FULLY RANDOM ACCESS DIFFERENTIAL LOOKUP TABLES

by
Yılmaz Serhan Gener

Submitted to Graduate School of Natural and Applied Sciences
in Partial Fulfillment of the Requirements
for the Degree of Master of Science in
Computer Engineering

Yeditepe University
2017

FULLY RANDOM ACCESS DIFFERENTIAL LOOKUP TABLES

APPROVED BY:

Prof. Dr. Sezer Gören Uğurdağ
(Thesis Supervisor) .....

Assoc. Prof. Dr. Gürhan Küçük .....

Assist. Prof. Dr. Tankut Barış Aktemur .....

DATE OF APPROVAL: ..../..../2017

ACKNOWLEDGEMENTS

It is with immense gratitude that I acknowledge the support and help of my professors, Prof. Dr. Sezer Gören Uğurdağ and Assoc. Prof. Dr. Hasan Fatih Uğurdağ. Pursuing my thesis under their supervision has been an experience which broadens the mind and presents an unlimited source of learning.

Finally, I would like to thank my family for their endless love and support, which makes everything more beautiful.

ABSTRACT

FULLY RANDOM ACCESS DIFFERENTIAL LOOKUP TABLES

Lookup Tables (LUTs) are often used to implement complex functions in hardware and software design to achieve low latency in computation of complex functions compared to algebraic implementations. However, the area of a LUT grows exponentially with the bitwidth of the input. This thesis presents a novel area-efficient and parameterized logic microarchitecture that behaves identical to a Conventional LUT (ConvLUT) implementing a continuous function. Six different architectures are implemented, all architectures keep a down-sampled version of the original LUT. Skipped LUT entries are replaced with one of the following; differential LUT entries, encoded differential LUT entries, or a method we called zone folding. These three architectures are also implemented by storing differences of differential LUT entries. By employing some combinational logic circuitry, all architectures can mimic a ConvLUT with a slight compromise in latency. The proposed architectures are fully random access, and are named as “Fully Random Access Differential LUT” (FR-dLUT). Later, multipartite tables method is combined and improved with the proposed architectures. In order to evaluate area and performance of FR-dLUT, all its variants for sine and 2^x functions are coded in Verilog, verified, synthesized, and implemented on FPGA. Results are compared to the state-of-the-art in terms of area and performance.

ÖZET

TAM RASGELE ERİŞİMLİ DIFERANSİYEL ARAMA TABLOLARI

Arama Tabloları (AT), cebirsel uygulamalara kıyasla karmaşık işlevlerin hesaplanmasında düşük gecikme sağlamak için, donanım ve yazılım tasarımında karmaşık işlevleri uygulamak için sıklıkla kullanılır. Bununla birlikte, bir AT alanı, girişin bit genişliği ile katlanarak büyür. Bu tezde, sürekli fonksiyonu uygulayan Konvansiyonel bir AT (KonvAT) ile aynı davranan, yeni bir alan etkili ve parametreli mantık mikro mimarisi sunmaktadır. Altı farklı mimari uygulanmakta, tüm mimariler orijinal AT'deki değerleri belli aralıklarla atlayarak saklar. Atlanan AT girişleri, aşağıdakilerden biri ile değiştirilir; Diferansiyel AT girişleri, kodlanmış diferansiyel AT girişleri veya zon katlama adı verilen bir yöntem. Bu üç örnek aynı zamanda farkların farklarından oluşan AT girdileri ile saklama yoluyla da uygulanmaktadır. Daha sonra bu yöntemlerden bazıları çok partili tablolar olan en son AT boyut azaltma yönteminde kullanılır. Bazı kombinasyonel mantık devrelerini kullanarak, tüm mimariler gecikme süresinde hafif bir uzlaşmayla bir KonvAT taklit edebilir. Önerilen mimariler tamamen rastgele erişime sahip ve "Tam Rasgele Erişimli Diferansiyel AT" (TR-dAT) olarak adlandırılmıştır. Daha sonra, çok partili tablolar yöntemi ile birleştirildi ve önerilen mimariler ile geliştirildi. FR-dLUT'un alanını ve performansını değerlendirmek için, sinüs ve 2^x işlevleri için tüm olası mimariler Verilog'da kodlanmış, doğrulanmış, sentezlenmiş ve FPGA üzerinde uygulanmıştır. Sonuçlar, alan ve performans bakımından en son teknolojiyle karşılaştırılmıştır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

lg	logarithm in base two
w_i	Input bitwidth
w_o	Output bitwidth
Δ	Delta
π	Pi value
3T	Three table
ConvLUT	Conventional lookup table
CCT	Column compression tree
ddLUT	Difference of differences lookup table
dLUT	Differential lookup table
FPGA	Field programmable gate array
FR	Fully random access
LUT	Lookup table
LSB	Least significant bit
mLUT	Main lookup table
MSB	Most significant bit
NR	Non-random access
PVL	Partial variable length
RoCoCo	Row and column compression
RTL	Register transfer level
SBTM	Symmetric bipartite table method
SR	Semi-random access
TIV	Table of initial values
TO	Table of offsets
VHDL	VHSIC hardware description language
VL	Variable length
XOR	Exclusive or
ZF	Zone folding

1. INTRODUCTION

Functions like sine, cosine, logarithm, reciprocal, square-root, and exponent is very important for digital signal processing, image processing, multimedia applications, and digital communication systems. Due to the computation complexity of these functions, Look-Up Tables (LUTs) are commonly used to achieve low latency in computation compared to algebraic implementations. In Figure 1.1 shows an example for a sine function, instead of computing the sine function, using a LUT will decrease the latency greatly. However, since these LUTs store precomputed values of the functions, the area of a LUT grows exponentially with the given inputs bitwidth and expected outputs bitwidth. With the increased area, overall systems latency can also increase. So, decreasing a LUTs size might save both time and space for the design.

$$\sin x = \cfrac{x}{1 + \cfrac{x^2}{2 \cdot 3 - x^2 + \cfrac{2 \cdot 3x^2}{4 \cdot 5 - x^2 + \cfrac{4 \cdot 5x^2}{6 \cdot 7 - x^2 + \ddots}}}}.$$

```
function lookup_sine(x)
    return sine_table[round(1000 * x / pi)]
```

Figure 1.1. Sine function evaluation and lookup

The design proposed in this thesis can offer significant area reduction if the LUT stores a continuous function. Significant area reduction is ensured if the differences between neighboring LUT entries are significantly smaller than LUT entries themselves. LUTs can be combined with some algebraic manipulations to lower the overall area of the design. The algebraic manipulations may be function-specific [1] or may be general-purpose [2]. However, these techniques cannot guarantee that their outputs match the original LUT bit by bit, especially when the output is wide. Our microarchitecture is complementary to techniques with algebraic manipulation, as it can be used to reduce the area of LUTs internal to these techniques.

The idea of storing the much smaller differences between neighboring LUT entries has been previously proposed in logic design context [3]. However, the circuit allows sequential access, i.e., location n+1 is output in each cycle if location n is output in the previous cycle. Which can be called a “Non-Random Access” differential LUT (NR-dLUT). The work in

[4] proposes, a similar method to NR-dLUT but instead of accessing location n+1 it can output any LUT location within the range [n-Δ, n+Δ] in each cycle if location n is output in the previous cycle. Which can be called, a "Semi-Random Access" dLUT (SR-dLUT), while [4] calls it "Compressed" LUT. In this thesis, our proposed idea is implementing "Fully Random Access" dLUT (FR-dLUT). Just like a conventional LUT, there is no restriction on which LUT location can be accessed in each cycle.

FR-dLUT consists of two type LUTs; one of them is the LUT where mid values are stored which is called main LUT, others are the LUTs where the differences are stored. Additionally, FR-dLUT is implemented with three additional methods. The first one uses variable length in the difference LUTs which is accomplished by using Huffman encoding, where each difference entry in the LUTs are encoded according to Huffman encoding algorithm (Entries with higher frequency receives less bits when encoded.). The second one uses a method which we called "Zone Folding". In this method, more than one encoded entry is stored in a single difference LUT by concatenating multiple encoded entries to a single entry. The last method stores the difference of the differences. In this method, there is a single main LUT like others, but there is also a single difference LUT, and other remaining LUTs store the difference of the differences. Shaded blocks in the left of the Figure 1.2 shows the contributions of these methods on top of the ConvLUT.

There have been other methodologies despite using differences to decrease the LUT area, such as using functional approximations and table driven methods. These methods include bipartite method [5-8] and multipartite method [9-10]. These methods decrease the area of a LUT greatly. Due to the large reduction in area, latency of the design also decreases. That is why these methods are considered as win-win in term of time and space trade off.

Bipartite method uses approximation of a function using two LUTs, table of initial values (TIV) and table of offsets (TO), and an adder. TIVs store the sampled function values which is sampled uniformly. Where TOs stores the difference of the actual values of the function and the initial values stored in TIVs. Due to the symmetry in offsets stored in TO, the size can be reduced by half. Also for further reduction in TO size, TOs can be partitioned into multiple smaller LUTs thus it is called the multipartite method. Each additional table increases the combinational logic complexity.

Bipartite method can be used for low precision function, since the LUT size exponentially increases with the function precision. Even, when multipartite method is used to decrease the size of TOs, TIVs still occupies most of the total area. Faithfully Rounded LUT [16] method has been proposed to decrease the TIV sizes. Which decomposes TIV into multiple tables, a table named TIV_{new} with middle values of the every 2^n value and two difference tables which takes difference with respect to the value stored in TIV_{new}. In this thesis, for further reduction in TIV size FR-dLUT method is implemented on the TIVs of the multipartite method. Contributions on top of TIV's are shown on the right side of the Figure1.1's shaded cells.

Like in every hardware design there are multiple steps in the design flow. First one is the generation of the RTL code of the proposed methods, which is done generally by Perl scripts during this thesis. Second part is the verification, for verification of the RTL code, a testbench and set of test values are generated. If the verification fails, RTL code is fixed until testbench is passed without an error. After the testbench, final step is synthesizing and implementing the design on an FPGA board to measure the timing and the area of design.

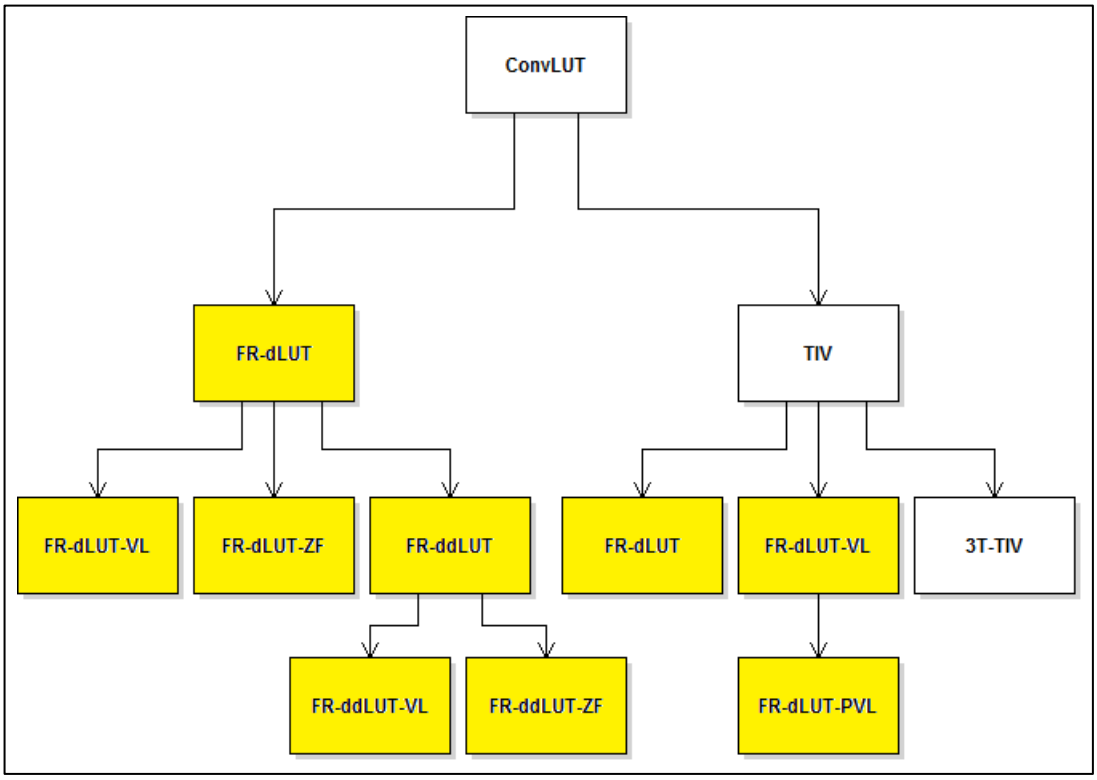

Figure 1.2. Contribution of the thesis

2. PREVIOUS WORK

In this part, previous works that are related to LUT size reduction is examined in more detail. There are four previous works closely related to the work done in this thesis and one work that is done on caches. The first one is [4], where SR-dLUT idea is proposed. The second one is [5-8], where bipartite method is proposed. The third one is [9-10], where multipartite method is proposed. The forth one is [16], where the authors propose an idea to reduce the TIV size of methods [5-10] by dividing TIV into three tables. Additionally, the column compression tree (CCT) idea used in this thesis is proposed in [10]. CCT idea and how it is being used in this thesis is explained. The work done on caches is the Base-Delta-Immediate compression in [18] where inside the caches instead of real data, differences of data are store in the caches. Method proposed in [18] is used for compressing data in on-chip caches.

2.1. SEMI-RANDOM ACCESS LOOKUP TABLE

In [4], the idea is achieving LUT size reduction with the loss of random access for a function. Main idea in this work is to store the difference of two consecutive outputs of a function instead of storing the actual value. This method is mostly ideal for transcendental functions, since difference between two consecutive outputs are much smaller than non-transcendental functions. Even though this method causes the loss of random access ability for a LUT, it still has some semi-random access capability due to the way differences are stored.

Block diagram of the overall design of SR-dLUT is shown in Figure 2.1. Which shows that the design is composed of three main modules. First one is the address generator, it takes the initial input and generates an address for the LUTs inside memory array and generates another output which indicates which outputs are selected for addition. Second module is the memory array, where the difference values generated from the given functions consecutive outputs are stored in delta (Δ) LUTs. Last one is data select unit, where the selection out of Δ LUTs output are done which is used during the final addition. Additionally, last received input and the last given output is stored in separate registers to be used on the next request.

Memory array consist of Δ parallel LUTs, where each LUT store the difference values. Figure 2.2 shows how the differences are stored in these LUTs. Since there are Δ parallel LUTs using the last output of the design it can reach Δ inputs above or below in a single cycle, thus the semi-random access. For example, the given input is five units above the last received input; if Δ is eight output can be received in the current cycle, else if Δ is four requested output is shown in the next cycle.

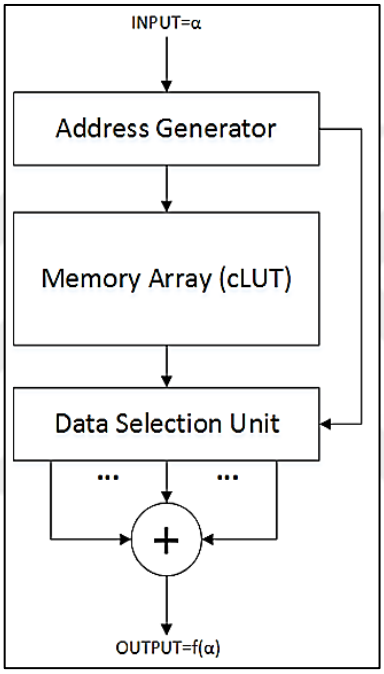

Figure 2.1. Block diagram of SR-dLUT algorithm [4]

LUT0	LUT1	LUT2	LUT3
F(1)-F(0)	F(2)-F(1)	F(3)-F(2)	F(4)-F(3)
F(5)-F(4)	F(6)-F(5)	F(7)-F(6)	F(8)-F(7)
F(9)-F(8)	F(10)-F(9)	F(11)-F(10)	F(12)-F(11)
F(13)-F(12)	F(14)-F(13)	F(15)-F(14)	F(16)-F(15)
⋮	⋮	⋮	⋮

Figure 2.2. The way differences are stored in the memory array when delta is four

Address generator module is used to calculate which memory address should be read and which LUTs outputs should be used in the final adder. Since there are Δ LUTs in memory

array if previous input is α, thanks to address generators output it is possible to access function values in range of [α-Δ, α+Δ].

Address generator initially subtracts previous input from the current input. Result of the subtraction consists of magnitude and sign. Sign differs whether a smaller or greater value input is requested with respect to previous input. Magnitude is used to determine the amount of shifting to be applied to thermometer vector. Thermometer is the vector that is used to decide which address to be read from LUTs inside the memory array by XORing thermometer value and the shifted result.

Figure 2.3 shows how XORed output of address generator is used to determine the memory location to be read for one of the LUTs inside memory array. Address to be read can be one more, one less, or equal to the current address in any given time for a single LUT. For example, if difference between current input and the previous input is any positive number address register is incremented by one.

