José Daniel Soares Caetano

Computação de Funções Elementares em FPGA

Graph-Based FPGA Computation of Elementary Functions

**José Daniel Soares
Caetano**

Graph-Based FPGA Computation of Elementary Functions

Dissertation presented to University of Aveiro to fulfil the requirements to obtain the grade of Master Degree in Electronics and Telecommunications Engineering, under scientific supervision by Dr. Iouliia Skliarova from the Department of Electronics, Telecommunications and Informatics of University of Aveiro and Prof. Jaakko Astola and Prof. Radomir Stankovic from Tampere University of Technology.

The jury

President

Prof. Dr. Arnaldo da Silva Oliveira
Assistant Professor from University of Aveiro

Prof. Dr. António José Duarte Araújo
Assistant Professor from Faculty of Engineering of the University of Porto

Prof. Dr. Iouliia Skliarova
Assistant Professor from University of Aveiro

Acknowledgements

To my family who always supported me, to my friends that helped me through rough times and to Sara.

Abstract   Since C.Y.Lee first proposed the idea of representing switching circuits as
                                decision diagrams, there has been some interest in developing these diagrams
                                in order to make them more compact and effective. One of the main ap-
                                plications of this technique is to represent circuits that perform elementary
                                functions, such as cosine, sine, square root, etc. In this thesis, we try to
                                prove that by choosing the right polarity for an Arithmetic Decision Dia-
                                gram we can compactly and effectively represent a switching function and
                                implement it in hardware. This thesis proposes algorithms that can com-
                                pactly implement a given elementary function in hardware by finding the
                                best possible polarity for the respective Arithmetic Decision Diagram.

Palavras-chave   Field-Programmable Gate Array, Transformação Espectral, Diagramas de Decisão Aritméticos, Transformada Aritmética de Polaridade Fixa

Resumo   Desde que C.Y.Lee propôs a ideia de representar funções de comutação sob a forma de diagramas de decisão, tem havido algum interesse em desenvolver estes diagramas de modo a torná-los mais compactos e eficientes. Uma das principais aplicações desta técnica é representar circuitos que realizem funções elementares, como é o caso do seno, coseno, raíz quadrada, etc. Nesta tese tentamos provar que escolhendo a polaridade certa para um Diagrama de Decisão Aritmético é possível representar compacta e eficazmente uma função de comutação e implementá-la em hardware. Esta tese propõe algoritmos que conseguem implementar compactamente uma dada função elementar em hardware encontrando a melhor polaridade possível para o respetivo Diagrama de Decisão Aritmético.

Contents

List of Tables

List of Figures

Part I

Introduction

Chapter 1

Introduction

1.1 Setting

The use of binary decision diagrams to represent switching circuits is as old as the 1950's, when C.Y. Lee[1] first proposed the idea based on the Shannon expansion. Over the following decades many advances in this technology have been achieved, namely by S.B. Akers[2] and Randal E. Bryant, whose paper Graph-Based Algorithms for Boolean Function Manipulation[3] is cited on most literature about the subject. Following the BDD's success in representing Boolean functions, many attempts have been made to extend its range to integer or real valued functions[4]. Over the most recent years the group lead by Prof. Tsutomu Sasao has been trying to realize elementary functions using data structures that have evolved from the traditional BDDs, like Multi-Valued Decision Diagrams[5], Edge-Valued Decision Diagrams[6], spectral transforms[7], and combinations of these structures.

1.2 Motivation

Numeric functions are widely used in computer graphics, digital signal processing, communication systems, robotics, etc. [8]. In these applications, Numerical Function Generators (NFGs) are often required as a way to enhance processing times. There has been an increasing interest in using arithmetic decision diagrams to create compact NFGs, and some studies have been made on the subject [9] [7]. As far as we know, no study has been published that presents algorithms to build NFGs using general purpose software, such as MATLAB, instead of specialised software for decision diagrams. The use of FPGA technology for testing is very practical, since its reconfigurability allows us to quickly change the function implemented in hardware, and thus allowing to quickly test many implementations of NFGs.

1.3 Goals

The task at hand is to realize a Numerical Function Generator that will take as input any real function f(x) that is bounded over a domain [a,b] as well as the precision and accuracy desired for the system. For this purpose the function will be segmented and linearised and an Arithmetic Transform Decision Diagram, a type of spectral transform

decision diagram, will be used to map each value of x, the function argument, to the corresponding segment. The NFG will be implemented in FPGA using VHDL language.

1.4   Structure of the Thesis

This thesis is divided in 4 parts. Besides this introduction, we have:

- **Part II - Function Representation** - In the first chapter of this part we will briefly explain the mathematical concepts that support decision diagrams, and we will derive the expressions of some of the relevant expansion rules in this project's context. The second chapter shows several types of decision trees, since their earlier concept up to the arithmetic decision tree, and the rules to reduce them into decision diagrams, along with some examples justifying their reduction.

- **Part III - Implementation** - In this part we present algorithms that implement a Numerical Function Generator in FPGA, using Arithmetic Decision Diagrams. We show the steps that go from the input in MATLAB up to the testing in FPGA, and see the details about some of the most important algorithms. The results are discussed, as well as the limitations of this implementation.

- **Part IV - Conclusions** - This part closes this dissertation. A summary of the project and the main conclusions are presented, as well as some possible improvements that could yield better results.

Part II

Function Representation

Chapter 2

Functional Expressions

Combinational electronic circuits can be conveniently expressed as switching functions. These functions, on their turn, can be represented in many ways using truth-tables, Karnaugh maps, Boolean algebra, among other techniques. From a truth-table representation of a function it is possible to write it as a Sum-Of-Products (SOP) by *disjuncting* the minterms that cause the function to yield the value "1"[10]. This disjunction of minterms is called the disjunctive normal form. Figure 2.1 shows an example of a 2-variable XOR function represented by a truth-table and its corresponding Sum-Of-Products expression.

x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	0

$$f = \overline{x_1}x_2 + x_1\overline{x_2}$$

Figure 2.1: Truth-table for the 2-variable XOR function and its corresponding SOP expression.

2.1 Shannon Expansion Rule

Shannon has shown that a switching function can be represented as

$$f = \overline{x_i}f_0 \oplus x_i f_1$$

with

$$f_0 = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$$

and

$$f_1 = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n).$$

Such representation is called the *Shannon expansion*[11]. The complete SOP for the function can be achieved by recursively applying the Shannon expansion to all its variables.

EXAMPLE 2.1 *Take a function $f(x_1, x_2)$. We first apply the Shannon expansion to x_1 and then to x_2. Note that the final result is independent of the order in which the expansion is applied.*

$$
\begin{aligned}
f(x_1, x_2) &= \overline{x_1}f(0, x_2) \oplus x_1 f(1, x_2) \\
&= \overline{x_1}(\overline{x_2}f(0,0) \oplus x_2 f(0,1)) \oplus x_1(\overline{x_2}f(1,0) \oplus x_2 f(1,1)) \\
&= \overline{x_1}\,\overline{x_2}f(0,0) \oplus \overline{x_1}x_2 f(0,1) \oplus x_1\overline{x_2}f(1,0) \oplus x_1 x_2 f(1,1),
\end{aligned}
$$

which is the complete SOP for f.

The SOP can also be obtained from the function's matrix notation with the help of the Kronecker product $\otimes$ defined as

$$
A \otimes B = \begin{bmatrix}
a_{1,1}B & a_{1,2}B & \cdots & a_{1,n}B \\
a_{2,1}B & a_{2,2}B & \cdots & a_{2,n}B \\
\vdots & \vdots & \ddots & \vdots \\
a_{m,1}B & a_{m,2}B & \cdots & a_{m,n}B
\end{bmatrix} (mp \ by \ nq),
$$

where A is an $m \times n$ matrix and B is a $p \times q$ matrix.

In matrix notation, defining $\mathbf{X}(1) = [\,\overline{x_i} \ x_i\,]$ for $i = 1,...,$n, $\mathbf{B}(1)$ as the identity matrix $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $\mathbf{F}$ as $\begin{bmatrix} f(0) \\ f(1) \end{bmatrix}$ we can represent the Shannon expansion as

$$
f = \left(\overset{n}{\underset{i=1}{\otimes}} X(1) \right) \left(\overset{n}{\underset{i=1}{\otimes}} B(1) \right) F.
$$

2.2   Reed-Muller expansion Rules

If we assume we are working in *modulo 2*, $\overline{x_i}$ can be replaced by the exclusive sum of x_i and the logic constant 1. Rewriting the Shannon expansion using this property we can derive

$$
\begin{aligned}
f &= \overline{x_i}f_0 \oplus x_i f_1 \\
&= (1 \oplus x_i)f_0 \oplus x_i f_1 \\
&= 1 \cdot f_0 \oplus x_i f_0 \oplus x_i f_1 \\
&= 1 \cdot f_0 \oplus x_i(f_0 \oplus f_1).
\end{aligned}
$$

With this derived expression, called the *positive Davio (pD) expansion* we can represent the function as the polynomial $f = c_0 \oplus c_1 x_i$ with coefficients $c_0 = f_0$ and $c_1 = f_0 \oplus f_1$.

