Efficient Polynomial Evaluation Algorithm and Implementation on FPGA

by

Simin Xu

School of Computer Engineering

A thesis submitted to Nanyang Technological University
in partial fullfillment of the requirements for the degree of
Master of Engineering

2013

Abstract

In this thesis, an optimized polynomial evaluation algorithm is presented. Compared to Horner's Rule which has the least number of computation steps but longest latency, or parallel evaluation methods like Estrin's method which are fast but with large hardware overhead, the proposed algorithm could achieve high level of parallelism with smallest area, by means of replacing multiplication with sqaure.

To enable the performance gain for the proposed algorithm, an efficient integer squarer is proposed and implemented in FPGA with fewer DSP blocks. Previous work has presented tiling method for a double precision squarer which uses the least amount of DSP blocks so far. However it incurs a large LUT overhead and has a complex and irregular structure that it is not expandable for higher word size. The circuit proposed in this thesis can reduce the DSP block usage by an equivalent amount compared to the tiling method while incurring a much lower LUT overhead: 21.8% fewer LUTs for a 53-bit squarer. The circuit is mapped to Xilinx Virtex 6 FPGA and evaluated for a wide range of operand word sizes, demonstrating its scalability and efficiency.

With the novel squarer, the proposed polynomial algorithm exhibits 41% latency reduction over conventional Horner's Rule for a 5^{th} degree polynomial with 11.9% less area and 44.8% latency reduction in a 4^{th} degree polynomial with 5% less area on FPGA. In contrast, Estrin's method occupies 26% and 16.5% more area compared to Horner's Rule to achieve same level of speed improvement for the same 5^{th} and 4^{th} degree polynomial respectively.

Acknowledgments

First of all, I feel tremendously lucky to have Professor Ian McLoughlin and Professor Suhaib Fahmy to be my supervisor. Professor Ian led me towards the field of fixed-point arithmetic and inspired me to further research on polynomial evaluation. Professor Suhaib greatly broadened my horizon in the area of reconfigurable hardware. I'd like to thank both of them for their kindness, patience, assistance and great advice.

My appreciation also goes to Eu Gene Goh, Mike Leary, Wilson Low, Susan John and Xilinx Asia Pacific (Singapore) for supporting me. Specially thank Wei Ting Loke for valuable technical discussions and comments on my research.

Last but not least, this thesis can't be finished without my family. Thanks my parents for always encouraging me and my wife for her endless love.

Author's Publications

S. Xu, S. A. Fahmy, and I. V. McLoughlin, "Efficient Large Integer Squarers on FPGA" to appear in Proceedings of the IEEE Symposium on Field programmable Custom Computing Machines (FCCM), Seattle, WA, May 2013.

Contents

List of Figures

List of Tables

List of Code

Chapter 1

Introduction

1.1 Motivation

Polynomials are commonly used in high-performance DSP algorithms to approximate the computation of functions, or to model systems parametrically. They are found inside many basic digital circuits including high-precision elementary functional evaluation circuits [1] and digital filters [2]. In fact, at a system level, many applications such as cryptography [3], speech recognition [4] and communications [5], involve the computation of polynomials.

The problem of polynomial evaluation has been investigated by many researchers, with the main research challenge usually being the speed of computation. The benefit of polynomials is that they only require multiplications and additions for the evaluation. However this does not mean that the computation is simple: High degree polynomial computation usually involve multiple large wordlength multiplications, which are time consuming operations. Therefore it is a common objective to reduce the computation time for polynomial evaluation in low latency applications, either through software or hardware approaches.

Earlier works focus on parallel schemes for software realization [6–13]. The parallelism achievable is highly dependent on the number of processing units available. In fact, most such works present results in terms of trade-off between latency and processing unit

cost. For example, Estrin's [12] method needs $2\lceil log_2(k+1)\rceil$ steps with $\lceil k/2\rceil$ processing units to evaluate a k^{th} order polynomial. Dorn [13] presents a solution with $\lceil log_2(k)\rceil$ processing units, where $\lceil log_2(k)\rceil + \lceil log_2(k+1)\rceil$ steps are needed to finish the computing. Several published solutions are closely tied to specific computer architectures, such as SIMD/MIMD and VLIW hardware, in [10, 11].

When software approaches are not able to meet performance requirements, alternative hardware implementations of polynomial evaluation algorithms are attractive, since arbitrary degrees of parallel computation can be easier to achieve. In general, many limitations restrict further improvement in software performance for general purpose processors. For example, processors usually support only a fixed operand wordlength and the number of arithmetic units is capped by the architecture of the processor design. Similarly, memory size and bandwidth or the cache size and register resources are usually limited too. With custom circuits, maximum parallelism can be achieved with lower cost in terms of overall hardware resources. This is partly due to the more flexible word length in the data paths and the fact that a more suitable architecture can be selected or customized. In fact, a number of publications present such approaches, implemented in VLSI [14, 15].

Another factor that influences the computation complexity is precision. Algorithms involving polynomial evaluation often have to consider the accuracy of computation. The maximum error in the system can be broken down into two components, namely approximation error and evaluation error [16]. Higher degree polynomials with larger wordlength in applications like functional evaluation are necessary to reduce approximation error. For example, to achieve 14 bits accuracy, a 5^{th} degree polynomial is needed for approximating $f(x) = 1/(1+x)$ within the range of x between $(0, 1)$ [17]. A 5^{th} degree polynomial with 64 bit operands is required to approximate the function $f(x) = log_2(1+2x)$ within the range of x between $(0, 2)$ [18]. However, this requirement often significantly increases

the computational complexity and latency. Therefore a certain tolerance of evaluation error may be allowed, and the designer could implement faithful rounding for coefficients, and perform truncation in intermediate computations while controlling the total error. Meanwhile, several other solutions have been proposed in the literature to reduce the complexity of polynomial evaluations based on the approximation requirement. Instead of using mathematical algorithms to evaluate polynomials, table based methods are fast and easy, and have been proposed for low degree polynomials in recent implementation [19]. However, this is only applicable to low precision design: to scale to large operand word length, the table size scales exponentially, which clearly becomes unsuitable for higher degree polynomials. On the other hand, polynomial degree can be reduced by carefully selecting the range to be approximated and the approximation methods [16, 20]. Post processing the coefficients derived from standard approximation methods with faithful precision loss can also simplify the polynomial. [20] presents a method to reduce the coefficient word length needed for polynomial approximation of a particular function, which can reduce the total circuit area by 40% and double the speed. [21] proposes a polynomial with sparse coefficients (several bits fixed to 0) which leads to 40% size reduction.

To satisfy both speed and high-precision computation requirements, reconfigurable hardware is increasingly being considered. In field programmable gate arrays (FPGA), a large amount of flexible hardware resources are available for parallelizing algorithms, with the further advantage of flexibility in the data path wordlength. By contrast, custom circuits are not feasible to be optimized for large range of coefficients on polynomials, as they are not re-programmable. Further more, implementing every polynomial algorithm with a dedicated custom circuit would obviously incur high development and engineering costs. Compared to custom integrated circuits, the cost of FPGA development is much lower, and this remains true even when amortized for moderate manufacturing volumes. Many designs with polynomial evaluation have been implemented in FPGA [17, 22, 23]

and naturally, various methods have been proposed to speed up polynomial evaluation methods in FPGAs [16, 20, 21, 24–26].

However, recent articles have only focussed on complexity reduction when implementing in FPGA. Interestingly, very few articles have described parallel evaluation algorithms implemented on modern FPGAs to boost the speed further and analyse the trade-offs on hardware overhead. To investigate parallel polynomial evaluation algorithm implementation on FPGA, algorithms proposed in the literature has been reviewed, and implemented Estrin's method on FPGA. To summarise the result, Estrin's method implementation yields half of the latency required by Horner's Rule. However, the hardware overhead is large, despite dedicated squarer circuits being used instead of multipliers whenever possible to reduce area. Nevertheless, this inspired the author to find a better optimized algorithm that utilize more squaring circuits in order to reduce the multiplication complexity. The resulting novel algorithm is proposed in Chapter 5.

Since the new algorithm relies on dedicated squaring circuits, the next question obviously concerns how good the squarer could be so as to improve the overall polynomial evaluation circuit.

For higher degree polynomial evaluation, the requirement for computation of a square occurs quite often, usually to calculate various exponential of x. It is common to implement this type of squaring operation using a standard multiplier. This may be for reasons of resource sharing, generalization or simply convenience. However, a dedicated squarer can be significantly faster, consume less power, and be smaller than when a general purpose multiplier is used. In fact dedicated squarers are not only used for polynomial evaluations, but also widely adopted in fixed point computations [27] and in various floating point arithmetic circuits [28–30]. Many techniques have been proposed in the literature to improve the performance of squaring circuits for ASIC solutions [31–34]. In reconfigurable hardware, [35] has presented 22% area savings in terms of configurable

logic blocks with 36.3% shorter delay and 45.6% less power when implement in Xilinx 4052XL-1 FPGA compared to an array multiplier on the same FPGA. With the newer DSP enabled FPGAs, squaring units have also been explored in various articles [36–38]. In order to fulfil the requirement for this work (i.e. an efficient squaring circuit for use within a polynomial evaluator), methods to realize a squarer more efficiently on FPGA have been further investigated and a novel efficient squarer in has been proposed Chapter 6.

1.2 Scope Definition

The problem of polynomial evaluation can be summarized as follow. Using a general format for k^{th} degree polynomial evaluation,

$$f(x) = \sum_{i=0}^{k} a_i x^i \qquad \text{(Eq. 1.1)}$$

The fixed point numbers x is defined as the input of the polynomial with a set of coefficients a_i of constant value. These coefficients can be obtained by various algorithms. In most of the system and within the consideration here, the coefficients will not be changed frequently although they could be updated from time to time. In other words, a system for which the *computation using* fixed a_i is the limiting factor, rather than the *computation of* a_i will be considered only.

Although the coefficients can be modified due to different approximation accuracy requirements or different ranges of input x, one particular set of coefficients could be used for long strings of different input x and do not change frequently with respect to the data in most of system implementations [15]. Such a system that could fulfil the requirement is commonly used in many applications, thus the discussion here is meaningful.

The range of the coefficients and the input x is defined to be $[0, 1]$. This is mainly for the ease of accuracy analysis performed later.

1.3 Contributions

1.3.1 Novel Polynomial Evaluation Algorithm

We propose a novel algorithm in this thesis to address the problem of polynomial evaluation in a fast manner with low hardware cost. The algorithm parallelizes the evaluation process by grouping different monomials. It then transforms each group into its vertex format which trades multiplication with square and utilizes the dedicated squarer to reduce the hardware cost. The result in this thesis shows that it is able to achieve short latency while maintaining minimum hardware cost. The general form of the novel design has been provided as well.

1.3.2 Comparison of Fixed Point Polynomial Evaluation Algorithms Implementation on FPGA

We implement the novel algorithm in FPGA as well as two algorithms from the literature to compare the performance and hardware utilization using two real applications. The applications are to approximate elementary functions. In the comparison, only the polynomial evaluation algorithm is different among the three designs and they are designed to fulfil same requirements. We will show the post-place and route results to prove that our theoretical findings, and mathematical analysis are applicable to real designs.

1.3.3 Novel Integer Squarer Design on FPGA

We have proposed a dedicated squarer design for implementation on FPGA, not only for our novel polynomial evaluation algorithm but for other uses as well. The design uses fewer DSP blocks while trading DSP blocks with limited hardware overhead, and shows better efficiency in terms of overall hardware utilization in the FPGA than competing designs. Considering DSP blocks are limited in modern FPGAs, the novel design is useful to reduce the overall DSP blocks usage and enable larger designs on single FPGA, while not demanding excessive amounts of additional logic.

1.4 Organization of the Thesis

Below is the organization of this thesis.

In Chapter 2, polynomial evaluation algorithms in the literature will be analyzed and various optimization methods proposed in recent work are described and discussed.

In Chapter 3, modern FPGA architectures are reviewed and multipliers based on these features are discussed.

In Chapter 4, conventional squarer designs and their FPGA implementation are presented.

In Chapter 5, the novel polynomial evaluation algorithm is proposed and is compared with the algorithms in the literature.

In Chapter 6, a novel squarer design is proposed, along with FPGA implementation details, and its performance will be compared with conventional designs.

In Chapter 7, the novel polynomial evaluation algorithm is implemented for real applications, and compared with conventional algorithms operating under the same conditions.

Chapter 8 concludes the thesis with discussion on possible improvements that could be considered in future.

Chapter 2

Conventional Polynomial Evaluation Algorithms, Implementation and Optimization

2.1 Polynomial Evaluation Algorithms

Polynomial evaluation procedures have been the subject of investigation since the 1950s. Mainly based on alternative computer architectures, many different software approaches have been proposed in the literature to optimize or accelerate the evaluation process. Most of the research focuses on the trade-off between general purpose processing units and system-level parallelism. A few prominent methods which have been published previously are presented and discussed below.

2.1.1 Direct Method

Directly computation of polynomials is by far the simplest method to describe. This involves computing exponentials of x (i.e. x^i in the general form) and multiplying each power with the corresponding coefficient a_i. This is most often used for low degree polynomial evaluation [39]. The computing process can be highly parallel, if the resources permit, but the total number of steps to complete the computation is large. More specifically, that a large number of multiplications are required. Meanwhile, the resultant

accuracy is relatively low [17]. Therefore it is much less commonly used for higher degree polynomials.