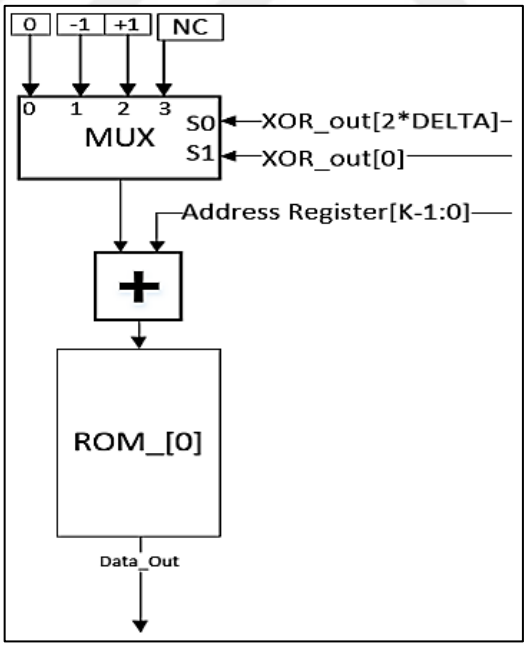

Figure 2.3. XORed output used for a single LUT in memory array [4]

Data selection unit is used to determine which values that has been read from Δ LUTs are used at the final addition. For example, if Δ is four and difference between current input (α+3) and previous input (α) is three (so that current input is three steps ahead), then three

values out of four values that came from memory array are added. Output of f (α+3) is calculated by adding these three values and the previous output f (α) in the final adder.

In [4], as a test parameter sine function with 16-bit input and 16-bit output resolution is used. With the expense of the random access ability with respect to conventional LUTs %75 area reduction is achieved.

2.2. BIPARTITE METHOD

In [5-8] bipartite table method, and in [7] symmetric bipartite table method (SBTM) is presented. For bipartite table approximation, f function is stored in two tables named TIV and TO. To get the approximation f (x), input x is divided into three segments; x0, x1, and x2, with bit lengths n0, n1, and n2, respectively. The value of x is equal to {x0, x1, x2} and if the bitwidth of x is n then n is equal to n0 + n1 + n2.

To generate TIV and TO tables, function is initially divided to multiple blocks. There is one entry in TIV for each block. Also, each block is divided into several segments. Figure 2.4 shows an example for a block, as shown in the Figure 2.4 a single block is segmented into four parts. Overlay segment graph shows how each segment is drawn if they start from the same point. The low and the high overlay segments, in this example segments 1 and 4, are shifted up and down by b. Where dotted curve shows the average of segment 1 and 4 as the result of shifting. This dotted curve is used for the values of that are going to be stored in TO. TIV values are the middle points of these dotted curves, and TO values are the differences between the middle one and the remaining ones.

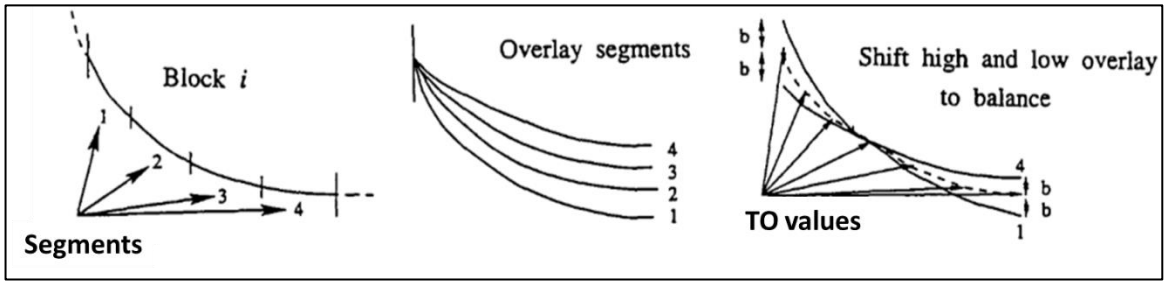

Figure 2.4. Segmentation of a block for generation of TIV and TO [5]

Most significant n0 + n1 bits of x are used for TIV as TIV (x0, x1), and most significant n0 and least significant n2 bits are used for the TO as TO (x0, x2). Carry-save approximation to f (x) is generated from the outputs for these two tables. Let output length of TIV and TO be p0 and p1 respectively. Generally, p0 > p1 so when a signed addition is required in the carry-propagate adder output of TO (x0, x2) is sign extended to p0 bits.

SBTM is using the symmetry and removal of leading zeros (ones if the number is negative) in TO to decrease the size of it with only using additional XOR gates. Block diagram of this method is shown in Figure 2.5. Since symmetry is used in TO, half of the entries in TOs are removed, thus input bitwidth of TOs are reduced by one (n0 + n2 - 1). Due to the removal of leading repeating bits, output bitwidth is reduced by one as well. The most significant bit of x2 is used to XOR the remaining n2 – 1 bits of x2. Since after half of the original TO values are read from the LUT, for the other half that is removed from the TO, input address that is used read from LUT must be decremented. For example, if n2 is three for each segment initially address 0, 1, 2, and 3 should be read from TO. For the next four values addresses 3, 2, 1, and 0 should be read from TO. Finally, to obtain the removed half of the original TO, most significant bit of x2 and the output of TO are XORed to get the ones complement of the output. Table 2.1 shows an example for cosine function, bolded values are stored in TO.

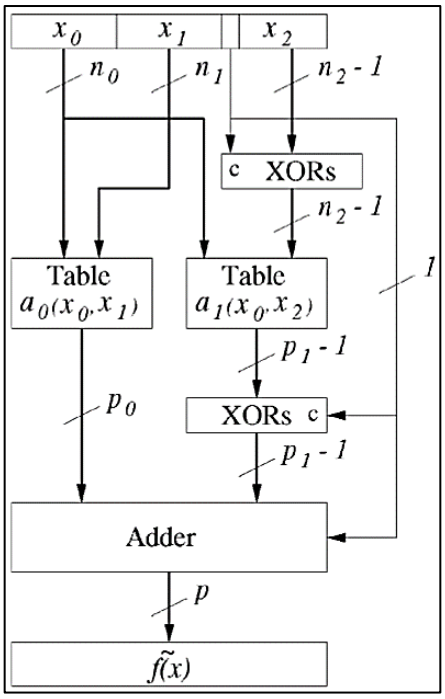

Figure 2.5. STBM module [7]

Table 2.1. Partial TO entries for cosine function [7]

x		TO (x_0, x_2)	
decimal	**binary**	**decimal**	**binary**
0.500000	0.1000000	+ 0.0166016	0.00000**1000**1
0.507812	0.1000001	+ 0.0107422	0.00000**0101**1
0.515625	0.1000010	+ 0.0068359	0.00000**0011**1
0.523438	0.1000011	+ 0.0029297	0.00000**0001**1
0.531250	0.1000100	- 0.0029297	1.1111111101
0.539062	0.1000101	- 0.0068359	1.1111111001
0.546875	0.1000110	- 0.0107422	1.1111110101
0.554688	0.1000111	- 0.0166016	1.1111101111

2.3. MULTIPARTITE TABLE METHOD

Multipartite method in [9], contains the basic characteristics of SBTM. In this method TIV still present in the design, but instead of having single TO there are m TOs. In case of m equals to 1 in multipartite is same as using bipartite. In multipartite method symmetry in TOs are still used for every m TO in the design. In multipartite method, same principal that is used on original LUT for the creation of TIV and TO in bipartite method is used on the created TO. In Figure 2.6 multipartite architecture is shown for m=3. Where w_o is the output bitwidth, w_i is the input bitwidth of each TO, and g is the number of guard bits used for faithful rounding.

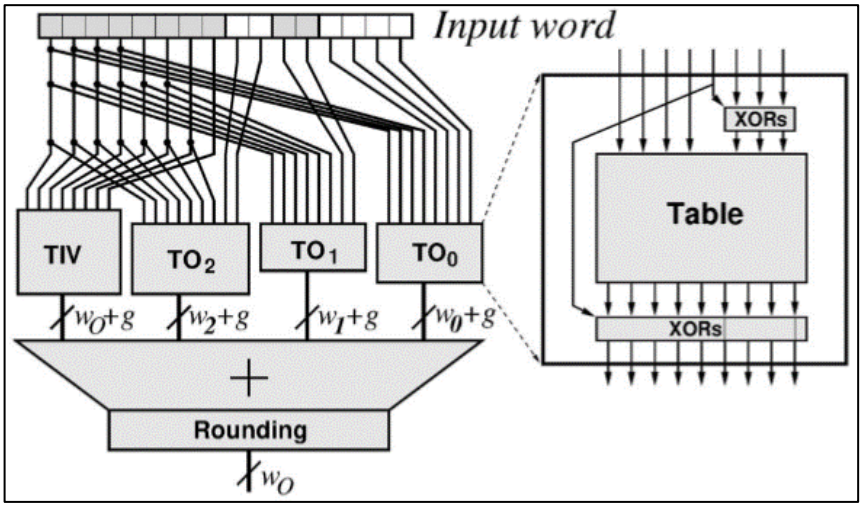

Figure 2.6. Multipartite architecture [9]

Guard bit calculation can be done by using the formula (2.1). Where c denotes the minimum output of the function, d denotes the maximum output of the function. For example, for a sine function with range [0, π/4), c is 0 and d is 1. ϵ_{approx}^{D} denotes the approximation error for each TO.

$$g = \left\lceil -w_O - 1 + \log_2 \frac{(d-c)m}{(d-c)2^{-w_0-1} - \epsilon_{approx}^{D}} \right\rceil \quad (2.1)$$

2.4. FAITHFULLY ROUNDED LOOKUP TABLE

As explained on the previous sections both SBTM and multipartite table method only decreases the size of the TO that is created. Also, it is important to note that both bipartite method and multipartite methods introduce an error due to the approximations that's done during the creation of tables. In [16], instead of decreasing TO sizes TIV size reduction is aimed, without introducing additionally error, thus the faithfully rounded tables.

In [16], there are two ideas a two-table method and the three-table method. In both the main aim is to decrease the TIV size by storing actual TIV values in one table named TIV_{new}, and storing difference values in table named TIV_{diff} for two-table method and in TIV_{diff1} and TIV_{diff2} for three-table method. Figure 2.7 show the decomposition of two-table and three-table methods.

Figure 2.8 show an example decomposition for eight consecutive entries in TIV into TIV_{new} and TIV_{diff}. Where TIV values are represented as B_i, i = 0, 1, …, 7. B_4 is the middle entry which is stored in TIV_{new}. Differences are denoted as D_i, i = 0, 1, …, 7, also $D_i = B_i - B_4$. Wordlength of D_i is usually smaller than wordlength of B_i. Also, original TIV values B_i can be recovered by adding D_i and B_4.

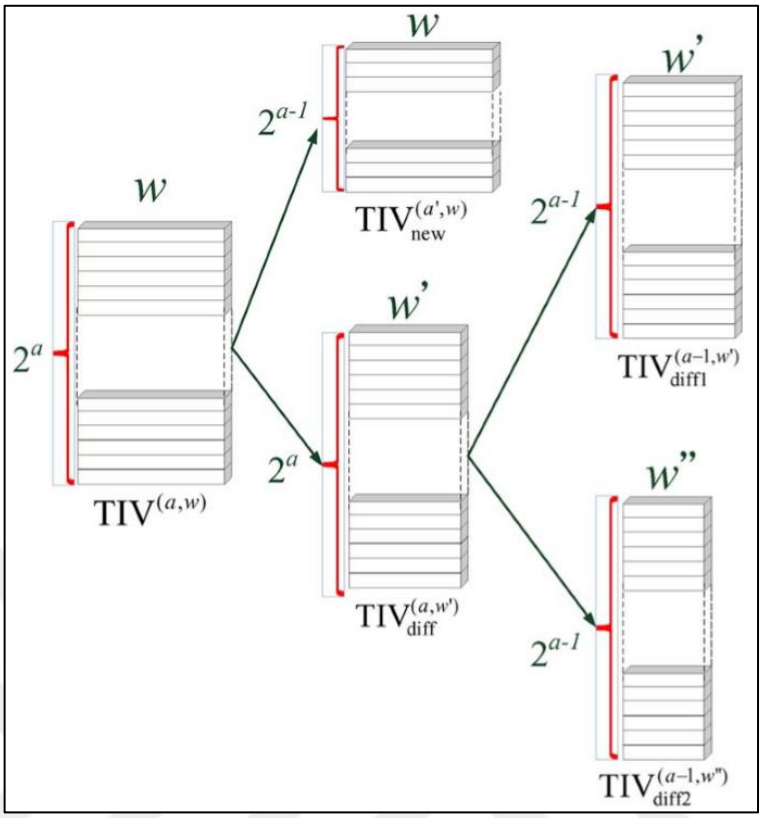

Figure 2.7. Decomposition of two-table and three-table methods [16]

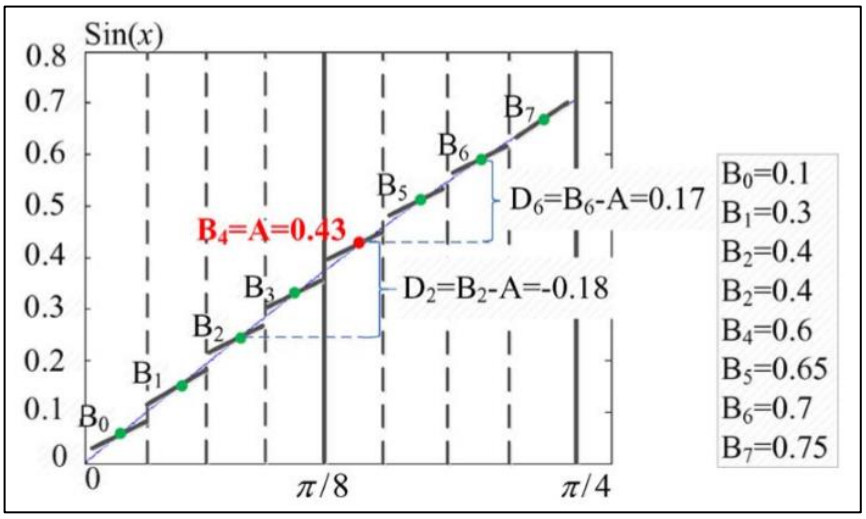

Figure 2.8. TIV decomposition into TIV_{new} and TIV_{diff} [16]

In two-table method, if consecutive 2^s entries are presented as a single entry in TIV_{new}, then the size of the TIV decreases from 2^{α} x ω to $2^{\alpha-s}$ x ω in TIV_{new}. Where α is the number entries

and ω is the wordlength. Since TIV_{new} only stores the middle values, number of entries in TIV_{new} is much less but wordlength of each entry is the same as original TIVs.

For the TIV_{diff} table, since there is a difference value to be stored for each entry in TIV, number of entries in TIV_{diff} and TIV are the same. Due to storing the differences in TIV_{diff}, wordlength of it is usually much smaller than TIV entries wordlength. Thus, the size of the TIV_{diff} is 2^{α} x ω′, where ω′ < ω.

For the three-table method (3T-TIV), TIV_{diff} is separated into TIV_{diff1} and TIV_{diff2}. Again, for eight consecutive entries in TIV_{diff} bottom half is stored directly in TIV_{diff1}. For TIV_{diff2} values, first entry is always zero and for the remaining three entries (for the case of using eight consecutive numbers in TIV_{diff}) differences of B_{4+i} and B_{4-i} are stored in TIV_{diff2}. An example of decomposition for TIV_{diff} into TIV_{diff1} and TIV_{diff2} is shown in Figure 2.9.

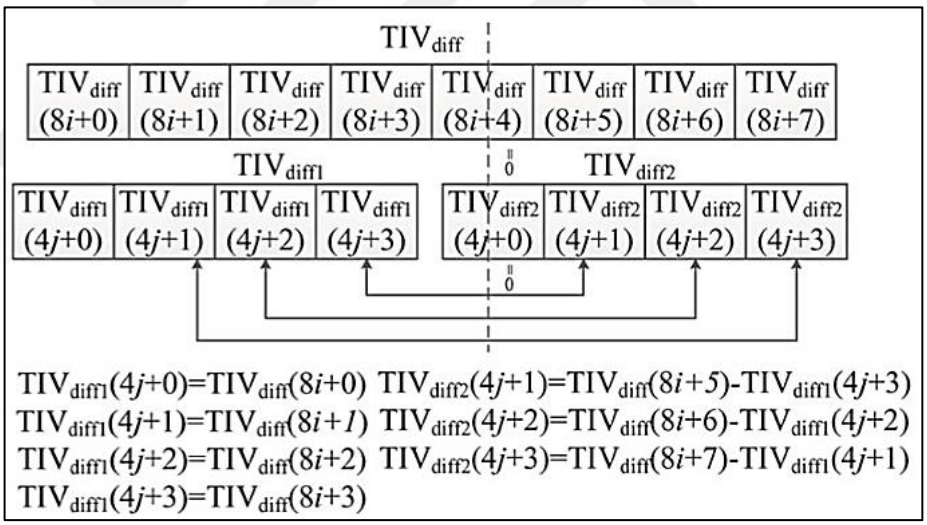

Figure 2.9. TIV_{diff} decomposition into TIV_{diff1} and TIV_{diff2} [16]

Algorithm for the creation of TIV_{new}, TIV_{diff}, TIV_{diff1}, and TIV_{diff2} from original TIV is shown in Algorithm 2.1. Before the creation of the tables every possible s value is tested from the range of 1 to α. And s is chosen, where the lowest total bit count is achieved from the newly created tables. Note that α represents number of entries in TIV, and s represents number of consecutive entries to be used for the creation of the tables as mentioned before.

