NATIONAL TECHNICAL UNIVERSITY OF UKRAINE
«IGOR SIKORSKY POLYTECHNICAL INSTITUTE»

Informatic and Computer Engineering Faculty
Computer Engineering Department

«On the rights of the manuscript»
УДК __004.942_____

«Defence is allowed»
Head of the Computer Science Dep-t

_____ S.G. Stirenko
(sign) (name)

“___”_____2018p.

Master's thesis

In speciality _____123 Computer Engineering_____

Specialization: _____123. Computer systems and networks_____

theme: _____Method of increasing the efficiency of devices_____
_____for the calculation of elementary functions _____

Fulfilled: student of VI course, group _ IO 64м _____
(group sign)

_____Hasan Muhammad Jamal_____ _____
(Full Name) (signature)

Науковий керівник Ass.Prof., Dr.Sci, S.Sci. Sergiyenko A.M. _____ _____
(position, scientific degree, academic rank, surname and initials) (signature)

Reviewer Ass.Prof., Dr.Sci, Docent Romankevich V.O. _____ _____
(position, scientific degree, academic rank, surname and initials) (signature)

I certify that in this master's thesis there are no borrowings from the works of other authors without the corresponding references.
Student _____
(signature)

Kyiv – 2018

РЕФЕРАТ

Метод підвищення ефективності пристроїв для обчислення елементарних функцій

Актуальність теми. Програмовані логічні інтегральні схеми (ПЛІС) — це сучасна елементна база, яка призначена для високопродуктивного виконання спеціалізованих алгоритмів з числами, які представлені з фіксованою комою. Дуже часто в таких алгоритмах зустрічається обчислення елементарних функцій. Але поставники САПР ПЛІС не забезпечують розробників готовими високопродуктивними віртуальними модулями обчислення елементарних функцій, а фірми-розробники таких модулів поширюють їх за велику ціну (близько тисячі доларів США). Крім того, серед них не зустрічаються модулі, які спроможні обчислювати кілька різних функцій. Отже, існує нестача у проектах пристроїв для обчислення елементарних функцій у ПЛІС та вони потребують удосконалення.

Об’єктом дослідження є організація обчислювальних процесів у високопродуктивних спеціалізованих процесорах.

Предметом дослідження є проектування конвеєрних пристроїв для обчислення елементарних функцій.

Мета роботи: створення методу проектування високопродуктивних спецпроцесорів для обчислення елементарних функцій у ПЛІС.

Наукова новизна полягає в наступному:

1. Удосконалено алгоритм та пристрій обчислення функції квадратного кореня, завдяки чому ця функція обчислюється утричі швидше при невисоких апаратних витратах.

2. Розроблено метод підвищення ефективності пристроїв для виконання елементарних функцій, який основано на комбінуванні кількох алгоритмів обчислення таких функцій, завдяки чому стає можливою побудова високопродуктивних багатофункціональних пристроїв.

Практична цінність отриманих в роботі результатів полягає в тому, що розроблені за запропонованим методом модулі обчислення елементарних функцій є готовими для використання у сучасних проектах високопродуктивних систем на ПЛІС, які використовуються для цифрової обробки сигналів, машинного навчання, розпізнавання образів, тощо.

Матеріали роботи використані у науково-дослідній роботі «Удосконалені методи та засоби проектування конфігурованих комп'ютерів на основі відображення просторового графу синхронних потоків даних у структури на базі програмованих логічних інтегральних схем», № ДР.047U005087, шифр ФІОТ-30Т/2017, яка проводиться у НТУУ "КПІ ім. Ігоря Сікорського.

Апробація роботи. Основні положення і результати роботи були представлені та обговорювались на 20-тій міжнародній конференції «Системний аналіз та інформаційні технології» SAIT-2018 21 – 24 травня 2018 року, Київ та міжнародній конференції "Безпека, Відмовостійкість, Інтелект" 10 – 12 травня 2018 року, Київ.

Структура та обсяг роботи. Магістерська дисертація складається зі вступу, трьох розділів та висновків.

У вступі подано загальну характеристику роботи, зроблено оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень, сформульовано мету і задачі досліджень, показано наукову новизну отриманих результатів і практичну цінність роботи, наведено відомості про апробацію результатів і їхнє впровадження.

У першому розділі досліджено особливості архітектури сучасних ПЛІС, розглянуто та проаналізовано алгоритми для обчислення елементарних функцій та їх відомі реалізації в паралельних обчислювальних системах і ПЛІС.

У другому розділі удосконалено алгоритм та пристрій обчислення функції квадратного кореня та розроблено метод підвищення ефективності пристроїв для виконання елементарних функцій.

У третьому розділі досліджено ефективність використання запропонованих алгоритму обчислення квадратного кореня та методу підвищення ефективності пристроїв для виконання елементарних функцій.

У висновках представлені результати проведеної роботи.

Робота представлена на 68 аркушах, містить посилання на список використаних літературних джерел та додатки.

Ключові слова: ПЛІС, квадратний корінь, елементарна функція, конвеєр, граф синхронних потоків даних.

ABSTRACT

Method of increasing the efficiency of devices for the calculation of elementary functions

Relevance of the topic. The field programmable gate array (FPGA) is a modern element basis that is effectively utilized for the high performance implementation of application-specific algorithms with the fixed-point numbers. Very often, such algorithms encounter the calculation of elementary functions. But the suppliers of the FPGA CAD tools do not provide the developers with ready-made high-performance intellectual property cores for calculating the elementary functions, and the providers of such modules distribute them at a high price (about a thousand dollars). In addition, there are no modules among them that can calculate several different functions. Consequently, there are shortages in the design of devices for the calculation of elementary functions in FPGA and they need to be improved.

The purpose of the work: the creation of a method for designing the application specific modules for the elementary function calculation.

The object of the research is the computational processes in high-performance application-specific processors.

The subject of the research is design of pipelined processors for the elementary function calculations.

The objective is the creation of a method for designing the high-performance application-specfic processors for the calculation of elementary functions in FPGA.

The scientific novelty is as follows:

1. An algorithm and a structure of the square root calculator are improved, so this function is calculated three times faster with low hardware costs.

2. **A method for increasing the efficiency of devices for calculating the elementary functions** is developed, which is based on the combination of several algorithms for calculating such functions, which makes it possible to build high-performance multifunction devices.

The practical value of the results obtained in the work is that the modules for calculating the elementary functions, which are developed by the proposed method, are ready for use in modern projects of high-performance systems on FPGAs, which are used for digital signal processing, machine learning, image recognition, and others like that.

The materials of the thesis were used in the research work "Advanced methods and tools of designing the configurable computers on the basis of mapping the spatial synchronous data flow graphs into the structure for FPGA", № ДР.047U005087, ФІОТ-30Т / 2017, which is held at NTUU “Igor Sikorsky’s KPI”.

Approbation of the work. Substantive provisions and results of the work were presented and discussed at a 20-th International Conference «System Analysis and Information Technology» SAIT 2018 May 21 – 24, 2018, Kyiv, and International Conference on Security, Fault Tolerance, Intelligence (ICSFTI2018), May 10 – 12, 2018, Kyiv.

The structure and scope of work. Master's thesis consists of an introduction, three sections and conclusions.

The introduction gives a general description of the work, assesses the current state of the problem, substantiates the relevance of the research direction, formulates the purpose and objectives of the research, shows the scientific novelty of the obtained results and the practical value of the work, provides information on the approbation of the results and their implementation.

In the first section, the features of the architecture of modern FPGA have been investigated, algorithms for calculation of elementary functions and their known realizations in parallel computing systems and FPGAs are analyzed.

In the second section, an algorithm and a square root function calculator are improved, and a method for increasing the efficiency of devices to perform the elementary functions is developed.

In the third section, the efficiency of using the proposed square root calculation algorithm and the method of increasing the efficiency of devices for performing the elementary functions are investigated.

The conclusions show the results of the work.

The work is presented in 68 pages, contains a reference to the list of used literature and addendums.

Key words: FPGA, square root, elementary function, pipeline, SDF graph.

РЕФЕРАТ

Метод повышения эффективности устройств для вычисления элементарных функций

Актуальность темы. Программируемые логические интегральные схемы (ПЛИС) - это современная элементная база, которая предназначена для высокопроизводительного выполнения специализированных алгоритмов с числами, которые представлены с фиксированной запятой. Очень часто в таких алгоритмах встречается вычисления элементарных функций. Но поставщики САПР ПЛИС не обеспечивают разработчиков готовыми высокопроизводительными виртуальными модулями вычисления элементарных функций, а фирмы-разработчики таких модулей распространяют их за большую цену (около тысячи долларов США). Кроме того, среди них не встречаются модули, которые способны вычислять несколько различных функций. Итак, существует нехватка в проектах устройств для вычисления элементарных функций в ПЛИС и они нуждаются в усовершенствовании.

Объектом исследования является организация вычислительных процессов в высокопроизводительных специализированных процессорах.

Предметом исследования является проектирование конвейерных устройств для вычисления элементарных функций.

Цель работы: создание метода проектирования высокопроизводительных спецпроцессоров для вычисления элементарных функций в ПЛИС.

Научная новизна заключается в следующем:

1. Усовершенствована алгоритм и устройство вычисления функции квадратного корня, благодаря чему эта функция вычисляется в три раза быстрее при низких аппаратных затратах.

2. Разработан метод повышения эффективности устройств для выполнения элементарных функций, который основан на комбинировании нескольких алгоритмов вычисления таких функций, благодаря чему становится возможным построение высокопроизводительных многофункциональных устройств.

Практическая ценность полученных в работе результатов заключается в том, что разработанные по предложенному методу модули вычисления элементарных функций являются готовыми для использования в современных проектах высокопроизводительных систем на ПЛИС, которые используются для цифровой обработки сигналов, машинного обучения, распознавания образов и тому подобное.

Материалы работы использованы в научно-исследовательской работе «Усовершенствованные методы и средства проектирования конфигурируемых компьютеров на основе отображения пространственного графа синхронных потоков данных в структуры на базе программируемых логических интегральных схем», № ДР.047U005087, шифр ФІОТ-30Т / 2017, которая проводится в НТУУ "КПИ им. Игоря Сикорского".

Апробация работы. Основные положения и результаты работы были представлены и обсуждались на 20-ой международной конференции «Системный анализ и информационные технологии» SAIT-2018, 21 - 24 мая 2018, Киев и международной конференции "Безопасность, Отказоустойчивость, Интеллект" 10 - 12 мая 2018, Киев.

Структура и объем работы. Магистерская диссертация состоит из введения, трех глав и выводов.

Во введении представлена общая характеристика работы, произведена оценка современного состояния проблемы, обоснована актуальность направления исследований, сформулированы цели и задачи

исследований, показано научную новизну полученных результатов и практическую ценность работы, приведены сведения об апробации результатов и их внедрение.

В первом разделе исследованы особенности архитектуры современных ПЛИС, рассмотрены и проанализированы алгоритмы для вычисления элементар- ных функций и их известные реализации в параллельных вычислительных системах и ПЛИС.

Во втором разделе усовершенствована алгоритм и устройство вычисления функции квадратного корня и разработан метод повышения эффективности устройств для выполнения элементарных функций.

В третьем разделе исследована эффективность использования предложенных алгоритма вычисления квадратного корня и метода повышения эффективности устройств для выполнения элементарных функций.

В выводах представлены результаты проведенной работы.

Работа представлена на 68 страницах, содержит ссылки на список использованных литературных источников и приложения.

Ключевые слова: ПЛИС, квадратный корень, элементарная функция, конвейер, граф синхронных потоков данных.

CONTENTS

ABBREVIATIONS

ASIC	Application Specific Integrated Circuit
CPU	Central Processing Unit
DSP	Digital Signal Processing
FPGA	field programmable gate array
GPU	Graphic Processing Uunit
IC	Integrated Circuits
IP core	Intellectual Property core
LUT	Look-Up Table
PU	processing unit
RAM	Random Access Memory
ROM	Read-Only Memory
RTL	Register Transfer Level logic
SDF	Synchronous Data Flow graph
VHSIC	Very High Speed Integrated Circuits
VHDL	VHSIC Hardware Description Language
VLSI	Very Large Scale Integration

INTRODUCTION

Nowadays, when gadgets and computers are present in everyday aspect of our life, more advanced algorithms for shorter computational timing are tremendously important. Algebraic functions, for instance square root, logarithm, as well as trigonometric functions embrace the main source of algorithm implementation in domains like digital signal processing (DSP), wireless communication, graphic processing units (GPU), image processing, communication systems and medical robotics.

The performance of only software implementations of these algorithms is not satisfactory all the time, thus in order to improve the functionality, a translation of the software into hardware is desired.

The square root $\sqrt{x}$ and other elementary functions ares important in the scientific calculations, digital signal and image processing [1]. The artificial neural nets need these functions as well [2]. At present, the field programmable gate arrays (FPGAs) are expanded for solving the problems, where the elementary function calculations are of demand. There are different IP cores of the elementary function calculation, which are proposed by the FPGA manufacturers and third-party companies [3]. But these IP cores were designed decades ago and they usually don't take into account the features of the new FPGA generations. Therefore, they need improvements.

This thesis proposes the method of the design of the application specific hardware design, which is intended for the high speed elementary function calculations. The use of FPGAs to implement these functions allows us to increase the speed, reduce the power consumption. Moreover, the modernizing the elementary function blocks can be implemented in the device in use by the way of the reconfiguration of FPGA.

The object of the research is the high-performance application-specific processors.

The subject of the research is the structure of pipelined processors for the elementary function calculations.

The objective is the creation of a method for designing the high-performance application-specfic processors for the calculation of elementary functions in FPGA.

To achieve the objective, the following tasks are solved in the thesis:

1. The methods of the mathematical modeling of the wave propagation in solids, and their comuter implementation are analysed.

2. The method of the waveguide modeling is analysed and its application to the modeling the solids is investigated.

3. The method of hardware simulation of the propagation of ultrasonic waves in a solid based on the waveguide models is developed.

4. The method of hardware simulation the propagation of ultrasonic waves is adapted for its implementation in modern FPGAs.

5. The proposed method effectiveness is proven by modeling of the wave propagations in the solid rod.

The research methods used in the work are based on the theory of graphs, algorithm theory, modeling theory, combinatorial optimization methods, as well as theorems, assertions and implications that are proved in the dissertation. The main provisions and theoretical evaluations are confirmed by the results of simulation on a computer, as well as by tests of a number of experimental samples of specialized calculators.

Experimental verification of scientific positions, proposals and results was carried out by designing computing tools by the developed method using their description in standard VHDL language with their further simulation in the simulator, compiling in the circuit and configuring the Xilinx FPGA.

The scientific novelty is as follows:

1. An algorithm and a structure of the square root calculator are improved, so this function is calculated three times faster with low hardware costs.

2. A method for increasing the efficiency of devices for calculating the elementary functions is developed, which is based on the combination of several algorithms for calculating such functions, which makes it possible to build high-performance multifunction devices.

The practical value of the results obtained in the work is that the modules for calculating the elementary functions, which are developed by the proposed method, are ready for use in modern projects of high-performance systems on FPGAs, which are used for digital signal processing, machine learning, image recognition, and others like that.

The materials of the thesis were used in the research work "Advanced methods and tools of designing the configurable computers on the basis of mapping the spatial synchronous data flow graphs into the structure for FPGA", № ДР.047U005087, ФІОТ-30Т / 2017, which is held at NTUU “Igor Sikorsky’s KPI”.

Approbation of the work. Substantive provisions and results of the work were presented and discussed at a 20-th International Conference «System Analysis and Information Technology» SAIT 2018 May 21 – 24, 2018, Kyiv, and International Conference on Security, Fault Tolerance, Intelligence (ICSFTI2018), May 10 – 12, 2018, Kyiv.

Publications of the work

The main features of these investigations are published in two works. In the work [41] the author has proposed an approach, which provides the hardware minimization. In the work [5] the author has proposed the way to speed-up the calculations.

The structure and scope of the work

Master's thesis consists of an introduction, three sections and conclusions.

The introduction gives a general description of the work, assesses the current state of the problem, substantiates the relevance of the research direction, formulates the purpose and objectives of the research, shows the scientific novelty of the obtained results and the practical value of the work, provides information on the approbation of the results and their implementation.

In the first section, the features of the architecture of modern FPGA have been investigated, algorithms for calculation of elementary functions and their known realizations in parallel computing systems and FPGAs are analyzed.

In the second section, an algorithm and a square root function calculator are improved, and a method for increasing the efficiency of devices to perform the elementary functions is developed.

In the third section, the efficiency of using the proposed square root calculation algorithm and the method of increasing the efficiency of devices for performing the elementary functions are investigated.

The conclusions show the results of the work.

The work is presented in 70 pages, contains a reference to the list of used literature and appendicies.

1 METHODS AND TOOLS FOR ELEMENTARY FUNCTION CALCULATIONS

1.1 Basics of the elementary function calculations

1.1.1 Preliminary conditions

Usually the elementary functions in computer engineering are the most commonly used mathematical functions: sin, cos, tan, $\sin^{-1}$, $\cos^{-1}$, $\tan^{-1}$, sinh, cosh, tanh, $\sinh^{-1}$, $\cosh^{-1}$, $\tanh^{-1}$, exponentials, and logarithms. From a mathematical point of view, $1/x$ is an elementary function as well [6,7].

Theoretically, the elementary functions are not much harder to compute than quotients. It was in [8] that these functions are equivalent to division with respect to the Boolean circuit depth. This means that a circuit can output n digits of a sine, cosine, or logarithm in a time proportional to log n. But for practical implementations, it is quite different, and much care is necessary if we want fast and accurate elementary functions.

There are many works devoted to the elementary function algorithms [7,9,10]. But at times those functions were implemented in software only. Since the Intel 8087 floating-point unit, elementary functions have sometimes been implemented, at least partially, in hardware, a fact that induces serious algorithmic changes. Furthermore, the emergence of high-quality arithmetic standards, such as the IEEE-754 standard for floating-point arithmetic, have accustomed users to very accurate results. So, the investigations of the elementary function implementation in hardware is of great demand.

Current circuit designers must build algorithms and architectures that are guaranteed to be much more accurate and effective. Among the various properties that are desirable, when the function is implemented in FPGA, one can cite:

- speed;
- accuracy;

- reasonable amount of resource (ROM/RAM, LUTs, registers, power consumptions);
- preservation of important mathematical properties such as monotonicity, and symmetry; ;
- preservation of the direction of rounding;
- range limits, for example, $1.0 \leq \sin(x) \leq 1.0$. [6].