2.1.2 Horner's Rule

Horner's Rule is the most basic and common rule for computing polynomials. It was developed in the very early days of computation and it is widely used in numerous complex applications [40, 41]. Horner's Rule has been proven by Pan [6] and Winograd [22] to involve the minimum number of steps to evaluate a particular polynomial, i.e. it is optimum in terms of computational steps. Furthermore, it also has a regular structure which is easy to be implemented. All these factors lead to widespread adoption of the methods since it was invented. The rule involves the transformation of the polynomial into a series of multiply-adds. Considering the polynomial equation stated in (Eq. 1.1), Horner's Rule re-writes the formula into

$$f(x) = \{...((a_k \cdot x + a_{k-1}) \cdot x + a_{k-2}) \cdot x + ... + a_0\} \qquad \text{(Eq. 2.1)}$$

One multiply-add unit is used to compute $a_i \cdot x + a_{i-1}$ and this can be reused recursively to complete the overall computation. Therefore, it is an ideal method for processors containing multiply-add arithmetic (or multiply-accumulate) units. However, Horner's Rule is inherently a serial process and consequently the critical path for this method is long.

2.1.3 Parallel Methods

For low latency designs, especially in communication or cryptography applications, parallel processing is desirable. A few such methods are suitable for these applications with specific computer architectures. Dorn [13] proposes a generalized form of Horner's rule, which can be applied on general computing architectures which have more than one

computational unit. For a k^{th} degree polynomial, this method can reach a n^{th} degree parallelism, where $n \leq \lceil log_2(k) \rceil$. The general format is,

$$f(x) = \sum_{m=0}^{n-1} p_m \cdot (x^n) \cdot x^m \qquad \text{(Eq. 2.2)}$$

where

$$p_m(x) = \sum_{i=0}^{log_2 k} a_{m+jn} \cdot x^j \qquad \text{(Eq. 2.3)}$$

The latency boost for Dorn's method is obvious. Compared to the original Horner's Rule, approximately $1/n$ the amount of evaluation time is needed, and the only serial process involved is to calculate the power of x. The trade-off of this method is circuit area, with hardware requirements naturally depending upon the degree of parallelism. In Dorn's original paper, the parallelism that the system can achieve is based on the available number of processing units.

Another well-known method which parallels the evaluation process was proposed by G. Estrin in 1960 [12]. In his famous paper discussing the future of computer system architecture, he presented a method with a sequence that uses multiple multipliers to increase the speed of function evaluation. The general format reformed after (Eq. 1.1) could be shown as below

$$f(x) = \sum_{i=0}^{n} (a_{2i+1}x + a_{2i}) \cdot x^4 + (a_{2n+2} \cdot x^{2n+2}) \cdot (k - 2n - 1) \qquad \text{(Eq. 2.4)}$$

where $n = [k/2]$. An implementation of computer architecture for this algorithm was presented in the original paper as well (details of it can be found in the graph given in the appendix [12]). Each sub-sequence can occupy one processor and another processor can be used to evaluate the powers of x. The proposed architecture can complete the computation within $2\lceil log_2(k+1) \rceil$ steps. Obviously the main trade-off is also the hardware cost. To achieve the maximum level of parallelism, $\lceil k/2 \rceil$ processors are needed

for Estrin's design. This has been used by various applications [17, 42] for fast parallel evaluation of polynomials.

Other than the two methods mentioned above, tree structures approaches have been presented in previous works [7–9]. These methods are discussed in the content of ultra higher degree polynomial computation. Different folding schemes are used to find the maximum degree of polynomial that could be evaluated by a particular given method in a certain number of steps with the aim being to achieve the minimum number of steps required to evaluate a polynomial. Thus, when a higher degree polynomial is broken into smaller segments, the sub groups still follow the folding scheme that can fully utilize the schemes capability. Tables are shown in the original articles of such degrees that individual schemes could achieve, but which are not included here due to limited space. As these methods only tend to manifest their advantages for polynomials of degree greater than 20, they are not the main comparison reference in this thesis and thus it is not going to discuss further details on these methods.

Some previous works have also presented designs targeting alternative computer architectures for specific purposes. For example, special grouping schemes are presented in [10] based on MIMD and SIMD computers. In [11], polynomials are evaluated using a modern multimedia processors with a MAC unit and parallel execution scheduling. As they are optimized for a particular architecture rather than being generalised, they are also not considered in this thesis.

2.2 Polynomial Evaluation Implementation

Polynomial evaluations are not only implemented in software, which is limited by the computer architecture. Many hardware approaches also adopt similar methods when building application specific circuits. Horner's Rule is widely used in many hardware designs [20, 43]. Dorn's method has been implemented by Burleson [15] in ASIC, where

arithmetic units are not limited by architecture. For the sake of comparison, simplified circuit diagrams will be used here to illustrate Horner's Rule, Dorn's method and Estrin's method. The circuit is broken down into macro blocks of multiplier, squarer and adder. Area and performance metric of polynomial evaluation algorithms will be discussed on the level of these macro blocks. As processing resources are virtually unlimited in custom circuits, only the maximum degree of parallelism of a certain method is going to be considered. A 5^{th} degree polynomial is used as an example, which can be written as

$$f(x) = a_5 \cdot x^5 + a_4 \cdot x^4 + a_3 \cdot x^3 + a_2 \cdot x^2 + a_1 \cdot x + a_0 \qquad \text{(Eq. 2.5)}$$

and this will also be used to compare against the proposed method later. Meanwhile, for the purpose of performance comparison in theory, a few assumptions are made below.

Assumption 1 *Define the following Macros used in theoretical comparison of polynomial evaluation,*

Parallel multiplier	M_{mul}
Parallel squarer	M_{sq}
Carry propagate adder	M_{add}

and these Macros have the following properties:

$$T_{mul} > T_{sq} >> T_{add}$$

$$A_{mul} >> A_{sq} >>> A_{add}$$

In the assumption, T_{mul} is defined as the latency for an optimum pipelined parallel multiplier and T_{add} is defined as the latency of an optimum pipelined adder. T_{sq} will be the notation for the latency of an optimum pipelined parallel squarer, and which should ideally be lower than that of T_{mul}. A will be the notation used to describe the area of an individual macro and the a parallel multiplier is much lager than a parallel squarer in similar architecture. Obviously, the adder is the smallest and fastest macro among three.

2.2.1 Horner's Rule

Using Horner's Rule, the polynomial in (Eq. 2.5) can be transformed into,

$$f(x) = ((((a_5 \cdot x + a_4) \cdot x + a_3) \cdot x + a_2) \cdot x + a_1) \cdot x + a_0 \qquad \text{(Eq. 2.6)}$$

The evaluation process for implementing (Eq. 2.6) can be illustrated in the graph shown in Figure 2.1. In Figure 2.1 and subsequent figures of circuit diagram for polynomial

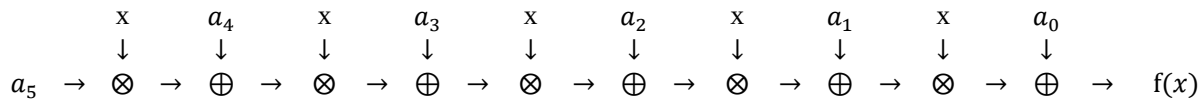

Figure 2.1: Diagram for evaluating 5^{th} degree polynomials using Horner's Rule.

evaluation, sequential operators will be denoted from left to right, and dependences denoted with arrows. Using Horner's Rule, 10 steps of computation are needed, which are namely five multiplications followed by five additions each. As there is no parallel execution, the latency of the evaluation largely depends upon how well the multiplier and adder have been designed. The overall latency using Horner's rule is $5 \cdot (T_{mul} + T_{add})$. Although, through proper pipelining, the multiplication and the addition can reach high throughput with a fast system clock rate, the latency is still relatively long due to the existence of longer pipeline stages. Table 2.1 summarizes the implementation details for Horner's Rule

Latency	Multiplier	Adder
$5 \cdot (T_{mul} + T_{add})$	5	5

Table 2.1: Latency and hardware macro count for evaluating 5^{th} degree polynomials using Horner's Rule.

2.2.2 Dorn's Method

Dorn's method can be used to achieve 3^{rd} degree parallelism on the same polynomial. The format of Dorn's equation to achieve this is shown below,

$$f(x) = (a_2 + a_5x^3) \cdot x^2 + (a_1 + a_4x^3) \cdot x + (a_0 + a_3x^3) \qquad \text{(Eq. 2.7)}$$

The whole equation can be subdivided into three subexpressions which can be computed in parallel. The cube of x can be shared among all three subexpresssions and thus should be computed upfront. It is obvious that the evaluation process is more complex than Horner's Rule (and in general the additional complexity grows with polynomial order). The evaluation process can be summarized in Figure 2.2 which shows that 12 steps are necessary, of which 2 steps are needed to compute x^3, 6 steps in parallel compute the three subexpressions, 2 steps are used to multiply with the respective power of x for the first two subexpressions, and 2 steps to sum all three parts together. Note that the last addition has to wait for both operands to be ready. With some degree of parallelism where the help of dedicated squarer, the latency of Dorn's method is shorter, but it is at the expense of additional hardware overhead. When this is implemented, the hardware

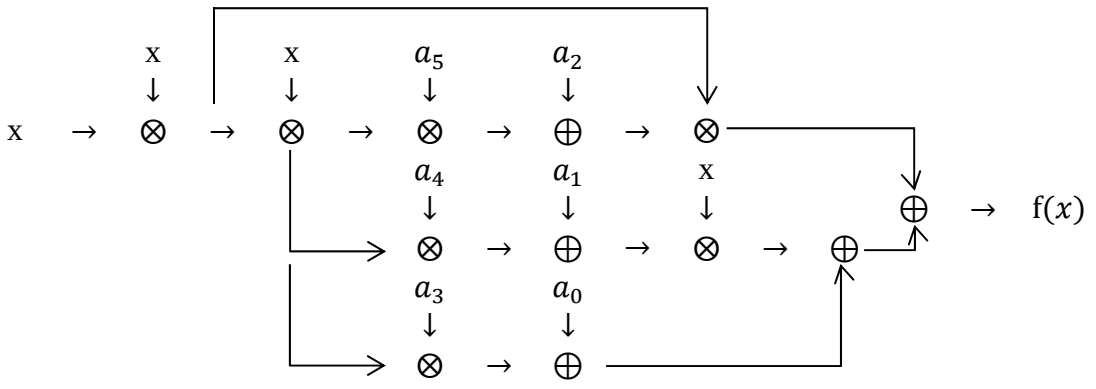

Figure 2.2: Diagram for evaluating 5^{th} degree polynomials using Dorn's method.

resource summary is as shown in Table 2.2, where one squarer and one multiplier is

for x^3(although one may argue a dedicated cubic operator could also be used in this case, it is outside the scope of discussion in this thesis), three multipliers followed by three adders are for three subexpressions, one multiplier is to multiply x with the second subexpression, one multiplier is to multiply x^2 with the first subexpression and two adders are for the final addition.

Latency	Multiplier	Squarer	Adder
$T_{sq} + 3 \cdot T_{mul} + 3 \cdot T_{add}$	6	1	5

Table 2.2: Latency and hardware macro count for evaluating 5^{th} degree polynomials using Dorn's method.

2.2.3 Estrin's Method

The converted format for the same polynomial using Estrin's method is,

$$f(x) = (a_5 \cdot x + a_4) \cdot x^4 + (a_3 \cdot x + a_2) \cdot x^2 + (a_1 \cdot x + a_0) \quad \text{(Eq. 2.8)}$$

Similar to Dorn's method, three subexpressions of the equations are able to be computed individually by one multiplier followed by one adder. Another two multipliers with two adders are used to multiply subexpressions with power of x and sum them together. However, different from Dorn's method, the even powers of x should be computed first, i.e. x^2 followed by $(x^2)^2$, which shall be implemented using two squarers. The total number of evaluation steps is still 12 but it has a shorter latency than Dorn's method. The critical path, is determined by the first subexpression in (Eq. 2.8). It contains the path of two squaring and one multiplication, which is longer than the other two paths in parallel. The evaluation process can be summarized graphically as shown in Figure 2.3 and the critical path is shown in bold arrow. The amount of hardware resource is given in Table 2.3. Estrin's method yields an improvement over Dorn's method in terms of latency at the same hardware cost and it will be selected as the parallel evaluation method to be compared with proposed method later.

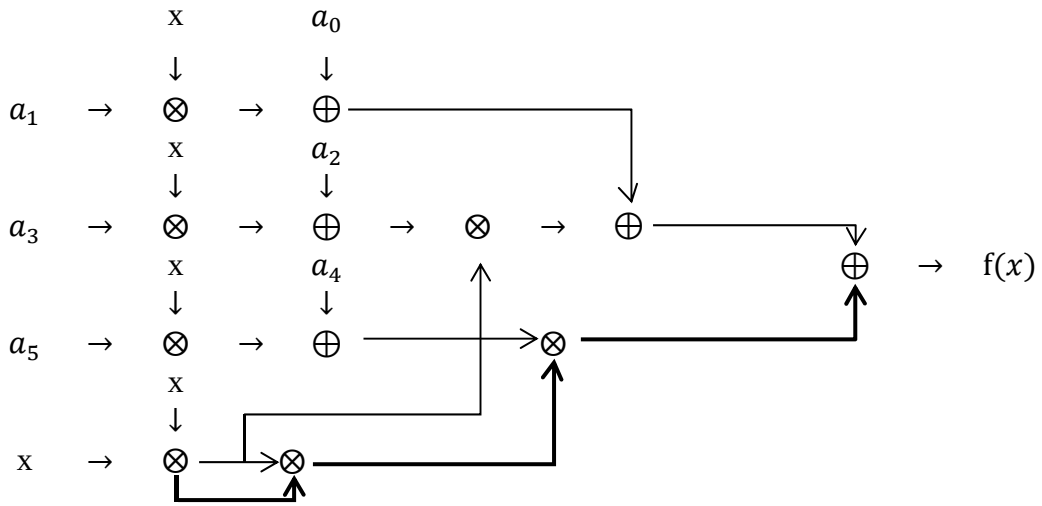

Figure 2.3: Diagram for evaluating 5^{th} degree polynomials using Estrin's method.

