Framework-based Arithmetic Datapath Generation to Explore Parallel Binary Multipliers

Leandro M. G. Rocha[1], Guilherme Paim[1], Gustavo M. Santana[1], Eduardo A. C. da Costa[2], Sergio Bampi[1]

[1]Graduate Program in Microelectronics (PGMicro) - Federal University of Rio Grande do Sul (UFRGS), Porto Alegre - Brazil
[2]Graduate Program in Electronic Engineering and Computing - Catholic University of Pelotas (UCPel), Pelotas - Brazil
e-mail: {lmgrocha, gppaim, gmsantana, bampi}@inf.ufrgs.br, {eduardo.costa}@ucpel.edu.br

***Abstract*— Arithmetic modules have a significant impact on performance, circuit area, energy, and power in digital circuits for DSP (Digital Signal Processing). Exploring implementation trade-offs in these circuits is paramount in low-power and low-cost devices such as sensors for IoT devices, which often have stringent requirements. Multipliers are of particular concern due to their ubiquitous use in DSP algorithms and their inherent hardware implementation complexity. This work proposes a framework to efficiently generalize and explore different compositions of arithmetic operators with an emphasis on parallel binary multipliers, guiding the designer through the micro-architecture development. Several partial product encoders, combined with multiple compression trees, can be generated automatically using the proposed framework. Here, we compare available multipliers in various configurations. Multipliers were synthesized using a commercial 65 nm standard cell library to obtain realistic area, power, and timing results.**

***Index Terms*— Parallel multipliers, Adders topologies, VLSI Design, ASIC.**

I. Introduction

Advances achieved by the semiconductor industry in recent years boosted circuit miniaturization which fostered the development of embedded applications like digital signal processing (DSP) system-on-a-chip (SoC), Internet of Things (IoT), wearable sensors, biomedical circuits, as well as high-speed and compute-intensive applications like neural network accelerators and general-purpose processors [1]. These applications have very different requirements in terms of speed, energy consumption, and development cost, requirements that create an optimization challenge that is hard to tackle and achieve a balanced solution.

One key optimization point relies on the circuit datapath as it usually comprises most of the circuit area and complexity. Specifically, arithmetic circuits like adders and multipliers tend to be the most significant modules in terms of performance, area, energy, and power in digital circuits. Although there is a vast literature for optimized arithmetic circuits, choosing the optimal implementation for each application is a cumbersome, long process, taking valuable time from designers. Each technology node has different characteristics that may impact the overall Quality of Results (QoR) of the implemented circuits. Therefore, the best architecture in one technology may not be optimized or the best solution in another. Hence, it is quintessential to simplify and accelerate the decision-making process in choosing the optimal circuit that will achieve the highest QoR.

Multipliers are of particular importance since they are inherently more complex than other arithmetic circuits. Moreover, they are used as a base module for other circuits like filters, convolution units, multiply-accumulate units (MAC), among others [2]. Parallel multipliers are, in their majority, composed of three blocks, wherein each may be optimized: the partial product generation, the compression tree, and the carry propagating adder. Among the most popular multiplier architectures are the Wallace multiplier [3] and the Booth multiplier [4]. However, newer structures like the Radix-2^m [5] have proven to be a promising alternative for power-aware designs as they have reduced switching activity.

In parallel multipliers, the efficiency of the addition tree of partial terms can have a substantial impact on the multiplier circuit performance. These addition trees rely on the carry-save scheme to avoid carrying propagation and improve the circuit speed and reduced power dissipation. Efficient compression trees like Wallace [3] and Dadda [6] can benefit from 3-2 compressor cells to reduce the tree depth and switching activity.

Despite the importance of arithmetic circuits for digital hardware, there are few works which aimed at the development of automatic arithmetic core generation. In [7], the authors proposed an RTL generator without any platform constraint, but it is limited to a single multiplier with limited input sizes. The work in [8] introduced FloPoCo, a library for automatic circuit generation for numerical functions used in scientific computations; however it targets only FPGA platforms and employs device-specific resources. In [9], the authors proposed an optimization to the FloPoCo library using bit-heaps. Despite being an FPGA-only framework, it features a bit-heap structure which is employed in this work.

Hence, this work is an extension of [10] which proposed RTLGen, a framework to simplify the description of arithmetic circuits, enabling the exploration of a multitude of circuit combinations to establish implementation trade-offs in multiple axes. This framework includes automatic arithmetic core generation, verification, synthesis, and QoR analysis, providing an environment to explore the trade-offs of binary parallel multipliers under different constraints. Therefore, the main contributions of this work are:

- A comprehensive explanation of the framework modules.
- Comparison of several multiplier architectures for several input sizes.
- Development of a web interface for public use to generate arithmetic circuits.

Digital Object Identifier 10.29292/jics.v15i3.91

The rest of the paper is organized as follows: Section II presents a review on multiplier circuits. Section III describes the proposed multiplier generation framework. The framework evaluation methodology with experimental results is shown in Section IV, and, finally, Section V concludes the paper.

II. Multiplier Circuits Background

To understand the binary multiplication, let's assume two numbers, A and X, that are, respectively, m- and n-bit wide, and $P_0, P_1 \cdots P_{n-1}$ the partial products of A times X, like the pencil-and-paper method. The logical operation *AND* between the multiplicand A and each bit of the multiplier X can generate these partial products. Next, these partial products are aligned according to their bit-weight, and shifted one position to the left at a time – i.e., one bit-weight higher. Then, these products are summed accordingly, obtaining the desired result. The partial products and their bits can be represented with a dot diagram to better visualize the design of the multiplier. Each dot represents a bit of a vector regardless of its value. Figure 1 shows a dot diagram for a 16-bit multiplication using the previously described algorithm, where each row corresponds to a partial product.

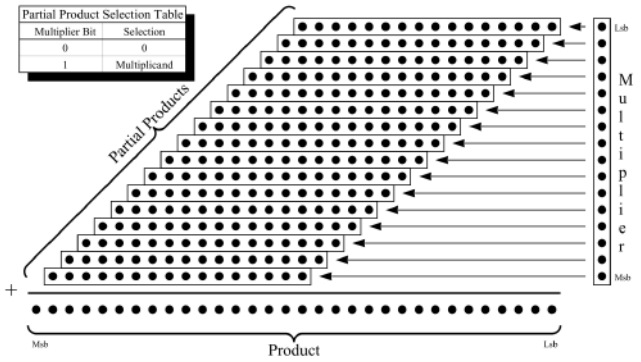

Fig. 1: Example of a classic binary multiplication.

