CLAST: Cross-Layer Approximate High-Level Synthesis with Configurable Approximate Three-Operand Adders

Jian Shi[1], Wenjing Zhang[1], and Weikang Qian[1,2]
[1]University of Michigan-SJTU Joint Institute, and [2]MoE Key Lab of AI, Shanghai Jiao Tong University, Shanghai, China
Emails: {timeshi, kurumi995, qianwk}@sjtu.edu.cn

***Abstract*—Approximate high-level synthesis (HLS) technique has been proposed in recent years to produce approximate designs automatically from a high-level description. All existing approximate HLS methods work on two-operand approximate units. In this paper, we propose CLAST, a cross-layer approximate high-level synthesis using configurable three-operand approximate adders. The experimental results show that CLAST outperforms previous HLS tools by achieving 57.0% improvement in area-delay product, while maintaining a high accuracy.**

I. Introduction

Approximate computing is a new computing paradigm aiming at introducing small errors to achieve large improvement in circuit area, delay, and power consumption. Many studies design approximate adders for error-tolerant applications. Some of them focus on two-operand adders [1], [2], while others propose adders with more operands [3] or adder trees [4]. However, these works often overlook system-level optimization, which offers opportunities to further simplify the adder design.

Besides the works on approximate adder design, which is at the circuit level, some other works design approximate circuits at the system level. This includes approximate high-level synthesis (HLS) [5], which synthesizes an approximate circuit from an application's high-level description. Examples of this approach include ABACUS [6], which generates approximate RTL descriptions, and AxHLS [7], which selects appropriate approximate units in RTL designs. However, none of these HLS methods utilize approximate multi-operand adders.

In this paper, to address the above issues, we propose CLAST, a cross-layer approximate high-level synthesis using configurable three-operand approximate adders.

II. Approximate Three-Operand Adder

A three-operand adder adds three N-bit input values, x, y, and z, together and produces a sum of $(N+2)$ bits. To shorten the critical path delay, we propose to cut the carry chain in three-operand adders, so that the adder is reduced to a group of separate exact three-operand sub-adders. For example, we can cut the carry chain at signal c_2 in Fig. 1 to obtain an approximate design with two three-operand sub-adders, each producing a 3-bit result. To compensate the accuracy loss due to the carry chain cut, the leftmost full adder in each sub-adder except the leftmost one is replaced by an three-input OR gate as shown in Fig. 1.

This work is supported by the National Natural Science Foundation of China under Grants T2293700, T2293701, and T2293704 and the National Key R&D Program of China under grant number 2020YFB2205501. Corresponding author: Weikang Qian.

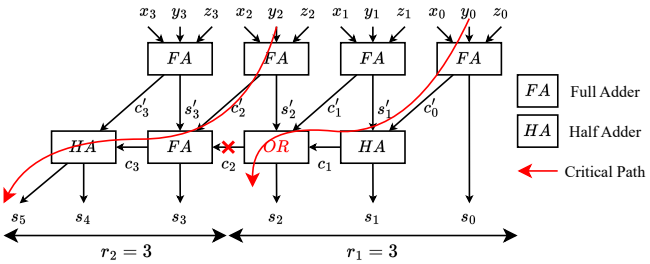

Fig. 1: An approximate three-operand adder.

Another widely-adopted design practice for approximate two-operand adder is truncating the least significant bits (LSBs), which can reduce both the area and delay of the adder [8]. With this technique, it leads to our proposed final design for a *configurable approximate three-operand adder (CToAd)*. Fig. 2 shows the architecture of an N-bit CToAd. It consists of k smaller exact three-operand sub-adders of various lengths, where the the number of the output bits of the i-th $(1 \leq i \leq k)$ sub-adder is r_i. Besides, its r_0 LSBs are truncated. The first sub-adder P_1 takes the most significant input in the truncated part to generate its carry-in signal. A CToAd can be represented by a vector $[r_k, r_{k-1}, \ldots, r_1, r_0]$ with $\sum_{i=0}^{k} r_i = N + 2$.

