Design Synthesis Comparative Analysis of Binary Adders

Guilherme B. Manske
CI-Inovador - UFRGS
Porto Alegre - RS - Brazil
gbmanske@inf.ufrgs.br

Paulo F. Butzen
PGMICRO - UFRGS
Porto Alegre - RS - Brazil
paulo.butzen@ufrgs.br

Renato P. Ribas
PGMICRO - UFRGS
Porto Alegre - RS - Brazil
rpribas@inf.ufrgs.br

***Abstract*—Optimizing adder efficiency is crucial to improving arithmetic units in artificial intelligence hardware, enabling faster and more energy-efficient accelerators. This project explores different adder architectures, including ripple carry adder, carry select adder, carry increment adder, carry skip adder, carry look-ahead adder, and some parallel prefix adders, evaluating their performance based on metrics obtained from logical and physical synthesis. The architectures are implemented in SystemVerilog, allowing the configuration of some design parameters. After the designs are simulated, they undergo the complete logical and physical synthesis process, resulting in the generation of the GDS2 layout. Different methodologies are used to analyze the different adder designs. The methodologies focus on area, time, and for fair comparisons. By comparing metrics such as area, delay, and power consumption, the study aims to identify the most efficient implementations. The insights from this analysis will contribute to future efforts in optimizing efficient arithmetic hardware accelerators.**

Keywords: Adder, PPA, Ripple-Carry, Carry-Look-Ahead, Arithmetic

I. Introduction

Computational systems are increasingly reliant on efficient digital arithmetic to meet the growing demands of modern applications, such as artificial intelligence, digital signal processing (DSP), and high-performance computing. At the core of these systems exist arithmetic components that must perform operations with high speed, low power consumption, and reduced area. Among the most critical components are those responsible for addition [1], as addition forms the basis of more complex operations like multiplication, division, and comparison.

Arithmetic Logic Units (ALUs) are core elements of digital processors, executing fast and efficient arithmetic and logical operations at the hardware level. Their efficiency directly impacts the overall performance and energy consumption of the entire system. Within the ALU, adder circuits are of particular interest due to their frequent use in both control and data-path operations. Improving adder performance can result in significant gains across the entire computational pipeline.

Adders are digital circuits designed to perform the binary addition of two numbers. From basic half and full adders to more complex multi-bit architectures, they are fundamental for implementing arithmetic functions in both general-purpose and specialized processors. Different architectures have been proposed to balance trade-offs among speed, area, and power consumption, each suited to specific design constraints and applications. The main problem of the most basic adders is the time necessary for the carry to go through the whole carry chain.

Parallel adder architectures were introduced to overcome the limitations of simple carry chain designs. Parallelism reduces the carry propagation delay, and structures such as Carry Select, Carry Skip, Carry Look-Ahead, and Parallel Prefix Adders (e.g., Kogge-Stone and Brent-Kung) enable faster computation. Most parallel adders use the concept of generate and propagate signals, and the group generate and propagate signals. These architectures are especially important in high-speed or high-throughput systems, where delays from sequential carry propagation are unacceptable.

To evaluate the practical impact of each adder architecture, it is essential to apply both logical and physical synthesis. Logical synthesis involves translating register-transfer level (RTL) designs into optimized logic gates, while physical synthesis maps these gates onto a die, taking into account placement, routing, and real-world constraints. Metrics such as area, timing (delay), and power are used to compare implementations in a standardized environment.

This study presents a comparative analysis of multiple 32-bit adder architectures synthesized using Cadence Genus and Innovus tools. The designs are implemented in SystemVerilog and subjected to different methodologies: one focused on the area, another on timing performance, a reference constraint comparison, and a behavioral synthesis test. The analysis highlights how each design performs under real-world ASIC implementation constraints. This analysis is crucial for identifying which adder architectures are best suited to particular design constraints, such as area, power, or performance.

The remainder of this paper is organized as follows: Section II reviews adder structures based on carry-chain mechanisms, while Section III discusses more advanced carry-propagate/generate architectures. Section IV details the methodology used in the logical and physical synthesis flows. Section V presents and analyzes the results for each methodology. Finally, Section VI concludes the paper, summarizing the findings and suggesting directions for future work.

II. Carry Chain Based Adders

Adders are fundamental building blocks in digital arithmetic circuits, responsible for performing binary addition. They are essential components of ALUs, DSPs, and various control units. At the most basic level, an adder takes two binary numbers and optionally a carry input (Cin) to produce a sum (S) and a carry output ($Cout$). The design and efficiency of adders significantly impact the overall performance, area, and power consumption of a digital system.

The simplest type of adder is the half adder which adds two 1-bit binary numbers and produces a sum and a $Cout$. The half adder schematic is shown in Fig.1. However, it cannot handle a Cin from a previous stage, which limits its usability in additions of more than 1-bit numbers. To overcome this limitation, the full adder was introduced and its schematic can be seen in Fig.2. A full adder adds three 1-bit inputs, two significant bits, and an incoming Cin, and generates a 1-bit sum and a $Cout$. Multibit addition is implemented by cascading multiple full adders, forming the basis for more complex adder structures.

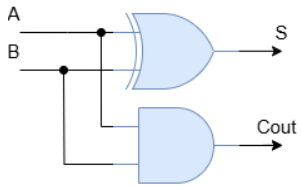

Fig. 1. Half Adder

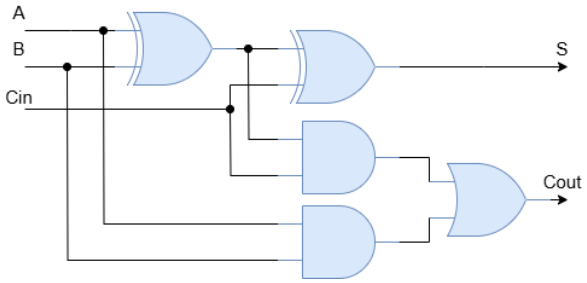

Fig. 2. Full Adder

A. Ripple Carry Adder

The Ripple Carry Adder (RCA) is the most straightforward digital adder architecture because it computes the addition the same way we learn to add in school. The RCA is built by cascading full adders so that the carry output from each stage becomes the carry input for the next. Although it is simple to design and area-efficient, the main problem of the RCA is its propagation delay. The carry signal must go through all full adders in sequence, making a delay that grows linearly with the number of bits, which limits its performance for high-speed applications. The RCA architecture is shown in Fig.3.

Despite its delay disadvantage, the RCA is widely used in low-power and area-constrained designs where speed is not critical. Its modularity and simple implementation make it an excellent choice for basic arithmetic operations in embedded systems. Because of its minimal hardware complexity, it is also commonly used in academic settings and basic processor designs to illustrate the concept of binary addition.