We can instead choose to use the relation $x_i = 1 \oplus \overline{x_i}$ to obtain the *negative Davio*

(nD) expansion from the Shannon expansion.

$$\begin{aligned}
f &= \overline{x_i}f_0 \oplus x_i f_1 \\
&= \overline{x_i}f_0 \oplus (1 \oplus \overline{x_i})f_1 \\
&= \overline{x_i}f_0 \oplus 1 \cdot f_1 \oplus \overline{x_i}f_1 \\
&= 1 \cdot f_1 \oplus \overline{x_i}f_0 \oplus \overline{x_i}f_1 \\
&= 1 \cdot f_1 \oplus \overline{x_i}(f_0 \oplus f_1) \\
&= c_0 \oplus c_1\overline{x_i}.
\end{aligned}$$

Once again we can recursively apply the Reed-Muller expansion rules to all variables in a function. Using the transform matrices $\mathbf{R}(1) = \left[\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right]$ and $\overline{\mathbf{R}}(1) = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right]$ for the positive and negative expansions respectively we get

$$f = \left(\overset{n}{\underset{i=1}{\otimes}} X(1)\right)\left(\overset{n}{\underset{i=1}{\otimes}} R(1)\right) F$$

and

$$f = \left(\overset{n}{\underset{i=1}{\otimes}} X(1)\right)\left(\overset{n}{\underset{i=1}{\otimes}} \overline{R}(1)\right) F.$$

2.2.1 Fixed Polarity RM-expressions

When applying the Reed-Muller rules to all variables in a function we are not restricted to either pD or nD on all variables. We can choose which variables we prefer using one or other according to our goals. The polarity we choose to each variable is usually defined in a polarity vector $H = (h_1, \ldots, h_n)$, where each $h_i \in \{0, 1\}$ refers to the polarity of the variable x_i. If $h_i = 0$ the i-th variable is represented by the positive literal $\overline{x_i}$. Otherwise, if $h_i = 1$ the i-th variable is represented by the negative literal x_i.

2.3 Word-Level Expressions

So far we have only worked with bit-level expressions, which means expressions whose output is a single bit. The coefficients we get when applying expansion rules to this kind of expressions are always in the boolean range (0 or 1). By treating the logic constants 0 and 1 as real numbers (0 and 1 respectively) we can extend the range of these expressions to the rational domain. An important application of this concept is the representation of multi-output functions. Example 2.2 shows how the use of word-level expressions allows us to treat a multi-output function as if it had a single output.

EXAMPLE 2.2: *Consider a system of functions*

$$(f_2(x_1, x_2, x_3), f_1(x_1, x_2, x_3), f_0(x_1, x_2, x_3)),$$

where

$$\begin{aligned}
f_0(x_1, x_2, x_3) &= x_2\overline{x_3} \oplus x_1 \\
f_1(x_1, x_2, x_3) &= x_1 \vee x_2\overline{x_3} \\
f_2(x_1, x_2, x_3) &= x_2(x_1 \vee x_3)
\end{aligned}$$

We can form a matrix $\boldsymbol{F}$ whose columns are the truth-vectors of f_2, f_1 and f_0.

$$\mathbf{F} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{F}_2 & \mathbf{F}_1 & \mathbf{F}_0 \end{bmatrix}.$$

We can read each row from the matrix $\boldsymbol{F}$ as if it were an integer number written in binary representation to get

$$\begin{bmatrix} 0 \\ 3 \\ 3 \\ 6 \\ 0 \\ 1 \\ 6 \\ 7 \end{bmatrix} = 2^2 \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} + 2 \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} + 1 \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = 4\mathbf{F}_2 + 2\mathbf{F}_1 + \mathbf{F}_0.$$

This matrix can then be processed by word-level transform rules.

2.3.1   Arithmetic Transform

The *Arithmetic Transform* can be viewed as the integer version of the Reed-Muller Expansion. To derive its expression from the Shannon expansion we must first look at the relation between boolean operations and arithmetic operations in table 2.1 to get to the deduction that follows.

$$\begin{aligned} f &= \overline{x_i} f_0 \oplus x_i f_1 \\ &= (1 - x_i) f_0 \oplus x_i f_1 \\ &= (1 - x_i) f_0 + x_i f_1 - 2(1 - x_i) f_0 x_i f_1 \\ &= 1 \cdot f_0 - x_i f_0 + x_i f_1 - 2x_i f_0 f_1 + 2x_i f_0 f_1 \\ &= 1 \cdot f_0 + x_i (f_1 - f_0) \end{aligned}$$

In this case, the property $\overline{x_1} = 1 - x_1$ was used to get the *positive polarity Arithmetic Transform*. We can choose to rearrange that property to $x_1 = 1 - \overline{x_1}$ to get the *negative polarity Arithmetic Transform* instead, which gives us

$$f = 1 \cdot f_1 + \overline{x_i}(f_0 - f_1).$$

Once again the matrix representation can help us applying this rule to all variables in the function. In this case, the variable matrix will be $\mathbf{X}_A(1) = \begin{bmatrix} 1 & x_i \end{bmatrix}$ and the transform matrix $\mathbf{A}^{-1}(1) = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$ for the *positive polarity Arithmetic Transform* and $\mathbf{X}_A(1) =$

Boolean	Arithmetic
$\overline{x_1}$	$1 - x_1$
$x_1 \wedge x_2$	$x_1 x_2$
$x_1 \vee x_2$	$x_1 + x_2 - x_1 x_2$
$x_1 \oplus x_2$	$x_1 + x_2 - 2x_1 x_2$

Table 2.1: Boolean and arithmetic operations.

$\begin{bmatrix} 1 & \overline{x_i} \end{bmatrix}$ and $\mathbf{A}^{-1}(1) = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}$ for the *negative polarity Arithmetic Transform*. Again, by setting a vector $H = (h_1, \ldots, h_n)$ we can create a *fixed polarity Arithmethic Expression*. Thus, the arithmetic expression is defined as

$$f = \left(\overset{n}{\underset{i=1}{\otimes}} X_{Ah}(1) \right) \left(\overset{n}{\underset{i=1}{\otimes}} A_h^{-1}(1) \right) F.$$

$\mathbf{X}_{Ah}$ and $\mathbf{A}_h^{-1}$ mean that these matrices can take the positive or negative form depending on the value of the h_i variable for the current iteration.

In this chapter we've seen many ways to analytically represent a switching function using the properties of the Shannon expansion. The arithmetic expansion is an example of how to extend the representation to word-level, which allows a multi-output function to be represented as a single expression. The flexibility in polarity choice allows us to find more compact expressions that will translate in simpler hardware implementations of said functions.

Chapter 3

Decision Diagrams

Decision diagrams are data structures used for graphical representation of discrete functions. Before discussing decision diagrams, let us first define *decision trees*. Decision trees are composed by vertices (also called nodes) and edges. Nodes are usually graphically represented by a circle or a square containing the name of a variable, a constant value or a character indicating what time of transform or expansion is performed at that node. Edges are lines or arrows connecting nodes hierarchically paired with a label which can consist of a constant value, the name of a variable or a logical expression. Let us start by an example of a simple *Binary decision diagram*.

EXAMPLE 3.1: *Consider the logic function $f(x_3, x_2, x_1) = x_1 x_3 \vee x_2$. From this function's truth table we see that its function vector is $\boldsymbol{F} = \begin{bmatrix} 00110111 \end{bmatrix}^T$. Now, looking at figure 3.1 it is intuitive to notice that the terminal nodes (the last row of nodes) matches $\boldsymbol{F}$ and that the output for each combination of inputs can be reached by, at each node, following the edge associated with the value of the current variable.*