Latency	Multiplier	Squarer	Adder
$2 \cdot T_{sq} + T_{mul} + T_{add}$	5	2	5

Table 2.3: Latency and hardware macro count for evaluating 5^{th} degree polynomials using Estrin's method.

2.3 Polynomial Evaluation Optimization

With the flexibility of hardware circuits, designers are able to not only build the evaluation circuit with custom arithmetic operators, but also customize the polynomial evaluation based on the precision requirement and available hardware components (such as implementation in FPGAs). In function approximation, procedures to derive an optimized polynomial have been proposed [16]. The framework, involving range reduction and bit-width optimization, is able to produce a design with simplified polynomials. [20] and [43] have presented similar methodologies, which focus on reducing the complexity of the polynomials as well.

The degree of polynomial used to approximate the function is largely determined by the range of the function and it directly links to the area, latency and throughput of the overall design. It is shown that the area could be half and the latency could be only

1/10 when range reduction is performed [16]. On the other hand, if designs have to be approximated across the whole range, sub-intervals could be divided. Although more memory might be needed for breaking down into segmented ranges, it could significantly reduce the degree required for the same precision [26]. Figure 2.4 shows the function before and after range reduction optimization and Figure 2.5 illustrats how the sub-intervals are formed.

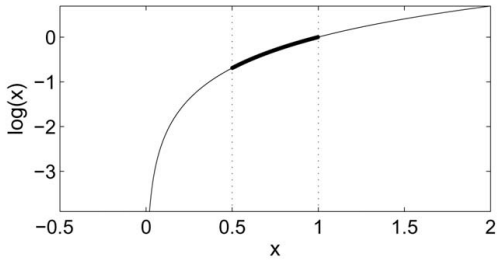

Figure 2.4: Function with range reduction

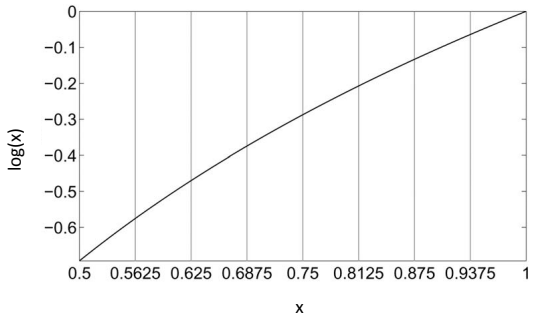

Figure 2.5: Function with sub-intervals

Rounding the coefficients to the nearest sweet spot numbers within the bound of evaluation error allowance is another way to simplify the evaluation process. It is obvious that custom adders and multipliers would be more efficiently fit with these rounded coefficients, rather than general purpose arithmetic units. Other coefficient optimization methods, like in [21,25], have been proposed to make the multiplier smaller with more '0's inside the multiplier operand, which could make one or more rows of partial products in

the multiplier to be constant '0' and therefore size of multiplier can be shrunk to increase the speed as well as to reduce area.

As design automation is used in these frameworks, the processing time to generate polynomial evaluator is negligible compared to the computation of constrained approximation [43]. What's more, the process is usually performed once for a particular polynomial and the results can be reused over and over (at least until the underlying FPGA or implementation architecture changes). Therefore, it is often worthwhile to use these methods to simplify the polynomial evaluation problem, if they are able to map to the required architecture.

Chapter 3

Conventional Arithmetic Implementation on FPGA

3.1 FPGA Background

Field Programmable Gate Array (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects [44]. In general, FPGAs are more flexible than ASICs as they are able to programmed easily to desired functions or applications, with the emphasis on the ease of reprogrammability. This is the feature that makes such devices suitable for building processing units for polynomials which are likely to have to adapt to parameter changes from time to time. The fundamental building block of an FPGA is its logic cells. Despite the different hardware used to realize the logic cell functions, and different input widths provided by various FPGA vendors, they can be mapped to certain logic functions with the help of the synthesis and mapping tools. Xilinx Virtex-4 FPGA cells contain a 4 input look up table (LUT), which can also be used for RAMs or as shift registers [45]. While in more recent Virtex-6 and 7 series FPGAs, 6 inputs LUTs are provided, to give more flexibility [46]. Other than the LUT, multiplexers, flip flops and carry chains are available as basic items of reconfigurable hardware. A typical CLB that contains all these basic elements is shown in Figure 3.1. Altera provide similar architectures in their basic

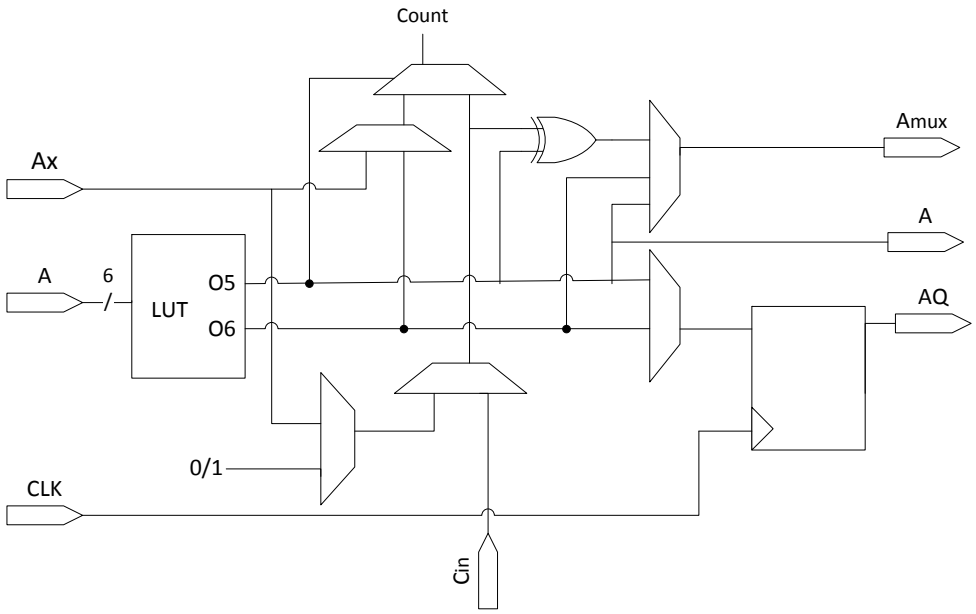

Figure 3.1: Simplified diagram of LUT, multipliexer, flip flops and carry chains in CLB in Xilinx Virtex 6 FPGA.

reconfigurable blocks, although they are based on 4 input LUTs. In order to interconnect the CLBs with each other, interconnect blocks have been designed to route different blocks, and these are programmed automatically by place and route tools.

In recent modern FPGA architectures, more functional blocks are added to optimize applications. Reconfigurable DSP blocks enable fast digital signal processing. In the Virtex 6 architecture they contain a 25×18 bit signed multiplier with a programmable ALU following its multiplication data path [47]. This extension of wordlength is an improvement from the DSP blocks found in Virtex 4 and previous devices, and was first supported in the Virtex 5. The simplified DSP48E1 block diagram is shown in Figure 3.2. The DSP block includes an optional pre-adder at one of its inputs. The ALU unit can be programmed as a 48 bit adder, which can accumulate the multiplier product and the adjacent DSP result through the dedicated 48 bit routes in a single clock cycle. The dedicated routes are from the PCOUT ports of the adjacent DSP to the PCIN ports of the current one. This bus also has an option to perform a 17 bit right shift. The

input of the adder can be the 48 bit number from input C with an optional associated CARRYIN, but it is mutually exclusive with the PCIN signals. The DSP48E1 has also includes internal pipeline registers, allowing it to run at a frequency of up to 450 MHz (for -1 speed grade Virtex 6 devices) [47]. However, as DSP blocks are combined for larger operands, further pipelining is required, which is implemented using LUTs and flip flops in the Slices.

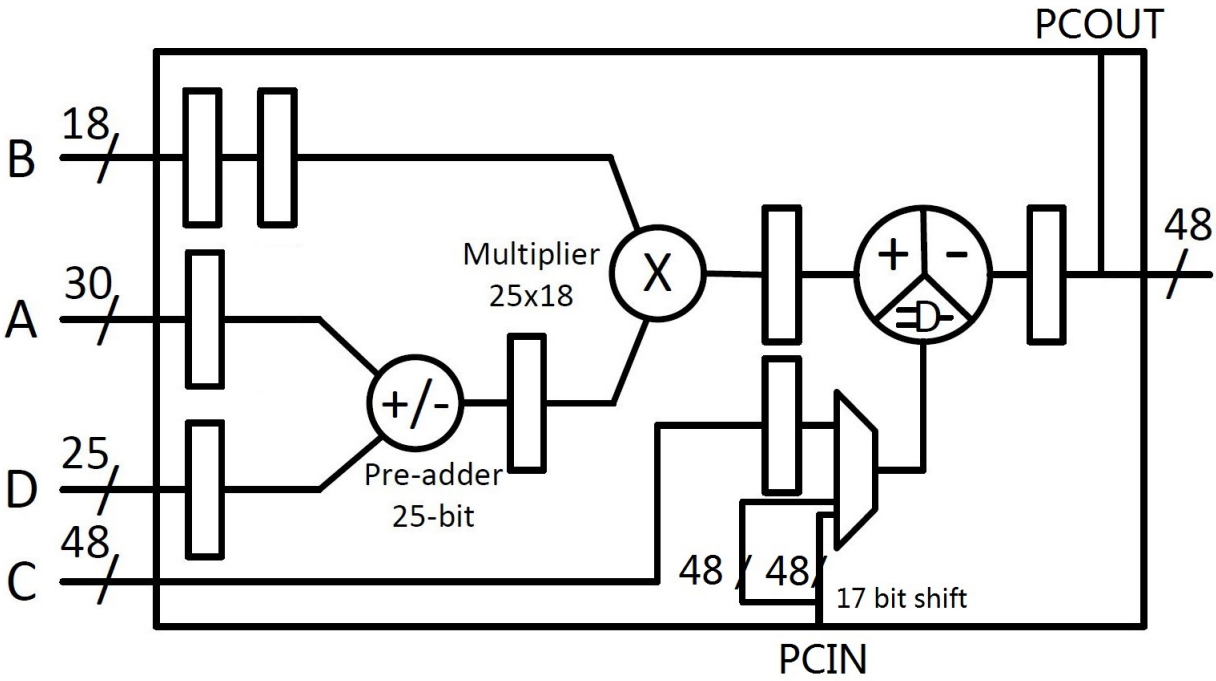

Figure 3.2: Simplified diagram of DSP48E1 in Xilinx Virtex 6 FPGA.

Block RAM has also been included in modern FPGAs to enable large memory implementations. Each RAM block, named RAMB36E1 in the current device, is 36Kb in size and can be configured as 64K×1, 32K×1, 16K×2, 8K×4, 4K×9, 2K×18, 1K×36 or 512×72. Both read and write operations require one clock cycle.

3.2 Parallel Multiplier in FPGA

3.2.1 Multiplier Based on LUT

The architecture of parallel multiplier involves partial product generator and adder tree. For example, a 4×4 bit parallel multiplier with inputs a and b will first generate 16 partial products shown in Figure 3.3.

The partial product array could be reduced using adder tree into one row of sum and one row of carry, which could be added using carry propagate adder at last. Parallel

$$
\begin{array}{cccccccc}
 & & & a_3 & a_2 & a_1 & a_0 & \\
 & & & b_3 & b_2 & b_1 & b_0 & \\
\hline
 & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 & \\
 & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & & \\
 & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & & \\
a_3b_3 & a_2b_3 & a_1b_3 & a_{30}b_3 & & & &
\end{array}
$$

Figure 3.3: Partial product alignment of 4×4 bit parallel multiplier.

multipliers have been explored for many years and thousands of designs have been implemented in hardware. Booth encoding schemes, Wallace tree [48], Baugh and Wooley's method [49] and many more algorithms have been introduced to do fast parallel multiplication in custom circuits. These circuits can be implemented in FPGAs with little difficulty, even without specialized DSP functional block being provided. Several works have proposed multiplier designs purely using LUTs in [50, 51].

3.2.2 Multiplier Based on Cascaded Chains of DSP Blocks

In modern FPGAs where DSP blocks are available, multiplication can utilize these functional blocks to reduce latency, rather than relying on LUTs. The cascaded chain through dedicated routes allows users to build large wordlength multipliers running at high speed. This decomposition is as follows: take a 35×35 bit multiplication $x \cdot y$ as an example.

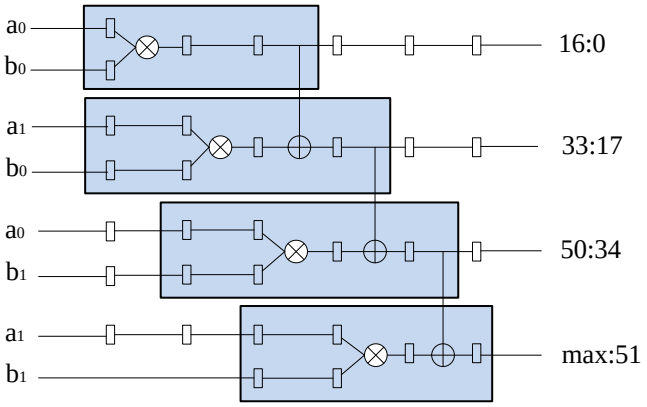

Figure 3.4: Pipeline schematic of general purpose multiplier.

Each of the inputs can be split into two sub operands with smaller wordlength:

$$x = a_0 + a_1 \cdot 2^{17} \quad \text{(Eq. 3.1)}$$

$$y = b_0 + b_1 \cdot 2^{17} \quad \text{(Eq. 3.2)}$$

Therefore, the full multiplication can be re-written as,

$$x \cdot y = a_0 b_0 + (a_1 b_0 + a_0 b_1) \cdot 2^{17} + a_1 b_1 \cdot 2^{34} \quad \text{(Eq. 3.3)}$$

which contains four sub products. Note that each operand in these four sub multiplications is equal to or less than 18 bits (17 bits unsigned numbers for lower splits a_0 and b_0 and 18 bits two's complement signed numbers for the most significant parts a_1 and b_1). After the sub multiplications are computed, the products can be added through the post-adder, one by one, within 4 cycles. No extra LUTs are needed for the addition in this case. Figure 3.4 shows a detailed implementation.

