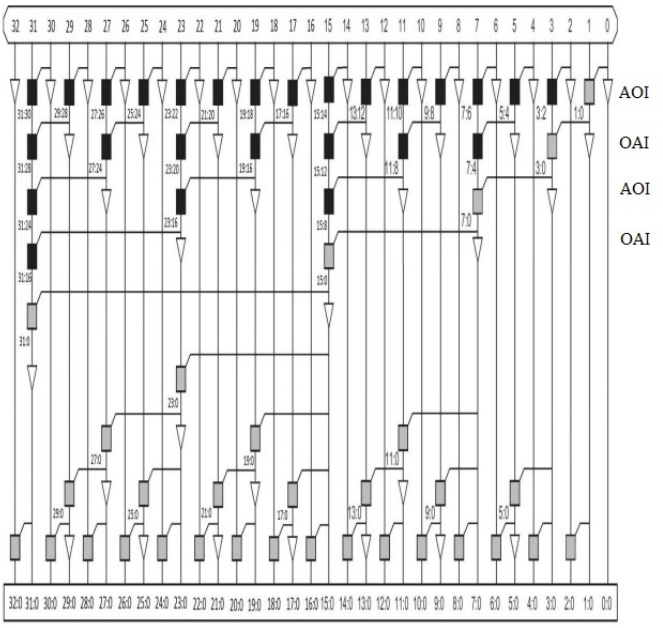

Fig. 13. 32 Bit Brent-Kung Adder Schematic

B. Verilog Implementation

The Brent-Kung adder was implemented in Verilog HDL with a modular approach, and below are the key components of the design:

```
//Black Cell Module (fig. 7)
module blackcell(input gik, pik, gkj, pkj,
output gij, pij);
wire a;
assign a = pik & gkj;
assign gij = gkj + a;
assign pij = pik & pkj;
endmodule


//Gray Cell Module (fig. 8)
module graycell(input gi, pi, gi1, output c)
wire b;
assign b = pi & gi1;
assign c = gi | b;
endmodule


//Pre-Processing Module (fig. 6)
module preprocess(input a, b, output g, p);
assign g = a & b;
assign p = a ^ b;
endmodule
```

```
//Post-Processing Module (fig. 10)
module postprocess(input c, p, output s);
wire f;
assign f = c ^ p;
buf b1 #10 (s, f);
endmodule
```

Main Adder Module: The main adder module integrates these modules, and uses them to implement the Brent Kung logic in the main module. It consists of:

- **Pre-processing stage:** Generating initial propagate and generate signals.
- **Prefix tree:** Multiple levels of black cells arranged in a logarithmic tree structure.
- **Carry computation:** Determining carry signals using gray cells.
- **Post-processing:** Computing the final sum bits.

The complete implementation, including the main adder module and testbench, is available in our GitHub repository [18].

C. Verilog Implementation - Testbench

The adder was verified using a testbench comprising of six comprehensive test cases:

- **Basic Addition:** Adding $1 + 1$ to verify fundamental operation.
- **Overflow Handling:** Testing maximum 32-bit value $(4,294,967,295 + 1)$.
- **Carry Propagation:** Adding two large numbers $(2,147,483,648 + 2,147,483,648)$.
- **Zero Addition:** Verifying the $0 + 0$ edge case.
- **Random Values:** Testing arbitrary values $(10 + 20)$.
- **Carry-in Effect:** Examining carry-in influence $(15 + 1 +$ Cin$)$.

All test cases were simulated and have been thoroughly analyzed in subsequent sections.