3.1 Binary decision diagrams

Example 3.1 is an example of a simple *Binary decision tree*. The difference between trees and diagrams in this context is that a tree is a direct representation of all the possible choices of input, without any kind of simplification, while a diagram is the result of a process of simplifying the tree according to specific rules. Looking at example 3.1 it is easy to see that for $\{x_1 = 0, x_2 = 0\}$, whatever value variable x_3 takes, the outcome will be the same. Therefore, that part of the tree can be omitted without any consequences. The same can be said about $\{x_1 = 0, x_2 = 1\}$ and $\{x_1 = 1, x_2 = 1\}$. In figure 3.2 we can see the simplified version of the tree from figure 3.1.
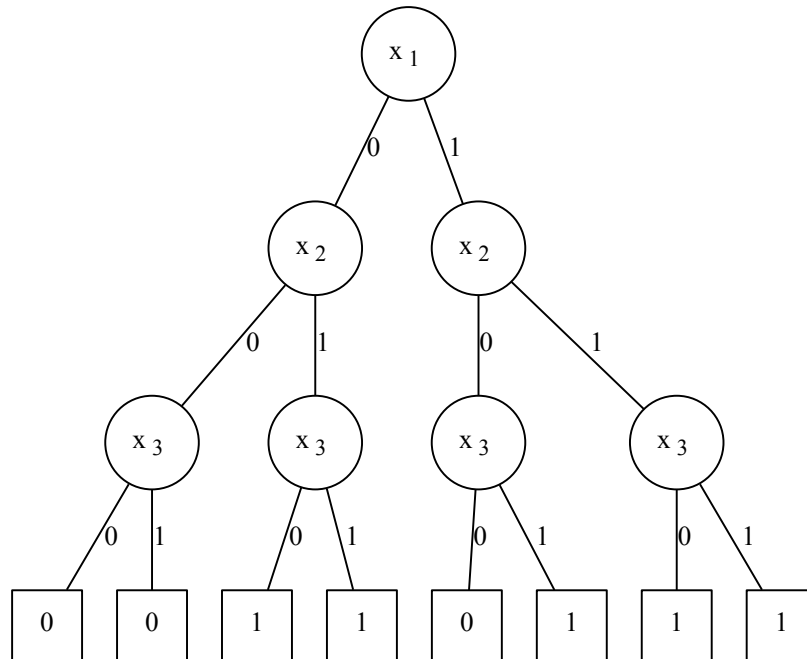
Figure 3.1: Decision Tree for f in Example 3.1
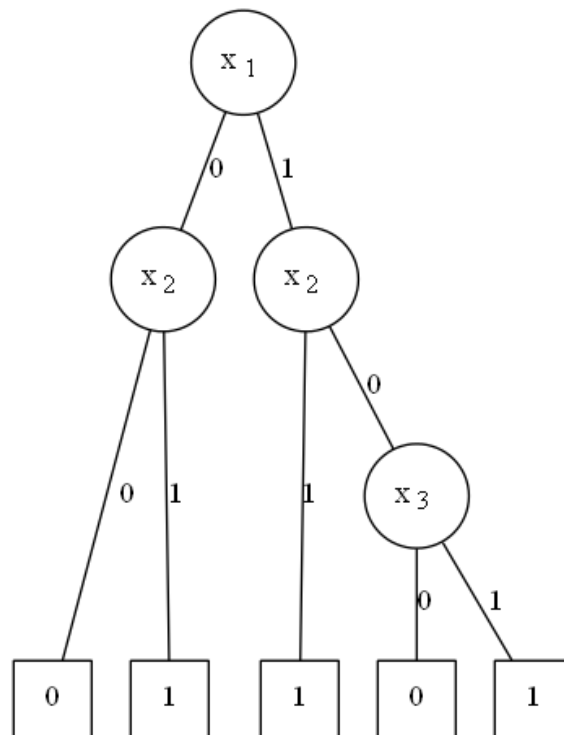
Figure 3.2: Simplified version of the tree in figure 3.1

We can still go further and delete the repeated nodes at the same level. In this case, in the 4th level (last row) there are three instances of a node "1" and two instances of a node "0", which can be reduced to the diagram in figure 3.3.
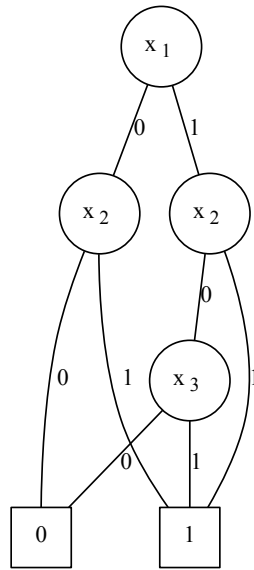
Figure 3.3: Decision diagram of the function in Example 3.1
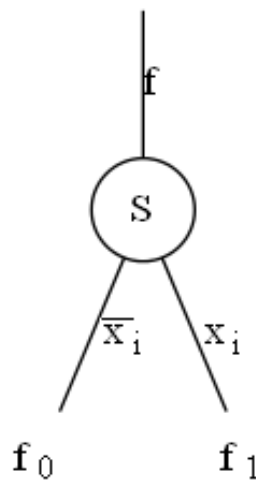
Figure 3.4: Shannon node.

The Shannon expansion can be represented in BDDs by changing the labels in the nodes and edges. Figure 3.4 shows a Shannon node, a graphical representation of the Shannon expansion.

It is easy to see that the operation performed at the node is indeed the Shannon expansion, written before as

$$f = \overline{x_i} f_0 \oplus x_i f_1.$$

We can build a complete decision tree by applying the Shannon expansion to all variables in the function. For n=2 variables, the respective tree is shown in figure 3.5
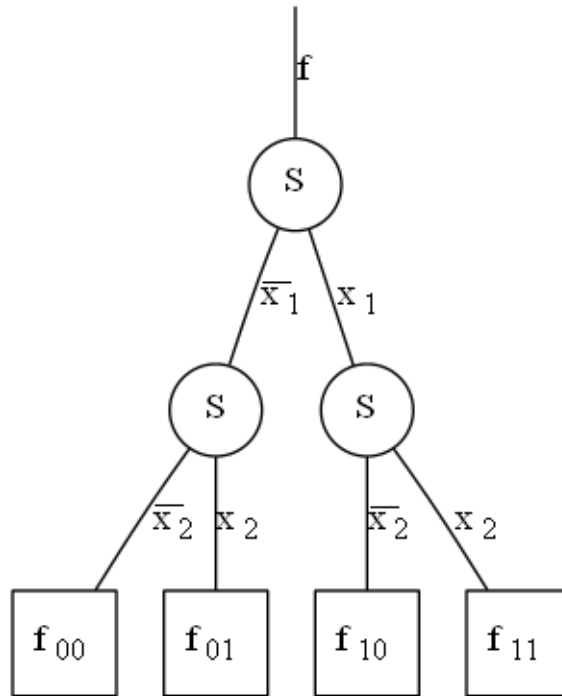
Figure 3.5: Binary Decision Tree for n=2.

3.2 Multi-terminal decision diagrams

BDDs are used to represent functions whose outputs are logic values. To handle integer-valued functions, such as the ones described in section 2.3, *Multi-terminal binary decision diagrams* can be used.

EXAMPLE 3.2 *Table 3.1 shows the truth-table for a 2-variable, 2-output function. Its outputs f_0 and f_1 can be combined into f_z and represented graphically as in figure 3.6.*

x_1,	x_2	f_0	f_1	f_z
0	0	1	0	2
0	1	1	1	3
1	0	0	1	1
1	1	0	0	0

Table 3.1: Truth-table for the 2-variable 2-output function in Example 3.2.
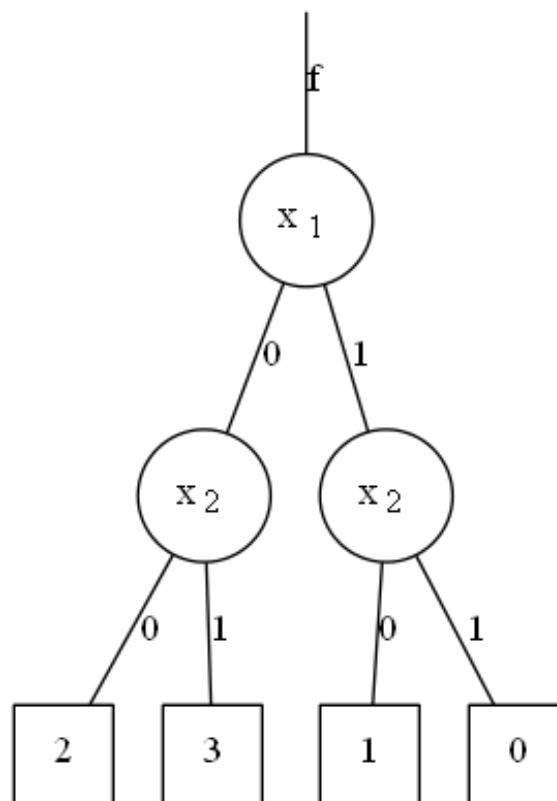
Figure 3.6: MTBDD for the function in Example 3.2.

3.3   Functional decision diagrams

Functional Decision Diagrams[12] are an extension of BDDs that aim on reducing the number of nodes generated for a certain function. FDDs use the properties of the Reed-Muller expansion to graphically represent a given function. A FDD can be created using either the pD or nD-expansion rules. Figure 3.7 shows the positive and negative Davio nodes that perform the functions $f = 1 \cdot f_0 \oplus x_i(f_0 \oplus f_1)$ and $f = 1 \cdot f_0 \oplus \overline{x_i}(f_0 \oplus f_1)$ respectively. It is important to notice that this kind of node always evaluates its edge labelled "1", since this term of the equation $(1 \cdot f_0)$ needs no condition to be true.

Figure 3.7: Positive and negative Davio nodes.

EXAMPLE 3.3 *Consider the function $f(x_3, x_2, x_1) = x_1 x_3 \vee x_2$ from example 3.1. We will begin by applying the pD expansion to the function.*

$$f = \left(\overset{n}{\underset{i=1}{\otimes}} X(1) \right) \left(\overset{n}{\underset{i=1}{\otimes}} R(1) \right) F$$

$$= X(3) \cdot \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$
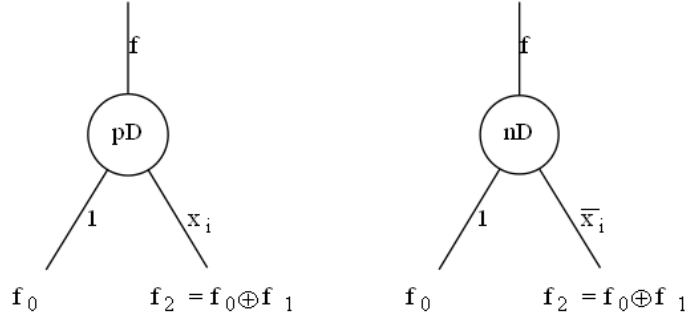
$$= X(3) \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

The vector resulting from $\boldsymbol{R}$(3)·F is called spectrum *of the function regarding the transform used. In this case it is represented as* S_R. *The spectrum is used to fill the terminal nodes in the decision tree. This fact is valid for all transforms. In the case of figure 3.1 the spectrum coincides with the function vector F because the transform matrix is the identity matrix. Figure 3.8 shows the FDT resulting from the pD transform.*

