Semi- and Fully-Random Access LUTs for Smooth Functions

Y. Serhan Gener[1], Furkan Aydin[2], Sezer Gören[3(✉)], and H. Fatih Ugurdag[4]

[1] Department of Computer Science, University of California Riverside,
Riverside, CA, USA
ygene001@ucr.edu

[2] Department of Electrical and Computer Engineering,
North Carolina State University, Raleigh, NC, USA
faydn@ncsu.edu

[3] Department of Computer Engineering, Yeditepe University, Istanbul, Turkey
sgoren@cse.yeditepe.edu.tr

[4] Department of Electrical and Electronics Engineering, Ozyegin University,
Istanbul, Turkey
fatih.ugurdag@ozyegin.edu.tr

Abstract. Look-Up Table (LUT) implementation of complicated functions often offers lower latency compared to algebraic implementations at the expense of significant area penalty. If the function is smooth, MultiPartite table method (MP) can circumvent the area problem by breaking up the implementation into multiple smaller LUTs. However, even some of these smaller LUTs may be big in high accuracy MP implementations. Lossless LUT compression can be applied to these LUTs to further improve area and even timing in some cases. The state-of-the-art in the literature decomposes the Table of Initial Values (TIV) of MP into a table of pivots and tables of differences from the pivots. Our technique instead places differences of consecutive elements in the difference tables and result in a smaller range of differences that fit in fewer bits. Constraining the difference of consecutive input values, hence semi-random access, allows us to further optimize designs. We also propose variants of our techniques with variable length coding. We implemented Verilog generators of MP for sine and exponential using conventional LUT as well as different versions of the state-of-the-art and our technique. We synthesized the generated designs on FPGA and found that our techniques produce up to 29% improvement in area, 11% improvement in timing, and 26% improvement in area-time product over the state-of-the-art.

1 Introduction

Computationally complex functions often need to be efficiently implemented in hardware so that they can be part of real-time systems. Look-Up Table (LUT) based methods (see [1] for a comprehensive survey) offer a good balance between latency and area in comparison to algebraic methods by a priori computing of the

Published by Springer Nature Switzerland AG 2020
C. Metzler et al. (Eds.): VLSI-SoC 2019, IFIP AICT 586, pp. 279–306, 2020.
https://doi.org/10.1007/978-3-030-53273-4_13

function values. LUT-based methods usually have shorter latency as compared to algebraic methods. A straight-forward LUT method, where function value f(x) is stored in (read-only) memory's location x, and the input x is tied to the address port of the memory and the output f(x) is tied to the dataOut port of the memory (without any additional logic) is what we call the Conventional LUT (ConvLUT) method shown in Fig. 1. However, for large bit widths of x, the area of a ConvLUT blows up.

What we are proposing in this work is a "lossless LUT compression method" that can shrink the hardware area of complex functions (sometimes even with improvements in latency) when applied to:

- ConvLUTs or
- the below smarter methods with multiple LUTs

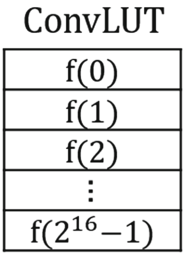


Fig. 1. LUT contents of a ConvLUT.

When a ConvLUT blows up due to large input bit width, "multi-LUT methods" (including ours) come to rescue. Although our lossless compression method can be directly applied to ConvLUT, better results are obtained if a lossy multi-LUT method is used first, and our method is applied to the LUTs within. Multi-LUT methods use some arithmetic logic to combine the values in multiple tables. These methods do not produce the exact same output as the ConvLUT method but can stay within a prescribed error range. Some such popular methods are BiPartite method (BP) [2,3], Symmetric Bipartite Method (SBTM) [4], Symmetric Table Addition Method (STAM) [5], MultiPartite method (MP) [6,7], and Hierarchical MultiPartite method (HMP) [8].

BP [2,3] uses approximation by the first two terms of the Taylor expansion of a function using two LUTs: i. Table of Initial Values (TIV) and ii. Table of Offsets (TOs). The microarchitecture in Fig. 2 becomes BP, when the three TOs are combined into a single TO. BP uses an adder to add TIV and TO outputs. TIV downsamples the function values and hence stores a subset of them (at $x0$ values), whereas TO stores the derivative times $\Delta x (= x - x0)$.

For further reduction in TO size, TO can be partitioned into multiple smaller LUTs, which thus leads to the MP method. MP combines STAM [5] and the approach in [9]. In MP, there are multiple TOs and a single TIV. Figure 2 depicts an MP

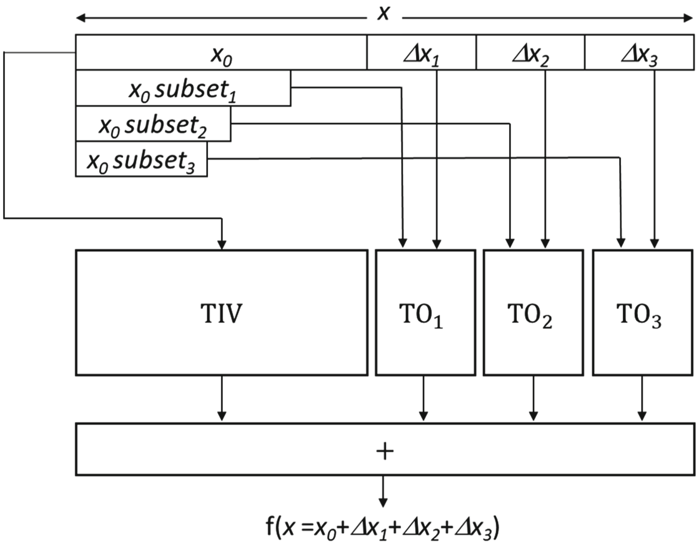


Fig. 2. MP microarchitecture.

microarchitecture with 3 TOs. A smaller Δx can work with a derivative with fewer bits, hence the smaller the Δx the narrower the $x0$ bits (subset) it uses. Each additional table reduces the total number of bits in all of the tables but then increases the combinational logic complexity - all in all reduces the total area.

In the recently proposed HMP method [8], the TIV is further decomposed into the sum of another TIV and TOs. HMP also performs global bit width optimization over all LUTs.

Figure 3 sketches the LUT compression approach within the context of MP. For any given function f, with any given input bit resolution (w_i), i.e., bit width, output resolution (w_o), and m value (which represents the number of TOs to be generated), MP can be implemented as shown in Fig. 3, where TIV$_{new}$ is a new TIV with fewer entries than TIV, while TIV$_{diff}$'s store differences of missing entries from the entries TIV$_{new}$. Our lossless LUT compression and the one in [11] can actually be applied to any LUT, hence TIV or TOs or both. However, we applied it on TIV, as there is more compression opportunity in TIV because it is bigger. Also, [11] is applied on TIV, and we wanted to compare our work to theirs.

The concept of general-purpose lossless LUT compression for hardware design was first introduced in [10]. The work in [10] proposes what we here classify as Semi-Random Access differential LUT (SR-dLUT), where differential LUT (dLUT) is a sort of TIV$_{diff}$. Note that [10] calls the dLUT in this chapter as cLUT, short for “compressed LUT”. SR-dLUT can output any LUT location within the range $[i - k, i + k]$ in a given cycle if location i is output in the previous cycle and k is the number of difference tables (dLUT). Note that there is no TIV$_{new}$ in SR-dLUT, there are only difference tables (cLUT in the case of [10]).

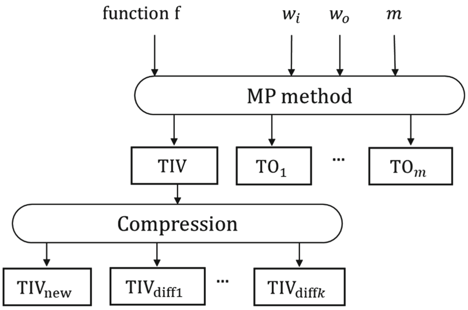


Fig. 3. MP method combined with LUT compression for TIV.

In [11], a lossless LUT compression approach was proposed and was used to compress the TIV of MP method. Two schemes were proposed in [11], namely, 2-table decomposition and 3-table decomposition. We abbreviate them as 2T-TIV and 3T-TIV, respectively. Details of both methods are discussed in Sects. 2.1 and 2.2. Furthermore, in [8] an improvement over the 2T-TIV method was introduced, which we call 2T-TIV-IMP and discuss in detail in Sect. 2.3.

In this chapter, we propose four lossless LUT compression methods, namely, Fully-Random Access differential LUT (FR-dLUT) and Semi-Random Access differential LUT (SR-dLUT), “Variable Length” encoded FR-dLUT (FR-dLUT-VL), and “Variable Length” encoded SR-dLUT (SR-dLUT-VL). Note that FR-dLUT and FR-dLUT-VL microarchitectures were first proposed in our earlier paper [16] and an earlier version of SR-dLUT (cLUT) was introduced in [10]. SR-dLUT-VL is unique to this chapter. Also, SR-dLUT is here used as a building block of MP. Note that although this chapter targets hardware implementation, software implementation of our proposed methods are also possible.

Section 2 below covers the details of the state-of-the-art, namely, 2T-TIV [11], 3T-TIV [11], and 2T-TIV-IMP [8] (which we call T-TIV methods), while Sect. 3 presents our proposed methods. Section 4 gives synthesis results (area, time, and area-time product) of all of the above methods and compares them, and Sect. 5 concludes the chapter.

2 State-of-the-Art

In this section, previous state-of-the-art of fully-random access lossless LUT compression are outlined, namely, T-TIV methods.

2.1 2T-TIV Microarchitecture

2T-TIV [11] decomposes TIV of the MP method into 2 LUTs, TIV_{new} and TIV_{diff} (i.e., a form of dLUT) as shown in Fig. 4, where the original TIV can be recovered from TIV_{new} and TIV_{diff} without introducing any errors.

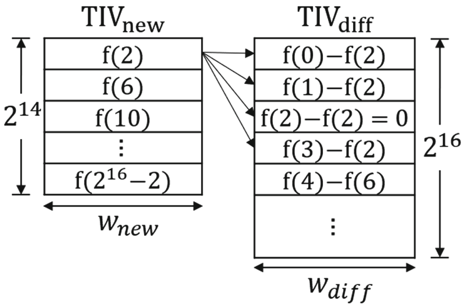


Fig. 4. LUT contents of 2T-TIV for s = 2.

In Fig. 4, we have $w_i = 16$ and $s = 2$ such that for every 2^s consecutive entries, TIV$_{new}$ stores one TIV value (i.e., the one in the middle of 2^s). That is why TIV$_{new}$ has 2^{wi-s} entries. TIV$_{diff}$ is used to store differences between the original TIV values and their corresponding entries in TIV$_{new}$. Equation (1) shows how TIV is reconstructed back from TIV$_{new}$ and TIV$_{diff}$.

$$\begin{gathered}\text{TIV}\,(i) = \text{TIV}_{new}\,(i') + \text{TIV}_{diff}\,(i)\\ \text{where } i' = integer\left(\tfrac{i}{2^s}\right)\\ \text{and } \text{TIV}_{new}\,(i') = \text{TIV}\left(i' * 2^s + 2^{s-1}\right)\end{gathered} \tag{1}$$

Note that every 2^s entries of TIV$_{diff}$ contains one zero entry (i.e., entries 2, 6, 10, etc. in Fig. 4).