For the hardware implementation, we can think of the multiplication algorithm as having three blocks (see Fig.2). First, a specific algorithm generates partial products from the inputs. Binary adders can only add two input operands at a time. Therefore, next, multipliers feature compression trees based on the carry-save scheme to reduce these partial products to only two values. Finally, these last two values are recombined using a carry propagating adder to obtain the final result. Figure 2 illustrates the overall architecture of a generic multiplier.

A. Multiplier Encoders

Choosing the optimal partial product generation (PPG) strategy for a given set of requirements is of utmost importance since it impacts both the encoder complexity and the size of the compression tree. Several PPG encoders have been proposed in the literature aiming for different optimization strategies, including examples like the Modified-Booth [7, 11] algorithm, Optimized Baugh-Wooley array [12], and Radix-2^m [5] for signed multiplication. Although these algorithms can compute unsigned multiplication with minor

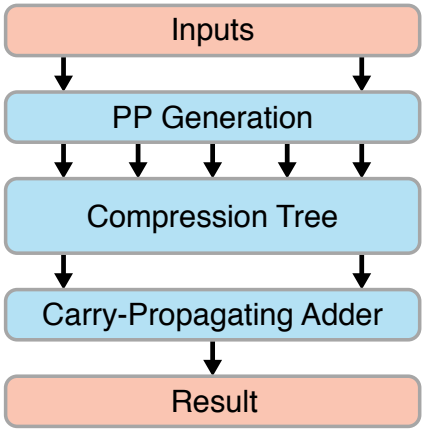

Fig. 2: General architecture for parallel binary multipliers.

tweaks, the traditional array-based multiplication is still used due to its low complexity.

For unsigned values, the *unsigned array* approach is the simplest partial product encoder as it is only composed of *AND* gates, as shown in Figure 1. For an n-bit input, there will be n partial products of n bits. This encoder is rarely used as it is very limited in terms of applications.

Most signal processing algorithms require signed operations. In this case, the Booth algorithm is very popular, given its reported performance [13]. The original Booth algorithm is not suitable for hardware implementation. Thus, all implementations are based on the Modified-Booth algorithm proposed in [14]. Several optimizations are proposed to make the algorithm more hardware-friendly, like pre-calculation of all sign bits [15] and pre-computation of least-significant bits (LSBs) for regular PPG circuitry [7], which are integrated into Figure 3 where x_n and y_n are the multiplicand and multiplier bits respectively, and p_{xy} represents the bits of each partial product. Note that for n-bit inputs, this algorithm generates $\lceil n/2 \rceil$ partial products of size $n + 1$.

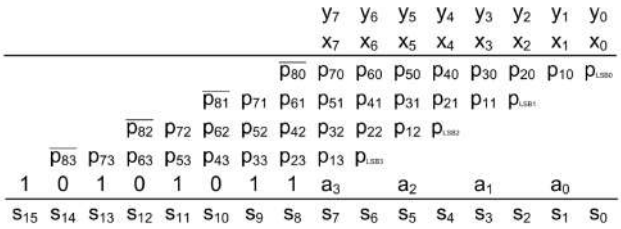

Fig. 3: Optimized Modified Booth algorithm [7].

Booth multipliers support higher radices to reduce even further the number of generated partial products. Still, this approach significantly increases the circuit complexity, which outweighs the benefits of reducing the compression tree. The work in [5] proposes a radix-based multiplier using a recoding technique similar to the Booth algorithm to tackle this issue. The *Radix-2^m* multiplier split the inputs into groups of *m* bits so that they can be seen as individual multiplications.

There are three basic encoder blocks (*Type-I*, *Type-II*, and *Type-III*). Each block performs the multiplications of m-bit inputs according to their signedness. *Type-I* blocks operate

on unsigned inputs, i.e., they are unused on the $N - m$ least significant bits of the operands. *Type-II* encoders compute the multiplication of an unsigned value by a signed value, whereas the *Type-III* encoder computes the multiplication of two signed inputs. These encoders are combined accordingly to generate partial products, as shown in Figure 4, for an 8-bit multiplier. The outputs of the encoders are summed using a carry propagating adder (CPA) to generate the partial product. Note that for the most significant bits multiplication, *Type-I* encoders are substituted by *Type-II* encoders, and *Type-II* encoders are substituted by *Type-III* encoders.

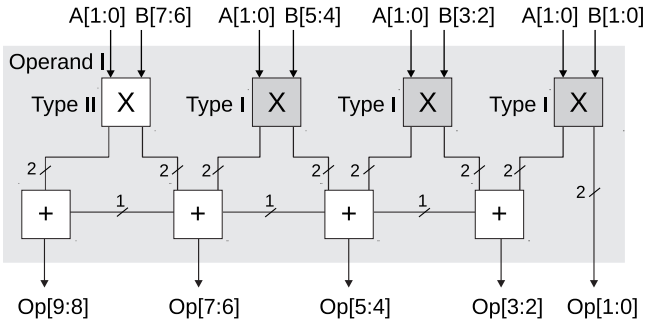

Fig. 4: Partial product generation on the Radix-2^m with $m = 2$.

In this work, we propose a different approach regarding the utilization of these blocks. Instead of summing the outputs of the encoders, we add them directly to the compression tree to remove the need of the CPA. This proposal is a trade-off in terms of design complexity since it reduces the critical path at the expense of removing the regularity of the partial product array and requiring sign-extension mechanisms to ensure the design correctness.

Instead of recoding the partial products as in the Booth and Radix-2^m multipliers, the *Baugh-Wooley* algorithm aims for simpler hardware based on the unsigned array. This scheme also considers both multiplicand and multiplier to be informed in two's complement representations, although the partial products are positive except from the last one, resulting in simpler hardware. Reordering the partial products, [12] proposes a very regular structure to map this algorithm to hardware efficiently. Figure 5 shows the array structure of a Baugh-Wooley multiplier, where x_n and y_n are the multiplicand and multiplier bits, respectively.

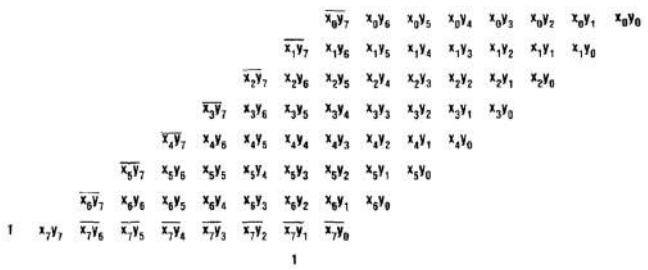

Fig. 5: Partial products scheme for the Baugh-Wooley algorithm [12].