By following the edges from the terminal "1" nodes up to the root node we can find the Positive Polarity Reed-Muller Expression (PPRM) for the function.

$$f = x_2 \oplus x_3 x_1 \oplus x_3 x_2 x_1$$
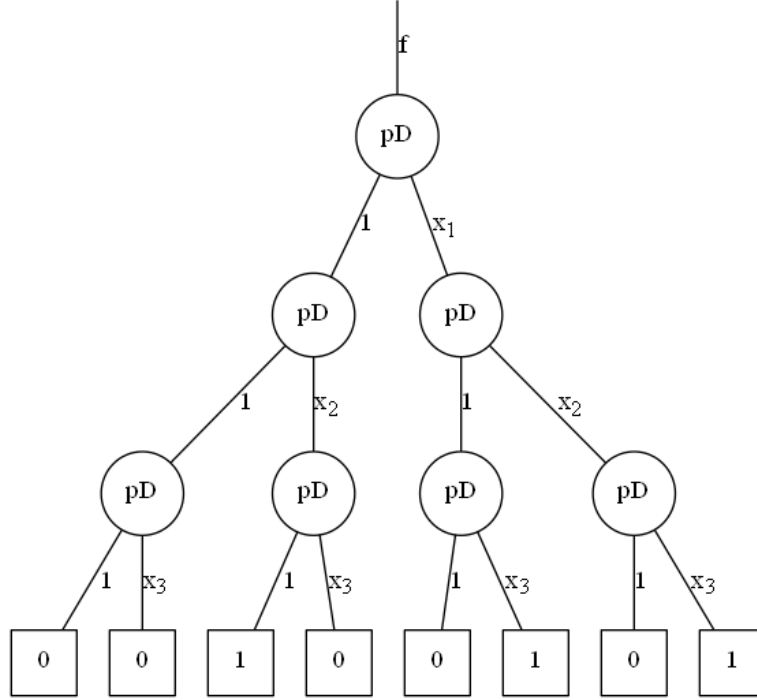
Figure 3.8: pD-FDT for f in example 3.3.

which is exactly function f *after applying the property* $a \vee b = a \oplus b \oplus ab$.
We could instead have chosen the nD expansion to get the spectrum as follows:

$$
\begin{aligned}
f &= \left(\overset{n}{\underset{i=1}{\otimes}} X(1) \right) \left(\overset{n}{\underset{i=1}{\otimes}} \overline{R}(1) \right) F \\
&= X(3) \cdot
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{bmatrix}
\cdot
\begin{bmatrix}
0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1
\end{bmatrix} \\
&= X(3) \cdot
\begin{bmatrix}
1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1
\end{bmatrix}
\end{aligned}
$$

which would result in the FDT in figure 3.9.
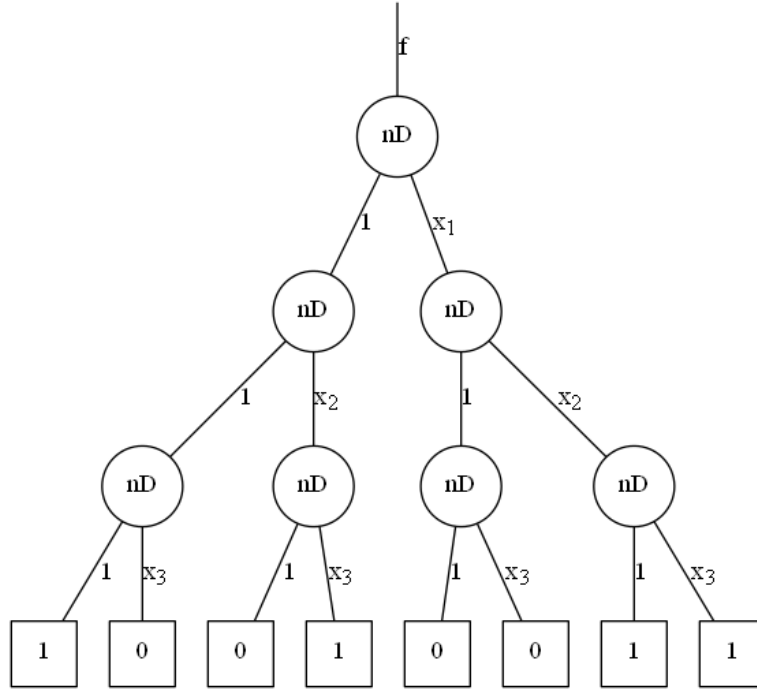
Figure 3.9: nD-FDT for f in example 3.3.

The resulting Negative Polarity Reed-Muller Expression(NPRM) is

$$f = 1 \oplus \overline{x_3 x_2} \oplus \overline{x_2 x_1} \oplus \overline{x_3 x_2 x_1}.$$

From example 3.3 we can see that choosing different polarities for the same transform we get expressions with different number of minterms. This fact is more visible after reducing the trees to diagrams. The reduction rules for FDDs are defined in [13] and are the folowing:

1. If the variable-labelled edge of a node points to "0", that node can be suppressed, since the value 0 does not contribute to the expression.

2. If two nodes in the same level are isomorphic (the sub-graphs constituted by the two nodes and their descendants are equal) one of the nodes and respective sub-graph can be deleted and each edge that pointed to it now points to the node that remained.
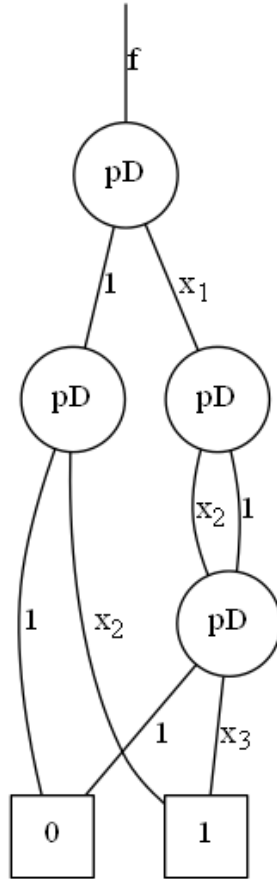
Figure 3.10: FDD of the tree in figure 3.8.

In figure 3.10 the rules above have been applied to the tree in figure 3.8. The 2 leftmost nodes in the 3rd level have been suppressed due to rule 1 and the edges that pointed to them now point directly to "1" and "0". The 2 rightmost nodes in the same level are isomorphic and have bee combined into one, according to rule 2. The same rules have been applied to the tree in figure 3.9 to get the diagram in figure 3.11

The number of minterms in the PPRM or NPRM is can be predicted by counting the number of edges that end in the terminal node "1" in FDD. Although not directly related, as with the number of minterms, the number of nodes in the FDD also varies with the choice of polarity.

3.3.1   Kronecker decision diagrams

The goal of using FDDs is to create more compact representations of functions. We have seen that different polarity choices yield different results in terms of number of nodes in the diagram and number of minterms in the expression. We can, however, try to refine the polarity choice by allowing the assignment of different polarities to each variable in a function as seen in section 2.2.1. To define the polarity assigned to each variable we can
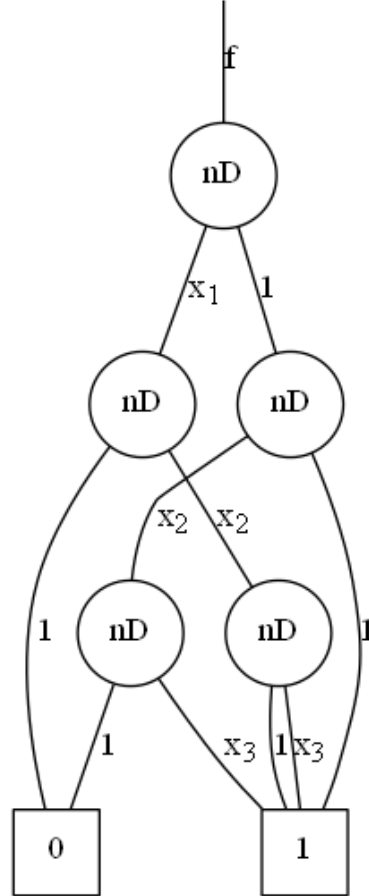
Figure 3.11: FDD of the tree in figure 3.9.

use a *Decision type list* (DTL) that can take the form *DTL= (pD, pD, nD)*, for example, or use, for simplicity, the H vector defined in section 2.2.1.

3.4   Arithmetic transform decision diagrams

In section 2.3 we introduced Word-level expressions, namely the Arithmetic transform defined as

$$f = \left(\mathop{\otimes}_{i=1}^{n} X_{Ah}(1) \right) \left(\mathop{\otimes}_{i=1}^{n} A_h{}^{-1}(1) \right) F,$$

$$A^{-1}(1) = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$$
$$X_A(1) = \begin{bmatrix} 1 & x_i \end{bmatrix}$$

for $h_i = 0$ and

$$A^{-1}(1) = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}$$
$$X_A(1) = \begin{bmatrix} 1 & \overline{x_i} \end{bmatrix}$$

for $h_i = 1$.

Its corresponding decision diagram is called *Arithmetic transform decision diagram (ACDD)*. It takes integer numbers as terminal node values and performs operations over the rational field (see table 2.1).