2.2 3T-TIV Microarchitecture

3T-TIV [11] exploits the “almost symmetry” of function values around the entries of TIV$_{new}$. 3T-TIV has 3 LUTs as shown in Fig. 5 for a given function with $w_i = 16$ and $s = 3$. In 3T-TIV, TIV$_{new}$ serves the same purpose as in 2T-TIV. Figure 5 shows an example, where $w_i = 16$ and $s = 3$. Hence, there is one entry in TIV$_{new}$ for every $8 (= 2^{s=3})$ entries of the original TIV. The entries in TIV$_{new}$ correspond to f(4), f(12), f(20), and so on. The values of f(0), f(1), and up to f(7) need to be computed from f(4) in TIV$_{new}$. Half of those values, i.e., f(0), f(1), up to f(3) are calculated using the differences stored TIV$_{diff1}$ as in 2T-TIV, hence (2) where i' is the same as in (1).

$$\begin{gathered}\text{TIV}\,(i) = \text{TIV}_{new}\,(i') + \text{TIV}_{diff1}\left(2^{s-1} * i' + j\right)\\ \text{for } j < 2^{s-1} \text{ where } j = mod\,(i, 2^s)\end{gathered} \tag{2}$$

However, for f(4) through f(7), the second-order differences in TIV$_{diff2}$ are also added on top of TIV$_{new}$ and TIV$_{diff1}$. The value of f(7) is computed from f(1), while f(6) is derived from f(2), and so on. That can be expressed as in (3).

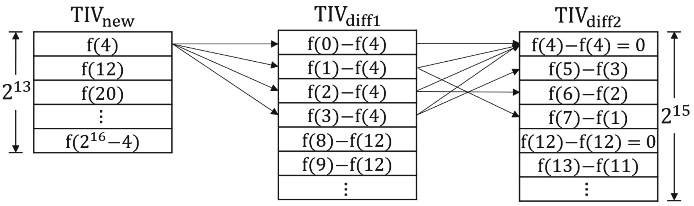


Fig. 5. LUT contents of 3T-TIV for s = 3.

$$\begin{aligned} \mathrm{TIV}\,(i) = \mathrm{TIV}_{new}\,(i') + \mathrm{TIV}_{diff1}\,\left(2^{s-1} * i' + j\right) \\ + \mathrm{TIV}_{diff2}\,\left(2^{s-1} * i' + j - 2^{s-1}\right) \text{ for } j \geq 2^{s-1} \\ \text{where } \mathrm{TIV}_{diff2}\,(k) = \mathrm{TIV}\,\left(k' + 2^{s-1}\right) \\ - \mathrm{TIV}\,\left(k' - 2^{s-1}\right) \\ \text{and } k' = 2 * integer\,\left(\tfrac{k}{s^{s-1}}\right) + mod\,\left(k, 2^{s-1}\right) \end{aligned} \tag{3}$$

In summary, when we group values of function f in groups of 2^s, the first 2^{s-1} values in every group of 2^s use TIV_{new} and TIV_{diff1}, while the second 2^{s-1} values use all three tables. Note that every 2^{s-1} entries of TIV_{diff2} contains one zero entry (i.e., entry 0, 4, 8, etc. in Fig. 5).

2.3 2T-TIV-IMP Microarchitecture

2T-TIV-IMP [8] improves 2T-TIV [11] by reducing the bit width of TIV_{new}. This is possible when every TIV_{diff} entry is summed offline with least significant w_{diff} bits of the corresponding TIV_{new} entry. As a result of this, the least significant w_{diff} bits of all TIV_{new} entries can be zeroed out, hence do not need to be stored, which leads to a bit width of $w_{new} - w_{diff}$ bits as shown in Fig. 6. On the other hand, TIV_{diff} entries become 1-bit larger (the $w_{diff} + 1$ in Fig. 6) if any TIV_{diff} entry overflows when summed with w_{diff} bits of TIV_{new}. If there is no overflow, then the bit width of TIV_{new} stays the same (w_{diff}).

This optimization not only reduces the bit width of TIV_{new} but also makes the subcircuit that sums TIV_{new} and TIV_{diff} smaller, when there is overflow resulting in 1 overlap bit, hence TIV_{diff} with $w_{diff} + 1$ bits. The optimization can even completely eliminate summing when there is no overflow during offline addition performed to compute the new TIV_{diff} values. In that case, summing TIV_{new} and TIV_{diff} can simply be realized by concatenating them.

One thing that is not addressed in [11] is that if TIV_{new} entries are truncated, TIV_{diff} entries may have to be 2-bit larger. For a guarantee on at most 1-bit larger TIV_{diff}, TIV_{new} entries have to be rounded down to $w_{new} - w_{diff}$ bits from w_{new} bits.

Note that the described improvement on 2T-TIV cannot be applied to 3T-TIV because this optimization breaks the symmetry property 3T-TIV uses.

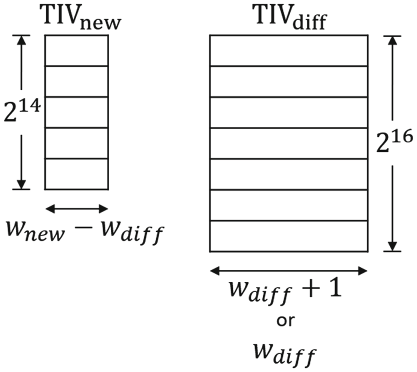


Fig. 6. 2T-TIV-IMP version of Fig. 4.

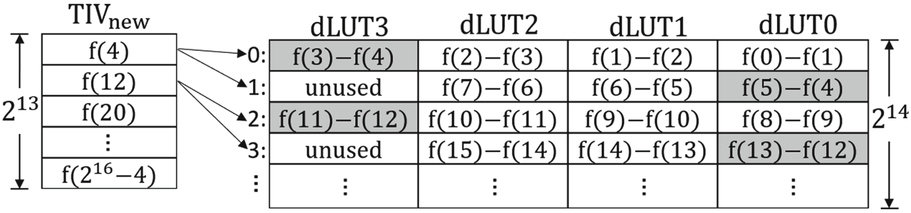
3 Proposed Methods

In this section, our proposed microarchitectures, FR-dLUT and SR-dLUT as well as their variable length coded variants, namely, FR-dLUT-VL and SR-dLUT-VL, are presented. The critical difference between our approach and the previous state-of-the-art (2T-TIV and variants) is that TIV$_{diff}$ (which we call dLUT) entries store the difference from the neighboring function value instead of the difference from the corresponding value in TIV$_{new}$ as shown in (4) and (5). This allows elimination of TIV$_{new}$ if what is desired is only semi-random access [10]. If fully-random access is desired, we still need a TIV$_{new}$ table but our dLUTs store smaller differences compared to the above explained state-of-the-art. Below, we will describe FR-dLUT, SR-dLUT, FR-dLUT-VL, and SR-dLUT-VL.

3.1 FR-dLUT Microarchitecture

FR-dLUT can be implemented with any number of dLUTs. Figure 7 shows an FR-dLUT with 4 dLUTs, which implements a function f with $w_i = 16$ and $s = 3$ (hence $2^3 = 8$ difference values per each TIV$_{new}$ entry). If TIV$_{diff}$ in 2T-TIV and variants contain differences, dLUTs in FR-dLUT, in a way, contain difference of differences. More specifically, the entries in the dLUTs that correspond to points that neighbor TIV$_{new}$ entries (shown as shaded in Fig. 7) contain the same values as in 2T-TIV, while the other differences are equal to the differences of neighboring TIV$_{diff}$ entries in 2T-TIV (see (4) and (5)).

This allows us to store smaller values (i.e., fewer bits) in our difference tables. However, there is a tradeoff since the summation circuit gets bigger because we have to sum multiple difference values in our case. Note that, in the actual implementation, the last dLUT (i.e., dLUT3 in Fig. 7) has half the number of entries of the other dLUTs. (If logic synthesis is used, logic minimization would do area reduction when unused entries are specified as don't cares.)



TIVnew
f(4)
f(12)
f(20)
⋮
f(2^16−4)

	dLUT3	dLUT2	dLUT1	dLUT0
0:	f(3)−f(4)	f(2)−f(3)	f(1)−f(2)	f(0)−f(1)
1:	unused	f(7)−f(6)	f(6)−f(5)	f(5)−f(4)
2:	f(11)−f(12)	f(10)−f(11)	f(9)−f(10)	f(8)−f(9)
3:	unused	f(15)−f(14)	f(14)−f(13)	f(13)−f(12)
⋮	⋮	⋮	⋮	⋮

Fig. 7. LUTs of FR-dLUT for $w_i = 16$, $s = 3$ (i.e., $\Delta = 2^{3-1} = 4$).

Equations (4) and (5) are two specific examples of how original TIV values are constructed from TIV$_{new}$ and dLUTs in FR-dLUT. They are based on the tables in Fig. 7.

$$\underbrace{\mathrm{f}(10)}_{\text{TIV}} = \underbrace{\mathrm{f}(12)}_{\text{TIV}_{new}} + \underbrace{\mathrm{f}(11) - \mathrm{f}(12)}_{\text{dLUT3}} + \underbrace{\mathrm{f}(10) - \mathrm{f}(11)}_{\text{dLUT2}} \tag{4}$$

$$\underbrace{\mathrm{f}(13)}_{\text{TIV}} = \underbrace{\mathrm{f}(12)}_{\text{TIV}_{new}} + \underbrace{\mathrm{f}(13) - \mathrm{f}(12)}_{\text{dLUT0}} \tag{5}$$

The generalized construction of TIV from TIV$_{new}$ and dLUTs for FR-dLUT method is given in (6) and (7). There are separate construction formulae for odd (7) and even (6) rows of dLUTs. Note that i' and TIV$_{new}(i')$ below use the definitions given earlier in (1). Also, in the summations of (6) and (7), if the upper index (finish index) is smaller than the lower index (start index), then the summation returns zero.

$$\begin{gathered} \mathrm{TIV}\,(i) = \mathrm{TIV}_{new}\,(i') + \textstyle\sum_{k=j}^{\Delta-1} \mathrm{dLUT}k(2i') \text{ for } j \leq 2^{s-1} \\ \text{where } j = mod\,(i, 2^s)\,, \text{ and} \\ \mathrm{dLUT}k\,(2i') = \mathrm{TIV}\,(i' * 2^s + k) - \mathrm{TIV}\,(i' * 2^s + k + 1) \end{gathered} \tag{6}$$

$$\begin{gathered} \mathrm{TIV}\,(i) = \mathrm{TIV}_{new}\,(i') + \textstyle\sum_{k=0}^{q} \mathrm{dLUT}k(2i'+1) \text{ for } j > \Delta = 2^{s-1} \\ \text{where } j = mod\,(i, 2^s)\,, q = j - 2^{s-1}, \text{ and} \\ \mathrm{dLUT}k\,(2i'+1) = \mathrm{TIV}\left(i' * 2^s + 2^{s-1} + k + 1\right) - \mathrm{TIV}\left(i' * 2^s + 2^{s-1} + k\right) \end{gathered} \tag{7}$$

The top-level of FR-dLUT microarchitecture is shown in Fig. 8 for a function f with n-bit input ($w_i = n$), hence 2^n possible output values, k-bit output resolution ($w_o = k$), Δ dLUTs ($\Delta = 2^s$), and d-bit differences. The top-level of FR-dLUT consists of 5 submodules, namely, AddressGenerator, TIV$_{new}$, a set of dLUTs, DataSelection, and SignedSummation.