B. *Compression trees*

Using binary adders to sum the partial products is not efficient in every aspect due to the carry propagation. Carry-save adders (CSA) address this issue with a redundant representation (*sum* and *carry*) that operates without carry propagation, resulting in much faster multipliers. The most common compression algorithms are the Wallace tree [3] and the Dadda tree [6].

1. **Wallace Tree:** This approach aims to compress the partial products as much as possible, as seen in Figure 6. This algorithm has four steps: (i) take any group of three bits with the same bit-weight and sum them using a full-adder. If there are more bits of the same weight, group them with either a full- or a half-adder; (ii) propagate the outputs for the next stage, with the *sum* bit having the same weight as the inputs while the *carry* bit will have a one-bit higher weight; (iii) if there is only a single bit left for the current weight, transfer it to the next level; (iv) repeat steps i-iii until there are no more than two bits left in any weight.

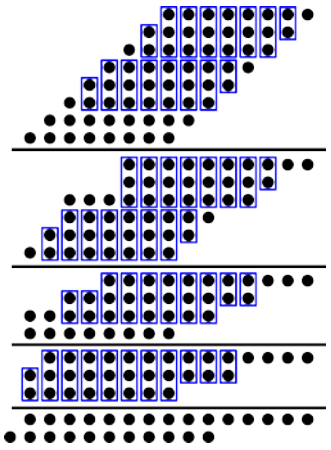

Fig. 6: Partial product compression using Wallace tree.

2. **Dadda Tree:** The Dadda multiplier uses a slightly different approach to distribute the adder cells in a carry-save fashion to sum the partial products. This approach uses an algorithm on which the maximum number of summands in each stage follows a geometric progression whose ratio is defined as a function of the compression ratio of the available adder cells.

C. *Carry-Propagating Adders (CPA)*

All binary adders require the carry to be propagated from the LSB to the MSB. Hence, the most straightforward way to execute this operation is using the ripple-carry adder (RCA), which sequentially computes the sum and the carry-out of each bit. From Figure 8, it is possible to observe that the carry-out of the first bit must be passed – or rippled – through all FA cells, until the leftmost bit.

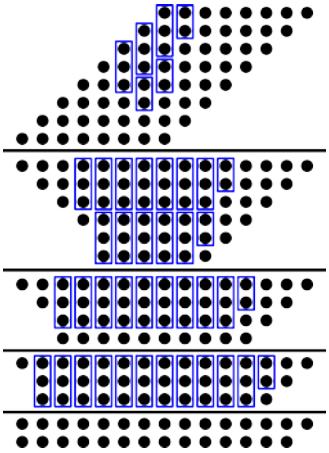

Fig. 7: Partial product compression using Dadda tree.

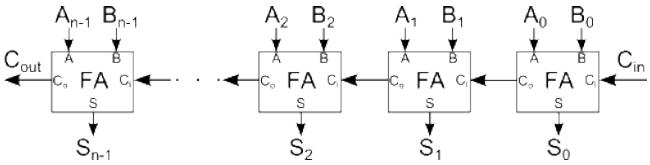

Fig. 8: Structure of an n-bit ripple carry adder.

For larger bit-widths, the CPA adder does not scale given the long critical path. An optimized variation of this architecture is the carry select adder, which uses RCA blocks with the conditional sum principle to pre-compute slices of the final result, thus reducing the time needed to operate. This adder relies on the duplication of each adder block to compute the two carry-in possibilities. Then, a multiplexer selects which block output will be used as the adder output. Consequently, the critical path becomes the logic to select the output in addition to the propagation delay of the first adder block, whose carry-out will be the control signal driving the selection circuit, as seen in Figure 9.

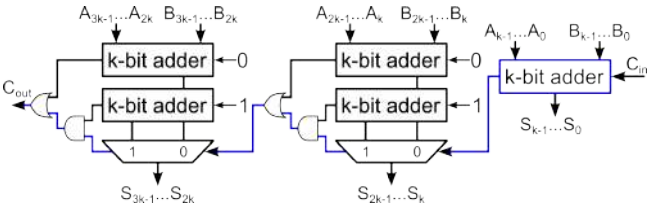

Fig. 9: Structure of a fixed-group size carry-select adder.

Variable block sizes offer better performance when compared to designs with fixed block sizes, especially if the gate delays of the targeted technology are well-known in advance. In this case, each block size can be adjusted according to the length of the selection circuit, calculated from the adder start-up to the beginning of the current block, matching both propagation times – ripple-carry group and selection chain – and effectively reducing the computation delay.

The fastest family of adders is the parallel-prefix adders (PPA), which are based on the carry look-ahead technique, enabling the computation of all carries in parallel with a logarithmic time complexity [16]. The Kogge-Stone adder [17] proposes a regular architecture layout that seeks to calculate

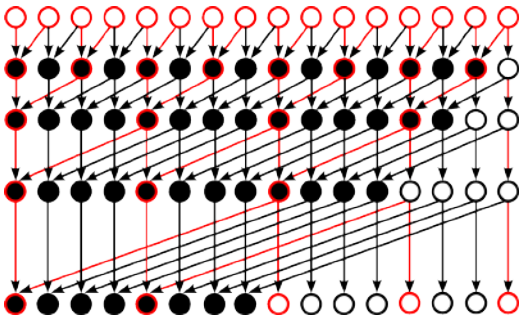

Fig. 10: Carry tree of a 16-bit Kogge-Stone adder.

each carry value as soon as possible while keeping a constant fanout for both black and white processors, as shown in Figure 10. The black processors implement the propagate-generate functions according to Equations 1 and 2, while the white processor only forwards the input data.

$$P_{out} = P_{in}\widehat{P_{in}} \tag{1}$$

$$G_{out} = G_{in} + (P_{in}\widehat{G_{in}}) \tag{2}$$

In the PPA design, the parallelism level of carry computation is at its maximum, and, theoretically, it represents the fastest binary adder architecture, with a temporal complexity of $O(log(n))$. However, the massive hardware replication and high quantity of interconnections might induce routing problems such as multiple metal layers and higher capacitance due to the number of grouped wires, degrading the final timing.

III. The RTLGen Framework

Recursive mathematical equations describe particular arithmetic architectures, and/or they demand extensive modifications when extending the circuit bit-widths. Hardware description languages do not easily implement this genericity, and they often require complex construction and advanced language features. Despite the development of the available interpreters integrated into the current electronic design automation (EDA) tools, they still do not process this type of design efficiently, leading to more unsatisfactory design results.

