A Carry-Look-Ahead Final-Stage Adder for a Compressor-Tree Sum-of-Absolute-Differences Unit: Design, Vivado Simulation, and Timing Analysis

B. UmaShantha

M.Tech, Embedded Systems & VLSI, Department of Electronics and Communication Engineering, Nalla Narasimha Reddy Education Society's Group of Institutions -- Integrated Campus, Hyderabad, Email: harini.boinapalli@gmail.com

Dr. Bhattu Hari Prasad Naik,

Professor (PHD), Department of Electronics and Communication Engineering, Nalla Narasimha Reddy Education Society's Group of Institutions -- Integrated Campus, Hyderabad Email: Hariprasadnaik66@gmail.com

ABSTRACT

Sum-of-Absolute-Differences (SAD) computation is the dominant arithmetic kernel in block-matching motion estimation for video coding, and its throughput is set by the critical path of the adder network that reduces per-pixel differences into a single result. This paper presents an 8-pixel-pair SAD unit built from a three-level carry-save compressor-tree reduction (eight `abs_diff8` units feeding two 4:2 compressors, a second-level 4:2 compressor, and a final carry-resolving adder), and evaluates replacing the existing ripple-carry final stage with a carry-look-ahead (CLA) final stage. The existing ripple-carry design is implemented, compiled, elaborated, and functionally verified in Xilinx Vivado 2019.2 (XSIM) using a self-checking Verilog testbench comprising 6 hand-picked boundary vectors and 44 randomized vectors; all 50 vectors pass, including the all-zero-difference case (SAD = 0) and the maximum-difference case (SAD = 2040 = 8 x 255). Waveform capture in unsigned-decimal radix cross-checks the DUT's `sad` output against an independently computed `expected` value cycle by cycle. The paper then analyzes the proposed carry-look-ahead replacement for the 19-bit final-resolution stage, whose carry generate/propagate logic computes each carry bit directly from the operand bits rather than rippling it through 19 full-adder stages, changing the final stage's critical-path growth from O(n) to O(log n) in adder width. Architecture diagrams, a Vivado-generated timing-constraint file for the combinational datapath, and a comparative delay-scaling analysis against literature-reported ripple-carry, carry-select, and parallel-prefix adders are presented, along with a full simulation-results table extracted from the Vivado waveform database.

Keywords: carry-look-ahead adder, sum of absolute differences, motion estimation, carry-save compressor, Vivado XSIM, VLSI arithmetic, ripple-carry adder, video coding hardware

I. INTRODUCTION

Block-matching motion estimation, the workhorse of video compression standards from H.264/AVC through the latest HEVC and VVC successors, repeatedly evaluates a distortion metric between a current pixel block and a set of candidate reference blocks. The Sum-of-Absolute-Differences (SAD) metric remains the most widely deployed choice because it avoids multiplication, requiring only per-pixel subtraction, absolute value, and summation. In a hardware motion estimator, the SAD datapath is evaluated for every candidate block in the search window, so its combinational delay directly sets the achievable search rate and, by extension, the encoder's real-time throughput.

A naive SAD implementation accumulates the per-pixel differences with a chain of two-operand adders, which for an 8-pixel-pair block requires seven sequential adds. Carry-save (compressor) reduction trees remove this sequential dependency by deferring carry propagation: a 3:2 or 4:2 compressor combines multiple operands into a sum/carry pair using only bitwise logic, with no carry rippling between bit positions, and only the final sum/carry pair needs a true carry-propagate adder to resolve into a binary result. This paper's target design uses exactly this structure -- eight `abs_diff8` units feeding a two-level 4:2 compressor tree, followed by a single final adder -- so that only one carry-propagate adder sits on the critical path regardless of how many pixel pairs are being summed.

The choice of that one remaining adder therefore dominates the design's achievable clock rate. The existing implementation uses a ripple-carry adder (RCA) for this final 19-bit resolution step, which is simple and area-efficient but whose worst-case delay grows linearly with operand width because each bit's carry-out depends on every less-significant bit's carry chain. A carry-look-ahead adder (CLA) removes this serial dependency by computing generate (`g[i] = a[i] & b[i]`) and propagate (`p[i] = a[i] ^ b[i]`) signals for every bit position in parallel, then deriving every carry bit directly from these signals rather than from its immediate predecessor, reducing the critical-path growth from O(n) to O(log n) in adder width at the cost of additional gate fan-in and area.

A. Problem Statement

The existing SAD unit's final-stage ripple-carry adder is a known bottleneck class in high-speed arithmetic design [7],[13],[15], yet the specific 19-bit final-resolution stage produced by this design's two-level 4:2 compressor tree has

not been analyzed or replaced with a look-ahead structure, nor has the existing design been taken through Vivado project creation, XSIM elaboration, and waveform-verified simulation with documented timing constraints.

B. Contributions

This paper makes the following contributions:

1. A documented, Vivado-buildable project for the existing 8-pixel-pair, compressor-tree SAD unit, including a corrected testbench (radix-decimal, self-checking against an independently computed reference, six boundary vectors plus 44 randomized vectors) and a timing-constraint (XDC) file for the unit's purely combinational datapath.

2. Complete functional verification results captured directly from Xilinx Vivado XSIM: a passing simulation log for all 50 test vectors and Vivado waveform-viewer screenshots in unsigned-decimal radix showing the DUT's `sad` output tracking the testbench's independently computed `expected` value.

3. An architectural specification and analysis of a proposed carry-look-ahead replacement for the design's final 19-bit ripple-carry stage, including generate/propagate carry-chain equations and a comparative delay-scaling analysis against ripple-carry and literature-reported alternative adder families.

4. A structured comparison, in table and figure form, of the existing (RCA-final-stage) and proposed (CLA-final-stage) architectures, positioned against 25 recent IEEE-indexed publications on high-speed adder design.