If there are Δ dLUTs, where Δ is a power of 2 (to make address generation is simple), TIV$_{new}$ stores k-bit TIV(α), where $\alpha = \Delta, 3\Delta, 5\Delta, 7\Delta, \ldots, 2^n - \Delta$. The number of locations in TIV$_{new}$ is $2^{(n-1)}/\Delta$. We denote the output of TIV$_{new}$ with $mval$ (short for main value). The address line of TIV$_{new}$, named as $directAddress$, has a bit width of $n - 1 - lg\Delta$ (lg denotes $\log_2$). dLUTs store

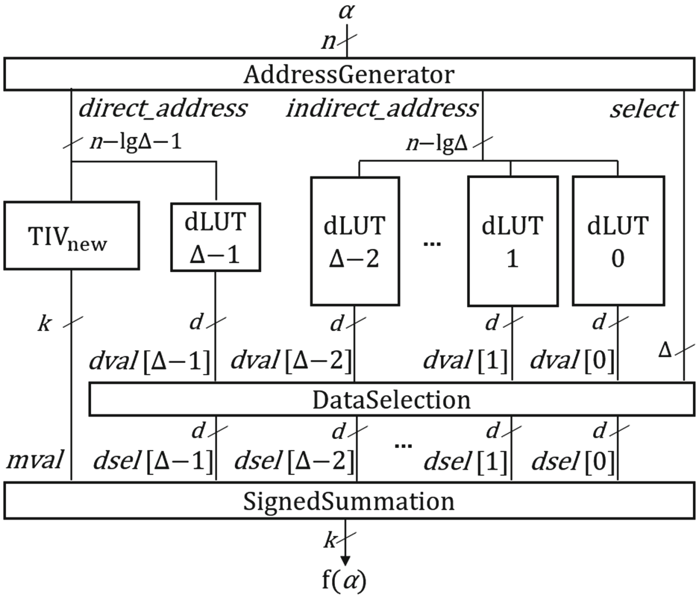


Fig. 8. Top module of FR-dLUT microarchitecture.

d-bit 2's complement differences. The number of locations in "dLUT $\Delta - 1$" is half the number of locations in the other dLUTs and is the same as in TIV$_{new}$. "dLUT of index $\Delta - 1$" shares the same address bus as TIV$_{new}$. The bit width of the address lines of other dLUTs, named as *indirectAddress*, is $n - lg\Delta$. The output of a dLUT is named *dval* (stands for "difference value").

AddressGenerator, detailed in Fig. 9, generates three signals: *directAddress*, *indirectAddress*, and *select*. For every 2Δ value in TIV, there is only one entry in TIV$_{new}$ (as well as in "dLUT $\Delta - 1$"), thus *directAddress* signal is the $n - 1 - lg\Delta$ most significant bits of the n-bit input α part-selected as $\alpha[n-1 : lg\Delta + 1]$. Moreover, for each value in TIV$_{new}$ there are two entries in each dLUT (except "dLUT $\Delta - 1$"), that is why *indirectAddress* signal is the higher $n - lg\Delta$ bits of n-bit input $\alpha(\alpha[n-1 : lg\Delta])$.

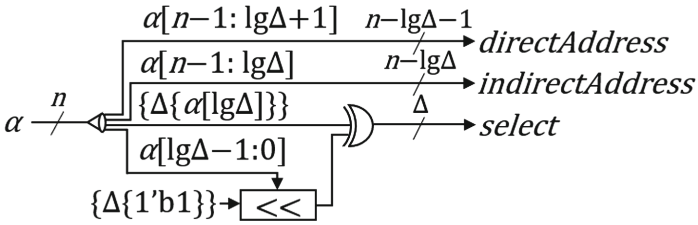


Fig. 9. AddressGenerator module.

Signal named *select* is a thermometer vector, which consists of Δ consecutive ones, each bit representing whether the corresponding *dval* will be summed (by SignedSummation module). Consider (4), which shows how we compute f(10) using the differences in the dLUTs. In the case of f(10), signal *select* is 1100, i.e., $select[3] = 1$ (the MSB) and $select[2] = 1$. Therefore, the outputs of dLUT3 and dLUT2 (find the terms in (4) in Fig. 7) are summed. Positions of signal *select*'s bits match the dLUT locations in Figs. 7 and 8, and the related figures to follow.

Now, consider f(13), which is given by (5). Its *select* vector is 0001 because only dLUT0's output is summed with TIV_{new}. If we were computing f(15), *select* would be 0111. Note that unused dLUT entries always have a *select* bit of 0. On the other hand, f(9) and f(8) would have *select* of 1110 and 1111, respectively. If the α of f(α) we are computing corresponds to an even row of dLUTs (i.e., row 0, 2, 4, etc.), then *select* is a contiguous block of 1's starting with the LSB. On the other hand, If the α corresponds to an odd row (i.e., row 1, 3, 5, etc.), then *select* is a contiguous block of 1's starting with the MSB.

Let us now look at how we generate *select* from α. Figure 9 shows how. Let us apply Fig. 9 to f(10), where Δ is 4, and hence $lg\Delta$ is 2. 10 in base 10 is 1010 in binary and $\alpha[lg\Delta - 1 : 0]$, lower $lg\Delta$ bits of α is binary $10 (= 2)$. We start with $\Delta 1'b1$, i.e., Δ 1's in Verilog notation. That is, binary 1111. Left-shift it by 2 positions to get $\Delta = 4$ bits, and we get 1100. Bitwise XOR that with $\alpha[lg\Delta]$ (bit 2 of α), which is 0, and we get 1100. That is indeed the *select* signal.

Let us do the same for f(13). 13 (α) in base 10 is 1101 in binary. Left-shift 1111 by binary 01 ($\alpha[lg\Delta - 1 : 0]$) positions, and we get 1110. Bitwise XOR that $\alpha[lg\Delta] = 1$, and all bits flip, resulting in 0001, which is the correct value.

DataSelection module shown in Fig. 10 is quite straight forward. It is practically a vector-multiplication unit or some sort of multiplexer as expressed in (8). It multiplies every *dval* (i.e., dLUT output) with the corresponding *select* bit, resulting in signal *dsel*. This can be done with an AND gate provided that each *select* bit duplicated d times (i.e., $dselect[i]$), which happens to be the bit width of *dval* as well as *dsel*.

$$dsel[i] = \begin{cases} dval[i], \text{ if } select[i] = 1 \\ \quad 0, \qquad otherwise \end{cases} \text{ where } i < \Delta = 2^{s-1} \tag{8}$$

After masking out some *dval*'s and obtaining *dsel*'s, we need to sum *dsel*'s and the corresponding entry from TIV_{new}. That is best done with a Column Compression Tree (CCT). In our work, the summation is done by SignedSummation shown in Fig. 11, which is based on the CCT generator proposed in [13] (called RoCoCo). RoCoCo handles only the summation of unsigned numbers, which is why the following conversion had to be done.

Consider the summation when *mval* is 16 bits, hence $\underline{a}$ and 15 x's in (9). The *dsel*'s are 4 bits each, and there are 4 dLUTs. That is why we have four 4-bit numbers in (9). These numbers are 2's complement, which is why their MSBs are negative. The underbar a, b, c, d, e show that these numbers are negative. In other words, a, b, c, d, e can be 0 or 1, and $\underline{a}$, $\underline{b}$, $\underline{c}$, $\underline{d}$, $\underline{e}$ are either $-0 = 0$ or -1. In summary, we need to perform the summation in (9), where

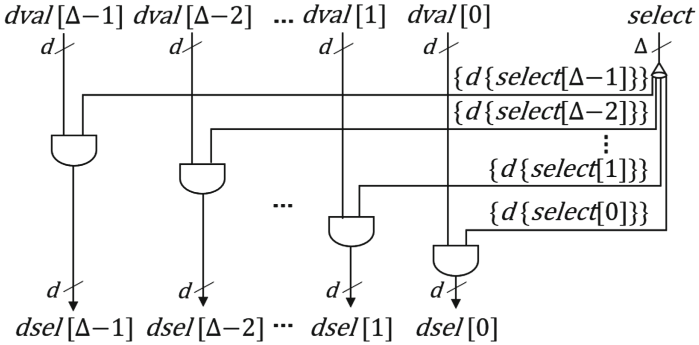


Fig. 10. DataSelection module.

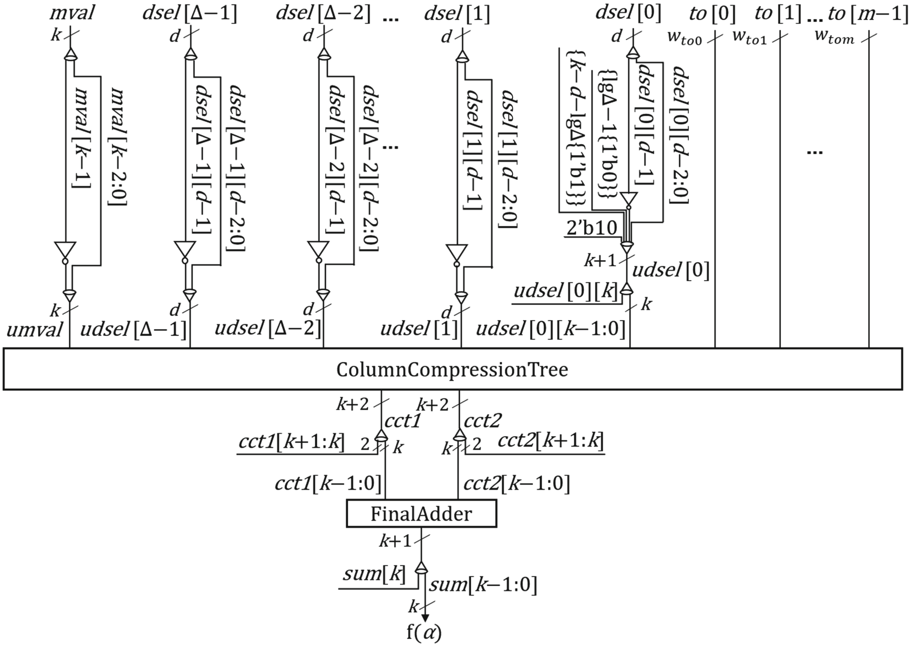


Fig. 11. SignedSummation module.

most bit positions have 0 and 1, while two bit positions have 0, 1, and −1. We can apply unsigned summation techniques only when −1 can be present in the highest bit position, that is, bit position 15 (the bit position $\underline{a}$). For this, we may sign-extend the 4-bit 2's complement numbers in (9) to 16 bits. Then, the result will also be 16-bit 2's complement, that is, a binary number with negative bit 15. We also have to make sure that the summation does not overflow. If it may, we need to sign-extend all numbers to 17 bits. However, in our case, the sum is also guaranteed to be 16-bit 2's complement.