1.1.2 Algoritm classification

The hardware approximation algorithms can be classified into four broad categories.

The first category is the polynomial approximation. This category is a diverse category. The general description of this class is as follows: the interval of the argument is divided into a number of sub-intervals. For each sub-interval the elementary function is approximated by a polynomial of a suitable degree. The coefficients of such polynomials are stored in a table [10].

The second category is called functional recurrence. Algorithms that belong to this category employ addition, subtraction and full multiplication operations as well as tables for the initial approximation. In this class of algorithms the algorithm starts by a given initial approximation and it is feededt to a polynomial in order to obtain a better approximation. This process ise repeated a number of times until the desired precision is reached. These algorithms converge quadratically or better. Examples from this category include Newton-Raphson for square root [10].

The third category is called digit recurrence techniques, or shift-and-add algorithms. The algorithms that belong to this category are linearly convergent and they employ addition, subtraction, shift and single digit multiplication operations. Example of such algorithms is CORDIC [11, 12]

The fourth category is the rational approximation algorithms. In this category the given interval of the argument is divided into a number of sub-intervals. For each sub-interval the given function is approximate by a rational function. A rational function is y a polynomial divided by another polynomial. It employs division operation in addition to tables, addition and multiplication operations, which are used in the polynomial approximation. The rational approximation is rather costly in hardware due to the fact that it uses division [10].

Range reduction is the first step in elementary functions computation. It aims to transform the argument into another argument that lies in a small interval. This approachnis often used before the calculating the function according to one of the general method mentioned above.

Let us consider the algorithms of these methods more precisely in order to select among them the best candidates for the implementation in FPGA.

1.2 Polynomial approximation

The polynomial approximation is the representation of an algorithm of the function calculation as a polynomial. A polynomial is an expression constructed from one or more variables and constants using the operations addition, subtraction, multiplication, and raising to the power of integer numbers. Examples of polynomial functions are: $x^3 - 6x^2 + 10$ and $x^3y^2 + 15x^2y^2 - 6x$. The first is a univariate polynomial, while the second is a multivariate polynomial.

The problem of the polynomial approximation has two parts. The first one is the finding out the coefficients, the second one is selection of the effective algorithm and structure for the polynomial calculating.

Three base techniques for computing the coefficients of the approximating polynomials are Taylor approximation, minimax approximation and interpolation.

Taylor approximation gives analytical formulas for the coefficients and the approximation error. It is useful for some algorithms namely the Bipartite, Multipartite, Powering algorithm and functional recurrence.

Minimax approximation is a numerical technique that gives the values of the coefficients and the approximation error numerically. It has the advantage that it gives the lowest polynomial order for the same maximum approximation error [10].

Interpolation is a family of techniques. Some techniques use values of the given function in order to compute the coefficients while others use values of the function and its higher derivatives to compute the coefficients. Interpolation can be useful to reduce the size of the coefficients table at the expense of more complexity and delay and that is by storing the values of the function instead of the coefficients and computing the coefficients in hardware on the fly [13].

Polynomial expressions are computational intensive as they contain a number of additions and multiplications which are expensive operations. These calculations take many clock cycles to compute on a processor. When implemented in an ASIC, or FPGA they occupy a large area and consume a lot of power in addition to increasing clock periods. It is, therefore, imperative to reduce the number of operations in polynomial expressions as much as possible. These reductions can be achieved by factoring these expressions and finding common subexpressions among multiple-polynomial expressions. Unfortunately, not many tools are available to perform this, especially for multiple-variable expressions.

The problem of optimization of polynomial expressions can be stated as follows: given a set of polynomial expressions of any degree and consisting of any number of variables, find an implementation that has the least number of operations (additions, subtractions, and multiplications).

The Horner method is the default method of evaluating Taylor series approximations to trigonometric functions in many libraries such as the GNU

CLibrary [14]. For example, consider the following expression for sin (x) which has been approximated to four terms:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}.$$

Assuming that the terms 1/3!, 1/5!, and 1/7! are precomputed, the naive evaluation of this polynomial representation requires 3 additions/subtractions, 12 variable multiplications, and 3 constant multiplications.

The Horner form of this expression can be written as:

$$\sin(x) = x\left(1 + x^2\left(-\frac{1}{3!} + x^2\left(\frac{1}{5!} - \frac{x^2}{7!}\cdot\right)\right)\right).$$

Most algorithms hand-optimize the resulting Horner form to remove the redundant computations of x^2. The expression is then rewritten as:

$$X = x^2;$$

$$\sin(x) = x\left(1 + X\left(-\frac{1}{3!} + X\left(\frac{1}{5!} - \frac{X}{7!}\cdot\right)\right)\right). \qquad (1.1)$$

The Horner form is a good representation for polynomials with single variables, but does not provide good results for multivariate polynomials. Furthermore, it cannot find common subexpressions automatically to further reduce the number of operations.

Consider the terms 1/3!, 1/5!, and 1/7! are precomputed and denoted as S_3, S_5, and S_7, respectively. Then, the four-term Taylor expansion of sin (x):

$$\begin{aligned} d_1 &= x \cdot x, \\ d_2 &= S_5 - S_7 \cdot d_1, \\ d_3 &= d_2 \cdot d_1 - S_3, \\ d_4 &= d_3 \cdot d_1 + 1, \\ \sin(x) &= x \cdot d_4. \end{aligned} \qquad (1.2)$$

Here, only three additions/subtractions, four variable multiplications, and one constant multiplication are needed.

Traditional optimization methods have been designed for general purpose applications and do not do a good job of optimizing polynomial expressions. Some of the early work in code generation for arithmetic expressions [15, 16]

proposed algorithms to minimize the number of program steps and the number of storage references given a fixed number of registers. In [17] these techniques were extended to handle expressions with common subexpressions. Some work was done to optimize code having arithmetic expressions using factorization techniques [18]. The technique presented in [18] was very limited in that it could only optimize expressions which contained one type of associative and/or commutative operator at a time. As a result it could not optimize general polynomial expressions which have multiplication, addition, and subtraction operations.

In many times the elementary function argument is divided into a set of intervals, and the function is approximated separately on each of them. Then, the small order polynomial is fit for such approximation. A special kind of approximation here is the table based approximation. A set of special algorithms are found for it.

The powering algorithm [19] which is a first order algorithm that employs a table, a multiplier and a special hardware for operand modification. This algorithm can be used for single precision results or as an initial approximation for the functional recurrence algorithms. Table and add algorithms can be considered a polynomial based approximation. These algorithms are first order polynomial approximation in which the multiplication is avoided by using tables. Examples of table add techniques include the work in [20] Bipartite [21,22], Tripartite [23] and Multipartite [24, 25, 26]. Examples of other work in polynomial approximation include [27, 28]. The convergence rate of polynomial approximation algorithms is function-dependent and it also depends to a great extent on the range of the given argument and on the number of the sub-intervals that we employ.

It is noteworthy that computing the polynomial expressions, even in their optimized form, is expensive in terms of hardware, cycle time, and power

consumption. If the arguments to these functions are known beforehand, the functions can be precomputed and stored in lookup tables in memory. However, in cases where these arguments are not known or the memory size is limited, these expressions must be computed during the execution of the application that uses them.

1.3 Functional recurrence algorithms

As it is mentioned above, the functional recurrence algorithm starts by a given initial approximation and it is feeded to a polynomial in order to obtain a better approximation. This process ise repeated a number of times until the desired precision is reached. The prominent example of such algorithm is the Newton's method, hich is a major tool in arbitrary-precision arithmetic.

Suppose that some function f has a zero x if $f(x)= 0$. Then, consider $f(x_0)$ is an initial approximation of this point, and that $f(x)$ has two continuous derivatives in the region of interest. From the Taylor's theorem:

$$f(x) = f(x_0) + (x - x_0) f\,'(x_0) + \frac{(x - x_0)^2}{2} f\,''(x_0),$$

for some point x in an interval including $\{\zeta, x_0\}$. Consider $f(\zeta) = 0$, then we see that

$$x_1 = x_0 - f(x_0)/f\,'(x_0)$$

is an approximation to ζ. If x_0 is sufficiently close to ζ, we have

$$|x_1 - \zeta| \le |x_0 - \zeta|/2 < 1.$$

This motivates the definition of Newton's method as the iteration

$$x_{j+1} = x_j - \frac{f(x_j)}{f\,'(x_j)}, j = 0, 1, \ldots$$

The error of such approximation of x is $e_n = x_n - x$. The fact is, that the error after the next iteration is

$$|e_n+1| \le K|e_n|^2,$$

i.e., the order of the algorithm convergention is 2.

Consider applying Newton's method to the function

$$f(x) = y - x^{-m},$$

where m is a positive integer constant, and y is a positive constant. Since $f'(x) = mx^{-(m+1)}$, the Newton's iteration is simplified to

$$x_{j+1} = x_j + x_j\,(1 - x_j^m\, y)/m. \tag{1.2}$$

This iteration converges to $\zeta = 1/\sqrt[m]{y}$, which is provided by the initial approximation x_0. It is surprising that (1.2) does not involve divisions. In particular, the reciprocal square roots (the case m = 2) can be computed by this method. In this situation, the iteration is obtained:

$$x_{j+1} = x_j + x_j\,(1 - x_j^2 y)/2, \tag{1.3}$$

which converges to $1/\sqrt{y}$ if x_0 is a sufficiently good approximation. From (1.3) the square root function is got as

$$\sqrt{y} = y*(1/\sqrt{y}\,).$$

Here, the method does not involve any divisions. In contrast, if the other the Newton's method is applied to the function $f(x) = x^2 - y$, the Heron's iteration formula is obtained:

$$x_{j+1} = \frac{1}{2}\left(1 + \frac{y}{x_j}\right), \tag{1.4}$$

This requires a division by x_j at iteration j, so it is essentially different from the iteration (1.3) [28].

There are a lot of algorithms of elementary function calculations, which are based on the functional recurrence algorithms. Among them are log x, a^x, and others [10]. The disadvantage of all of them is the computational complexity in the number of multiplications and divisions. However, this figure is proportional to the log n, where n is the argument bit width.

1.4 Digit recurrence algorithms

1.4.1. Introduction

The digit recurrence techniques, or shift-and-add algorithms often are named as the bti-by-bit algorithms because for each iteration, a single exact resulting bit is achieved. This feature goes form the fact that these algorithms are linearly convergent.

Among these algorithms the CORDIC algorithm is the well-known. The CORDIC algorithm was introduced in 1959 by Volder [29]. In Volder's version, CORDIC makes it possible to perform rotations and to multiply or divide numbers using only shift-and-add elementary steps. The results are sine, cosine, and arctangent functions.

In 1971, this algorithm was generalized to compute logarithms, exponentials, and square roots [30]. CORDIC is not the fastest way to perform multiplications or to compute logarithms and exponentials but, since the same algorithm allows the computation of most mathematical functions using very simple basic operations, it is attractive for hardware implementations. CORDIC has been implemented in many pocket calculators and in arithmetic coprocessors such as the Intel 8087 [31].

1.4.2 CORDIC algorithm substantiatiation

The Volder's CORDIC algorithm can be denoted in the C-like language as

$\varphi_0 = \varphi$;

$x_0 = 0{,}607252935$;

$y_0 = 0$;

for(i = 0, i < n, i++) {

if ($\varphi_i \geq 0$)

{ $x_{i+1} = x_i - y_i * 2^{-i}$;

$y_{i+1} = y_i + x_i * 2^{-i}$;

$\varphi_{i+1} = \varphi_i - \text{atan}(2^{-i})$;}

else

{ $x_{i+1} = x_i + y_i * 2^{-i}$;

$y_{i+1} = y_i - x_i * 2^{-i}$;

$\varphi_{i+1} = \varphi_i + \text{atan}(2^{-i})$;}

}

The results are $y_n = \sin \varphi$, $x_n = \cos \varphi$, $\varphi_n = 0$. The terms atan^{2-n} are precomputed and stored in ROM. If

$$|\varphi_0| < \sum_{k=0}^{\infty} \text{atan}\, 2^{-k} = 1.783287\ldots,$$

then

$$\lim_{n\to\infty} \begin{pmatrix} x_n \\ y_n \\ \varphi_n \end{pmatrix} = K \begin{pmatrix} x_0 \cos \varphi_0 - y_0 \sin \varphi_0 \\ x_0 \sin \varphi_0 + y_0 \cos \varphi_0 \\ 0 \end{pmatrix},$$

where the scale factor K is equal to $\prod_{j=1}^{\infty} \sqrt{1 + 2^{-2j}} = 1.64676\ldots$. Therefore, to compute the sine and the cosine of a number φ, the initial data are $\varphi_0 = \varphi$; $x_0 = 0{,}607252935$; $y_0 = 0$, as shown above.

That algorithm is based on the decomposition of $\varphi_0 = \varphi$ on the discrete base $w_k = \text{atan}\ 2^{-k}$, using the nonrestoring algorithm. The nonrestoring algorithm gives a decomposition of φ :

$$\varphi = \sum_{k=0}^{\infty} d_k w_k, \quad , d_k = \pm 1.$$

The basic idea of the rotation mode of CORDIC is to perform a rotation of angle φ as a sequence of elementary rotations of angles $d_k w_k$. The algorithm starts

from (x_0,y_0), and obtains the point (x_{k+1},y_{k+1}) from the point (x_k,y_k) by a rotation of angle $d_k w_k$. This gives:

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} \cos(d_k w_k) - \sin(d_k w_k) \\ \sin(d_k w_k) + \cos(d_k w_k) \end{pmatrix} \begin{pmatrix} x_k \\ y_k \end{pmatrix}.$$

This can be simplified as:

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \cos(w_k) \begin{pmatrix} 1 - d_k 2^{-k} \\ d_k 2^{-k} + 1 \end{pmatrix} \begin{pmatrix} x_k \\ y_k \end{pmatrix}.$$

Since, $\cos(w_k) = 1/\sqrt{1 + 2^{-2k}}$ is stable in each iteration, it is taken into account as the common factor K. Then, the formula can be simplified to

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} 1 - d_k 2^{-k} \\ d_k 2^{-k} + 1 \end{pmatrix} \begin{pmatrix} x_k \\ y_k \end{pmatrix},$$

which is the basic CORDIC step, in the trigonometric type of iteration: it is no longer a rotation of angle w_k, but a similarity, or a “rotation-extension” of angle w_k and factor $1/\cos w_k$.

The choice of d_k can be slightly simplified. If the angles are defined as $\varphi_0 = \varphi$; $\varphi_{i+1} = \varphi_i - d_k w_k$; $d_k = 1$ if $\varphi_i > 0$, and -1 otherwise. So, the algorithm is got, which is mentioned above.

The feature of the algorithm is that it performs only shifts (multiplies by 2^{-k}) and additions (subtractions) [6].

1.4.3 CORDIC-like algorithms

Similarly, to the described above algorithm, the rest of the CORDIC-like algorithms are got. Below some of them are represented, which are selected in [7,10].

Algorithm for the functions $\varphi = \text{arctg}(y/x)$ and $M = k\sqrt{x^2 + y^2}$ by $-\pi \le \varphi < \pi$, $k = 1.64676025812$.

$\varphi_1 = 0$; $x_0 = x$, $y_0 = y$.

for(i = 0, i < n, i++) {

if ($x_i \ge 0$)

{ $x_{i+1} = x_i - y_i * 2^{-i}$;

$y_{i+1} = y_i + x_i * 2^{-i}$;

$\varphi_{i+1} = \varphi_i - \text{atan}(2^{-i})$;}

else

{ $x_{i+1} = x_i + y_i * 2^{-i}$;

$y_{i+1} = y_i - x_i * 2^{-i}$;

$\varphi_{i+1} = \varphi_i + \text{atan}(2^{-i})$;}

}

The results are $y_n = 0$, $x_n = k\sqrt{x^2 + y^2}$, $\varphi_п = \text{arctg}(y/x)$.

Algorithm for the functions sh φ, ch φ.

$y_0 = y$, $x_0 = 1.2051366$, $\varphi_0 = \varphi$.

$i = 0$; $j = 0$;

while ($i <= n$){

if ($\varphi_i \geq 0$){

$y_{i+1} = y_i + x_i * 2^{-j}$;

$x_{i+1} = x_i + y_i * 2^{-j}$;

$\varphi_{i+1} = \varphi_i - \text{arth}(2^{-j})$;

}

else

{ $y_{i+1} = y_i - x_i * 2^{-j}$;

$x_{i+1} = x_i - y_i * 2^{-j}$;

$\varphi_{i+1} = \varphi_i + \text{arth}(2^{-j})$;

}

if ($i = 4$) $j = 4$;

else if ($i = 13$) $j = 13$;

else j++;

i++;

}

The results are x_n = ch φ, y_n = sh φ, $\varphi_n = 0$.

Algorithm for the functions φ = arth(y/x), $M = k\sqrt{x^2 - y^2}$, k = 0.82978162.

$y_0 = y$, $x_0 = x$, $\varphi_0 = 0$.

i =0; j = 0;

while (i <=n){

if ($\varphi_i \geq 0$){

$y_{i+1} = y_i - x_i * 2^{-j}$;

$x_{i+1} = x_i - y_i * 2^{-j}$;

$\varphi_{i+1} = \varphi_i + \text{arth}(2^{-j})$;

}

else

{ $y_{i+1} = y_i + x_i * 2^{-j}$;

$x_{i+1} = x_i + y_i * 2^{-j}$;

$\varphi_{i+1} = \varphi_i - \text{arth}(2^{-j})$;

}

if (i = 4) j = 4;

else if (i = 13) j =13;

else j++;

i++;

}

The results are $x_n = k\sqrt{x^2 - y^2}$, φ_n = arth(y/x).

Algorithm for the function y = e^x. $0 \leq x < 1$.

$y_0 = 1$, $x_0 = x$.

i =0; j = 0;

while (i <=n){

if ($x_i \geq 0$){

$y_{i+1} = y_i + x_i * 2^{-j}$;

$x_{i+1} = x_i - \ln(1 + 2^{-j})$;

}

else{ $y_{i+1} = y_i - x_i * 2^{-j}$;

$x_{i+1} = x_i + \ln(1 - 2^{-j})$;

}

if ($i = 4$) $j = 4$;

else if ($i = 13$) $j = 13$;

else j++;

i++;

}

The results are $y_n = e^x$, $x_n = 0$.