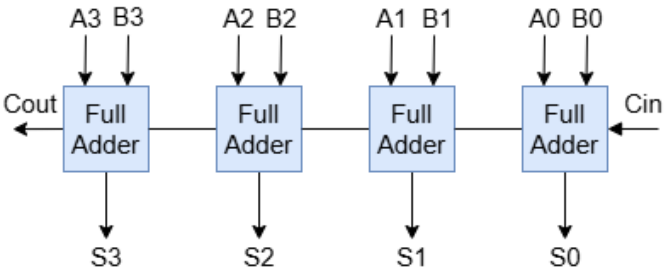

Fig. 3. 4-bit Ripple Carry Adder

B. Carry Select Adder

The Carry Select Adder (CSLA) computes the first block outputs the same way the RCA computes its outputs. The CSLA addresses the RCA delay problem by precomputing the sum and $Cout$ outputs for both possible Cin values ('0' and '1') in parallel for all the following blocks. Each block of bits, excluding the first one, is calculated twice, and then the correct result is selected using a multiplexer once the actual Cin is known. This approach significantly reduces the overall propagation delay compared to an RCA, especially for wider bit widths.

However, this speed improvement comes at the cost of increased area and power consumption due to duplication of the adder logic. CSLA is typically used in applications where performance is more important than area efficiency, such as in high-performance ALUs or DSPs. The design can be optimized further by employing variable block sizes to balance delay and resource usage.

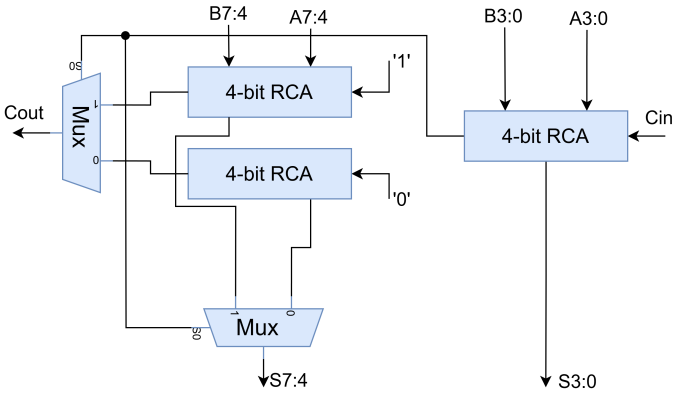

Fig. 4. 8-bit Carry Select Adder

C. Carry Increment Adder

The Carry Increment Adder (CIA) builds on the ripple carry concept by dividing the addition into blocks and incrementing

the result instead of computing two versions, as in CSLA. Each block computes the sum assuming a Cin of '0'. Therefore, if the actual Cin is '1', a simple incrementer is necessary to adjust the outputs. This strategy eliminates the need for duplicate adder logic, reducing area and power consumption.

CIA offers an efficient compromise between speed and resource usage. The CIA is a good alternative to accelerate the RCA while also consuming significantly less area and power than the CSLA, making it suitable for medium-performance arithmetic units. Its delay is lower than RCA because the carry propagation is shortened by block-level partitioning, and incrementers are typically faster and smaller than full adders.

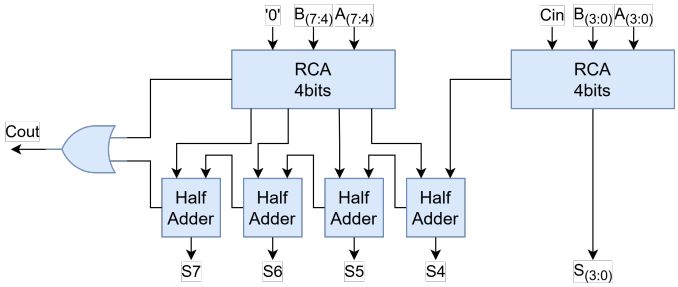

Fig. 5. 8-bit Carry Increment Adder

III. Carry Propagate/Generate Based Adders

As the bit-width of digital circuits increases, the propagation of carries becomes a critical problem in addition operations. Parallel prefix adders address this limitation by computing carries in parallel using a structured signal flow, enabling fast, logarithmic time addition. Unlike RCAs, where each bit waits for the previous carry, parallel prefix adders use a series of processing blocks to precompute and combine intermediate signals, significantly improving performance in high-speed processors.

The core idea of a parallel prefix adder relies on computing two key signals for each bit position i: the generate signal (G_i) and the propagate signal (P_i) [2]. The generate signal indicates that bit position i produces a carry regardless of the Cin, while the propagate signal indicates that bit position i passes an incoming carry to the next stage. Additionally, a kill/null signal (K_i) can be defined to indicate that neither a carry is generated nor propagated, occurring when both input bits are zero. This last signal is not used for the designs described in this paper. For input bits A_i and B_i, these signals are defined as follows:

$$G_i = A_i \cdot B_i \tag{1}$$

$$P_i = A_i \oplus B_i \tag{2}$$

$$K_i = \neg A_i \cdot \neg B_i \tag{3}$$

These individual signals are processed in a series of blocks, as shown in Fig. 7. The first block receives the inputs A_i, B_i, and Cin, computing G_i and P_i. The second block combines these signals to form group generate ($G^*_{i,j}$) and group propagate ($P^*_{i,j}$) signals, which determine the carry behavior for a range of bits from position i to j. The group generate signal indicates that the bit group produces a carry, while the group propagate signal indicates that the group passes an incoming carry. For a 4-bit group spanning bits 3 to 0, these signals are computed as:

$$G^*_{3,0} = G_3 + (P_3 \cdot G_2) + (P_3 \cdot P_2 \cdot G_1) + (P_3 \cdot P_2 \cdot P_1 \cdot G_0) \tag{4}$$

$$P^*_{3,0} = P_3 \cdot P_2 \cdot P_1 \cdot P_0 \tag{5}$$

It is possible to group sequential group generates and group propagates as follows:

$$G^*_{i,k} = G^*_{i,j} + (P^*_{i,j} \cdot G^*_{j+1,k}) \tag{6}$$

$$P^*_{i,k} = P^*_{i,j} \cdot P^*_{j+1,k} \tag{7}$$

The single generate and propagate signals can also be used as inputs, assuming that i and j or $j+1$ and k are the same and used as the index for G_i and P_i. Fig. 6 shows the merge of $G^*_{i,j}$ and $P^*_{i,j}$ with $G^*_{j+1,k}$ and $P^*_{j+1,k}$, resulting in $G^*_{i,k}$ and $P^*_{i,k}$.

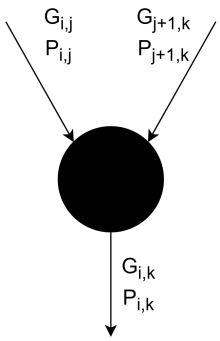

Fig. 6. Merge of two group generate and group propagate.