IV. Simulation and Synthesis

A. Initial Simulation

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

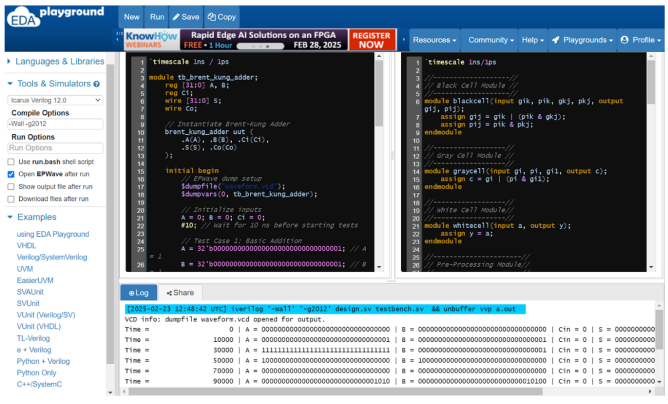

Fig. 14. Icarus Verilog - EDA-Playground Simulator

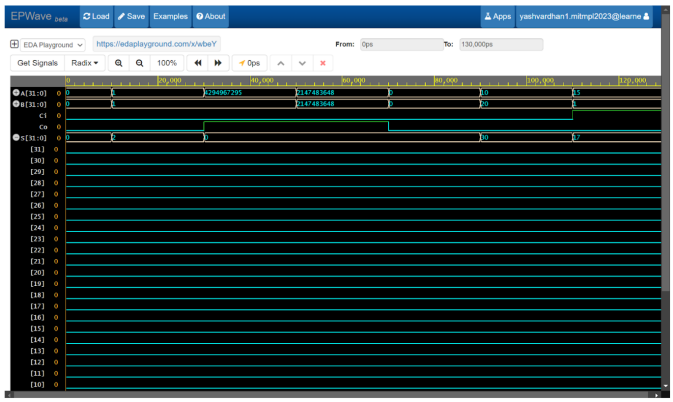

Fig. 15. Icarus Verilog - EDA-Playground Simulation - EPWave

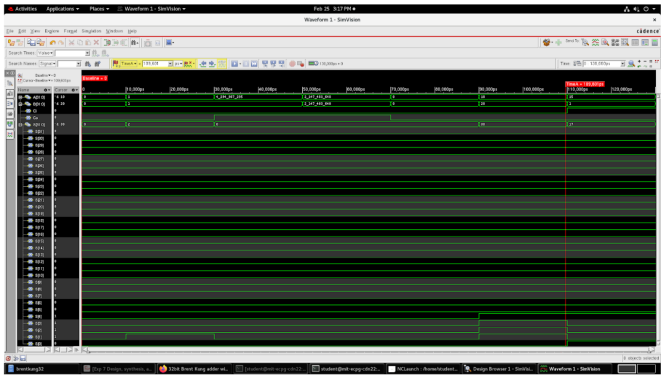

Fig. 16. Cadence NC-Launch Simulation

B. Initial Simulation: Detailed Analysis

Test	A	B	Cin	Sum	Cout	Time (ns)
1	0	0	0	0	0	0
2	1	1	0	2	0	10
3	4.29B	1	0	0	1	30
4	2.14B	2.14B	0	0	1	50
5	0	0	0	0	0	70
6	10	20	0	30	0	90
7	15	1	1	17	0	110

TABLE I
SIMULATION RESULTS FOR 32-BIT BRENT-KUNG ADDER

The simulation results in Table I demonstrate the 32-bit Brent-Kung adder's functionality across several test cases. Test 1 verified the initial state with zero inputs, producing the expected zero output. Test 2 confirms basic addition functionality by correctly computing 1+1=2. Tests 3 and 4 validated overflow handling: adding the maximum 32-bit value (4.29B)(B indicates Billion) to 1, and adding two large numbers (2.14B+2.14B), both correctly produced zero sum with carry-out=1, demonstrating proper overflow detection. Test 5 rechecked the zero-addition edge case. Test 6 verified the correct operation with arbitrary values (10+20=30). Finally, Test 7 confirmed the adder's ability to process carry-in signals by correctly computing 15+1+1=17. All test cases demonstrate that the Brent-Kung adder is functioning correctly, handling various scenarios.