Algorithm 2.1. Best decomposition of TIV in three-table method [16]

Given: $\mathrm{TIV}^{(a,w)}$ of size $2^a \times w$ with 2^a entries and wordlength w bits
Find: decomposed tables $\mathrm{TIV}_{\mathrm{new}}^{(a',w)}$, $\mathrm{TIV}_{\mathrm{diff1}}^{(a-1,w')}$, and $\mathrm{TIV}_{\mathrm{diff2}}^{(a-1,w'')}$

$Cost\_\min = 2^a \times w$;
for (i=0 to $a-1$) {
// the first part decomposes $\mathrm{TIV}^{(a,w)}$ into $\mathrm{TIV}_{\mathrm{new}}^{(i,w)}$ and $\mathrm{TIV}_{\mathrm{diff}}^{(a,w_i')}$
  $s = a - i$;
  for every 2^s consecutive entries in $\mathrm{TIV}^{(a,w)}$ {
    store the middle entry into $\mathrm{TIV}_{\mathrm{new}}^{(i,w)}$;
    compute differences between the middle entry and the other entries;
  }
  determine wordlength w_i' of all the difference values;
  generate table $\mathrm{TIV}_{\mathrm{diff}}^{(a,w_i')}$ that stores all the difference values;

// the second part decomposes $\mathrm{TIV}_{\mathrm{diff}}^{(a,w_i')}$ into $\mathrm{TIV}_{\mathrm{diff1}}^{(a-1,w_i')}$ and $\mathrm{TIV}_{\mathrm{diff2}}^{(a-1,w''_i)}$
  for every consecutive 2^s entries in $\mathrm{TIV}_{\mathrm{diff}}^{(a,w_i')}$ {
    store into $\mathrm{TIV}_{\mathrm{diff1}}^{(a-1,w_i')}$ the first half entries;
    compute differences between the first half and second half entries;
  }
  determine wordlength w_i'' for the differences between the two halfs;
  store into $\mathrm{TIV}_{\mathrm{diff2}}^{(a-1,w_i'')}$ the differences;

// the third part computes total cost and find the optimal decomposition
  compute table size in bits $Cost(i) = 2^i \times w + 2^{a-1} \times w_i' + 2^{a-1} \times w_i''$;
  if $Cost(i) < Cost\_\min$) {
    $Cost\_\min = Cost(i)$; $a' = i$;
  }
}

2.5. COLUMN COMPRESSION TREE

In this thesis, before of using normal adders, a much more speed-optimized and yet area-efficient Column Compression Tree (CCT) is used. CCT is proposed in [11] (called RoCoCo). RoCoCo handles only the summation of signed numbers. Main propose of this idea is to speed up multiplication operations by decreasing number off addition in the final adder. To do that both row and column compression is used. In RoCoCo instead of

propagating carry bits they are saved, thus the method is also called carry save tree. CCT consist of full (3-bit input) and half (2-bit input) adders. Originally full adders and half adder produces two outputs, one is the summation and the other is the carry. In CCT unless different logic levels are connected, there are no carry bit all bits treated as summation output. Thus, there is no longer a carry propagation delay. Until there is only two addend signal remains, CCT compresses every generated partial product. Finally, remaining two addends are added using a fast adder.

RoCoCo is compared with the Dadda Tree [12], Wallace Tree [13], and the Xilinx ISE's native multiplication operator. In [11], 22 cases are tested. 9 out of these 22 cases [11] is the fastest one between the unsigned multipliers mentioned. In this thesis, every method at least require addition of 3 (at most 21) numbers. Instead of using regular adders for the additions RoCoCo's CCT is used.

3. PROPOSED FULLY RANDOM ACCESS DIFFERENTIAL LOOKUP TABLES

FR-dLUT proposed in this thesis is similar to the two-table method explained in the Section 2.4, it does not introduce any errors and uses the differences to decrease the size of a table. However, in FR-dLUT instead of storing the differences between the middle entry and the remaining entries, difference of every consecutive entry is stored. Additionally, instead of having single difference LUT there are Δ parallel LUTs to be accessed simultaneously.

3.1. FULLY RANDOM DIFFERENTIAL LUT

The main idea of FR-dLUT for sine function is depicted in Figure 3.1. For example, any value between sin (8) to sin (15), are obtained directly from sin (12) by adding the differences to sin (12) as the following:

- sin (11) = sin (12) + (sin (11) - sin (12))
- sin (14) = sin (12) + (sin (13) - sin (12)) + (sin (14) - sin (13))

Note that sin (8) is obtained from sin (12), not from sin (4), in our FR-dLUT implementation. That allows the last difference LUT (dLUT) to be half the size of the other dLUTs. That's why they are shaded, hence not needed, entries for dLUT3 in Figure 3.1.

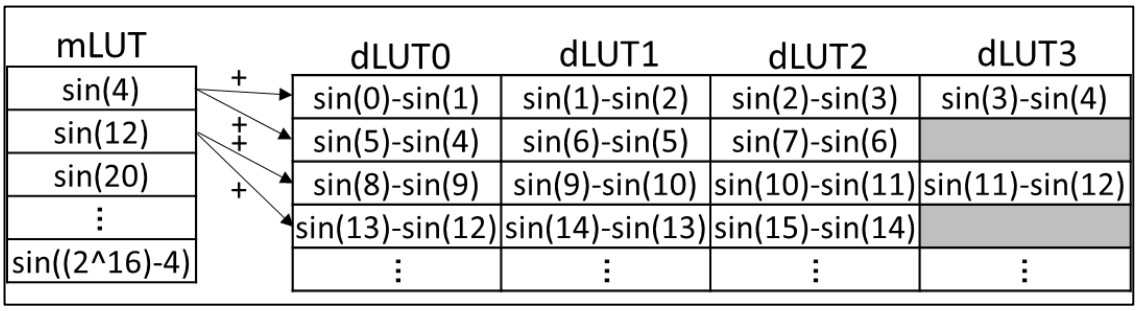

Figure 3.1. Conceptual depiction of FR-dLUT

The key difference between SR-dLUT and FR-dLUT is that FR-dLUT also contains a down-sampled version of the ConvLUT. This LUT is called the main LUT (mLUT). Like in SR-dLUT, differences are stored in Δ parallel dLUTs.

Consider a conventional LUT for sine function with n-bit input resolution (i.e., there are 2^n points between the input range of 0 and 2π or $\pi/4$ if symmetry is used) and k-bit output resolution. The number of bits required for the differences, m is equal to $\max(\lceil \lg(-mostnegdiff) \rceil, \lceil \lg(mostposdiff + 1) \rceil)$, where lg denotes $\log_2$, "mostnegdiff" denotes the most negative difference and "mostposdiff" denotes the most positive difference. Instead of k-bit output values of the implemented function, m-bit differences are stored, where m<k.

The top-level of FR-dLUT is shown in Figure 3.2 for a function f, with n-bit input, k-bit output resolution, and m-bit differences. The top module consists of five submodules, namely "Address Generator", "mLUT", "dLUT array", "Data Selection", and "Signed Summation". There are Δ number of dLUTs, where Δ is a power of 2 to make sure that address generation is simple.

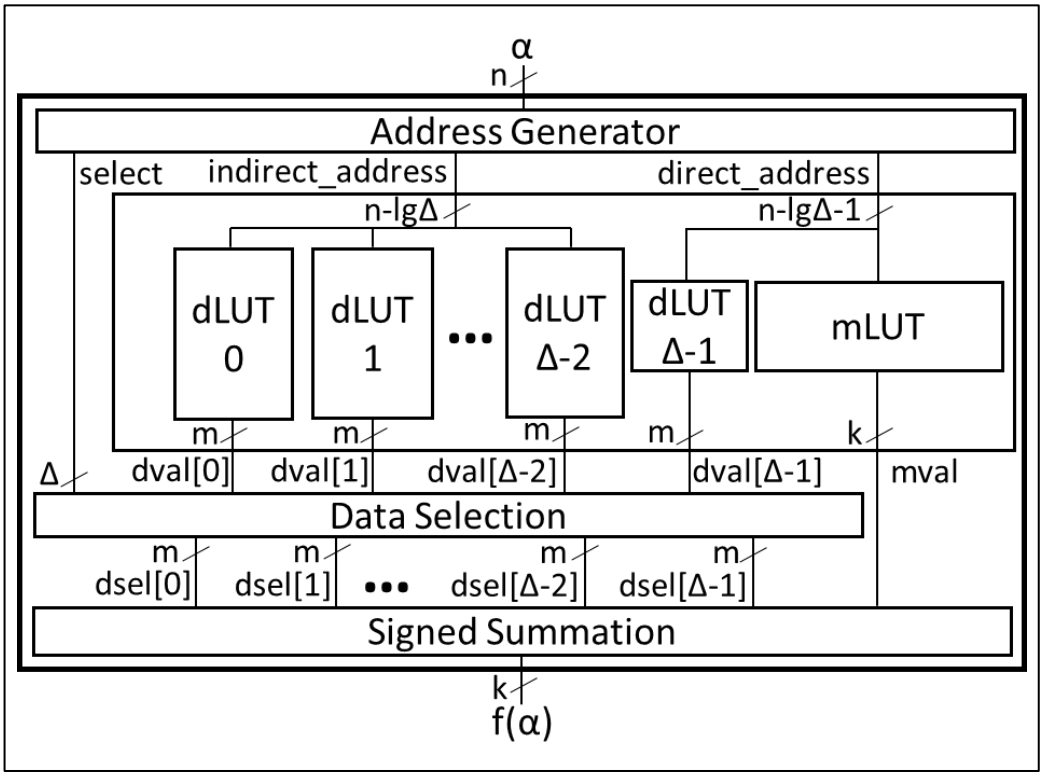

Figure 3.2. Top-level of FR-dLUT

mLUT stores k-bit signed f (α) for α = Δ, 3Δ, 5Δ, 7Δ, …, 2^n - Δ. The number of entries in mLUT is $2^{n-1}/\Delta$. The data line of mLUT is mval. The width of the address line of mLUT, named as direct_address, is n – 1 – lgΔ. dLUTs store m-bit twos complement (signed) differences. As shown in Figure 3.1, at an even dLUT address, f (α_x) – f (α_{x+1}) is stored, whereas at an odd dLUT address, f (α_{x+1}) – f (α_x), is stored, where $\alpha_x < \alpha_{x+1}$. The number of locations in each dLUT, except the last dLUT, is twice the number of locations in mLUT, which is $2^n/\Delta$. The last dLUT shares the address line of mLUT. The address width of the address line of other dLUT, named as indirect_address, is n - lgΔ. The data line of a dLUT is named as dval.

Address generator module that is shown in Figure 3.3, part-selects n-bit input α and generates direct_address signal for the address line of mLUT and indirect_address signal for the address line of dLUTs. α's most significant n - lgΔ bits represents indirect_address (α[n - 1 : lgΔ]) signal. α's most significant n - lgΔ + 1 bits represents direct_address (α[n – 1 : lgΔ + 1]) signal. Address generator has one more output, select. Signal select is generated by left-shifting Δ ones α[lgΔ - 1 : 0] positions and every bit of the result is XORed with α[lgΔ]. Basically, select signal is used to decide which difference values (dvals) are to be added to mval. For example, when Δ = 4, if sin (11) is required, shift amount is 3, α[lgΔ] is 0, and as a result select signal is $(1000)_2$ indicating that only dLUT3 is added to mval. If sin (14) is required, shift amount is 2, α[lgΔ] is 1, and as a result select signal is $(0011)_2$ indicating that dLUT0 and dLUT1 are added to mval.

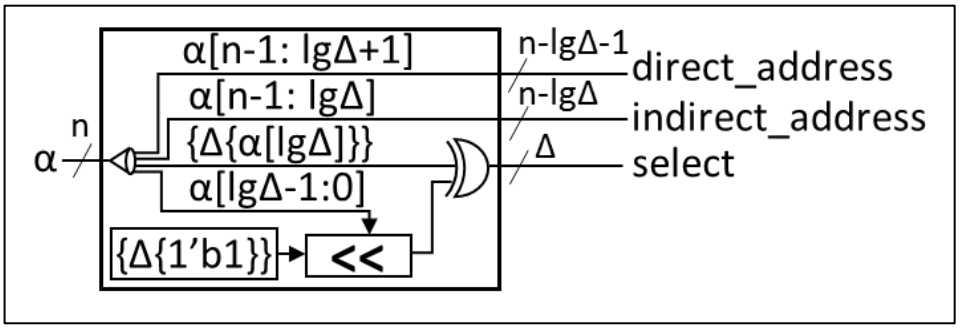

Figure 3.3. Address generator module for FR-dLUT

Data selection module is shown in Figure 3.4. Every dval goes thru this module. According to the select signal that comes from the address generator module dvals can be directly given

to the output of this module, which is dsel, or they can be removed by setting dsel to 0 for the corresponding dval. Selection of dvals with the select signal is done by using AND gates. For all Δ dvals existing, there is select signal. If select signals i^{th} bit is set than that means i^{th} dval is selected. Otherwise, if i^{th} signal is not set, i^{th} dval will not be selected and i^{th} dsel is reset. Where i is less than Δ. For example, if select signal is $(0011)_2$, then first and second dsel signals are equal to the corresponding dval signals and remaining dsels are zero.

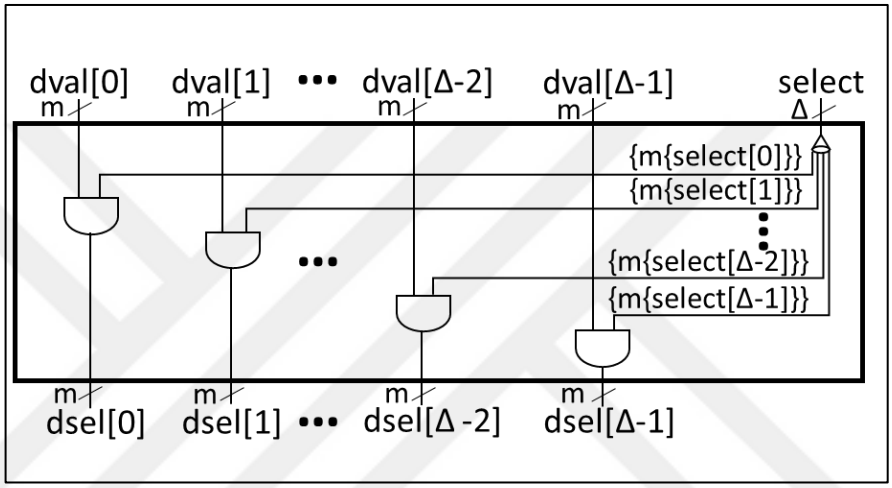

Figure 3.4. Data selection module FR-dLUT

Table 3.1. Truth table for $\underline{a} = \overline{a} + \underline{1}$

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

Before the signed summation module, since the method that is being used in FR-dLUT method consist of signed number and RoCoCo handles only the summation of unsigned numbers some conversions are required so that RoCoCo can be enabled to add signed numbers. The equality, $\underline{a} = \overline{a} + \underline{1}$, is shown as true in the Table 3.1, where $\underline{a}$ denotes the sign-bit and $\overline{a}$ denotes the inverse of the sign-bit. Consider the following summation of mval

and dsels for m = 4, k = 16, and Δ = 4 carried out using the equality in Table 3.1. Note that x denotes a "don't care" bit.

$$\begin{aligned}
&\underline{a}xxxxxxxxxxxxxxx + \underline{b}xxx + \underline{c}xxx + \underline{d}xxx + \underline{e}xxx \\
&\quad = \overline{a}xxxxxxxxxxxxxxx + \underline{1}0000000000000000 + \overline{b}xxx + \underline{1}000 \\
&\quad + \overline{c}xxx + \underline{1}000 + \overline{d}xxx + \underline{1}000 + \overline{e}xxx + \underline{1}000 \\
&\quad = \overline{a}xxxxxxxxxxxxxxx + \overline{b}xxx + \overline{c}xxx + \overline{d}xxx + \overline{e}xxx \\
&\quad + \underline{1}0111111111100000 \\
&\quad = \overline{a}xxxxxxxxxxxxxxx + \overline{b}xxx + \overline{c}xxx + \overline{d}xxx \\
&\quad + \underline{1}011111111110\overline{e}xxx
\end{aligned}$$

Each signed number shown above is transformed into a summation of an unsigned number (by reverting the MSB) and a signed constant. The sum of all signed constants ($\underline{1}0111111111100000$) is then merged with one of the numbers, i.e., $\underline{1}011111111110\overline{e}xxx$. Generalizing the above example to k-bit mval and Δ number of m-bit dsel, the signed constant is an (k + 1)-bit signed number and equal to the concatenation of bit chunks, $\underline{1}0$, (k – m – lgΔ)-bit ones, and (lgΔ + m – 1)-bit zeros. An example for the method used is shown in Figure 3.5.