The third block uses these group signals to compute carries, and the final block produces the sum outputs (S_i) and the $Cout$. The carry C_i for bit position i is determined as:

$$C_i = G^*_{i,0} + (P^*_{i,0} \cdot Cin) \tag{8}$$

The sum output for each bit position i is then calculated as:

$$S_i = P_i \oplus C_{i-1} \tag{9}$$

where C_{i-1} is the carry into bit position i. It is important to notice that the signals $G^*_{i,0}$ and $P^*_{i,0}$ are required to be computed for all values of i, from $n-1$ to 1. This block-based signal flow, illustrated in Fig. 7, ensures efficient carry and sum computation, making parallel prefix adders essential for modern processor designs.

The concepts described in Section III are important to understand the parallel adder structures explained in the subsections below. The first one uses the concept of group propagate ($P^*_{i,j}$), while the other focuses on optimizing the generation of the group generate ($G^*_{i,j}$) and group propagate ($P^*_{i,j}$), the second block of Fig. 7.

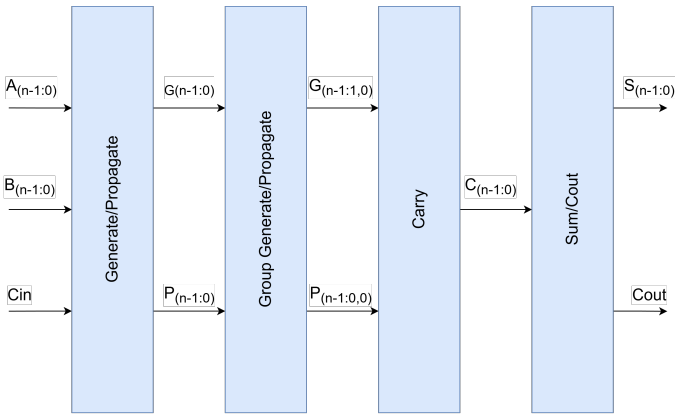

Fig. 7. Block diagram of a Parallel Prefix Adder, showing four blocks: the first computes generate and propagate signals from inputs A, B, and C_{in}; the second produces group generate and propagate signals; the third computes carries; and the fourth outputs the sum and carry-out.

A. Carry Skip/Bypass Adder

The Carry-Skip Adder (CSkA) improves the average-case delay of binary addition by allowing carries to bypass blocks of full adders when the group propagate signal $P^*_{i:j} = 1$. In such cases, the Cin of a block is directly propagated to the next block via a multiplexer, or a combinational logic, avoiding the internal carry chain, as shown in Fig.8. The internal chain can be seen as a group generate signal $G^*_{i:j}$, indicating if a $Cout$ is generated in the internal carry chain. The skipping mechanism reduces latency for input patterns where $P^*_{i:j} = 1$, compared to RCAs. However, in the worst case, when no skipping occurs, the delay is comparable to an RCA, with additional overhead from multiplexers. Performance depends on block size: smaller blocks increase skipping opportunities, as $P^*_{i:j} = 1$ is more likely, but requires more multiplexers, increasing hardware overhead. Larger blocks reduce overhead but get closer to RCA delays.

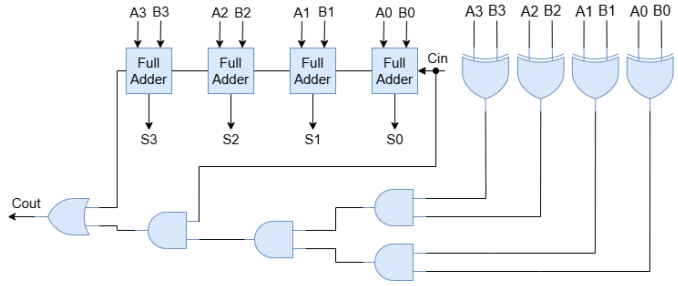

Fig. 8. 4-bit Carry Skip Adder. The Cin signal can be propagated to Cout faster if all propagate signals are '1'.

B. Kogge-Stone

The Kogge-Stone adder prefix tree [3],is one of the fastest parallel prefix adders, characterized by its fully parallel and regular tree structure, as depicted in Fig. 9. It computes the intermediate group generate and propagate signals in a way that minimizes logic depth, allowing carries to be computed very quickly. The prefix tree depth is $log_2(n)$, for an n-bit adder, making it extremely efficient in terms of speed. Each bit receives its carry from a path of minimal length, resulting in minimal worst-case delay.

However, this performance comes with significant trade-offs. The Kogge-Stone structure requires a large number of wiring tracks and logic cells due to the high number of prefix operations and fan-out buffers. This leads to increased area and routing congestion in layout, especially as the bit-width grows. Additionally, its high power consumption makes it less suitable for energy-efficient applications. Despite these drawbacks, Kogge-Stone is often used in high-performance systems like modern CPUs where speed is the top priority.

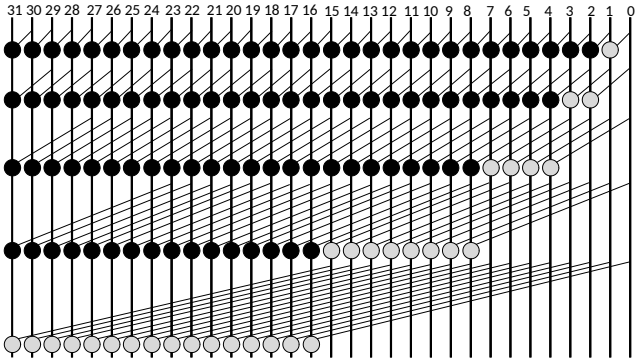

Fig. 9. 32-bits Kogge-Stone adder prefix tree. Each circle merges two group generate/propagate signals. The gray balls indicate that they are an important group ($G^*_{i,0}$ and $P^*_{i,0}$) and are going to be used for the next block.

C. Brent-Kung

The Brent-Kung adder prefix tree is designed with a focus on reducing area and wiring complexity while maintaining reasonable performance [4]. Its prefix tree is constructed using a balanced, binary structure that limits the number of prefix operations, using significantly fewer than Kogge-Stone. Although it requires more logic stages $2\log_2 n - 1$ to complete the carry computation, the reduced fan-out and fewer wires make it easier to implement in VLSI designs. The increased depth introduce more delay compared to Kogge-Stone, but this delay is often acceptable in applications where area and power efficiency are more important than performance. As a result, it is commonly used in processors and ASICs that require a balanced trade-off between delay, area, and power.