EXAMPLE 3.4: *Let f be a function with n=3 variables defined by the function vector $F = [0, 1, 2, 2, 2, 1, 2, 1]^T$. Its arithmetic spectrum for the polarity vector $H = (0, 0, 0)$ is*

$$A_{000} = [0, 1, 2, -1, 2, -2, -2, 1]^T$$

from which we can build the ACDT in figure 3.12.
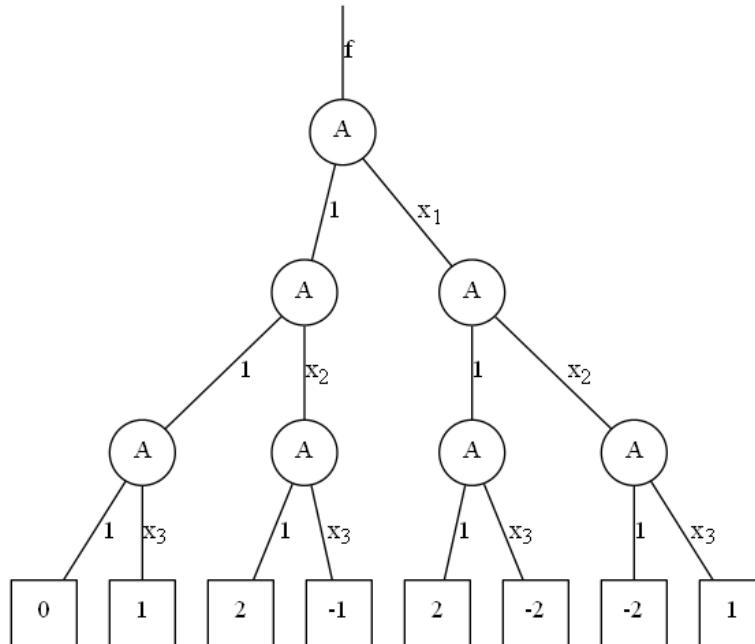
Figure 3.12: ACDT for the function f in example 3.4 using the polarity vector $H = (0, 0, 0)$.

Looking at figure 3.12 it is easy to notice that the polarity chosen for this example didn't produce a very efficient decision tree. Knowing that the rules to reduce ACDTs are the same as the ones to reduce FDTs, we can see that the only possible reduction for the tree in this example would be to merge some of the terminal nodes.

EXAMPLE 3.5 *Taking the function from the previous example, let's now try to find the spectre for the polarity vector $H = (1, 0, 1)$, which is*

$$A_{101} = [1, 1, 0, 0, 0, -2, 1, 1]^T.$$

With this spectrum we can build the tree in figure 3.13,
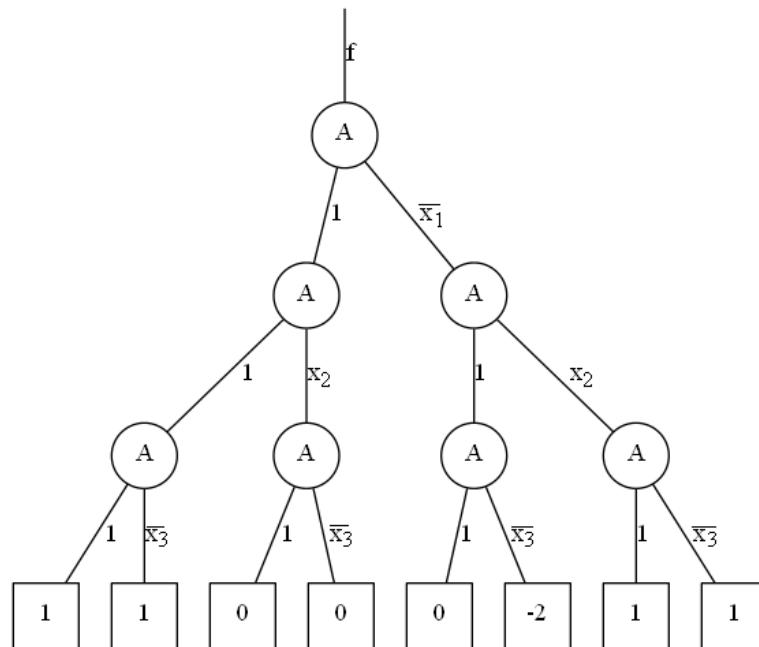
Figure 3.13: ACDT for the function f in example 3.4 using the polarity vector $H = (1, 0, 1)$.

which can be reduced to the ACDD in figure 3.14.
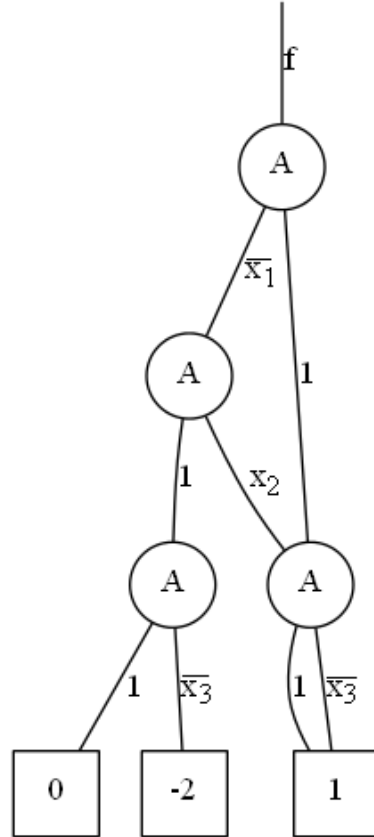
Figure 3.14: ACDD corresponding to the ACDT in figure 3.13.

Example 3.5 shows us how important it is to choose the right polarity when using the arithmetic transform. In this case, choosing the right polarity allowed us to reduce the tree to less than half of the nodes it originally had. Finding the right polarity is an iterative process, and may cost a lot of time for functions with a large number of variables.

The functional expressions from chapter 2 directly translate to the decision diagrams presented in this chapter. Besides the polarity choices discussed for functional expressions, the reduction of decision diagrams has a major influence in the complexity of the hardware implementation. For instance, the number of resulting non-terminal nodes will correspond to the number of modules in the network that constitutes the circuit, and the number of edges will determine the number of interconnections. The reduction rules presented in this chapter aim for the reduction of the number of nodes, which will provide more compact implementations of these diagrams in hardware.

Part III

Implementation

Chapter 4

Implementation

To test the efficiency of a good polarity choice for ACDDs and their ability to effectively represent a logic function, we will try to replicate the Numerical Function Generator built in [14] using ACDDs instead of MTBDDs to implement the segment index encoder. The purpose of this NFG is to generate combinational circuits that compute mathematical functions in a compact and fast way. The NFG takes as input any real function *f(x)* bounded over a domain [a,b] as well as the precision and accuracy desired for the system. The function will be first partitioned into segments and linearised according to the method described in [15]. An ACDD will be found that conveniently maps each input x into its corresponding segment, and a *segment index encoder* will be built based on the ACDD. The rest of the circuit will be constructed based on the coefficients obtained from the segmentation.

4.1   Stages of the Creation of the NFG

.

Many different steps need to be taken when creating an NFG. In this particular case, those steps are illustrated in figure 4.1. First, a MATLAB function receives the inputs by the user which define the function to be generated, as well as its domain, precision, accuracy and acceptable error. This function will compute the right coefficients as well as an appropriate ACDD to implement it. The results of this function will be printed in VHDL templates previously generated. A simulation tool allows us to verify if the results correspond to the ones expected. In the development stage of the project this allowed us to correct some imperfections both in the templates and in the MATLAB functions. If the configured templates are deemed good to go, a synthesis tool converts the VHDL code into a bit stream that will configure the FPGA where the NFG will be implemented. A design report is generated by the synthesis tool that allows us to verify the resources occupied and the maximum combinational delay between the NFG inputs and outputs.

4.2   Segmentation

The problem of segmenting a function can be solved in two ways, implying we have a *maximum approximation error*. One can opt for a *uniform segmentation* in which all
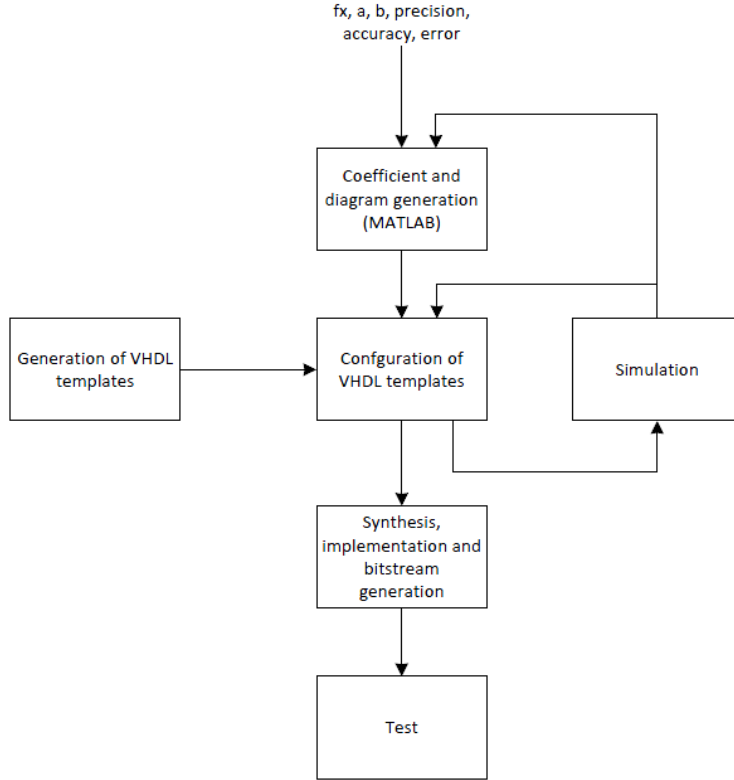
Figure 4.1: Flowchart depicting the stages in the creation of an NFG.