Algorithm for the function y = ln(x). $0 \le x < 1$.

$y_0 = 0$, $x_0 = x$.

$i = 0$; $j = 0$;

while ($i <= n$){

if ($1 - x_i < 0$){

$x_{i+1} = x_i + x_i * 2^{-j}$;

$y_{i+1} = y_i - \ln(1 + 2^{-j})$;

}

else{

$x_{i+1} = x_i - x_i * 2^{-j}$;

$y_{i+1} = y_i + \ln(1 + 2^{-j})$;

}

if ($i = 4$) $j = 4$;

else if ($i = 13$) $j = 13$;

else j++;

i++;

}

The results are $y_n = e^x$, $x_n = 0$.

Algorithm for the function 2^x by the Brigg's method.

$x_0 = x$.

for(i = 1, i <= n, i++) {

if ($x_i < \log_2(1 + 2^{-(i+1)})$)

{ $x_{i+1} = x_i$;

$a_{i+1} = 0$; }

else

{ $x_{i+1} = x_i - \log_2(1 + 2^{-i})$;

$a_{i+1} = 1$; }

}

$y_0 = 1$;

for(i = 1, i <= n, i++) {

if ($a_i = 1$) $y_{i+1} = y_i*(1 + 2^{-i})$;

}

The result is $y_n = 2^x$.

1.4.4 Square root algorithm

The well-known CORDIC algorithm of the $\sqrt{x}$ calculations consists in the following. It calculates the function atanh(x/y) as it is shown above. But the side result is the function $K\sqrt{x^2 - y^2}$, and by the substitution $x = A + 0.25$, $y = A - 0.25$, we get $x_n = K\sqrt{A}$ [32].

The disadvantages of this algorithm are additional multiplication to the coefficient $1/K \approx 1.207$, and repeating some iterations (4-th and 13-th when $n < 32$) for the algorithm convergence.

1.5 Hardware implementation of the elementary functions

As it is shown above, both the polynomial expressions and rational approximations are computational intensive as they contain a number of additions, multiplications, and even divisions, which are expensive operations. When implemented in an ASIC, or FPGA they occupy a large area and consume a lot of power in addition to increasing clock periods.

When the function argument is divided into a set of intervals, and the small order polynomial is fit for such approximation, then, such is approximation often used in hardware [33]. A special kind of approximation here is the table based approximation [34].

The CORDIC algorithms have got the most intensive use in the FPGA implementation due to their simplicity. The problems and solutions of these algorithm implementations are shown in the popular work [35].

1.6 Preliminary conclusions

In this section, the algorithms for the elementary function calculation are reviewed. Among them are polynomial approximation, functional recurrence, and digit recurrence algorithms.

It is found out that the hardware implementation of the elementary function computations is not investigated at the proper level.

It is noted, that the algorithms, which utilize only additions, shifts, table functions, and small number of multiplications are the best candidats for the FPGA implementations. Among them the CORDIC like algorithms play the leading role.

In the next section, the theoretical basics of the new methods are developed, which satisfy the mentioned above features.

2 DESIGN OF THE PROCESSING UNITS FOR THE ELEMENTARY FUNCTION CALCULATION

2.1 FPGA as the computing environment for elementary functions

2.1.1 FPGA architecture

Below, the properties of the Xilinx FPGAs are considered, because this company is considered as the larger FPGA supplier. But the proposed reasons are true for FPGAs of other companies as well.

In Xilinx FPGAs, the basic building blocks are Configurable Logic Blocks (CLBs). In Spartan-6 devices, the CLBs are made up of two logic slices which are independently connected to the general routing on the FPGA and to a carry chain structure [36]. There are two types of logic slices in Spartan-6, SLICEL and SLICEM. SLICEL can be seen as the basic logic slice type, and contains four 6-input look-up-tables (LUTs), together with four D-type flip-flops(DFFs) and multiplexers for routing purposes. The LUTs can implement any 6-input logic function. SLICEM slices contain shift register functionality and provide the option of using the LUTs as distributed user RAM, as well as the basic resources described for SLICEL slices. When used as distributed RAM, LUTs are configured as memories for user data storage.

Other resources on the FPGA include Digital Clock Managers (DCM), Phase-Locked Loops (PLL), Block RAMs, DSP blocks, I/O blocks (IOBs) and buffers for connecting package pins. The FPGA resources are connected together by a configurable routing matrix. A common way of describing FPGAs is as configurable logic “islands” connected together by a “sea” of configurable routing paths.

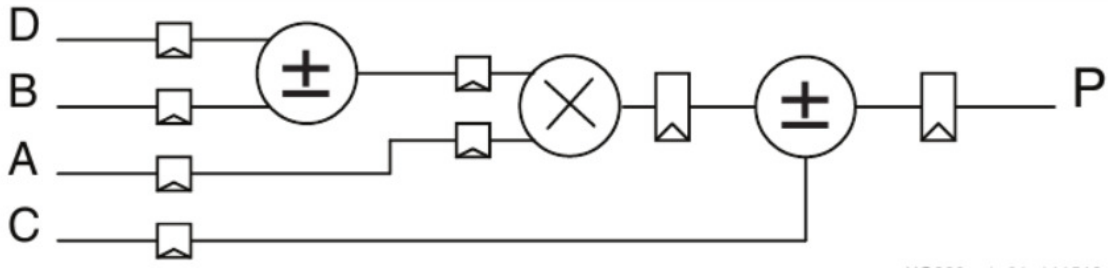
When synthesising an FPGA design, the circuit function defined by the designer is mapped to these resources by synthesis tools. This mapping makes up

the configuration of the device, and is stored in the SRAM-based configuration memory.

The configuration memory defines the function and operation of all the described resources as well as the routing and connections on the FPGA, and can be seen as an underlying device definition layer.

SRAM-based FPGAs are programmed using a binary bit-stream, usually stored offchip. For space applications, this off-chip configuration storage is usually in the form of EEPROM or Flash. Since the SRAM-based configuration memory is volatile, the bit stream has to be reprogrammed onto the FPGA on startup and power-cycling. The programming logic is responsible for writing the configuration memory via one of the configuration interfaces.

Xilinx Spartan-6 FPGAs contain dedicated DSP circuitry, in the form of DSP48A slices. Fig. 2.1 shows a simplified view of a DSP48A slice, featuring a 25x18 multiplier, internal pipelining registers and an arithmetic unit. DSP blocks are hard ASIC blocks embedded in the FPGAs array of programmable logic, and are much more area efficient compared to soft logic implementations of the same functionality [37]. As such, DSP blocks are not defined by an underlying configuration layer. The DSP48A is well suited for common DSP operations such as multiply-accumulate.



.

Fig.2.1. Simplified view of a DSP48A slice

The configuration vectors can be synthesised as constants or as signals originating from other parts of the system. DSP slices are arranged on the FPGA so that they can be cascaded through the use of fixed carry and shift lines to create wider operators than what would fit into a single DSP slice.

Block RAM, or BRAM, in Spartan-6 are made up of 36 kB SRAM memory blocks. These blocks can be cascaded and divided into a number of different configurations. For example, a single 36kB block can be used as a 36kx1 RAM, or as two functionally separate 18kx1 RAMs. It is also possible to create wider or larger RAM blocks by cascading BRAMs together.

So, when choosing an elementary function algorithm, one should keep in mind the features of an FPGA structure that has CLB resources, multipliers, adders, multiplication blocks, but does not have divisions. For its rapid execution, the elementary function should be implemented as a parallel structure that allows the pipelinined operations, because this mode is effectively supported in FPGA.

2.1.2 FPGA project optimization critera

Mentioned above FPGA resources are valuable. Different projects for FPGA, which perform the same task, can be distinguished in different folume of these resources. Moreover, these projects can be of different throughput. To select properly the best project, the effective effectiveness criteria must be selected. Below, some considerations to these criteria selection are considered.

Hardware volume criterium

In advance, we consider, that the processing unit bit width is equal to n, and its hardware is proportional to n in some limitations, and by other equal conditions.

The adder is the main operational unit in FPGA project. Usually, one bit of the adder is implemented in a single LUT, not taking into account the proper carry propagation network. Besides, each LUT output can be stored to the

respective register (trigger), as in is shown in Fig. 2.2,a. Thus, the *n*-bit adder, and the *n*-bit register have the same complexity, or cost. Then, such register, and adder have the relative cost, which is equal to a 1.

Also it is important to consider that LUT has the mode SRL16, in which it operates as a shift register with the programmable length of 1 to 16 bits (Fig.2.2,b).

In the FPGA chip one DSP48 unit takes 60–300 CLB slices, averagely, 160 CLB slices. For reference, the hardwired 18x18 bit multiplier is implemented as an equivalent circuit of 208 CLB slices. Consider a DSP processor configured in FPGA with the hardware resources being used effectively. Then all multiplier resources should be loaded by the useful computations, and other computations are distributed among all adders and multiplexers implemented in FPGA. By this condition, one multiplier takes 160 CLB slices. These CLBs are enough to implement up to 20 adders and 20 registers of the same bit width. Thus, the complexity of the multiplier unit is estimated as the complexity of 20 adders. Similarly, the complexity of the Distributed RAM can be estimated.

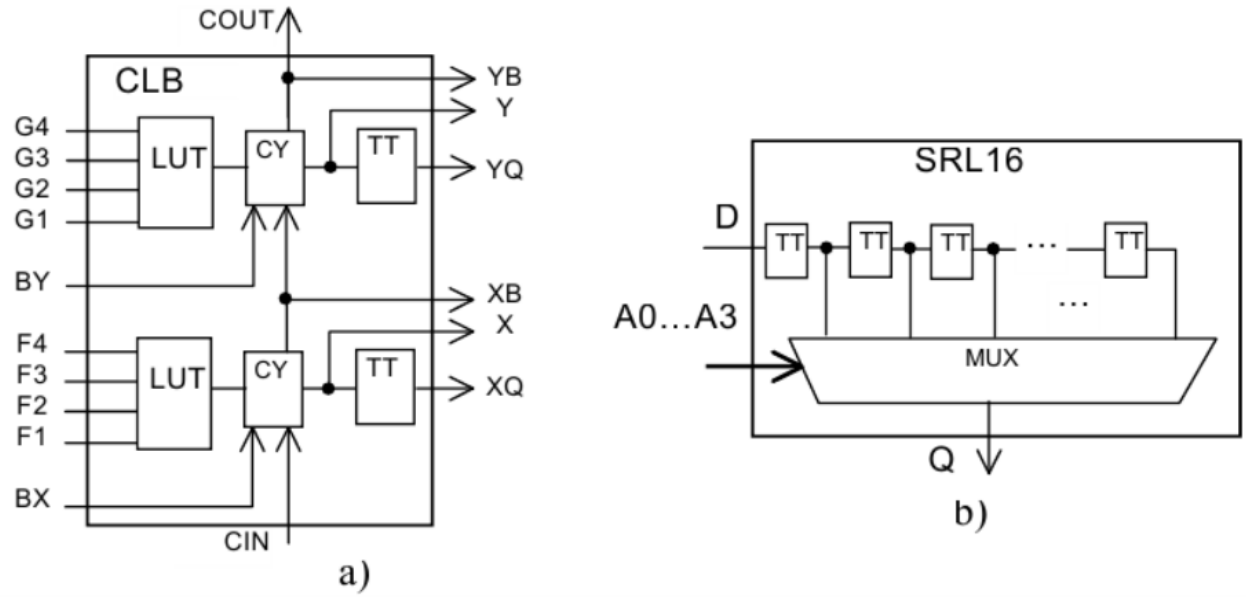


Fig. 2.2. Structure of the Xilinx FPGA elements: CLBS (a), SRL16 (b)

Table 2.1 shows the complexity of the different elements of the same bit width configured in FPGA, which is expressed in the complexity of a single register.

Table 2.1.

Complexity of elements, configured in FPGA

Type	Complexity
Register	1
Adder	1
Adder-subtractor	1
2-input multiplexor	1
3,4 -input multiplexor	2
5,6-input multiplexor	3
7,8-input multiplexor	4
Registered delay to 2-16 registers (FIFO)	1−2
Multiplier unit	20
16 word RAM	1
1024 word RAM	20

Its analysis shows, that multiplying units should be minimized primarly. Since in the actual application specific processors the 2–5 input multiplexers frequently are used, then the complexity of the multiplexer, which takes to a single input, is equal approximately to 0.27. This means that it is necessary to mimimize not only the number of registers and adders, but also number of multiplexot inputs.

According to the arguments above, the following complexity criterion of the FPGA project is proposed:

$$Q_S = n_R + n_A + 20n_M + 0.27n_x, \quad (2.1)$$

Where n_R is the register number, including the FIFO number, which are mapped into SRL16 primitive, excluding the registers in the DSP48 modules;

n_A is the adder number, due to the CLB construction, up to three input adder is implemented in a single CLB column, therefore, n_A considers 2- or 3-input adders;

n_M is the multiply unit number;

n_x is the number of the multiplexor inputs [38].

Performance criterion

The signal delay in the multiplier blocks is approximately equal to 4.5 ns for Spartan-6 FPGA. In the two-staged pipelined multiplier the minimum multiplication period is equal to 2–2.5 ns. The adder delay is derived from the carry signal propagation and therefore, it is proportional to the bit width. Since the adder is formed as a line of the locally coupled DLB slices, then its delay is stable, and for 16-bit adder is equal to 1.4–2.5 ns.

It has to taken into considerations, that the proportion of the delay in the logic elements is 35–85% of the clock period depending on the degree of the placing and routing optimization, and on the complexity of the structure.

In the practice, the multiplier delay is about twice te adder delay, taking into account the interconnection delays.

The multiplexer network has far less latency then the adder has. It is not depended on the word length, and is nearly independed on the input number., but depends on the quality of the wiring of the lines, which connect it to the neighboring elements. As a result, the connection of the additional multiplexor to the adder adds a delay of 0.4–1.6 ns depending on the multiplexor number (1 or 2) and routing quality.

Thus, the proposed performance criterion is:

$$Q_{\mathrm{T}} = n'_A + c_{TM}\, n'_M + c_{TX}\, n'_x, \qquad (2.2)$$

where c_{TM}, c_{TX} are the ratios of the multiplier and multiplexor delay to the adder delay, $c_{TM} = 2.2$, $c_{TX} = 0.5$;

n'_A is the adder number;

n'_M is the number multipliers;

n'_x is the number of multiplexers,

staying in the critical path, which connects the output of one register and the input of another one. Here, a single unit delay is estimated as the delay of the adder with the average delays in the communication lines.

Really, Q_{T} is equal to the minimum clock period, derived for the current placed and routed project, when the results are outputted in each clock cycle. It is hold on when the processing unit is implemented as a whole combinational network, which performs the elementary function, or if it is wholly pipelined network.

The real processing unit projects can calculate the algorithm for $L > 1$ clock cycles not in the pipelined mode. Thus, the expression (2.2) must be multiplied by the value of L:

$$Q_{\mathrm{T}} = L\,(n'_A + c_{TM}\, n'_M + c_{TX}\, n'_{x.}). \qquad (2.3)$$

The integral criterium has to take into account both hardware volume and performance criteria. Then, it can be selected as:

$$Q = Q_S \cdot Q_{\mathrm{T}} \qquad (2.4)$$

This criterium shows, how many adders are needed to calculate, say, one million of results per second. The better solution has the smaller value of Q, because it has smaller hardware volume and/or higher clock frequency, which is proportional to the processor performance.

2.2 Synchronous dataflow graph for the elementary function calculations

The processing module for the elementary function calculation belongs to the datapaths. The modern high-performance computers operate with high clock frequencies, thanks to the pipelined mode of data processing and transmission. There are various methods for the design and optimization of the pipelined datapaths. These methods are based on the structural synthesis of the datapath, describing it at the register transfer level and further conversion to the gate level. The basis of many methods is a representation of the algorithm as a synchronous dataflow graph (SDF) and its transformation [39].

Such SDF optimization techniques as retiming, folding, unfolding and pipelining, are widely used in microelectronics, and design of digital signal processing (DSP) devices [40].

SDF is isomorphic to the graph of the computer structure, which performs a predetermined algorithm. The nodes of such a graph correspond to the computing resources like adders, multipliers, processing units (PUs). The edges correspond to the communication lines, and the labels on them are mapped to the registers. Consequently, SDF is a directed graph $G = (V, E)$, representing the computer structure, where $v \in V$ represent some logic network with delay of d time units. The edge $e \in E$ corresponds to a link and is loaded by $w[e]$ labels, which is equal to the depth of the FIFO buffer.

The minimum duration of the clock cycle T_C is equal to the maximum delay of the signal from one register output to the input of another register, i.e., to the critical path through the adjacent nodes with delays d, for which $w[e] = 0$. It should be noted, that with such a one-to-one mapping of SDF, the duration of the algorithm cycle T_A coincides with the duration of a clock period, i.e., $T_A = T_C$, that in the other algorithm mapping is not respected.

The retiming is such a exchange of the labels in SDF edges, which does not affect the algorithm results. Usually it is realized as a sequence of elementary retimings, each of them consists of a transferring a group of labels (i.e., registers) from the input edges of some node *v* to its outputs.

In most cases, it is allowed to increase the latent delay of the algorithm and to insert the additional registers on the inputs or outputs of SDF. After retiming such modified SDF, the pipelined network with low value of T_C is achieved. This technique is called as *SDF pipelining*.

A *cut-set retiming* is an effective metod, which implements the pipelining, and therefore, is widely used for the pipelined datapath design. The *cut-set* in an SFG is a minimal set of edges, which partitions the SFG into two parts. The procedure is based upon two simple rules [1].

Rule 1: *Delay scaling*. All delays *D* presented on the edges of an original SFG may be scaled, i.e., $D' \rightarrow \alpha D$, by a single positive integer α, which is also known as the pipelining period of the SFG. Correspondingly, the input and output rates also have to be scaled by a factor of α (with respect to the new time unit *D'*). Time scaling does not alter the overall timing of the SFG.

Rule 2: *Delay transfer*. Given any cut-set of the SFG, which partitions the graph into two components, we can group the edges of the cut-set into inbound and outbound, depending upon the direction assigned to the edges. The delay transfer rule states that a number of delay registers, say *k*, may be transferred from outbound to inbound edges, or vice versa, without affecting the global system timing.