D. Sklansky

The Sklansky adder prefix tree constructs its prefix tree with an emphasis on minimizing the number of logic levels. Like Kogge-Stone, it achieves a logarithmic delay of $\log_2 n$, making it a good choice for performance. The tree is built in such a way that each bit computes its carry in the fewest possible stages, often making Sklansky one of the lowest-delay PPAs available in theory. However, this architecture suffers from high fan-out at the upper levels of the tree, especially near the root. A single node may drive many others, leading

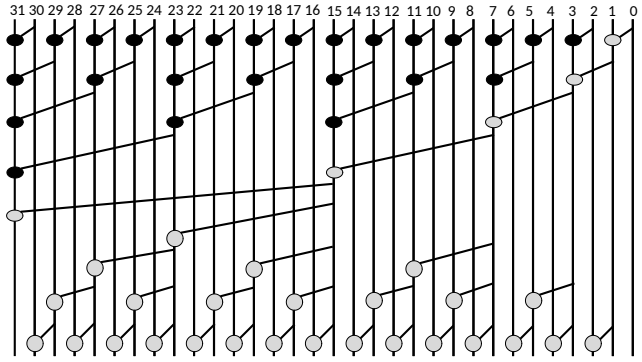

Fig. 10. 32-bits Brent-Kung adder prefix tree. Each circle merges two group generate/propagate signals. The gray balls indicate that they are an important group ($G^*_{i,0}$ and $P^*_{i,0}$) and are going to be used for the next block.

to increased loading, routing complexity, and timing issues during layout. This can reduce practical performance despite its theoretical efficiency. Therefore, Sklansky adders are less frequently used in physical implementations unless the fan-out problem is mitigated through buffering or other mitigation techniques.

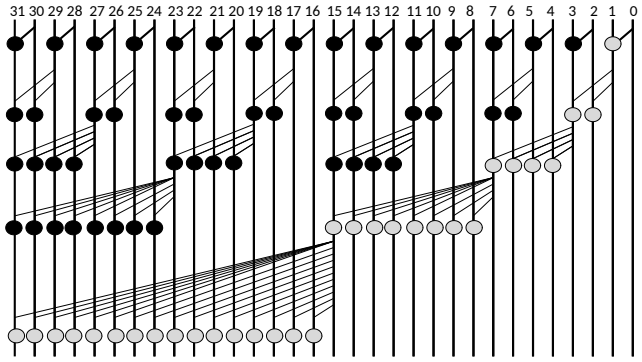

Fig. 11. 32-bits Sklansky adder prefix tree. Each circle merge two group generate/propagate signals. The gray balls indicate that they are an important group ($G^*_{i,0}$ and $P^*_{i,0}$) and are going to be used for the next block.

E. Ladner-Fischer

The Ladner-Fischer adder prefix tree attempts to balance logic depth, area, and fan-out in a systematic way and is shown in Fig.12. It builds the prefix tree by selectively computing group generate and propagate signals based on an optimized path rather than using fully populated or minimal structures. This results in a structure with moderate depth and reduced wiring complexity compared to Kogge-Stone.

One of the advantages of the Lander-Fischer adder is its adaptability it can be tuned to fit different design goals such as lower power or reduced fan-out [5]. Although not as fast as Kogge-Stone or as compact as Brent-Kung, it provides a balanced alternative where neither delay nor area dominates excessively. This makes it useful in custom design scenarios where a middle ground is required between extreme speed and minimal hardware.

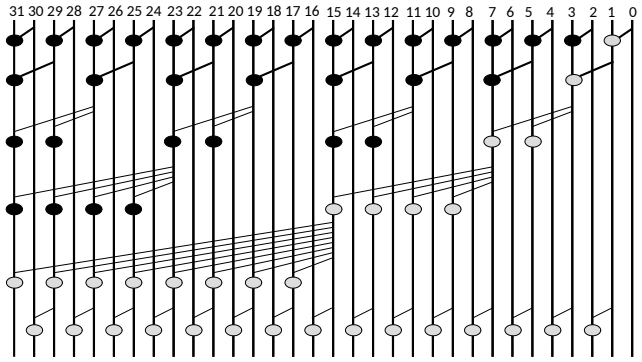

Fig. 12. 32-bits Ladner-Fischer adder prefix tree. Each circle merge two group generate/propagate signals. The gray balls indicate that they are an important group ($G^*_{i,0}$ and $P^*_{i,0}$) and are going to be used for the next block.

F. Han-Carlson

The Han-Carlson adder prefix tree combines the speed advantages of the Kogge-Stone adder with the area efficiency of the Brent-Kung adder, forming a hybrid architecture [6]. It uses a sparse Kogge-Stone prefix tree in the early stages to quickly propagate carries through major bit positions, and then switches to a Brent-Kung style reduction in later stages. This hybrid tree results in logarithmic delay while significantly reducing the number of logic cells and wiring resources compared to Kogge-Stone.

This adder is often chosen for practical designs because it balances speed, and area. The early Kogge-Stone portion ensures fast global carry computation, while the Brent-Kung stages limit fan-out and help manage layout complexity. The Han-Carlson adder is a popular choice in commercial processors and optimized datapaths, offering near-optimal performance without the high power and congestion costs of pure Kogge-Stone implementations.

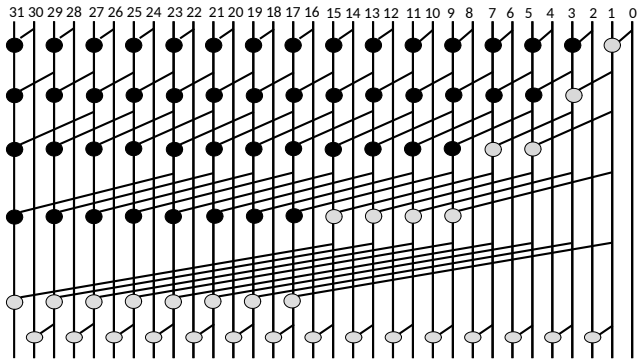

Fig. 13. 32-bits Han-Carlson adder prefix tree. Each circle merges two group generate/propagate signals. The gray balls indicate that they are an important group ($G^*_{i,0}$ and $P^*_{i,0}$) and are going to be used for the next block.

G. Carry Look Ahead Adder

The Carry Look Ahead Adder (CLA) achieves high-speed addition by computing carry signals in advance using generate and propagate logic, rather than waiting for carries to ripple through each bit. It uses Boolean logic to derive expressions for each carry output, enabling parallel computation and significantly reducing delay, especially in wider adders. The main block used for CLAs is the 4x3 group generate and propagate, shown in Fig. 14. It has four input pairs of generate and propagate signals, can be groups, and computes the three group generate and propagate. It needs to follow the rules explained before that the index needs to be sequential. Different blocks can be used for hybrid versions of the CLA, as the one developed in [7]

Due to its fast operation, the CLA is often used in high-performance computing systems where speed is critical. However, its complexity increases rapidly with the number of bits, requiring more hardware resources and intricate logic. To mitigate this, large CLAs are often implemented in hierarchical structures to balance speed and area. Despite the added complexity, CLA remains one of the fastest adder architectures. Fig. 15 shows a 16-bit CLA built using 4x3 group generate and propagate blocks. The max-delay goes through 3 of these blocks for the worst case.