To address this problem, we propose a VHDL-based arithmetic circuit generator framework, named RTLGen, that moves the design definition complexity to another working space with a higher abstraction level. It enables the generation of hierarchical or flat designs without any advanced language features. Further, it simplifies the circuit composition from other blocks as they can be built independently and then integrated seamlessly. The framework is freely available and can be used by the community to generate custom arithmetic circuits using the following website: `http://lmgrocha.pythonanywhere.com`.

Since the generated circuits are created using simple VHDL constructs in a hierarchical design, the framework does not rely on any technology node, target platform, or synthesis tool vendor. This interoperability is especially useful for digital designers that must have granular control of the

design – like critical applications such as cryptography circuits – or must prototype the hardware in an FPGA platform before moving to the ASIC flow.

Each generated circuit is contained in a single file with all the necessary components, instances, and specifications to describe a hierarchical VHDL design. It removes the necessity of using custom libraries or language constructs that require pre-processing to determine the number of instances, signal width, etc. Additionally, each architecture is accompanied by a testbench that enables design verification as well as the generation of dump files needed for a realistic power estimation using commercial synthesis tools. Figure 11 illustrates the generation flow for a given set of architecture parameters using the RTLGen framework.

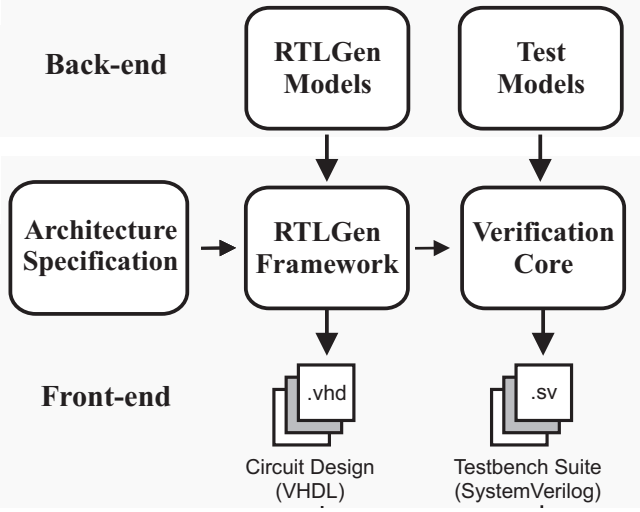

Fig. 11: Circuit generation flow of the RTLGen

A. *Framework architecture*

Creating an abstraction for HDL-described circuits to generate reusable designs is not trivial. Hence, to tackle this issue, the RTLGen framework is divided into back-end and front-end engines. Such modularity simplifies the framework to be extended, and to support new features and design constructs attached to a simple interface to generate the arithmetic circuits.

The back-end engine provides the language construction to describe the architectures and the basic arithmetic modules, enabling the integration of new components such as partial product encoders, compression trees, and carry-propagating adders. This engine has three modules whose interaction is shown in Figure 12. The OPERATIONSCORE module maps all VHDL combinational and sequential statements, such as logic operations, signal attributions, and others into Python classes. It supports operands with different bit-widths and manages such RTL declarations internally.

Reusing previously described circuits is enabled by the COMPONENTCORE as it maps the previously generated designs (adders, multipliers, etc.) and specific technology-related cells into components. This module manages the automatic instantiation of the component, taking into account different data sizes and the component declaration. For the components with an associated RTL, the COMPONENTCORE also ensures that the component architecture definition is integrated into the final circuit design file.

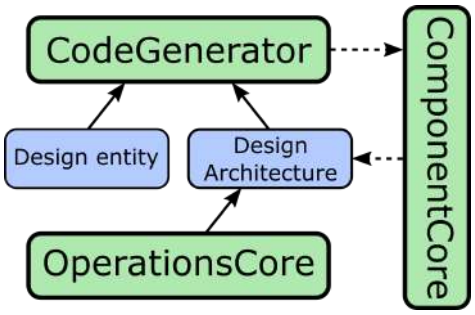

Fig. 12: Back-end framework architecture

Following the VHDL standard, every design has an associated entity, for the interface, and an architecture for the functionality. The module COREGENERATOR provides the link between the design architecture and its entity, and it manages the integration with the testbench generation. Hence, its main attributions are: (i) check all component dependencies, that is, instantiate and declare all external components used in the design; (ii) declare all internal signals; (iii) create all combinational operations according to the design model described using the OPERATIONSCORE functionalities, and (iv) instantiate the test mechanism according to the parameters specified for the current design.

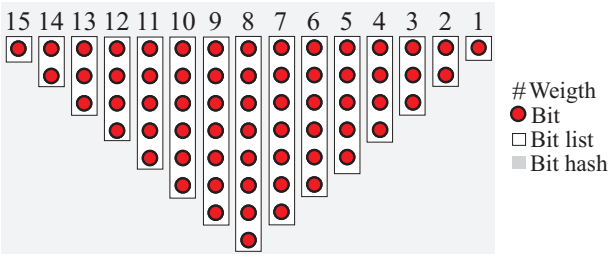

Fig. 13: Weight-aligned bit-hash for signal management

Once the basic design modules are described using the framework, the front-end engine provides a simplified view of such modules, enabling the generation of complete arithmetic architectures. A key feature in the proposed engine is the bit-hash data structure similar to the one proposed in [9] to handle all signal links among submodules used in the design. This structure operates in a weight-aligned view, i.e., each column has a list of signals that are indexed by its bit-weight, as illustrated in Figure 13. This approach provides a flexible construction to handle the signals that interconnect the circuit components, which is quintessential for circuits like compression trees.

To illustrate how an arithmetic circuit can be described using RTLGen, Algorithm 1 shows how to generate a multiplier using the functions provided by the front-end framework. Line 2 builds the hardware that encodes the partial products (PPs) from the data inputs according to the implemented algorithms (Booth, Radix-2^m, and so on). All generated signals representing the PPs are added to the bit-hash in line 3 of the algorithm. The logic operations and compo-

nents employed for PP generation are added to the circuit architecture in lines 5 and 6. Lines 8 to 15 process the bit-hash and reduce it to only two elements in each weight following a tree compression algorithm, such as Dadda or Wallace. Finally, line 16 instantiates a carry propagating adder to calculate the final result. When all these steps are finished, the framework enables the VHDL file generation to describe the entire circuit.