C. Cadence NC-Launch Simulation

Testing and verification was next done on Cadence NCLaunch software, which gave the same results as the Icarus Verilog 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, let's explore performance in terms of parameters like power, area, and time.

D. Synthesis using Cadence Genus

Power, Area, and Timing (PAT) analysis was performed using Cadence Genus 21.14 after synthesizing the design. The Verilog code from previous sections was synthesized using Cadence Genus 21.14 with a 90nm standard-cell library, specifically using the slow corner library for conservative performance estimates.
Due to the combinational nature of the design, unconstrained timing conditions were applied via an TCL file to enable accurate timing analysis. No explicit timing constraints (.sdc file) were specified during synthesis. Instead, the -unconstrained option in Genus was used, allowing the tool to perform default timing optimization without predefined frequency or timing targets.

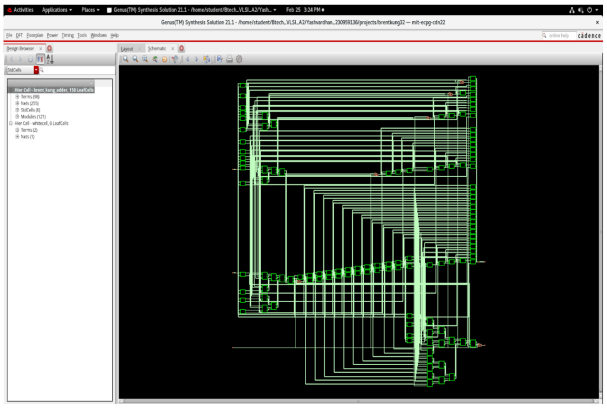

Fig. 17. Brent Kung Adder Design Synthesis using Cadence Genus

*1) **Power Report**:* The synthesized design consumes a total power of $43.32\,\mu W$, with the following power components:

- **Leakage Power:** $8.63\,\mu W$ $(19.93\,\%)$
- **Internal Power:** $26.03\,\mu W$ $(60.09\,\%)$
- **Switching Power:** $8.66\,\mu W$ $(19.98\,\%)$

This indicates that internal power dominates total consumption, followed by switching and leakage power. The obtained Power Report is as follows:

```
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%
-------------------------------------------------------------------------------------
```

Fig. 18. Power Report

*2) **Area Report**:* Total cell area of 1223.91 µm². The obtained Area Report is as follows:

```
====================================================
  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)
```

Fig. 19. Area Report

*3) **Timing Report**:* The critical path delay is of 3.78 ns. The obtained Timing Report is as follows:

```
====================================================
  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 Trans Delay Arrival Instance
#                                                                            (fF)  (ps)  (ps)   (ps)  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    (-,-)
#-------------------------------------------------------------------------------------------------------------------
```

Fig. 20. Timing Report

E. Compiled Data of PAT Reports:

Data is as follows:
Total Power Consumption: **43.32 µW**
Total Cell Area: **1223.91 µm²**
Critical Path Delay: **3.78 ns/3780ps**

F. Comparison with some existing designs:

To evaluate the effectiveness of the proposed Brent-Kung adder (BKA) architecture, we performed a comparative delay analysis of 32-bit semicustom adder implementations. Table II summarizes the critical path delays of eleven adder architectures synthesized under similar semi-custom design methodologies, ensuring a fair comparison of their performance characteristics.

Some Key observations from the analysis include:

- **BKA's Competitive Delay:** The proposed Brent-Kung Adder achieves a delay of 3.780 ns, outperforming traditional architectures such as the Ripple Carry Adder (RCA, 57.897 ns) and Carry Lookahead Adder (CLA, 44.897 ns). This demonstrates its suitability for high-speed arithmetic applications.
- **Parallel Prefix Efficiency:** Parallel prefix adders like the Kogge-Stone Adder (KSA, 21.326 ns) and Spanning Tree Adder (SPA, 31.128 ns) exhibit significantly lower delays than linear architectures (e.g., RCA), validating their logarithmic carry propagation advantage.