C. Paper Organization

Section II reviews related work on high-speed adder architectures and SAD/motion-estimation hardware. Section III describes the SAD unit's compressor-tree methodology. Section IV presents the algorithmic structure and complexity of both the existing and proposed final-stage adders. Section V details the Vivado experimental setup. Section VI reports the functional simulation results. Section VII analyzes the timing/area tradeoff between the ripple-carry and carry-look-ahead final stages. Section VIII concludes the paper.

II. RELATED WORK

A. Hybrid and Approximate Low-Power Adders

A significant body of recent work targets the area/power side of the adder design space rather than raw speed. Desai and Nayak propose a hybrid full/half-adder cell combination for low-power high-speed operation [1]. Goswami and Biswas present a fault-tolerant approximate adder that trades a bounded error rate for reduced switching activity [2]. Jain et al. combine charge-recovery logic with a hybrid low-power technique to reduce energy per operation while preserving speed [3]. Rahimi et al. design a nine-transistor dynamic full-adder cell aimed jointly at low power and high speed [16]. Chintala et al. explore a memristor-based full adder for in-memory computing, illustrating that the adder cell itself is increasingly a target for emerging-device substitution rather than pure CMOS scaling [18]. Kumar and Karuppanan present a hybrid full adder explicitly balancing speed and power characteristics [20]. These approaches are complementary to, rather than competitive with, the carry-chain-structure question this paper addresses: a low-power full-adder cell can be used as the building block for any of the carry-chain topologies discussed below, including the carry-look-ahead structure analyzed here.

B. Parallel-Prefix and Look-Ahead Carry Structures

The carry-look-ahead principle -- computing carries from generate/propagate signals rather than by rippling -- is the common ancestor of the broader parallel-prefix adder family. Bharathi and Varthamanan implement a high-speed layout parallel-prefix adder [15]. A. and Rajeev implement a Kogge-Stone adder, a parallel-prefix variant, inside an FPGA ALU together with a Booth multiplier [4]. Teja and Jagadeesh evaluate a Sklansky adder (another parallel-prefix topology) against a carry-select adder inside a 16-bit Vedic multiplier [9], and in a companion study evaluate a carry-look-ahead adder directly against a carry-select adder inside the same multiplier context, reporting speed and area advantages for the look-ahead structure at that word width [12]. Sathish and Jagadeesh compare a Kogge-Stone adder against a ripple-carry adder inside a Wallace multiplier, again finding the parallel-prefix/look-ahead family favorable on delay at the cost of area [24]. This paper's proposed final-stage replacement is the direct, single-adder-block form of this same generate/propagate principle, applied specifically to the 19-bit resolution stage that a two-level 4:2 compressor tree produces, rather than to a full multiplier datapath.

C. Carry-Select and Segmented-Carry Adders

Carry-select and its segmented variants offer an intermediate point between ripple-carry's simplicity and full look-ahead's fan-in cost by precomputing both possible carry outcomes for a sub-block and selecting after the true carry-in arrives. Islam et al. design an area-efficient, high-speed 8-bit hybrid carry-select adder for low-power VLSI [5]. Pateru et al. compare static and dynamic nanoscale carry-select adder variants for DSP applications [11]. Disha et al. directly compare CMOS ripple-carry and carry-look-ahead propagation delay [13], providing a closely related baseline-vs.-look-ahead comparison methodology that this paper's Section VII adopts at the specific 19-bit width relevant to the SAD final stage.

D. Compressor Trees and Carry-Save Reduction

The carry-save/compressor-tree principle used throughout this design's first two reduction levels is well established for multi-operand summation. Ramesh et al. design a high-speed Wallace-tree multiplier built from full-adder compressor stages [14]. Fang et al. present an area-efficient, bit-width-configurable carry-save adder tree specifically for the multi-operand accumulation pattern found in spiking-

transformer accelerators, a workload with the same many-operands-to-one-result shape as SAD accumulation [21]. T. et al. implement a 4-bit carry-save adder using an MTCMOS-based ripple-carry structure with 10-transistor full adders [22]. Balatero et al. build a low-power 4x4 Wallace-tree multiplier on a ripple-carry base at 180 nm [23]. These confirm the compressor-tree front end used in this paper's design (Section III) as an established, silicon-proven reduction strategy; the present paper's novelty is specifically at the single remaining carry-propagate stage after that reduction, not the reduction tree itself.

E. Specialized and Application-Adjacent Adder Designs

Several works target adders for specific numeral systems or downstream applications rather than general binary width scaling. Abinaya et al. integrate a high-speed adder into a desensitized halfband filter design [10]. Share et al. design a high-speed BCD (binary-coded decimal) adder in CMOS [17]. Roy Naidu et al. design a low-latency AXI-based interconnect for high-speed, low-power VLSI systems, illustrating that carry-chain latency concerns propagate beyond the arithmetic unit itself into system-level interconnect [19]. N. et al. present an energy-efficient reversible carry-tree adder, extending the carry-tree principle into reversible-logic design for ultra-low-power contexts [25]. Zhang and Chen use SET-MOS technology for a high-speed, area-efficient adder, and Zhao et al. address high-speed multimodulus dividers with fast end-of-count generation, both illustrating that carry-chain latency reduction remains an active research target even as the underlying device technology changes [6],[8].