$$\underline{\mathrm{a}}\mathrm{xxxxxxxxxxxxxxx} + \underline{\mathrm{b}}\mathrm{xxx} + \underline{\mathrm{c}}\mathrm{xxx} + \underline{\mathrm{d}}\mathrm{xxx} + \underline{\mathrm{e}}\mathrm{xxx} \tag{9}$$

$$\begin{aligned} &= \bar{\mathrm{a}}\mathrm{xxxxxxxxxxxxxxx} + \underline{1}000000000000000 + \bar{\mathrm{b}}\mathrm{xxx} \\ &\quad + \underline{1}000 + \bar{\mathrm{c}}\mathrm{xxx} + \underline{1}000 + \bar{\mathrm{d}}\mathrm{xxx} + \underline{1}000 + \bar{\mathrm{e}}\mathrm{xxx} + \underline{1}000 \end{aligned} \tag{10}$$

$$\begin{aligned} &= \bar{\mathrm{a}}\mathrm{xxxxxxxxxxxxxxx} + \bar{\mathrm{b}}\mathrm{xxx} + \bar{\mathrm{c}}\mathrm{xxx} + \bar{\mathrm{d}}\mathrm{xxx} + \bar{\mathrm{e}}\mathrm{xxx} \\ &\quad + \underline{1}0111111111100000 \end{aligned} \tag{11}$$

$$= \bar{\mathrm{a}}\mathrm{xxxxxxxxxxxxxxx} + \bar{\mathrm{b}}\mathrm{xxx} + \bar{\mathrm{c}}\mathrm{xxx} + \bar{\mathrm{d}}\mathrm{xxx} + \underline{1}011111111110\bar{\mathrm{e}}\mathrm{xxx} \tag{12}$$

$$= \bar{\mathrm{a}}\mathrm{xxxxxxxxxxxxxxx} + \bar{\mathrm{b}}\mathrm{xxx} + \bar{\mathrm{c}}\mathrm{xxx} + \bar{\mathrm{d}}\mathrm{xxx} + 011111111110\bar{\mathrm{e}}\mathrm{xxx} \tag{13}$$

Having said all that, a smarter (more area and speed efficient) approach, is to use the identity in Table 1, and hence, to replace the $\underline{\mathrm{a}}$ in (9) with $\bar{\mathrm{a}} + \underline{1}$, where $\bar{\mathrm{a}}$ is an unsigned 0 or 1 and is the NOT of a. Through this we get (10), which is a summation of variable unsigned numbers and some 2's complement constant numbers with still varying positions for the negative bits. However, these 2's complement constants can be summed offline to obtain a single 16-bit 2's complement number, hence (11). Then, we happen to have negative bits only in bit position 15. Equation (12) does a little optimization by adding the rightmost two numbers in (10) offline and hence combining them into a single number ($\underline{1}011111111110\bar{\mathrm{e}}\mathrm{xxx}$). Equation (13), on the other hand, eliminates bit position 16 of that combined number as the sum is guaranteed to be 16 bits (bit positions 0 through 15).

Table 1. Proof of $\underline{\mathrm{a}} = \bar{\mathrm{a}} + 1$.

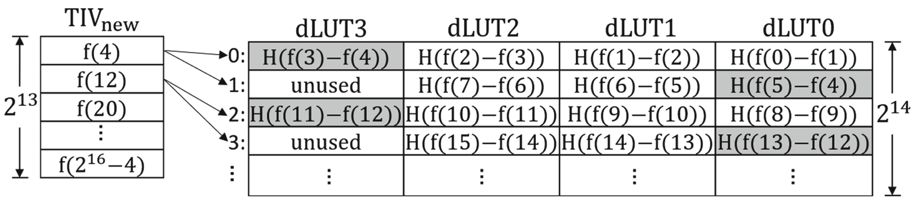
$\underline{\mathrm{a}}$	$\bar{\mathrm{a}} + \underline{1}$
$\underline{0}$	$\bar{0} + \underline{1} = 1 + (-1) = 0$
$\underline{1}$	$\bar{1} + \underline{1} = 0 + (-1) = -1$

The SignedSummation module in Fig. 8 is detailed in Fig. 11. It is composed of CCT and a Final Adder. The little cones in Fig. 11 expose bits of signals and recombine them. The bit-level manipulation before the CCT is a pictorial representation of equations (9) through (13). Note that the number of zeros to the left of $\bar{e}$ does not have to be one as in (13) in the general case. It is $lg\Delta - 1$ zeros (assuming Δ is a power of 2) as in Fig. 11 (look under $dsel[0]$). Also, note that the CCT in Fig. 11 not only sums TIV_{new} and dLUT outputs but also the outputs of TOs (shown in Fig. 2). There is no reason why we should do two separate summations.

3.2 FR-dLUT-VL Microarchitecture

FR-dLUT-VL has a similar microarchitecture to FR-dLUT. They both have Δ dLUTs and a TIV$_{new}$. The contents of TIV$_{new}$ are the same for both architectures. The difference between FR-dLUT-VL and FR-dLUT is the dLUT contents (compare Fig. 12 with Fig. 7). FR-dLUT-VL's dLUTs store the compressed versions of the differences stored in the FR-dLUT's dLUT. Due to the nature variable length coding, dLUTs of FR-dLUT-VL are wider than those of FR-dLUT. However, unused bits of those dLUTs' entries are don't care, and logic synthesis can optimize them out.

Figure 12 shows the entries of the FR-dLUT-VL with 4 dLUTs for a function with $w_i = 16$ and $w_o = 16$.



TIV$_{new}$
f(4)
f(12)
f(20)
⋮
f(2^{16}−4)

	dLUT3	dLUT2	dLUT1	dLUT0
0:	H(f(3)−f(4))	H(f(2)−f(3))	H(f(1)−f(2))	H(f(0)−f(1))
1:	unused	H(f(7)−f(6))	H(f(6)−f(5))	H(f(5)−f(4))
2:	H(f(11)−f(12))	H(f(10)−f(11))	H(f(9)−f(10))	H(f(8)−f(9))
3:	unused	H(f(15)−f(14))	H(f(14)−f(13))	H(f(13)−f(12))
⋮	⋮	⋮	⋮	⋮

Fig. 12. LUT contents of FR-dLUT-VL implementation for $\Delta = 4$.

It is obvious that one can apply any compression method on dLUT contents. In this work, Huffman coding [12] is chosen for this purpose. For each dLUT, the frequency of the values is calculated. According to the frequency, each value is assigned a Huffman code. Frequency for each value in a dLUT is calculated separately from the other dLUTs. Additionally, due to the encoded values stored in the dLUTs, after each value is read from its respective dLUT, it needs to go through a Decoder module before it can be used in SignedSummation module.

In FR-dLUT-VL microarchitecture, instead of encoding the whole value, a portion of the entry is taken and then the encoding method is applied. For example, if three MSBs are selected, each entry's three MSBs are encoded. To determine the separation point in a dLUT, starting from the two MSBs of an entry to all bits of entry, each possible combination is tested. For each combination, dLUT sizes and the decoder sizes are calculated. Calculation of dLUT sizes are done by adding encoded values' bit width and the bit width of the remaining bits that are not used in the encoding. Among these combinations, the one with the lower bit count is selected for implementation. Pseudocode of separation point selection is shown in Algorithm 1.

Algorithm 1. MSB calculation of FR-dLUT-VL design.

```
Input: $w_i$ : input bit width, $w_o$ : output bit width
$LUTsizeMin \leftarrow 2^{w_i} * w_o$
$MSBcountMin \leftarrow 0$
for $MSBcountMin \leftarrow 2$ to $w_o$ do
    compute encoded representation of every entry
    $LUTsize \leftarrow 0$
    $DecoderSize \leftarrow 0$
    for all unique encoded value do
        $LUTsize \leftarrow LUTsize +$ (encoded value bit length) $*$ (frequency of the value)
        $DecoderSize \leftarrow DecoderSize +$ (encoded value bit length)
    end for
    $LUTsize \leftarrow LUTsize + 2^{w_i} * (w_o - MSBcount) + DecoderSize$
    if $LUTsize < LUTsizeMin$ then
        $LUTsizeMin \leftarrow LUTsize$
        $MSBcountMin \leftarrow MSBcount$
    end if
end for
```

Encoded part of the accessed dLUT entry is sent to the Decoder module. After the encoded value is decoded, output bits and the non-encoded segment of the stored value in dLUT is concatenated and sent to the DataSelection module. Top module of FR-dLUT-VL method is shown in Fig. 13. As shown in Fig. 13, after the values are read from the dLUT (*dvals*), part of *dval* goes through the Decoder, then the decoded value merges with the rest of the *dval* and becomes the decoded difference value (*ddval*) which is the input for the DataSelection module.

3.3 SR-dLUT Microarchitecture

The main idea in SR-dLUT is to eliminate the TIV$_{new}$ in FR-dLUT and instead add dLUT entries on top of the previous function value. This can be done when not only the function f is smooth but also the input (x in f(x)) is smooth. Smoothness of x means the consecutive values of x are close to each other. This is, for example, the case when x comes from a sensor for example in a closed-loop control application, or when x is a timer tick or a smooth function of time. Elimination of TIV$_{new}$ makes SR-dLUT quite competitive in terms of area.

Figure 15 shows the top level of the proposed SR-dLUT microarchitecture, which consists of AddressGenerator (Fig. 16), dLUTs, DataSelection (Fig. 19), and SignedSummation (Fig. 20).

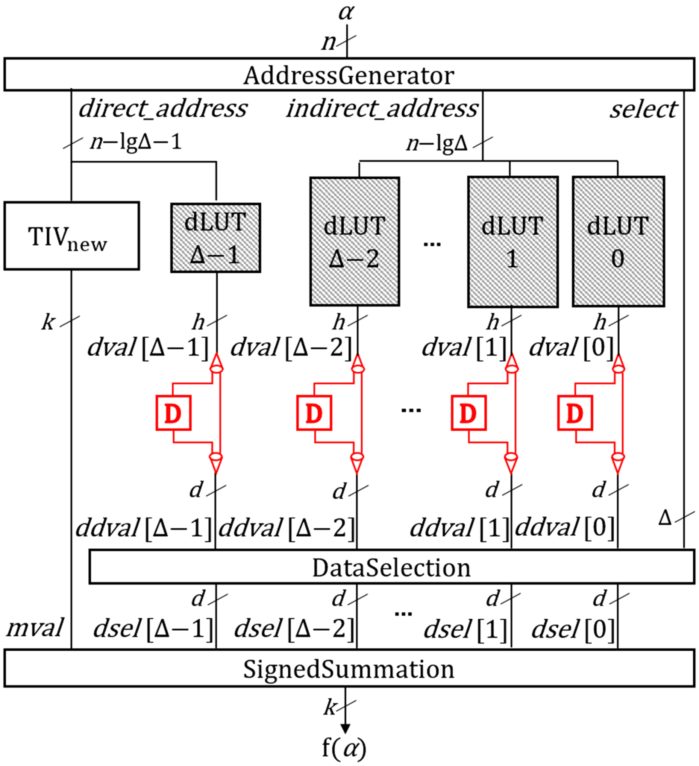


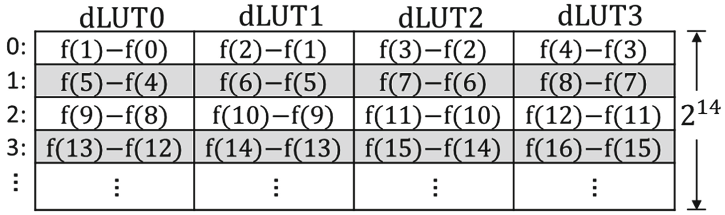
Fig. 13. Top module of FR-dLUT-VL microarchitecture.

Note that Δ of SR-dLUT is not the same as Δ of FR-dLUT. For FR-dLUT, Δ is a purely internal parameter stating the number of dLUTs, and hence, it has nothing to do with how it functions. For SR-dLUT, Δ is not only an internal parameter but also shows how smooth x is, that is the maximum amount of change between consecutive x values.