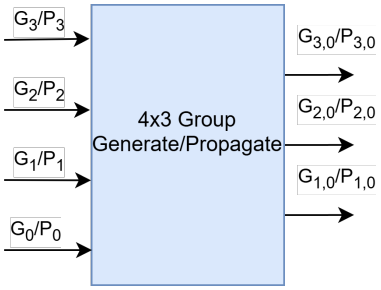

Fig. 14. 4x3 group generate and propagate. Combine the four pairs of (group) generate and propagate in inputs and generate three group generate and propagate.

IV. Methodology

The performances of different adder designs were evaluated using Cadence Genus for logical synthesis, and Cadence Innovus for physical synthesis. Each design was previously tested using Intel Questasim, executing multiple random sums, with and without Cin. The designs are fully combinational, so they do not have a clock. Therefore, the constraints were not tight, just fixing the loads of the outputs and the delay transition of the input. The only variable constraint was the set_max_delay, which limits the maximum delay acceptable from all inputs to all outputs. Varying this constraint, it is possible to do different analyses to discover how each design adapts to a tighter time constraint.

The Genus and Innovus commands were automated using Tool Command Language (Tcl) scripts. The Genus Tcl script begins by reading the necessary technology files and HDL sources, then elaborates on the target design. It reads the

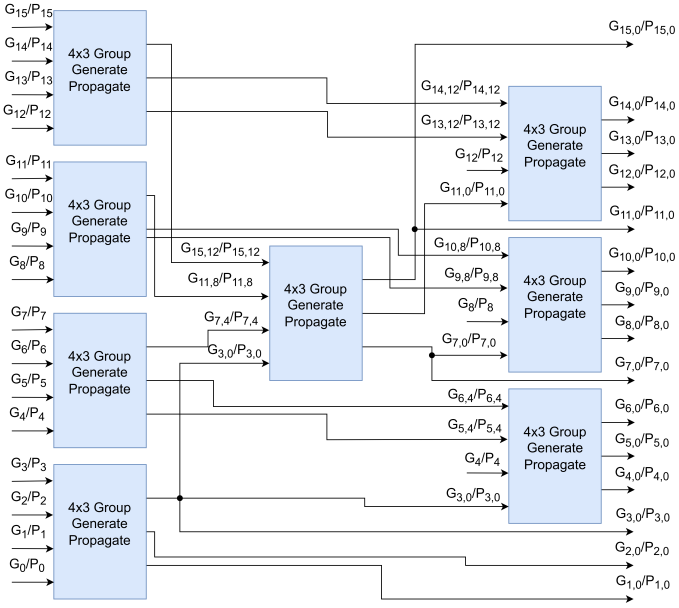

Fig. 15. 16-bits Carry Look Ahead adder prefix tree

constraint file, sets the synthesis effort to high, and disables the ungroup option to preserve the original design structure. It is possible to see in Fig. 16 that the blocks are separated from each other, preserving the adder structure. After that, it performs generic synthesis (syn_generic), technology mapping (syn_map), and final optimization (syn_opt). Finally, it generates timing, area, and power reports and exports the required output files.

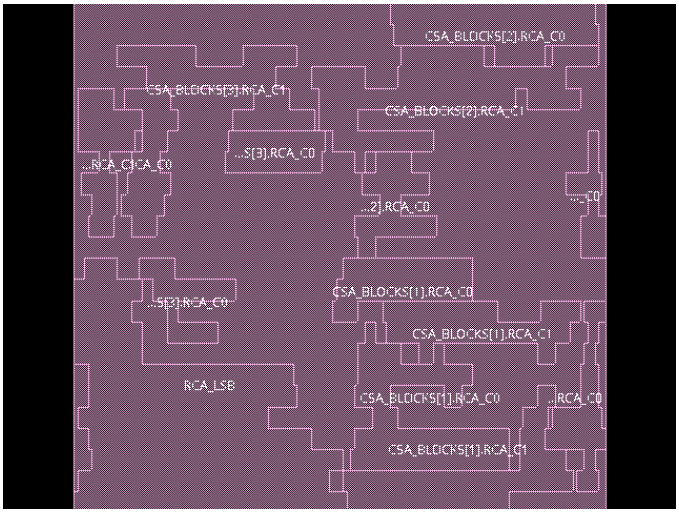

Fig. 16. 32-bits CSLA with four 8-bit blocks

The Innovus Tcl script begins by reading the necessary technology files and some files generated by Genus, like netlists and constraint files. It then adjusts the floorplan using the area obtained by Genus, and does the power planning, as the circuits are not very big, only the rings are necessary. Next, it places the standard cells and connects the inputs.

The clock tree synthesis is skipped as the designs are fully combinational, so the next step is routing. Lastly, it adds filler cells and generates the area, timing, power reports, and other necessary outputs.

A max delay constraint optimization algorithm is used to find the maximum frequency the design can work. A script to run this algorithm is important for some of the intended methodologies. A shell script realizes multiple syntheses to find the best time constraint for a desired design. First, it runs only Genus, doing a binary search to get the minimum delay constraint that obtains no-negative slack. Next, it runs both Genus and Innovus with the same time constraint, increasing it by the amount of time the Innovus report is below zero. The script finishes when both tools obtain no-negative slacks.

A maximum delay constraint optimization algorithm is used to determine the highest frequency for each design. A shell script automates this process and is essential for implementing several of the proposed methodologies. It performs multiple synthesis iterations to identify the most suitable timing constraint for a given design. The shell script also coordinates all proposed methodologies and automatically extracts area, power, and timing metrics from Genus and Innovus reports.

First, the script runs Cadence Genus alone, applying a binary search to find the minimum delay constraint that results in non-negative slack. Once this constraint is identified, the script executes both Genus and Innovus using the same delay constraint. If Innovus reports a negative slack, the constraint is adjusted by the same amount as the negative slack. This process continues until both tools achieve non-negative slack, indicating that the constraint is feasible for both syntheses. The algorithm is described by the fluxogram in Fig. 17.