These rules provide a method of systematically adding, removing and distributing delays in a SFG and therefore adding, removing and distributing registers throughout a circuit, without changing the function. The cut-set retiming procedure is then employed, to cause sufficient delays to appear on the

appropriate SFG edges, so that a number of delays can be removed from the graph edges and incorporated into the processing blocks, in order to model pipelining within the processors; if the delays are left on the edges, then this represents pipelining between the processors.

SDF has the properties that it can be described by VHDL, and then, be translated into the FPGA bit stream [38].

2.3 Example of the processing module synthesis

Consider the design of the processing module, which implements the equations (1.2). The initial SDF is illustrated by the Fig.2.3,a. After implementing a set of cut-set retimings, the SDF becomes balanced, as in Fig.2.3,b, where the black bars represent the delay marks.

The balanced SDF is acyclic SDF, in each route of it the same number of delay marks stays. Each delay mark is mapped to a single pipeline register. So, the balanced SDF can be described directly in VHDL as follows.

```
process(CLK) begin
    if RISING_EDGE(CLK) then
        if RESET ='1' then
            d11<=0; d12<= 0; d13<= 0; d14<= 0;
            d15<= 0; d16<= 0; d17<= 0;
            d1s7<= 0; d1d2<= 0;d1d3<= 0;
            d2<= 0; d3<= 0; d4<= 0; y<=0; xd<=0;
        else
            xd <= X;
            d11<= xd*xd;
            d12<= d11; d13<= d12; d14<= d13;
            d15<= d14; d16<= d15; d17<= d16;
            d1s7<= d11*S7;
            d2<= d1s7 + S5;
```

```
                    d1d2<= d13*d2;
                    d3<= d1d2 + S3;
                    d1d3<= d15*d3;
                    d4<= d1d3 + S1;
                    y<= d17*d4;
                end if;
            end if;
        end process;
```

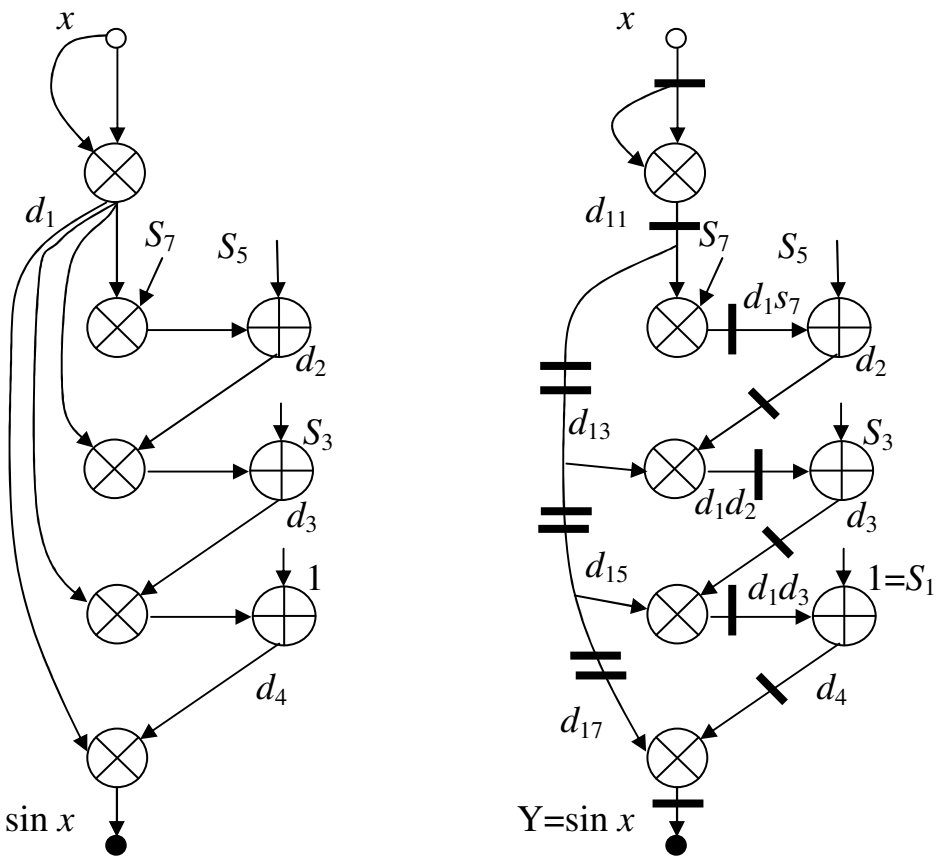


Fig.2.3. SDF for equations (1.2) (a), and SDF after pipelining (b)

Here, di means the signal, which is delayed to i clock cycles. All the signals and constants except clock signal CLK and reset signal RESET are considered to be integers, which have scaled properly. Due to the balanced SDF, the derived processing unit operates in the pipelined mode. Its critical path goes only through a single multiplier unit. Therefore, according to (2.2) its performance is $Q_T = 2.2$. The hardware volume (2.1) is $Q_S = 8 + 3 + 20 \cdot 5 = 111$, taking into account that the registers d11, d1s7, d1d2, d1d3, y are considered as the registers of the DSP48 modules, couples of adjacent registers are implemented in SRL16 units.

The resulting criterium (2.4) is $Q = Q_S \cdot Q_T = 111 \cdot 2{,}2 = 244{,}2$ adders per bln. results per second. This figure is rather high, and the most fraction in it (90%) is the multiplier costs. This proves the fact that the polynomial approximation is bad solution for the elementary function approximation.

2.4. Development of the square root computing module

2.4.1 Introduction

The function of the square root is the very popular elementary function in the science computations, DSP, and image processing, and pattern recognition [1,41]. Most often it is computed in a floating-point coprocessor, which has a certain delay. But the common low-cost microprocessors do not have such coprocessors.

In our time, FPGAs are used to solve the same problems, which require the use of the function $\sqrt{x}$. There are IP cores for the function $\sqrt{x}$, which are offered by FPGA manufacturers, and other firms that supply the licenses to such modules for their configuration in FPGAs [42]. Such a module is able to calculate the function of the square root in hardware in a pipelined mode with high speed.

These modules have been developed one to two decades ago, and generally, they do not take into account the features of new FPGAs that appeared on the market a few years ago. So, such modules need to be improved.

Next, we will consider the square root extraction algorithms with an evaluation of their efficiency for 24-bit input data and fixed-point results that can be claimed for implementation in the FPGA. This level is acceptable for most signal processing algorithms and for the implementation of floating point calculations of single accuracy.

2.4.2 Base algorithm selection

Polynomial approximation

The traditional solution for calculating an elementary function is a polynomial calculation, which is, for example, a Taylor series, as the next [43]:

$$\sqrt{1+x} = 1 + \frac{1}{2}x - \frac{1}{8}x^2 + \frac{1}{16}x^3 - .$$

It is impossible to achieve a calculation error less than 0,2% if $x \in (0; 1)$. In addition, the algorithm requires the implementation of many multiples. Therefore, it is inappropriate for implementation in the FPGA, though, it may be agreed on a piecewise polynomial approximation.

Functional recurrence algorithm

The following iterative algorithm is based on the Newton-Raffson formula (1.3), which does not require dividing operations. Here $x_0 \approx 1/\sqrt{y}$ is the approximate value of the function, $\sqrt{y} \approx xy_n$,. Each subsequent iteration of the algorithm approximately doubles the number of correct result bits. Therefore, in order to calculate the correct 24-bit result, it is necessary to perform $n = 2$ iteration of the algorithm and obtain the value of x_0 from the table with a seven-digit input of the address, that is, volume 2^7. The algorithm can be executed in one iteration, if the table has a 13-bit input, that is, it has a volume of 2^{13} words.

The performance Q_T and hardware Q_S costs of this algorithm an previous one are given in Table 2.2. When calculating Q_S, it was considered that the mentioned tables are implemented in the FPGA as a ROM, which has an approximate complexity as the complexity of two and sixty adders, respectively.

Digit recurrence algorithm

A well-known CORDIC algorithm for calculating $\sqrt{x}$ is based on the following. In the calculation of the arctgh(x/y) function, the $\sqrt{x}$ function is the by-result of the function $x_n = K\sqrt{x^2 - y^2}$, with substitution $x = A + 1$, $y = A - 1$, we obtain $x_n = K\sqrt{A}$ [44,45]. This algorithm has been successfully implemented in many FPGA projects, such as in [46].

The disadvantages of this algorithm are the need for additional multiplication by the factor $1/K \approx 1{,}204$, as well as the repetition of some iterations for the convergence of the algorithm.

A more constructive algorithm is the Digit recurrence algorithm, which aims to obtain the function x [44,47]. It is based on the following relations. For each number $x \in [0{,}25;\ 1.0]$ we can choose the following coefficients $a_i \in [0,\ 1]$ that

$$\prod_{i=1}^{\infty}\left(1 + a_i 2^{-i}\right)^2 = 1.0. \qquad (2.5)$$

Therefore,

$$1/\sqrt{x} \approx \prod_{i=1}^{m}\left(1 + a_i 2^{-i}\right)$$

or

$$\sqrt{x} \approx x\prod_{i=1}^{m}\left(1 + a_i 2^{-i}\right). \qquad (2.6)$$

The implementation of the algorithm consists in repeating a series of iterations. During the m-th iteration, the coefficient a_m is chosen to ensure equality

(2.5) and the found coefficient is substituted in (2.6). In order to handle the numbers $x \in [0; 1.0)$, they can be normalized if (2.6) and (2.7) initially accept $i = 0$ and $a_i = 1$ until the first overflow of the product in (2.5). As a result, we get the following algorithm [44].

```
y0 = x;  x0 = x; m = 0; f = 0;
for (i = 0; i < n; i++)
{
    t  = xi + 2^-m*xi;
    u  = t + 2^-m*t  ;
    if   (u ≥ 1.0)  {
          f = 1;
          xi+1 = xi ;
          yi+1 = yi ;
        }
    else {
          xi+1 = u;
          yi+1 = yi  + 2^-m*yi;
        }
    if (f == 1) m++;
}
```

When performing the algorithm initially, when $m = 0$, the normalization of the operand x_i is performed with the correction of the partial result y_i. Then $m = 1, 2,..., n$ and in the process of convergence, x_i goes to one, and y_i goes to $\sqrt{x}$, where n is the number of binary digits of the result.

To implement the algorithm in FPGA, it is desirable to perform the normalization of x_0 and the corresponding correction y_n in the normalization block based on the shift unit.

Table 2.2

Costs to calculate the function $\sqrt{x}$

Algorithm	DSP48 modules	Q_S	Q_T
Polynomial algorithm	5	111	8
Functional recurrence algorithm, 1 iteration	2	102	7
Functional recurrence algorithm, 2 iterations	4	86	13
Digit recurrence algorithm	–	52	50
Modified digit recurrence algorithm	1	35	17

Then, the algorithm receives an acceleration in the worst case by one third. The experience of building a normalization unit shows that its complexity, together with the complexity of the denormalization block for 24-bit data, is evaluated as the complexity of four adders. In addition, $2n$ adders for the parallel calculation (2.5) and (2.6). Then the algorithm is executed for $2n = 48$ clock cycles for obtaining the resulting digits (two cycles of calculating t and u for n cycles) and two cycles for normalization and denormalization. Thus, the algorithm has the complexity of $Q_S = 52$ and $Q_T = 50$ (in the non-pipelined mode).

So, the digit recurrence algorithm for calculating $\sqrt{x}$ is preferable for its FPGA implementation.

2.4.3 Modernization of the digit recurrence algorithm

The largest delay in the digit recurrence algorithm, discussed above, gives a double addition of a shifted datum that distinguishes this algorithm from other algorithms of this type:

$$t = x_i + 2^{-m} x_i;$$

$$u = t + 2^{-m} t.$$

These two steps of addition can be reduced to one:

$$u = x_i + 2^{-m} x_i + 2^{-m}(x_i + 2^{-m} x_i) = x_i + 2^{-m+1} x_i + 2^{-2m} x_i.$$

Since in modern FPGA the three-input adder is implemented in a single layer of six-input LUTs, then such calculation can be performed in one cycle without additional time and hardware costs. Considering this feature, for even *n* the algorithm looks like the following.

```
k = FLO(x);
y0 = SHR(x,k/2);
x0 = SHR(x,k/2*2);
m = 1;
for (i = 0; i < n; i++)
{
    u = xi + 2^(-m+1)*xi + 2^(-2m)*xi;
    if (u ≥ 1.0) {
        xi+1 = xi ;
        yi+1 = yi ;
      }
    else {
        xi+1 = u;
        yi+1 = yi + 2^(-m)*yi;
      }
    m++;
}
Y = SHL(yn,k/2);
```

Here, the FLO function determines the number of digits before the most significant bit, and the SHL, and SHR functions perform a shift the data to the left

and to the right for a given number of bits. Consequently, the number of equivalent adders for this algorithm is the same, but the delay of calculations decreases to $Q_T = 26$ cycles.

When analyzing the execution of this algorithm, it can be seen that when reaching i the limit $n/2$, then the most significant $i - 1$ bits of the data x_i become equal to a one for any x_0. Consequently, the most significant bits of y_i are the exact bits of the result. One can put forward the hypothesis that the least significant bits of the result can be calculated by analyzing and processing the difference $1 - x_i$. For example, this could be determined using the table function.

Let $\varepsilon_1 = 1 - x_i$ and $\varepsilon_x = \sqrt{x} - y_i$ or $\sqrt{x} = \varepsilon_x + y_i$. That is, in order to obtain the refined value of the result, the value of the correction ε_x should be calculated and added to the approximate result, and the correction should be calculated taking into account the difference ε_1.

Due to (2.5) and (2.6),

$$\varepsilon_1 = 1 - x\prod_{i=1}^{m}\left(1 + a_i 2^{-i}\right)^2,$$

$$\varepsilon_x = \sqrt{x} - x\prod_{i=1}^{m}\left(1 + a_i 2^{-i}\right).$$

Let $z = \sqrt{x}\prod_{i=1}^{m}(1 + a_i 2^{-i})$, then

$$\varepsilon_1 = 1 - z^2 = (1 + z)(1 - z);$$

and $\varepsilon_x = \sqrt{x}\,(1 - z)$.

Since $z \approx 1$, then $\varepsilon_1 \approx 2(1 - z)$;

And $\varepsilon_x \approx \sqrt{x}\,\varepsilon_1/2 \approx y_i\,(1 - x_i)/2$.

So, in order to obtain a refined result, $y_i\,(1 - x_i)/2$ should be added to the approximate result y_i. To do this, you need to perform an additional subtraction

and one multiplication. Moreover, because of the difference in ε_1 and the corrections ε_x half of the highest bits are zero, then multiplication can be performed at twice the smaller bit. That is, the hardware complexity of such multiplication can be estimated by five adders. The resulting modified algorithm looks like the following.

```
k = FLO(x);
y0 = SHR(x,k/2);  x0 = SHR(x,k/2*2);
for (i = 0; i < n/2; i++)
{
      u  = xi + 2^-i*xi + 2^-2i-2*xi;
      if   (u ≥ 1.0)  {
            xi+1 = xi ;
            yi+1 = yi ;
          }
      else {
            xi+1 = u;
            yi+1 = yi  + 2^-i-1*yi;
          }
 }
y = yi+1 + yi+1*(1.0 - xi+1)/2;
y  = SHL(yn, k/2);
```

Thus, the costs for this algorithm for $n = 24$ are $Q_S = 35$ and $Q_T = 17$. Thus, due to the modification, the algorithm received an acceleration about 50/17 ≈ 3 times and has a minimal latent delay among all considered algorithms.

SDF of a single iteration of this algorithm is shown in Fig. 2.4.

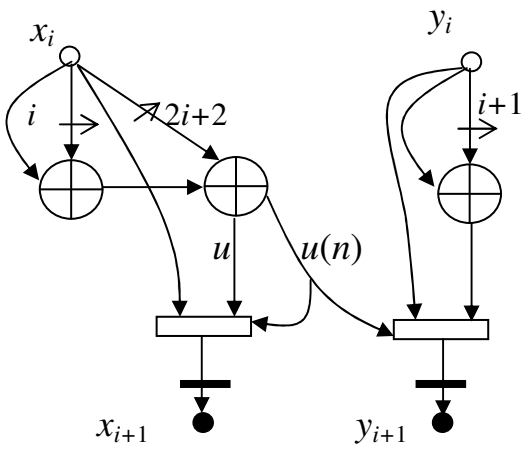


Fig.2.4. SDF of a single iteration of the $\sqrt{x}$ calculating

The arrow "→" in it means arithmetical shift right to the given bit number of the data in the respective edge, the white bar represents a multiplexor, which throughputs left or right edge data depending on the Boolean operand, which enters the multiplexor side. Here, this Boolean operand is the sign bit $u(n)$ of the intermediate result u.

This SDF is the base for the IP core description in VHDL, Which is shown in Appendix. The development and investigation of this IP core are shown in [4,5].

As a result, the modernized digit recurrence algorithm is the best of considered algorithms for the function $\sqrt{x}$ calculating for implementing in FPGA.

2.5 Method of the multifunction processor module design

2.5.1 Background of the method

A set of algorithms of calculating the elementary function are considered above. Among them, the digit recurrence algorithms have the features of the minimum hardware volume for their FPGA implementation. And really, such algorithms are often implemented in FPGA. But they usually implemented as a single function in the separate IP core.

The multifunction processing modules are often needed for design of complex computer systems. Such processing module serves as the mathematical coprocessor for the general purpose microprocessor, is used for implementing complex algorithms of the parallel-sequential nature.

But the multifunction processor modules are not proposed by the providers. Some experimental multifunction processors are found very rarely. The polynomial approximation fits the most of elementary function calculation because the processor structure remains the same, but only the coefficient set is exchanged. But as it is shown above, the hardware volume of such processor is too high.

The most of multifunction processors for the FPGA implementation are based on the CORDIC algorithm [48] because they utilize the similarity of the equations for the different functions [35]. For example, to calculate the functions like sin, cos, atan, sinh, cosh, $\sqrt{x^2 - y^2}$, $\sqrt{x^2 + y^2}$ the same structure is used, but only the control of signs of adders is exchanged.