Full Adder Names in Table: For clarity, the abbreviated adder names in Table II correspond to the following architectures:

- **BKA:** Brent-Kung Adder
- **RCA:** Ripple Carry Adder
- **CIA:** Carry Increment Adder
- **CLA:** Carry Lookahead Adder
- **LFA:** Ladner-Fischer Adder
- **KSA:** Kogge-Stone Adder
- **CSKA:** Carry Skip Adder
- **SPA:** Spanning Tree Adder
- **MMCCA:** Modified Manchester Carry Chain Adder
- **SKSA:** Sparse Kogge-Stone Adder
- **HCA:** Han-Carlson Adder

While the Brent-Kung Adder (BKA) is not the absolute fastest design in this comparison, it strikes an optimal balance between speed and hardware complexity. With a delay of 3.780 ns, it outperforms most conventional and parallel prefix adders while avoiding the excessive area or power overheads associated with highly speculative designs like the Han-Carlson Adder (HCA). This makes the BKA a versatile choice for applications requiring reliable high-speed arithmetic without compromising on design simplicity or scalability.

Sr. No.	Adder Type	Delay (ns)	Bit Width	Source
1	BKA	3.780	32-bit	This Paper
2	RCA	57.897	32-bit	[8]
3	CIA	26.57	32-bit	[9]
4	CLA	44.897	32-bit	[8]
5	LFA	21.879	32-bit	[10]
6	KSA	21.326	32-bit	[8]
7	CSKA	25.514	32-bit	[9]
8	SPA	31.128	32-bit	[8]
9	MMCCA	31.87	32-bit	[11]
10	SKSA	19.895	32-bit	[12]
11	HCA	0.225	32-bit	[13]

TABLE II
ADDER PERFORMANCE COMPARISON (32-BIT SEMICUSTOM DESIGNS)

V. Conclusion

The Brent-Kung Adder (BKA) emerges as an optimal choice for high-speed addition requirements across various digital applications. This 32-bit semicustom frontend design demonstrates superior performance with a critical path delay of 3.78 ns, significantly outperforming traditional architectures like the Ripple Carry Adder (57.897 ns) and the Carry Lookahead Adder (44.897 ns). Additionally, it maintains a balanced power consumption of 43.32 W, with internal power contributing 60.09% of the total. In terms of hardware efficiency, the BKA utilizes an area of 1223.91 m², effectively balancing speed and complexity.

These characteristics make the BKA particularly well-suited for digital VLSI systems requiring fast arithmetic operations, high-performance computing applications, and real-time signal processing tasks. It is especially valuable in scenarios where efficient, high-speed binary addition is critical. While not the absolute fastest among all adder architectures, the BKA offers a compelling trade-off between speed, area, and power consumption. Its logarithmic delay characteristics and optimized carry propagation make it a versatile choice for modern digital system design, particularly in applications where nanosecond-level performance improvements can significantly enhance overall system efficiency.

The Codes and Files shown in this paper can be accessed on GitHub [18].

VI. References

References

[1] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Transactions on Computers*, vol. C-31, no. 3, pp. 260-264, 1982.

[2] Testbook, "Adders in Digital Electronics." [Online]. Available: https://testbook.com/digital-electronics/adders.

[3] R. R. Patil and M. B. Mulla, "Design and Implementation of Brent-Kung Adder," *International Journal of Advance Research, Ideas and Innovations in Technology (IJARIIT)*, vol. 4, no. 3, pp. 1245-1250, 2018. [Online]. Available: https://www.ijariit.com/manuscripts/v4i3/V4I3-1383.pdf.

[4] A. Z. Abidin, S. A. M. Al Junid, K. K. M. Sharif, Z. Othman, and M. A. Haron, "4-Bit Brent Kung Parallel Prefix Adder Simulation Study Using Silvaco EDA Tools," *International Journal of Simulation: Systems, Science & Technology (IJSSST)*, vol. 13, no. 3A, pp. 51-58, 2012.