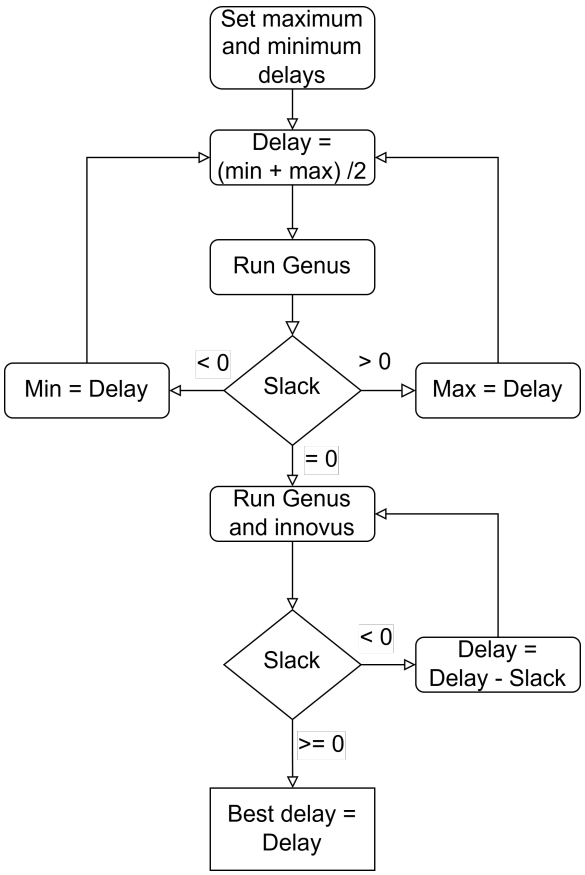

Fig. 17. Max Delay Constraint Optimization

A. Area Oriented Methodology

In this approach, a very relaxed timing constraint was set, allowing Genus and Innovus to focus primarily on minimizing power consumption and area. Since timing is not a limiting factor, the tools have greater flexibility to choose slower, more compact, and energy-efficient implementations. This methodology is particularly useful for understanding the power and area trade-offs of each adder architecture when speed is not critical.

B. Time Oriented Methodology

This method tries to determine the tightest achievable timing constraint for each adder architecture individually. The goal was to push each design to its maximum performance potential by minimizing its critical path delay, regardless of area or power implications. This allows for a fair comparison of the maximum attainable speed for each architecture under aggressive timing requirements. The algorithm described in Fig. 17 is applied here for all designs.

This method takes longer to execute but pushes every design to its maximum performance. Each design runs Genus around 8 times to find the first delay, and around 5 times both Genus and Innovus to find the best delay.

C. Reference Frequency Methodology

In this methodology, the time constraint of the 32-bit Ripple Carry Adder, typically the slowest adder architecture, was optimized first. The corresponding timing constraint was then used as a common constraint across all other adder designs. This allowed for a consistent and fair evaluation of how each architecture performs when subjected to the same real-time constraint, highlighting their relative efficiencies in area and power under equal timing pressure.

D. Behavioral Adder Synthesis

The last methodology plans to test how Genus optimizations work in a behavioral description. The HDL file with a $+$ signal is used to see how the area and power metrics adjust according to how tight the time constraint is. The maximum delay value for this test is found by the first methodology, using the path delay when the time constraint is loose. The minimum delay value is obtained using the algorithm presented in Fig. 17. Also, it will test the other adder designs with the minimum and maximum delays, but removing the ungroup attribute in the Genus Tcl. This test is done to see if Genus can identify our designs as an adder and treat it the same way it treats the behavioral design using the $+$ signal.

V. Results

In this study, we implemented and analyzed several 32-bit adder architectures using SystemVerilog. Each design was carefully described at the RTL to ensure functional accuracy and enable a fair comparison. The objective was to evaluate and compare their performance in terms of area, delay, and power metrics. To ensure consistency and reproducibility, all designs followed the same logical and physical synthesis flow.

The focus was on a variety of adder topologies, ranging from basic to optimized architectures, to better understand their trade-offs when targeting ASIC implementations. In total, 14 different designs were implemented, one for each architecture discussed in Sections II and III. The CSkA was implemented in two versions: one with blocks of size 4, and another with blocks of size 8. The CSLA has three different implementations, which vary in the number and size of blocks; these sizes are encoded in the design name, with each digit representing the size of one block. Lastly, we included a behavioral adder that uses the $+$ operator in its implementation.

The synthesis process was performed using Cadence Genus [8] for logical synthesis and Cadence Innovus [9] for physical synthesis. The RTLs were synthesized with the Skywater130 technology. All steps of the design flow were automated using custom shell scripts, ensuring a standardized methodology for all adder types. These scripts managed the complete flow, including RTL compilation, synthesis, placement, and metric extraction, minimizing human error and allowing the execution of multiple experiments. The final layout can be seen in Fig. 18.

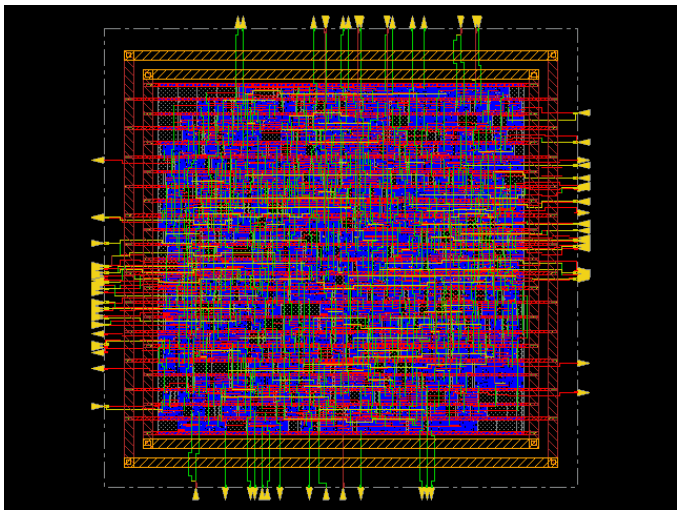

Fig. 18. Layout of the 32-bit Kogge-Stone Adder

The first methodology is focused on area and power optimization. Fig. 19 shows how much power the design needs and the frequency it can execute without any time constraint. The CIA architecture is the adder that uses the least power, beating the RCA and the behavioral designs. The CSkAs are slower, which is expected as they have to go through the whole ripple carry and some extra combinational logic in the worst case. The other adders are faster, but consume more power, with the Kogge-Stone being the fastest and consuming the most power. As the first methodology wanted to maximize area and power, the CIA looks like a good option to compete with the RCA with this constraint.