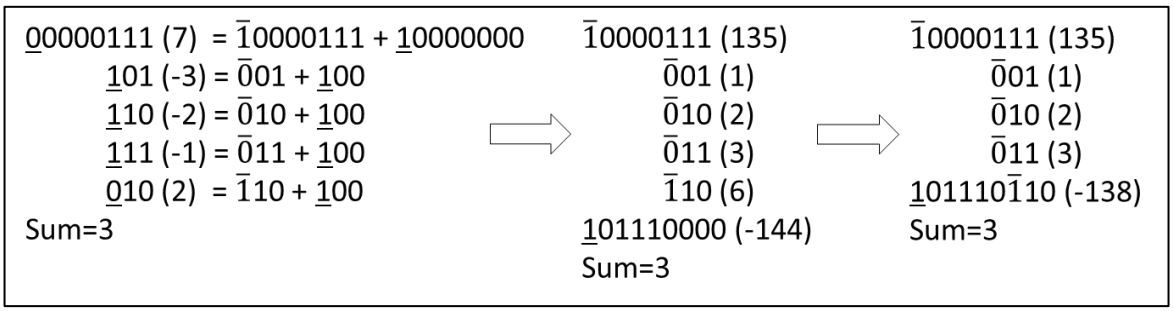

Figure 3.5. Example for signed summation method

Signed summation of FR-dLUT shown in Figure 3.6 is composed of CCT and Final Adder. As it can be seen in Figure 3.6, k-bit mval and Δ number of m-bit dsels are converted into unsigned numbers, and the constant is merged with the first dsel (dsel[0]) based on the summation technique explained above. Since the number of bits in the final sum cannot exceed k bits the MSB of the signed constant is excluded before it is fed to CCT (shown in

Figure 3.6). Therefore, there are now Δ – 1 number of m-bit and two k-bit numbers. One of the k-bit number is the mval received from mLUT and the concatenation of bit chunks, 1-bit zero, (k – m – lgΔ)-bit ones, (lgΔ – 1)-bit zeros and the first dsel. Then, CCT in Figure 3.6 reduces the summation of (Δ – 1) numbers of m-bit numbers and two k-bit numbers to the summation of two (k + 2)-bit numbers assuming (m + lgΔ) ≤ k. These two CCT outputs are named as cct1 and cct2. Similarly, since the number of bits in the final sum cannot exceed k-bits, two MSBs of these (k +2)-bit numbers are not fed to the final adder. Then, the final adder adds two k-bit numbers, and similarly, MSB is discarded from the final sum to obtain the k-bit f (α).

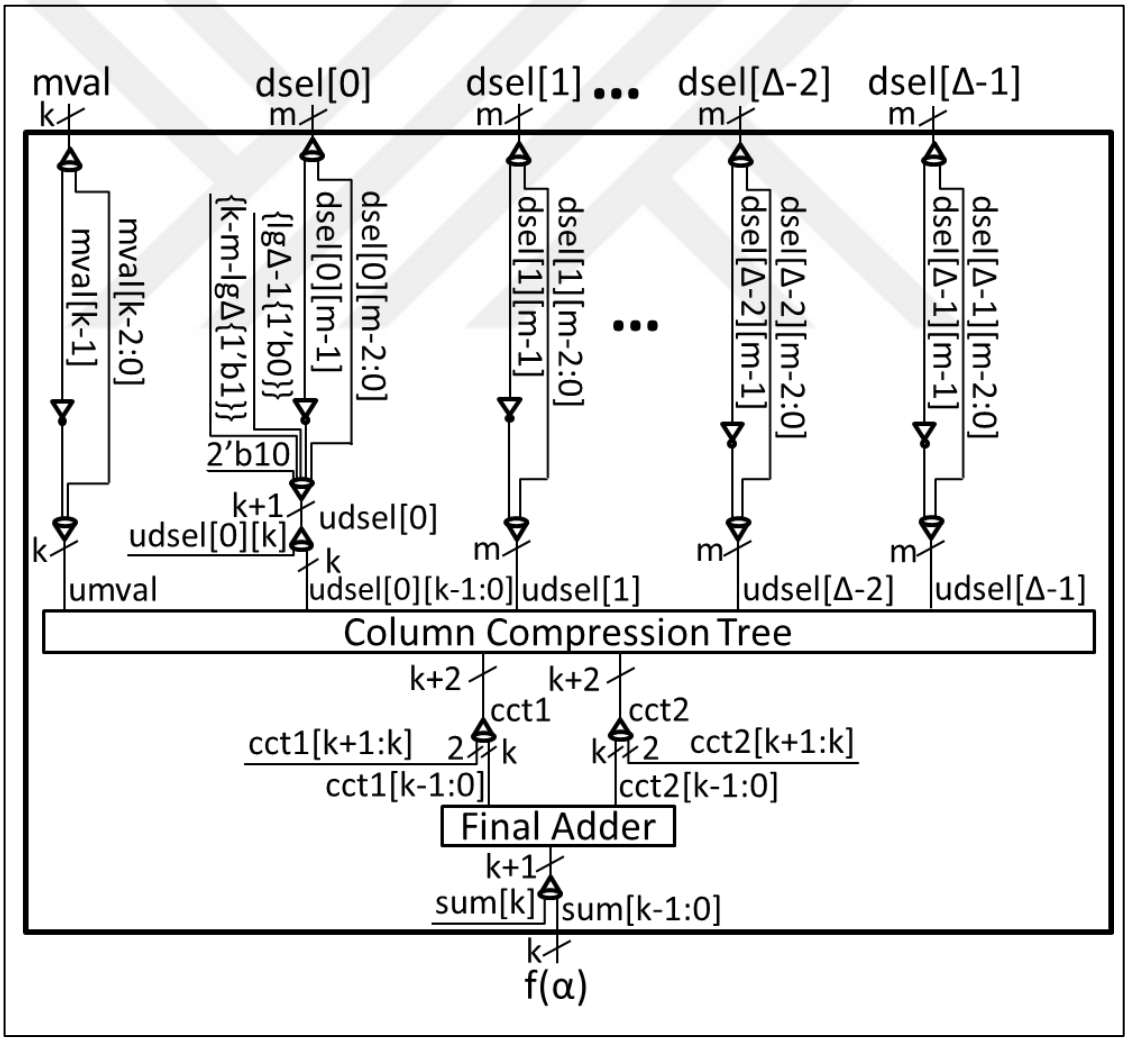

Figure 3.6. Signed summation module FR-dLUT

3.2. VARIABLE LENGTH DIFFERENTIAL LUT

In variable length differential lookup table (FR-dLUT-VL) method same functionalities of FR-dLUT is still present. There are Δ dLUTs, fully random access still exists, mLUT contains the same values in both methods, and before the final adder values go thru CCT. The difference of FR-dLUT-VL from FR-dLUT is the values stored inside the dLUTs. Figure 3.7 shows the conceptual depiction of the FR-dLUT-VL.

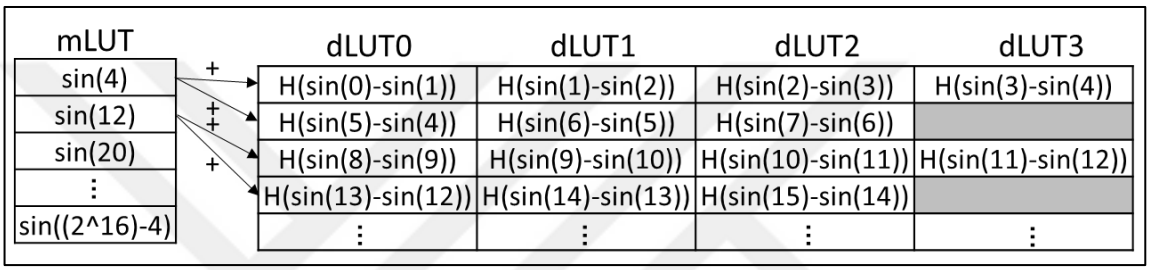

Figure 3.7. Conceptual depiction of the FR-dLUT-VL

For the values stored inside the dLUTs, any decompression method can be used. In this thesis, Huffman coding is used. For each dLUT the values frequencies are calculated. According to the frequency each value is assigned with a Huffman code. Although each frequency of each value in a dLUT is calculated separately from the other dLUTs. Every time the order of frequencies are the same for the all designs. Additionally, due to the encoded values stored in the dLUTs, after each value is read from the dLUT it needs to go through a decoder module. The top-level of FR-dLUT-VL is shown in Figure 3.8.

Decoder module in FR-dLUT-VL gets Δ h-bit inputs. These h-bit inputs are the dvals. The reason they are not m-bit anymore is that these dvals are the Huffman encoded values which may differ in size. Usually smallest one is much less than m. dvals go through a decoder that is generated during the calculation of frequencies. Output of this module is the decoded dvals (d_dval), which is again m-bits. An example selection unit is shown in Figure 3.9, where minimum difference value is -4 and maximum difference value is +4. It can be seen that most frequent number in this case is +3 and the most infrequent numbers are +4 and -4.

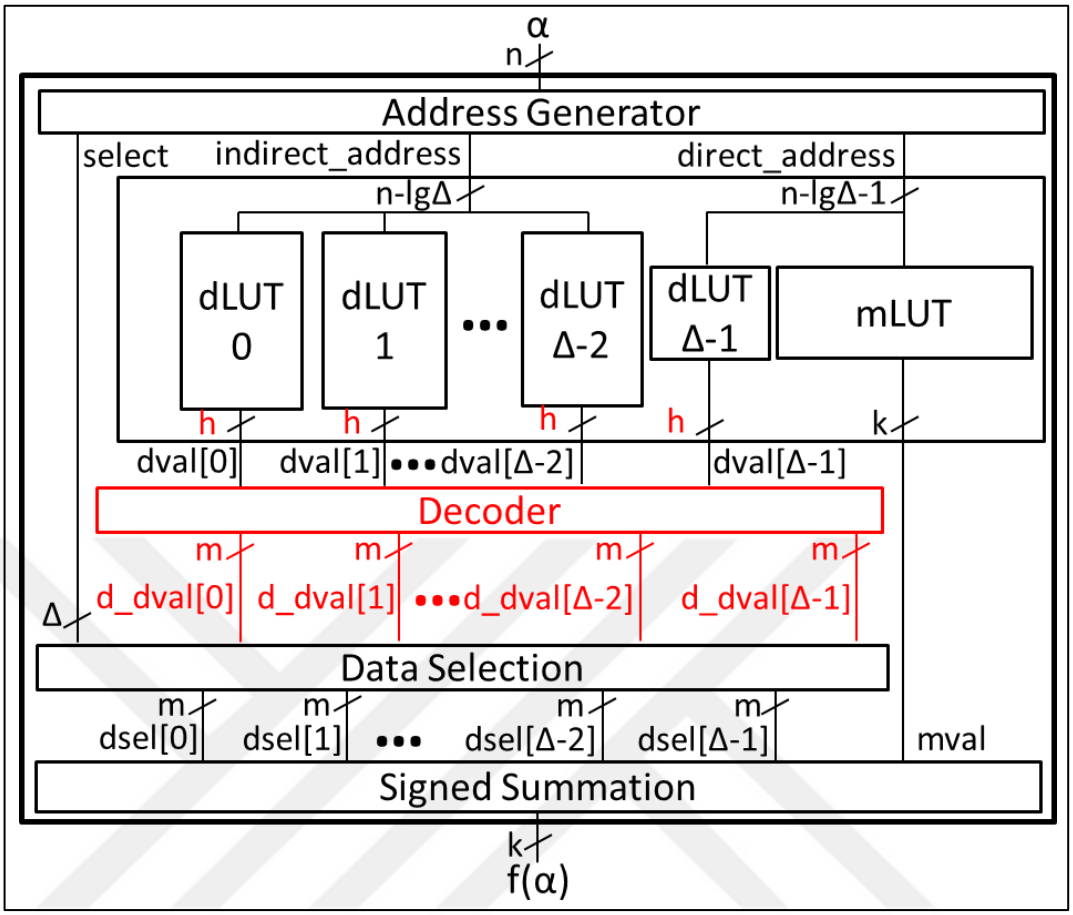

Figure 3.8. Top-level of FR-dLUT-VL

```
casex (dval)
                5'bxxx10 : out = 4'b0011;
                5'bxx111 : out = 4'b1101;
                5'bxx011 : out = 4'b0010;
                5'bxx101 : out = 4'b1110;
                5'bxx100 : out = 4'b0001;
                5'bxx000 : out = 4'b1111;
                5'bx1001 : out = 4'b0000;
                5'b10001 : out = 4'b0100;
                5'b00001 : out = 4'b1100;
endcase
```

Figure 3.9. Decoder example with range [-4, +4]

3.3. VARIABLE LENGTH DIFFERENTIAL LUT WITH ZONE FOLDING

In this method on top of FR-dLUT-VL, a method we called “Zone Folding” (FR-dLUT-ZF) is applied, where a similar method in computer architecture is called “narrow-bitwidth” operands [17]. The main idea is concatenating multiple encoded values to a single entry in the dLUT. Real differences from these encoded and concatenated entries fed to an improved decoder. Number of concatenated entries represents folding rate. For example, if two entries in a dLUT concatenated in a single entry its folding rate is 2. Also, each dLUT might have different folding rate.

Selecting between concatenated multiple entries is done by the fold_num signal that is generated in the address generator module. New address generator module can be seen in Figure 3.10. Let maximum folding in the design be max_fold. Signal fold_num is selected from α’s bit range of lgΔ to lgΔ + lg(max_fold) (α[(lgΔ + lg(max_fold)):lgΔ]). Value of max_fold is calculated during generation of the RTL. Also, folding rate of each LUT is selected as the maximum bit count reduction achieved for that dLUT. Finally, for the addresses to be read for each dLUT changes due to the reduction in total address. For indirect_address or direct_address signals, if the folding rate of a dLUT is n, first n bits of the signals are skipped and the rest is given the dLUT’s address port.

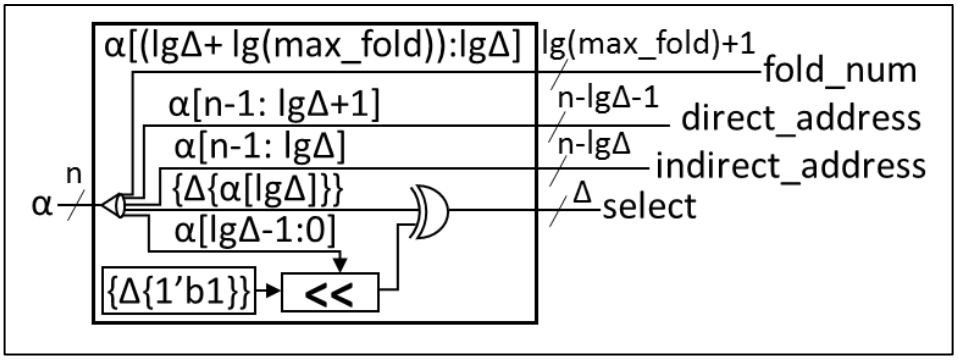

Figure 3.10. Address generator of FR-dLUT-ZF method

Decoders of the ZF method gets the output of the dLUTs and the fold_num signal that comes from the address generator. According to the fold_num first decoded entries are encoded and the entry which is the fold_numth number in the dLUTs output is given as an output from decoder. For example, if the folding rate of a dLUT is 2, which means each dLUT entry

consist of 2 encoded values concatenated together. When fold_num signal is 0, first decoded number from the output of the dLUT is selected as output from the decoder module. Otherwise, when the fold_num is 1, first encoded in the dval is decoded and the remaining bits are decoded once more so the second value can be received and send as the output. Figure 3.11 shows and example of the decoder with folding rate 2.

```
if(fold_num==1'b0) begin
    casex (in[FOLDEDWIDTH-1:FOLDEDWIDTH-HUFFMANWIDTH])
        5'b10000 : out = 4'b1100;
        5'b00xxx : out = 4'b1101;
        5'b110xx : out = 4'b1110;
        5'b011xx : out = 4'b1111;
        5'b1001x : out = 4'b0000;
        5'b010xx : out = 4'b0001;
        5'b101xx : out = 4'b0010;
        5'b111xx : out = 4'b0011;
        5'b10001 : out = 4'b0100;
    endcase
end
else if(fold_num==1'b1) begin
    casex (in[FOLDEDWIDTH-1:FOLDEDWIDTH-HUFFMANWIDTH])
        5'b10000 : temp = {in[FOLDEDWIDTH-6:0],2'bxx};
        5'b00xxx : temp = in[FOLDEDWIDTH-3:FOLDEDWIDTH-3-HUFFMANWIDTH+1];
        5'b110xx : temp = in[FOLDEDWIDTH-4:FOLDEDWIDTH-4-HUFFMANWIDTH+1];
        5'b011xx : temp = in[FOLDEDWIDTH-4:FOLDEDWIDTH-4-HUFFMANWIDTH+1];
        5'b1001x : temp = {in[FOLDEDWIDTH-5:0],1'bx};
        5'b010xx : temp = in[FOLDEDWIDTH-4:FOLDEDWIDTH-4-HUFFMANWIDTH+1];
        5'b101xx : temp = in[FOLDEDWIDTH-4:FOLDEDWIDTH-4-HUFFMANWIDTH+1];
        5'b111xx : temp = in[FOLDEDWIDTH-4:FOLDEDWIDTH-4-HUFFMANWIDTH+1];
        5'b10001 : temp = {in[FOLDEDWIDTH-6:0],2'bxx};
    endcase
    casex (temp)
        5'b10000 : out = 4'b1100;
        5'b00xxx : out = 4'b1101;
        5'b110xx : out = 4'b1110;
        5'b011xx : out = 4'b1111;
        5'b1001x : out = 4'b0000;
        5'b010xx : out = 4'b0001;
        5'b101xx : out = 4'b0010;
        5'b111xx : out = 4'b0011;
        5'b10001 : out = 4'b0100;
    endcase
end
```

Figure 3.11. FR-dLUT-ZF decoder example with folding rate 2

3.4. LUTS WITH DIFFERENCE OF DIFFERENCES

LUTs with difference of differences method (FR-ddLUT) instead of storing difference of two consecutive entry, difference between two consecutive difference is stored. Every LUT except the first one (dLUT0) stores the differences of the differences, but the first LUT still stores the original differences. Figure 3.12 shows an example for the values stored in dLUTs with ddLUT method when applied on top of FR-dLUT-VL. FR-ddLUT can be applied on top of original FR-dLUT, FR-dLUT-VL, or FR-dLUT-ZF.