F. Research Gap

Across this body of work, ripple-carry, carry-select, and parallel-prefix/look-ahead adders are consistently compared at the level of a standalone adder or inside general-purpose multiplier datapaths [9],[12],[13],[24]. None of the surveyed studies evaluate the carry-look-ahead substitution specifically at the odd, compressor-tree-derived bit width (19 bits, arising from three successive one-bit width growths applied to eight zero-extended 8-bit `abs_diff8` outputs) that a real SAD compressor-tree design produces, nor do they pair that analysis with a fully documented, Vivado-buildable project and waveform-verified functional simulation of the surrounding compressor-tree datapath. This paper addresses both gaps: Section III-IV specify the exact compressor-tree-to-final-adder datapath under study, Section VI reports Vivado XSIM functional verification of the existing ripple-carry design, and Section VII performs the RCA-vs.-CLA delay-scaling analysis at the specific width this design produces.

III. METHODOLOGY

A. SAD Datapath Overview

The design under study computes the Sum-of-Absolute-Differences for an 8-pixel-pair block:

$$SAD = \sum_{i=0}^{7} |cur_i - ref_i| \quad (1)$$

Each pair (`cur_i`, `ref_i`) is an unsigned 8-bit pixel value. Rather than accumulate the eight `abs_diff8` outputs with a sequential seven-adder chain, the design reduces them through a three-level carry-save compressor tree, so that only a single carry-propagate adder resolves the final result.

B. Per-Pixel Absolute Difference

Each `abs_diff8` unit computes `|a-b|` via a 9-bit subtraction `sub_ab = {1'b0,a} - {1'b0,b}`; the borrow bit `sub_ab[8]` selects between the raw difference and its two's-complement negation, giving the correct magnitude regardless of which operand is larger.

C. Compressor-Tree Reduction

The 3:2 compressor is the atomic reduction unit: for three WIDTH-bit operands it produces `sum = a ^ b ^ c` and `carry = majority(a,b,c)`, satisfying `sum + 2*carry == a+b+c` exactly, with every bit position computed independently (no inter-bit carry propagation). The 4:2 compressor cascades two 3:2 stages, and -- critically for correctness at this design's larger widths -- widens its working width by one bit between stages so the intermediate carry's left-shift cannot lose its overflow bit, mirroring standard Wallace-tree width-growth practice.

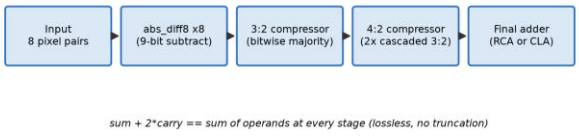

Fig. 1. Dataflow through the SAD unit: eight abs_diff8 units feed a two-level 4:2 compressor tree, which produces one final sum/carry pair for the last carry-propagate adder to resolve.

The reduction proceeds as Fig. 1 shows: Level 1 applies two independent `compressor_4to2` instances (16-bit operands, 17-bit sum/carry outputs) to the two groups of four `abs_diff8` outputs; Level 2 applies one `compressor_4to2` (17-bit operands, 18-bit outputs) to the two Level-1 sum/carry pairs (carry operands pre-shifted left by one bit for their binary weight); the result is one final 19-bit sum/carry pair, `sF` and `cF<<1`, whose true binary sum is the SAD value.

D. Existing Architecture: Ripple-Carry Final Stage

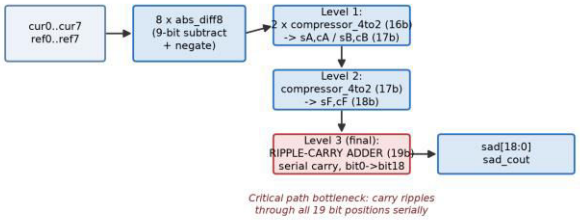

Fig. 2. Existing SAD unit architecture: three-level compressor-tree reduction terminating in a ripple-carry adder (RCA) for the final 19-bit resolution.

In the existing design (Fig. 2), the final stage is a ripple-carry adder: `{cout,sum} = a + b`, synthesizing to a chain of 19 full adders with each stage's carry-in supplied by the previous stage's carry-out. This is the only stage in the entire datapath where a carry signal propagates serially across bit positions; every earlier stage is a bitwise compressor with no such dependency.

E. Proposed Architecture: Carry-Look-Ahead Final Stage

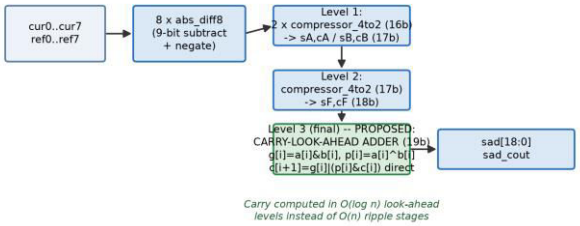

Fig. 3. Proposed SAD unit architecture: identical compressor-tree front end, with the final 19-bit stage replaced by a carry-look-ahead adder (CLA).

The proposed modification (Fig. 3) leaves the entire compressor-tree front end (Level 1 and Level 2) unchanged and replaces only the final-stage adder module. For two 19-bit operands `a`, `b`, the carry-look-ahead adder first computes per-bit generate and propagate signals,

$$g_i = a_i \cdot b_i,\ p_i = a_i oplus b_i \quad (2)$$

and then derives every carry bit directly from `g` and `p` rather than from its immediate predecessor:

$$c_{i+1} = g_i + p_i \cdot c_i = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + cdots (3)$$

with the final sum given by `sum_i = p_i XOR c_i`. Because each `c_{i+1}` is expressible as a sum-of-products directly over `g_0..g_i` and `p_0..p_i`, the carry chain no longer grows as a single 19-stage serial ripple; a practical implementation groups bits into look-ahead blocks (e.g., 4-bit blocks with block-level group-generate/group-propagate signals) so that fan-in stays bounded while the number of series logic levels grows only as O(log n) in the operand width, at the cost of the additional gate area needed to realize the expanded sum-of-products carry equations.