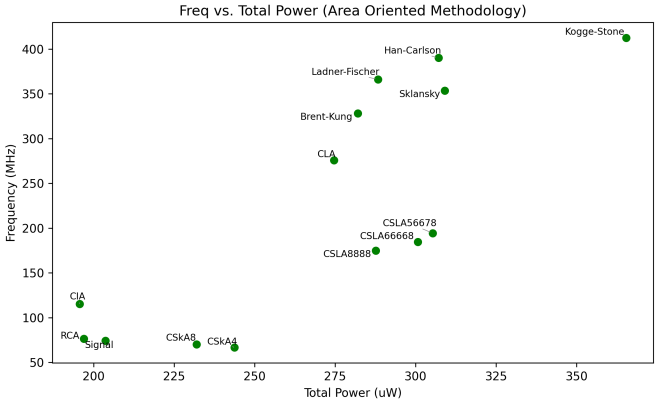

Fig. 19. Frequency x Total Power - Area Oriented Methodology (Genus)

The Innovus reports confirm these metrics, with the Kogge-Stone being the adder that consumes the most power and requires the most area. Interesting to note that the CIA is faster and has more than 50% extra area when compared with the RCA. However, the CIA keeps needing less power. Table I shows all the designs, with the frequency in MHz in the second column, total area in μm^2, and total power in μW.

TABLE I
Comparison of Adder Designs using area oriented methodology - Innovus

Design	Freq (MHz)	Total Area (μm^2)	Total Power(μW)
Kogge-Stone	358	3971	360
Han-Carlson	353	3003	292
Ladner-Fischer	349	2680	269
Sklansky	330	3003	289
Brent-Kung	312	2567	264
CLA	268	2455	256
CSLA56678	184	2122	283
CSLA66668	176	2082	278
CSLA8888	168	1982	268
CIA	115	1415	184
RCA	75	888	186
Signal	73	922	191
CSkA8	69	1489	209
CSkA4	66	1489	222

The second methodology aims to optimize the frequency of every adder design. Fig. 20 shows that the RCA uses the least area when the time constraint is strict, but it is one of the slowest architectures too, operating at 230MHz. The CSkAs are the slowest described designs. Genus manages to make the behavioral adder the fastest of all the adders, getting to a frequency of 719MHz.

This happens because Genus recognizes that it is an adder with multiple optimizations, mainly for adders. Another characteristic that gives the behavioral design an edge in this strict

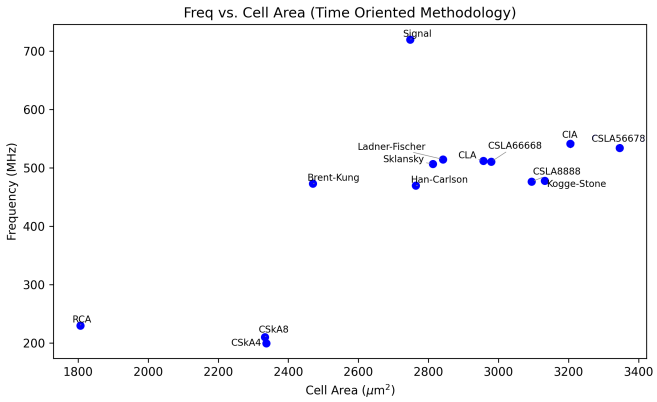

Fig. 20. Frequency x Cell Area - Time Oriented Methodology (Genus)

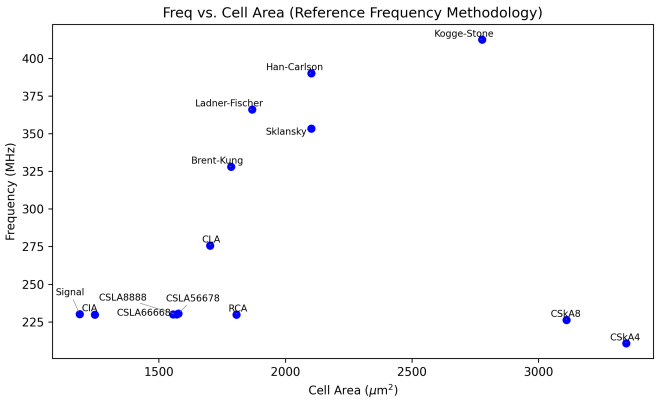

Fig. 22. Frequency x Cell Area - Reference Frequency Methodology (Genus)

constraint is the ungroup attribute, because it has complete freedom to optimize however is necessary, while the other adders have to keep the structure unchanged. The same trade-offs can be seen in Fig. 21, using Innovus. The objective of this method is to optimize all adders and find the fastest one. The behavioral design has a noticeable performance gap, but the Brent-Kung or the RCA can be an option if the area becomes a problem.

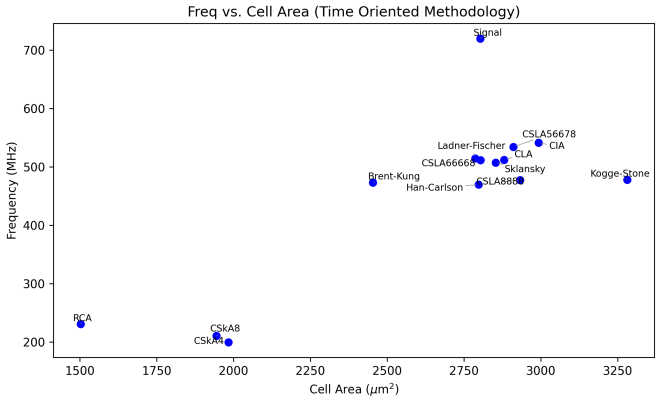

Fig. 21. Frequency x Cell Area - Time Oriented Methodology (Innovus)

The next methodology fixes a time constraint, the frequency of the optimized RCA, and compares it with all other designs under the same conditions. Fig. 22 shows that the CSkAs are the worst designs, both in area and frequency. The CIA and the behavioral are the two smallest designs, working with a similar frequency to the RCA, with the latter requiring more area. The parallel prefix adders have a high slack because their structures are not very flexible. The area of parallel prefix adders cannot be reduced with a smaller time constraint, because their structures do not support much optimization with the ungroup attribute disabled. The Kogge-Stone presents the biggest area and highest performance.

The behavioral adder was stressed with multiple time constraints, increasing the delay constraint from the minimum value obtained in the first methodology, 13.7 ns, to the maximum value obtained by the second method, 1.39 ns. The steps between each iteration are 0.1 ns, with the last iteration step being only 0.01 ns. The results obtained are shown in Fig. 23. The orange line shows the cell area for each iteration, and it is noticeable that it has small jumps in area at certain thresholds. The blue bars show the frequency metric for every iteration. It follows the area metric but has smoother transitions. The maximum frequency is 719 MHz, with 2748 μm^2 and consuming 415 μW. The gray bar represents the power metric, and varies less than the other two metrics, with the biggest changes happening in the last eight iterations.