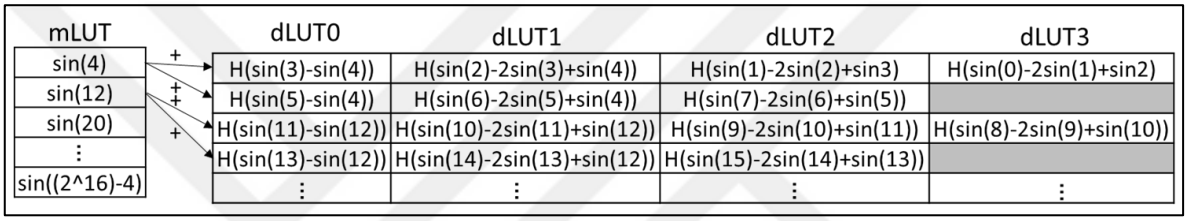

Figure 3.12. Conceptual depiction of the FR-ddLUT-VL

Another difference between FR-ddLUT and FR-dLUT is that in FR-ddLUT values stored in every odd address is reversed between LUTs. As shown in Figure 3.12 dLUT0's first entry is sin (3) - sin (4) where it is sin (0) - sin (1) in Figure 3.1. The reason these values are reversed is, to store all the difference (not the difference of differences) values in a single LUT.

Top-level view of FR-ddLUT-VL can be seen in Figure 3.13. As we are storing differences of differences in the dLUT there is a need for a multiplication operation to produce a result. For example, if α is 9 or 14, sin (9) or sin (14) is generated as the following:

- sin (9) = sin (12) + 3x(sin (11) - sin (12)) + 2x((sin (10) - 2sin (11) + sin (12)) + ((sin (9) - 2sin (10) + sin (11))
- sin (14) = sin (12) + 2x(sin (13) - sin (12)) + (sin (14) - 2sin (13) + sin (12))

Due to the required multiplication, the address generator module has been changed and a multiplication module has been added to the design. This introduced multiplication is expected to increase the latency for the overall design.

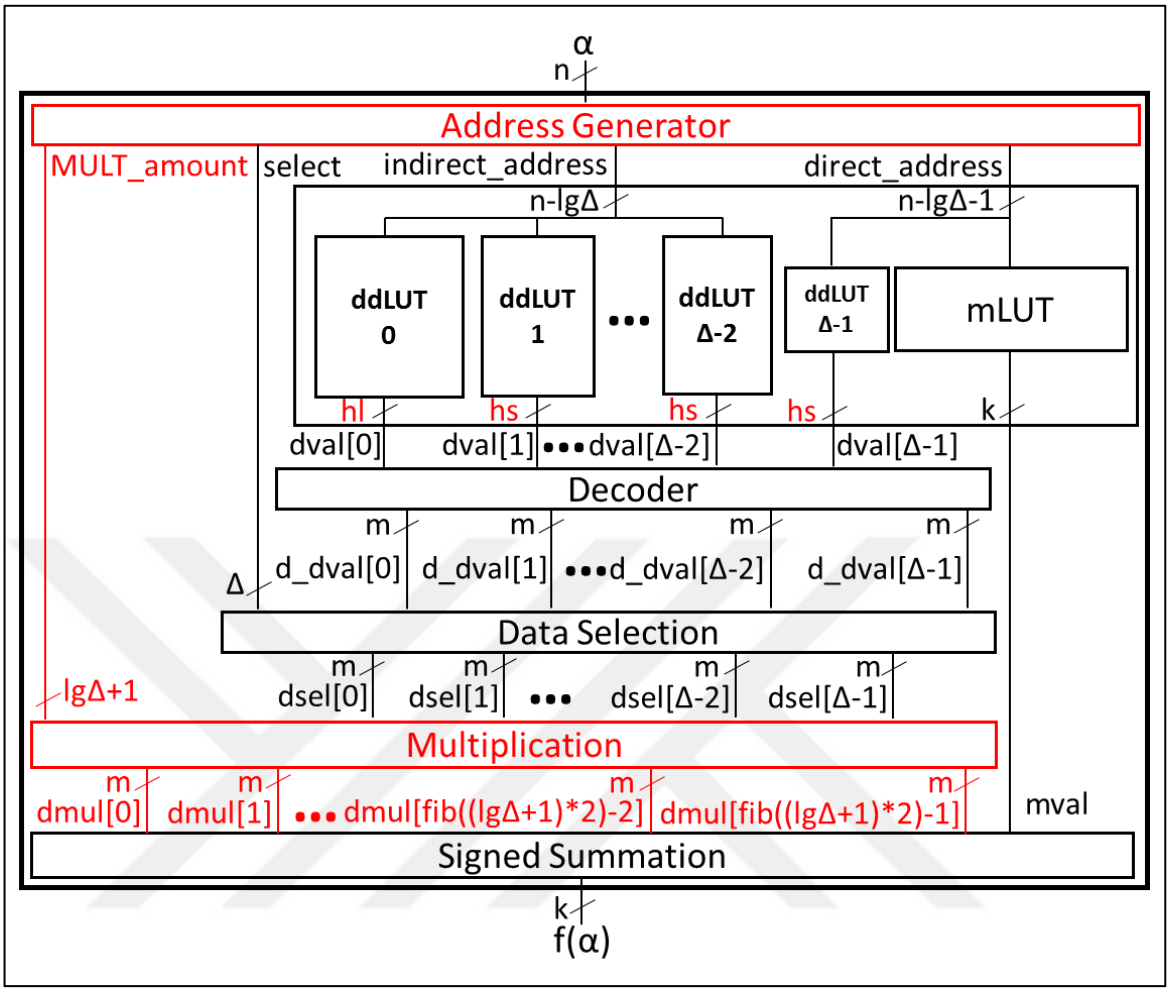

Figure 3.13. Top-level of FR-ddLUT-VL

New address generator module is shown in Figure 3.14. Signals indirect_address and direct_address are still the same as the FR-dLUT. Since the ordering of the values inside the ddLUTs are changed, select signals generation is also changed. Instead of XORing the left shifted Δ ones and the α[lgΔ], Δ ones are right shifted by the amount of either α's least significant (lgΔ + 1)-bits (α[lgΔ:0]) or the result of subtraction of Δ minus α's least significant lgΔ bits. The selection between two is done by looking α[lgΔ], in other words is based on whether indirect address is even or odd. If the indirect address is odd (an odd entry is read from the ddLUTs) results of subtraction is taken as shift amount, otherwise least significant bits of α is taken. For example, if α is 9 and Δ is 4, α[lgΔ] is 0 indicating that an even address is going to be read from ddLUTs, so the shift amount is $(001)_2$ which is the least significant (lgΔ + 1)-bits of α. Finally, select signal is $(0111)_2$, which is what we wanted. If we look at an example where the difference is used, like when α is 14. α[lgΔ] is 1 indicating

an odd address is going to be read from ddLUT. Shift amount is 4 minus 2 (since least significant lgΔ bits of α is $(10)_2$), which is 2. So, the select signal is $(0011)_2$, which is again what we wanted.

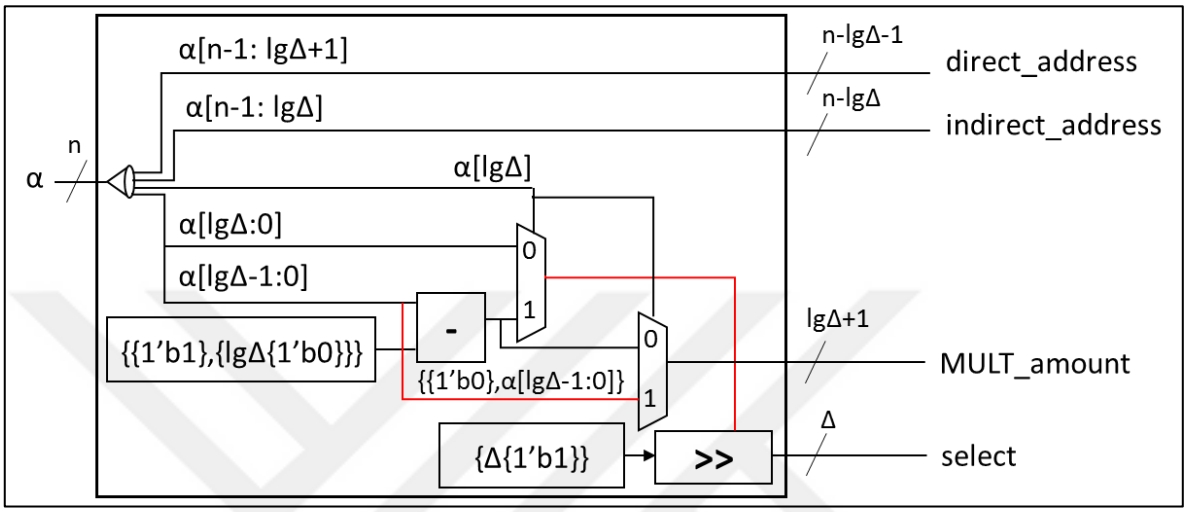

Figure 3.14. Address generator for FR-ddLUT

The new output of FR-ddLUT's address generator is multiplication amount (MULT_amount) signal. What this signal represents is the maximum multiplication that is required for a given input α. For the calculation of the signal, signal can be either least significant lgΔ bits of α (α[lgΔ-1:0]) or the subtraction result calculated during the generation of the select signal. Selection between two is again done by the α's lgΔ bit (α[lgΔ]) and correspond to memory address to be read from ddLUT is odd or even. If it is odd, α's least significant bits selected, else the subtraction result is selected. As an example, let's look at when Δ is 4 and α is 9 first. Since α[lgΔ] is 0 to generate the MULT_amount signal, subtraction result is calculated by subtracting least significant lgΔ bits of α from the Δ, so it is 4 minus 1 (since least significant lgΔ bits of α is $(01)_2$), which is 3. So, the maximum multiplication required when α = 9 is 3, which is correct. If α is 14, since α[lgΔ] is 1 MULT_amount signal is equal to the least significant lgΔ bit of α, which is 2 and correct.

Multiplication is done by creating multiple new signals from a single signal. Figure 3.15 shows the block diagram of multiplication module. Each dsel signal that enter multiplication module is ANDed with every bit of the multiplication amount that correspond to the signal. There are Δ dsel signal, so the maximum multiplication for a signal can be Δ. Since when all

Δ dsel signals are selected dsel that comes from the first ddLUT (ddLUT0) need to be multiplied Δ times. Maximum possible multiplications will decrease thru the last ddLUT, second ddLUT's dsel can be multiplied by Δ - 1, and so on till the last ddLUT's dsel which can be multiplied by 1. Like the maximum possible multiplication of each dsel signal, for a given input α signal MULT_amount represents the current maximum multiplication and decreases starting from first ddLUT up to the last used ddLUT. Last used ddLUT's dsel signal is multiplied with the 1. Figure 3.16 shows an example of how single multiplication is done for the first ddLUT's dsel signals multiplication case where α is 9 and Δ is 4. As seen, by using AND gates and appending some 0's to the left multiplication is achieved. Finally all these signals are added. That's why each dsel (except the last ddLUT's dsel) outputs multiple signals. Total number of outputs for multiplication module is calculated with formula (3.1).

$$\sum_{k=0}^{\Delta}(1+\lfloor \log_2 k \rfloor) \tag{3.1}$$

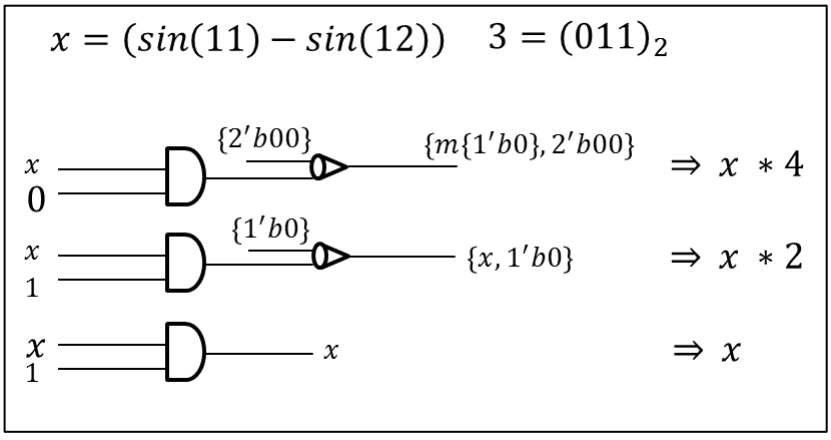

Figure 3.15. Single dsel multiplication

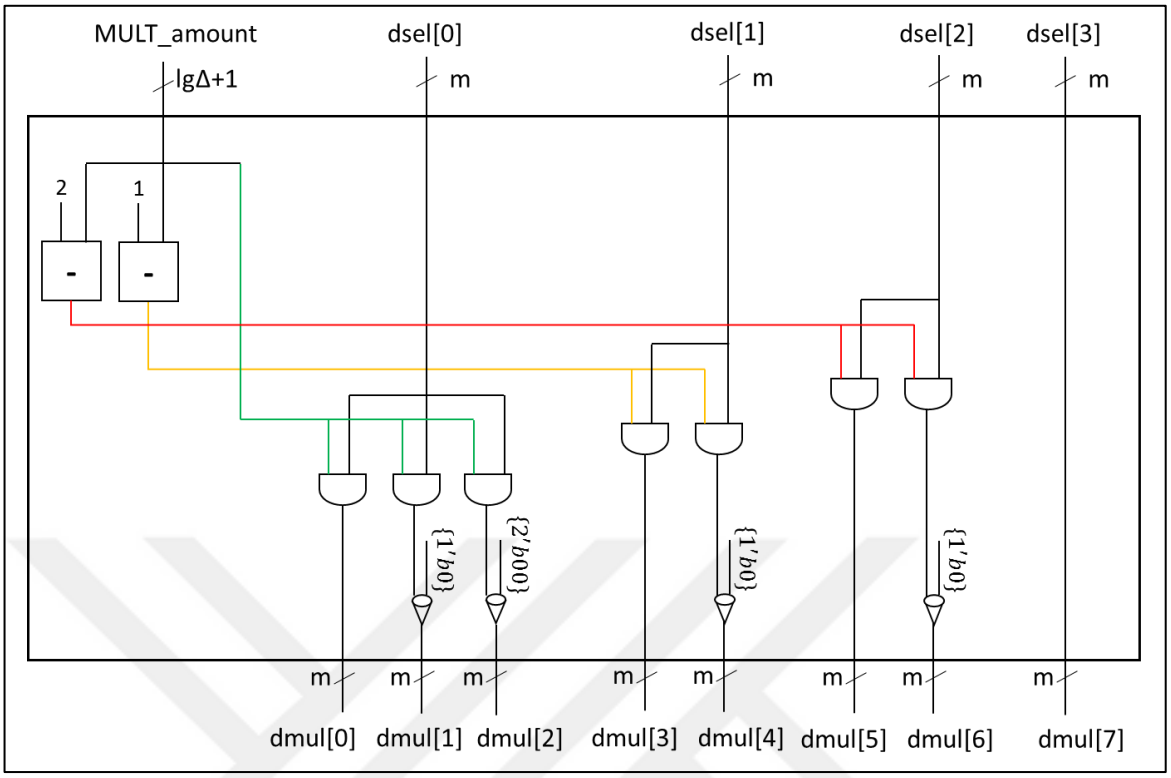

Figure 3.16. Multiplication module for ddLUT when Δ is 4

3.5. TIV SIZE REDUCTION

Multipartite methods are the current state of the art methods for LUT size reduction. Where a LUT is partitioned into multiple tables called TIV and TOs. During the creation of the TIV and TOs multipartite method introduces an error due to the approximation of the function. Instead of applying the proposed methods for ConvLUTs, proposed methods can be applied to TIV's of these multipartite methods. Since the size of a TIV in a multipartite method is much larger than the TOs or the rest of the design, aiming to decrease the size of TIVs are much more applicable than trying to decrease the size of TOs. For the creation of dLUTs, TIV entries are used. Methods implemented by TIVs are the FR-dLUT and the FR-dLUT-VL. Also, as an addition the partial variable length differential lookup table (FR-dLUT-PVL) method is implemented.