As either a_1 or b_1 can be at most 25 bits in size, if an operand exceeds 35 bits, for example 42 bits, then $a_1 b_1$ is not able to fit within one DSP and a_1 must be split into sub operands of 17 bits and 8 bits respectively. In this case, an additional DSP would be chained after the last DSP to perform the multiplication of the 8 msbs of a_1 with b_1.

When the wordlength is greater than 43 bits, further splits are required to complete the addition, and correspondingly more clock cycles being needed to obtain the result. For a given wordlength w where k splits are needed, the number of DSP blocks needed for a standard multiplier is:

$$f(k) = \begin{cases} k^2 & \text{if } w - 17k > 1 \\ k^2 + 1 & \text{if } w - 17k \leq 1 \end{cases} \quad \text{(Eq. 3.4)}$$

It can be seen that, as the wordlength increases, the number of DSP blocks required by the multiplier grows quadratically. We will revisit this point later when demonstrating that it is much more efficient to perform squaring using a specialized squarer, especially as wordlength becomes large. This will be discussed more fully in Chapter 4.

The standard cascaded method has been provided as a IP core from Xilinx CoreGen. It has been optimized to run at the maximum speed of the DSP, however, it is limited to a maximum wordlengh of 64 bits.

3.2.3 Multiplier with Fewer DSP Blocks

For complex and computationally intensive algorithms, large multipliers could be designed with fewer DSPs. One method is to use Karatsuba-Ofman algorithm [52]. Classic Karatsuba-Ofman algorithm reduces the multiplier complexity by exchanging multiplication with addition and it was extended by [36] to reduce DSP blocks usage for multiplier. A two split example is shown as,

$$\begin{aligned} x \cdot y &= a_1 \cdot b_0 + a_0 \cdot b_1 \\ &= a_1 \cdot b_1 + a_0 \cdot b_0 - (a_1 - a_0) \cdot (b_1 - b_0) \end{aligned} \quad \text{(Eq. 3.5)}$$

Noted that three multiplications, instead of four are needed in (Eq. 3.5). It is reported in [36] that it could reduce the DSP blocks usage from 4 to 3 in a 34×34 bit multiplier at the cost of 68 additional LUTs and from 9 to 6 in a 52×52 bit multiplier at the cost of 312 additional LUTs.

The other method, which is more efficient, is to arrange the DSP in the form of "tiling" where 25 bits input is available in DSP blocks. [36] has also reported a 58×58 bit multiplier with 8 DSP blocks and 388 LUTs where non-standard tiling arrangement of DSP blocks is used. Due to the special tiling arrangement, asymmetric DSP blocks are fit into the 58×58 bit operand with only one small multiplier in the middle which must be implemented using LUTs. The diagram of such method is shown in Figure 3.5. In fact, there are many variations of "tiling like" multiplier design where operands of the multiplier are not equal. When this happens, for example a 48×34 multiplier, can easily be mapped to four DSP blocks, by dividing 48 bits into $24 + 24$ bits and dividing 34 bits into $17 + 17$ bits. Noted that with the tiling arrangement, it is not simple to chain the

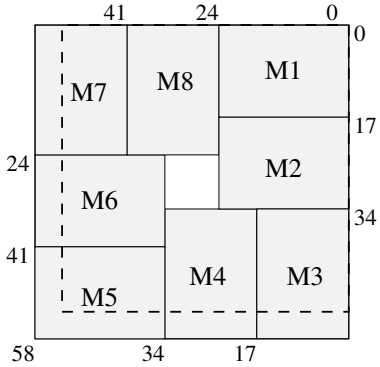

Figure 3.5: Tiling multiplier example

DSPs through dedicated routes and the adders are then implemented in LUTs.

More recently [53,54] have proposed advanced usages of such asymmetric DSP blocks in building large multipliers. They are none-pipelined design and thus not suitable for polynomial evaluators running in high speed.

Chapter 4

Conventional Squarer Design

It is common to implement a squaring operation using a standard multiplier. This may be for reasons of resource sharing, generalization or simply convenience. According to (Eq. 3.4), it is much more efficient to perform squaring using a specialized squarer when multiplier grows quadratically as wordlength increases The computation of a square is required in many DSP algorithms and thus for high performance applications, specialized hardware for squaring may be desired. A dedicated squarer can be significantly faster, consume less power, and be smaller than a multiplier and therefore they are widely adopted in fixed point function evaluation [27] or in various floating point arithmetic computations [28–30].

4.1 Squarer Designed for ASICs

In theory, a squarer can reduce the number of partial products by half compared to a parallel multiplier, as both inputs are the same. For example, Figure 4.1 can be easily dervied from Figure 3.3 with identical partial products identified in bold and thus they can be added with one bit left shift. There are many variations of squarer design proposed in the literature. In [51], 40% area reduction and 18.6% speed improvement is achieved compared to the multiplier in ASIC, which applies the folding of partial products (which can be seen in Fig. 4.1). Several other squarer designs have been reported to have even

			a_3	a_2	a_1	a_0
			a_3	a_2	a_1	a_0
			$\boldsymbol{a_3a_0}$	$\boldsymbol{a_2a_0}$	$\boldsymbol{a_1a_0}$	a_0a_0
		$\boldsymbol{a_3a_1}$	$\boldsymbol{a_2a_1}$	a_1a_1	$\boldsymbol{a_0a_1}$	
	$\boldsymbol{a_3a_2}$	a_2a_2	$\boldsymbol{a_1a_2}$	$\boldsymbol{a_0a_2}$		
a_3a_3	$\boldsymbol{a_2a_3}$	$\boldsymbol{a_1a_3}$	$\boldsymbol{a_0a_3}$			
a_3a_3		a_2a_2	$2a_0a_3$	a_1a_1		a_0a_0
	$2a_2a_3$	$2a_1a_3$	$2a_1a_2$	$2a_0a_2$	$2a_0a_1$	

Figure 4.1: Partial product alignment of 4×4 parallel squarer.

higher area reductions as well as speed improvements for ASIC solutions [31–33, 55]. In the context of FPGA implementation, [35] has reported a squarer with Wallace-tree and carry-select adder that requires 22% less LUTs, is 36.3% faster and enjoys a 45.6% power saving on Xilinx 4052XL-1 FPGA compared to a generalized multiplier.

4.2 Squarer Based on Cascaded Method

In a DSP enabled FPGA, it is more efficient to build squarers using DSP blocks wherever possible. A straightforward method for building large wordlength squaring circuits in FPGA is by cascading DSPs as is briefly discussed in [36]. For squaring, x and y are identical thus (Eq. 3.3) becomes:

$$x \cdot x = a_0^2 + (a_1a_0 + a_0a_1) \cdot 2^{17} + a_1^2 \cdot 2^{34} \qquad \text{(Eq. 4.1)}$$

Since the middle two terms in the bracket are the same, their summation can be simplified to a one bit left shift. The alignment of DSP blocks shown in Figure 4.2 acheives this, and has a similar pipeline to the general purpose multiplier. This alignment allows a DSP block to be eliminated compared to using a standard multiplier – and as the wordlength increases, more DSPs can be saved. Equations (Eq. 4.2) and (Eq. 4.3) below show that using this method, only 6 and 10 DSPs are needed for three splits and four splits operands respectively. This compares to 9 and 16 DSPs required, respectively, for

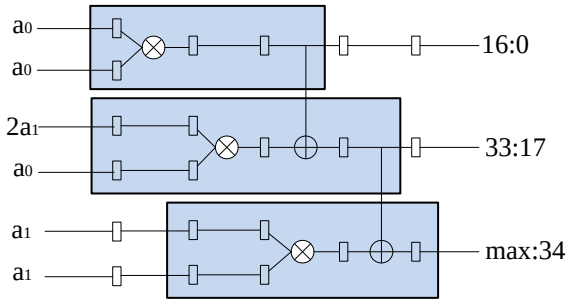

Figure 4.2: Pipeline schematic of squarer based on cascading DSP chains.

the general purpose CoreGen multiplier.

$$x \cdot x = a_0^2 + 2a_1a_0 \cdot 2^{17} + \left(a_1^2 + 2a_2a_0\right) \cdot 2^{34} + 2a_2a_1 \cdot 2^{51} + a_2^2 \cdot 2^{68} \quad \text{(Eq. 4.2)}$$

$$x \cdot x = a_0^2 + 2a_1a_0 \cdot 2^{17} + (a_1^2 + 2a_2a_0) \cdot 2^{34} + (2a_2a_1 + 2a_3a_0) \cdot 2^{51} + (a_2^2 + 2a_3a_1) \cdot 2^{68} + 2a_3a_2 \cdot 2^{85} + a_3^2 \cdot 2^{102} \quad \text{(Eq. 4.3)}$$

We can derive a similar equation to (Eq. 3.4) for the DSP count required for this method with w bits split into k parts as given in (Eq. 4.4). Note that the relationship between w and k is slightly different to that defined for the multiplier, for example, four splits are needed for 59 bits instead of three.

$$f(k) = \begin{cases} (k^2 + k)/2 & \text{if } w - 17k > 1 \\ (k^2 + k)/2 + 1 & \text{if } w - 17k \leq 1 \end{cases} \quad \text{(Eq. 4.4)}$$

In common with the multiplier, all the additions for the above squarer design lie within the boundary of DSP blocks so the only circuit elements outside the DSP blocks are the pipeline registers.

A similar rationale was used in [36] (targeting the Xilinx Virtex 4) for the FloPoCo project [30][1]. We have mapped the auto-generated FloPoCo squarer to the Virtex 6

[1]FloPoCo version 2.4.0, `http://flopoco.gforge.inria.fr/`

FPGA for comparison purposes. However, we have not been able to produce expected performance using ISE v13.4 as the design is not optimally pipelined for the newer DSP architectures. Hence, in the course of our work, we have also rewritten an optimized design for FloPoCo, based on the same decomposition and design principles, that we refer to as the "cascaded method". The implementation results of original FloPoCo and the cascaded method will be presented in Section 4 where they are compared to the novel method developed during the course of this research. We have adopted and extended this method to the Virtex 6 FPGA, yielding good results (which will be discussed later in Chapter 6).

4.3 Squarer Based on Non-standard Tiling Method

The cascaded method is a simple approach to build squaring circuits, but it still uses a larger number of DSP blocks than is necessary. To further reduce DSP usage, the tiling method [36] was proposed, applied to Xilinx Virtex 5 or newer FPGA devices where 25×18 bit DSP blocks are supported. The main approach is to achieve efficiency gains through maximizing the utilization of the asymmetrically sized DSP inputs. This is similar to what has been proposed for the multiplier in Chapter 3. Two different tiling methods were suggested for squaring circuits optimized for 53-bit operands; one of them can reduce the number of DSPs from 6 (cascaded method) to 5. Figure 4.3 shows the alignment of the 5 DSPs required to complete the square in this method. The main drawback to this approach is the potentially high LUT usage, as there are four small multiplications that must be implemented (two are shown in white in the middle top and bottom left, and two in dark grey colour). As pointed out by the original author of [36], "It is expected that implementing such equations will lead to a large LUT cost, partly due to the many sub-multipliers, and partly due to the irregular weights of each line (no 17-bit shifts) which may prevent optimal use of the internal adders of the DSP48E blocks." [36].

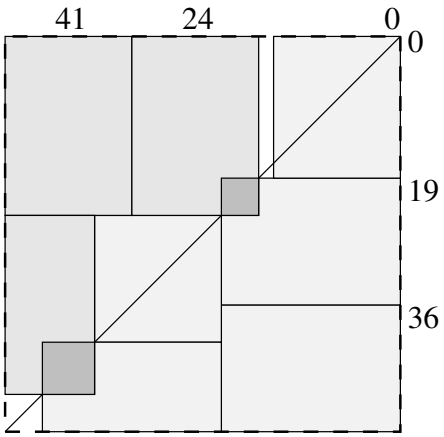

Figure 4.3: Tiling method for square

Although he did not provide full implementation details, we have reproduced the design based on Figure 4.3, which applies to operand widths between 43 and 53 bits, to compare with the other methods evaluated.

Chapter 5

Novel Polynomial Evaluation Algorithm

In this chapter, a novel polynomial evaluation algorithm is proposed. It involves transformation of the general form of polynomial into "square rich" format. The main benefit of the algorithm is to achieve high level of parallelism with minimum hardware cost. Although the total steps will be more than those used for Horner's Rule, the implementation in hardware will be even smaller than the "optimum method in theory", thanks to the efficiency gains in the use of squarer units. The latency of this method is similar to Estrin's method as described in the literature but uses much less hardware resource than the highly paralleled method. The algorithm can be applied to polynomials of arbitrary degree.

5.1 First Hypothesis

The novel method is based on the hypothesis below, beginning with a 2^{nd} degree example.

Hypothesis 1 *To evaluate the following 2nd order polynomial,*

$$f(x) = \sum_{i=0}^{2} a_i x^i \qquad \text{(Eq. 5.1)}$$

it is advantageous to use,

$$f(x) = a_2 \cdot ((x + m_2)^2 + n_2) \qquad \text{(Eq. 5.2)}$$

to evaluate the polynomial.

The new format completes the square and converts the original polynomial into its vertex form. In this equation (Eq. 5.2) m_2 and n_2 are coefficients derived from a_1 and a_0, thus they are considered as being known values. Therefore, four steps are needed to compute (Eq. 5.2), which are one addition, one squaring followed by another addition, and a final multiplication. The same number of steps are needed if Horner's Rule is used to compute the 2^{nd} degree polynomial, however, it requires two multipliers and two adders. The advantage of the proposed method is obvious in terms of hardware cost where one multiplier is shrunk to a squarer, even though the total number of steps is the same as for Horner's Rule.