Algorithm 1 Multiplier Generation Algorithm

Input: List of operands (A, B) and the desired algorithm for partial product generation, compression tree and carry propagating adder;
Output: Data structure containing all VHDL components and operations to realize the multiplier;

```
1: Multiplier ← new Architecture
2: PPG ← GeneratePP(PPGAlg, A, B)
3: BitHash ← GetBitHash(PPG)
4: for Operation, Component in PPG do
5:     AddOperation(Multiplier, Operation)
6:     AddComponent(Multiplier, Component)
7: end for
8: while max(LenCol(BitHash)) > 2 do
9:     Tree ← CompressTree(CompressAlg, BitHash)
10:    BitHash ← GetBitHash(Tree)
11:    for Operation, Component in Tree do
12:        AddOperation(Multiplier, Operation)
13:        AddComponent(Multiplier, Component)
14:    end for
15: end while
16: Adder ← new CPAdder(AdderType,BitHash)
17: AddComponent(Multiplier, Adder)
```

The Multiplier structure may now be printed to a regular text file that will contain the VHDL description.

According to the algorithm, at each iteration, the signals are removed from the bit-heap and attached to the compression cells. The signals that are not subject to compression at this point will be ignored. This approach prevents the carry propagation within the same stage, i. e., the delay in each stage is guaranteed to be only one compressor cell. Before the iteration ends, all compressors outputs are inserted back into the bit-heap. If there is at least one container whose signal list is longer than two, a new iteration starts, otherwise the compression stops.

B. Automatic testbench generation

Another embedded feature is the support for automatic testbench generation. The framework generates SystemVerilog test files that use the state-of-the-art Universal Verification Methodology (UVM) [18], which is currently supported by all major EDA tools. Further, it exploits the HDL language capabilities to generate dump files during simulation, enabling a realistic power estimation in the synthesis tool.

The front-end engine provides methods to set up the pass/fail checks from a collection of inputs, outputs, and equations that describe the circuit behavior. This approach simplifies the verification of adders, multipliers with or without carry-save outputs, approximate arithmetic operators, and others. It has two verification strategies:

- **All-automatic strategy:** the simulation tool randomly generates input vectors and feeds them to the circuit. The pass/fail equations use these values to calculate the golden result to be, then, compared against the circuit output.

- **Semi-automatic strategy:** the user provides stimuli files containing both inputs and outputs. It is useful when real case vectors (images, videos, waves, etc.) are needed to characterize the device under test (DUT) or when the outputs cannot be directly described by a simple logical or arithmetical equation as in the approximate operators, for instance.

When setting all testbench parameters, the framework generates the testbench suite comprising two files. The interface file instantiates the DUT and binds it to the stimuli generator. Furthermore, it controls the simulation clock period as it is fundamental for simulations with post-synthesis netlists. The second file contains the verification core, which is responsible for adjustments in the UVM environment to integrate the pass/fail tests. Depending on the adopted verification strategy, the core will either contain a file reading mechanism that provides external stimuli or some routines that generates random values for the input vectors. Additionally, it controls the number of tests to be executed, which is mandatory for an all-automatic strategy, or it stops the test once all external file values have been read.

IV. Framework Analysis

Embedding the UVM verification suite into the proposed framework guarantees that all described designs are correct. Therefore, the framework evaluation consists of analyzing the improvements in the project time-to-market and the team resources optimization. To evaluate the framework benefits, consider first a small set of algorithms for the three components of a multiplier. Hence, the scope is limited to three partial product generation algorithms (Modified-Booth, modified version of the Radix-2^m [5] and Baugh-Wooley), two compression algorithms (Wallace and Dadda), and three recombination line adders (synthesis tool-inferred adder, carry select and Kogge-Stone). We also included the tool-inferred multiplier using the * operator in VHDL.

Some architectures may be more suitable for smaller data sizes, while others might scale better in larger datapaths. Hence, we generated multipliers of 8, 16, 32, and 64 bits, considering all the aforementioned components. This task would be impractical if they were to be implemented by hand, even for an experienced designer. Using the framework, however, this task is easily accomplished in seconds, reducing the time and workload on the design team. Further, this exploration helps on the characterization of the standard cell library in which the designs are synthesized, given their different logic structures.

A. Power Extraction Methodology

Accurate power extraction requires real input vectors to excite the circuit inputs because the probabilistic switching

Table I.: Circuit Speed, Area and Power Dissipation Comparison @ Maximum Frequency