segments will have the same length, dictated by how small a certain segment has to be to fulfil the *approximation error* required. This technique requires that we store in memory the length of the segments,the value of f at the starting point of each segment, which will be called c_0 and the slope c_1 of the linear function that approximates the segment. This will require a total of $(1 + 2 \times \#segments) \times precision$ bits to be stored in memory. Although each segment only needs $2 \times precision$ bits, this technique can lead to an unnecessarily large number of segments in functions that have a fast growth in a small part of their domain and remain relatively stagnant in most of it. An alternative technique is to use *non-uniform segmentation*, in which we try to make each individual segment as long as possible, while respecting the *approximation error*. This will require that we also store in memory the starting point of each segment, since their lengths will no longer be constant. A particular segment's length can be inferred by subtracting its starting point to the starting point of the next segment, or the end of the function's domain, be it the last segment. This will require a total of $(3 \times \#segments) \times precision$ bits to be stored in memory. Although each individual segment takes more memory than the ones that result from *uniform segmentation*, this technique will in most cases require a much smaller number of segments, resulting in a net memory space saving.

The method chosen to perform the *non-uniform segmentation* was proposed by *D.H. Douglas* and *T.K. Peucker* in [15]. This algorithm is not optimal in some cases, but it is simple and robust, and can be applied to any function in this project's context. [14] has a very good description of the algorithm that has been adapted in figure 4.2

Notice that this algorithm has an output v called *vertical correction value*. This

Input:	Numerical function $f(x)$, domain $[a, b]$ over x, accuracy m and acceptable approximation error ε_a.
Output:	Segments $[s_0, e_0], \ldots, [s_{l-1}, e_{l-1}]$ and vertical correction values $v_0, \ldots, v_{l-1}$.

Process:	Recursively compute segments. Set the current segment to $[a, b]$
1.	Approximate $f(x)$ in the current segment $[s, e]$ by the linear function $g(x) = c_1 x + c_0$, where $c_1 = \frac{f(e) - f(s)}{e - s}$ and $c_0 = f(s) - c_1 s$.
2.	Find a value p_{max} of the variable x that maximizes $f(x) - g(x)$ in $[s, e]$, and let $max_{fg} = f(p_{max}) - g(p_{max})$, $(max_{fg} \geq 0)$.
3.	Find p_{min} that minimizes $f(x) - g(x)$ in $[s, e]$ and let $min_{fg} = f(p_{min}) - g(p_{min})$, $(min_{fg} \leq 0)$.
4.	Let $p = p_{max}$ if $|max_{fg}| > |min_{fg}|$, and let $p = p_{min}$ otherwise.
5.	Let $error = \frac{|max_{fg} - minfg|}{2}$ and $v = \frac{(maxfg + minfg)}{2}$.
6.	If $error \leq \varepsilon_a$, then declare $[s, e]$ to be a complete segment. If all segments are complete, stop.
7.	For any segment $[s, e]$ that is not complete, partition $[s, e]$ into two segments $[s, p]$ and $[p, e]$, declare each as the current segment, and go to 1.

Figure 4.2: Douglas-Peucker segmentation algorithm.

value is used to improve the approximation by shifting it vertically when $f(x)$ is convex ot concave in the given segment. Using the outputs from the Douglas-Peucker algorithm we can now derive the necessary coefficients to implement the function in hardware. For each segment i the function can be rebuilt as

$$g(i) = c_{1i} x + c_{0i}$$

with

$$c_{1i} = \frac{f(e_i) - f(s_i)}{e_i - s_i} \; and \; c_{0i} = f(s_i) - c_{1i} s_i + v_i.$$

In order to save memory space in the hardware implementation we can substitute c_{0i} into $g_i(x)$ to yield

$$g_i(x) = c_{1i}(x - s_i) + f(s_i) + v_i,$$

which does not require the coefficient c_{0i} to be stored in memory. If we also store $f(s_i) + v_i$ as a single coefficient we will save a total of $(2 \times \#segments \times precision)$ bits.

4.3   Arquitecture of the NFG

Figure 4.3 shows the architecture for NFG realizing $g_i(x) = c_{1i}(x - s_i) + f(s_i) + v_i$, where n is the precision used in the design and l is the number of segments obtained by the Douglas-Peucker algorithm. The input z is the domain of the function, $[a, b]$ divided in 2^n equidistant points encoded in n-tuples starting from $(00 \ldots 00)$, $(00 \ldots 01)$ up to $(11 \ldots 11)$. Using this encoding allows to treat z as an unsigned variable, which in turn will cause the segment encoding function to be a monotonically increasing function,
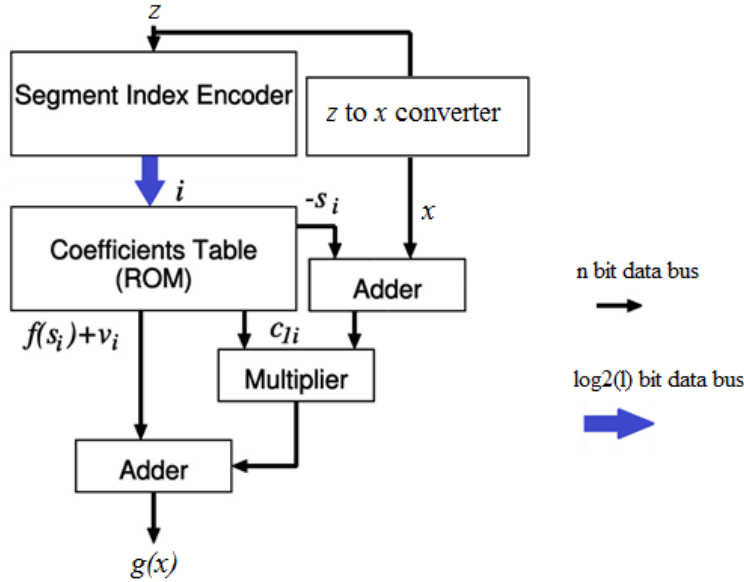
Figure 4.3: Overview of the NFG architecture.

making its implementation simpler, and any incorrection will be easier to notice during its development.

Apart from z and the segment number i, every variable and coefficient in this NFG is treated as a signed, n-precision, m-accuracy fixed-point binary number.

4.4   Segment Index Encoder

The segment index encoder detects to which segment the input z corresponds its output is read by the coefficients table to decide which are the right coefficients to be used by the arithmetic blocks of the NFG. To build it, we create a table of correspondence between each input and its segment and treat it as if it were the function vector of a word-level expression. Then, we apply all possible polarities of the Arithmetic Transform to this function vector and find the one that gives us the spectre with the least non-zero coefficients. This step is the lengthiest of all this process, and the one that limits the precision we are allowed to use in the representation, as the number of Kronecker products required is $n \times 2^n$. Although the time it takes to compute the Kronecker product itself depends on the number of variables, as do other parts of the program, the number of iterations required is the main cause of high execution times for a large number of variables.

After we find the spectre with the least non-zero coefficients the proposed algorithm builds an ACDD as a $k \times 3$ matrix, k being the number of nodes, where each row represents a node, the first column being a tag that identifies the node, and the last two columns being pointers to its left and right children, respectively. The algorithm is represented in figure 4.4.

This algorithm will not add a new node to the tree if an equal node is already there. This will guarantee that isomorphic nodes will be shared. The condition in point 2 makes

Input:	$1 \times n$ vector $Seq = Spectrum^T$
Output:	$k \times 3$ matrix $Tree$.

Process:	Recursively compute which nodes will belong in the reduced diagram. Set Tree= [].
1.	If $length(Seq) > 1$, $A = Seq(1 : length(Seq)/2)$, $B = Seq(length(Seq)/2 + 1 : end)$. If Seq has length 1, go to 4.
2.	If B is filled with zeros, replace each pointer to Seq with a pointer to A, Set $Seq = A$ and go to 1.
3.	If condition 2 does not verify, add the row $[Seq, A, B]$ to the Tree. Declare both A and B as the current Seq, and go to 1.
4.	If $length(Seq) = 1$, we have arrived at a terminal node. Add the row $[Seq,' X',' X']$ to the tree.

Figure 4.4: Algorithm for building the ACDD from the arithmetic spectrum.

sure we suppress nodes whose variable-labelled edge points to zero. As we can see, this algorithm immediately builds a reduced diagram from the spectrum without having to build the tree first and then reducing it.