The traditional method of the multifunction processor design consists in selection of the set of hardware resources, finding out the schedules for each algorithm, and in forming the structure, which implements each of given algorithms in a sequence [49]. But the resulting structures can be far from the optimum because each of the steps of tsuch structure synthesis has different criteria.

Therfore, it is valuable to develop a method for the multifunction processor designing.

2.5.2 SDF of the combined algorithm

In the subsection 2.2 and 2.3 it was shown that SDF is mapped by the one-to-one mapping to the pipelined datapath. So, if SDF represents a set of algorithms, then the respective datapath implements each of the algorithms belonging to this set.

The example of the SDF in Fig. 2.4 shows that SDF can express the algorithm, in which the data streams are dynamically interchanged.

Consider two algorithms, each of them implement the same operation set $\{V\}_1 = \{V\}_2 = \{V\}$, but they are distinguished in the algorithm graphs. Then the combined SDF is possible, which contains the node set $\{V\}$, to some nodes $V_i \in \{V\}$ are connected the multiplexor nodes. So, when these multiplexers are switched in one position, then SDF performs the first algorithm, and when they are swithed in another position, then SDF performs the second algorithm. As a result, such combined SDF is mapped into the multifunction datapath structure, which performs both algorithms.

Definition. *Combined SDF* is SDF, which contains a set of multiplexor nodes, due to that it performs a set of different algorithms.

2.5.3 Formulation of the method

Using the features of the combined SDF a method of the multifunction processor design can be formed. The method is formulated as follows.

The method of the multifunction processor module design consists in forming the combined SDF, which performs a set of algorithms of the elementary function calculation, in balancing this SDF, and in mapping it into the pipelined datapath.

Comparing to other methods, this method is simpler because the steps of resource selection, task scheduling and resource allocation, and structure forming are combined, and it provides better hardware and performance effectiveness.

2.5.4. Example of the multifunctional processor unit design

Consider the design of the multifunctional processing module, which calculates the functions of $\sqrt{x}$, sin *x,* and cos *x.* The first function is calculated using the algorithm, described in the paradraph 2.4.3, and the rest of functions are calculated by the CORDIC algorithm.

SDF of the first algorithm is based on the cycle SDF shown in Fig. 2.4. The respective SDF of the CORDIC cycle is shown in Fig. 2.5, which is built on the base of the algorithm, described in the paragraph 1.4.2.

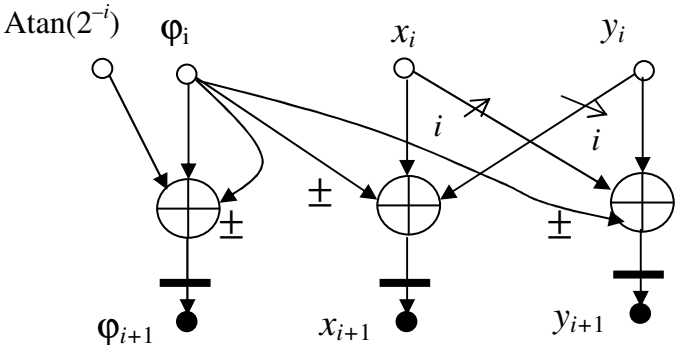


Fig.2.5. SDF of a single iteration of the CORDIC algorithm

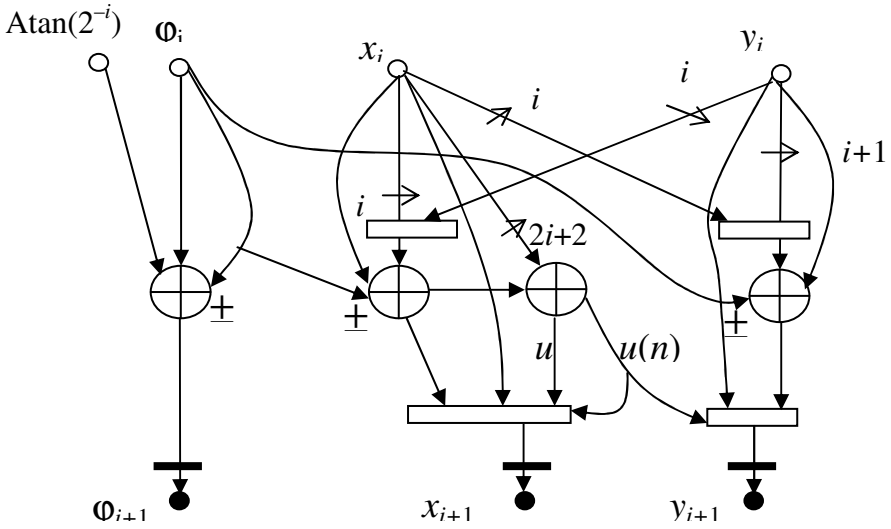


Fig.2.6. SDF of a single iteration of the CORDIC algorithm

After performing the combining the SDFs in Fig 2.4 and 2.5, we have got SDF, which is illustrated by Fig. 2.6.

The resulting combined SDF in Fig.2.6 forms the n-staged pipelined SDF, which is described as a GENERATE operator in VHDL:

```
STAGES: for i in 0 to n-1 generate
        process(CLK)
            variable u:STD_LOGIC_VECTOR(n+1 downto 0);
            variable ii:STD_LOGIC_VECTOR(5 downto 0);
        begin
            ii:=conv_std_logic_vector(i,6);
            if rising_edge(CLK) then
                if F='0' then
                    u:=x(i)+SHR(x(i),ii) +SHR(x(i), (ii&'0')+2);
                    if (u(n) ='0' or u(n+1) ='0') then
                        x(i+1)<= u ;
                        y(i+1)<= y(i) + SHR(y(i), ii+1) ;
                    else
                        x(i+1)<= x(i);
                        y(i+1)<= y(i) ;
                    end if;
                else
                    if (fi(i)(n) = '0') then
                        u  := x(i) - SHR(y(i), ii);
                        y(i+1)<= y(i) + SHR(x(i), ii);
                        fi(i+1)<= fi(i) - atan(i);
                    else
                        u  := x(i) + SHR(y(i), ii);
                        y(i+1)<= y(i) - SHR(x(i), ii);
                        fi(i+1)<= fi(i) + atan(i);
                    end if;
                    x(i+1)<= u;
```

```
                    end if;
                end if;
            end process;
        end generate;
```

The whole IP core description is presented in the Appendix 1.

2.6 Preliminary conclusions

In this section, the FPGA architecture is investigated to select its features, which infer the selection of the elementary function algorithm implementation. This investigation helped to select the hardware cost and performance criteria for the processor module optimization.

It was proven, that the digit recurrence algorithms are best fitted for the FPGA implementation.

A new modification of the digit recurrence algorithm for the function $\sqrt{x}$ calculating is proposed, which provides the decreasing the latent delay up to three times.

The method of SDF mapping into the pipelined structure of the processor module was studied, which helps to derive the effective structures for the elementary function calculations.

A new method of the multifunction processor module design is proposed, which consists in forming the combined SDF, which performs a set of algorithms of the elementary function calculation, in balancing this SDF, and in mapping it into the pipelined datapath, which is simpler and provides better hardware and performance effectiveness comparing to the other similar methods.

The method was used in the design of the processor module for the $\sqrt{x}$, sine, and cosine function calculations.

The effectiveness of the proposed method and algorithms is proven in the next section.

3 IMPLEMENTATION OF THE ELEMENTARY FUNCTION PROCESSOR MODULES IN FPGA

3.1 Synthesis of the processor module for the $\sqrt{x}$ function calculation

The project o the processor module for the $\sqrt{x}$ function calculation is described in VHDL as the entity SQRT_C5, and implements the algorithm, described in the paragraph 2.4.3. Its text is shown in Appendix 1.

The module is tunable by the generic constants:

```
generic(ni:natural:=24; -- input bit width
        no:natural:=24; -- output bit width
        norm:natural:=0;  --0- unnormalized input data, 1 - normalized
        pipe:natural:=1);--  1 –fully pipelined , 0 – combinatorial network
```

The module has the following ports:

```
port(
    CLK : in STD_LOGIC;
    DI : in STD_LOGIC_VECTOR(ni-1 downto 0);  --initial data
    DO : out STD_LOGIC_VECTOR(no-1 downto 0)-- result
);
```

By testing this IP core, the signal of the linear form was feeded its inpit port, and the output signal was investigated. The output signal represents the function $\sqrt{x}$ with the error, which is not sucseed a single least significant bit. For the purposes to preserve such precision, the IP core has the inner data bit width, which is to 5 bits higher than the input data bit width. The resulting modeled diagrams are shown in Fig.3.1.

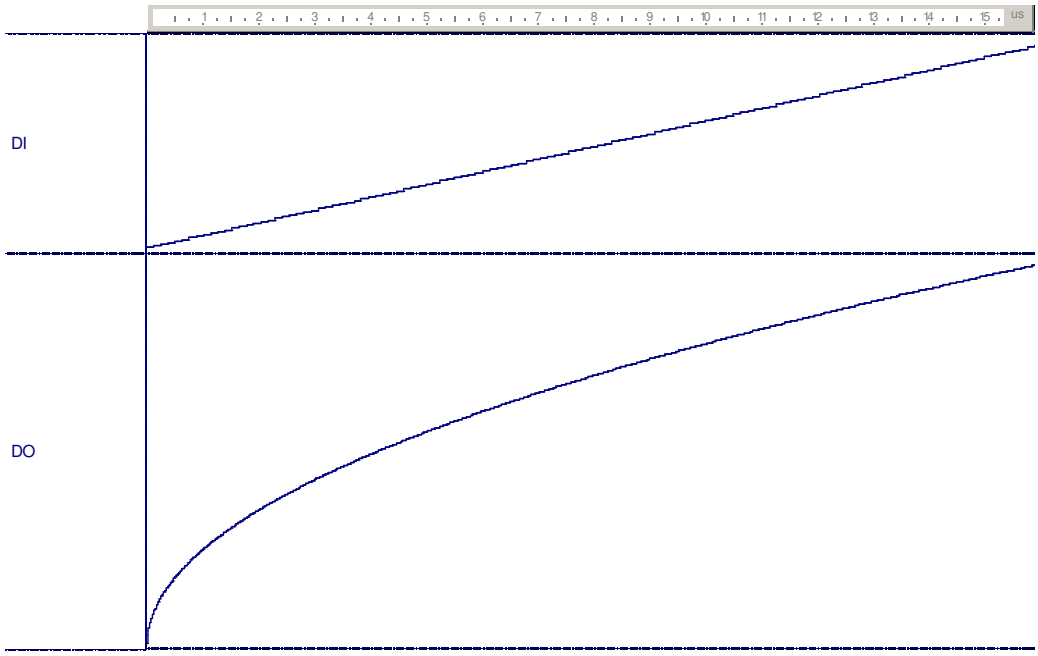


Fig.3.1. Input and output signals of the processor module for computing $\sqrt{x}$

Then, the processor module was synthesized, mapped, placed and routed in the Xilinx FPGA xc6lx-16 (Spartan-6) by the CAD system ISE ver. 13.3. The results of mapping for the input and output bit width of 24 bits are shown in Fig.3.2. The timing result message table for this core is the following:

Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
Autotimespec constraint for clock net CLK_BUFGP	SETUP	N/A	5.428ns	N/A	0
	HOLD	0.469ns		0	0

In fig. 3.3, the graphs of the dependences of hardware volume in the number of LUTs on the bitwidth of input data are shown. Note, that this bitwidth is equal to the one results, the modules have the conventional and pipelined structure. It should be noted that the modules with a bitwidth up to 32 inclusive additionally have a multiplication unit DSP48, and the rest of then four such blocks have.

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	88	18,224	1%
Number used as Flip Flops	80		
Number used as Latches	0		
Number used as Latch-thrus	0		
Number used as AND/OR logics	8		
Number of Slice LUTs	1,591	9,112	17%
Number used as logic	1,591	9,112	17%
Number using O6 output only	1,012		
Number using O5 output only	59		
Number using O5 and O6	520		
Number used as ROM	0		
Number used as Memory	0	2,176	0%
Number of occupied Slices	515	2,278	22%
Nummber of MUXCYs used	652	4,556	14%
Number of LUT Flip Flop pairs used	1,616		
Number with an unused Flip Flop	1,528	1,616	94%
Number with an unused LUT	25	1,616	1%
Number of fully used LUT-FF pairs	63	1,616	3%
Number of unique control sets	1		
Number of slice register sites lost to control set restrictions	0	18,224	0%
Number of bonded IOBs	49	232	21%

Fig.3.2. Results of mapping the square root processor for the input and output bit width of 24 bits

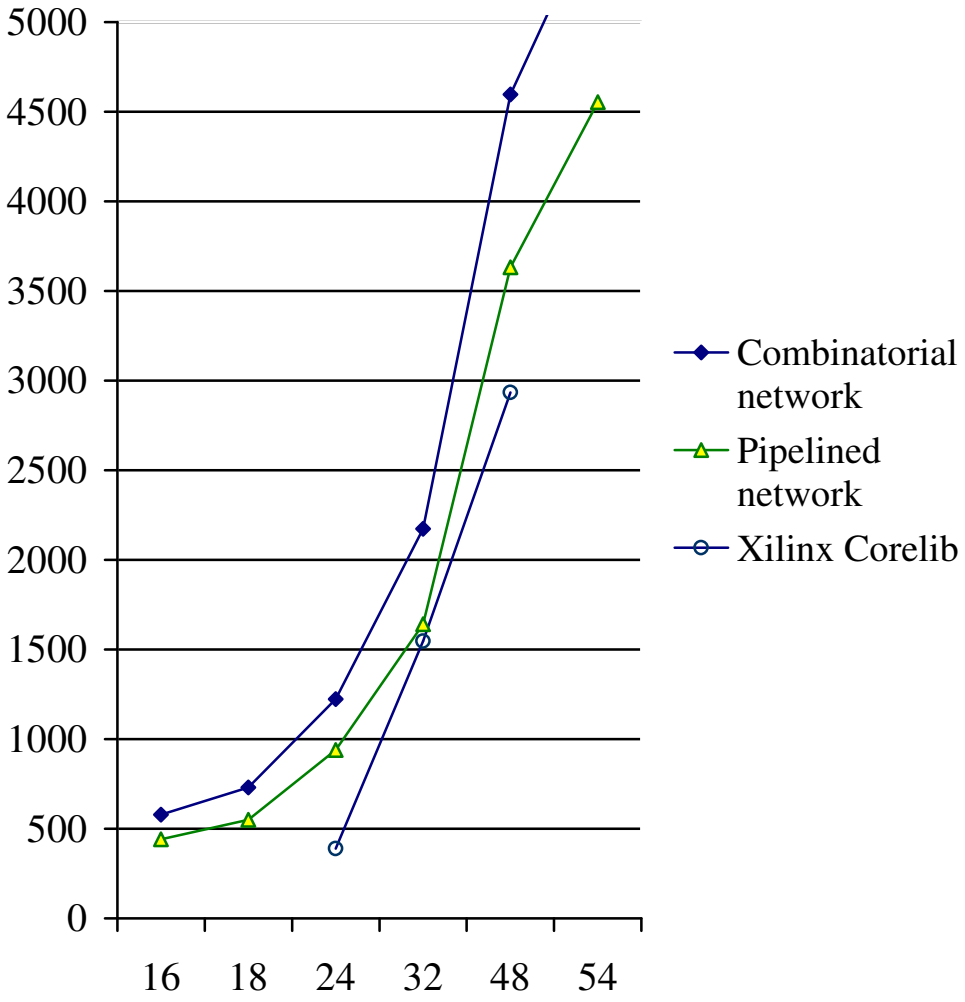


Fig.3.3. Hardware volume of the processor $\sqrt{x}$ depending on the bitwidth *n*

According to Fig. 3.3, the hardware volume of the module with the combinatorial network significantly outperform the volume of the pipelined module. This can be interpreted in that the compiler-synthesizer is better able to optimize the pipelined network because the parts of the network to be optimized, that is, the gates and LUTs located between the two layers of registers have much less complexity.

Fig. 3.4 shows the maximum clock frequency of the synthesized modules. When implementing the bitwidth 48 or more, the maximum clock frequency significantly decreases because the compiler-synthesizer builds a multiplication

unit with a bitwidth, which is more than 24, and at the same time, it manifests itself unable to build a pipelined network of the multiplication block.

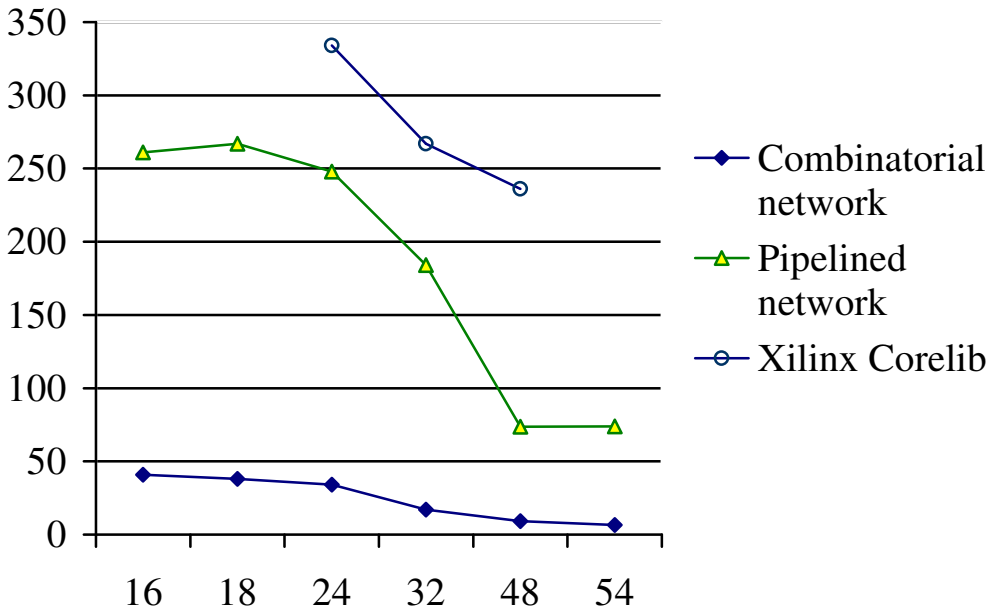


Fig.3.4. Maximum clock frequency of the processor $\sqrt{x}$ depending on the bitwidth n

For comparison, Fig. 3.3 and Fig. 3.4 show the characteristics of the licensed modules offered by Xilinx company. Consequently, the proposed module approximates the hardware costs to the "firm" module at $n = 32$, but in general, it loses to him including the speed. Its advantages are that it is free and can be configured for arbitrary input and output bit width. In addition, the proposed module has a lower latency delay.