			Wallace Tree				Dadda Tree			
Size	**Mult.**	**Adder**	F_{max} (MHz)	C. Area (μm^2)	D. Power (μW)	T. Power (μW)	F_{max} (MHz)	C. Area (μm^2)	D. Power (μW)	T. Power. (μW)
8×8	Array-Uns	**Tool**	452.7	1890.2	749.0	751.5	499.7	1962.0	917.7	920.2
		C-Select	451.3	2096.1	823.6	826.5	485.8	2212.6	1012.8	1015.8
		K-Stone	451.8	2226.6	870.4	873.5	412.1	1855.4	721.8	724.2
	M-Booth	**Tool**	462.5	2450.8	1158.1	1161.7	501.0	2134.6	1273.7	1276.7
		C-Select	467.3	2647.3	1243.6	1247.6	497.2	2260.4	1171.5	1174.7
		K-Stone	475.3	2840.8	1252.5	1256.7	495.7	2435.2	1263.7	1267.3
	Radix-4	**Tool**	471.7	3571.4	1372.8	1378.0	475.0	2993.1	1195.9	1200.0
		C-Select	443.4	3138.2	1177.5	1181.8	452.0	2752.9	1121.8	1125.6
		K-Stone	454.5	2918.2	1129.9	1134.0	476.2	3288.0	1243.8	1248.6
	B-Wooley	**Tool**	451.8	1763.8	803.7	806.0	489.6	2068.0	979.0	981.7
		C-Select	452.2	2135.6	957.6	960.6	469.6	2126.3	1001.4	1004.3
		K-Stone	454.5	2032.2	945.7	948.6	463.1	1966.6	913.9	916.5
	S. Tool	N/A	401.7	1612.0	709.5	711.6				
16×16	Array-Uns	**Tool**	380.5	8904.9	2531.5	2542.8	383.3	7100.1	2274.1	2282.5
		C-Select	352.4	9483.8	2577.6	2589.8	356.4	7664.8	2221.5	2230.8
		K-Stone	357.1	8637.2	2356.5	2367.4	375.5	8087.0	2547.4	2557.5
	M-Booth	**Tool**	396.9	8091.7	2898.2	2909.6	403.0	8522.3	3282.8	3294.6
		C-Select	392.8	9111.4	3265.4	3278.4	398.1	8263.8	3202.1	3213.9
		K-Stone	390.2	9571.1	3411.4	3425.2	395.1	8700.1	3207.9	3220.1
	Radix-4	**Tool**	365.0	12600.1	3395.8	3412.1	365.4	10657.4	3161.4	3174.7
		C-Select	357.1	13119.6	3432.7	3450.0	366.5	12060.4	3417.9	3433.7
		K-Stone	352.4	12679.2	3381.6	3398.1	367.8	12212.7	3478.1	3494.0
	B-Wooley	**Tool**	381.3	9107.3	2633.1	2644.7	374.4	6505.7	2151.0	2158.5
		C-Select	366.0	9594.4	2789.6	2801.9	372.1	7911.3	2532.3	2542.2
		K-Stone	369.3	9814.0	2794.1	2806.9	370.4	7437.6	2369.6	2378.5
	S. Tool	N/A	395.2	7142.2	2684.2	2693.9				
32×32	Array-Uns	**Tool**	318.1	35870.1	7845.9	7890.1	322.8	28901.1	7288.1	7321.3
		C-Select	290.9	35885.2	7230.9	7274.4	297.9	31844.8	7165.4	7203.7
		K-Stone	292.6	36265.3	7264.9	7308.5	316.2	34193.1	8179.1	8220.5
	M-Booth	**Tool**	321.4	29599.4	8119.4	8159.4	331.5	26371.3	7755.7	7789.5
		C-Select	308.8	31203.6	8105.6	8149.0	322.6	28861.6	8008.7	8048.3
		K-Stone	311.3	31944.6	8241.3	8284.6	324.5	29484.5	8636.5	8675.6
	Radix-4	**Tool**	300.2	44667.0	9564.8	9618.6	293.8	36141.6	8164.8	8204.8
		C-Select	275.5	43820.4	8702.3	8754.7	289.6	40287.0	8973.6	9021.1
		K-Stone	285.3	43739.8	8921.2	8973.0	293.2	40943.2	9123.8	9172.1
	B-Wooley	**Tool**	311.1	35196.2	7586.0	7629.1	325.7	30972.8	7771.0	7807.7
		C-Select	283.1	34287.8	6810.8	6852.2	298.1	32492.7	7336.2	7375.3
		K-Stone	282.6	32685.1	6530.9	6570.1	318.2	34278.9	8238.2	8280.2
	S. Tool	N/A	309.3	22642.9	6596.9	6614.5				
64×64	Array-Uns	**Tool**	267.1	133899.5	23567.4	23725.3	277.2	107101.3	20835.5	20954.0
		C-Select	238.1	128600.2	19900.2	20049.4	250.0	110924.8	19177.5	19300.1
		K-Stone	259.7	139051.6	23674.0	23838.7	272.3	117946.4	23352.0	23486.3
	M-Booth	**Tool**	268.9	102998.0	21533.9	21663.3	281.3	95656.1	22675.2	22791.1
		C-Select	243.7	104598.0	19983.4	20115.8	255.7	98441.2	20418.2	20543.2
		K-Stone	260.6	109949.3	22333.9	22472.3	268.9	99602.4	22138.7	22262.3
	Radix-4	**Tool**	262.1	163242.7	29093.1	29284.9	258.1	145539.7	26902.6	27067.9
		C-Select	246.1	172206.8	28994.0	29202.1	242.5	159011.8	26542.3	26729.1
		K-Stone	258.2	177336.1	32195.2	32406.6	257.4	168142.0	31122.2	31321.4
	B-Wooley	**Tool**	267.3	128695.8	22744.6	22894.6	281.6	114199.8	23171.1	23301.5
		C-Select	243.5	130369.7	20772.5	20925.5	250.0	113492.1	19598.3	19726.8
		K-Stone	254.2	129878.8	21729.9	21881.4	270.0	115556.5	22140.9	22270.7
	S. Tool	N/A	267.5	85302.4	18903.8	19004.9				

activity used as default by synthesis tools is usually too pessimistic since they do not model the effects of signal propagation on the power dissipation accurately. These stimuli vectors are obtained by simulating the synthesized netlist using a testbench. Further, it is necessary to add the interconnection delay through the Standard Delay Format (SDF) file, which is generated by the synthesis tool. The simulation tool dumps the switching activity into stimuli files like Value Change Dump (VCD), Toggle Count Format (TCF), or Switching Activity Interchange Format (SAIF). Finally, the synthesis tool runs again but using as input the previously generated netlist and the stimuli file for the power estimation.

Therefore, we use a methodology for precise power dissipation extraction with the following steps [19]. First, all designs are synthesized with the Cadence Genus Synthesis [20] tool with the Physically-Aware Layout Estimation (PLE) mode, which includes an estimation of the wire length and the effects of these wires in terms of area, power dissipation and critical path. Then, stimuli files are generated through gate-level netlists simulations with SDF files using the Cadence Incisive Simulation Tool. Finally, the synthesis tool employs this stimuli file as the switching activity profile, as summarized in Figure 14.

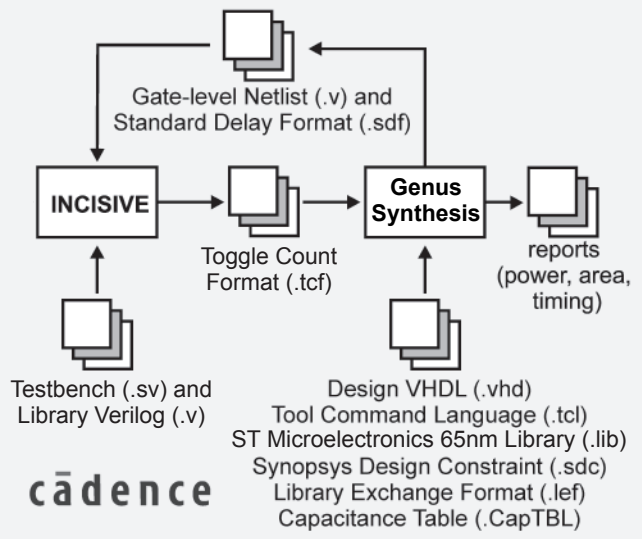

Fig. 14: Synthesis and simulation flow.

B. *Synthesis Results*

All multipliers were synthesized using the Cadence Genus Synthesis solution with an ST Microelectronics 65 nm standard cell library operating at 1.0V supply voltage. Each multiplier size was synthesized for a specific frequency range to find the maximum operating frequency within that range. As our focus relies on exploring the versatility of the RTLGen framework, the synthesis tool was configured to use the standard synthesis run settings.