AddressGenerator module calculates the required addresses of dLUTs for input α. One register stores the previous input value. The previous input is subtracted from the current input to find *magnitude* and *diff_sign* that determines whether the shift direction may be forward or backward. The `thermo` vector is shifted in direction of *diff_sign* as the amount of the magnitude. (Fig. 18) shows the ThermoRegulator block that resolves the overflow issue due to the shift operation. The `thermo` vector is XOR'ed with the shifted `thermo` vector. Then, the output of the first XOR is shifted by 1 in a backward direction, and the resulting signal is named *tmp0*. The specific bit portion of *tmpO* is XOR'ed to determine *select* signals. Similarly, the specific bit portion of thermo is ORed to determine *dLut_Out_En* which is used in control mechanisms of both ThermoRegulator and DataSelection modules. (Fig. 17) shows the AddressSelection module that uses a conditional control mechanism by using *tmp_α* signals as input *tmpO* signals as select to determine addresses of dLUTs.

Note that SR-dLUT has access to much more than $\pm\Delta$ values, since each TIV value is coupled with multiple TO tables in MP. In Fig. 2, we can see that the same TIV value is used for $2^{(\Delta x_1 + \Delta x_2 + \Delta x_3)}$ in the case of $m = 3$. While we can access $2 * \Delta$ different entries in dLUTs, we can compute the function for $2 * \Delta * 2^{(\Delta x_1 + \Delta x_2 + \Delta x_3)}$ different x values. For the reasons explained above, $\Delta = 1$ is a meaningful case for SR-dLUT unlike FR-dLUT.

There are Δ parallel dLUTs that store difference values in SR-dLUT microachitecture. Figure 14 shows an example of SR-dLUT with 4 dLUTs, which implements a function f with $w_i = 16$. SR-dLUTs store subsequent differences values in-order and each dLUT stores same amount of data with a length of $2^{(n - lg\Delta)}$. Since the SR-dLUT method does not require a TIV$_{new}$ table, we can no longer have half entries for one of the dLUTs like in Fig. 7. In other words, there is no unused memory location in dLUTs.



	dLUT0	dLUT1	dLUT2	dLUT3
0:	f(1)−f(0)	f(2)−f(1)	f(3)−f(2)	f(4)−f(3)
1:	f(5)−f(4)	f(6)−f(5)	f(7)−f(6)	f(8)−f(7)
2:	f(9)−f(8)	f(10)−f(9)	f(11)−f(10)	f(12)−f(11)
3:	f(13)−f(12)	f(14)−f(13)	f(15)−f(14)	f(16)−f(15)
⋮	⋮	⋮	⋮	⋮

Fig. 14. LUT contents of SR-dLUT implementation for $w_i = 16$ and $\Delta = 4$.

DataSelection module of SR-dLUT is similar to DataSelection module of FR-dLUT. The difference from FR-dLUT is that there is an extra control mechanism that determines which signal is multiplied by the select signal. The multiplication is implemented using an AND gate. The results of multiplication are *dsel* signals that are connected to the SignedSummation module.

SignedSummation module of SR-dLUT is different from the FR-dLUT's SignedSummation. Since it needs to hold the previous value of the TIV output in a register, and it could not use a single CCT to add dLUT outputs and the TOs output from MP method. Therefore, there is an additional CCT. The first CCT uses the current values of *dsel* and the previous outputs of the CCT. It outputs two values, which are the values generated just before going through the second CCT. The outputs of the first CCT are saved in registers for the next iteration, at the same time used in the second CCT as inputs together with the TOs outputs. As a conclusion, a final adder sums outputs of the second CCT.

One of the differences between SR-dLUT and the earlier version of this work [10] is the form of thermo vector. While the size of thermo vector is 3Δ-bit SR-dLUT, the size of the thermo vector is Δ-bit in [10]. The advantage of using this new form of thermo vector is to find the dLUT addresses easier. In other words, the AdressSelection module in our proposed SR-dLUT is simpler than in [10]. Additionally, we used CCTs for the summation of *dsels* in our proposed SR-dLUT. On the other hand, conventional addition units are used in [10].

Our proposed summation module has an advantage in terms of area when Δ increases because the addition unit in [10] is proportional to Δ.

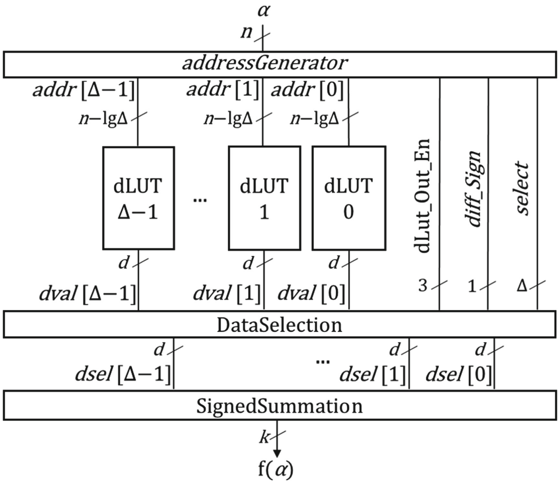


Fig. 15. Top module of SR-dLUT microarchitecture.

3.4 SR-dLUT-VL Microarchitecture

Difference between SR-dLUT-VL and SR-dLUT is similar to the difference between FR-dLUT and FR-dLUT-VL, where values inside dLUTs are partially compressed according the algorithm shown in Algorithm 1. As in FR-dLUT-VL, partial output of dLUTs' (*dval*) in SR-dLUT-VL first go through a decoder, then output of the decoder is concatenated with the remaining part of the *dval*. The concatenated value represents the actual *dval* from the SR-dLUT microarchitecture, which can now be used in the SignedSummation module.

4 Results

In this section, we compare our proposed microarchitectures with the state-of-the-art (T-TIV methods) as well as ConvLUT (i.e., RegularTIV). For each of the 8 TIV construction methods listed above, we wrote a code generator in Perl that generates Verilog RTL for a complete MP design. The code generators take in w_i, w_o, and m (number of MP's TOs) as well as parameters related to the specific TIV construction method such as Δ for our methods.

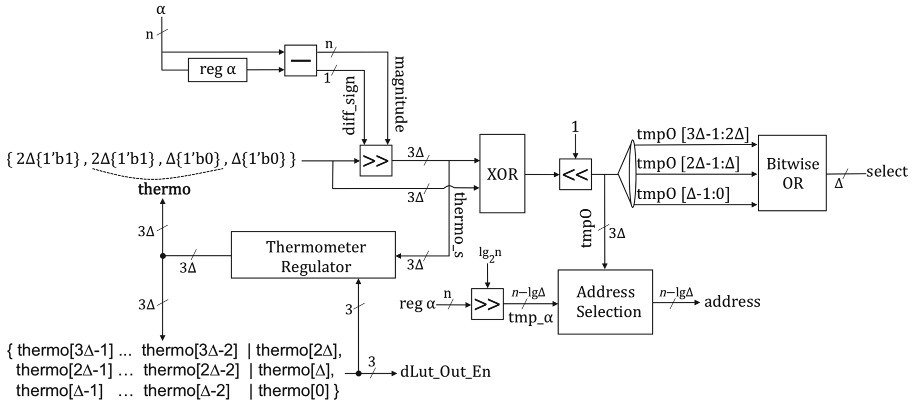


Fig. 16. SR-dLUT AddressGenerator module.

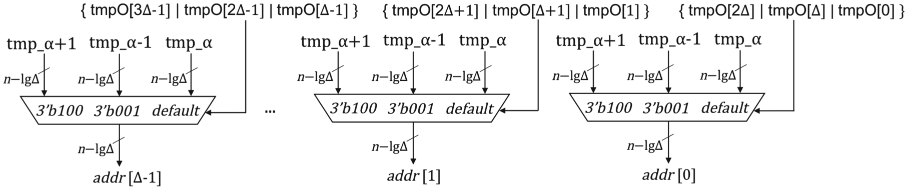


Fig. 17. SR-dLUT AddressSelection module.

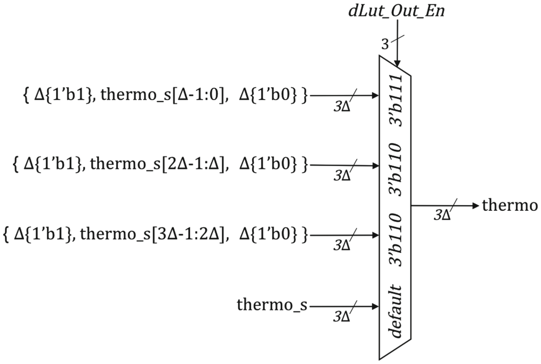

dLut_Out_En
3
{ Δ{1'b1}, thermo_s[Δ-1:0], Δ{1'b0} }
3Δ
3'b111
{ Δ{1'b1}, thermo_s[2Δ-1:Δ], Δ{1'b0} }
3Δ
3'b110
{ Δ{1'b1}, thermo_s[3Δ-1:2Δ], Δ{1'b0} }
3Δ
3'b110
thermo_s
3Δ
default
3Δ
thermo

Fig. 18. SR-dLUT ThermometerRegulator module.

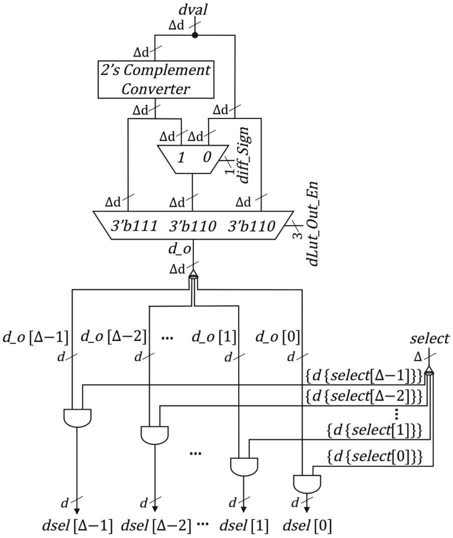


Fig. 19. SR-dLUT DataSelection module.

Generation of the original TIV (RegularTIV) and TOs, i.e., the MP method [6], is done using a tool [14], which is also part of FloPoCo [15]. This tool (written in Java) generates RTL code in VHDL targeting various mathematical functions and parameters W_i, W_o, and m. Our Perl script runs the Java MP generator with W_i, W_o, and m, where W_o shows the precision of the output rather than the bit width of any of the embedded tables. MP generator decides what sizes RegularTIV and TOs have to be for the given set of w_i, w_o, and m. This information is embedded into Table 2 of [6] which, for example, shows that $w_i = 10$ and $w_o = 17$ for the RegularTIV part, when the function is sine and $w_i = 16$, $w_o = 16$, and $m = 1$. Our Perl takes in w_i, w_o, and m, runs the Java MP generator with the input parameters. Then, it parses the VHDL produced by the MP generator, extracts w_i and w_o, and runs the TIV generator. Then, our script produces Verilog, in which we have an MP design equivalent to the input VHDL but the RegularTIV is replaced with our TIV of choice (one of FR-dLUT, FR-dLUT-VL, SR-dLUT, SR-dLUT-VL, 2T-TIV, 2T-TIV-IMP, 3T-TIV). Our script automates verification as well as design. We exhaustively test all function values by comparing the output of the VHDL and Verilog designs.