For example, if the input data is normalized, then for bitwidth 24, the latent delay is only 15 cycles versus 24 cycles per competitor. If the circuit is not pipelined, then the delay from the input to the output is $T_L = 40.3$ ns and 71.9 ns, respectively. This means, that when implementing the floating-point calculations, the proposed module provides the greater performance.

As a result, the gesigned processing module has the very high effectiveness. Comparing to the CORDIC processor (see below), it has in 1.6 times less hardware volume in LUT number. It has in 2 times less latent delay due to the fact, that the modernized algorithm is calculated for *n*/2 clock cycles, and in 1.6 times higher clock frequency by the same bit width *n*.

3.2 Synthesis of the multifunction processor module

The project o the processor module for the $\sqrt{x}$, sine and cosine function calculations is described in VHDL as the entity SQRT_SIN, and implements the algorithm, described in the paragraph 2.5.4. Its text is shown in Appendix 1.

The module is tunable by the generic constant:

```
generic (n : natural := 12);
```

which gives the input and output bit width.

The module has the following ports:

```
port(
        CLK : in STD_LOGIC;
        RESET : in STD_LOGIC;
        F : in STD_LOGIC;        -- function select F=0 when SQRT
        XIN : in STD_LOGIC_VECTOR(n-1 downto 0);
        YOUT : out STD_LOGIC_VECTOR(n-1 downto 0);
        XOUT : out STD_LOGIC_VECTOR(n-1 downto 0)
        );
```

The processor module was synthesized, mapped, placed and routed in the Xilinx FPGA xc6lx-16 (Spartan-6) by the CAD system ISE ver. 13.3. The results of mapping for the input and output bit width of 24 bits are shown in Fig.3.5. The timing result message table for this core is the following:

Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
Autotimespec constraint for clock net CLK_BUFGP	SETUP	N/A	7.206ns	N/A	0
	HOLD	0.426ns		0	0

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	1,875	18,224	10%
Number used as Flip Flops	1,855		
Number used as Latches	0		
Number used as Latch-thrus	0		
Number used as AND/OR logics	20		
Number of Slice LUTs	3,251	9,112	35%
Number used as logic	3,215	9,112	35%
Number using O6 output only	2,797		
Number using O5 output only	4		
Number using O5 and O6	414		
Number used as ROM	0		
Number used as Memory	0	2,176	0%
Number used exclusively as route-thrus	36		
Number with same-slice register load	12		
Number with same-slice carry load	24		
Number with other load	0		
Number of occupied Slices	960	2,278	42%
Nummber of MUXCYs used	2,564	4,556	56%
Number of LUT Flip Flop pairs used	3,321		
Number with an unused Flip Flop	1,477	3,321	44%
Number with an unused LUT	70	3,321	2%
Number of fully used LUT-FF pairs	1,774	3,321	53%

Fig.3.52. Results of mapping the square root processor for the input and output bit width of 24 bits

In fig. 3.6, the graphs of the dependences of hardware volume in the number of LUTs on the bitwidth of input data are shown. Note, that this bitwidth

is equal to the one results, the modules have the conventional and pipelined structure..

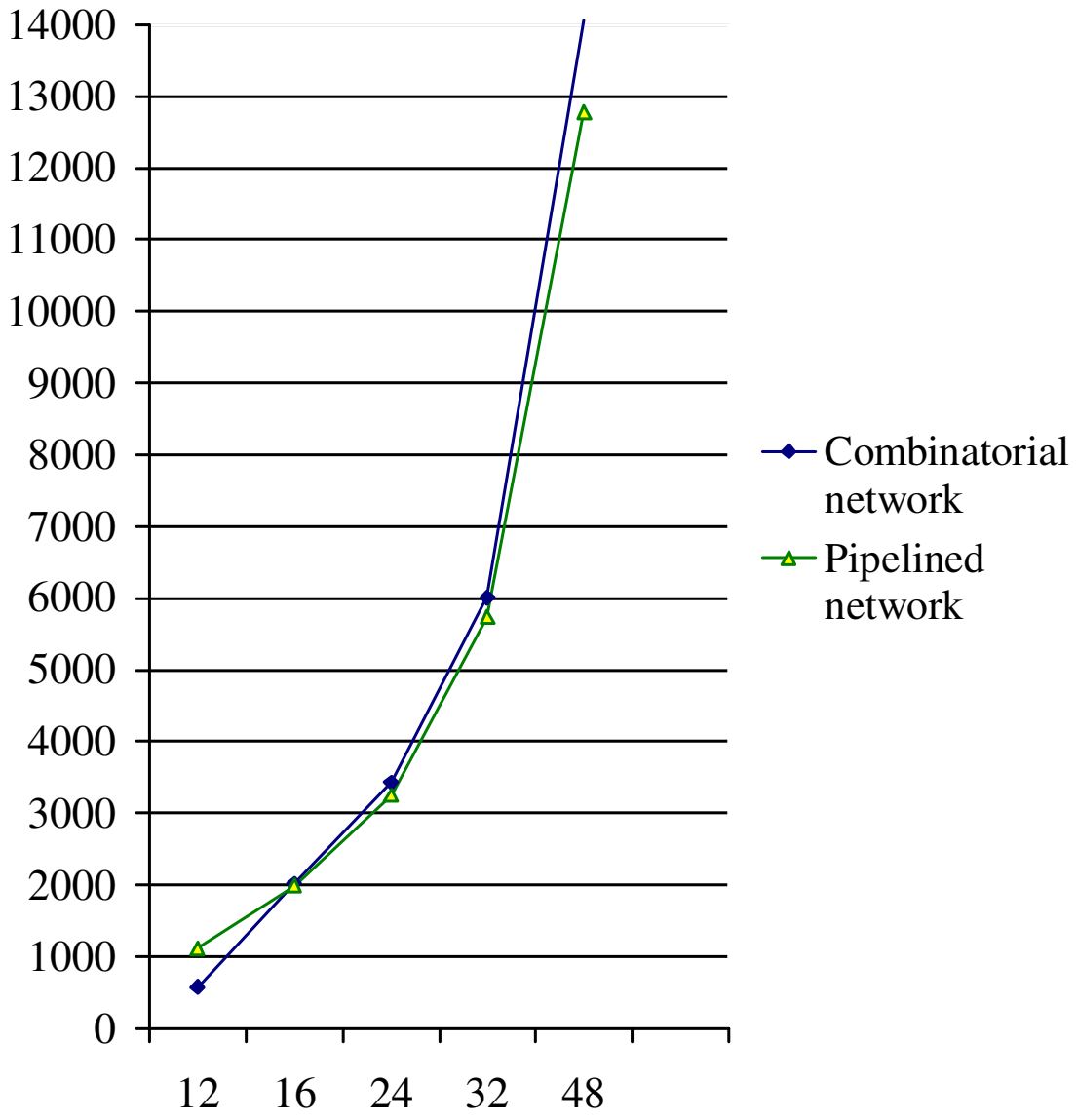


Fig.3.3. Hardware volume of the multifunction processor module depending on the bitwidth *n*

According to Fig. 3.6, the hardware volume of the module with the combinatorial network significantly outperform the volume of the pipelined module. This can be interpreted in that the compiler-synthesizer is better able to optimize the pipelined network because the parts of the network to be optimized,

that is, the gates and LUTs located between the two layers of registers have much less complexity.

Fig. 3.7 shows the maximum clock frequency of the synthesized modules. When implementing the bitwidth 48 or more, the maximum clock frequency significantly decreases because the compiler-synthesizer builds a multiplication unit with a bitwidth, which is more than 24, and at the same time, it manifests itself unable to build a pipelined network of the multiplication block.

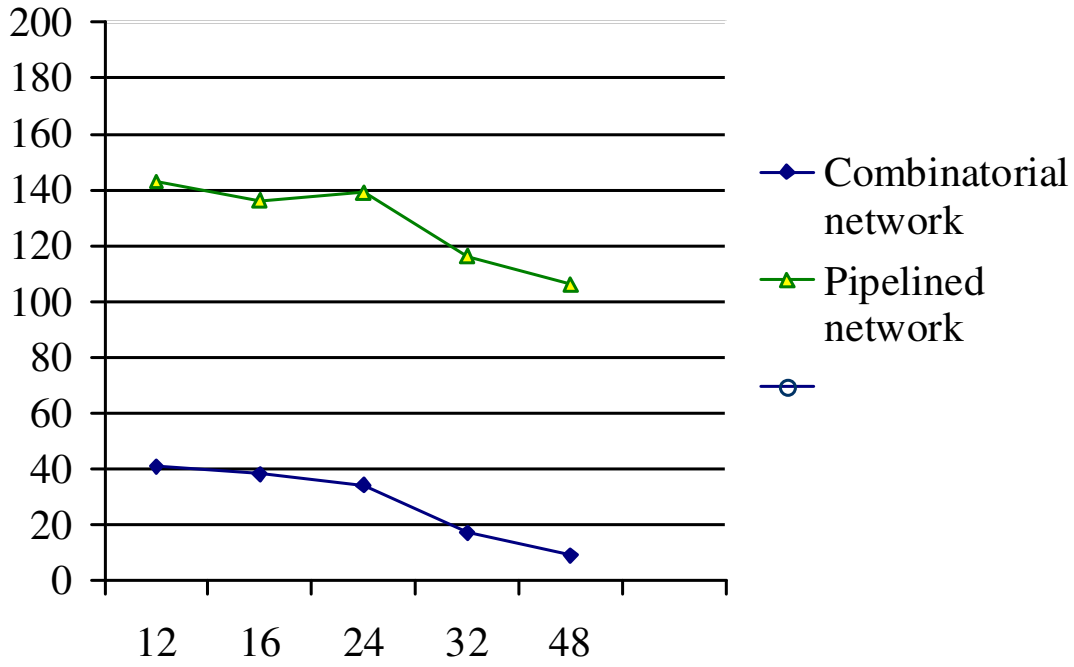


Fig.3.7. Maximum clock frequency of the multifunction processor module depending on the bitwidth *n*

Additionally, the module of the multifunction processor was synthesized with the fixed input F = 0 and F =1. This means that the synthesized network performs only either the function $\sqrt{x}$ or functions sin(x), cos(x), as the genuine CORDIC processor. The results of this synthesis are shown in Table 3.1.

Table 3.1

Parameters of different processor structures

Processor Structure	Hardware volume, LUTs	Maximum clock frequency, MHz
$\sqrt{x}$	2402	145
CORDIC	1665	156
Combined	3251	139

The Table 3.1 analysis shows that the combined structure has the hardware volume 3251 LUTs, which is smaller in 1.25 times than the overall hardware volume of 4067 LUTs of the processor computing $\sqrt{x}$, and the CORDIC processor. This means that really, the combined processor has the effect of the minimized hardware volume. Besides, its hardware volume is less than one of the analogous processor, which performs the function $\sqrt{x}$ but using the CORDIC algorithm [46].

But the speed of the combined processor (139 MHz) is slightly less than the speed of the processors, which perform the separate functions (145 and 156 MHz). This is explained, that the combined processor has the network, in which the critical path delay is expanded to the multiplexor delay.

As a conclusion, this example shows the rather good effectiveness of the prposed method of design the multifunction processors for calculating the elementary functions.

3.3 Preliminary conclusions

In this section, a set of processors for the elementary function implementation, which are designed according to the proposed method and algorithms are tested and probed. The results are the following.

The designed processing module for the square root function has the very high effectiveness. Comparing to the CORDIC processor, it has in 1.6 times less hardware volume in LUT number, has in 1.6 times higher clock frequency by the same bit width *n.* and has in $1.6 \cdot 2 = 3.2$ times less latent delay due to the fact, that the modernized algorithm is calculated for $n/2$ clock cycles.

The designed multifunction processing module has in 1.25 times less hardware volume than the processors, which perform the same algorithms but separately, by decreasing the clock performance only to 4 – 12%. This shows the rather good effectiveness of the prposed method of design the multifunction processors for calculating the elementary functions.

CONCLUSIONS

This thesis has presented a detailed description and analysis of the algorithm selection and design of the high-speed processing modules for the elementary function computing, and development of a new method for such modules design. On the base of the thesis materials the following conclusions are made.

1) The algorithms for the elementary function calculation, like polynomial approximation, functional recurrence, and digit recurrence algorithms are reviewed. It is found out that the algorithms, which utilize only additions, shifts, table functions, and small number of multiplications are the best candidates for the FPGA implementations. Among them the CORDIC-like algorithms play the leading role.

2) The FPGA architecture is investigated to select its features, which infer the selection of the elementary function algorithm implementation. This investigation helps to select the hardware cost and performance criteria for the processor module optimization.

3) It was proven, that the digit recurrence algorithms are best fitted for the FPGA implementation.

4) A new modification of the digit recurrence algorithm for the function $\sqrt{x}$ calculating is proposed, which provides the decreasing the latent delay up to three times.

5) The method of the synchronous dataflow graph (SDF) mapping into the pipelined structure of the processor module was studied, which helps to derive the effective structures for the elementary function calculations.

6) A method of the multifunction processor module design is proposed, which consists in forming the combined SDF, which performs a set of algorithms of the elementary function calculation, in balancing this SDF, and in mapping it

into the pipelined datapath, which is simpler and provides better hardware and performance effectiveness comparing to the other similar methods.

7) The proposed method of the multifunction processor module design was used in the design of the processor module for $\sqrt{x}$, sine, and cosine function calculations. Their configuring in FPGA and testing has shown that the designed processing module for the $\sqrt{x}$ function has the very high effectiveness. Comparing to the CORDIC processor, it has in 1.6 times less hardware volume in LUT number, has in 1.6 times higher clock frequency by the same bit width *n.* and has in 3.2 times less latent delay.

8) The designed multifunction processing module has in 1.25 times less hardware volume than the processors, which perform the same algorithms but separately, by decreasing the clock performance only to 4 – 12%. This shows the rather good effectiveness of the proposed method of design the multifunction processors for calculating the elementary functions.

9) The future works at this theme can be directed to the selection of the effective algorithms for the elementary function calculation and implementing them in the multifunction processors using the proposed method with the goals of the method improvement and proving its effectiveness.

REFERENCES

1. Woods R. FPGA-based Implementation of Signal Processing Systems / J. McAllister, G. Lightbody, and Y. Yi. / J. Wiley and Sons, Ltd., Pub. 2008. 364 p.
2. FPGA Implementations of Neural Networks. A. R. Omondi, and J. C. Rajapakse, Eds. Springer. 2006. 360 p.
3. Yoshikawaa K. Development of Fixed-point Square Root Operation for High-level Synthesis / N. Iwanagaa, and A.Yamawaki // Proc. 2nd Int. Conf. on Industrial Application Engineering. 2014. pp. 16 – 20.
4. Sergiyenko A.M. Square root calculations in FPGA / H.M.Jamal., P.A. Sergiyenko // System analysis and information technology: 20-th International conference SAIT 2018, Kyiv, Ukraine, May 21 – 24, 2018. Proceedings. – ESC "IASA" NTUU "Igor Sikorsky Kyiv Polytechnic Institute", 2018. – P. 163–164.,
5. Джамал Х. М. Алгоритм і структура модуля для обчислення квадратного кореня у ПЛІС / А.Сергієнко, П. Сергієнко // Праці міжнародної конференції "Безпека, Відмовостійкість, Інтелект", 10-11 травня 2018. — С. 74—77.
6. Muller J.M. Elementary functions. Algorithms and implementation. 2-nd ed. Springer. 2006. – 265 P.
7. Теслер Г.С. Вычисление элементарных функций на ЭВМ / Благовещенский Ю.В. / Киев: Техніка 208 с.–1977.
8. Alt H. Comparison of arithmetic functions with respect to Boolean circuits // In Proceedings of the 16th ACM STOC, 1984. – P. 466–470.
9. Cody W. Software Manual for the Elementary Functions /W. Waite / Prentice-Hall, Englewood Cliffs, NJ, 1980.

10. Попов Б.А., Теслер Г.С. Вычисление функций на ЭВМ / Теслер Г.С. / Киев: Наукова думка. 1984. — 600 с.
11. Volder J. E. The CORDIC trignometric computing technique // IRE Transactions on electronic Computers, pp. 330–334, Sept. 1959.
12. Walther J. S. A unified algorithm for elementary functions // in Proceedings of the AFIPS Spring Joint Computer Conference. 1971, P. 379–385.
13. Jun Cao J. C. High-performance architectures for elementary function generation / B. W. Y. Wei, // in Proceedings of the 13th IEEE Symposium on Computer Arithmetic, July 6-9, 1997. – P.184–188.
14. GNU C Library, [electronic resource]. Available at http://www.gnu.org/software/libc
15. Aho A. V. Optimal code generation for expression trees / S. C. Johnson // Journal of the ACM, N.23, 1976. –P. 488–501.
16. Sethi R. The generation of optimal code for arithmetic expressions / J. D. Ullman // Journal of the ACM, No17, 1970. – P. 715–728.
17. Aho A. V. Code generation for expressions with common subexpressions / ., S. C. Johnson, and J. D. Ullman // Journal of the ACM, No. 24, 1977. –P. 146–160.
18. Breuer M. A. Generation of optimal code for expressions via factorization // Communication of the ACM, No.12, 1969. – P. 333–340.
19. Takagi N. Generating a power of an operand by a table look-up and a multiplication //Proceedings of the 13th IEEE Symposium on Computer Arithmetic, Asilomar, California, USA, July 1997. – P. 126–131.
20. Hassler H. Function evaluation by table look-up and addition / N. Takagi // Proceedings of the 12th IEEE Symposium on Computer Arithmetic, Bath, England, July 1995. – P. 10–16.

