Area-Efficient LUT-Based Multipliers for AMD Versal FPGAs

Zetao Miao, Xander Pottier, Jonas Bertels, Wouter Legiest, Ingrid Verbauwhede
COSIC, KU Leuven
Leuven, Belgium
{zetao.miao, xander.pottier, jonas.bertels, wouter.legiest, ingrid.verbauwhede}@kuleuven.be

***Abstract*—AMD Versal FPGAs introduce a new CLB micro-architecture featuring the LOOKAHEAD8 carry structure in place of the legacy CARRY4/8 chains, on which existing area-efficient LUT-based multiplier designs map inefficiently. This paper proposes a LUT-based integer multiplier architecture tailored to the Versal fabric. By jointly exploiting radix-4 modified Booth recoding and the new Versal LUT micro-architecture, only $\sim n^2/4$ LUTs are required to generate the partial-product bit heap for an n-bit multiplication. A new heuristic for compressor-tree synthesis further improves the area–delay product by 8–20% over state-of-the-art Versal heuristics. Overall, the proposed multipliers achieve up to 40% LUT reduction relative to AMD LogiCORE IP multipliers at comparable critical-path delay. An open-source Python RTL generator with configurable operand widths and pipeline depths is provided for scalable deployment.**

***Index Terms*—LUT-based multiplier, radix-4 Booth recoding, compressor tree synthesis, FPGA, Versal**

I. Introduction

Multiplication is a fundamental and performance-critical operation across many field-programmable gate array (FPGA) applications. Different applications require multipliers of different operand precisions and performance characteristics. For instance, modern neural networks increasingly adopt low-precision integer arithmetic, typically INT8 or even INT4. DSP algorithms such as filtering and spectral analysis commonly require medium-precision fixed-point or floating-point multiplications. In contrast, cryptographic algorithms rely on large integer multiplications, often up to several hundred bits.

To accelerate multiplication, modern FPGAs integrate fixed-size DSP blocks that provide efficient embedded multipliers. However, their fixed operand formats, rigid placement, and limited quantity make them an inflexible, scarce resource. In contrast, look-up-table (LUT)-based multipliers support arbitrary operand sizes and flexible placement, built from the most abundant computational resource in FPGA fabrics. They can also be combined with DSP blocks to construct wide-precision multipliers. Consequently, efficient LUT-based multiplier design remains critical for precision-diverse FPGA workloads.

Most prior work has focused on optimizing LUT-based multipliers for AMD Xilinx 7-series and UltraScale/UltraScale+ architectures, exploiting the configurable logic block (CLB) micro-architecture of 6-input LUTs coupled with CARRY4/8 primitives to generate and accumulate partial products [1]–[6] or to compress partial-product bit heaps [6]–[8]. The CLB micro-architecture of AMD Versal devices differs substantially, most notably by replacing CARRY4/8 with the LOOKAHEAD8 carry structure [9]. Consequently, many prior designs no longer map efficiently to Versal, motivating new multiplier architectures tailored to Versal CLBs.

This paper presents a LUT-based multiplier architecture designed for Versal CLBs. Partial products are generated efficiently by exploiting the new cascade multiplexers in Versal LUTs, and accumulated through compressor trees synthesized by a new heuristic under the LOOKAHEAD8 carry constraints. We further provide an open-source Python-based RTL generator[1] that produces either standalone compressor trees or complete multipliers, with user-configurable operand widths and pipeline depths. Evaluation shows up to 40% LUT reduction over AMD LogiCORE IP multipliers [10] at comparable critical-path delay, and an 8–20% area–delay product improvement over state-of-the-art Versal compressor-tree heuristics.

II. Background

A. Versal CLB Adaptations

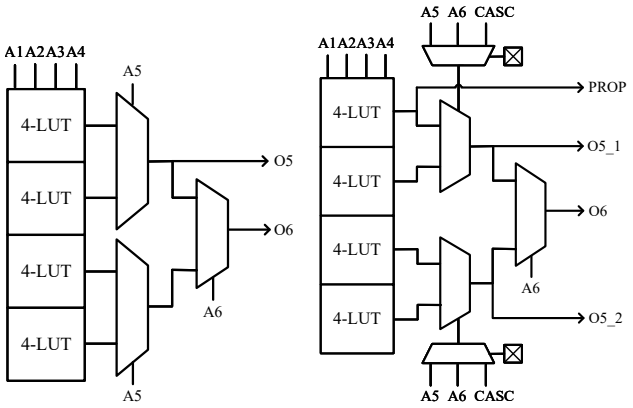
While Versal CLBs retain the overall organization of LUTs, carry logic, and flip-flops, the internal architectures of both the LUT and the carry chain differ significantly from previous FPGA generations.

Fig. 1 compares the Versal LUT micro-architecture with those of UltraScale/UltraScale+ devices. One major change is the introduction of two independently configurable cascade multiplexers, which control two A5 multiplexers, allowing the A5, A6, or CASC pins to serve as the fifth input of a

This work was supported in part by the Horizon 2020 ERC Advanced Grant (101020005 Belfort) and the CyberSecurity Research Flanders with reference number VOEWICS02. Xander Pottier is funded by FWO (Research Foundation – Flanders) as Strategic Basic (SB) PhD fellow (project number 1S93126N). Wouter Legiest is funded by FWO (Research Foundation – Flanders) as Strategic Basic (SB) PhD fellow (project number 1S57125N).

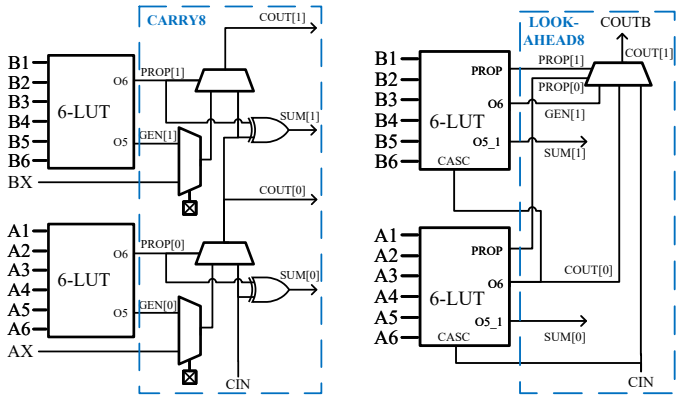


[1] https://github.com/KULeuven-COSIC/crypt-arith



(a) UltraScale/UltraScale+ (b) Versal

Fig. 1: LUT micro-architectures based on [9].