There are m TOs in a multipartite architecture, where each TO's output and the output of the TIV are added, every time an input is requested. Also, in the dLUT method there are Δ dLUTs and a mLUT, where again their outputs are added each time an input (in this case

input is a TIV entry) is requested. Instead of adding these numbers separately, the TO values are also fed to the CCT in the summation module of dLUT.

3.5.1. Partial Variable Length Differential LUT

In FR-dLUT-PVL method, instead of applying encoding methods to the whole entry, a part of the entry is taken and then the encoding method is applied. Encoding is done by separating part of each entry and using them as an input for the encoding method that is being used. For example, if three MSBs are selected, each entries' three MSBs are given to the encoding method. For separation, starting from the two MSB of an entry to the 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 size and the bit size of the remaining bits that are not used in the encoding. Among these combinations, the one with the lesser bit count is selected for the implementation. Algorithm 3.1 also shows how the selection is done.

In PVL values stored in the dLUTs can be categorized in two segments one is the encoded part, the other is some of the LSBs of original value. Encoded part is sent to the decoder same as the VL method. After the encoded value is decoded in the decoder, output bits and the second segment of the stored value in dLUT (LSBs of the original value) is concatenated and send to the data select module. Top module of PVL method is shown in Figure 3.17.

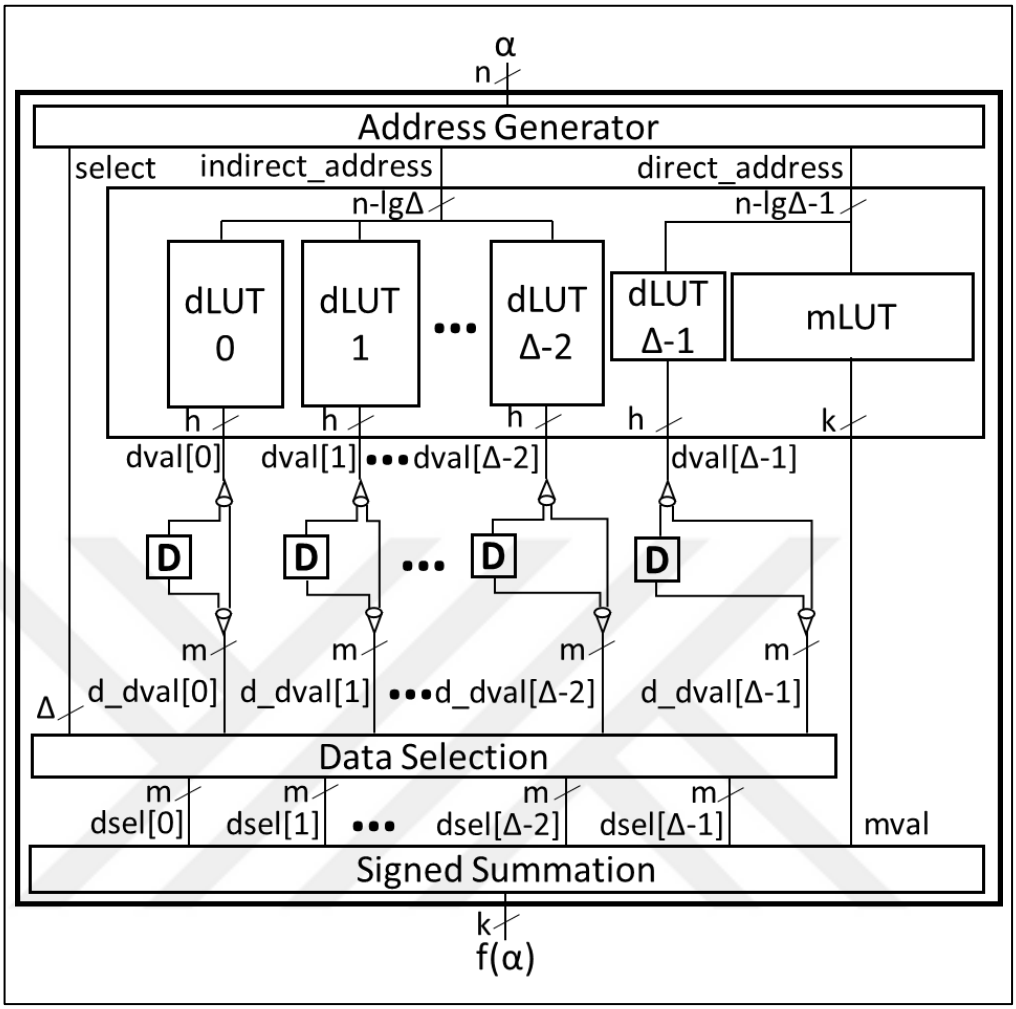

Figure 3.17. Top module of FR-dLUT-PVL method

4. RTL CODE GENERATORS AND VERIFICATION

In this thesis, generation of the RTL code has been done with the help of scripts. After the generation of the RTL codes, before the synthesis, these codes are verified using simulation tool of Xilinx ISE (ISim). For the verification, for each input in the given range of the function, expected outputs are generated and compared with the output of the proposed methods design. There is a different flow for creation of the dLUT's from ConvLUT and creation of dLUT's from the TIV's of multipartite method.

4.1. FOR CONVENTIONAL LUTS

There are two stages for the creation of dLUT's from ConvLUT's. First stage is the generation of dLUTs or ddLUT depending on the design in "coe" file format using MATLAB. Second stage is the generation of Verilog files using Perl scripts. Depending on the design second stage may include additional Perl scripts for the generation of the new or changed modules. Flow diagram is shown in Figure 4.1.

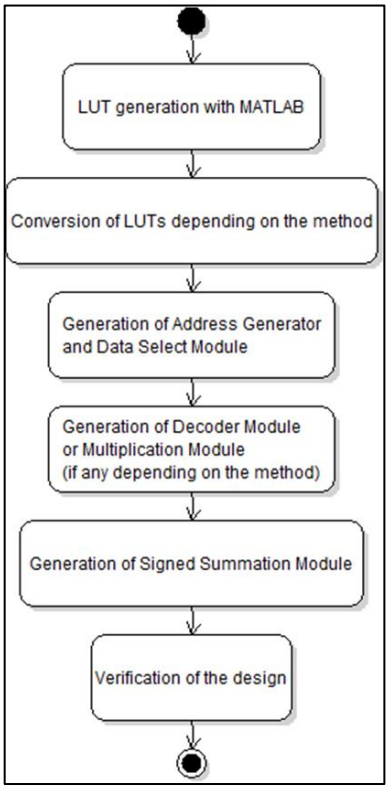

Figure 4.1. ConvLUT reduction flow diagram

4.1.1. LUT Generation with MATLAB

For FR-dLUT, FR-dLUT-VL, FR-dLUT-ZF, FR-ddLUT, FR-ddLUT-VL, and FR-ddLUT-ZF methods LUT generation starts from the MATLAB. MATLAB's flow diagram is shown in Figure 4.2. Where sine function with range $[0,2\pi)$ is mapped into 2^{16} input points and output of the sine function is mapped to 2^{16} points. The mapped values are stored in an array to be used later. Depending on the Δ every middle entry is stored for the mLUT. For example, if Δ is 4, sine values for inputs 4, 12, 20, … are stored for mLUT. Then for methods that uses dLUT's every two-consecutive difference is stored in an array called "diff". For VL methods every entries count should be known for each LUT, and since for the last dLUT there are some skipped entries another array called "diff2" is used to store values without last LUTs skipped entries. For the ddLUT methods difference of every two-consecutive entry in "diff" array is stored in a separate array called "diff3". Where the first entry in every Δ entry is the original difference and not the difference of differences, since the method requires them. Before file generations "diff2" and "diff3" arrays values counted according given Δ so the frequencies of the dLUTs and ddLUTs can be known to be used for VL methods. Then, again according to the Δ and the method used "diff" array or the "diff3" array separately written to "coe" files, and array that holds mLUT's values also written to a "coe" file. Finally, a file is created that contains mapped sine value outputs to be used during the verification.

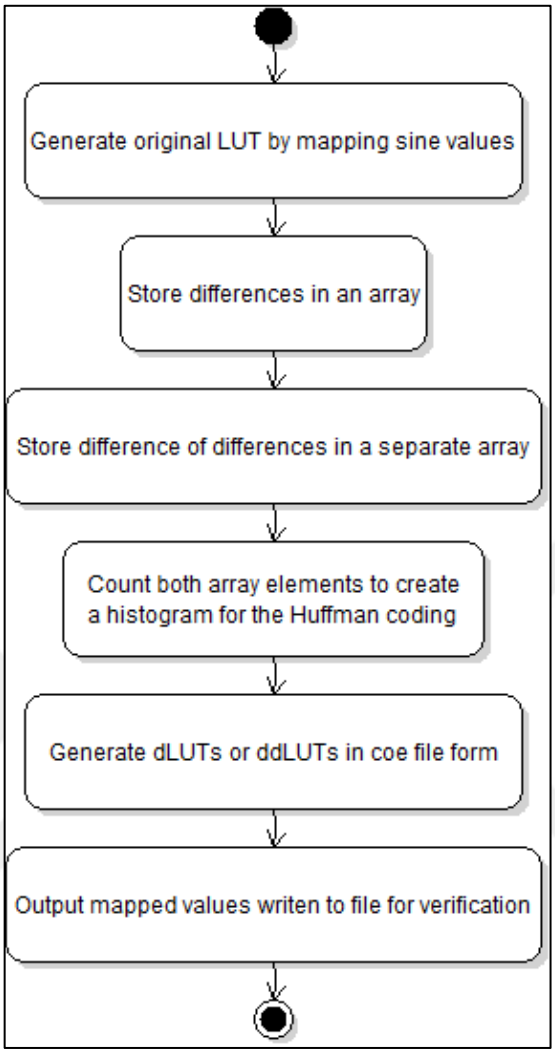

Figure 4.2. MATLAB flow diagram

4.1.2. Verilog Generation

A shell script calls multiple Perl scripts according to the method that is going to be used. For the generation of LUTs from the “coe” files generated from the MATLAB, every method calls the Perl script “LUT_Generator.perl” where the “coe” files are parsed and Verilog files generated with case statements. In case of the VL methods, before the LUT generation script is called the “cpp” program for Huffman encoding is called. Which is the encoding method used for the VL methods for this thesis.

Huffman encoding is done by taking the file that stores the frequencies of the LUTs which is generated using MATLAB. Taken values and their frequencies are used to generate a binary tree for the Huffman coding where each leaf represents a value. The values with the

highest frequency is placed in a lower depth where the value with the lowest frequency is stored in a higher depth. Starting from the root and going to a leaf, for each down left move 0 is added to the code and for each down right move 1 added to the code. When reached to a leaf, generated value is the encoded value that leaf node. Figure 4.3 shows an example for the Huffman encoding. After each value is assigned with its encoded representation a Perl script is generated to be used for the creation of the LUTs and the decoder modules for VL methods.

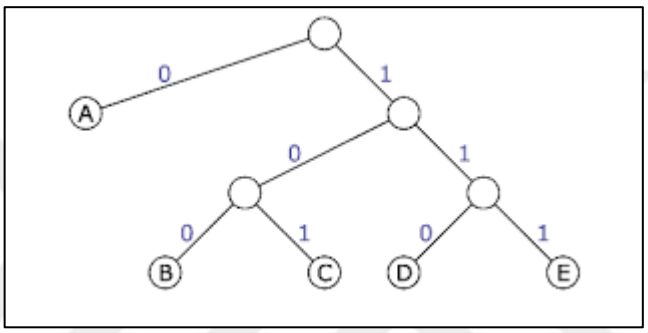

Figure 4.3. Huffman encoding tree example

After the creation of the LUTs top module, CCT, address generator module, and a wrapper module (needed for calculation of latency) is generated for all methods depending on the Δ. During the creation CCT's instance constant that is mentioned in Section 3.1 is concatenated to one of the values. For the generation of the CCT, RoCoCo's generator script is used. Output of the RoCoCo's generator script is changed a little to discard unnecessary bits (explained in Section 3.1) from the summation. For VL methods decoder generator script is called from the shell script, which uses the output Perl script generated by Huffman encoding program. At the end, for ZF methods to create changed decoders its own decoder generation module is called where the method explained in Section 3.4 is applied, and multiplication module generation script is called. Before calling the multiplication module generation script number of outputs the multiplication module should have is calculated using formula (3.1).

4.1.3. Verification

Verification of the created module is done by traversing each 2^{16} values stored in the LUT with the implemented method and comparing each of them with the output generated from MATLAB which stores the output of each possible case. Generated testbench first reads every value of MATLAB's output file and stores them in a memory array. Then, every α value starting from 0 to 2^{16} - 1 is given as input to one of the proposed methods and given as an address to created memory array. If there is mismatch between the output of the proposed method and the memory array, then error is displayed and verification stopped. If there are no errors, generated Verilog files are taken to synthesize and implementation processes. Flow diagram of the verification process is shown in Figure 4.4.

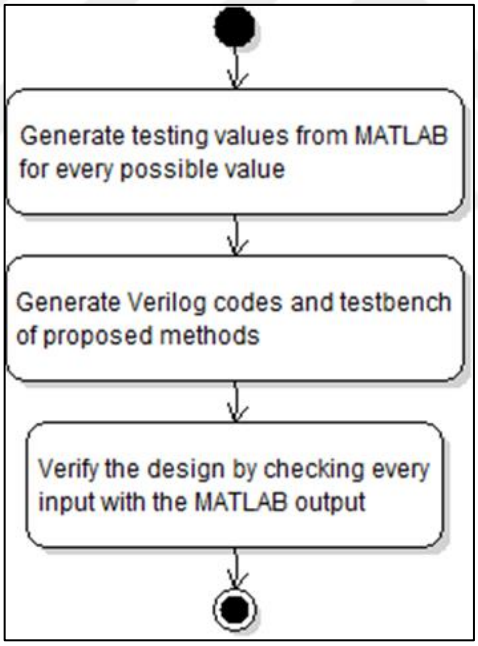

Figure 4.4. ConvLUT verification flow diagram

4.2. FOR TIV SIZE REDUCTION

For TIV size reduction methods there are three stages for the generation of the RTL codes. First stage is using the tool for generation of multipartite method proposed in [9] which generated VHDL code. Second stage is converting the VHDL code generated from the first

stage to a Verilog code. Final stage is the implementation of the proposed methods to the TIV's using generator script. Flow diagram of the whole design is shown in Figure 4.5.

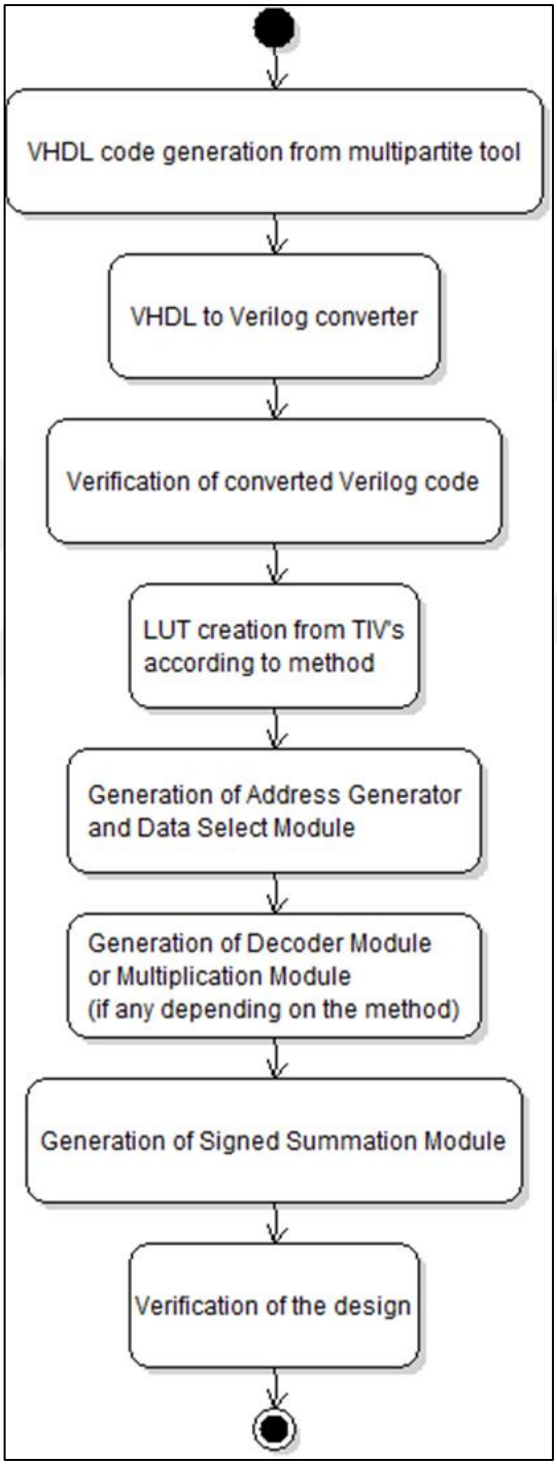

Figure 4.5. TIV reduction flow diagram

4.2.1. VHDL Code Generation of Multipartite Method

Multipartite method proposed in [9], has a java file to generate required tables for some functions. Outline of the program and possible function that can be generated from the program is shown in Figure 4.6. “wI” number represents the intended number of bits in the input (input precision of the given function), “wO” number represents the intended number of bits in the output (output precision of the given function), and “m” number represents the number of TO’s. For this thesis, as function sin (x) on [0, π/4) and 2^x-1 on [0, 1) is selected. Both, wI = 16, wO = 16 and wI = 24, wO = 24 pairs for sine function and wI = 16, wO = 15 and wI = 24, wO = 23 pairs for 2^x-1function. For 16-bit input precisions m values 1, 2, and 3 are selected where, for 24-bit input precisions m values 1, 2, 3, and 4 are selected.