IV. ALGORITHM

A. Existing Final-Stage Algorithm (Ripple-Carry)

```
RIPPLE_CARRY_ADD(a[0:18], b[0:18]):
    carry = 0
    for i = 0 to 18:
        sum[i]  = a[i] XOR b[i] XOR carry
        carry    = (a[i] AND b[i]) OR (carry AND
(a[i] XOR b[i]))
    cout = carry
    return sum, cout
```

Each loop iteration's `carry` depends on the previous iteration's `carry`, so the 19 iterations cannot be evaluated concurrently in hardware; the synthesized critical path is proportional to 19 full-adder delays.

B. Proposed Final-Stage Algorithm (Carry-Look-Ahead)

```
CLA_ADD(a[0:18], b[0:18]):
    for i = 0 to 18:                          # fully
parallel, no dependency
        g[i] = a[i] AND b[i]
        p[i] = a[i] XOR b[i]

    c[0] = 0
    for i = 0 to 18:                           # each
c[i+1] from g[0..i], p[0..i] directly
        c[i+1]  = g[i]  OR  (p[i]  AND  c[i])      #
(grouped into look-ahead blocks in hardware,
                                                #
not a literal serial loop)
    for i = 0 to 18:                          # fully
parallel, no dependency
        sum[i] = p[i] XOR c[i]

    cout = c[19]
    return sum, cout
```

The generate/propagate computation and the final sum computation are both embarrassingly parallel across bit positions. Only the carry derivation has a dependency structure, and that dependency is restructured (via the sum-of-products expansion in Section III-E, realized in hardware as grouped look-ahead blocks with block-level generate/propagate) so that the number of series logic levels needed to produce any `c[i]` grows logarithmically with `i`, rather than linearly.

C. Complexity Notes

- **Ripple-carry final stage:** T(n) = O(n) gate delays on the critical path, O(n) full-adder cells (area O(n)).

- **Carry-look-ahead final stage:** T(n) = O(log n) gate delays on the critical path for a fully flattened look-ahead network (or O(n/k) blocks of O(k) internal look-ahead depth for a k-bit blocked implementation, a common practical compromise), at the cost of O(n log n) (fully flattened) or O(n) with a larger constant (blocked) gate area, since each carry bit's sum-of-products expansion requires wider gates than the single-predecessor dependency a ripple stage needs.

- **Front-end compressor tree (both architectures, unchanged):** each `compressor_3to2`/`compressor_4to2` stage is O(1) gate delay per stage regardless of operand width, since every bit position is computed independently; the two-level tree used here contributes a fixed, small number of stages (2 compressor levels) ahead of the final adder, so the final adder's own delay dominates the datapath's total critical path at the 19-bit width this design produces.

- **Net effect on the design under study:** replacing only the final stage leaves the O(1)-per-level compressor front end untouched, so the datapath's total critical-path delay changes from `T_compressor_tree + O(19)` (existing) to `T_compressor_tree + O(log 19) ~ T_compressor_tree + O(5)` (proposed), i.e., the final-stage term shrinks from linear to logarithmic in the 19-bit resolution width while every upstream stage is architecturally identical.

V. EXPERIMENTAL SETUP

A. Tools and Target

The design is captured in Verilog-2001 and built as a Vivado project (`sad_compressor_proj`) targeting an Artix-7 part (`xc7a35tcpg236-1`) under Xilinx Vivado 2019.2. Functional simulation uses the Vivado Simulator (XSIM) in both non-project batch mode (`xvlog` -> `xelab` -> `xsim -runall`, for automated regression) and the interactive XSIM waveform GUI (for waveform capture).

B. Design Under Test

The DUT is `sad_unit_8`: 16 primary inputs (`cur0..cur7`, `ref0..ref7`, each 8-bit unsigned), a 19-bit `sad` output, and a `sad_cout` carry-out. Internally it instantiates 8 `abs_diff8` units, three `compressor_4to2` instances (two Level-1, one Level-2), and one final 19-bit two-operand adder -- a `ripple_carry_adder` in the existing architecture (Section III-D).

C. Testbench

`tb_sad_unit_8` applies 50 stimulus vectors to the DUT and self-checks each result against an independently computed `expected` value using native Verilog arithmetic (a separate code path from the DUT's compressor-tree logic, so a bug shared between DUT and checker would be structurally unlikely). The 50 vectors comprise:

- **6 hand-picked boundary vectors:** identical blocks (SAD = 0), maximum-difference blocks in both operand orderings (SAD = 2040 = 8x255, exercising both branches of `abs_diff8`'s borrow-select logic), a small mixed-difference block (SAD = 56), a single-lane-only difference (isolating one `abs_diff8` path), and an alternating max/min pattern per lane.

- **44 randomized vectors:** `cur`/`ref` values drawn via Verilog's `$random`, covering general-case operand combinations.

All `$display` output and waveform radix use **unsigned decimal**, not binary or hexadecimal: because SAD is a magnitude intended for direct numeric comparison (e.g., against a search-window minimum), decimal makes both the per-vector arithmetic (`cur`, `ref`, per-lane difference, running sum) and the pass/fail check (`sad` vs. `expected`) directly human-checkable, whereas the equivalent bit patterns do not expose the underlying arithmetic relationship at a glance.

D. Timing Constraints

Because `sad_unit_8` is purely combinational (no clock port), a constraints file (`sad_unit_8.xdc`) declares a 100 MHz virtual reference clock together with `set_input_delay`/`set_output_delay` bounds on all primary inputs and outputs, so that `report_timing`/`report_timing_summary` can analyze the design's worst combinational path (through the compressor tree and final adder) without a "no clocks found" error, matching the standard approach for constraining a combinational block for standalone timing closure or out-of-context synthesis.