Table I shows the maximum frequency (F_{max}), circuit cell area (C. Area), dynamic power dissipation (D. Power), and total power dissipation (T. Power) for all the combinations of partial products processing, compression trees, and carry propagating adders. In this analysis, the synthesis tool-generated multiplier is used as a reference model to compare the power dissipation, circuit area, and timing trade-offs. The results shown are related to the maximum frequency of each circuit combination.

Results show that although the partial product algorithm has a meaningful impact on the overall circuit characteristic, the compression tree has a more substantial impact in all aspects (speed, area, and power). Dadda tree-based multipliers outperformed the Wallace tree-based multipliers in almost all cases. On average, Dadda-based circuits are 3.3% faster than their counterparts with a 9.3% smaller cell area, even at higher speeds.

Regarding the partial product encoders, the Booth multiplier performance surpassed the other algorithms in all aspects. The good performance is attributed to all the optimizations integrated into this architecture, from the LSB pre-computation to the sign propagation optimization. It is worth mentioning that the Radix-4 multiplier did not present a good performance as it is a tweak version if compared to the original version proposed in [5].

Another remark concerns the Kogge-Stone-based multipliers as, theoretically, they should be the fastest circuits. Given that the synthesis tool estimates the wire effects, the wire congestion can become a bottleneck and penalize the circuit performance, as can be seen in some cases in Table I. For instance, the 16-bit Booth multiplier using Dadda tree and carry select adder is slightly faster and is 5% smaller than the version using the Kogge-Stone adder.

Aiming for a direct quality of results comparison, we synthesized each set of multipliers with the same size aiming a specific operational frequency, i.e., the target synthesis frequency of multipliers with input size of 8, 16, 32 and 64 bits are 400 MHz, 200 MHz, 133 MHz, and 100 MHz, respectively. The results are shown in Table II and illustrate the cell area and power dissipation of each multiplier. In this case, the selected frequencies are considerably lower than the maximum attainable frequency, which reduces the fan-out requirements to cope with timing constraints. For all cases, the multiplier inferred by the synthesis tool has worse results in terms of area and power when compared to the multipliers generated by the RTLGen framework.

Further, the results in Table II follow the same trend as the ones presented in Table I. Dadda-based multipliers have a smaller area and power dissipation than Wallace-based multipliers in nearly all cases. The benefits of RTLGen can clearly be seen in multipliers of 32 and 64 bits as the synthesis tool struggles to find a suitable architecture for that particular frequency target. For instance, the 32-bit tool-inferred multiplier has a slightly smaller area than the Baugh-Wooley version with Wallace tree and tool-inferred adder, yet it dissipates nearly two times more power. The increased energy dissipation in the tool-inferred multiplier is more prominent in the 64 bits version, as it dissipates nearly four times more than the best RTLGen-generated multiplier (Baugh-Wooley, Dadda Tree, tool-inferred adder).

V. Conclusions

This paper presented a highly flexible framework to generate RTL designs for arithmetic operators. The front-end and back-end engines provide the modularity and extensibility

Table II.: Synthesis QoR Comparison at same frequency.

Size	Mult.	Adder	Wallace Tree C. Area (μm^2)	Wallace Tree D. Power (μW)	Wallace Tree T. Power (μW)	Dadda Tree C. Area (μm^2)	Dadda Tree D. Power (μW)	Dadda Tree T. Power. (μW)
8×8[1]	Array-Uns	**Tool**	1566.8	595.4	597.3	1635.4	601.0	603.0
		C-Select	2057.1	761.4	764.2	2225.6	774.9	778.0
		K-Stone	1880.8	674.7	676.9	2350.9	727.3	730.6
	M-Booth	**Tool**	1603.2	675.8	677.7	2261.0	888.9	892.0
		C-Select	2195.4	941.7	944.7	2478.3	994.9	998.3
		K-Stone	2019.7	812.2	814.7	2516.3	954.3	957.9
	Radix-4	**Tool**	2136.7	773.6	776.2	2724.3	889.0	892.6
		C-Select	2619.8	1037.0	1040.4	3125.2	1134.4	1138.8
		K-Stone	3086.7	1062.7	1066.7	2944.2	1017.9	1021.9
	B-Wooley	**Tool**	1694.2	669.9	671.9	1711.8	720.2	722.5
		C-Select	2160.6	924.3	927.3	2183.0	826.9	829.8
		K-Stone	2050.4	788.5	791.2	2433.1	815.7	819.2
	S. Tool	N/A	1663.5	814.3	816.5			
16×16[2]	Array-Uns	**Tool**	4346.7	840.6	844.6	4599.4	786.3	790.2
		C-Select	4765.3	933.7	937.8	5181.8	844.1	848.8
		K-Stone	3933.3	872.7	875.0	4397.6	783.7	786.6
	M-Booth	**Tool**	4318.1	1072.1	1076.1	4694.6	1123.1	1127.6
		C-Select	4705.0	1176.5	1180.7	4887.5	1102.7	1107.2
		K-Stone	4471.0	1121.0	1124.3	4784.5	1223.0	1226.9
	Radix-4	**Tool**	5542.2	1001.4	1006.0	6408.0	1031.8	1037.9
		C-Select	6128.7	1055.9	1061.5	6526.5	1081.2	1087.1
		K-Stone	5386.7	1034.3	1037.7	5303.5	1029.7	1032.9
	B-Wooley	**Tool**	4339.9	814.4	818.0	4637.9	795.4	799.3
		C-Select	4585.9	870.7	874.5	5157.9	861.8	866.5
		K-Stone	4001.9	909.9	912.3	4422.6	815.6	818.5
	S. Tool	N/A	4004.0	934.6	938.4			
32×32[3]	Array-Uns	**Tool**	13600.1	1535.8	1543.7	15838.2	1604.2	1615.2
		C-Select	16759.6	2120.6	2133.4	18066.4	1825.7	1840.1
		K-Stone	14791.4	2344.6	2352.6	15358.2	1972.4	1980.6
	M-Booth	**Tool**	16138.2	2045.9	2059.5	16860.0	2275.0	2289.4
		C-Select	17155.3	2736.0	2751.0	18426.2	2485.7	2502.6
		K-Stone	16293.2	2952.7	2964.8	17084.1	2640.7	2653.3
	Radix-4	**Tool**	18439.2	1867.0	1877.3	19914.4	1912.3	1925.0
		C-Select	21792.2	2176.3	2194.0	22954.4	2238.1	2256.7
		K-Stone	19338.8	2480.2	2490.2	19615.4	2198.2	2208.2
	B-Wooley	**Tool**	13229.8	1565.5	1572.7	15778.4	1596.1	1607.0
		C-Select	16574.5	1921.3	1933.9	18015.4	1840.8	1855.2
		K-Stone	14455.5	2328.1	2335.6	15116.9	1978.5	1986.5
	S. Tool	N/A	13136.2	3088.0	3098.9			
64×64[1]	Array-Uns	**Tool**	51385.9	4012.2	4038.2	54114.3	4132.3	4160.6
		C-Select	64745.7	5061.8	5111.1	66808.6	4800.7	4851.4
		K-Stone	54457.0	5780.3	5807.1	56538.5	5226.9	5254.9
	M-Booth	**Tool**	58032.0	6450.0	6495.0	64734.3	5765.4	5820.4
		C-Select	64599.1	6934.9	6991.5	69560.9	6523.5	6587.4
		K-Stone	63822.7	8074.5	8123.3	62884.6	7570.1	7616.8
	Radix-4	**Tool**	70862.5	4695.7	4734.2	72205.6	4660.5	4701.8
		C-Select	86803.6	6144.1	6210.1	85311.2	5740.2	5803.9
		K-Stone	73161.9	6452.3	6488.9	73372.5	5977.2	6013.8
	B-Wooley	**Tool**	51506.5	4158.7	4185.0	54276.0	4054.3	4082.6
		C-Select	63635.5	4849.8	4896.9	66435.2	4820.7	4870.4
		K-Stone	54552.2	5767.3	5794.2	56642.0	5170.1	5198.2
	S. Tool	N/A	56331.6	16262.2	16317.3			