21. Sarma D. D. Faithful bipartite ROM reciprocal tables / D. W. Matula // Proceedings of the 12th IEEE Symposium on Computer Arithmetic, Bath, England, July 1995. – P. 17–28.
22. Stine J. E. Symmetric bipartite tables for accurate function approximation / M. J. Schulte // Proceedings of the 13th IEEE Symposium on Computer Arithmetic, Asilomar, California, USA, July 1997. – P. 175–183.
23. Muller J.-M. A few results on table-based methods / Muller J.-M. // Research Report, V. 5, Oct. 1998.
24. Schulte M. J. The symmetric table addition for accurate function approximation / J. E. Stine // Journal of VLSI Signal Processing, V. 21, No. 2, 1999. – P. 167–177.
25. Florent de Dinechin A. T. Some improvements on multipartite table methods / A. T. Florent de Dinechin // Proceedings of the 15th IEEE Symposium on Computer Arithmetic, Vail, June 2001.Colorado, USA, 2001. – P. 128–135.
26. Tang P. T. P. Table-lookup algorithms for elementary functions and their error analysis // Proceedings of the 10th IEEE Symposium on Computer Arithmetic, Grenoble, France, June 1991. – P. 232–236,
27. Muller J.-M. Partially rounded small-order approximation for accurate, hardware-oriented, table-based methods / J.-M. Muller // Proceedings of the 16th IEEE Symposium on Computer Arithmetic, Santiago de Compostela, Spain, June 2003.
28. Brent R. Modern Computer Arithmetic /P. Zimmermann / Cambridge University Press. 2011. – 221 P.
29. Volder J. E.. The birth of CORDIC // Journal of VLSI Signal Processing Systems, Vol. 25, No. 2. 2000. –P.101–105.
30. Walther J.S. The story of unified CORDIC // Journal of VLSI Signal Processing Systems, Vol. 25, No. 2. 2000. –P.107–112.

31. R. Nave. Implementation of transcendental functions on a numerics processor // Microprocessing and Microprogramming, 1983. No.11. – P. 221–225.
32. Yoshikawaa K. Development of Fixed-point Square Root Operation for High-level Synthesis / N. Iwanagaa, A.Yamawaki // Proc. 2nd Int. Conf. on Industrial Application Engineering. 2014. – P. 16 – 20.
33. Parhami B. Computer Arithmetic: Algorithms and Hardware Designs. Oxford University Press, 2010 - 641 P.
34. Muller J.-M, Handbook of Floating-Point Arithmetic / N. Brisebarre, F. Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N.Revol, D. Stehlé,Serge Torres / Springer. 2010. – 571.
35. Andraka R. A survey of CORDIC algorithms for FPGA based computers // FPGA '98 Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays. – P. 191-200.
36. Series FPGAs CLB User Guide. UG474 (v1.7). November 17, 2014. – 74 P. [Electronic resource]. Available at www.xilinx.com
37. Spartan-6 FPGA DSP48A1 User Guide. UG389 (v1.2). May 29, 2014. – 46 P. [Electronic resource]. Available at www.xilinx.com
38. Сергиенко А.М. Отображение периодических алгоритмов в программируемые логические интегральные схемы / В.П. Симоненко // Электронное моделирование. Т. 29. No.2. 2007. – С.49–61.
39. Edwards S. Design of Embedded Systems: Formal Models, Validation, and Synthesis / L. Lavagno, E.A. Lee, A. Sangiovanny-Vincentelli // Proc. IEEE, vol.85, pp.366–390, March 1997.
40. Khan S. A, Digital Design of Signal Processing Systems. John Wiley & Sons. 2011.
41. FPGA Implementations of Neural Networks / A. R. Omondi, J. C. Rajapakse, eds. — Springer. — 2006. — 360 p.

42. 7 Series FPGAs CLB User Guide. UG474 (v1.7). // Xilinx com. –2014. – 58 P. [Electronic resource} Available at: www.xilinx.com /support/documentation/user_guides/ug474_7Series_CLB.pdf
43. Люстерник Л.А. Математический анализ. Вычисление элементарных функцій /., Червоненкис О.А., Янпольский А.Р. / – М.: ГИЗ физ.-мат. лит. – 1963. – 247 с.
44. Благовещенский Ю.В., Теслер Г.С. Вычисление элементарных функций на ЭВМ. – Киев: Техніка. –1977. – 208 с.
45. Смолов В. Б. Специализированные процессоры: Итерационные алгоритмы и структуры / В. Д. Байков, В. Д. Смолов / М.: Радио и связь. — 1985. —288 с.
46. Бікташева С.Р. CORDIC-метод обчислення квадратного кореня / С.Р. Бікташева, Л.В. Мороз, М.Ю. Стахів // Вісн. Нац. Ун-ту "Львівська політехніка". Сер.: Електроніка : [зб. наук. пр.] — Львів : Вид-во Нац. ун-ту "Львів. політехніка", — 2006. — С. 152-155.
47. Chen T.C. Automatic computations of exponentials, logarithms, ratios and square roots. IBM J. Res. and Develop. – 1972. – №4. – P. 380-388.
48. Szyzaki M. FPGA computation of magnitude of complex numbers using modified CORDIC algorithm / Smyk R. // Zeszyty Naukowe Wydziału Elektrotechniki i Automatyki Politechniki Gdańskiej Nr 47. 2015. – P. 35–38.
49. ЭВМ и теория расписаний . Ред. И.Г. Коффман. М.: Мир. – 1985. – 400 с.

APPENDICES

APPENDIX 1

Processor for calculating the square root function

```
--------------------------------------------------------------------------------
--
-- Title       : SQRT_C
-- Design      : SQRT_C
-- Author      : Jamal
-- Company     : КПИ
--
--------------------------------------------------------------------------------
--
-- File        : SQRT_C.vhd
-- Generated   : Thu Nov  9 20:09:16 2017
-- From        : interface description file
-- By          : Itf2Vhdl ver. 1.20
--
--------------------------------------------------------------------------------
--
-- Description :
--y0 = x;  x0 = x; m = 0; f = 0;
--for (i = 1; i < n; i = i + 1) {
--       t  = xi  + 2-m xi;
--       u  = t  + 2-m t  ;
--       if   (u i 1)  {
--                f = 1;
--               xi+1  = xi ;
--               yi+1  = yi ;
--            }
--       else {
--               xi+1  = u;
--               yi+1  = yi  + 2-m yi;
--            }
--       if (f == 1) m = m + 1;
--}
--                        correction of result:
--  yi(m+1) = yi(m) + yi(m)*(1-u(m))= yi(m)(2 - u(m))
-- by n/2 = 8, max(1-u(m))= 1012
-- u  = t  + 2-m t = xi + 2-m-1 xi + 2-2m xi --zgodom;

--------------------------------------------------------------------------------
--Spartan6
-- 3-input adders
```

```
-- fully pipelined, control Y is delayed
-- 16  496 LUT 160 cLb 1DSP - 4.63 ns
-- 18  601 LUT 220 cLb 1DSP - 4.453ns
-- 24 1040 LUT 329 cLb 1DSP - 4.507ns
-- 24  960 LUT 299 cLb 1DSP - 4.79 ns --unnorm.
-- 32 1865 LUT 561 cLb 1DSP - 5.662ns
-- 48 4158 LUT 1203cLb 4DSP - 14.308ns
-- 48 4171 LUT 1252cLb 4DSP - 14.104ns -- retiming
-- 54 5229 LUT 1604cLb 4DSP - 14.43ns -- unnorm
-- 54 5229 LUT 1604cLb 4DSP - 14.43ns -- unnorm

-- not pipelined normalized
-- 24 1040 LUT 329 cLb 1DSP - 4.507ns

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_STD.all;
entity SQRT_C5 is
	generic(ni:natural:=24;
		no:natural:=24;
		norm:natural:=1;  --0- unnormalized input data
		pipe:natural:=0);    -- 1 - fully pipelined
	port(
		CLK : in STD_LOGIC;
		DI : in STD_LOGIC_VECTOR(ni-1 downto 0);  --	исходное данное
		DO : out STD_LOGIC_VECTOR(no-1 downto 0)	-- результат
		);

end SQRT_C5;
architecture synt of SQRT_C5 is

	function flo(x:std_logic_vector) return natural  is
		variable n:natural;
	begin
		n:=0;
		for i in x'left downto 1 loop
			if x(i)='1' then exit;
			else
				n:=n+1;
			end if;
		end loop;
		return n;
	end ;

	signal xn:unsigned(ni-1 downto 0);
	signal n:natural;
	type Tstage is array (1 to no+1) of unsigned(ni+2 downto 0);
```

```
        type Tstage1 is array (1 to no+1) of unsigned(ni+3 downto 0);
        signal yi,xi,t: Tstage:=(others=>(others=>'0'));
        signal u: Tstage1:=(others=>(others=>'0'));
        signal doi:unsigned(ni+1 downto 0);
        signal e1,r,xx,em:unsigned(ni downto 0);
        signal ex:unsigned(ni/2+1 downto 0);
        signal yc:unsigned(ni+1 downto 0);
        signal p,pd:unsigned(ni+1 downto 0);
        constant a0:unsigned(ni-1 downto 0):=(others=>'0');
        constant a1:unsigned(ni downto 0):=  '1'&a0;

begin

        n<= flo(DI);

        process(clk,di,doi) begin
                if rising_edge(clk) then
                        if norm = 0 then
                                xn<= SHIFT_LEFT(unsigned(DI),n/2*2);
                        else
                                xn<=unsigned(DI);
                        end if;

                        DO<= std_logic_vector(doi(ni downto ni-no+1));
                end if;
        end process;


        yi(1)<='0'&xn&"00";
        xi(1)<='0'&xn&"00";

        NR:    if pipe=0 generate

                STAGES: for m in 1 to ni/2+4 generate

          u(m)<= '0'&xi(m) + SHIFT_RIGHT(xi(m),m+1)+ SHIFT_RIGHT(xi(m),2*m);

           xi(m+1)<= u(m)(ni+2 downto 0) when u(m)(ni+3 downto ni+2)="00" else
                                        xi(m);
yi(m+1)<= yi(m)+ SHIFT_RIGHT(yi(m),m) when u(m)(ni+3 downto ni+2)="00" else
                                        yi(m);
                        end generate;
                        xx<=xi(ni/2)(ni+2 downto 2);
                        r<=yi(ni/2)(ni+2 downto 2);

                        e1<=a1 - xx; --xi(ni/2+1)(ni+4 downto 4);
                        p<=    r(ni downto ni/2+1)* e1(ni/2+1 downto 0);
```

```
                ex<= p(ni downto ni/2-1);
                yc <= yi(ni/2)(ni+2 downto 1) + ex;

        end generate;



        RR:     if pipe=1 generate
                STAGES: for m in 1 to ni/2 generate
                        process(CLK, xi,t)
                                variable ut:unsigned(ni+3 downto 0);
                        begin
                ut:= '0'&xi(m) + SHIFT_RIGHT(xi(m),m+1)+ SHIFT_RIGHT(xi(m),2*m);
                                if rising_edge(CLK) then
                                        u(m)<= ut;
                                        if u(m)(ni+3 downto ni+2)="00" then
                                                yi(m+1)<= yi(m)+ SHIFT_RIGHT(yi(m),m);
                                        else
                                                        yi(m+1)<= yi(m);
                                        end if;

                                        if ut(ni+3 downto ni+2)="00" then

                                           xi(m+1)<= ut(ni+2 downto 0);
                                        else
                                         xi(m+1)<=xi(m);
                                        end if;
                                end if;
                        end process;

                end generate;
                           process(CLK, xi,yi,p) begin
                                        xx<=xi(ni/2)(ni+2 downto 2);
                                        ex<= p(ni+1 downto ni/2);
                                if rising_edge(CLK) then
                                        r<=yi(ni/2)(ni+2 downto 2);       --+1st
                                        e1<=a1 - xx;
                                p<=    r(ni downto ni/2-1)* e1(ni/2-1 downto 0); --2st
                                         pd<=p;
                                yc <= yi(ni/2)(ni+2 downto 1) + ex/2;            --+3st
                                end if;
                        end process;
        end generate;
        doi<=SHIFT_RIGHT(yc,n/2) when norm = 0 else yc;

end synt;
```

Processor for calculating square root, sine and cosine functions

```
--------------------------------------------------------------------------------
--
-- Title       : SQRT_SIN
-- Design      : sine
-- Author      : Aser
-- Company     : KPI
--
--------------------------------------------------------------------------------
--
-- File        : c:\MY_Designs\Sine_approx\sine\src\SQRT_SIN.vhd
-- Generated   : Sun May  6 17:24:50 2018
-- From        : interface description file
-- By          : Itf2Vhdl ver. 1.22
--
--------------------------------------------------------------------------------
--
-- Description :
--
--------------------------------------------------------------------------------

--Spartan6
-- 3-input adders
-- fully pipelined, control Y is delayed
-- 24 1875 tt 3251 LUT 960 cLb  - 7.206 ns
-- 24 1270 tt 2402 LUT 653 cLb  - 6.894ns ns F=0
-- 24 1688 tt 1665 LUT 462 cLb  - 6.394ns ns F=1
-- 12  506 tt 1117 LUT 345 cLb  - 6.957 ns
-- 16  875 tt 1983 LUT 605 cLb  - 7.328 ns
-- 32 3284 tt 5735 LUT 1674cLb  - 8.608 ns
-- 48 7302 tt12785 LUT 3779cLb  - 9.429 ns

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_signed.all;
use IEEE.STD_LOGIC_arith.all;
use IEEE.Math_real.all;


entity SQRT_SIN is
        generic (n : natural := 8);
        port(
                CLK : in STD_LOGIC;
                RESET : in STD_LOGIC;
                F : in STD_LOGIC;     --    function select          F=0 when SQRT
                XIN : in STD_LOGIC_VECTOR(n-1 downto 0);
```

```
        YOUT : out STD_LOGIC_VECTOR(n-1 downto 0);
        XOUT : out STD_LOGIC_VECTOR(n-1 downto 0)
        );
end SQRT_SIN;


architecture synt of SQRT_SIN is
    type Tarr is array (0 to n) of  STD_LOGIC_VECTOR(n downto 0);

    signal x,y,fi,atan : Tarr;

    constant Ki: integer:= integer(0.607252935*2.0**(n-2));
    constant K: STD_LOGIC_VECTOR(n downto 0):=conv_std_logic_vector(Ki,n) ;

begin

    ARCS: for i in 0 to n generate
    atan(i)<= conv_std_logic_vector( integer(ARCTAN(2.0**(-i)*2.0**(n-2))), n );
    end generate;


    process(CLK)   begin
        if rising_edge(CLK) then
            fi(0)<= SXT(XIN,n);
            if F='0' then
                x(0)<=SXT(XIN,n) ;
                y(0)<=SXT(XIN,n) ;
            else
                x(0)<= K;
                y(0)<= (others=>'0');
            end if;
            YOUT<= y(n)(n-1 downto 0);
            XOUT<= x(n)(n-1 downto 0);
        end if;
    end process;


    STAGES: for i in 0 to n-1 generate
        process(CLK)
            variable u:STD_LOGIC_VECTOR(n+1 downto 0);
            variable ii:STD_LOGIC_VECTOR(5 downto 0);
        begin
            ii:=conv_std_logic_vector(i,6);
            if rising_edge(CLK) then
                if F='0' then
                    u  := x(i) + SHR(x(i), ii) + SHR(x(i), (ii&'0')+2);
                    if (u(n) ='0' or u(n+1) ='0') then
```

```
                                        x(i+1)<= u ;
                                        y(i+1)<= y(i) + SHR(y(i), ii+1) ;
                                else
                                        x(i+1)<= x(i);
                                        y(i+1)<= y(i) ;
                                end if;
                        else
                                if (fi(i)(n) = '0') then
                                        u  := x(i) - SHR(y(i), ii);
                                        y(i+1)<= y(i) + SHR(x(i), ii);
                                        fi(i+1)<= fi(i) - atan(i);
                                else
                                        u  := x(i) + SHR(y(i), ii);
                                        y(i+1)<= y(i) - SHR(x(i), ii);
                                        fi(i+1)<= fi(i) + atan(i);
                                end if;
                                x(i+1)<= u;
                        end if;
                end if;
        end process;
    end generate;




end synt;
```

APPENDIX 2

Copies of publications

УДК 004.383

Анатолій Сергієнко, Хасан Мухамед Джамал, Павло Сергієнко

АЛГОРИТМ І СТРУКТУРА МОДУЛЯ ДЛЯ ОБЧИСЛЕННЯ КВАДРАТНОГО КОРЕНЯ У ПЛІС

Anatoliy Sergiyenko, Hasan Muhammad Jamal, Pavlo Serhienko

ALGORITHM AND STRUCTURE OF THE SQUARE ROOT CALCULATOR IMPLEMENTED IN FPGA

Розглядається розробка апаратних пристроїв для обчислення функції квадратного кореня за ітераційним алгоритмом. Запропонований алгоритм дає змогу прискорити обчислення функції квадратного кореня та зменшити апаратні витрати за рахунок обчислення кількох ітерацій табличним методом. Запропонований алгоритм розрахований на реалізацію у програмованих логічних інтегральних схемах.

Ключові слова: ПЛІС, квадратний корінь, конвеєр.

Рис.: 3. Табл.:1. Бібл.: 4.

The development of the hardware units for the square root (SQRT) function calculations is considered, which is based on the CORDIC-like iterative algorithm. The proposed algorithm helps both to speed-up the SQRT function calculations and to minimize the hardware volume due to substituting some iterations by the look-up tables. The algorithm is intended for the SQRT function implementation in FPGA.

Key words: FPGA, SQRT, CORDIC, pipeline.

Fig.: 3. Tabl.:1. Bibl.: 4.

Вступ. Функція квадратного кореня $\sqrt{x}$— важлива елементарна функція в наукових розрахунках, обробці цифрових сигналів та обробці зображень [1]. Наприклад, вона використовується у нейронних мережах [2]. В даний час багато задач вирішуються у програмованих логічних інтегральних схемах (ПЛІС), де також необхідно розраховувати функцію $\sqrt{x}$.

Існують різні віртуальні модулі для обчислення функції $\sqrt{x}$, які пропонуються виробниками ПЛІС та сторонніми компаніями [3]. Але ці модулі були розроблені десятиліття тому, і вони, як правило, не враховують особливості нових поколінь ПЛІС. Тому такі модулі потребують модернізації. У роботі [4] запропоновано вдосконалений алгоритм обчислення функції $\sqrt{x}$, який орієнтований на реалізацію у ПЛІС. У даній роботі пропонується ще один алгоритм, який ефективно реалізується у ПЛІС.