(a) UltraScale/UltraScale+ (b) Versal

Fig. 2: First two-bit section of carry hardware with LUTs.

Versal LUT. For example, in dual 5-LUT mode, the two 5-input sub-LUTs of an UltraScale/UltraScale+ LUT must share all five inputs (A1–A5), whereas Versal allows them to take independent fifth inputs from A5 and A6. The output of the second 5-input sub-LUT now appears on pin O5_2 instead of O6. Versal LUTs also expose an extra PROP output, an internal signal dedicated to the LOOKAHEAD8 carry structure.

Versal also introduces a dedicated cascade path between each pair of adjacent LUTs, as illustrated in Fig. 2b: a direct wire connects the O6 output of the lower LUT to the cascade input (CASC) of the upper LUT, enabling fast signal propagation without consuming general-purpose routing resources.

Fig. 2 illustrates the first two-bit section of the UltraScale/UltraScale+ CARRY8 block and the corresponding Versal LOOKAHEAD8 block. Compared to CARRY8, LOOKAHEAD8 removes the XOR gates and multiplexers that previously allowed signals outside the CLB to directly access the carry chain. Propagation signals are now generated by a dedicated 4-input sub-LUT inside each Versal LUT and exposed through the PROP port instead of O6. The carry multiplexers are organized in two-bit sections, with carry propagation within each two-LUT pair routed through the LUT cascade paths. Each LOOKAHEAD8 block contains four such sections in a carry-lookahead arrangement, detailed in Section IV.

B. Partial Product Generation

Given two signed integers A and B in two's complement form with n and m bits respectively, partial-product bits can be generated in several ways. The most straightforward is radix-2 generation, in which each bit is the two-input AND (or NAND) of one bit of A and one bit of B, producing approximately $n \times m$ partial-product bits, with the Baugh–Wooley technique [11] applied to simplify sign extension.

To reduce the number of partial products, radix-4 modified Booth recoding [12] is widely used. It recodes one operand (typically B) into radix-2^2 digits $b'_i = -2b_{2i+1} + b_{2i} + b_{2i-1}$, where $b_{-1} = 0$, $i \in \{0, 1, \ldots, m/2\}$, and B is sign-extended to an even number of bits if necessary. The number of partial products is thereby roughly halved, and since each digit satisfies $b'_i \in \{-2, -1, 0, 1, 2\}$, the corresponding partial products are obtained by shift-and-add/subtract operations on the multiplicand A.

C. Generalized Parallel Counters

Once partial products are generated, they are reduced to fewer operands before the final addition stage. This reduction is typically performed by parallel counters that compress multiple input bits of the same or different weights into fewer output bits while preserving the overall numerical value. Generalized parallel counters (GPCs) provide an abstract representation of such circuits, enabling systematic analysis and construction of compressor trees [13]. A GPC is denoted as $(p_{m-1}, p_{m-2}, \ldots, p_0 : q_{n-1}, q_{n-2}, \ldots, q_0)$, where p_i and q_i denote the numbers of input and output bits of weight 2^i, respectively, and the weighted sum of outputs equals that of the inputs. When each output column contains exactly one bit, we abbreviate the notation as $(p_{m-1}, p_{m-2}, \ldots, p_0 : n]$, where n is the total number of output bits. For example, $(1, 5 : 1, 1, 1)$ is written $(1, 5 : 3]$.

Two metrics are commonly used to compare GPCs: *efficiency* E and *strength* S. Efficiency E, defined as the ratio of bits reduced to LUTs used, is implementation-dependent and measures how effectively LUTs reduce bits [13]. Strength S, defined as the ratio of input to output bits, is implementation-independent and captures the inherent bit-reduction capability [14].

$$E = \frac{\sum_{i=0}^{m-1} p_i - \sum_{i=0}^{n-1} q_i}{\#\text{LUTs}}, \quad S = \frac{\sum_{i=0}^{m-1} p_i}{\sum_{i=0}^{n-1} q_i}. \tag{1}$$

Compressor-tree synthesis on FPGAs iteratively applies GPCs across bit positions to reduce bit-heap height until a final adder can resolve the remaining bits. Existing methods fall into two categories: heuristic approaches select GPCs by metrics such as *efficiency* and *strength*, achieving good area–delay trade-offs while being computationally inexpensive [15], [16], [18], whereas optimization-based approaches formulate

TABLE I: Basic Versal GPCs used for Compressor Tree Synthesis in [18]

Counter	#LUTs	E	S	LO.AH.[1]	Row/Column[2]
$(3:2]$	1	1.0	1.5	✓	Both
$(6:3]$	3	1.0	2.0	×	Neither
$(10:4,2)$	3	1.33	1.67	×	Neither
$(2,5:1,2,1)$	2	1.5	1.75	×	Column
$(1,5:3]$	2	1.5	2.0	✓	Row
$(2,2,3:4]$	2	1.5	1.75	×	Row

[1]if the GPC is compatible with carry-lookahead structure
[2]if the GPC is used to construct row/column counters

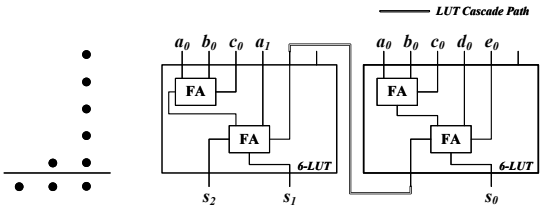


Fig. 3: $(1,5:3]$ GPC in [18].