[1] Target frequency 400 MHz [2] Target frequency 200 MHz [3] Target frequency 133 MHz [4] Target frequency 100 MHz

needed to support new algorithms that may be used in arithmetic operations, such as new compression trees, pipelining, etc. Due to the algorithmic-level description of circuits and the automatic testbench generation system, the framework proves to be a very efficient tool for digital designers to explore design alternatives to fulfill the project requirements.

Several architectures were generated through the combination of partial product generators, compressor trees, and

recombination adders and synthesized in industrial technology for QoR analysis of such designs. Results showed that state-of-the-art synthesis tools do not have optimized multiplier-aware mapping algorithms, and the framework provided several design alternatives that lead to improved quality of results.

VI. Acknowledgements

The authors would like to thank IFRS, CNPq, CAPES and Fapergs Brazilian agencies for financial support to our research.

References

[1] S. Sabeetha, J. Ajayan, S. Shriram, K. Vivek, and V. Rajesh, "A study of performance comparison of digital multipliers using 22nm strained silicon technology," in *2015 2nd International Conference on Electronics and Communication Systems (ICECS)*, Feb 2015, pp. 180–184.

[2] L. Z. Pieper, E. A. C. da Costa, and J. C. Monteiro, "Combination of radix-2mmultiplier blocks and adder compressors for the design of efficient 2's complement 64-bit array multipliers," in *2013 26th Symposium on Integrated Circuits and Systems Design (SBCCI)*, Sep. 2013, pp. 1–6.

[3] C. S. Wallace, "A Suggestion for a Fast Multiplier," *IEEE Trans. on Electronic Computers*, vol. EC-13, no. 1, pp. 14–17, 1964.

[4] R. C. H. S. A. Goldovsky, B. Patel M. Schulte and G. Burns, "Design and implementation of a 16 by 16 low-power two's complement multiplier," in *IEEE International Symposium on Circuits and Systems. ISCAS 2000, Geneva.*, vol. 5, 2000, pp. 345–348 vol.5.

[5] E. Costa, S. Bampi, and J. Monteiro, "A new architecture for signed radix-2m pure array multipliers," in *Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 2002, pp. 112–117.

[6] L. Dadda, "Some Schemes for Parallel Multipliers," *Colloque sur l'Algèbre de Boole*, 1965.

[7] M. Själander and P. Larsson-Edefors, "The Case for HPM-Based Baugh-Wooley Multipliers," Chalmers University of Technology, Göteborg, Sweden, Tech. Rep. 08, 2008.

[8] F. de Dinechin and B. Pasca, "Designing Custom Arithmetic Data Paths with FloPoCo," *IEEE Design Test of Computers*, vol. 28, no. 4, pp. 18–27, 2011.

[9] N. Brunie, F. de Dinechin, M. Istoan, G. Sergent, K. Illyes, and B. Popa, "Arithmetic core generation using bit heaps," in *2013 23rd International Conference on Field programmable Logic and Applications*, Sept 2013, pp. 1–8.

[10] L. M. G. Rocha, G. Paim, R. Ferreira, E. Costa, and S. Bampi, "Framework-based arithmetic core generation to explore ASIC-based parallel binary multipliers," in *2017 24th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2017, pp. 478–481.

[11] G. W. Bewick, "Fast Multiplication : Algorithms and Implementation," Ph.D. dissertation, Stanford University, 1994.

[12] M. Hatamian and G. Cash, "A 70-MHz 8-bit x 8-bit parallel pipelined multiplier in 2.5-μm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 21, no. 4, pp. 505–513, 1986.

[13] A. D. Booth, "A signed binary multiplication technique," *Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

[14] O. Macsorley, "High-Speed Arithmetic in Binary Computers," *Proceedings of the IRE*, vol. 49, no. 1, 1961.

[15] A. A. Farooqui and V. G. Oklobdzija, "General data-path organization of a MAC unit for VLSI implementation of DSP processors," *ISCAS '98. Proceedings of the 1998 IEEE International Symposium on Circuits and Systems*, vol. 2, pp. 260–263, 1998.

[16] S. Knowles, "A family of adders," *Proceedings 15th IEEE Symposium on Computer Arithmetic*, pp. 277–284, 2001.

[17] P. M. Kogge and H. S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Trans. on Computers*, vol. C-22, no. 8, pp. 786–793, 1973.

[18] Accellera Organization, "Universal Verification Methodology (UVM)," 2012.

[19] G. Paim, L. M. G. Rocha, G. M. Santana, L. B. Soares, E. A. C. da Costa, and S. Bampi, "Power-, Area-, and Compression-Efficient Eight-Point Approximate 2-D Discrete Tchebichef Transform Hardware Design Combining Truncation Pruning and Efficient Transposition Buffers," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, pp. 680–693, 2019.

[20] Cadence, "Genus Synthesis Solution," http://www.cadence.com, 2015.