[5] K. Srinivasa Rao and D. V. G. K. Reddy, "A Comparative Study of High-Speed Parallel Prefix Adders," *International Journal of VLSI Design & Communication Systems (IJVDCS)*, vol. 8, no. 5, pp. 103-112, 2017. [Online]. Available: https://www.ijvdcs.org/uploads/423615IJVDCS11719-175.pdf.

[6] P. K. Panda and S. Mahapatra, "Efficient Design of High-Speed Adders for Arithmetic Computations," *Advanced Computing: An International Journal (ACST)*, vol. 10, no. 10, pp. 123-134, 2017. [Online]. Available: https://www.ripublication.com/acst17/acstv10n1014.pdf.

[7] Elprocus, "Brent Kung Adder: Design, Architecture, and Applications," Available: https://www.elprocus.com/brent-kung-adder/.

[8] Chilaka & Kumar, "32-bit Adder Implementation and Performance Analysis," JETIR, 2020. Available: http://ijmtst.com/documents/189IJMTST021107.pdf.

[9] Yamini Divya and BKL Aruna, "Implementation of 32-bit Adders using Different Full Adders," IJERT, Oct 2020. Available: https://www.ijert.org/research/implementation-of-32-bit-adders-using-different-full-adders-IJERTV9IS100070.pdf.

[10] P. Vara Lakshmi et al., "Analysis and Design of Efficient Adder Architectures," IJARSE, 2022. Available: https://www.ijarse.com/images/fullpdf/1656228150_B1375.pdf.

[11] Ravindran and Pradeep, "Modified Manchester Carry Chain Adder Design," Trends in Innovative Computing, 2012 - Intelligent Systems Design. Available: https://www.mirlabs.net/ict12/download/Paper26.pdf.

[12] Navdeep Prashar, "VHDL Implementation of 32-bit Sparse Kogge-Stone Adder using Carry Select Logic," IJTRA, 2015. Available: https://www.researchgate.net/publication/284492604_VHDL_IMPLEMENTATION_OF_32-BIT_SPARSE_KOGGE-STONE_ADDER_USING_CARRY_SELECT_LOGIC.

[13] Sasireka S. Marimuthu, "Implementation of Han-Carlson Adder for High-Speed Computation," CN IJEER, Dec 2015. Available: https://www.researchpublish.com/upload/book/Implementation%20of%20Han-Carlson-2361.pdf.

[14] M. H. M. El-Haroun, "Design and Implementation of Low Power Brent-Kung Adder," *Hindawi Journal of Electrical and Computer Engineering*, vol. 2012, Article ID 253742, 2012. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.5402/2012/253742.

[15] A. Rahim *et al.*, "Optimized Design of Brent-Kung Adders for High-Performance Computing," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 29, no. 3, pp. 1345-1354, Mar. 2023. DOI: 10.11591/ijeecs.v29.i3.pp1345-1354.

[16] S. Gundi, "Implementation of 32-Bit Brent-Kung Adder Using GDI Technique," *Semantic Scholar*. [Online]. Available: https://www.semanticscholar.org/paper/Implementation-of-32-Bit-Brent-Kung-Adder-Using-Gundi/abbb8432b1cd28aee84eada6702536a3b3cf1305.

[17] B. Chandrika, G. Poorna Krishna, Implementation and Estimation of Delay, Power and Area for parallel Prefix Adders, International Journal for Modern Trends in Science and Technology, Vol. 02, Issue 11, 2016, pp. 41-45. Available:http://ijmtst.com/documents/189IJMTST021107.pdf.

[18] Y. Singh, "32-Bit Brent-Kung Adder," GitHub, 2025. Available: https://github.com/yashv373/32-Bit-Brent-KungAdder/tree/main.