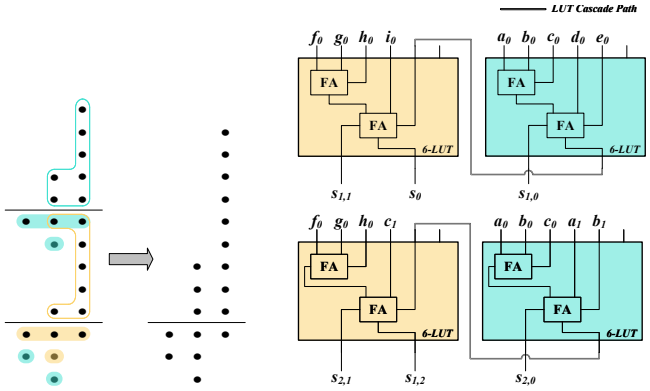


Fig. 4: $(3,9:2,3,1)$ GPC in [18], composed of two (1,5:1,2,1) GPCs.

the problem as integer linear programming (ILP), yielding optimal or near-optimal solutions but at significant computational expense [16], [17]. Practical FPGA designs therefore mostly rely on heuristic synthesis.

III. Related Work

A. Versal GPCs and Compressor Tree Synthesis

Hoßfeld *et al.* [18] were the first to design GPCs for Versal, revisiting UltraScale/UltraScale+ counters and proposing new structures tailored to the Versal architecture. Table I summarizes the basic GPCs in their compressor-synthesis framework. A defining feature of these GPCs is their use of the new LUT cascade path: depending on the construction strategy, the cascade propagates either carry signals, as in the $(1,5:3]$ counter shown in Fig. 3, or sum signals, as in the $(3,9:2,3,1)$ counter in Fig. 4.

Beyond basic GPCs, Hoßfeld *et al.* [18] compose two larger structures: row counters and column counters. Row counters extend horizontally, with eligible GPCs forwarding their carry outputs across bit positions, while column counters grow vertically by cascading sum signals to target tall bit heaps. Column counters such as $(2n+1:n,1)$ and $(n+1,4n+1:n,n+1,1)$ are formed by cascading $(3:2]$ and $(2,5:1,2,1)$ GPCs, respectively, with rippled sum. For example, the $(3,9:2,3,1)$ counter in Fig. 4 (called the dual-rail ripple-sum counter in [18]) is built by cascading two $(2,5:1,2,1)$ counters ($n=2$). Row counters, on the other hand, cascade $(1,5:3]$, $(3:2]$, and $(2,2,3:4]$ GPCs through carry propagation. Column counters are incompatible with the LOOKAHEAD8 carry-lookahead structure, so their GPCs must cascade through general-purpose routing. Among row counters, only $(1,5:3]$ and $(3:2]$ are LOOKAHEAD8-compatible; the $(2,2,3:4]$ GPC is therefore limited, as its cascading likewise relies on general-purpose routing.

To construct compressor trees, Hoßfeld *et al.* [18] proposed a stage-wise heuristic that iteratively reduces the height of the bit heap through dynamically selected GPCs. At each stage, candidate counters, including row counters and column counters, are evaluated and selected by prioritizing their efficiency or strength. Experimental results demonstrate significant LUT savings (around 45%) compared to Vivado adder-tree implementations.

B. Versal LUT-Based Multipliers Using Gate Absorption

For LUT-based tree multipliers, gate absorption is a compressor-tree optimization that merges radix-2 partial-product logic into the compressors instead of implementing it as a separate pre-processing stage. By exploiting unused LUT inputs in feasible GPCs, it eliminates the additional logic layer in the first compression stage. Hoßfeld *et al.* [18] report that absorbing two-input gates into the compressor tree reduces LUT utilization by at least 21%.

IV. Proposed Multipliers

We propose a tree multiplier with two stages: partial-product bit heap generation followed by GPC-based compression. Both stages are optimized for the Versal fabric.

A. Partial-Product Generation

Partial-product generation on FPGAs depends on the chosen scheme. For radix-2 partial products, two AND (or NAND) gates fit in a single LUT5:2 on both UltraScale/UltraScale+ and Versal devices, requiring approximately $n^2/2$ LUTs for an n-bit multiplication. Radix-4 modified Booth recoding halves the partial-product count but makes each bit a five-input Boolean function. On UltraScale/UltraScale+ FPGAs, this typically yields a similar LUT cost as radix-2, since one LUT produces only one such bit.

On Versal, however, radix-4 modified Booth recoding can be implemented much more efficiently thanks to the new LUT micro-architecture. The proposed multiplier employs this scheme as summarized in Table II, where $P'_{i,j}$ denotes a partial-product bit with weight 2^{2i+j} and c_i a carry bit with weight 2^{2i}. Each signed partial product P'_i consists of bits

TABLE II: Partial-Product Generation of Proposed Multipliers

b_{2i+1}	b_{2i}	b_{2i-1}	P'_i	$P'_{i,n}$	$P'_{i,n-1}$	$P'_{i,n-2}$	...	$P'_{i,2}$	$P'_{i,1}$	$P'_{i,0}$	c_i
0	0	0	0	0	0	0	...	0	0	0	0
0	0	1	$+A$	a_{n-1}	a_{n-1}	a_{n-2}	...	a_2	a_1	a_0	0
0	1	0	$+A$	a_{n-1}	a_{n-1}	a_{n-2}	...	a_2	a_1	a_0	0
0	1	1	$+2A$	a_{n-1}	a_{n-2}	a_{n-3}	...	a_1	a_0	0	0
1	0	0	$-2A$	$\overline{a_{n-1}}$	$\overline{a_{n-2}}$	$\overline{a_{n-3}}$	...	$\overline{a_1}$	$\overline{a_0}$	1	1
1	0	1	$-A$	$\overline{a_{n-1}}$	$\overline{a_{n-1}}$	$\overline{a_{n-2}}$	...	$\overline{a_2}$	$\overline{a_1}$	$\overline{a_0}$	1
1	1	0	$-A$	$\overline{a_{n-1}}$	$\overline{a_{n-1}}$	$\overline{a_{n-2}}$	...	$\overline{a_2}$	$\overline{a_1}$	$\overline{a_0}$	1
1	1	1	0	1	1	1	...	1	1	1	1

						1	$\overline{P'_{0,8}}$	$P'_{0,7}$	$P'_{0,6}$	$P'_{0,5}$	$P'_{0,4}$	$P'_{0,3}$	$P'_{0,2}$	$P'_{0,1}$	$P'_{0,0}$
					1	$\overline{P'_{1,8}}$	$P'_{1,7}$	$P'_{1,6}$	$P'_{1,5}$	$P'_{1,4}$	$P'_{1,3}$	$P'_{1,2}$	$P'_{1,1}$	$P'_{1,0}$		c_0
			1	$\overline{P'_{2,8}}$	$P'_{2,7}$	$P'_{2,6}$	$P'_{2,5}$	$P'_{2,4}$	$P'_{2,3}$	$P'_{2,2}$	$P'_{2,1}$	$P'_{2,0}$		c_1		
		1	$\overline{P'_{3,8}}$	$P'_{3,7}$	$P'_{3,6}$	$P'_{3,5}$	$P'_{3,4}$	$P'_{3,3}$	$P'_{3,2}$	$P'_{3,1}$	$P'_{3,0}$		c_2			
+)								1		c_3						
	p_{15}	p_{14}	p_{13}	p_{12}	p_{11}	p_{10}	p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Fig. 5: Partial products of an 8-bit multiplication with Baugh–Wooley sign extension [11].