After pressing the “START” button shown in Figure 4.6, window shown in Figure 4.7 appears. This window shows Alpha, Beta, Gamma i, Beta i, number of guard bits, and the error calculation results, which is explained in [9] in detail. The “VHDL” button in the bottom generates the VHDL code of the given setup.

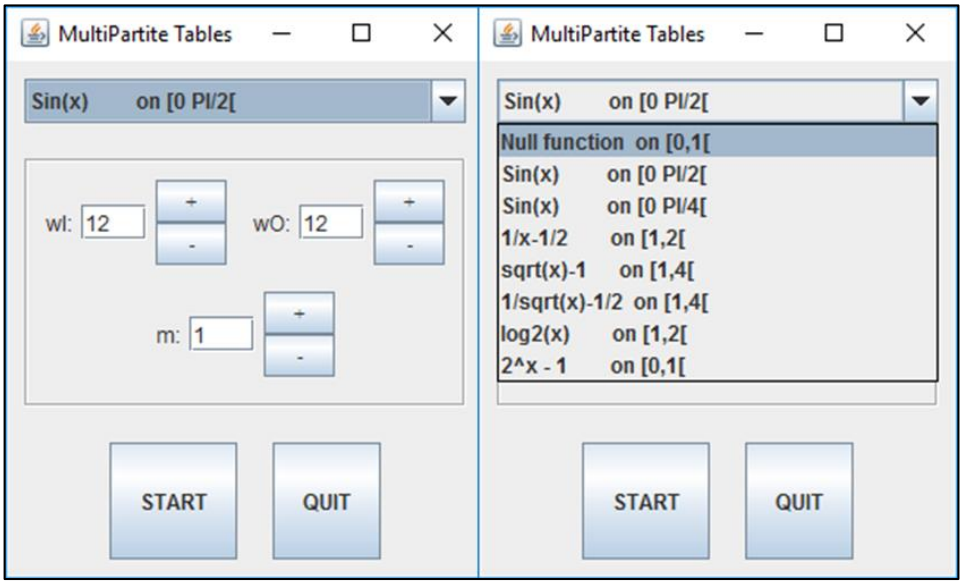

Figure 4.6. Tool for generating multipartite tables [15]

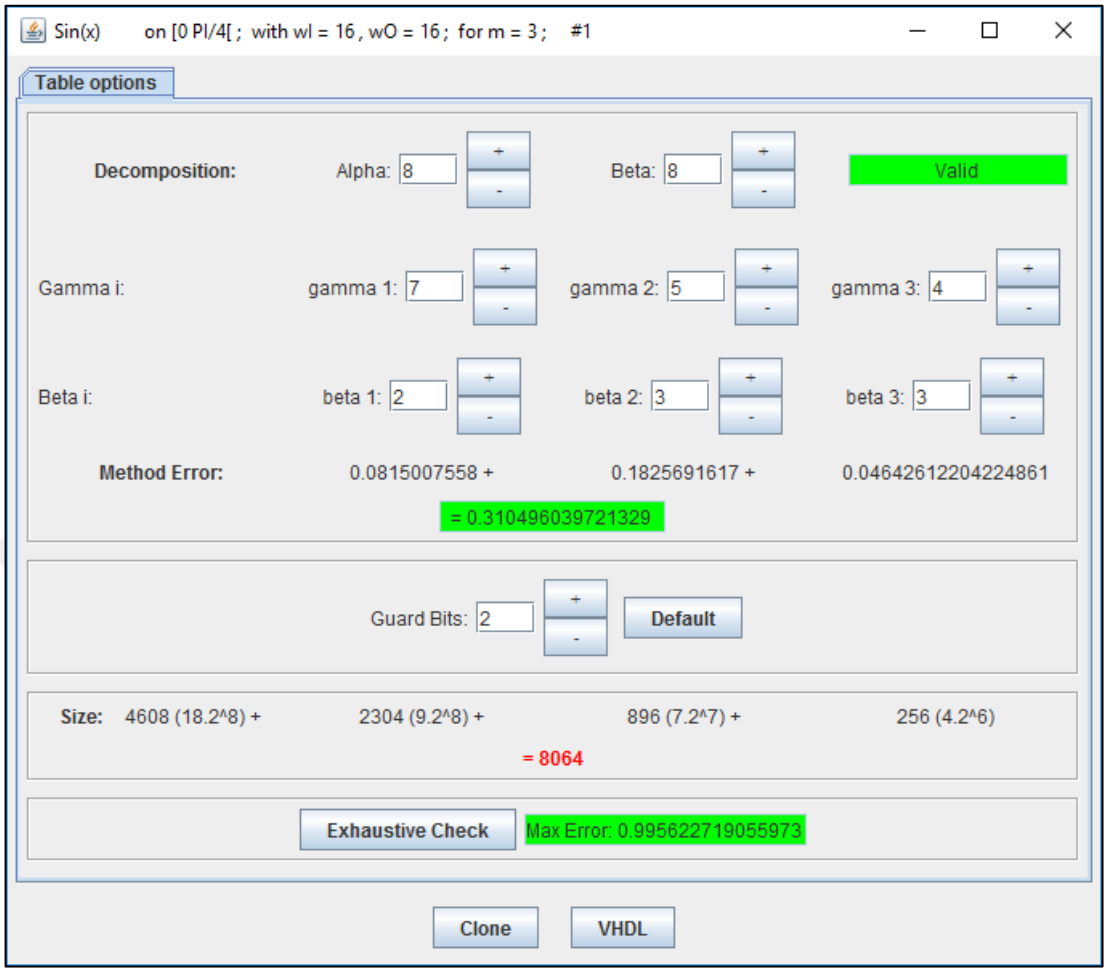

Figure 4.7. Example setup of the multipartite methods tool [15]

4.2.2. VHDL to Verilog Converter

Using a Perl script generated VHDL codes are translated to Verilog. For each module, the first thing is gathering bitwidth of the inputs, outputs, and the inner signals. Then, for the TIV and TO modules tables created with the “when” statements are read and change to “case” statements in Verilog. For the XOR modules where addresses of TIV and TOs are generated, Verilog representation of the module is written using the gathered bitwidth information (no real transformation, modules are recreated). Then, for the transformation of the top module, all gathered bitwidth information again used to create Verilog equivalent. Finally, in top module, input for XOR modules and the instantiation of modules transformed from VHDL representation to Verilog equivalent.

4.2.3. Implementation of Proposed Method for TIV

Initially, from the top module, design's input, output, and guard bitwidths gathered. Then, from the TIV module, TIV's input and output bitwidth is parsed and the TIV values are stored in an array. Then using the gathered values, difference values are produced, for mLUT and dLUTs. According to the difference values produced for dLUTs output bitwidth is also calculated. For each Δ value of 2, 4, 8, and 16, mLUT and dLUTs are created. While dLUTs are created, for each dLUT, count of each value in the dLUT is recorded to create histogram of the values to use in VL method. After mLUT and dLUTs created, address generator module is created using the gathered bitwidth information. Then, top module of the design is generated, which will replace the TIV module in the original top module. During the generation of new TIV top module instance of the CCT module and the data selection module are added. Again, while the instance of CCT is created constant mentioned in Section 3.1 is concatenated to one of the values. Additionally, as mentioned before to decrease the total number of additions every TOs output are given to the CCT. For the creation of CCT module again scripts from RoCoCo project used, and edited to discard unnecessary bits generated from CCT and discard the guard bits, since TO's addition is also done in this module. Since TO's output is send to the new TIV module, top module is also updated to direct these outputs to new TIV module. Finally, a wrapper is created to measure the latency.

For VL method Huffman codes are generated for each value using the histograms generated during the dLUT creation. According to the Huffman encoding, decoder modules are generated. Since for VL there are minor changes, like creating instance of decoders, top module of the TIV is updated. For the case of PVL method everything is same but instead of generating Huffman codes from the whole values in dLUTs, Huffman codes for every possible MSB selection combination is generated and the one with the least bit count is selected to be implemented.

4.2.4. TIV Verification

The verification of TIVs has two parts where the verification of the VHDL to Verilog transformation is done, and the verification of the newly created method. For each verification, first the VHDL codes generated from [15] is used to generate output values for

every input in input bit range. Then these values are first tested on the VHDL to Verilog transformed files. If the outputs received from the VHDL code matches the outputs of the transformed files, the Verilog files are used to create the proposed methods. Later, the generated methods are tested with the same values that are received from VHDL files. If the output of the generated new methods design matches the output of the VHDL files, new methods is used for synthesis and implementation. Figure 4.8 shows verifications flow diagram.

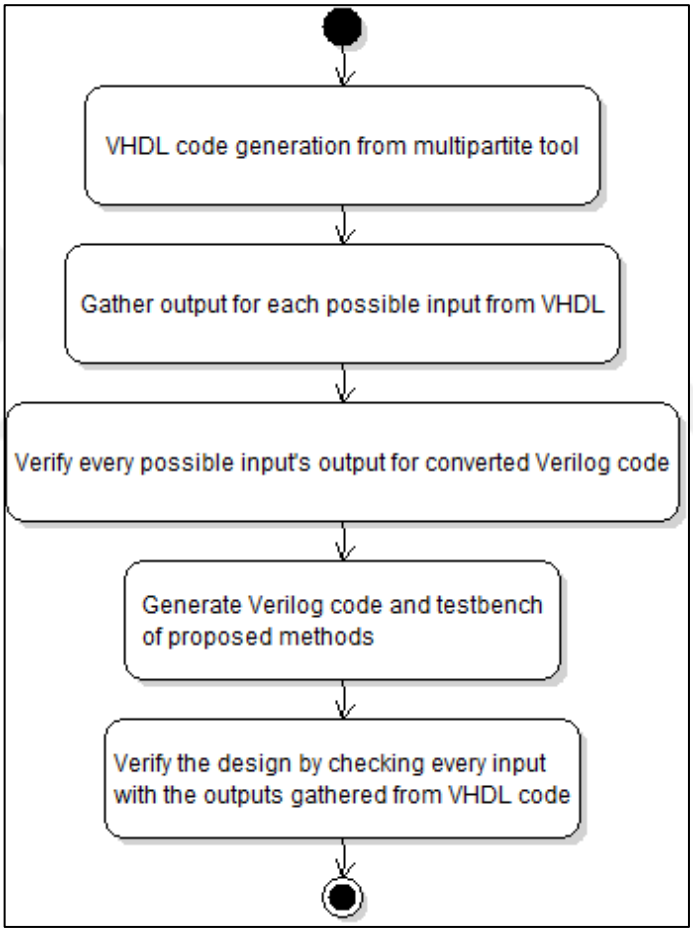

Figure 4.8. TIV reduction verification flow diagram

5. RESULTS

To evaluate our proposed methods several different FPGA devices are used for implementation. For a regular function, LUT size reduction methods (FR-dLUT, FR-dLUT-VL, FR-dLUT-ZF, FR-ddLUT, FR-ddLUT-VL, and FR-ddLUT-ZF), are synthesized using Xilinx Vivado, on the board Xilinx Virtex-7 (more specifically XC7V2000T-FLG1925). Where for TIV size reduction methods Xilinx ISE tool is used. For functions with 16-bit precision, syntheses are done with the board Xilinx Spartan-6 (more specifically XC6SLX45-3CSG324), but the functions with 24-bit precisions are synthesized using the board Xilinx Artix-7 (more specifically XC7A100T-3CSG324) since designs do not fit the previous board. All cases are synthesized for four Δ values (2, 4, 8, and 16). In addition, all used functions are synthesized with the ConvLUT method (without any alteration to LUT), which is shown as $\Delta = 0$. Note that the LUTs (mLUT and dLUTs) are logic-synthesized instead of instantiating memory blocks. This yield designs with smaller area for all methods including ConvLUT.

For the implementation of the methods in the Vivado, synthesis options Flow_AreaOptimized_High and Flow_PerfOptimized_High are selected for high area optimization and high performance optimization, respectively. Since Vivado does not provide latency of the implemented design a timing constraint is required as an input. For every implementation timing constraint is changed multiple times. After every change, implementation is repeated until the timing constraint is met with the minimum possible point. On the other hand, for ISE only area optimization with high effort is selected, and since ISE implementation provides the latency of the design only one implementation is enough.

5.1. CONVENTIONAL LUT SIZE REDUCTION

Results presented in Table 5.1 are for sine function with 16-bit input and 16-bit output precision. There are area and timing results for FR-dLUT, FR-dLUT-VL, and FR-dLUT-ZF methods. With Flow_AreaOptimized_High and Flow_PerfOptimized_High Vivado synthesis options, both area and timing optimization strategies are applied during synthesis. Since for ConvLUT (where $\Delta = 0$) only one result is generated it is been placed under FR-

dLUT column. Between the proposed methods, FR-dLUT is better than others except for the case Δ is 4, where FR-dLUT-VLs area is the best. As compared to ConvLUT case, best area of proposed methods is better by 58%, 61%, 68%, and 69%, but corresponding latency is worse by 0%, 30%, 23%, and 30%, respectively for Δ values 2, 4, 8, and 16 with Flow_AreaOptimized_High setting. With Flow_PerfOptimized_High setting, best latency of proposed methods is 60%, 67%, and 70% better in area and corresponding latencies are 9%, 19%, and 21% worse with respect to ConvLUT for Δ values, 4, 8, and 16. When Δ is 2 area is 60% better and timing is 2% better with respect to ConvLUT, in case of performance optimization.

Table 5.1. Area and timing comparison for difference LUT methods for 16-bit sine

	FR-dLUT				**FR-dLUT-VL**				**FR-dLUT-ZF**			
	Area-optimized		**Performance-optimized**		**Area-optimized**		**Performance-optimized**		**Area-optimized**		**Performance-optimized**	
Δ	**# of LUTs**	**Latency (ns)**	**# of LUTs**	**Latency (ns)**	**# of LUTs**	**Latency (ns)**	**# of LUTs**	**Latency (ns)**	**# of LUTs**	**Latency (ns)**	**# of LUTs**	**Latency (ns)**
0	6949	4.4	7524	4.3								
2	2921	4.4	3044	4.2	3210	5	3390	4.6	3193	6.1	3378	5.5
4	2731	5	3039	4.7	2712	5.7	2927	5.2	3703	10.5	3893	9.4
8	2235	5.4	2481	5.1	2330	6.2	2599	5.7	8359	21.6	8754	18.2
16	2182	5.7	2257	5.2	2593	6.4	2939	5.8	12315	22.1	13389	19.5

Results presented in Table 5.2 are again for sine function with 16-bit input and 16-bit output precision. Implementation results of FR-ddLUT, FR-ddLUT-VL, and FR-ddLUT-ZF methods are presented for both area and timing. During synthesis Flow_AreaOptimized_High and Flow_PerfOptimized_High Vivado strategies applied again. Case of Δ = 0 is not included in the Table 5.2 since it is the same as in Table 5.1. When compared between each other FR-ddLUT-VL is the best for area and FR-ddLUT is the best for timing for all Δ values. Compared to ConvLUT, FR-ddLUT-VL's area is better by 56%, 57%, 62%, and 63%, where its latency is worse by 18%, 36%, 55%, and 73%, for Δ values 2, 4, 8, and 16 respectively (when synthesis setting is Flow_AreaOptimized_High). FR-ddLUT has best latency and when compared with the ConvLUT it is 56%, 58%, 63%, and 62% better in area while worse in latency by 2%, 19%, 37%, and 61% for Δ values, 2, 4, 8, and 16 (when synthesis setting is Flow_PerfOptimized_High). For most of the case FR-dLUT method is better than the VL and ZF methods due to logic used in these methods occupies larger area in the design then the reduction done on the bit sizes of the LUTs.