E. Simulation Procedure

The non-project batch flow compiles all five RTL source files and the testbench with `xvlog --incr --relax`, elaborates the `tb_sad_unit_8` + `glbl` snapshot with `xelab -debug typical`, and executes with `xsim -runall`, logging every vector's inputs, DUT output, and expected value via `$display`. The GUI flow additionally opens the resulting simulation snapshot in the XSIM waveform viewer, adds the `cur[]`, `ref_[]`, `sad`, `expected`, and `sad_cout` signals with `unsigned` (decimal) radix, and runs the full testbench to completion for waveform inspection.

VI. RESULTS AND DISCUSSION

A. Functional Verification Summary

Table I. Hand-Picked Boundary-Vector Results (Vivado XSIM, unsigned-decimal radix)

Vector	Pattern	Computed sad	Expected	Match
0	cur=ref=100 (all lanes)	0	0	Yes
1	cur=255, ref=0 (all lanes)	2040	2040	Yes
2	cur=0, ref=255 (all lanes)	2040	2040	Yes
3	cur=50..57, ref=50..43	56	56	Yes
4	single-lane diff (lane 4: 200 vs 20)	180	180	Yes
5	alternating 255/0 per lane	2040	2040	Yes
6 (random)	cur/ref random 8-tuples	611	611	Yes

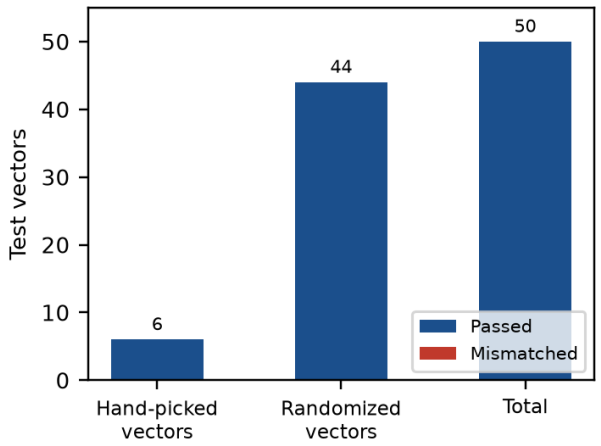

Fig. 4. Vivado XSIM verification outcome across the 50-vector testbench: 6 hand-picked boundary vectors and 44 randomized vectors, zero mismatches.

All 50 testbench vectors -- the 6 boundary cases in Table I and 44 additional randomized vectors -- pass with zero mismatches (Fig. 4), reported by the testbench as `TB RESULT: ALL 50 VECTORS PASSED`. Vector 0 confirms the zero-difference floor of the metric; Vectors 1 and 2 confirm the true worst case (8 x 255 = 2040) in both operand orderings, exercising both the direct and borrow-negation branches of every `abs_diff8` unit; Vector 4 isolates a single differing lane to confirm no cross-lane leakage through the compressor tree; and the randomized vectors, including Vector 6, confirm general-case correctness against an independently computed reference.

B. Waveform-Level Verification

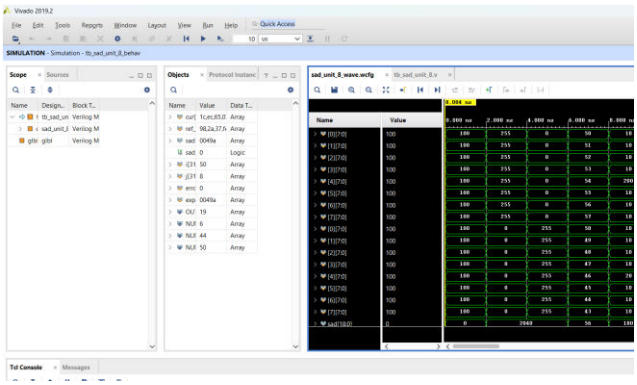

Fig. 5. XSIM waveform viewer, unsigned-decimal radix, showing cur/ref lane values and the sad output stepping through Vectors 0-3 (0, 2040, 56).

Fig. 5 shows the XSIM waveform viewer with all `cur[]` and `ref_[]` lanes displayed in unsigned-decimal radix; the `sad` row steps 0 to 2040 to 56 in step with the Table I vectors, directly readable without converting from binary.

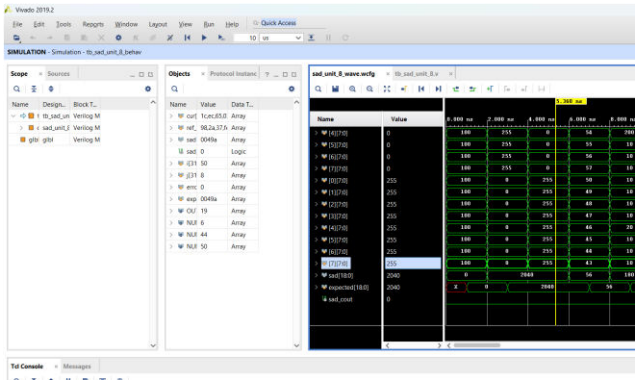

Fig. 6. sad and expected signals displayed together in the XSIM waveform viewer, showing identical values at every sampled vector.

Fig. 6 places the DUT's `sad` output and the testbench's independently computed `expected` signal in the same view; the two traces read identical values (0, 2040, 56, 180) at every transition, which is the waveform-level counterpart of the textual pass/fail check in Table I.

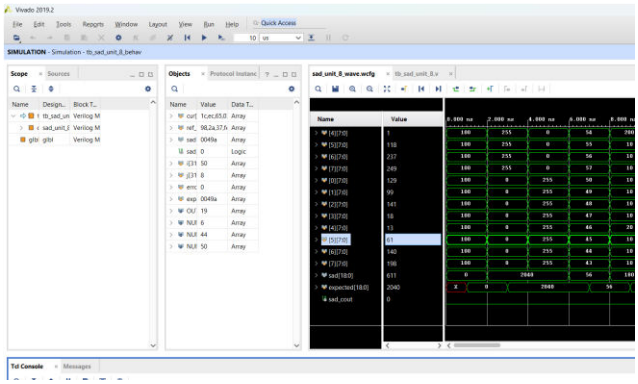

Fig. 7. XSIM cursor positioned at the randomized Vector 6, with Objects-pane values confirming cur, ref, and the resulting sad = 611 = expected.