TABLE III: Proposed LUT Mapping for Radix-4 Booth Partial-Product Generation.

LUT Pin	**A1**	**A2**	**A3**	**A4**	**A5**	**A6**	**O5_1**	**O5_2**
Signal	b_{2i-1}	b_{2i}	b_{2i+1}	a_j	a_{j-1}	a_{j+1}	$P'_{i,j}$	$P'_{i,j+1}$

$P'_{i,j}$ for $j \in \{0, 1, \ldots, n\}$ together with c_i. Fig. 5 illustrates the partial products for an 8-bit multiplication.

Table II shows that c_i can be derived directly from b_{2i+1} without LUT cost. Each partial-product bit $P'_{i,j}$ is a Boolean function of five inputs: $(b_{2i+1}, b_{2i}, b_{2i-1})$ from operand B and (a_j, a_{j-1}) (with $a_{-1} = 0$) from operand A. Moreover, adjacent bits $(P'_{i,j+1}, P'_{i,j})$ share four inputs, $(b_{2i+1}, b_{2i}, b_{2i-1}, a_j)$. This property enables the efficient LUT mapping shown in Table III, exploiting the dual 5-LUT mode of Versal LUTs with independent fifth inputs (Section II). A single Versal LUT therefore generates two radix-4 partial-product bits, so the entire partial-product bit heap requires only $\sim n^2/4$ LUTs for an n-bit multiplication.

B. Versal GPCs for Bit Heap Compression

While Hoßfeld *et al.* [18] introduced a set of GPCs together with row and column counters, this subsection refines that set to better exploit the Versal architecture. Table IV lists all fundamental GPCs used in this work.

First, we propose a new GPC $(9:4,1)$ (Fig. 6, $E = 1.33$, $S = 1.8$) to replace the $(10:4,2)$ counter for higher strength, and the column counter $(2n+1:n,1)$ with $n = 4$ for improved efficiency. Second, we do not adopt column counters in the proposed construction, since they must be depth-limited to meet timing constraints. Specifically, the $(2n+1:n,1)$ counters are excluded for low efficiency and strength, while counters of the form $(n+1, 4n+1:n, n+1, 1)$ are treated as independent GPCs for $n = 4, 3, 2$. The $(2, 5:1, 2, 1)$ counter $(n = 1)$ is also removed, since it is subsumed by the stronger $(1, 5:3]$ counter.

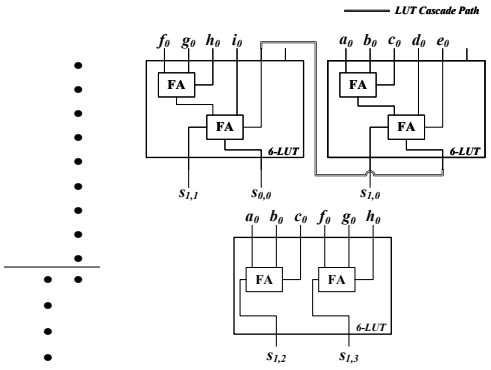


Fig. 6: New Proposed $(9:4,1)$ GPC on Versal fabric.

TABLE IV: Basic Versal GPCs used for Compressor Tree Synthesis in This Work.

Counter	**#LUTs**	**E**	**S**	**LO.AH.**[1]	**Row**[2]
$(5, 17:4, 5, 1)$	8	1.5	2.2	×	×
$(4, 13:3, 4, 1)$	6	1.5	2.125	×	×
$(3, 9:2, 3, 1)$	4	1.5	2.0	×	✓
$(9:4,1)$[3]	3	1.33	1.8	×	✓
$(6:3]$	3	1.0	2.0	×	✓
$(2, 2, 3:4]$	2	1.5	1.75	×	✓
$(3:2]$	1	1.0	1.5	✓	✓
$(1, 5:3]$	2	1.5	2.0	✓	✓

[1]if the GPC is compatible with carry-lookahead structure
[2]if the GPC is used to construct row counters
[3]new GPC proposed by this work

Row counters are constructed from an expanded set of eligible GPCs under LOOKAHEAD8 constraints, as detailed in the next subsection.

C. Versal Row Counter Design with LOOKAHEAD8

LOOKAHEAD8 fundamentally changes row-counter design by providing fast carry paths that avoid general-purpose routing. Its internal structure is illustrated in Fig. 7, where four carry-lookahead multiplexers are controlled by attributes LOOKB/D/F/H. When enabled, these attributes activate lookahead carry selection; otherwise, CYMUXB/D/F/H propagate CYB/D/F/H logically. Moreover, LOOKB also affects downstream multiplexers: when only LOOKB is FALSE, CYA and CIN are no longer selectable by CYMUXD/F, and the A5 input of LUTA replaces CIN as a carry candidate for CYMUXH.

These constraints introduce several limitations on row-counter construction. First, the carry chain must start from the A5 input of LUTA, since CIN can only be driven by the COUT of a neighboring LOOKAHEAD8 block. Second, disabling LOOK attributes modifies downstream carry selection, limiting which signals can propagate through the chain. Our timing-arc analysis further shows that when all LOOKx attributes are FALSE, only the CYB-to-COUTB timing arc exists, while other CYx-to-COUTx arcs are undefined in the timing model. Consequently, GPCs incompatible with the carry-lookahead structure cannot rely on logical propagation through LOOKAHEAD8 by setting LOOKD/F/H to FALSE; they must use general-purpose routing, which degrades performance.