4.5 FPGA Testing

The algorithms and procedures detailed in this chapter are performed in MATLAB software and implemented and tested in a Nexys 2 FPGA development kit [16]. The general structure of the NFG, the arithmetic blocks, and the templates for the segment index encoder, z to x converter and coefficients table are coded in VHDL, as well as the node block seen in figure 4.5, that will be replicated in the encoder to build the ACDD. The templates for the coefficients table and the z to x converter can be found in Appendix A.1 and A.2 respectively. The tags –*Start* and –*Finish* are used by MATLAB to find the place to insert the respective values.
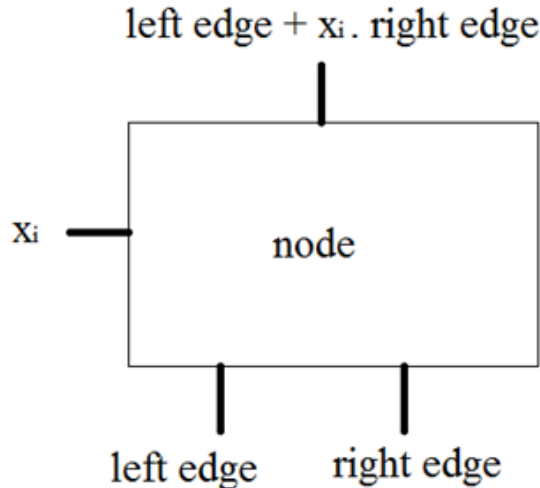

Figure 4.5: Node block.

The MATLAB function implemented will, after performing all the procedures already described, write the coefficients c_{1i}, s_i and $f(s_i) + v_i$ in the VHDL template for the coefficients table and write the correspondence table between z and x in the z to x converter template. As for the segment index encoder, it will first check the polarity vector H and negate the variables z_i when $h_i = 1$ and then create a decision diagram by replicating and connecting node blocks according to the matrix *tree*, generated previously. The terminal nodes will be treated as constants and will be connected directly into their parent nodes. Appendix A.3 shows the encoder generated for a 4 variable implementation of the $sin(x)$ function. Figure 4.6 gives us an overview of the processes executed in software.
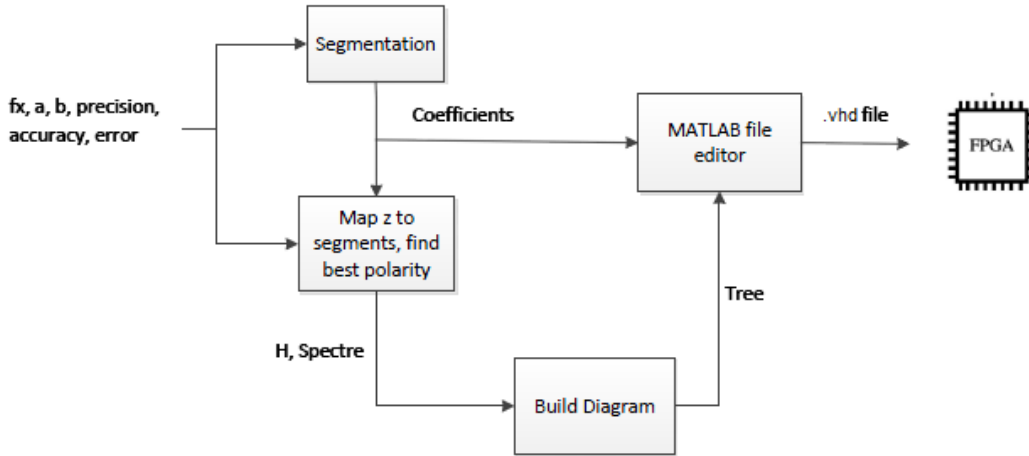
Figure 4.6: Software flowchart depicting the processes from the input of the function up to the writing of the data into the .vhd files.

4.6   Results

In this section some results will be presented and discussed. Table 4.1 shows the processing times for the polarity search algorithm for a varying number of variables. As stated in section 4.4, the execution time grows at a rate higher than $n \times 2^n$, as the time to compute the Kronecker product also increases with the number of variables. For 12 variables, this algorithms takes over 17 minutes (up from 2 for n=11), which makes it unreasonable to test it for any larger number.

Tables 4.2 and 4.3 show some results for the optimal polarity, as chosen by our algorithm, and for zero polarity ($H = (0, 0, \ldots, 0)$). From these results we can see that, in fact, the polarity choice algorithm is what is taking most of the processing time, as seen by the difference between the 2 tables for the same functions. An interesting thing to notice is that, although in most cases the optimal polarity produces a lower number of nodes, there are 2 instances where the opposite occurs. This shows that the criterion used to choose an "optimal" polarity is not entirely correct. Our assumption that the least number of non-zero coefficients produces smaller diagrams may not hold true, due to the position in which these coefficients may occur. Lets think, for example, that a

Number of variables	Processing time (s)
5	0.0117
6	0.0322
7	0.0857
8	0.2718
9	2.3226
10	17.5825
11	135.9343
12	1060.2

Table 4.1: Processing time of the iterative polarity search algorithm.

certain polarity yields 50% zero coefficients, but they all occur in terminal nodes that are connected to the left edge of their parent nodes. In this extreme case, the rule 1 stated in chapter 3.3 can not be applied to this diagram, while another polarity may yield more non-zero coefficients, but they may occur in more suitable places. These processing times were obtained using an Intel Core i5 CPU @ 2.3GHz with 4Gb of RAM, running the 64-bit version of the Windows 7 operating system.

Function	Processing time (s)			Number of nodes			Number of segments
	n=8	n=9	n=10	n=8	n=9	n=10	
$\sin(x)$ $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$	0.7658	3.12	19.4063	29	52	64	7
$\cos(x)$ $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$	0.8008	3.2391	20.2206	65	103	107	8
$\sqrt{x}$ $[0, 1]$	0.6793	3.0241	19.7362	9	9	12	5
$\ln(x)$ $\left[\frac{1}{100}, 1\right]$	0.7940	3.1979	19.9687	46	60	117	9

Table 4.2: Results for optimal polarity.

Function	Processing time (s)			Number of nodes			Number of segments
	n=8	n=9	n=10	n=8	n=9	n=10	
$\sin(x)$ $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$	0.5258	0.8395	1.8297	44	71	105	7
$\cos(x)$ $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$	0.0.5511	0.8967	1.9794	59	100	159	8
$\sqrt{x}$ $[0, 1]$	0.4398	0.6940	1.4745	16	16	33	5
$\ln(x)$ $\left[\frac{1}{100}, 1\right]$	0.6436	1.0034	2.4481	80	115	318	9

Table 4.3: Results for zero polarity.

From table 4.4 we can see that the resources occupied by the 8 variable implementation are minimal for optimal polarity, since in the worst case for optimal polarity only 2% of the total resources are used. For that same worst case, the delay would allow this circuit to operate at speeds up to 26MHz. As expected, the number of occupied resources increases with the number of nodes. The maximum delay can also be roughly correlated to the number of nodes, although it actually depends on the disposition of the nodes in the tree. This means that the maximum delay will be dictated by the maximum number of nodes in a certain path from the root of the tree to each terminal node, since each node induces additional delay in that path.

Tables 4.5 and 4.6 show the results for 9 and 10 variable implementation. The

n=8	Optimal polarity			Zero polarity		
Function	Slices		Delay (ns)	Slices		Delay (ns)
	Number	%		Number	%	
$\sin(x)$ $[-\frac{\pi}{2}, \frac{\pi}{2}]$	58	1	31.898	80	1	32.335
$\cos(x)$ $[-\frac{\pi}{2}, \frac{\pi}{2}]$	110	2	38.377	105	2	37.423
$\sqrt{x}$ $[0, 1]$	27	1	26.487	29	1	21.375
$\ln(x)$ $[\frac{1}{100}, 1]$	63	1	32.648	185	3	44.841

Table 4.4: Resources occupation and maximum combinational delay for n=8.

difference in occupation between the optimal and the zero polarity implementation is increasingly noticeable as the number of variables increases. The worst case for optimal polarity still only takes 4% of the total resources available, and allows the circuit to operate at 23.9MHz.

n=9	Optimal polarity			Zero polarity		
Function	Slices		Delay (ns)	Slices		Delay (ns)
	Number	%		Number	%	
$\sin(x)$ $[-\frac{\pi}{2}, \frac{\pi}{2}]$	79	1	33.185	130	2	38.052
$\cos(x)$ $[-\frac{\pi}{2}, \frac{\pi}{2}]$	152	3	39.491	158	3	41.868
$\sqrt{x}$ $[0, 1]$	28	1	25.375	29	1	29.352
$\ln(x)$ $[\frac{1}{100}, 1]$	95	2	37.947	331	7	46.705

Table 4.5: Resources occupation and maximum combinational delay for n=9.

n=10	Optimal polarity			Zero polarity		
Function	Slices		Delay (ns)	Slices		Delay (ns)
	Number	%		Number	%	
$\sin(x)$ $[-\frac{\pi}{2}, \frac{\pi}{2}]$	114	1	38.762	227	4	45.549
$\cos(x)$ $[-\frac{\pi}{2}, \frac{\pi}{2}]$	174	3	39.334	319	6	44.874
$\sqrt{x}$ $[0, 1]$	42	1	30.947	53	1	33.929
$\ln(x)$ $[\frac{1}{100}, 1]$	211	4	41.688	860	18	50.189

Table 4.6: Resources occupation and maximum combinational delay for n=10.

In this chapter we presented an example of the implementation of an NFG using ACDDs. This is one of many examples where the arithmetic transform can be used to represent switching functions. In this particular case an elementary function was segmented and linearised and an ACDD was used to map its arguments to the corresponding segment. All the coefficients and the structure of the diagram were obtained in MATLAB and the circuit was implemented in FPGA.