We generated the above listed 8 designs for $\sin(x)$ ($x = [0, \pi/2[$) and 2^x ($x = [0, 1[$) functions with $w_i = 16$, $w_o = 16$ and $w_i = 24$, $w_o = 24$, both with various m for each resolution. We then synthesized them on to a Xilinx Artix-7 FPGA (more specifically XC7A100T-3CSG324). FR-dLUT and FR-dLUT-VL were synthesized for four Δ values (2, 4, 8, and 16) where SR-dLUT and SR-dLUT-VL synthesized for an additional Δ value, where $\Delta = 1$. For 24-bit resolution, m values of 1, 2, 3, and 4 were implemented. For 16-bit resolution, m values (number of TOs) of 1, 2, and 3 were implemented, since the Java tool at

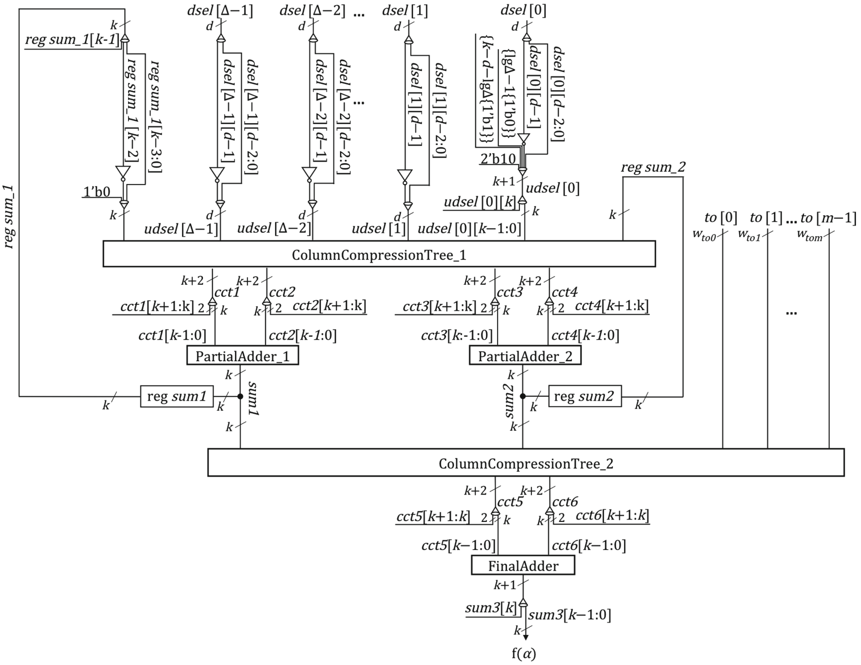


Fig. 20. SR-dLUT SignedSummation module.

[14] gave an error and did not produce any VHDL. Note that all LUTs (RegularTIV, dLUTs, TIV$_{new}$, TIV$_{diff}$, TIV$_{diff1}$, and TIV$_{diff2}$) are logic-synthesized instead of instantiating memory blocks. This yields designs with smaller area for all methods including the original TIVs. The synthesis option of Area with High effort is selected for better area minimization in Xilinx ISE.

Tables 2, 3, and 4 show the Area, Time, and Area-Time Product (ATP) results we obtained for sine and exponent function (2^x) for 16-bit resolutions, respectively. Similarly, Tables 5, 6, and 7 show the Area, Time, and ATP results we obtained for 24-bit versions of sine and exponent function. Area results are in terms of FPGA LUTs. Time (also called timing) is the latency (i.e., critical path) of the circuit measured in terms of nanoseconds (ns). ATP simply shows the product of Area (#LUTs) and Time (ns) columns divided by 10^3. ATP of a design shows the tradeoff between area and timing, also ATP is usually correlated with power consumption. The best Area, Time, and ATP results are shaded for each m value in the tables.

In Table 2 through 7, all designs (60 for 16-bit and 80 for 24-bit) are equivalent at MP level. That is, they all produce the same function with the same resolution. However, the TIV microarchitectures are equivalent within the

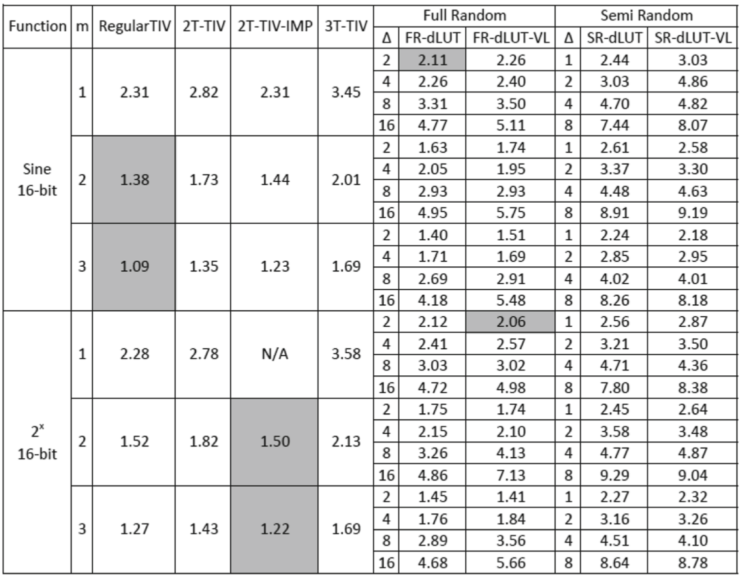
Table 2. Area results for 16-bit resolution

Function	m	RegularTIV	2T-TIV	2T-TIV-IMP	3T-TIV	Full Random			Semi Random		
						Δ	FR-dLUT	FR-dLUT-VL	Δ	SR-dLUT	SR-dLUT-VL
Sine 16-bit	1	346	343	346	357	2	249	277	1	245	268
						4	249	271	2	285	306
						8	313	349	4	368	380
						16	435	454	8	521	547
	2	188	192	187	212	2	208	215	1	253	257
						4	245	245	2	287	293
						8	320	320	4	362	364
						16	451	448	8	606	610
	3	147	157	159	185	2	181	187	1	205	210
						4	220	222	2	242	244
						8	277	277	4	317	325
						16	402	406	8	550	555
2^x 16-bit	1	341	334	N/A	365	2	255	262	1	251	258
						4	265	283	2	293	306
						8	323	320	4	375	382
						16	438	438	8	521	551
	2	196	199	196	230	2	215	214	1	259	262
						4	247	242	2	290	295
						8	321	320	4	368	372
						16	443	442	8	591	588
	3	156	165	156	200	2	191	188	1	217	221
						4	232	231	2	252	257
						8	292	288	4	331	330
						16	420	416	8	553	552

Table 3. Timing results for 16-bit resolution

Function	m	RegularTIV	2T-TIV	2T-TIV-IMP	3T-TIV	Full Random			Semi Random		
						Δ	FR-dLUT	FR-dLUT-VL	Δ	SR-dLUT	SR-dLUT-VL
Sine 16-bit	1	6.69	8.22	6.67	9.66	2	8.46	8.16	1	9.94	11.30
						4	9.09	8.85	2	10.62	15.90
						8	10.58	10.02	4	12.77	12.69
						16	10.97	11.25	8	14.28	14.75
	2	7.34	9.02	7.68	9.47	2	7.82	8.09	1	10.32	10.04
						4	8.35	7.94	2	11.75	11.27
						8	9.16	9.16	4	12.38	12.71
						16	10.98	12.83	8	14.70	15.07
	3	7.39	8.62	7.73	9.15	2	7.73	8.09	1	10.95	10.40
						4	7.77	7.60	2	11.76	12.07
						8	9.70	10.50	4	12.69	12.35
						16	10.40	13.49	8	15.02	14.73
2^x 16-bit	1	6.69	8.31	N/A	9.81	2	8.32	7.88	1	10.19	11.11
						4	9.08	9.08	2	10.94	11.42
						8	9.38	9.45	4	12.57	11.42
						16	10.78	11.38	8	14.97	15.20
	2	7.77	9.14	7.65	9.27	2	8.14	8.14	1	9.45	10.07
						4	8.70	8.66	2	12.34	11.81
						8	10.16	12.92	4	12.96	13.08
						16	10.97	16.13	8	15.72	15.38
	3	8.16	8.69	7.81	8.44	2	7.60	7.50	1	10.45	10.50
						4	7.57	7.95	2	12.55	12.70
						8	9.90	12.36	4	13.61	12.43
						16	11.14	13.61	8	15.62	15.90

Table 4. ATP results for 16-bit resolution



Function	m	RegularTIV	2T-TIV	2T-TIV-IMP	3T-TIV	Full Random			Semi Random		
						Δ	FR-dLUT	FR-dLUT-VL	Δ	SR-dLUT	SR-dLUT-VL
Sine 16-bit	1	2.31	2.82	2.31	3.45	2	2.11	2.26	1	2.44	3.03
						4	2.26	2.40	2	3.03	4.86
						8	3.31	3.50	4	4.70	4.82
						16	4.77	5.11	8	7.44	8.07
	2	1.38	1.73	1.44	2.01	2	1.63	1.74	1	2.61	2.58
						4	2.05	1.95	2	3.37	3.30
						8	2.93	2.93	4	4.48	4.63
						16	4.95	5.75	8	8.91	9.19
	3	1.09	1.35	1.23	1.69	2	1.40	1.51	1	2.24	2.18
						4	1.71	1.69	2	2.85	2.95
						8	2.69	2.91	4	4.02	4.01
						16	4.18	5.48	8	8.26	8.18
2^x 16-bit	1	2.28	2.78	N/A	3.58	2	2.12	2.06	1	2.56	2.87
						4	2.41	2.57	2	3.21	3.50
						8	3.03	3.02	4	4.71	4.36
						16	4.72	4.98	8	7.80	8.38
	2	1.52	1.82	1.50	2.13	2	1.75	1.74	1	2.45	2.64
						4	2.15	2.10	2	3.58	3.48
						8	3.26	4.13	4	4.77	4.87
						16	4.86	7.13	8	9.29	9.04
	3	1.27	1.43	1.22	1.69	2	1.45	1.41	1	2.27	2.32
						4	1.76	1.84	2	3.16	3.26
						8	2.89	3.56	4	4.51	4.10
						16	4.68	5.66	8	8.64	8.78

same m. Therefore, it is best to compare the 8 microarchitectures separately for every m. However, out of curiosity we compare all solutions for each of the 4 cases (16-bit sine, 16-bit 2^x, 24-bit sine, 24-bit 2^x).