5.2 Novel Method Example

The above method can be applied to general polynomial evaluation as will be illustrated in an example for the same 5^{th} degree polynomial in (Eq. 2.5),

Hypothesis 2 *A 5^{th} degree polynomial which can be expressed as,*

$$f(x) = \sum_{i=0}^{5} a_i x^i \qquad \text{(Eq. 5.3)}$$

can be evaluated using,

$$f(x) = a_5 \cdot x \cdot [(x^2 + m_5)^2 + n_5] + a_4 \cdot [(x^2 + m_4)^2 + n_4] \qquad \text{(Eq. 5.4)}$$

Similar to hypothesis 1, n_4, n_5, m_4 and m_5 are all considered to be known real values derived from the original coefficient set. To derive the above arrangement of terms, the original polynomial can be divided into two subexpressions,

$$f(x) = x \cdot (a_5 x^4 + a_3 x^2 + a_1) + (a_4 x^4 + a_2 x^2 + a_0) \qquad \text{(Eq. 5.5)}$$

If the method in hypothesis 1 is applied to complete the square in each bracket then the equation becomes (Eq. 5.4).

In this equation (Eq. 5.4), the total multiplication can be completed by employing three multipliers and three squarers. To evaluate using the new format, x^2 is calculated first to be shared among both subexpressions and to be added with a coefficient in each subexpression. This is subsequently followed by another square function and by an addition. Meanwhile, the computation of term $a_5 \cdot x$ should be completed and both subexpressions can then immediately multiply their coefficients. The final addition is performed using one adder. In terms of latency, the novel method could be as fast as $2 \cdot T_{sq} + T_{mul} + 3 \cdot T_{add}$. Figure 5.1 shows the evaluation sequence.

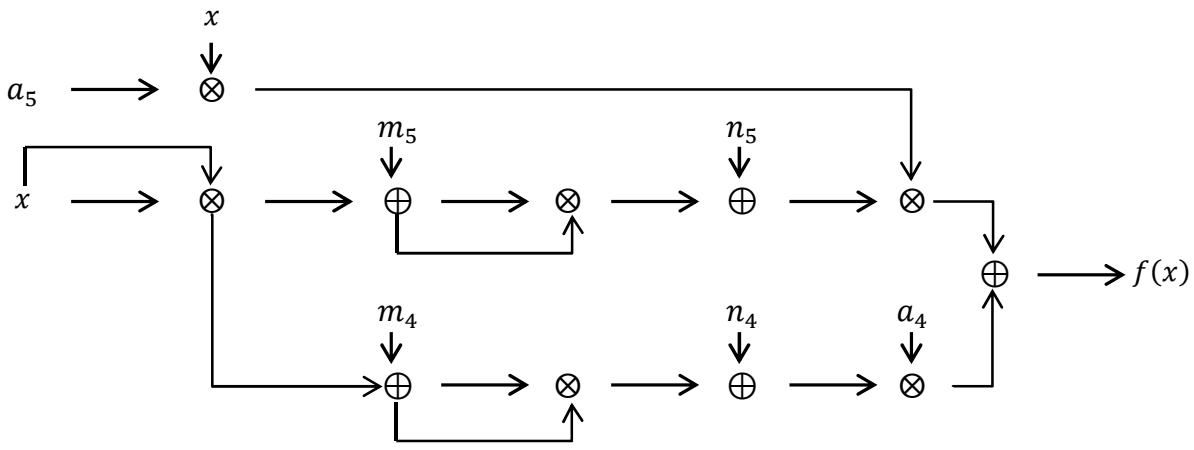

Figure 5.1: Diagram for evaluating 5^{th} degree polynomials using proposed method.

Table 5.1 compares the latency with conventional methods. It is shown to be similar to Estrin's Method (only two adders slower and the gap could be smaller in real application), and is much faster than the other two. The effective hardware cost, is summarized in Table 5.2. Compared to five multipliers and five adders used by Horner's Rule, the proposed method involves one more step but can achieve hardware savings if an efficient

Method	Latency
Horer's Rule	$5 \cdot (T_{mul} + T_{add})$
Dorn's Method	$T_{sq} + 3 \cdot T_{mul} + 3 \cdot T_{add}$
Estrin's Method	$2 \cdot T_{sq} + T_{mul} + T_{add}$
Proposed Method	$2 \cdot T_{sqr} + T_{mul} + 3 \cdot T_{add}$

Table 5.1: Latency for evaluating 5^{th} degree polynomials using proposed method vs. conventional methods.

Method	Multiplier	Squarer	Adder
Horner's Rule	5		5
Dorn's Method	6	1	5
Estrin's Method	5	2	5
Proposed Method	3	3	5

Table 5.2: Hardware macro count for evaluating 5^{th} degree polynomials using proposed method vs. conventional methods.

squarer exists (i.e. where a squarer circuit is more efficient than a general multiplier). The proposed method shows a tremendous advantage over the other two parallel methods, where the expense of parallelism is very much higher.

As mentioned, the advantage over Horner's rule presupposes the existence of an efficient squarer. If the design published in [51] is used, the requirement is met without doubt. In fact, a novel efficient squarer (on reconfigurable hardware) has been invented during the course of this research, which will be discussed in the Chapter 6. The combined benefit of the novel polynomial evaluation process using the novel squarer will be shown in the system implementation results for the overall polynomial evaluation process, presented later.

5.3 Generalized Format

The novel polynomial method presented here can, of course, be generalized to higher degree. The process of generalization will now be explained in detail. Consider a general k^{th} degree polynomial in (Eq. 1.1), Define an integer number j, which is the binary

logarithm of k:

$$j = \lceil log_2 k \rceil \tag{Eq. 5.6}$$

Then the equation can be divided into the following groups,

$$\begin{aligned} f(x) = &(a_k x^k + a_{k-j} x^{k-j} + a_{k-2j} x^{k-2j}) \\ &+ (a_{k-1} x^{k-1} + a_{k-j-1} x^{k-j-1} + a_{k-2j-1} x^{k-2j-1}) + ... \\ &+ (a_{k-j+1} x^{k-j+1} + a_{k-2j+1} x^{k-2j+1} + a_{k-3j+1} x^{k-3j+1}) \end{aligned} \tag{Eq. 5.7}$$

If $k - 3j + 1 > 0$, i.e. the polynomial is greater than 20^{th} degree, more elements shall be grouped after the above equation using the same general principle. Here, for reasons of space and the scope of the thesis, only examples for degree less than 20 are presented. Let us now define,

$$p_t(x) = (a_{k-t} x^{k-t} + a_{k-j-t} x^{k-j-t} + a_{k-2j-t} x^{k-2j-t}) \tag{Eq. 5.8}$$

Where, t ranges from [0, j-1]. In $p_t(x)$, a common divisor of $a_{k-t} x^{k-2j-t}$ must be extracted so that the equation becomes,

$$p_t(x) = a_{k-t} x^{k-2j-t} (x^{2j} + \frac{a_{k-j-t}}{a_{k-t}} x^j + \frac{a_{k-2j-t}}{a_{k-t}}) \tag{Eq. 5.9}$$

Applying the method proposed in Hypothesis 1 and then (Eq. 5.9) becomes,

$$p_t(x) = a_{k-t} x^{k-2j-t} (x^j + m_{k-t}^2) + n_{k-t} \tag{Eq. 5.10}$$

and

$$f(x) = p_0 + p_1 + ... + p_{j-1} \tag{Eq. 5.11}$$

Equation (Eq. 5.10) and (Eq. 5.11) are the general format for the proposed method to evaluate polynomials with arbitary degree.

5.4 Derivation of the Coefficient Set and Accuracy Concern

The proposed method needs a new set of coefficients, which are a one-off derivation from the original coefficients. For example, in (Eq. 5.4) the new set of coefficients would be computed from the following,

$$m_5 = \frac{a_3}{2a_5} \tag{Eq. 5.12}$$

$$m_4 = \frac{a_2}{2a_4} \tag{Eq. 5.13}$$

$$n_5 = \frac{a_1}{a_5} - \frac{a_3^2}{4a_5^2} \tag{Eq. 5.14}$$

$$n_4 = \frac{a_0}{a_4} - \frac{a_2^2}{4a_4^2} \tag{Eq. 5.15}$$

For an application such as an adaptive filter, this may result in an additional overhead every time the polynomial adapts, however the evaluation process can be performed when deriving the polynomial. Even when the coefficients are computed on the fly, compared to the approximation process, this overhead is small and tends to zero as the number of evaluations performed using the new coefficient set becomes large. In the system considered in this thesis, the coefficients are all pre-stored in RAM when developing the RTL and this overhead will not need to be accounted for. In fact, this is considered to be a typical real-world scenario – where the polynomial is generated during system setup or programming, rather than on the fly.

In such real systems, especially when fixed point format processing is used for computation, the new set of coefficients might result in a larger evaluation error than if the original set was used. This important consideration will be discussed as part of the system implementation discussions, where guard bits may be needed to maintain the accuracy.

One has to note that in certain conditions, when n_5 and n_4 requires large wordlength to keep precisions on both a_1 and a_0, extremely large multiplications might be incurred.

This is not desired as it will penalize the performance gain. An alternative equation could be used instead of (Eq. 5.4) and the new format is shown in (Eq. 5.16).

$$f(x) = [a_5 \cdot (x^2 + m'_5)^2 + n'_5] \cdot x + a_4(x^2 + m'_4)^2 + n'_4 \quad \text{(Eq. 5.16)}$$

where,

$$m'_5 = \frac{a_3}{2a_5} \quad \text{(Eq. 5.17)}$$

$$m'_4 = \frac{a_2}{2a_4} \quad \text{(Eq. 5.18)}$$

$$n'_5 = a_1 - \frac{a_3^2}{4a_5} \quad \text{(Eq. 5.19)}$$

$$n'_4 = a_0 - \frac{a_2^2}{4a_4} \quad \text{(Eq. 5.20)}$$

The alternative equation has the same hardware cost compared to hypothesis 2 shown in Table 5.3. The penalty is one more multiplication stage inserted in the critical path for polynomial whose degree is higher than 4 (but only one even when degree goes up). This will increase the latency for polynomial evaluators, however, it may not be significant in real-world and it will be discussed in the later chapters.

The alternative equation is used in Chapter 7 to account for the problem in the real application. It will compare the latency between novel method and Estrin's method in both 4^{th} and 5^{th} degree polynomial, where the later one has slightly longer latency in the novel method compared to Estrin's method.

Method	Multiplier	Squarer	Adder
Alternative Equation	3	3	5
Proposed Method	3	3	5

Table 5.3: Hardware macro count for evaluating 5^{th} degree polynomials using alternative equation vs. proposed methods.

5.5 Motivation of Squarer and Reconfigurable Hardware Implementation

implemention the novel method in reconfigurable hardware, specifically FPGAs, rather than ASIC has been completed. It is not cost effective to build a polynomial evaluation unit in ASIC for each individual application, where different applications need to be optimized based on different precision, area and performance requirements. However, it is convenient to build the design in FPGA, where data paths are highly customizable and the development duration and financial cost are acceptable at the same time. In the modern FPGAs, where more dedicated functional blocks such as DSPs and BRAMs are available, it becomes much more efficient to map such application specific processing units onto FPGA.

The efficiency of the novel method largely depends on how well the squarer is designed. Conventional squarers mainly rely on the benefits of folding the partial products, compared to the situation in the parallel multiplier. When targeting to implement the system on FPGA, exploration of various squarer structures has been done and it has been decided to build an efficient squarer based on modern FPGA architecture, proposed in the subsequent chapter.

Chapter 6

Novel Squarer Implementation on FPGA

6.1 Proposed Design

In this chapter we propose an optimised squaring algorithm implementation on FPGA with lower DSP block cost. The algorithm is flexible with respect to operand size, though it targets higher wordlength, since (as will be shown later) it improves on existing methods when operand sizes are equal or larger than 42 bits.

The algorithm was inspired by the same Karatsuba-Ofman algorithm.However, the same reduction in [36] does not apply for squaring. According to equations in [56], there are no more advantages in using the reformation if both inputs are the same. We have thus modified the classic Karatsuba-Ofman algorithm, only applying specific parts of the process. By rearranging the squaring equation using our proposed algorithm, functions performed by the DSP blocks can be mapped to a small number of LUTs. As wordlength increases, more DSPs can be exchanged with LUTs, and the LUT count grows only gradually. We demonstrate the algorithm using two cases in the following.

6.1.1 Squarers for Three Splits Input

We will take a 52×52 bit squarer to illustrate the algorithm, where equation (Eq. 4.2) was the equivalent squaring equation using the cascaded method. We now define m and

n as,

$$m = a_2 - a_1 \tag{Eq. 6.1}$$

$$n = 2a_0 - a_1 \tag{Eq. 6.2}$$

and their product as,

$$m \cdot n = 2a_2a_0 - 2a_1a_0 - a_2a_1 + a_1^2 \tag{Eq. 6.3}$$

This can be rewritten to provide the term $a_1^2+2a_2a_0$ (which was in the middle of (Eq. 4.2)) as,

$$a_1^2 + 2a_2a_0 = mn + 2a_1a_0 + a_2a_1 \tag{Eq. 6.4}$$

Now substituting (Eq. 6.4) into (Eq. 4.2) gives the following,

$$\begin{aligned} x \cdot x = {} & a_0^2 \\ & + 2a_1a_0 \cdot 2^{17} \\ & + (mn + 2a_1a_0 + a_2a_1) \cdot 2^{34} \\ & + 2a_2a_1 \cdot 2^{51} \\ & + a_2^2 \cdot 2^{68} \end{aligned} \tag{Eq. 6.5}$$

In the formulation of (Eq. 6.5), we can see that only 5 DSP blocks are required. The DSPs are configured to compute the sub products a_0^2, $2a_1a_0$, a_2a_1, mn and a_2^2 (named DSPs p_0 p_1 p_2 p_3 and p_4). The repeated terms will be added using LUTs or post-adders in the DSP blocks. Compared to the 6 DSP blocks needed in the cascaded method, one fewer DSP block is required (16.7% saving), matching the DSP block savings of the tiling method (also note that DSP reduction becomes greater as the operand word size increases).