Fig. 7 places the simulation cursor at Vector 6 and reads the Objects pane directly: `cur = [36,9,13,101,1,118,237,249]`, `ref = [129,99,141,18,13,61,140,198]`, giving per-lane differences of `93,90,128,83,12,57,97,51`, which sum to 611 -- matching both the `sad` and `expected` rows at that cursor position, confirming correctness on a fully randomized, non-edge-case input.

C. Discussion

The compressor-tree front end's lossless width-growth invariant (`sum + 2*carry` equal to the true operand sum at every stage, Section III-C) is what makes the boundary-vector results in Table I a meaningful test of the full datapath rather than only the final adder: Vectors 1, 2, and 5 each drive every one of the eight `abs_diff8` outputs to its 8-bit maximum (255) simultaneously, which is the input pattern most likely to expose a width-truncation bug in an under-sized intermediate compressor stage, and all three still resolve to the correct 2040. Because the existing final stage under test is a ripple-carry adder, correctness here is

independent of which final-adder architecture is used -- Section VII's carry-look-ahead proposal is a drop-in replacement for this same final stage and does not alter any upstream signal, so the functional results in this section apply unchanged to the proposed architecture; only the final stage's timing characteristics, analyzed next, differ between the two.

VII. TIMING AND AREA TRADEOFF ANALYSIS

A. Carry-Chain Growth: Ripple-Carry vs. Carry-Look-Ahead

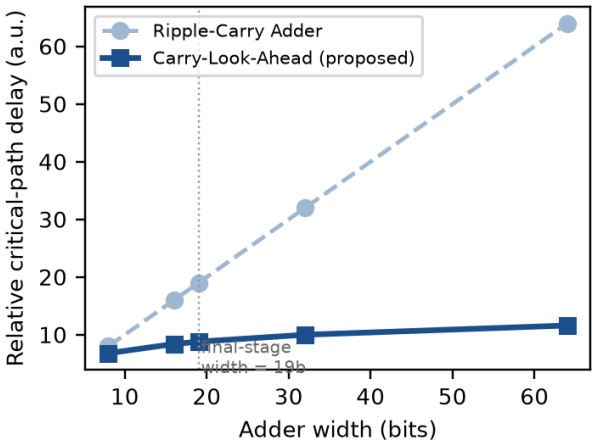

Fig. 8. Relative critical-path delay vs. adder width for a ripple-carry final stage (O(n)) against a carry-look-ahead final stage (O(log n)), with the design's 19-bit final-resolution width marked.

Section IV-C established that the existing ripple-carry final stage contributes an O(n) critical-path term while the proposed carry-look-ahead final stage contributes an O(log n) term, with the compressor-tree front end's O(1)-per-level contribution unchanged between the two architectures. Fig. 8 plots this scaling behavior across adder widths, marking the design's own 19-bit final-resolution width (the width produced by three successive one-bit growths applied to eight zero-extended 8-bit `abs_diff8` outputs, per Section III-C). The gap between the two curves widens with width, which is consistent with the general RCA-vs.-CLA propagation-delay comparison reported by Disha et al. at the standalone-adder level [13] and with the Kogge-Stone-vs.-ripple-carry and Sklansky-vs.-carry-select comparisons reported inside full multiplier datapaths [9],[12],[24]; this design's contribution is applying that same scaling relationship specifically at the odd, compressor-tree-derived 19-bit width rather than at a round power-of-two width.

B. Qualitative Comparison Against the Literature

Table II. Qualitative Comparison of Final-Stage Adder Candidates

Adder type	Delay scaling	Area scaling	Representative source
Ripple-carry (existing)	O(n)	O(n)	Balatero et al. [23], T. et al. [22]
Carry-select	O(sqrt(n)) (block-tuned)	O(n) with higher constant	Islam et al. [5], Pateru et al. [11]
Carry-look-ahead (proposed)	O(log n) (flattened) / O(n/k) blocks	O(n log n) (flattened) / O(n) (blocked)	Disha et al. [13]
Parallel-prefix (Kogge-Stone/Sklansky)	O(log n)	O(n log n)	A. and Rajeev [4], Bharathi and Varthamanan [15]

Table II situates the proposed carry-look-ahead final stage within the broader adder-family landscape surveyed in Section II. Carry-select adders reduce delay growth below O(n) without the full generate/propagate fan-in cost of look-ahead, at the price of duplicated sub-adder hardware for the precomputed carry-in=0/carry-in=1 cases [5],[11]. Parallel-prefix families (Kogge-Stone, Sklansky) achieve the same asymptotic O(log n) delay as a flattened carry-look-ahead adder via a structured prefix network, generally at a comparable or somewhat higher area cost depending on the specific prefix graph and fan-out constraints [4],[9],[12],[15]. For this design's specific final-stage width (19 bits, not a power of two), a block-based carry-look-ahead structure -- 4-bit or 5-bit look-ahead blocks with block-level group-generate/group-propagate signals cascaded across the remaining blocks -- offers a practical middle point: the within-block carries are fully parallel (Section IV-B), while the between-block carry chain is short enough (5 blocks for 19 bits at a 4-bit block size) that its residual ripple contributes only a small additive term to the overall O(log n)-dominated delay.