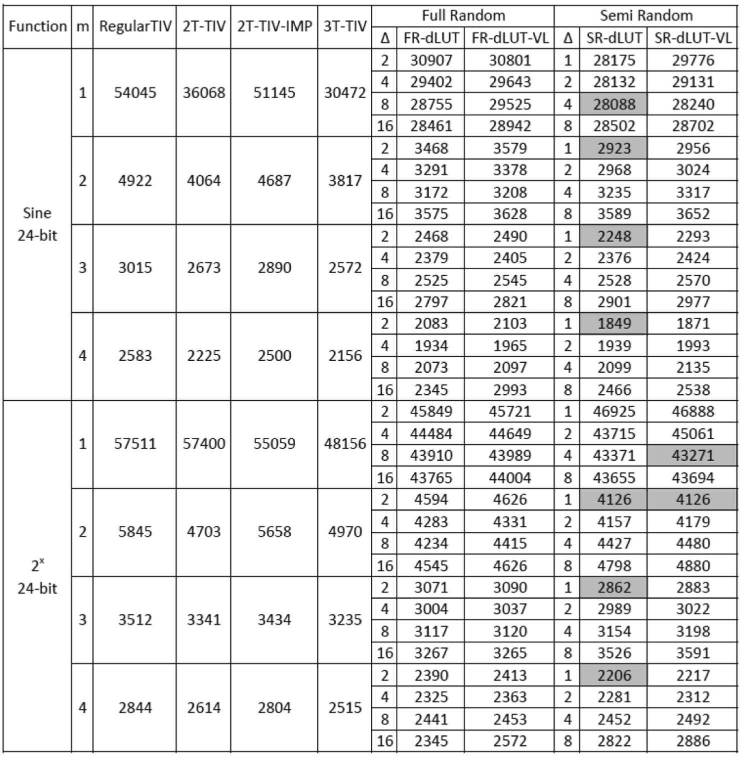
Tables 2, 3, and 4 implement 16-bit sine and 2^x 60 different ways each, where 57 comes from (RegularTIV + 2T-TIV + 2T-TIV-IMP + 3T-TIV + 4×FR-dLUT + 4×FR-dLUT-VL + 4×SR-dLUT + 4×SR-dLUT-VL) × (3 m values). In the above tables, one result is missing for 2T-TIV-IMP because TIV$_{new}$ and TIV$_{diff}$ in the corresponding 2T-TIV have equal bit width, hence no optimization is possible.

Tables 5, 6, and 7 implement 24-bit sine and 2^x for an additional 20 different ways due to the $m = 4$ case. The Time and ATP of RegularTIV for $m = 1$ are missing for both functions and for 2T-TIV-IMP for 2^x function in Table 6 and 7 because the synthesis tool was not able to complete routing. The tool reported the Area but did not report the timing (Time).

Looking at Tables 2, 3, and 4, we can see that, except for 5 cases (2 in Area, 1 in Timing, and 2 in ATP), the proposed method is surpassed by the RegularTIV and the T-TIV variants for 16-bit implementations. Though in a few cases, our proposed methods offer up to 29%, 4%, 9% improvement in area, timing, ATP, respectively over the state-of-the-art and RegularTIV. The area improvements come from SR-dLUT. Time and ATP improvements come from FR-dLUT and FR-dLUT-VL.

Looking at Tables 5, 6, and 7 show the results for 24-bit implementations of sine and 2^x. In area, SR-dLUT (and a few times SR-dLUT-VL) gives the best results. In timing, FR-dLUT gives the best result in 3 out of the 8 cases

Table 5. Area results for 24-bit resolution



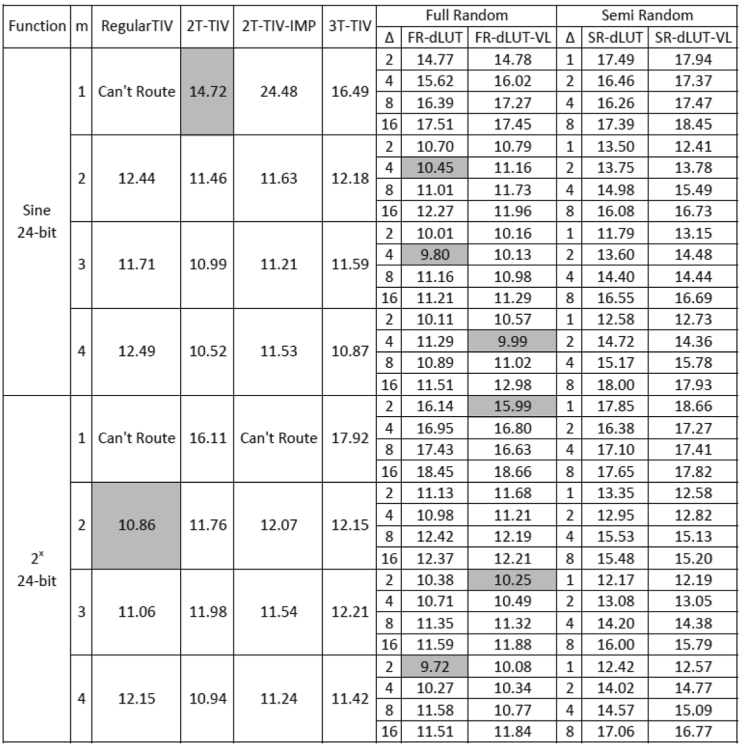
Function	m	RegularTIV	2T-TIV	2T-TIV-IMP	3T-TIV	Full Random			Semi Random		
						Δ	FR-dLUT	FR-dLUT-VL	Δ	SR-dLUT	SR-dLUT-VL
Sine 24-bit	1	54045	36068	51145	30472	2	30907	30801	1	28175	29776
						4	29402	29643	2	28132	29131
						8	28755	29525	4	28088	28240
						16	28461	28942	8	28502	28702
	2	4922	4064	4687	3817	2	3468	3579	1	2923	2956
						4	3291	3378	2	2968	3024
						8	3172	3208	4	3235	3317
						16	3575	3628	8	3589	3652
	3	3015	2673	2890	2572	2	2468	2490	1	2248	2293
						4	2379	2405	2	2376	2424
						8	2525	2545	4	2528	2570
						16	2797	2821	8	2901	2977
	4	2583	2225	2500	2156	2	2083	2103	1	1849	1871
						4	1934	1965	2	1939	1993
						8	2073	2097	4	2099	2135
						16	2345	2993	8	2466	2538
2^x 24-bit	1	57511	57400	55059	48156	2	45849	45721	1	46925	46888
						4	44484	44649	2	43715	45061
						8	43910	43989	4	43371	43271
						16	43765	44004	8	43655	43694
	2	5845	4703	5658	4970	2	4594	4626	1	4126	4126
						4	4283	4331	2	4157	4179
						8	4234	4415	4	4427	4480
						16	4545	4626	8	4798	4880
	3	3512	3341	3434	3235	2	3071	3090	1	2862	2883
						4	3004	3037	2	2989	3022
						8	3117	3120	4	3154	3198
						16	3267	3265	8	3526	3591
	4	2844	2614	2804	2515	2	2390	2413	1	2206	2217
						4	2325	2363	2	2281	2312
						8	2441	2453	4	2452	2492
						16	2345	2572	8	2822	2886

(2 different designs and 4 different ms for each design). FR-dLUT-VL, on the other hand, is the best in 3 cases. 2T-TIV and RegularTIV each are the best in 1 case. In ATP, FR-dLUT, FR-dLUT-VL, and SR-dLUT are best in 4, 3, and 1 case, respectively. Our proposed methods offer up to 23%, 11%, 26% improvement in area, timing, ATP, respectively over the state-of-the-art and RegularTIV.

In summary, proposed methods offer, in greater bit widths, significant improvement in all of Area, Time, and ATP even beyond the state-of-the-art T-TIV methods.

The bar graphs (Fig. 21) summarize the results by comparing RegularTIV, the best of T-TIV methods, the best of FR-dLUT methods, and the best of SR-dLUT methods. RegularTIV and T-TIV methods combined are, on the average, superior to proposed FR-dLUT and SR-dLUT methods in 16-bit results.

Table 6. Timing results for 24-bit resolution

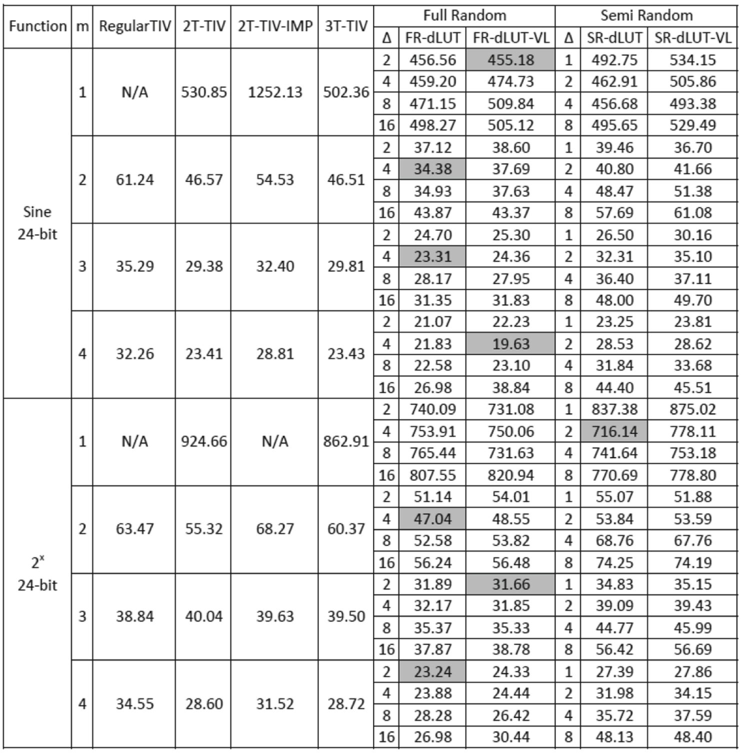


Function	m	RegularTIV	2T-TIV	2T-TIV-IMP	3T-TIV	Full Random			Semi Random		
						Δ	FR-dLUT	FR-dLUT-VL	Δ	SR-dLUT	SR-dLUT-VL
Sine 24-bit	1	Can't Route	14.72	24.48	16.49	2	14.77	14.78	1	17.49	17.94
						4	15.62	16.02	2	16.46	17.37
						8	16.39	17.27	4	16.26	17.47
						16	17.51	17.45	8	17.39	18.45
	2	12.44	11.46	11.63	12.18	2	10.70	10.79	1	13.50	12.41
						4	10.45	11.16	2	13.75	13.78
						8	11.01	11.73	4	14.98	15.49
						16	12.27	11.96	8	16.08	16.73
	3	11.71	10.99	11.21	11.59	2	10.01	10.16	1	11.79	13.15
						4	9.80	10.13	2	13.60	14.48
						8	11.16	10.98	4	14.40	14.44
						16	11.21	11.29	8	16.55	16.69
	4	12.49	10.52	11.53	10.87	2	10.11	10.57	1	12.58	12.73
						4	11.29	9.99	2	14.72	14.36
						8	10.89	11.02	4	15.17	15.78
						16	11.51	12.98	8	18.00	17.93
2^x 24-bit	1	Can't Route	16.11	Can't Route	17.92	2	16.14	15.99	1	17.85	18.66
						4	16.95	16.80	2	16.38	17.27
						8	17.43	16.63	4	17.10	17.41
						16	18.45	18.66	8	17.65	17.82
	2	10.86	11.76	12.07	12.15	2	11.13	11.68	1	13.35	12.58
						4	10.98	11.21	2	12.95	12.82
						8	12.42	12.19	4	15.53	15.13
						16	12.37	12.21	8	15.48	15.20
	3	11.06	11.98	11.54	12.21	2	10.38	10.25	1	12.17	12.19
						4	10.71	10.49	2	13.08	13.05
						8	11.35	11.32	4	14.20	14.38
						16	11.59	11.88	8	16.00	15.79
	4	12.15	10.94	11.24	11.42	2	9.72	10.08	1	12.42	12.57
						4	10.27	10.34	2	14.02	14.77
						8	11.58	10.77	4	14.57	15.09
						16	11.51	11.84	8	17.06	16.77

However, in 24-bit results either FR-dLUT or SR-dLUT method ore their variants (-VL) are superior. Only with one m value, we lose to our competition for each function (sine and 2^x), and that is in time metric. In each of the two 24-bit functions, there are 4 cases (i.e., 4 different m's), and 3 metrics (Time, Area, ATP). Therefore, we are speaking of 12 ways to compare the TIV microarchitectures. Ours are better in 11 out of 12.