The main benefit of this transformation is the reduced cost for the replacement logic: The overhead in the algorithm is two 17 bit adders needed to compute m and n, and

adders to accumulate those sub products that cannot be added using the post-adders in the DSP blocks. Compared to the tiling method, which requires small multiplies and complex additions, a careful mapping of the adders in our algorithm can significantly reduce the extra LUT overhead. Specifically, for Virtex 6 FPGAs, it only requires one 17 bit adder using LUTs to work out one of m or n, since the other can be computed using the pre-adder in DSP block p_3. On the other hand, the remaining addition has to be mapped efficiently with two cascaded chains which can be found from (Eq. 6.5). The two chains are,

$$chain_0 = 2a_1a_0 + a_2a_1 \cdot 2^{17} + mn \cdot 2^{17} \quad \text{(Eq. 6.6)}$$

$$chain_1 = 2a_1a_0 + 2a_2a_1 \cdot 2^{17} \quad \text{(Eq. 6.7)}$$

$chain_0$ is the output directly from DSPs p_3, p_2 and p_1 with the cascaded addition chain. $chain_1$ can be computed in the form of

$$2a_1a_0 + a_2a_1 \cdot 2^{17} + a_2a_1 \cdot 2^{17} \quad \text{(Eq. 6.8)}$$

which can be easily derived from p_2 and p_1 in LUTs as well. During the same cycle of finishing $chain_0$, the second chain finishes the addition so that they are summed up in the following cycle. The whole process requires one 34-bit adder to obtain $chain_1$ and one 51-bit adder to add the two chains. Figure 6.1 shows the overall implementation schematic of this squaring circuit with pipeline registers.

As a result, both additions for lsb and msb of the squarer are performed separately instead of in the cascaded chain. The 17 bit addition for the lsbs can be implemented in LUTs and the msbs can make use of the post adder in DSP p_4, by using its input bus C as well as the input CARRYIN. Consider that the input timing for input bus C and CARRYIN are longer than the dedicated routes, the signals are further pipelined inside the DSP block by enabling CREG and CINREG. Compared to the dedicated routes

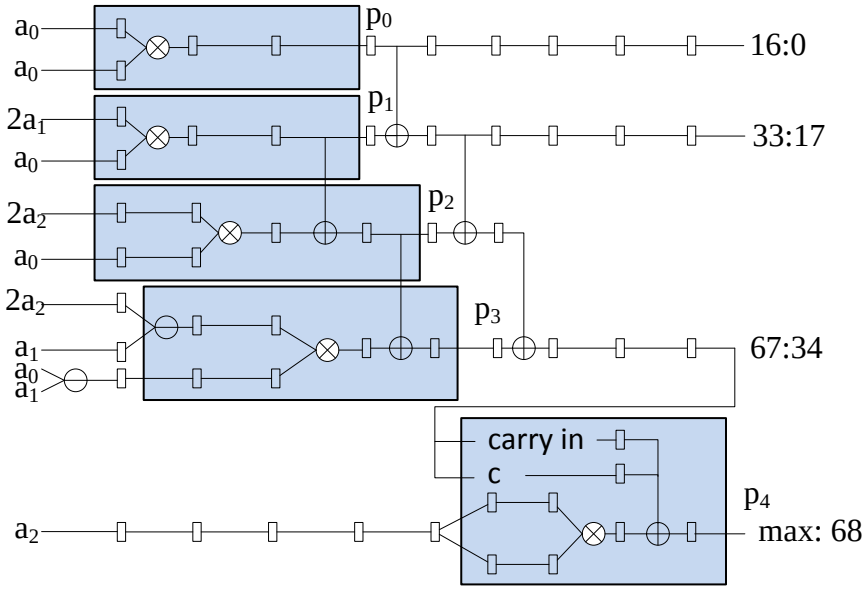

Figure 6.1: Pipeline schematic of squarer for three splits input.

between DSPs, the routes from DSP to Slice are slower and buses of registers are used to capture the data from the DSP before being manipulated in the FPGA fabric. This increases the latency by one cycle. With the pipeline design shown in Figure 6.1 having a total latency of 9, the system clock speed can be maintained at nearly the maximum frequency of the DSP, which is 450 MHz for the target device.

The new algorithm can be applied to longer wordlengths. When operands are above 53 bits, not only does a_2^2 need two DSPs limited by the input width, but $m \cdot n$ cannot fit into one DSP as well, due to both operands exceeding 18 bits. Therefore a minor change is required to the definition of m and n by making,

$$m = 2a_2 - a_1 \tag{Eq. 6.9}$$

$$n = a_0 - a_1 \tag{Eq. 6.10}$$

This now ensures that n is only 18 bits and hence (Eq. 6.4) gives,

$$a_1^2 + 2a_2a_0 = mn + a_1a_0 + 2a_2a_1 \tag{Eq. 6.11}$$

Therefore, (Eq. 6.5) becomes,

$$\begin{aligned} x \cdot x = a_0^2 & \\ & + 2a_1a_0 \cdot 2^{17} \\ & + (mn + a_1a_0 + 2a_2a_1) \cdot 2^{34} \\ & + 2a_2a_1 \cdot 2^{51} \\ & + a_2^2 \cdot 2^{68} \end{aligned} \qquad \text{(Eq. 6.12)}$$

Consequently, the relations between the two chains will change from deriving $2a_1a_0 + 2a_2a_1 \cdot 2^{17}$ based on $2a_1a_0 + a_2a_1 \cdot 2^{17}$, to the opposite, requiring a wider adder for the computation.

Specifically for the 53-bit squarer, since a_2 is 19 bits, a_2^2 requires only one DSP with an adder made from LUTs. For both designs above, the architecture is similar to Figure 6.1 and they are shown in the appendix.

6.1.2 Squarer for Four Splits and More

When operands are above 59 bits, the inputs have to split into four parts. Take 64×64 bit as an example. Here, m and n are defined in the same manner as in the case of three splits and the term $a_1^2 + 2a_2a_0$ can be represented by (Eq. 6.11). Similarly, p and q can be defined as,

$$p = 2a_3 - a_2 \qquad \text{(Eq. 6.13)}$$

$$q = a_1 - a_2 \qquad \text{(Eq. 6.14)}$$

and term $a_2^2 + 2a_3a_1$ can then be written as,

$$a_2^2 + 2a_3a_1 = pq + a_2a_1 + 2a_3a_2 \qquad \text{(Eq. 6.15)}$$

Hence, the squaring equation becomes,

$$\begin{aligned} x \cdot x = a_0^2 & \\ & + 2a_1a_0 \cdot 2^{17} \\ & + (mn + a_1a_0 + 2a_2a_1) \cdot 2^{34} \\ & + (2a_2a_1 + 2a_3a_0) \cdot 2^{51} \\ & + (pq + a_2a_1 + 2a_3a_2) \cdot 2^{68} \\ & + 2a_3a_2 \cdot 2^{85} \\ & + a_3^2 \cdot 2^{102} \end{aligned} \quad \text{(Eq. 6.16)}$$

Equation (Eq. 6.16) reveals that only 8 DSPs are needed to compute the square. We enumerate the DSPs from p_0 to p_7 for a_0^2, a_1a_0, a_2a_1, mn, $2a_3a_0$, pq, $2a_3a_2$ and a_3^2 respectively. The method saves 2 DSPs over the cascaded equivalent (a 20% saving). In contrast, the tiling method does not propose a solution for operands this large.

Clearly, as operand size increases, the additional circuitry needed to add the partial products is more complex. To utilise the post-adder more efficiently, three chains of DSPs are defined, which are

$$chain_0 = a_1a_0 + a_2a_1 \cdot 2^{17} \quad \text{(Eq. 6.17)}$$

$$chain_1 = chain0 + mn + 2a_3a_0 \cdot 2^{17} + pq \cdot 2^{34} \quad \text{(Eq. 6.18)}$$

$$chain_2 = a_2a_1 + 2a_3a_2 \cdot 2^{17} \quad \text{(Eq. 6.19)}$$

The equation (Eq. 6.16) could be represented using these chains as,

$$\begin{aligned} x \cdot x = a_0^2 & \\ & + 2chain_0 \cdot 2^{17} + chain_1 \cdot 2^{34} \\ & + chain_2 \cdot 2^{51} + chain_2 \cdot 2^{68} \\ & + a_3^2 \cdot 2^{102} \end{aligned} \quad \text{(Eq. 6.20)}$$

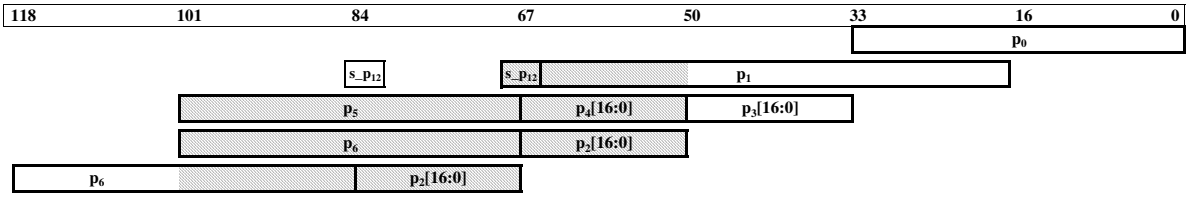

Figure 6.2: Adder alignment of squarer for four splits input.

It is interesting that a_2a_1 is used in both $chain_0$ and $chain_2$, which is not possible to achieve by using the dedicated routes from PCOUT to the adjacent PCIN in both cases. However, we can use the C inputs as an alternative way to connect the post-adder with the previous DSP instead. In fact, we implement this in the addition for $chain_0$. When $chain_0$ is used to form $chain_1$, the results from that DSP p_1 can then be added inside the next DSP p_3 through dedicated routes with no shift. In this way, only DSP p_0 and p_2 have the post-adder bypassed. There is one limitation in this arrangement: a 4-bit adder is needed on top of the post-adder to complete the 51 bit addition, due to the wordlength limitation of the C input of the DSP. This small adder, implemented in LUTs, takes the carry from the p_1 output and adds the 4 msbs from p_2. The equation (Eq. 6.16) can be then represented using p_0 to p_6, which is,

$$\begin{aligned} x \cdot x = & \; p_0^2 \\ & + 2\left\{s\_p_{12}, p_1\right\} \cdot 2^{17} \\ & + \left\{p_5, p_4\left[16:0\right], p_3\left[16:0\right]\right\} \cdot 2^{34} \\ & + \left\{p_6, p_2\left[16:0\right]\right\} \cdot 2^{51} \\ & + \left\{p_6, p_2\left[16:0\right]\right\} \cdot 2^{68} \\ & + s\_p_{12} \cdot 2^{81} \\ & + a_3^2 \cdot 2^{102} \end{aligned} \qquad \text{(Eq. 6.21)}$$

s_p_{12} in (Eq. 6.21) is the 4 bit adder which is defined as

$$s\_p_{12} = p_1[47] + p_2[33:30] \qquad \text{(Eq. 6.22)}$$

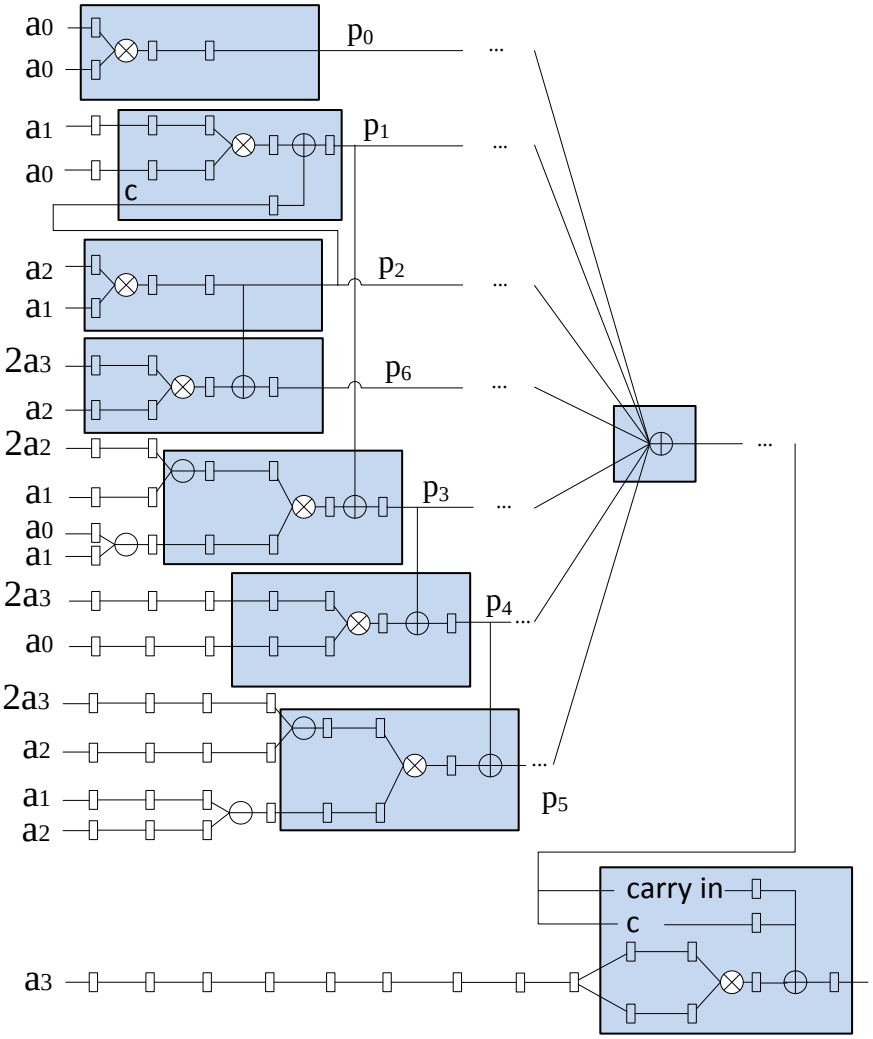

Figure 6.3: Pipeline schematic of squarer for four splits input.