Part IV

Conclusions

Chapter 5

Conclusions

The interest in developing compact NFGs using Decision Diagrams has been growing in recent years. The group led by Prof. Sasao in particular has made some relevant advances in this technology [5] [6] [7] [17]. However, we are interested in creating algorithms for general-purpose software that can properly compute the necessary coefficients to implement an NFG in FPGA.

The algorithms developed for MATLAB were mostly satisfactory, being able to create functional NFGs for the various functions that were tested. For the user, the process is almost transparent. All it takes is to input the function, the domain, the precision and accuracy desired and the maximum acceptable error, and the outputs are automatically printed into the VHDL templates, ready for synthesis.

The diagram reduction algorithm is particularly efficient, being able to directly map the diagram from the Arithmetic Spectrum, with no need to build a tree first and performing reductions *a posteriori*. There is, of course, room for improvement in this project. The iterative polarity search algorithm is inefficient in terms of time consumption, as not always yields the best possible polarity. To improve the time consumption, a recursive algorithm could be used instead, such as the one being developed by D. Jankovic, R. Stankovic and C. Moraga [18] [19]. The criterion for the choice of polarity has been shown to not always be optimal. We could, for instance, chose the polarity yielding the lower number of distinct non-zero coefficients. Such criterion would also not be optimal, which can be seen by looking, for example, at the spectres $A = [0102]^T$ and $B = [1230]^T$. This criterion would choose spectre A, but we can see that its diagram could not be reduced, while spectre B's rightmost node could be suppressed. A possible improvement would probably involve predicting the positions where the non-zero coefficients would be, which would slow our algorithm even further. The limitation in precision prevents us from comparing our results with those published by other authors, but we believe that with a more efficient polarity search algorithm we could compete with NFGs built using software specialised for that purpose, since we are obtaining really low hardware resource consumption.

Bibliography

[1] C. Y. Lee. "Representation of Switching Circuits by Binary-Decision Programs". Bell Systems Technical Journal, 38:985-999, 1959.

[2] Sheldon B. Akers. "Binary Decision Diagrams", IEEE Transactions on Computers, C-27(6):509-516, June 1978.

[3] Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". IEEE Transactions on Computers, C-35(8):677-691, 1986.

[4] Lai, Y.F., Pedram, M., Vrudhula, S.B.K., "EVBDD-based algorithms for integer linear programming, spectral transformation, and functional decomposition," IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, Vol.13, No.8, 1994, 959-975.

[5] Sasao, T., Butler, J.T., Nagayama, S., "A systematic design method for two-variable numeric function generators using multiple-valued decision diagrams", IEICE TRANS. INF SYST., VOL.E93-D, NO.8 AUGUST 2010.

[6] S. Nagayama and T. Sasao, "Representations of two-variable elementary functions using EVMDDs and their applications to function generators", 38th International Symposium on Multiple-Valued Logic, pp.50.56, Dallas, U.S.A., May 2008.

[7] Sasao, T., Butler, J.T., Nagayama, S., "Numeric Function Generators Using Piecewise Arithmetic Expressions", 41st IEEE International Symposium on Multiple-Valued Logic, 2011

[8] J.-M. Muller, "Elementary Function: Algorithms and Implementation", Birkhauser Boston, Inc., Secaucus, NJ, 1997.

[9] R. Stankovic and J. Astola, "Remarks on the complexity of arithmetic representations of elementary functions for circuit design," Workshop on Applications of the Reed-Muller Expansion in Circuit Design and Representations and Methodology of Future Computing Technology, pp. 5?11, May 2007.

[10] Edward A. Bender, S. Gill Williamson, 2005, "A Short Course in Discrete Mathematics", Dover Publications, Inc., Mineola, NY, ISBN 0-486-43946-1.

[11] Shannon, C.E., "A symbolic analysis of relay and switching circuits", emphBell Sys. Tech. J., Vol. 28, No. 1, 1949, 59-98.

[12] Kebschull, U., Schubert, E., Rosenstiel, W., "Multilevel logic synthesis based on functional decision diagrams," EDAC 92, 1992, 43-47.

[13] Minato, S., "Zero-suppressed BDDs for set manipulation in combinatorial problems," Proc. 30th ACM/IEEE DAC, June 1993, 272-277.

[14] Sasao, T., Butler, J.T., Nagayama, S., "Numerical function generators using LUT cascades", IEEE Transactions on Computers, Vol. 56, No. 6, pp. 826-838, Jun. 2007

[15] D.H. Douglas and T.K. Peucker, "Algorithms for the Reduction of the Number of Points Required to Represent a Line or Its Caricature," The Canadian Cartographer, vol. 10, no. 2, pp. 112-122, 1973.

[16] Digilent Nexys2 Board Reference Manual. 2011. Digilent. 11 July 2011 http://www.digilentinc.com/Data/Products/NEXYS2/Nexys2_ rm.pdf.

[17] Sasao, T., Nagayama, S., "Complexities of Graph-Based Representations for Elementary Functions", IEEE Transactions on Computers, Vol. 58, No. 1, pp. 106-119 Jan. 2009

[18] D. Jankovic, R. Stankovic, C. Moraga, "Arithmetic Expressions Optimisation Using Dual Polarity Property," Serbian Journal of Electrical Engineering, Vol. 1, No. 1, November 2003, 71 - 80

[19] D. Jankovic, R. Stankovic, C. Moraga "Optimization of Polynomial Expressions by Using the Extended Dual Polarity", IEEE Transactions on Computers, Vol. 58, No. 12, Dec. 2009

Appendix A

VHDL Templates

A.1   Coefficients Table

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.all;


entity coef_table_backup is
generic(prec : integer := 8;
  segments : integer:= 32;
  addrBusSize : integer := 3);
port(segment : in  std_logic_vector((addrBusSize - 1) downto 0);
 coef_FsV : out std_logic_vector((prec - 1) downto 0);
 coef_S : out std_logic_vector((prec - 1) downto 0);
 coef_C : out std_logic_vector((prec - 1) downto 0));
end coef_table;


architecture RTL of coef_table_backup is


constant NUM_WORDS : integer := segments;


subtype TDataWord is std_logic_vector((prec - 1) downto 0);
type TMemory is array (1 to (NUM_WORDS)) of TDataWord;
signal FsV_memory : TMemory:=(--Start1
 --Finish1
 others => (others => '0'));
signal S_memory : TMemory:=(--Start2
 --Finish2
 others => (others => '0'));
signal C_memory : TMemory:=(--Start3
 --Finish3
 others => (others => '0'));


begin
coef_FsV <= FsV_memory(to_integer(unsigned(segment)-1));
```

```
coef_S <= S_memory(to_integer(unsigned(segment)-1));
coef_C <= C_memory(to_integer(unsigned(segment)-1));

end RTL;
```

A.2   Z to X Converter

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.all;

entity sw_to_arg is
generic(prec : integer := 8;
  acc : integer:= 5);
port(X : in  std_logic_vector((prec - 1) downto 0);
 arg : out std_logic_vector((prec - 1) downto 0));
end sw_to_arg;

architecture RTL of sw_to_arg is

subtype TDataWord is std_logic_vector((prec) downto 0);
type TMemory is array (0 to 2**prec-1) of TDataWord;
signal args : TMemory:=(--Start
 --Finish
 );

begin

arg <= args(to_integer(unsigned(X)));

end RTL;
```

A.3   Encoder for sin(x) [-pi/2, pi/2], n=4

```
library IEEE;use IEEE.STD_LOGIC_1164.ALL;use IEEE.NUMERIC_STD.all;
entity tree_encoder is
port(X: in std_logic_vector(1 to 4);
segment : out std_logic_vector((2 - 1) downto 0));
end tree_encoder;
architecture RTL of tree_encoder is
signal x1 : std_logic;
signal x2 : std_logic;
signal x3 : std_logic;
signal x4 : std_logic;
signal long_seg : std_logic_vector((4- 1) downto 0):=(others=>'0');
signal res_node10001000 : std_logic_vector((4 - 1) downto 0):=(others
=>'0');
signal res_node1 : std_logic_vector((4 - 1) downto 0):=(others=>'0');
begin


x1 <= X(1);
x2 <= X(2);
x3 <= X(3);
x4 <= X(4);


node_block1000100010000000 : entity WORK.node_block(Behavioral)
generic map(prec=>4,acc=>2)
port map(Xi=>x1,left_edge=>res_node10001000,right_edge=>res_node1,
f=>long_seg);



node_block10001000 : entity WORK.node_block(Behavioral)
generic map(prec=>4,acc=>2)
port map(Xi=>x2,left_edge=>res_node1,right_edge=>res_node1,f=>
res_node10001000);



res_node1<="0100";
segment <= long_seg(3 downto 2);

end RTL;
```

Acronyms

ACDD	Arithmetic decision diagram
ACDT	Arithmetic decision tree
BDD	Binary decision diagram
BDT	Binary decision tree
DTL	Decision type list
FDD	Functional decision diagram
FDT	Functional decision tree
FPGA	Field-programmable gate array
MTBDD	Multi-terminal decision diagram
NFG	Numerical function generator
NPRM	Negative-polarity Reed-Muller expression
PPRM	Positive-polarity Reed-Muller expression
SOP	Sum of products
VHDL	VHSIC hardware description language