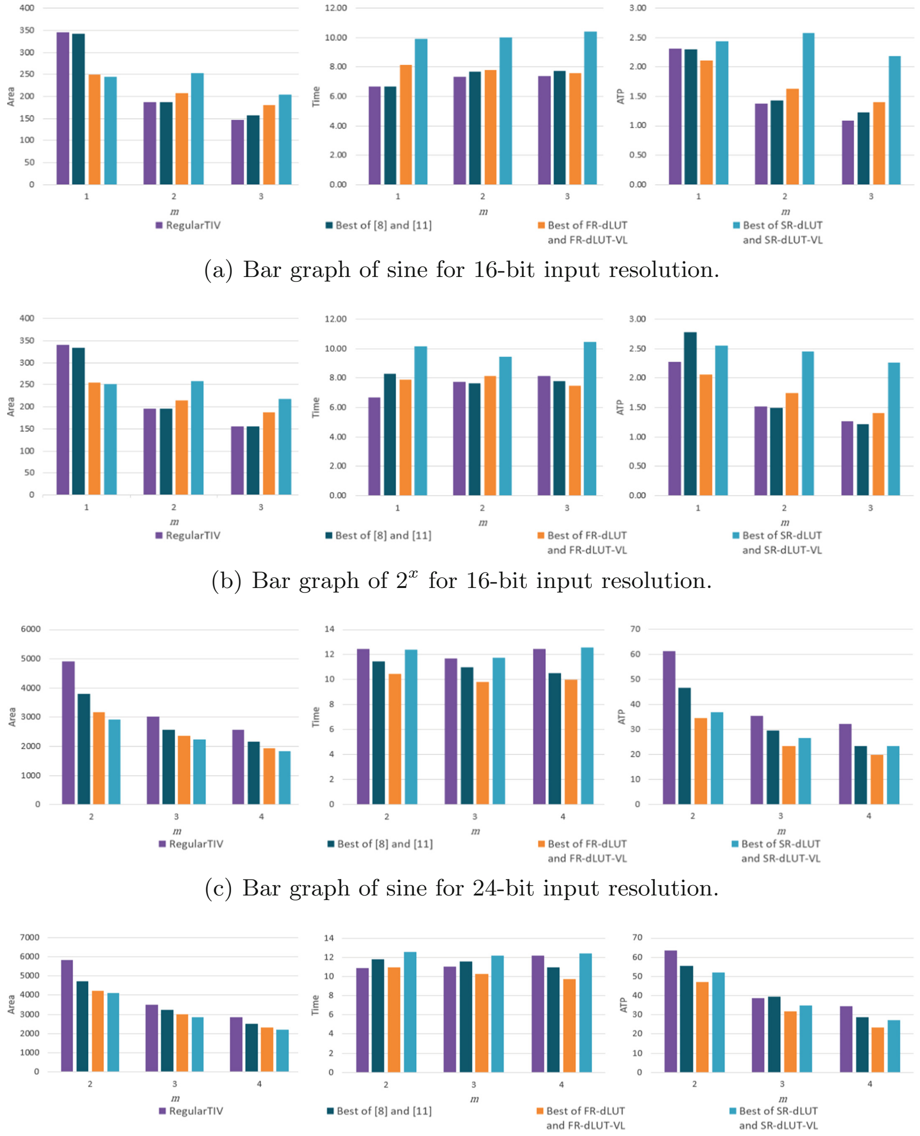
Figure 21(a) and (b) shows the 16-bit results. For 16 bits, RegularTIV has the best outcome in 8 out of 18 cases, 2T-TIV-IMP is the best in 7 cases, and our proposed metods are the best in 5 cases. Te total is 20, not 18, because there are 2 cases with a tie.

Table 7. ATP results for 24-bit resolution



Function	m	RegularTIV	2T-TIV	2T-TIV-IMP	3T-TIV	Full Random			Semi Random		
						Δ	FR-dLUT	FR-dLUT-VL	Δ	SR-dLUT	SR-dLUT-VL
Sine 24-bit	1	N/A	530.85	1252.13	502.36	2	456.56	455.18	1	492.75	534.15
						4	459.20	474.73	2	462.91	505.86
						8	471.15	509.84	4	456.68	493.38
						16	498.27	505.12	8	495.65	529.49
	2	61.24	46.57	54.53	46.51	2	37.12	38.60	1	39.46	36.70
						4	34.38	37.69	2	40.80	41.66
						8	34.93	37.63	4	48.47	51.38
						16	43.87	43.37	8	57.69	61.08
	3	35.29	29.38	32.40	29.81	2	24.70	25.30	1	26.50	30.16
						4	23.31	24.36	2	32.31	35.10
						8	28.17	27.95	4	36.40	37.11
						16	31.35	31.83	8	48.00	49.70
	4	32.26	23.41	28.81	23.43	2	21.07	22.23	1	23.25	23.81
						4	21.83	19.63	2	28.53	28.62
						8	22.58	23.10	4	31.84	33.68
						16	26.98	38.84	8	44.40	45.51
2^x 24-bit	1	N/A	924.66	N/A	862.91	2	740.09	731.08	1	837.38	875.02
						4	753.91	750.06	2	716.14	778.11
						8	765.44	731.63	4	741.64	753.18
						16	807.55	820.94	8	770.69	778.80
	2	63.47	55.32	68.27	60.37	2	51.14	54.01	1	55.07	51.88
						4	47.04	48.55	2	53.84	53.59
						8	52.58	53.82	4	68.76	67.76
						16	56.24	56.48	8	74.25	74.19
	3	38.84	40.04	39.63	39.50	2	31.89	31.66	1	34.83	35.15
						4	32.17	31.85	2	39.09	39.43
						8	35.37	35.33	4	44.77	45.99
						16	37.87	38.78	8	56.42	56.69
	4	34.55	28.60	31.52	28.72	2	23.24	24.33	1	27.39	27.86
						4	23.88	24.44	2	31.98	34.15
						8	28.28	26.42	4	35.72	37.59
						16	26.98	30.44	8	48.13	48.40

In Fig. 21(c) and (d), due to the routing errors we removed the results of $m = 1$ from the bar graphs. From the remaining m values (2, 3, and 4) we can see that proposed method is the best in terms of area, time, and ATP except for one case where $m = 2$, in which RegularTIV shows better timing results. When comparing FR-dLUT and SR-dLUT, we can see that SR-dLUT has the best performance in terms of area, and FR-dLUT has the best timing. In the current implementation of SR-dLUT, FR-dLUT shows much better timing performance but only falls behind a little in terms of area, that is why for almost all cases, FR-dLUT outperforms SR-dLUT in terms of ATP.



(a) Bar graph of sine for 16-bit input resolution.

(b) Bar graph of 2^x for 16-bit input resolution.

(c) Bar graph of sine for 24-bit input resolution.

(d) Bar graph of 2^x for 24-bit input resolution.

Fig. 21. Bar graphs of Area, Time, and ATP for sine and 2^x.

5 Conclusion

In this chapter, we have presented lossless LUT compression methods, called FR-dLUT and SR-dLUT as well as their variants FR-dLUT-VL and SR-dLUT-VL, which can be used to replace TIVs of Multi-Partite (MP) function evaluation method among other applications. It is possible to use our methods and the previous state-of-the-art within the context of any LUT-based method for evaluation of smooth functions. Although it is possible to implement these techniques in software, we implemented our proposed methods in hardware (i.e., low-level Verilog RTL) and benchmarked at 16-bit and 24-bit resolutions within MP implementations of sine and exponential.

It was observed that our four methods yield significant and consistent improvements at high resolution (i.e., 24 bits) over the previous state-of-the-art, which we also implemented through design generators for fair comparison. We synthesized the generated designs on FPGA and found that our methods result in up to 29% improvement in area, 11% improvement in latency, and 26% improvement in ATP over the previous state-of-the-art.

This chapter introduces SR-dLUT-VL for the first time. SR-dLUT is, on the other hand, is based on [10]. Yet, it is optimized and plugged into MP here, besides getting compared with the other methods in many cases. Moreover, FR-dLUT and FR-dLUT-VL are presented in greater detail compared to [16].

In future work, SR-dLUT can be modified to utilize pre-fetching. That may make SR-dLUT better than all other techniques in latency at the expense of some additional area. Since SR-dLUT is already efficient in area, it can afford some additional area. However, pre-fetching can also allow memory packing, which can lower area at the expense of some additional latency.

References

1. Lih, L.J.Y., Jong, C.C.: A memory-efficient tables-and-additions method for accurate computation of elementary functions. IEEE Trans. Comput. **62**, 858–872 (2013)
2. Hassler, H., Takagi, N.: Function evaluation by table look-up and addition. In: Proceedings of Symposium on Computer Arithmetic (ARITH), pp. 10–16 (1995)
3. Sarma, D.D., Matula, D.W.: Faithful bipartite ROM reciprocal tables. In: Proceedings of Symposium on Computer Arithmetic (ARITH), pp. 17–28 (1995)
4. Schulte, M.J., Stine, J.E.: Approximating elementary functions with symmetric bipartite tables. IEEE Trans. Comput. **48**, 842–847 (1999)
5. Stine, J.E., Schulte, M.J.: The symmetric table addition method for accurate function approximation. J. VLSI Sig. Proc. **21**, 167–177 (1999)
6. Dinechin, F.D., Tisserand, A.: Multipartite table methods. IEEE Trans. Comput. **54**, 319–330 (2005)
7. Detrey, J., Dinechin, F.D.: Multipartite tables in JBits for the evaluation of functions on FPGA. Diss. INRIA (2001)
8. Hsiao, S.F., Wen, C.S., Chen, Y.H., Huang, K.C.: Hierarchical multipartite function evaluation. IEEE Trans. Comput. **66**, 89–99 (2017)
9. Muller, J.-M.: A few results on table-based methods. Reliable Comput. **5**, 279–288 (1999)
10. Unlu, H., Ozkan, M.A., Ugurdag, H.F., Adali, E.: Area-efficient look-up tables for semi-randomly accessible functions. In: Proceedings of WSEAS Recent Advances in Electrical Engineering, pp. 171–174 (2014)

11. Hsiao, S.F., Wu, P.H., Wen, C.S., Meher, P.K.: Table size reduction methods for faithfully rounded lookup-table-based multiplierless function evaluation. IEEE Trans. Circ. Syst. II Express Briefs **62**, 466–470 (2015)
12. Huffman, D.A.: A method for the construction of minimum-redundancy codes. Proc. IRE **40**, 1098–1101 (1952)
13. Ugurdag, H.F., Keskin, O., Tunc, C., Temizkan, F., Fici, G., Dedeoglu, S.: RoCoCo: row and column compression for high-performance multiplication on FPGAs. In: Proceedings of East-West Design and Test Symposium (EWDTS), pp. 98–101 (2011)
14. Dinechin, F.D.: The multipartite method for function evaluation. http://www.ens-lyon.fr/LIP/Arenaire/Ware/Multipartite/. Accessed 8 Mar 2020
15. Dinechin, F.D., Pasca, B.: Designing custom arithmetic data paths with FloPoCo. IEEE Des. Test Comput. **28**, 18–27 (2011)
16. Gener, Y.S., Gören, S., Ugurdag, H.F.: Lossless look-up table compression for hardware implementation of transcendental functions. In: Proceedings of IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), pp. 52–57 (2019)