The last DSP performs the final addition similar to the last DSP in the 52×52 bit example. Other than the last adder, the rest are implemented in LUTs, which are aligned as shown in Figure 6.2.

The grey area in Figure 6.2 is implemented using a 3:1 compressor, which only uses one LUT per bit by the default mapping method in ISE. However, based on our experiments, this becomes the critical path in the design. The schematic of a 1 bit 3:1 compressor can be found in [57], where two functions are shared in one LUT. Limited by the $O5$ to $A/B/C/DMUX$ delay in a particular Slice, which is a compulsory path if the logic is combined in one LUT, the negative setup slack is around 20% of the minimum period of the DSP, regardless of speed grade. Hence, we map the 3:1 compressor into two LUTs per bit instead to avoid the shared logic overhead. In this way, the $O6$ to $A/B/C/D$ paths are available for each logic function in individual LUTs, which is much faster than the $O5$ to $A/B/C/DMUX$ paths. Overall, six pipeline stages are designed to finish the addition (one box in Figure 6.3) and the total latency is 13 cycles. With this pipeline design, the system can sustain the 450 MHz clock rate.

The approach for even higher wordlengths is similar to the above examples and the DSP count for higher wordlength can be as low as:

$$f(k) = \begin{cases} (k^2 - k + 4)/2 & \text{if } w - 17k \geq 1 \\ (k^2 - k + 4)/2 + 1 & \text{if } w - 17k < 1 \end{cases} \qquad \text{(Eq. 6.23)}$$

where $k > 2$. Note that the reduced DSP block usage, compared to the standard cascaded method, increases much slower, resulting in higher gains with extremely large operands. With carefully aligned adders and formations of DSP chains, the extra LUT overhead can be constrained to be relatively small compared to the number of DSP blocks saved, however the exact implementation details of higher numbers of splits beyond four are not presented here for reasons of space.

Size	CoreGen	FloPoCo	Cascaded	Tiling	Proposed
42–52	9	6	6	5	5
53	10	6	7	5	5
54	10	6	7	N/A	6
55–58	10	7	7	N/A	6
59	10	10	10	N/A	8
60–64	16	10	10	N/A	8

Table 6.1: DSP block usage for all methods.

6.2 Implementation Results

The proposed algorithm has been synthesised, placed and routed using ISE version 13.4, targeting the Xilinx Virtex 6 XC6VLX240T-1 FPGA. A design generator has been built to expand the algorithm across input operand wordlengths from 42 to 64 bits, inclusive. Both three splits (42 to 58 bits) and four splits (59 to 64 bits) are supported. The cascaded squarer designs as well as the general purpose multiplier designs provided by Xilinx CoreGen have been implemented for comparison across the same range of wordlengths. Similarly, squarers from the FloPoCo tool have been compiled as another reference point. The tiling method has been built for wordlengths between 43 and 53 bits.

Table 6.1 shows the total number of DSPs used in each algorithm. The proposed algorithm shows a gain across wordlengths and consumes up to 50% fewer DSP blocks compared to a general purpose multiplier. The tiling method has the same DSP count but only applies to size between 43 bits and 53 bits. Across most of the range, the proposed method uses 14.3 to 20% fewer DSP blocks than the cascaded method and FloPoCo squarers.

To quantify the total cost among different methods, we compute an equivalent LUT cost to represent one DSP block. This is computed by taking the total number of LUTs in the device and dividing by the total number of DSP blocks. For the given FPGA, which has 150720 LUTs and 768 DSP blocks, this equals 196 (it ranges from 160 to 240

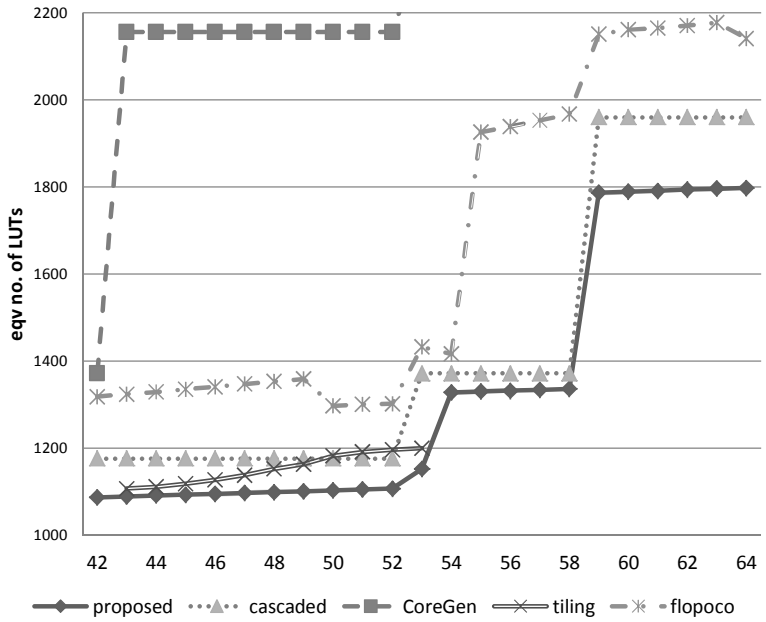

Figure 6.4: Equivalent LUT usage for all methods against operand word length.

for most general purpose Xilinx Virtex series FPGAs). This metric is used to quantify the trade-off between DSP blocks and LUTs.

Figure 6.4 shows the equivalent cost for all the implemented designs for the range of wordlengths. Compared to the cascaded method and the design from FloPoCo, both the tiling method and the proposed method show an advantage in terms of total equivalent LUTs. For the 53-bit squarer, which is the optimal size for the tiling method, both methods save two DSP blocks over the cascaded method but the proposed method uses 21.8% fewer LUTs to achieve this. Between 43 and 52 bits, where both methods use 5 DSPs, the tiling method uses 127 to 216 LUTs to replace each DSP compared to 107 to 127 LUTs for the proposed method. This translates to up to a 41.2% improvement over the tiling method for the given operand word sizes. It is not possible to show the equivalent number of LUTs for the general purpose multiplier on the same axis, as for high wordlength the cost becomes as high as 3528 LUTs.

The proposed method is flexible in terms of operand size without significant efficiency

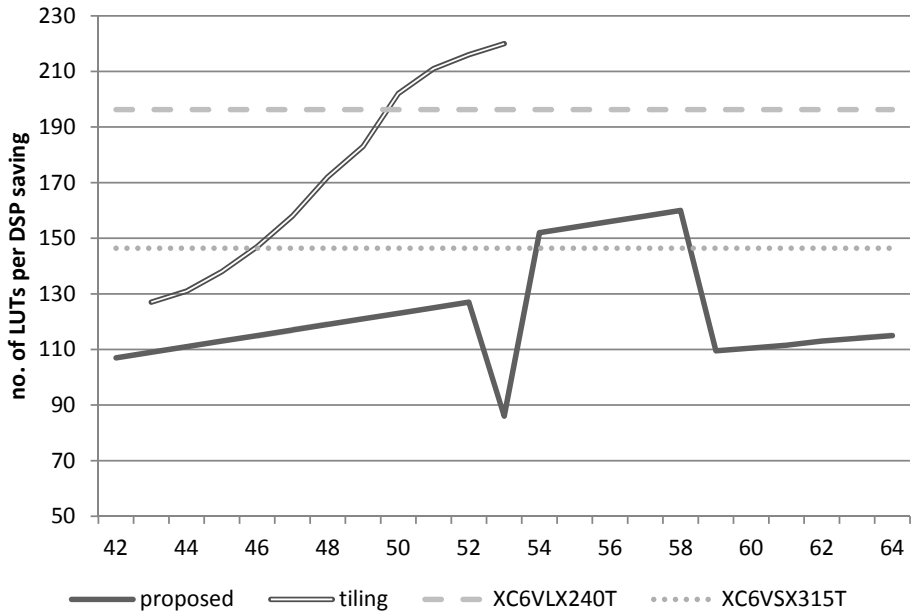

Figure 6.5: LUTs per DSP saved (from the cascaded method) ratio for tiling and proposed methods against operand word length.

penalties. Figure 6.5 shows the LUTs required per saved DSP compared to the cascaded method for the proposed method and the tiling method. Two horizontal lines show the LUTs:DSP block ratio for the target device and for the Xilinx Virtex6 XC6VSX315T which is a DSP-rich FPGA. The tiling method reaches an architectural limit for replacing DSP with LUTs above a wordlength of 49 bits. In contrast, even for the DSP rich FPGAs, where a DSP block is worth as few as 146 LUTs, it is still worthwhile to use the proposed algorithm to reduce the DSP count for a large range of wordlenghts. The trend in the graph repeats every 17 bits as for between 42 and 59 bits, when more DSPs can be saved compared to the cascaded method.

Figure 6.6 plots the maximum post place and route frequency of the evaluated methods. The proposed method, along with the cascaded method and the CoreGen multiplier are able to sustain speed as operand wordlength increases. The tiling method, due to different synthesized results for the small LUT multipliers, exhibits wide speed variations as operand size increases. For the FloPoCo squarer, the speed drops significantly at larger

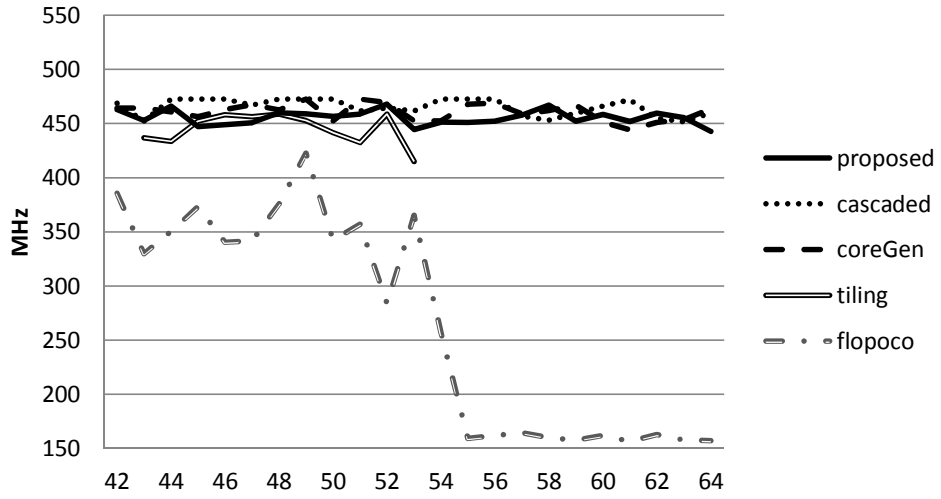

Figure 6.6: Maximum frequency for all methods against operand word length.

wordlengths due to the pipeline design adopted by the method not being optimized for the Virtex 6 architecture.

Chapter 7

Implementation of Polynomial Evaluator on FPGA

We've built the polynomial evaluator on the targeted Xilinx Virtex 6 FPGA (XC6VLX240T-1) as well, synthesized, placed and routed using ISE v13.4 with default settings. We leverage much of the work done by FloPoCo project to generate the test cases for polynomial evaluator using its fixed point function evaluator module (Figure 7.1), while replacing the polynomial evaluator core with alternative methods, shown in shaded rectangular in Figure 7.1. Two function evaluators are built in this thesis, one is a 5^{th} degree polynomial used to approximate the function $sin(x \cdot \pi)$ within the range of $x \in [0, 1]$, and the other is a 4^{th} degree polynomial used to approximate the function $log_2(\frac{x}{2} + 1)$ within the same range. We define both input and output of the polynomials to be 52 bit fixed point

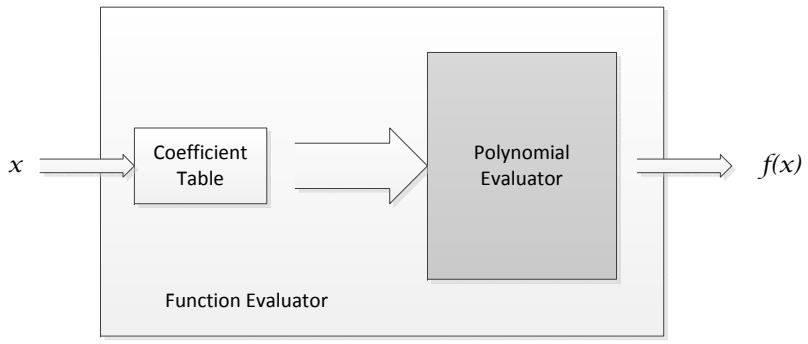

Figure 7.1: System diagram of fixed point function evaluator.

number. Any new coefficients will be derived based on the coefficients generated from FloPoCo tool to maintain the approximation error. Squaring are implemented using novel method. The intermediate steps are truncated to reduce the unnecessary computation complexity with only guard bits to ensure that the evaluation error is within requirement. The design is also optimized to fit the size of primitive in the targeted FPGA. The pipeline design of the alternative methods follows the FloPoCo default strategy in pipelining adders and multipliers except for DSP instance. For the same reason stated in Chapter 6, DSP block pipeline is done by instantiate Xilinx primitive with optional registers turned on by attributes (this applies to FloPoCo design as well). Though the design can't reach maximum frequency of DSP block in the end, it is a more fair comparison among different methods. The implementation results are presented to compare the hardware cost and performance of all three methods.