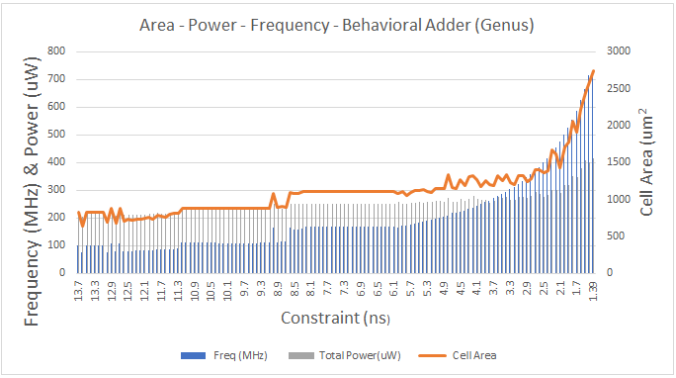

Fig. 23. Area - Power - Frequency - Behavioral Adder (Genus)

The last analysis was conducted by activating the ungroup attribute. All the other adders were synthesized with the maximum and minimum time constraints of the behavioral adder. The objective was to check if Genus would identify the design as an adder and do the same optimizations that happen with the behavioral design. Fig. 24 shows the frequency (MHz) and power (μW) comparison. The Ladner-Fischer obtains the highest frequency, but the power consumption is high, consuming less than the three CLSA designs only. The behavioral design did not perform well in this test, not being a good option in this analysis. The CIA consumes the least power again, with a decent performance, being a good option in this

situation.

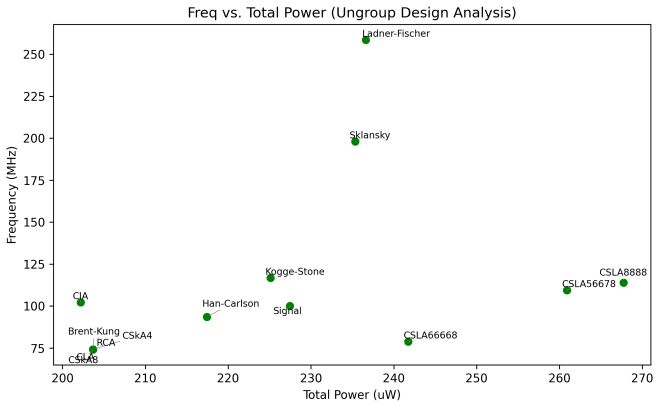

Fig. 24. Frequency x Total Power - Ungroup Design Analysis (Genus)

Finally, Fig. 25 shows the frequency (MHz) and total power (μW) comparison, but from Innovus this time. The results look different because all designs, besides the behavioral adder, ended up with negative slack, so even if the design is the slowest, Genus and Innovus did everything possible to optimize the design. The behavioral adder beats every other design in this analysis, but the CSLA which was not presenting good results, got close metrics. Especially the CSLA that uses four 8-bit blocks, that is 500 KHz slower, is 16 μm^2 bigger, and consumes 3 μW less than the behavioral adder.

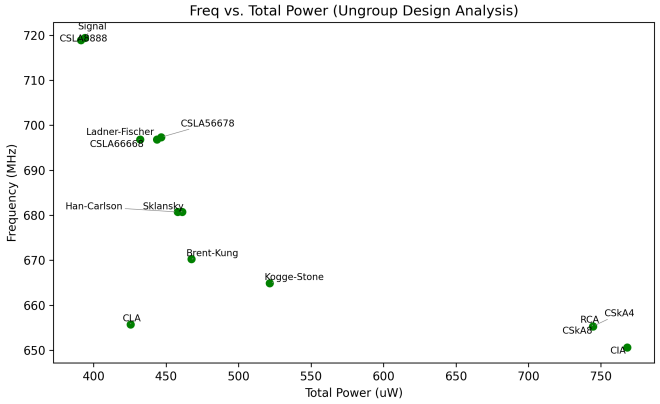

Fig. 25. Frequency x Total Power - Ungroup Design Analysis (Innovus)

VI. Conclusion

This work presented a comprehensive comparative analysis of various 32-bit adder architectures, including both classical and parallel prefix designs, synthesized using Cadence Genus and Innovus. By implementing different synthesis methodologies, area-oriented, time-oriented, reference-frequency-based, and behavioral, the study captured how each design behaves under distinct optimization goals.

The results demonstrate that no single architecture is optimal across all design scenarios. While the Ripple Carry Adder (RCA) offers the smallest area footprint, the Carry Increment Adder (CIA) balances area and power efficiently. Parallel prefix adders such as Kogge-Stone and Ladner-Fischer provide high performance but at the expense of increased power and routing complexity. The behavioral design using the '+' operator benefited from tool-level optimizations, highlighting the impact of synthesis attributes such as ungrouping on final performance. Important to notice that, even if the behavioral adder is one of the best for most constraints, it is beaten by different adder structures for different constraints. Also, the CLSA presented bad results throughout all methodologies and got the closest to behavioral design once the ungroup attribute was active.

These findings emphasize the importance of aligning adder architecture selection with specific design constraints, such as area, power, or timing, based on the intended application. The proposed flow and automated methodology also enable reproducible and fair comparisons of arithmetic blocks in ASIC design flows.

Future work will focus on expanding the design space to include hybrid adder architectures, which aim to combine the strengths of different adders (e.g., integrating carry-skip stages into parallel-prefix trees) for more optimized performance and area trade-offs. Another promising direction is the manual design and layout optimization of specific logic cells to overcome limitations imposed by standard-cell libraries and further push the efficiency of critical paths. These custom cells can be tuned to match the delay and power characteristics required by different adder topologies, offering another layer of control in the physical design process.

Acknowledgment

This study was financed in part by Softex, and by the Ministério de Ciência, Tecnologia e Inovação (MCTI).

References

[1] I. Koren, *Computer arithmetic algorithms*. AK Peters/CRC Press, 2018.
[2] A. Raju, R. Patnaik, R. K. Babu, and P. Mahato, "Parallel prefix adders—a comparative study for fastest response," in *2016 international conference on communication and electronics systems (ICCES)*. IEEE, 2016, pp. 1–6.
[3] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE transactions on computers*, vol. 100, no. 8, pp. 786–793, 1973.
[4] Brent and Kung, "A regular layout for parallel adders," *IEEE transactions on Computers*, vol. 100, no. 3, pp. 260–264, 1982.
[5] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *Journal of the ACM (JACM)*, vol. 27, no. 4, pp. 831–838, 1980.
[6] T. Han and D. A. Carlson, "Fast area-efficient vlsi adders," in *1987 IEEE 8th symposium on computer arithmetic (ARITH)*. IEEE, 1987, pp. 49–56.
[7] Z. Wang, G. A. Jullien, W. C. Miller, J. Wang, and S. S. Bizzan, "Fast adders using enhanced multiple-output domino logic," *IEEE Journal of Solid-State Circuits*, vol. 32, no. 2, pp. 206–214, 1997.
[8] Cadence, "Genus User Guide: Product version 23.1," 2024.
[9] ——, "Innovus User Guide: Product version 23.14," 2025.