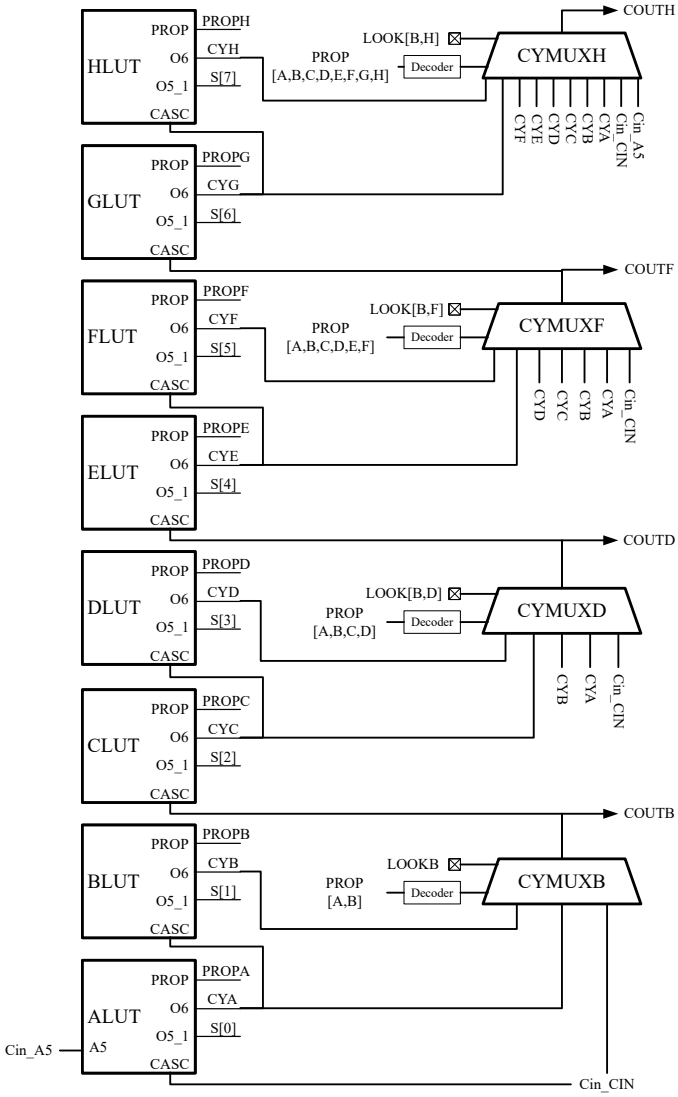


Fig. 7: Diagram of LOOKAHEAD8 logic model.

Based on these observations, we construct row counters from the eligible GPCs in Table IV according to the following rules:

1) GPCs compatible with the carry-lookahead structure can be cascaded in the row counter without restriction.
2) Incompatible GPCs may only appear at the beginning of the chain (where propagation can be forced via LOOKB configuration) or at the end (where no further carry propagation is required).
3) When a row counter starts with an incompatible GPC, its length is limited to 8 LUTs; otherwise, CYMUXH behavior may produce incorrect carry outputs.

A notable exception is the $(3, 9 : 2, 3, 1)$ GPC of Fig. 4. Despite being incompatible with the carry-lookahead scheme, its dual-rail property enables effective cascading via a parallel physical carry path, bypassing the 8-LUT length limit; an example is shown in Fig. 8. This GPC may also reduce compressor-tree depth compared to equivalent constructions based on two $(1, 5 : 3]$ GPCs connected through general-purpose routing.

D. New Implementation of Quaternary Terminal Adder

For terminal addition on Versal, Hoßfeld *et al.* [18] proposed a quaternary adder that sums four operands at two LUTs per bit, saving one LUT per bit over adder trees composed of two-operand adders. The design absorbs the carry and sum functions of a carry-save adder into LUTs implementing ripple-carry addition, as illustrated in Fig. 9. While effective in many cases, this implementation is not optimal for bit heaps where higher-weight columns contain only a single bit. Direct application of the quaternary adder prevents LUT merging in such columns due to parallel carry-propagation paths. Alternatively, combining it with a two-operand adder to propagate carries across single-bit columns introduces an additional logic stage and general-purpose routing, increasing critical-path delay.

To address these limitations, we propose an alternative quaternary adder implementation built from two layers of row counters, as shown in Fig. 10. The design primarily uses $(1, 5 : 3]$ GPCs and may additionally incorporate $(3 : 2]$ GPCs. The $(3 : 2]$ GPCs efficiently propagate carries through single-bit columns toward the MSB, requiring only one LUT per bit without introducing additional routing overhead. This structure improves adaptability to diverse bit-heap shapes while preserving the area and delay advantages of multi-operand addition.

E. Proposed Heuristic for Compressor Tree Synthesis

Efficient individual counters do not necessarily yield efficient compressor trees unless properly selected and scheduled. We propose a new stage-wise heuristic for compressor-tree synthesis on Versal. Unlike the heuristics of Hoßfeld *et al.* [18], which prioritize either the efficiency or the strength metric of GPCs, our heuristic accounts for both area and delay, aiming to minimize LUT cost while keeping the critical path short. To this end, counters within each GPC-compression stage are arranged in parallel, and GPC cascading in row counters is restricted to dedicated fast paths, avoiding general-purpose routing.

In the proposed heuristic, a GPC is scheduled only when it is both *applicable* and *necessary*. A GPC is *applicable* if sufficient free bits exist in the required input columns to fully populate the counter. A GPC is *necessary* if it is expected to achieve the lowest LUT cost for reducing the current column (and a limited span of subsequent columns) to a height of at most four. Table V lists the necessity conditions when a counter is aligned to column c, where H_c is the height of column c, counting unassigned bits and outputs from counters already scheduled in the current stage.