Алгоритми "цифра за цифрою". Виконання алгоритму розрахунку елементарної функції типу "цифра за цифрою" зводиться до повторення одноманітних ітерацій, результатами яких є чергові точні цифри результату. Відомий алгоритм CORDIC, який призначений для розрахунків $\sqrt{x}$, полягає в наступному. Він обчислює функцію atanh(x/y). Але побічним результатом є функція $K\sqrt{x^2+y^2}$, а за рахунок заміни $x = A + 0.25$, $y = A - 0.25$, отримують $K\sqrt{A}$ [3,5]. Недоліками цього алгоритму є додаткове множення результату на коефіцієнт $1/K \approx 1.207$ і повторення деяких ітерацій для збіжності алгоритму.

Більш конструктивним алгоритмом є алгоритм "цифра за цифрою" обчислення функції $\sqrt{x}$ [4], який базується на наступних співвідношеннях. Для кожного числа $x \in [0.25; 1.0]$ знаходять коефіцієнти $a_i \in [0; 1]$, такі що

$$\prod_{i=1}^{\infty}(1 + a_i 2^{-i})^2 = 1.0. \qquad (1)$$

Звідси

$$1/\sqrt{x} \approx \prod_{i=1}^{m}(1 + a_i 2^{-i}) \quad \text{або} \quad \sqrt{x} \approx x\prod_{i=1}^{m}(1 + a_i 2^{-i}). \qquad (2)$$

Отже, обчислення функції $\sqrt{x}$ полягає у виконанні конвергентного процесу, згідно з яким вираз (1) наближається до одиниці, в той час як вираз (2) наближається до шуканого значення. Алгоритм цього процесу виражається наступним чином:

```
x[0] = x; y[0] = x;
for(i = 0, i < n, i++) {
     t = x[i] + 2^(-i)*x[i];
     q = t + 2^(-i)*t;
     if (q < 1) {
          x[i+1] = q;
          y[i+1] = y[i] + 2^(-i)*y[i];}// a[i]=1
     else {
          x[i+1] = x[i];
          y[i+1] = y[i];}// a[i]=0
}
```

Результатом є $y[n] = \sqrt{x}$.

Модернізований алгоритм. Найбільшу затримку розглянутого алгоритму дає подвійне додавання зсунутих даних з результатами t та q. Ці обчислення можуть бути замінені одним етапом:

$$q = x_i + 2^{-m} x_i + 2^{-m}(x_i + 2^{-m} x_i) = x_i + 2^{-m+1} x_i + 2^{-2m} x_i.$$

Оскільки у сучасних ПЛІС трьохвходовий суматор реалізується як один ступінь, який будується на основі шестивходових логічних таблиць (ЛТ), то такі обчислення можуть бути виконані за один такт без додаткових затримок і витрат на обладнання. Аналіз алгоритму показує, що коли i досягає межі $n/2$, то старші розряди числа x_i стають одиничними а i старших розрядів y_i є точними старшими розрядами результату. Отже, решту отриманих бітів можна обчислити після аналізу та розрахунку різниці $1 - x_i$.

Нехай $\varepsilon_1 = 1 - x_{n/2}$, $\varepsilon_x = \sqrt{x} - y_{n/2}$. Ці величини згідно з (1) і (2) дорівнюють

$$\varepsilon_1 = 1 - x\prod_{i=1}^{n/2}(1 + a_i 2^{-i})^2; \quad \varepsilon_x = \sqrt{x} - x\prod_{i=1}^{n/2}(1 + a_i 2^{-i}).$$

Представимо $z = \sqrt{x}\prod_{i=1}^{n/2}(1 + a_i 2^{-i})$, тоді

$$\varepsilon_1 = 1 - z^2 = (1 + z)(1 - z); \qquad \varepsilon_x = \sqrt{x}\,(1 - z).$$

Оскільки $z \approx 1$, то $\varepsilon_1 \approx 2(1 - z)$; і $\varepsilon_x \approx \sqrt{x}\ \varepsilon_1/2 \approx y_{n/2}(1 - x_{n/2})/2$. Тоді результат дорівнює $y_n = y_{n/2} + y_{n/2}(1 - x_{n/2})/2$ та модернізований алгоритм є наступний:

```
x[0] = x; y[0] = x;
for (i = 0; i < n/2; i++){
     q = x_i + 2^(-i+1)*x_i + 2^(-2i)*x_i;
     if (q < 1.0){
        x_(i+1) = u;
        y_(i+1) = y_i  + 2^(-i-1)*y_i;}
     else {
        x_(i+1) = x_i ;
        y_(i+1) = y_i ;}
      }
y = y_(i+1) + y_(i+1)*(1.0 - x_(i+1))/2;
```

Експериментальні результати. Отриманий алгоритм був описаний мовою VHDL як віртуальний модуль. Цей модуль було зконфігуровано для ПЛІС Xilinx Spartan-6 для різної розрядності вхідних і вихідних даних. На рисунках 1 та 2 наведено залежність апаратних витрат в кількості ЛТ, а також максимальної тактовової частоти від розрядності n вхідних даних і результатів для комбінаційної і конвеєрної схем цього модуля, відповідно. Слід відзначити, що модулі з розрядністю 32 додатково мають один блок множення DSP48, а решта — чотири таких блоки.

Для порівняння, на рис. 1, 2 показані характеристики віртуальних модулів, які пропонуються компанією Xilinx Inc. Загалом, запропонований

модуль має більші апаратні витрати і меншу тактову частоту. Це пояснюється тим, що модуль, який генерується засобом Xilinx Coregen, описаний на рівні ЛТ і тригерів і тому адаптований до архітектури ПЛІС конкретного типу. А запропонований модуль, хоча і не потребує тривалого генерування, повинен бути зкомпільованим синтезатором, який виконує ефективну, але не оптимальну оптимізацію.

Переваги запропонованого модуля полягають в тому, що він є безкоштовним і може бути налаштований на довільну розрядність вхідних та вихідних даних, а також для будь-якого типу ПЛІС. Крім того, запропонований модуль має нижчу латентну затримку, що є важливим, наприклад, при виконанні на його основі операцій з плаваючою комою.

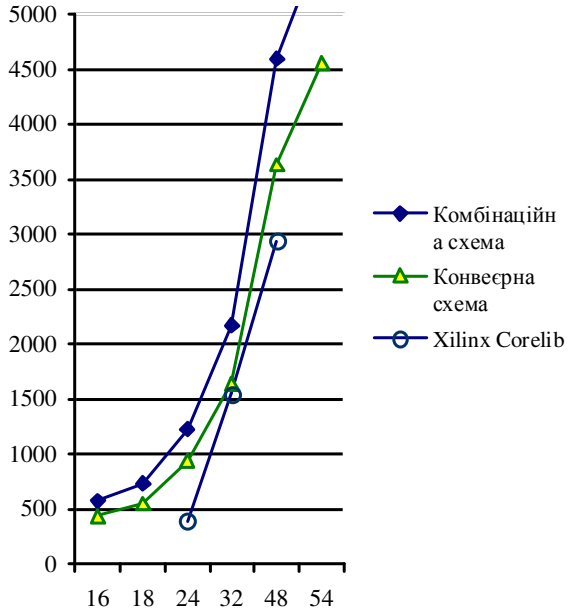


Рис.1. Апаратні витрати блоку $\sqrt{x}$, ЛТ, в залежності від *n*

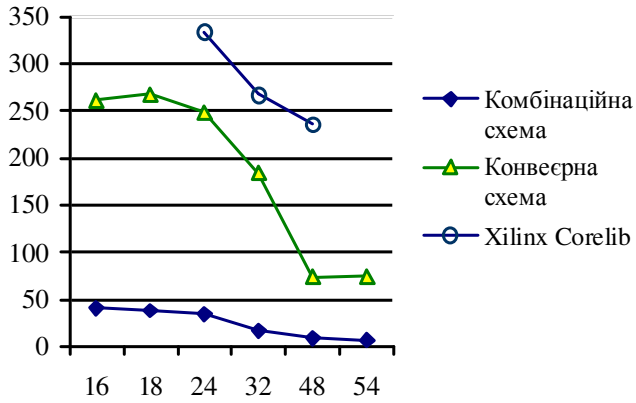


Рис.2. Максимальна тактова частота, МГц, блоку $\sqrt{x}$ в залежності від *n*

Наприклад, для розрядності 24 біт, латентна затримка становить лише 15 тактів проти 25 тактів у конкурентного ядра. Це означає, що при виконанні розрахунків з плаваючою комою запропонований модуль забезпечує більшу продуктивність.

Висновки. Запропоновано модифікований алгоритм “цифра за цифрою” для обчислення функції квадратної кореня. Алгоритм відрізняється мінімізованою кількістю ітерацій, яка приблизно удвічі менша за кількість розрядів результату. Алгоритм описаний мовою VHDL і призначений для реалізації у ПЛІС будь-якої серії. Найбільш ефективна його реалізація при обчисленні функції $\sqrt{x}$ з плаваючою комою.

Список використаних джерел

1. Woods R. FPGA-based Implementation of Signal Processing Systems / J. McAllister, G. Lightbody, Y. Yi / J. Wiley and Sons, Ltd., Pub. 2008. 364 p.
2. FPGA Implementations of Neural Networks". A. R. Omondi, and J. C. Rajapakse, Eds. Springer. 2006. 360 p.
3. Yoshikawaa K. Development of Fixed-point Square Root Operation for High-level Synthesis / N. Iwanagaa, A. Yamawaki // Proc. 2nd Int. Conf. on Industrial Application Engineering. 2014. P. 16 — 20.
4. Сергієнко А. М. Реалізація функції квадратного кореня у ПЛІС / П. А. Сергієнко // Вісник НТУУ «КПІ»: Інформатика, управління та обчислювальна техніка: [зб. наук. пр.] Київ. 2014. Т.60, С. 40 — 45.
5. Бікташева С. Р. CORDIC-метод обчислення квадратного кореня / С. Р. Бікташева, Л.В. Мороз, М. Ю. Стахів // Вісн. Нац. Ун-ту "Львівська політехніка". Сер.: Електроніка : [зб. наук. пр.] Львів : Вид-во Нац. ун-ту "Львів. політехніка", 2006. С. 152—155.
6. Chen T.C. Automatic computations of exponentials, logarithms, ratios and square roots. // IBM J. Res. and Develop. 1972. №4. P. 380 — 388.

***Sergiyenko A. M.*[1], *Hasan M. J.*[1], *Sergiyenko P. A.*[1]**
[1] *Computer Engineering Department of NTUU "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine*

Square root calculations in FPGA

Introduction. The square root function $\sqrt{x}$ is important elementary function in the scientific calculations, digital signal and image processing [1]. The artificial neural nets need this function as well [2]. At present, the field programmable gate arrays (FPGAs) are expanded for solving the problems, where the function $\sqrt{x}$ calculations are of demand. There are different IP cores of the function $\sqrt{x}$, which are proposed by the FPGA manufacturers and third-party companies [3]. But these IP cores were designed decades ago and they usually don't take into account the features of the new FPGA generations. Therefore, they need improvements. In the presentation, an improved algorithm of the function $\sqrt{x}$ is proposed, which is suitable for the FPGA implementation.

CORDIC-type algorithms. The CORDIC-type algorithm of the elementary function calculation derives a single exact digit of the result in each computation step. The well-known CORDIC algorithm of the $\sqrt{x}$ calculations consists in the following. It calculates the function atanh(x/y). But the side result is the function $K\sqrt{x^2-y^2}$, and by the substitution $x = A + 0.25, y = A - 0.25$, we get $x_n = K\sqrt{A}$ [3]. The disadvantages of this algorithm are additional multiplication to the coefficient $1/K \approx 1.207$, and repeating some iterations for the algorithm convergence.

More constructive algorithm is the CORDIC-like algorithm of the function $\sqrt{x}$ calculation [4], which is based on the following relations. For each number $x \in [0.25; 1.0]$ the coefficients $a_i \in [0; 1]$ are found so

$$x\prod_{i=1}^{n}(1+a_i2^{-i})^2 \approx 1.0; \Rightarrow 1/\sqrt{x} \approx \prod_{i=1}^{n}(1+a_i2^{-i}) \Rightarrow \sqrt{x} \approx x\prod_{i=1}^{n}(1+a_i2^{-i}). \quad (1)$$

The algorithm is the following:

```
x[0] = x; y[0] = x;
for(i = 0, i < n, i++) {t = x[i] + 2^(-i)*x[i];
   q = t + 2^(-i)*t;
   if (q < 1) {x[i+1] = q; y[i+1] = y[i] + 2^(-i)*y[i];}// a[i]=1
   else {x[i+1] = x[i];  y[i+1] = y[i];}// a[i]=0
}
```

The result is $\sqrt{x} \approx$ y[n].

Modernized algorithm. The most delay in the considered algorithm gives the twofold addition of the shifted data.These stages of addition can be substituted by a single stage:

$q = (x_i + 2^{-i}x_i) + 2^{-i}(x_i + 2^{-i}x_i) = x_i + 2^{-i+1}x_i + 2^{-2i}x_i$.

Because modern FPGAs perform the three input adder as a single stage of the six input look-up tables (LUTs), then such computations can be implemented for a single clock cycle without additional time and hardware overheads.

The algorithm analysis shows that when i reaches the limit $n/2$, then the most i–1 significant bits of x_i become equal to a one by any x_0, and i most significant bits of y_i are exact digits of the result. Therefore, the rest of resulting bits can be calculated after analysis and computation the difference 1–x_i.

Consider $\varepsilon_1 = 1–x_{n/2}$, and $\sqrt{x} = \varepsilon_x + y_{n/2}$. Then, to get the exact result, the correction ε_x is derived from the value ε_1, and it is added to the approximated result. Due to (1),

$$\varepsilon_1 = 1 - x\prod_{i=1}^{n/2}(1+a_i2^{-i})^2; \varepsilon_x = \sqrt{x} - x\prod_{i=1}^{n/2}(1+a_i2^{-i}).$$

Then

$$z = \sqrt{x}\prod_{i=1}^{n/2}(1+a_i2^{-i}); \varepsilon_1 = 1 - z^2 = (1+z)(1-z); \varepsilon_x = \sqrt{x}(1-z).$$

Consider $z \approx 1$, then $\varepsilon_1 \approx 2(1-z); \varepsilon_x \approx \sqrt{x}\varepsilon_1/2 = y_{n/2}(1-x_{n/2})/2$. The result is $y_n = y_{n/2} + y_{n/2}(1-x_{n/2})/2$. So, in order to obtain a refined result, the correction is added to the approximate result $y_{n/2}$. To do this, a subtraction and a multiplication should be taken. Moreover, due to the fact that ε_1 and ε_x have the zeroed most significant bits, then the multiplication is performed at half bit width. The resulting algorithm looks like the following:

```
x[0] = x; y[0] = x;
for (i = 0; i < n/2; i++) {
        q  = xi + 2^(-i+1)*x[i] + 2^(-2i)*x[i];
        if   (q < 1.0)  {x[i+1] = q; y[i+1] = y[i]  + 2^[-i]*y[i]; }
        else {x[i+1] = x[i]; y[i+1] = y[i];}
 }
y = y[n/2] + y[n/2]*(1.0 - x[n/2])/2;
```

In modern FPGA the two and three input adders have the same hardware volume and speed. So, the modified algorithm provides the speed-up approximately in four times comparing to the initial algorithm.

Experimental results. The derived algorithm was described by VHDL as the IP core. It was compiled for Xilinx Spartan-6 FPGA for various input and output data bit widths. Fig. 1 and Fig.2 show the relation of the hardware costs in the number of LUTs, and the maximum clock frequency on the bit width for the combinatorial and pipelined networks of this IP core, respectively. It should be noted that the modules with a bit width up to 32 additionally have a single multiplication block DSP48, and the rest of them have four such blocks.

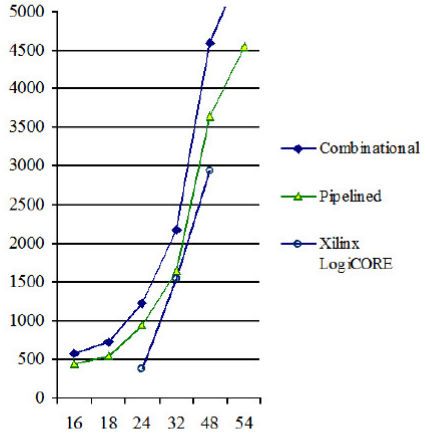


Figure 1. Hardware volume of the $\sqrt{x}$ IP core

For comparison, Fig. 1, 2 show the characteristics of the IP cores offered by Xilinx Inc. In general, the proposed IP core loses to the branded one. But its advantages are that it is free and it can be configured for arbitrary input and output bit width, and for any FPGA type. In addition, the proposed IP core has a lower latent delay.

For example, for 24 bits, its latent delay is only 15 cycles versus 24 cycles of the competitor. This means that when implementing the floating-point calculations, the proposed module provides greater performance.

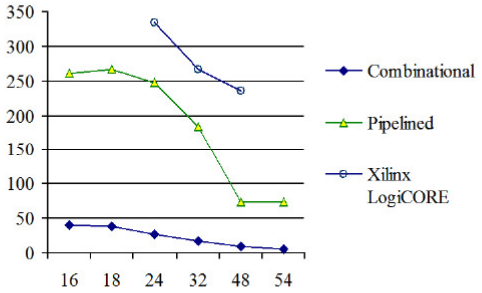


Figure 2. Clock frequency, MHz, of the $\sqrt{x}$ IP core depending on the bit width

Conclusion. The modified CORDIC-like algorithm for deriving the square root function is proposed. The algorithm is distinguished by the minimized number of steps, which is proportional to the given data and result bit width. The algorithm is described in VHDL and is intended for the FPGA implementation. It is the most effective during its implementation in the floating point square root module.

References. **1.** R. Woods, J. McAllister, G. Lightbody, and Y. Yi "FPGA-based Implementation of Signal Processing Systems"J. Wiley and Sons, Ltd., Pub. 2008. 364 p. **2.** "FPGA Implementations of Neural Networks". A. R. Omondi, and J. C. Rajapakse, Eds. Springer. 2006. 360 p. **3.** K. Yoshikawaa, N. Iwanagaa, and A.Yamawaki "Development of Fixed-point Square Root Operation for High-level Synthesis". Proc. 2nd Int. Conf. on Industrial Application Engineering. 2014. pp. 16 - 20. **4.** T.C. Chen "Automatic computations of exponentials, logarithms, ratios and square roots". IBM J. Res. and Develop. 1972. №4. pp. 380-388.