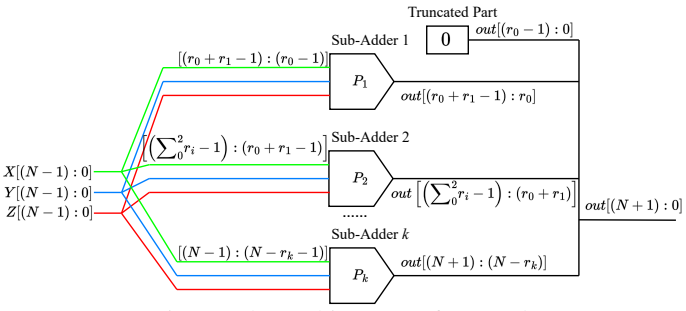

Fig. 2: The architecture of CToAd.

III. System-Level Optimization

Consider a tree of operations shown in Fig. 3(a), where all additions have two operands and are surrounded by other types of operations. Thus, CToAds cannot be applied. However, our study found that by a system-level optimization, we can convert this tree into a new one shown in Fig. 3(c), which can take advantage of CToAds to reduce area and delay. We propose a cross-layer approximate high-level synthesis (CLAST) to automatically achieve this transformation.

Given a dataflow graph (DFG), CLAST first identifies all optimizable trees in it. Each tree is optimized through the

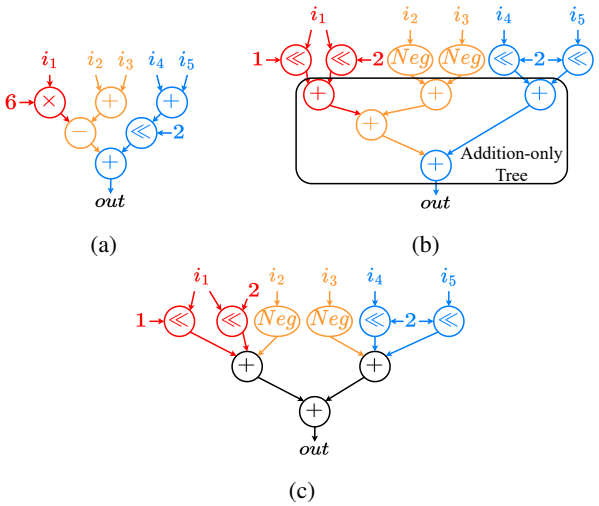

Fig. 3: Original tree and its corresponding optimization.

following conversions to create a tree with the most connected additions, which are similar to the techniques proposed in [9]:
1) *Constant Multiplication Conversion*: Constant multiplications can be beneficially converted into the addition of shifted numbers. For example, $6x$ can be rewritten as $(x \ll 1)+(x \ll 2)$, where $\ll$ denotes a left shift operation. If a binary encoding of a constant has a consecutive sequence of more than two ones, using subtractions can decrease the number of additions. For instance, constant multiplication $23x$ can be expressed as $(x \ll 4)+(x \ll 3)-x$, which is a more efficient computation than using a multiplier. The red part in Fig. 3 shows the effect of this conversion.
2) *Subtraction Conversion*: Subtractions can be converted into additions by adding the minuend to the negation of the subtrahend. Although the negation operation slightly increases area and delay, its impact is minimal. The distributive property of negation, $-(x+y)=(-x)+(-y)$, allows the re-arrangement of negating nodes to the tree's leaves, further expanding the addition-only trees. The orange part of Fig. 3 illustrates this conversion.
3) *Shift Re-arrangement*: With the above conversion of multiplication, shift operations become more prevalent in DFGs. Although these operations do not affect hardware resources, they can fragment addition-only trees. Hence, it is crucial to rearrange them to form a larger addition-only tree. The distributive property allows for this re-arrangement. For example, $(i_1+i_2) \ll 2$ can be rewritten as $(i_1 \ll 2)+(i_2 \ll 2)$. This conversion is illustrated by the blue part of Fig. 3.