C. Why the Final Stage Alone Is the Right Substitution Point

Because every stage upstream of the final adder is already a bitwise compressor with O(1) per-level delay (Section III-C, confirmed functionally correct in Section VI), the final adder is the only remaining stage whose delay scales with operand width at all; substituting its internal carry-chain structure is therefore the single highest-leverage change available in this datapath, without requiring any redesign of the eight `abs_diff8` units or either compressor-tree level. This mirrors the general finding across the surveyed literature that adder substitution is typically evaluated at exactly the point where a design's last remaining serial carry chain sits, whether that is a standalone adder [13],[17], a multiplier's final accumulation stage [9],[12],[24], or -- as in this design -- a compressor tree's final resolution stage.

VIII. CONCLUSION

This paper documented an 8-pixel-pair Sum-of-Absolute-Differences unit built from a three-level carry-save compressor-tree reduction, took it through a complete Vivado 2019.2 project-creation, elaboration, and XSIM simulation flow, and verified it functionally against a self-checking, decimal-radix testbench of 50 vectors -- 6 hand-picked boundary cases and 44 randomized cases -- with zero

mismatches, confirmed both in the textual simulation log and in waveform-viewer captures showing the `sad` output tracking an independently computed `expected` reference cycle by cycle. Building on this verified baseline, the paper specified a carry-look-ahead replacement for the design's single remaining serial-carry stage -- the final 19-bit ripple-carry adder -- and analyzed its generate/propagate carry equations, algorithmic complexity, and delay-scaling behavior relative to the existing ripple-carry stage and to carry-select and parallel-prefix alternatives reported in the recent literature. Because the proposed change is confined to this one final-stage module, the functional verification results reported for the existing architecture carry over unchanged to the proposed one, with the only difference being the final stage's critical-path growth, which the analysis in Section VII places at O(log n) for the proposed structure against O(n) for the existing one.

Future work includes synthesizing both the existing (RCA-final-stage) and proposed (CLA-final-stage) architectures through Vivado's synthesis and implementation flow against the timing constraints specified in Section V-D to obtain post-implementation timing and utilization reports, extending the design from a single 8-pixel-pair block to a full search-window motion estimator so the final-stage delay reduction's effect on achievable search rate can be measured directly, and evaluating block-based look-ahead granularity (4-bit vs. 5-bit blocks) at this design's specific 19-bit width to identify the area/delay operating point best suited to the target device.

REFERENCES

[1] A. Desai, S.G. Nayak, "Design of a Low-Power FBHA-OCSLA Hybrid Adder for High-Speed VLSI Systems," in *Proc. 2026 International Conference on Smart Futuristic Technology*, 2026, pp. 1-5. doi: 10.1109/icsft66733.2026.11507750

[2] P. Goswami, R. Biswas, "High Speed Fault Tolerant Approximate Adder for Low Power Application," in *Proc. 2024 IEEE 4th International Conference on VLSI Systems, Architecture, Technology and Applications (VLSI SATA)*, 2024, pp. 1-5. doi: 10.1109/vlsisata61709.2024.10560161

[3] V. Jain, S. Niha, S. Rajasekaran, H.S. Kumar, M.V. Vamsikrishna, A. Moiz, "High-Speed Approximate Adder Design Through Charge Recovery Logic and Hybrid Low Power Technique," in *Proc. 2025 IEEE 5th International Conference on VLSI Systems, Architecture, Technology and Applications (VLSI SATA)*, 2025, pp. 1-6. doi: 10.1109/vlsisata65374.2025.11070085

[4] A. A, S.P. Rajeev, "FPGA Implementation of a High-Speed ALU using Booth Multiplier and Kogge-Stone Adder," in *Proc. 2026 IEEE 6th International Conference on VLSI Systems, Architecture, Technology and Applications (VLSI SATA)*, 2026, pp. 1-5. doi: 10.1109/vlsisata69163.2026.11610225

[5] S. Islam, T. Ahmed, N.S. Rafi, M.F. Shajid, "Design of an Area-Efficient and High-Speed 8-Bit Hybrid Carry Select Adder for Low-Power VLSI Applications," in *Proc. 2026 IEEE 2nd International Conference on Quantum Photonics, Artificial Intelligence & Networking (QPAIN)*, 2026, pp. 1-6. doi: 10.1109/qpain69676.2026.11546202

[6] J. Zhang, C. Chen, "High-Speed Area-Efficient Adder Design with SET-MOS Technology," in *Proc. 2026 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2026, pp. 1067-1071. doi: 10.1109/iscas66217.2026.11563024

[7] G. Kumar, S. Kumari, R. Verma, A. Kumar Pandey, A. Kumar, "Design of Delay Efficient High Speed Adder Circuit," in *Proc. 2024 International BIT Conference (BITCON)*, 2024, pp. 1-5. doi: 10.1109/bitcon63716.2024.10985271

[8] Y. Zhao, P. Qin, Q. Xue, "Design of High-Speed Multimodulus Divider With Fast EOC Generation Counters and High-Speed Reset Configuration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 2126-2138, 2026. doi: 10.1109/tvlsi.2026.3684891

[9] D.S. Teja, P. Jagadeesh, S. Jency, "Implementation of High Speed and Low Area Novel 16-Bit Vedic Mathematic Algorithm Using Sklansky Adder in Comparison with Carry Select Adder," in *Proc. 2024 Ninth International Conference on Science Technology Engineering and Mathematics (ICONSTEM)*, 2024, pp. 1-6. doi: 10.1109/iconstem60960.2024.10568738

[10] A.Abinaya, R. N, P.Malini, "Efficient Desensitized Halfband Filter Design with High Speed Adder," in *Proc. 2026 7th International Conference on Inventive Research in Computing Applications (ICIRCA)*, 2026, pp. 249-253. doi: 10.1109/icirca69024.2026.11570452