For each compression stage, scheduling starts from the right-most column whose height exceeds four. Candidate counters in Table V are examined in priority order, and the first counter that is both applicable and necessary is placed with its LSB aligned to the current column. Whenever possible,

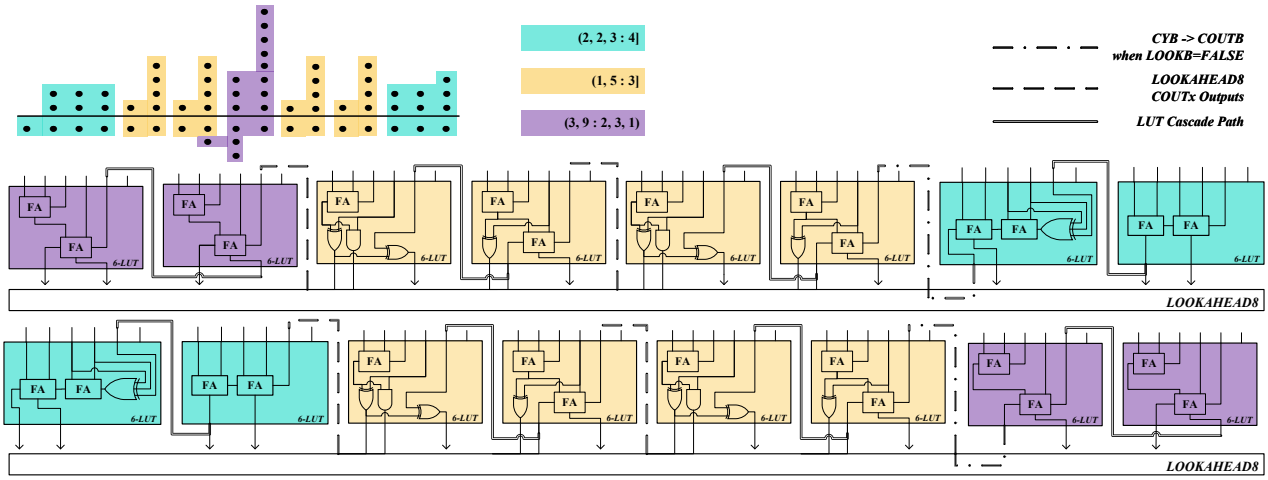


Fig. 8: Row counter benefits from dual-rail property of $(3, 9 : 2, 3, 1)$ GPC.

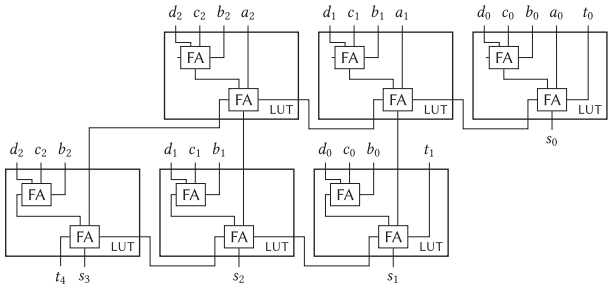


Fig. 9: Quaternary adder implementation in [18].

TABLE V: Counter Necessity Conditions

	Counter	Necessity Conditions
priority ↓	$(5, 17 : 4, 5, 1)$	Always Necessary
	$(4, 13 : 3, 4, 1)$	$H_c \geq 16$
	$(3, 9 : 2, 3, 1)$	$H_c \geq 12$
	$(9 : 4, 1)$	$H_c \geq 12$, $5\,H_c > 17\,H_{c+1}$
	$(6 : 3]$	$H_c = 9$, $H_{c+1} \leq 3$, $H_{c+2} \leq 3$
	$(2, 2, 3 : 4]$	$5 \leq H_c \leq 6$, $4 \leq H_{c+1} \leq 5$, $4 \leq H_{c+2} \leq 5$
	$(3 : 2]$	$5 \leq H_c \leq 6$
	$(1, 5 : 3]$	Always Necessary

the heuristic prioritizes starting or extending a row counter (i.e., cascading eligible GPCs) under the LOOKAHEAD8 constraints of the previous subsection. After placement, if the counter is a row-counter candidate, the scan advances to the column of its MSB output to continue the cascade; otherwise, it moves to the next column. The process repeats until no further counters can be scheduled in the current stage.

Stages are generated iteratively until the reduced bit heap can be finalized by the terminal quaternary adder. A consolidation step is then performed: if counters in the last GPC-compression stage can be under-utilized (i.e., with some inputs supplied by leftover bits from the preceding stage) without preventing terminal addition, the last two GPC-compression stages are merged. This reduces critical-path delay without increasing LUT cost.

F. RTL Generator for Proposed Multipliers

We developed a Python-based RTL generator for the proposed multipliers. The generator performs compressor-tree synthesis with the proposed heuristic and produces synthesizable SystemVerilog via primitive instantiation and explicit wiring. From user-defined parameters, it generates the partial-product logic and compressor-tree structures, along with XDC constraints, testbenches, and random test vectors; standalone compressor generation is also supported. Pipeline insertion is automatic: for a given pipeline depth, registers are placed across the partial-product generation, GPC-based compression, and terminal-addition logic to balance stage delays. All experimental results in Section V are obtained with designs from this generator.

V. Experimental Results

A. Evaluation Method

Designs are synthesized with default settings in Vivado, targeting xcvc1902-vsva2197-2MP-e-S device. They are embedded within a register sandwich to ensure consistent timing evaluation. The reported delay is the critical-path delay at the tightest successful timing closure, found by iteratively tightening the timing constraint in 0.1 ns steps. LUT utilization reported by Vivado is used as the area metric. For fair comparison, Vivado 2023.1 is used when comparing compressors against the designs of Hoßfeld *et al.* [18]; Vivado 2025.2 is used for all other designs.

B. Evaluation Results of Compressor Trees

Compressor trees are generated using the proposed generator with the same input shapes as those used by Hoßfeld *et al.* [18]; results are shown in Fig. 11.