As shown in Figs. 3(a) and (b), the above optimizations push non-addition nodes to the tree's boundary, constructing a large addition-only tree. After that, all leaves are grouped into triples, with any remaining leaves forming another group. For each group with more than one element, a new exact two-operand adder or a CToAd is assigned as a new node. Single-element groups are preserved for grouping in the next iteration. This process repeats until only one node remains. This results in a highly optimized DFG that can be implemented with the maximum number of CToAds. See Fig. 3(c) for an example. As CToAds are configurable, a design space exploration based on random sampling is further carried out to find a good configuration for each CToAd to achieve an optimized trade-off between accuracy and performance.

IV. Experimental Results

This section presents our experimental results. Our tests involve 8 C++ cases from S2CBench [10] and AxHLS [7]. For image processing applications, we test their effect on an image of size 512×512 and calculate the PSNR of the results. For signal processing applications, we use a one-phase sin signal with normal distribution noise as input, and choose MSE as the error metric. We compare the performance of designs generated by Xilinx Vitis HLS 2022.2 [11] and CLAST in terms of delay and area-delay-product (ADP). As shown in Table I, compared to Vitis HLS, CLAST can reduce ADP by 57.0% while maintaining a high accuracy.

TABLE I: Performance comparison of Vitis HLS and CLAST.

	Delay (ns)		ADP		ADP	Accuracy
	Vitis	CLAST	Vitis	CLAST	Improvement	
Ave8	4.51	3.44	424	282	33.4%	1.54 (MSE)
FIR	13.1	5.82	1959	786	59.9%	0.50 (MSE)
Gau3	9.10	2.95	309	77	75.3%	38.6 (PSNR)
Gau5	14.1	6.56	1669	807	51.6%	40.1 (PSNR)
Lap4	4.49	4.25	170	115	32.7%	40.3 (PSNR)
Lap8	9.40	4.65	536	200	62.7%	34.4 (PSNR)
Sharpen	4.04	4.34	85	191	-124.7%	40.1 (PSNR)
Sobel	15.8	5.92	1312	319	75.7%	30.4 (PSNR)
	Total		6464	2777	57.0%	-

V. Conclusion

In this paper, we introduce CToAd, a configurable approximate three-operand adder design, and CLAST, a cross-layer approximate HLS using CToAds. The experimental results show that CLAST outperforms Vitis HLS by achieving 57.0% improvement in ADP, while maintaining a high accuracy.

References

[1] H. Jiang, J. Han, and F. Lombardi, "A comparative review and evaluation of approximate adders," in *GLSVLSI*, 2015, p. 343–348.
[2] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *DAC*, 2012, pp. 820–825.
[3] S. Boroumand, H. P. Afshar, and P. Brisk, "Approximate quaternary addition with the fast carry chains of FPGAs," in *DATE*, 2018, pp. 577–580.
[4] S. Boroumand and P. Brisk, "Approximate adder tree synthesis for FPGAs," in *ReConFig*, 2019, pp. 1–8.
[5] B. C. Schafer and Z. Wang, "High-level synthesis design space exploration: Past, present, and future," *TCAD*, vol. 39, no. 10, pp. 2628–2639, 2020.
[6] K. Nepal, Y. Li, R. I. Bahar, and S. Reda, "ABACUS: A technique for automated behavioral synthesis of approximate computing circuits," in *DATE*, 2014, pp. 1–6.
[7] J. Castro-Godínez, J. Mateus-Vargas, M. Shafique, and J. Henkel, "AxHLS: Design space exploration and high-level synthesis of approximate accelerators using approximate functional units and analytical models," in *ICCAD*, 2020, pp. 1–9.
[8] B. K. Mohanty, "Efficient fixed-width adder-tree design," *TCAS-II*, vol. 66, no. 2, pp. 292–296, 2019.
[9] A. K. Verma, P. Brisk, and P. Ienne, "Data-flow transformations to maximize the use of carry-save representation in arithmetic circuits," *IEEE TCAD*, vol. 27, no. 10, pp. 1761–1774, 2008.
[10] B. C. Schafer and A. Mahapatra, "S2CBench: Synthesizable SystemC benchmark suite for high-level synthesis," *Embedded Syst. Lett.*, vol. 6, no. 3, pp. 53–56, 2014.
[11] *Vitis High-Level Synthesis User Guide*. Xilinx Inc., 2023.