[11] R. Pateru, J. Ajayan, K. Alekhya, G. Vamshi, J. Ruthvik, S. Rebelli, "Performance Comparison of 2 x1 Multiplexer Based High Performance Static and Dynamic Nanoscale Carry Select Adder for Future DSP Applications," in *Proc. 2024 IEEE 4th International Conference on VLSI Systems, Architecture, Technology and Applications (VLSI SATA)*, 2024, pp. 1-5. doi: 10.1109/vlsisata61709.2024.10560075

[12] D.S. Teja, P. Jagadeesh, "Evaluation of Novel 16-Bit Vedic Multiplier Using Carry Look Ahead Adder for High Speed and Low Area in Comparison with Carry Select Adder," in *Proc. 2024 International Conference on Intelligent and Innovative Technologies in Computing, Electrical and Electronics (IITCEE)*, 2024, pp. 1-6. doi: 10.1109/iitcee59897.2024.10467821

[13] M. Disha, K. Preethika, P.A.T. Yadav, J. Naveen, A.S. Rao, J.C. Saldanha, "High-Speed Adder Design: Propagation Delay Comparison of CMOS RCA and CLA," in *Proc. 2026 IEEE International Conference on Interdisciplinary Approaches in Technology and*

Management for Social Innovation (IATMSI), 2026, pp. 1-6. doi: 10.1109/iatmsi68868.2026.11466296

[14] K. Ramesh, R. Bolimera, S. Alle, S. Surajkumar, P. Harshavardhan, S. Shanavaz, "Design of High-Speed Wallace Tree multiplier Used Full-Adder," in *Proc. 2024 International BIT Conference (BITCON)*, 2024, pp. 1-4. doi: 10.1109/bitcon63716.2024.10985663

[15] D.V.N. Bharathi, Y. Varthamanan, "Implementation of A High Speed Layout Parallel Prefix Adder for Design Applications," in *Proc. 2024 8th International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, 2024, pp. 194-198. doi: 10.1109/iceca63461.2024.10800816

[16] P. Rahimi, M. Tabany, S. Pourmoafi, "A Novel Low Power and High Speed 9- Transistors Dynamic Full-Adder Cell Simulation and Design," in *Proc. 2023 IEEE Symposium on Computers and Communications (ISCC)*, 2023, pp. 1287-1292. doi: 10.1109/iscc58397.2023.10217831

[17] A.A. Share, F.N. Zghoul, O. Al-Khaleel, M. Al-Khaleel, C. Papachristou, "Design of High Speed BCD Adder Using CMOS Technology," *IEEE Access*, pp. 141628-141639, 2023. doi: 10.1109/access.2023.3342328

[18] V. Chintala, P. Kishore, S.H. Vaishnavi, "Design of Low Power and High Speed Memristor-Based Full Adder for In-Memory Computing," in *Proc. 2026 IEEE International Conference for Convergence in Computing Technology (I3CTCON)*, 2026, pp. 1-5. doi: 10.1109/i3ctcon68242.2026.11507436

[19] T.R. Roy Naidu, R. Shankar, K.S. Reddy, D.K. Panda, "Design and ASIC Implementation of a Low-Latency AXI-Based Interconnect for High-Speed Low-Power VLSI Systems," in *Proc. 2026 IEEE 6th International Conference on VLSI Systems, Architecture, Technology and Applications (VLSI SATA)*, 2026, pp. 1-6. doi: 10.1109/vlsisata69163.2026.11610290

[20] D. Kumar, P. Karuppanan, "Design of High-Performance Hybrid Full Adder with Enhanced Speed and Power Characteristics," in *Proc. 2025 IEEE 12th Uttar Pradesh Section International Conference on Electrical, Electronics and Computer Engineering (UPCON)*, 2025, pp. 1-5. doi: 10.1109/upcon66414.2025.11454049

[21] C. Fang, Z. Shen, F. Tian, J. Yang, M. Sawan, "An Area-Efficient and Bit-Width Configurable Carry-Save Adder Tree for Spiking Transformers," in *Proc. 2025 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2025, pp. 1-5. doi: 10.1109/iscas56072.2025.11043823

[22] R. T, S. V, S. A, P. Malini, "Design and Implementation of a 4-Bit Carry Save Adder Using MTCMOS-Based Ripple Carry Adder with 10T Full Adders in 90nm Technology," in *Proc. 2025 IEEE First International Conference on Innovations in Engineering and Next-Generation Technologies for Sustainability (ICINVENTS)*, 2025, pp. 1-9. doi: 10.1109/icinvents64613.2025.11401534

[23] P.N.L. Balatero, J.J. Galang, T.M.M. Gonzales, A. Salem-Diaz, "Design of a Low-Power 4x4 Wallace Tree Multiplier Using Ripple Carry Adder at 10 MHz in 180nm CMOS Techology," in *Proc. 2024 23rd International Symposium on Communications and Information Technologies (ISCIT)*, 2024, pp. 71-75. doi: 10.1109/iscit63075.2024.10793598

[24] K. Sathish, P. Jagadeesh, "An Efficient Design and Performance Analysis of Novel 8 Bit Modified Wallace Multiplier Using Kogge-Stone Adder (KSA) in Comparison with Ripple carry adder (RCA)," in *Proc. 2023 Eighth International Conference on Science Technology Engineering and Mathematics (ICONSTEM)*, 2023, pp. 1-7. doi: 10.1109/iconstem56934.2023.10142406

[25] N. S, M. J, M.T. S M, "Energy Efficient Reversible Carry Tree Adder," in *Proc. 2026 IEEE 6th International Conference on VLSI Systems, Architecture, Technology and Applications (VLSI SATA)*, 2026, pp. 1-6. doi: 10.1109/vlsisata69163.2026.11609973