The proposed heuristic achieves the best area efficiency across all evaluation cases. For single-column bit heaps, it produces critical-path delay comparable to the efficiency-first heuristic; for two-column and multi-column bit heaps, it achieves delay close to the strength-first heuristic. Overall, it

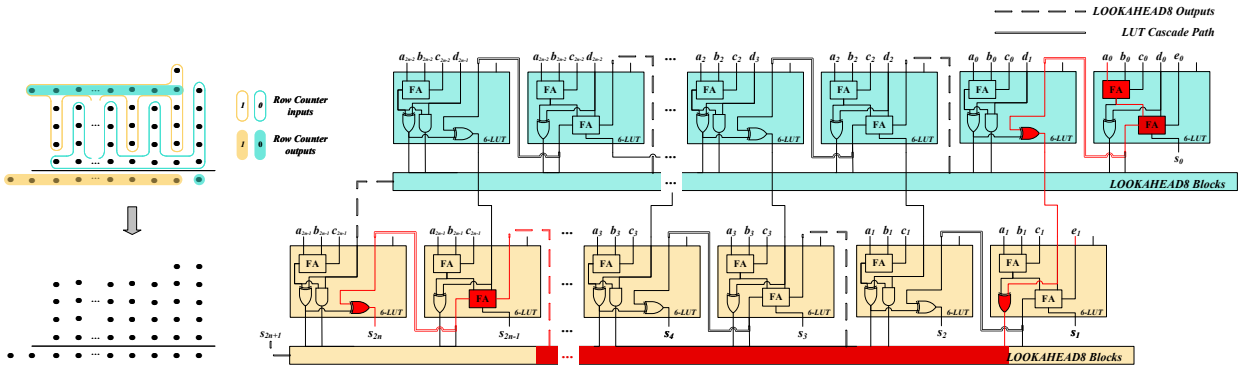


Fig. 10: Proposed implementation of quaternary adder as two layers of row counters.

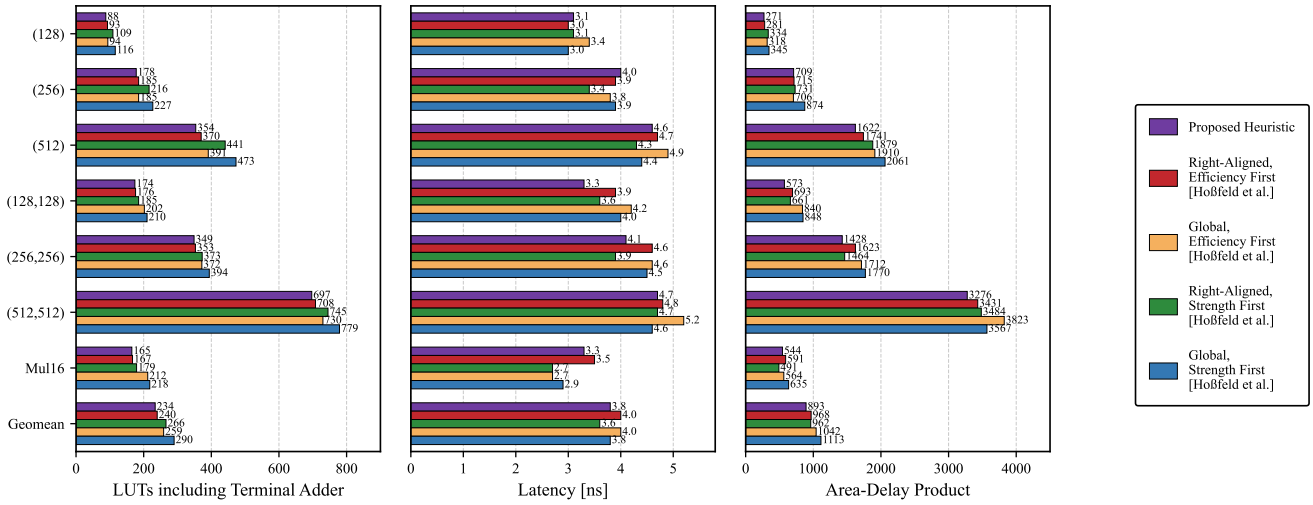


Fig. 11: Comparison of proposed compression heuristic and heuristics used by Hoßfeld *et al.* [18]. (128) denotes a bit heap consisting of a single column with 128 bits, (128,128) denotes a bit heap with two columns containing 128 bits each, and so on. Mul16 denotes the bit heap of 16-bit radix-2 multiplication. Geomean represents the geometric mean of all seven cases. The same Vivado version and Versal device as in [18] are used for fair comparison.

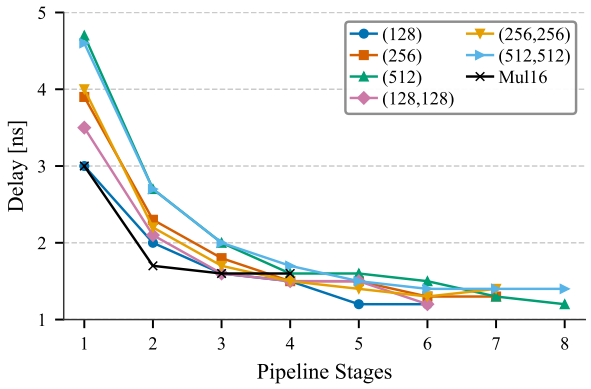


Fig. 12: Critical-path delay of the proposed compressors versus pipeline depth. Results obtained by using Vivado 2025.2.

reduces the area–delay product by 8–20% compared to state-of-the-art heuristics. Fig. 12 reports the critical-path delay of the proposed compressors under different pipeline depths, demonstrating the effectiveness of the generator's pipeline insertion.

C. Evaluation Results of Proposed Multipliers

We compare the proposed multipliers against AMD LogiCORE IP multipliers [10], configured for speed optimization since the area-optimized configuration consumes significantly more LUTs in our evaluation. Results are shown in Fig. 13. The proposed multipliers reduce LUT utilization by up to 40% compared to the speed-optimized LogiCORE designs while maintaining comparable critical-path delay. For operand widths between 28 and 32 bits, the proposed multipliers exhibit slightly longer critical-path delay due to the earlier introduction of an additional compression stage in the compressor tree.

Compared with the 16-bit multiplier with gate absorption from [18], which requires 245 LUTs, the proposed 16-bit multiplier reduces LUT usage by more than 25%, and even the proposed 18-bit multiplier requires over 10% fewer LUTs.