Table 5.2. Area and timing comparison for difference of differences LUT methods for 16-bit sine function

	FR-ddLUT				FR-ddLUT-VL				FR-ddLUT-ZF			
	Area-optimized		Performance-optimized		Area-optimized		Performance-optimized		Area-optimized		Performance-optimized	
Δ	# of LUTs	Latency (ns)	# of LUTs	Latency (ns)	# of LUTs	Latency (ns)	# of LUTs	Latency (ns)	# of LUTs	Latency (ns)	# of LUTs	Latency (ns)
2	3118	5.3	3309	4.4	3035	5.2	3200	4.9	3099	6.3	3284	6
4	3125	5.4	3146	5.1	3022	6	3106	5.6	3137	6.3	3269	6.2
8	2634	6.3	2748	5.9	2609	6.8	2652	6	2679	7.1	2822	6.7
16	2650	7.5	2864	6.9	2538	7.6	2837	7	2594	7.8	2772	7.1

When best results of dLUT methods and the best results of ddLUT methods are compared in terms of area and timing with both Flow_AreaOptimized_High and Flow_PerfOptimized_High Vivado strategies, it is seen that dLUT methods are better than ddLUT. Since in area with Flow_AreaOptimized_High option, dLUT is better by 4%, 10%, 14%, and 14% with respect to ddLUT, where also the timing is better by 15%, 5%, 21%, and 25% for Δ values 2, 4, 8, and 16. Also in timing with Flow_PerfOptimized_High option dLUT is better by 8%, 3%, 10%, and 21% than ddLUT, and with 5%, 8%, 14%, and 25% better in terms of area respectively for Δ values 2, 4, 8, and 16.

5.2. TIV SIZE REDUCTION

For TIV reduction of the multipartite method, Table 5.3 presents results for sine function implementation results with 16-bit input and 16-bit output precision and 2^x function implementation results with 16-input and 15-bit output precision. For both functions three m values 1, 2, and 3 are used. For each m value five Δ values 0, 2, 4, 8, and 16 are implemented where, Δ = 0 is the case with no alterations. There are area and timing results for FR-dLUT, FR-dLUT-VL, and FR-dLUT-PVL methods. During synthesis, high area optimization is selected from ISE strategies. For each m value, Δ = 0 case is reported in the FR-dLUT column. As compared to original case, only for m = 1 case FR-dLUT is better for both functions. For other m values, none of the proposed methods beats the area of the original case for all Δ values 2, 4, 8, and 16. Shaded cells are the best area or timing between the three methods implemented for each m value. Cells with "-" in FR-dLUT-PVL column

represents that best case of encoding is done when entries are not split which is the case of FR-dLUT-VL.

Bar graphs in Figure 5.1 and Figure 5.2 show the results of Δ = 0 case, best result of the FR-dLUT, FR-dLUT-VL, and FR-dLUT-PVL. Best results of FR-dLUT, FR-dLUT-VL, and FR-dLUT-PVL are compared with the Δ = 0 case, improvements in areas are 27%, -12%, and -24% for the sine function and 25%, -12%, and -24% for the 2^x function, respectively.

Table 5.3. TIV size reduction results for 16-bit input precision sine and 2^x functions

Function	m	Δ	FR-dLUT		FR-dLUT-VL		FR-dLUT-PVL	
			Area	Time (ns)	Area	Time (ns)	Area	Time (ns)
Sine wi:16 wo:16	1	0	346	9.20				
		2	254	10.72	308	12.61	282	11.48
		4	248	11.35	398	14.24	270	12.29
		8	310	12.56	553	15.30	339	14.46
		16	435	13.90	486	15.77	463	15.61
	2	0	188	9.97				
		2	210	10.35	231	11.15	219	10.87
		4	245	12.45	247	11.67	245	11.23
		8	320	12.84	317	13.34	320	12.84
		16	449	12.60	448	12.83	-	-
	3	0	147	9.74				
		2	182	10.95	202	12.48	188	10.70
		4	218	11.16	219	11.49	221	11.52
		8	278	12.83	281	12.94	279	13.22
		16	404	13.91	406	13.49	-	-
2^x wi:16 wo:15	1	0	341	8.95				
		2	255	10.24	355	12.57	264	10.86
		4	265	11.53	448	13.74	285	11.89
		8	323	12.59	492	14.43	320	12.64
		16	438	13.75	451	13.86	438	13.95
	2	0	196	9.86				
		2	219	10.38	239	11.57	216	10.67
		4	249	11.43	253	11.83	243	12.40
		8	324	13.03	320	12.92	-	-
		16	446	13.75	442	16.13	-	-
	3	0	156	9.62				
		2	193	11.08	209	11.62	189	11.31
		4	231	11.36	233	10.93	227	11.94
		8	287	13.35	288	12.36	-	-
		16	422	13.84	416	13.61	-	-

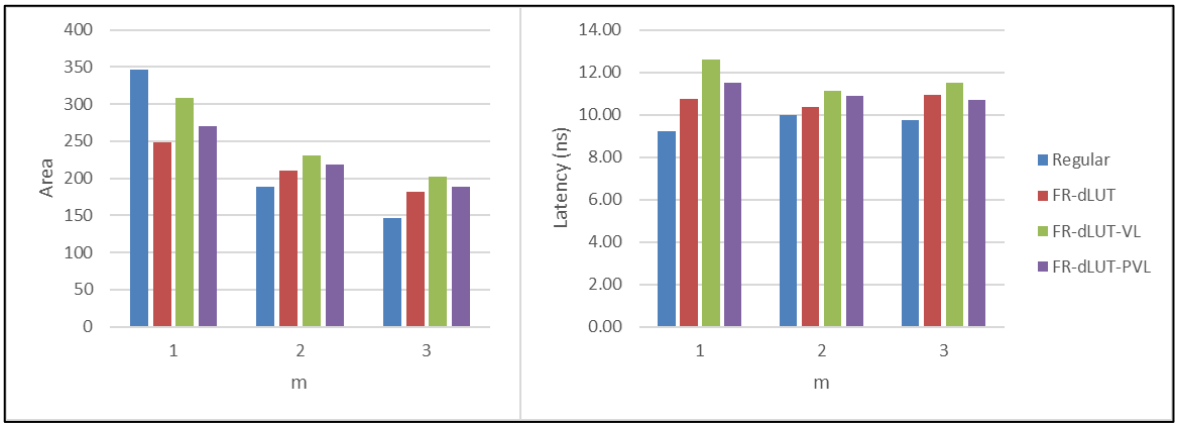

Figure 5.1. Bar graph of best 16-bit sine function results for TIV reduction

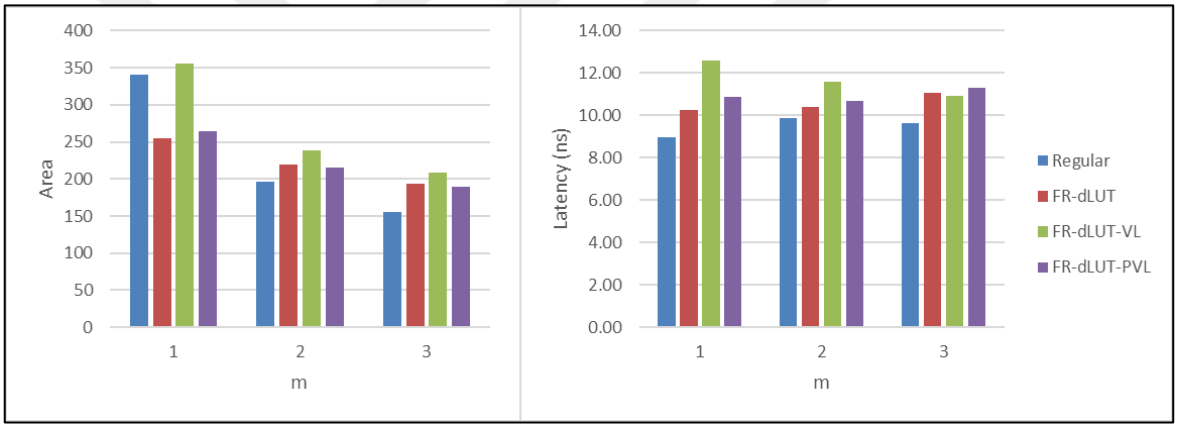

Figure 5.2. Bar graph of best 16-bit 2^x function results for TIV reduction

During the testing of methods in higher precision functions, FR-dLUT method is implemented on the TIVs of multipartite tables of sine with 24-bit input and 24-bit output precision, and 2^x function with 24-bit input and 23-bit output precision. Table 5.4 shows the implementation results of sine function and Table 5.5 shows the implementation results for 2^x function. Again $\Delta = 0$ case shown in the tables is the result of the original multipartite methods implementation. Additionally, there is a $\Delta = 1$ row for each m value, which represents the implementation results of 3T-TIV method in [16]. All syntheses are done with ISE's high area and high effort optimization selected during the synthesis stage. Last column of the table shows the area saving percent of each method with respect to original implementation. Shaded cells represent the best area and the best timing for each m values. Due to large decrease in the area best timing is also achieved in the method implemented.

Figure 5.3 shows the bar graph of the Table 5.4 where Figure 5.4 shows the bar graph of the Table 5.5.

Table 5.4. TIV size reduction results with FR-dLUT method for 24-bit sine function

m	Δ	Area (# of LUTs)	Time (ns)	Area Saving %
1	0	54045	Can't Route	0
	1	30472	16.49	43.62
	2	30907	14.77	42.81
	4	29402	15.62	45.60
	8	28755	16.39	46.79
	16	28461	17.51	47.34
2	0	4922	12.44	0
	1	3817	12.18	22.45
	2	3468	10.70	29.54
	4	3291	10.45	33.14
	8	3172	11.01	35.55
	16	3575	12.27	27.37
3	0	3015	11.71	0
	1	2572	11.59	14.69
	2	2468	10.01	18.14
	4	2379	9.80	21.09
	8	2525	11.16	16.25
	16	2797	11.21	7.23
4	0	2583	12.49	0
	1	2156	10.87	16.53
	2	2083	10.11	19.36
	4	1934	11.29	25.13
	8	2073	10.89	19.74
	16	2345	11.51	9.21

Figure 5.3. TIV size reduction for 24-bit precision sine function result bar graphs

Table 5.5. TIV size reduction results with FR-dLUT method for 24-bit 2^x function

m	Δ	Area (# of LUTs)	Time (ns)	Area Saving %
1	0	57511	Can't Route	0
	1	48156	17.92	16.27
	2	45849	16.14	20.28
	4	44484	16.95	22.65
	8	43910	17.43	23.65
	16	43765	18.45	23.90
2	0	5845	10.86	0
	1	4970	12.15	14.97
	2	4594	11.13	21.40
	4	4283	10.98	26.72
	8	4234	12.42	27.56
	16	4545	12.37	22.24
3	0	3512	11.06	0
	1	3235	12.21	7.89
	2	3071	10.38	12.56
	4	3004	10.71	14.46
	8	3117	11.35	11.25
	16	3267	11.59	6.98
4	0	2844	12.15	0
	1	2515	11.42	11.57
	2	2390	9.72	15.96
	4	2325	10.27	18.25
	8	2441	11.58	14.17
	16	2345	11.51	17.55

As seen from the results of both 24-bit's and 16-bit's, sine and 2^x functions, for higher bitwidths FR-dLUT method has much more impact on the area and timing. For 16-bit results there is only improvement when m = 1, however for 24-bit results there is more improvement in m = 1 case then 16-bit results and there are improvements for every m values 1, 2, 3, and 4. Additionally, in 24-bit results timing is also improved where there is no such case in 16-bit. Also, improvements in 24-bit results are better than the 3T-TIV methods improvement for both functions.

Figure 5.4. TIV size reduction for 24-bit precision 2^x functions result bar graphs

6. CONCLUSION AND FUTURE WORK

In this thesis, multiple novel microarchitectures are presented, called FR-dLUT, FR-dLUT-VL, FR-dLUT-ZF, FR-ddLUT, FR-ddLUT-VL, FR-ddLUT-ZF, and FR-dLUT-PVL. Some implemented on top the ConvLUT's and some implanted on the TIV's of multipartite tables of functions. It is seen that these methods can replace a ConvLUT if the LUT corresponds to a continuous function, i.e., a smooth function that has no sudden jumps. On the other hand, these methods can replace the TIV's of multipartite methods when high precision input or output is required.

When methods applied on a ConvLUT; the results show that methods implemented with ddLUT's are worse than the methods implemented with dLUT's. Due to the increased logic in the circuits of ddLUT, decrease in the bit size of the tables are not observed on the overall design. In the case of a sine function with 16-bit input and 16-bit output, for the best-case scenario in terms of area FR-dLUT with $\Delta = 16$, reduces area by 69% with 28% penalty in the latency compared to ConvLUT. Where for the lowest latency penalty, FR-dLUT with $\Delta = 2$, can reduce the area by 52% without any penalty in latency.

When the methods proposed in this thesis applied on the most state of the art methods for LUT size reduction, which is using multipartite table method; results show that with smaller bit precision there is not much improvement but for high bit precisions there is significant improvements. For 16-bit input resolutions, results show that only for the case where m is 1 there is an improvement in the area with some loss in latency. Which is viable for the cases where higher m values cannot be used (cannot tolerate the extra errors). For 24-bit input resolutions, for all m values 1, 2, 3, and 4 proposed method improves area significantly. Also, due to the improvement in the area latency of the overall design decreases despite the extra logic used for the implementation. Finally, method used in 24-bit precision is compared with the latest TIV size reduction method proposed in [16] and the improvement of the method proposed in this thesis is better than the method proposed in [16].

As a future work, instead of testing two functions with the multipartite method, more functions can be added to the test. Also, some methods used for ConvLUT is not implemented for the TIV's they can be tested with high precision TIV's where they can yield better results. Finally, instead of making everything fully random access, methods proposed

can be implemented for a semi-random access. Since most real-life applications requires sequential access to functions, like inverted pendulum implementation, where there are no sudden leaps between requested trigonometric function angles. Also, semi random access method will be implemented on the TIV's of the multipartite method, due to the existence of TOs, the design will provide some semi random access to function values.

REFERENCES

1. P. Suganth, N. Jayakumar, and S. P. Khatri. A Fast Hardware Approach for Approximate, Efficient Logarithm and Antilogarithm Computations. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17.2:269-277, 2009.

2. L. J. Y. Lih, and C. C. Jong. A Memory-Efficient Tables-and-Additions Method for Accurate Computation of Elementary Functions. *IEEE Transactions on Computers*, 62.5: 858-872, 2013.

3. P. D. Hyun, H.S. Ko, J. G. Kim, and J. D. Cho. Real Time Tectification Using Differentially Encoded Lookup Table. *Proceedings of the 5th International Conference on Ubiquitous Information Management and Communication (ICUIMC)*, 47, 2011.

4. H. Unlu, M. A. Ozkan, H. F. Ugurdag, and E. Adali. Area-Efficient Look-Up Tables for Semi-Randomly Accessible Functions, *Proceedings of WSEAS Recent Advances in Electrical Engineering*, 171-174 2014.

5. D. D. Sarma, and D. W. Matula. Faithful Bipartite ROM Reciprocal Tables. *Computer Arithmetic, 1995, Proceedings of the 12th Symposium on*. IEEE, 1995.

6. H. Hassler, and N. Takagi. Function Evaluation by Table Look-up and Addition. *Computer Arithmetic, 1995, Proceedings of the 12th Symposium on*. IEEE, 1995.

7. M. J. Schulte, and J. E. Stine. Approximating Elementary Functions with Symmetric Bipartite Tables. *IEEE Transactions on Computers*, 48.8: 842-847, 1999.

8. J. E. Stine, and M. J. Schulte. The Symmetric Table Addition Method for Accurate Function Approximation. *The Journal of VLSI Signal Processing*, 21.2:167-177, 1999.

9. F. D. Dinechin, and A. Tisserand. Multipartite Table Methods. *IEEE Transactions on Computers*, 54.3:319-330, 2005.

10. J. Detrey, and F. D. Dinechin. *Multipartite Tables in JBits for the Evaluation of Functions on FPGA*. Diss. INRIA, 2001.

11. H. F. Ugurdag, O. Keskin, C. Tunc, F. Temizkan, G. Fici, and S. Dedeoglu. RoCoCo: Row and Column Compression for High-Performance Multiplication on FPGAs. *East-West Design and Test Symposium (EWDTS)*, 98-101, 2011.

12. L. Dadda. Some Schemes for Parallel Multipliers. *Alta Frequenza*, 34:349–356, 1965.

13. C. Wallace. A Suggestion for a Fast Multiplier. *IEEE Transactions on Electronic Computers*, 1:14-17, 1964.

15. F. D. Dinechin, and A. Tisserand. The Multipartite Method for Function Evaluation, http://www.ens-lyon.fr/LIP/Arenaire/Ware/Multipartite/ [retrieved 30 May 2017].

16. S. F. Hsio, P. H. Wu, C. S. Wen, and P. K. Meher. Table Size Reduction Methods for Faithfully Rounded Lookup-Table-Based Multiplierless Function Evaluation. *IEEE Transactions on Circuits and Systems II: Express Briefs,* 62.5:466-470, 2015.

17. D. Brooks, and M. Martonosi. Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance. *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium on.* IEEE, 13-22, 1999.

18. G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches. *In Proceedings of the 21st international conference on Parallel architectures and compilation techniques,* ACM, 377-388, 2012.