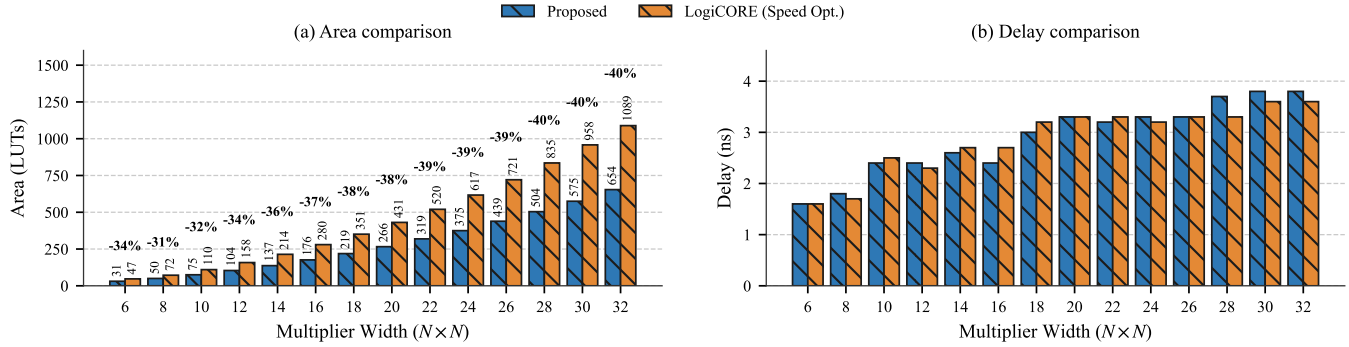


Fig. 13: Comparison of proposed multipliers and AMD LogiCORE IP multipliers (speed optimized).

VI. Conclusion

This paper presents an area-efficient LUT-based integer multiplier architecture tailored to AMD Versal FPGAs. By jointly exploiting radix-4 modified Booth recoding, the Versal LUT micro-architecture, and the LOOKAHEAD8 carry structure, the proposed design enables efficient partial-product generation and compressor-tree construction. We further introduce an architecture-aware heuristic for GPC-based compressor-tree synthesis that improves delay while preserving high area efficiency, and an automated Python-based RTL generator supporting compressor-tree synthesis and pipeline insertion. Experimental results show up to 40% LUT reduction over AMD LogiCORE IP multipliers at comparable delay, and an 8–20% area–delay product improvement over existing Versal compressor-tree heuristics.

Future work will extend the proposed architecture-aware methodology to other arithmetic operators on Versal, further leveraging the new LUT structures and LOOKAHEAD8 carry architecture.

References

[1] E. G. Walters, "Partial-product generation and addition for multiplication in FPGAs with 6-input LUTs," in *Proc. 48th Asilomar Conf. on Signals, Systems and Computers*, pp. 1247–1251, 2014.

[2] M. Kumm, S. Abbas, and P. Zipf, "An efficient softcore multiplier architecture for Xilinx FPGAs," in *Proc. IEEE 22nd Symp. on Computer Arithmetic*, pp. 18–25, 2015.

[3] M. Shu and Q. Liu, "LHAM: Low-cost and high-accuracy approximate multiplier for FPGA-based computing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 18, no. 4, Art. no. 48, pp. 1–25, Dec. 2025.

[4] S. Ullah, S. Rehman, B. S. Prabakaran, F. Kriebel, M. A. Hanif, M. Shafique, and A. Kumar, "Area-optimized low-latency approximate multipliers for FPGA-based hardware accelerators," in *Proc. 55th Annu. Design Automation Conf. (DAC)*, pp. 1–6, 2018.

[5] R. Chen, Y. Lyu, H. Bao, and B. da Silva, "A power-efficient hardware implementation of L-Mul," arXiv preprint arXiv:2412.18948, 2024.

[6] A. Böttcher and M. Kumm, "Multiplier design addressing area-delay trade-offs by using DSP and logic resources on FPGAs," in *Proc. IEEE 35th Int. Conf. on Application-Specific Systems, Architectures and Processors (ASAP)*, pp. 217–225, 2024.

[7] A. Böttcher and M. Kumm, "Small logic-based multipliers with incomplete sub-multipliers for FPGAs," in *Proc. IEEE 31st Symp. on Computer Arithmetic (ARITH)*, pp. 124–131, 2024.

[8] A. Böttcher and M. Kumm, "Towards globally optimal design of multipliers for FPGAs," *IEEE Trans. Comput.*, vol. 72, no. 5, pp. 1261–1273, 2023.

[9] AMD, "Versal Adaptive SoC Configurable Logic Block Architecture Manual, AM005," AMD, Release 1.4, May 14, 2025. [Online]. Available: https://docs.amd.com/r/en-US/am005-versal-clb/Overview

[10] AMD, "Multiplier v12.0 Product Guide (PG108)," AMD, Nov. 18, 2015. [Online]. Available: https://docs.amd.com/v/u/en-US/pg108-mult-gen

[11] C. R. Baugh and B. A. Wooley, "A two's complement parallel array multiplication algorithm," *IEEE Trans. Comput.*, vol. C-22, no. 12, pp. 1045–1047, Dec. 1973.

[12] O. L. MacSorley, "High-speed arithmetic in binary computers," *Proc. IRE*, vol. 49, no. 1, pp. 67–91, 1961.

[13] H. Parandeh-Afshar, A. Neogy, P. Brisk, and P. Ienne, "Compressor tree synthesis on commercial high-performance FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 4, pp. 1–19, 2011.

[14] T. B. Preußer, "Generic and universal parallel matrix summation with a flexible compression goal for Xilinx FPGAs," in *Proc. 27th Int. Conf. on Field Programmable Logic and Applications (FPL)*, pp. 1–7, 2017.

[15] H. Parandeh-Afshar, P. Brisk, and P. Ienne, "Efficient synthesis of compressor trees on FPGAs," in *Proc. Asia and South Pacific Design Automation Conf. (ASP-DAC)*, pp. 138–143, 2008.

[16] M. Kumm and J. Kappauf, "Advanced compressor tree synthesis for FPGAs," *IEEE Trans. Comput.*, vol. 67, no. 8, pp. 1078–1091, 2018.

[17] H. Parandeh-Afshar, P. Brisk, and P. Ienne, "Improving synthesis of compressor trees on FPGAs via integer linear programming," in *Proc. Design, Automation and Test in Europe Conf. (DATE)*, pp. 1256–1261, 2008.

[18] K. Hoßfeld, H. J. Damsgaard, J. Nurmi, M. Blott, and T. B. Preußer, "High-efficiency compressor trees for latest AMD FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 17, no. 2, pp. 1–32, 2024.