7.1 Function Evaluator in FloPoCo

In FloPoCo project, a fixed point function evaluator can be generated by using the following command.

flopoco FunctionEvaluator function wI wO degree

The function evaluator approximate the function using Sollya tool [1]. Sollya is a tool targeted to the automatized implementation of mathematical floating-point libraries and it offers a fast Remez algorithm to generate polynomials for approximation. A simple range reduction is performed which consists of splitting the input intervals into 2^i sub-intervals. For each intervals, the approximated polynomial $f_i(x)$ is provided by Sollya. The coefficients for all the polynomials are built into a table and implemented in Block RAM, when it is mapped to FPGA. The polynomial evaluator uses the Horner's Rule and with the flexibility of FPGA, the wordlength of coefficients are minimized and

[1] Version 3.0 http://sollya.gforge.inria.fr/

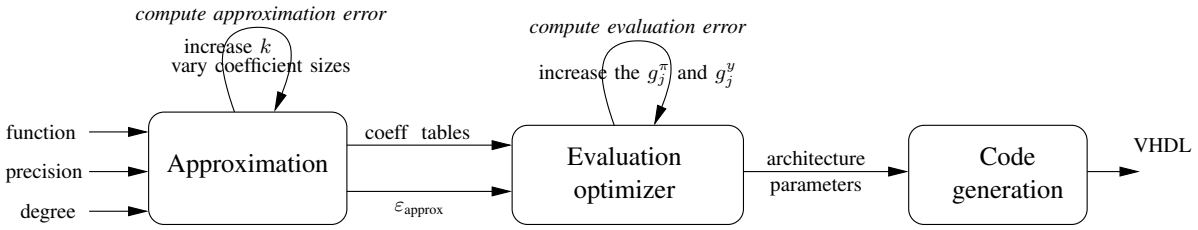

Figure 7.2: Procedure of generating function evaluator

rounded while maintaining within the error bound. The overall error analysis is presented in [43] written in C++, however, it is out of the scope of this thesis, as we only interest in evaluation error. Figure 7.2 is the diagram of the process to generate the function evaluator.

In this thesis, a VHDL design is generated from the above process, with a polynomial evaluator and a table of coefficients. The range under approximation is divided into 256 intervals with a set of coefficients in each intervals. The coefficient table will take in the input x and output a set of coefficients for each interval in one clock cycle. The polynomial evaluator will compute the equation based on the coefficients and input x.

It has been discussed in previous chapter that by using Horner's Rule, five multipliers and five adders are used for 5^{th} degree polynomial based on (Eq. 2.6). However, the size of them could be different based on the precision requirement, and this is done by rounding the coefficients and truncating the intermediate results after each step. Table 7.1 shows the size of the wordlength of the coefficients after rounding. The overall size of the coefficients is 246 bit. With the 256 intervals, the coefficient table is then formed into an array of size 256×246, which is 22% less than the coefficients without rounding. The array is further optimized and re-shaped to 512×123 in order to fit more efficiently into three RAMB36E1 block, configured as 512×72.

The rounding of the coefficients also help on reducing the multiplier sizes. Column two in Table 7.2 shows the effect of rounding the coefficients. However, the multiplier could be further shrunk by truncation. Table 7.2 summarizes the DSP count for each

Coefficient	Wordlength
a_0	56
a_1	50
a_2	44
a_3	38
a_4	30
a_5	22

Table 7.1: Coefficient table for 5^{th} degree polynomial evaluator.

Multiplier	Without Optimization	After Rounding	After Truncation
p_1x	9	6	6
p_2x	9	6	5
p_3x	9	5	3
p_4x	9	4	3
a_5x	9	2	1

Table 7.2: DSP count for each multiplication for 5^{th} degree polynomial evaluator.

multiplication after truncation as well. The DSP cost reduces to 18 or 45% less in total compared to no optimization is performed. Noted that p_i is defined as $p_i = (p_{i-1} + a_i)x$ in the table. The author of FloPoCo tool does not use conventional cascaded chain to build multipliers. "Tiling like" multiplier are used instead which fits more tightly with the DSP block. Consequently, the post-adders in DSP blocks are not able to be used. The partial products are generated from the DSP blocks and added using 3:1 compressors and carry propagate adders built from LUTs instead. Small multipliers are used to perform the multiplication of guard bit that required, however exceeds the operand wordlength of DSP. Figure 7.3 shows the overall schematic of the functional evaluator.

In the other application where 4^{th} degree polynomial is used, four multipliers and four adders are required by Horner's Rule, which shows in (Eq. 7.1).

$$f(x) = (((a_4x + a_3)x + a_2)x + a_1)x + a_0 \qquad \text{(Eq. 7.1)}$$

Table 7.3 shows the coefficients generated from FloPoCo tool. 19% of the bit is rounded from the coefficients in 4^{th} degree polynomial evaluator and the array is re-shaped to

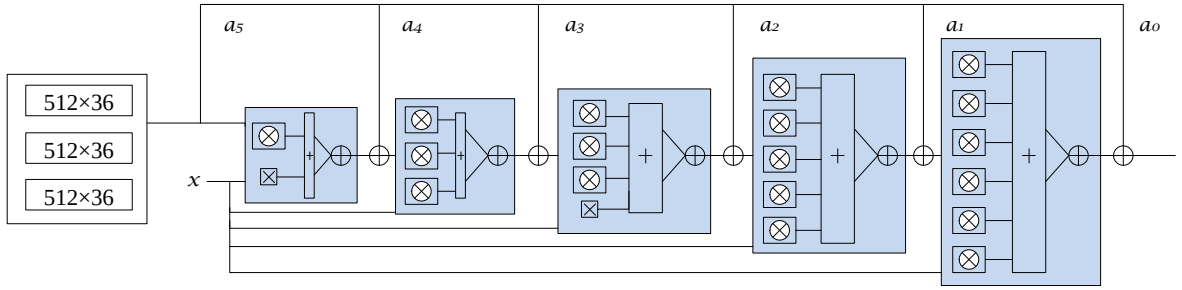

Figure 7.3: Schematic of 5^{th} degree polynomial evaluator.

Coefficient	Wordlength
a_0	56
a_1	48
a_2	39
a_3	31
a_4	22

Table 7.3: Coefficient table for 4^{th} degree polynomial evaluator.

512×100. However, limited by the smallest size of RAMB36E1 configuration, it still needs three block RAM to store the table. Table 7.4 summaries the DSP count, which shows 61.2% DSP reduction in total after optimization. The schematic of 4^{th} degree evaluator is similar to the 5^{th} degree one and it is not shown here for the reason of space.

Multiplier	Without Optimization	After Rounding	After Truncation
p_1x	9	6	5
p_2x	9	5	4
p_3x	9	4	3
a_4x	9	2	2

Table 7.4: DSP count for each multiplication for 4^{th} degree polynomial evaluator.

7.2 Polynomial Evaluator using Estrin's Method

Estrin's method can be used to parallelize the polynomial evaluation process. In this thesis, we've tried to replace the design from FloPoCo first with the Estrin's method.

In order to keep the approximation error in the function evaluator, coefficients, input range and degree of polynomial are consistent with the design from FloPoCo. Based on (Eq. 2.8), it requires five multipliers, two squarers and five adders to complete the computation. The size of x is altered to reduce the computation complexity while maintaining the evaluation error no worse than FloPoCo evaluator could achieve. Figure 7.4 shows the optimization process. As the polynomial evaluator has to be able to compute all 256 polynomials within the acceptable evaluation error, we pick the set of coefficient that has the worst evaluation error using FloPoCo design, indicated by the log from FloPoCo. Both evaluation error of the FloPoCo design and design using Estrin's method are computed. If Estrin's method is not accurate enough, the precision of intermediate step will be increased and the error will be calculated again. Noted that the coefficients from FloPoCo are not rounded any more in Estrin's method. The iteration will only finish when the difference is within the acceptable limit. Therefore, the newly developed evaluator, which has the evaluation error no worse than the worst case of FloPoCo evaluator, shall be able to plug and play into the system without losing on precision. The size of x is also optimized to fit into DSP operand wordlength for efficiency purpose. The optimized size of x will be gone through the evaluation error verification again to ensure the error is within the bound.

In order to meet the evaluation error no worse than the Horner's Rule while further optimizing the operand wordlength, Gappa [2] is used to verify the optimized design is within the error bound. Gappa is a tool intended to help verifying and formally proving properties on numerical programs dealing with floating-point or fixed-point arithmetic [58] and it has been used in many research work [59,60]. It computes the range of a given function based on the constraints by using interval arithmetics. In this case, we use it to verify the post-optimized fixed point polynomial evaluator is able to remain in the same error bound as the FloPoCo evaluator.

[2]Version 0.16.4 gappa.gforge.inria.fr

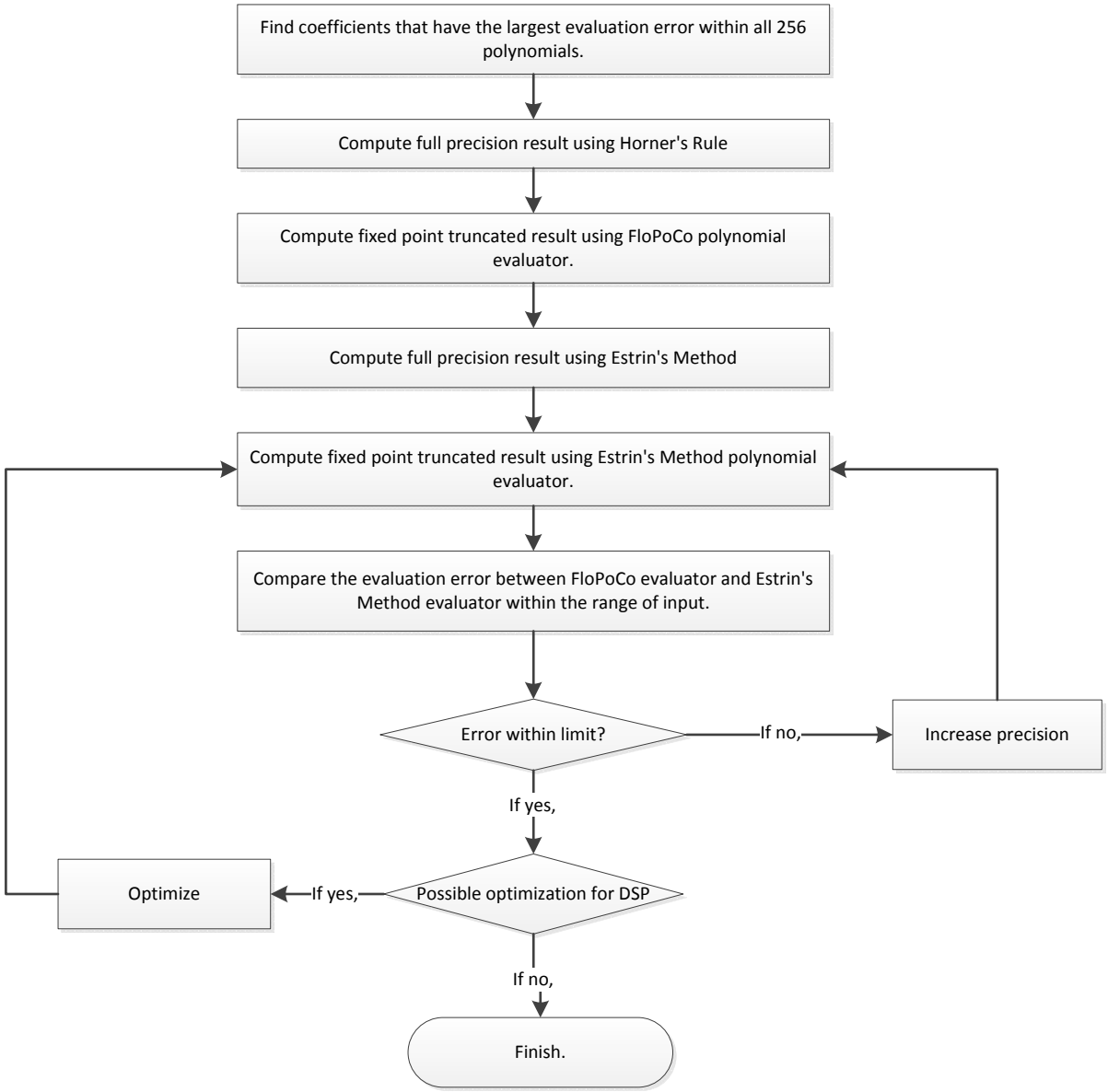

Figure 7.4: Optimization flow for Estrin's method.

Source code in Listing A.1 in the appendix shows the first part of the verification program using Gappa, which defines the polynomial evaluation error using Horner's Rule. Line 1 to 7 first defines the value of coefficients in decimal. Line 10 defines the equation to compute full precision value Mp of the polynomial using Horner's Rule. The coefficients and input x are both rounded to reduce the area of multipliers and adders. This is followed by fixed point arithmetic with truncations. The final fixed-point truncated value is store in p.

Source code in appendix Listing A.2 shows the part to compute the evaluation error based on Estrin's method. Similar as in Horner's Rule, both full precision value Mq and fixed point value q after truncation are computed from line 1 to 22. Line 24 tries to evaluate both error bound by deducting p and q from the full precision value of Mp. This is within the range of $[0, 2^{-8}]$ which is the smallest interval we choose in range reduction. Line 27 gives a hint that the full precision value of Mp and Mq should be the same, otherwise Gappa is not able to recognize the relationship between q and Mp. Line 28 indicates the commutative property stands true.

If the verification shows the error for two methods are not within the same bound, rounding bit of x and truncation bit is then modified for the next iteration to verify the error again. Listing A.3 in the appendix shows the final result of the verification from Gappa, which also indicates a finish in the flow shown in Figure 7.4.

The results of the flow shows that Estrin's method could only achieve the same precision as Horner's Rule by maintaining similar wordlength in intermediate steps as FloPoCo design. If the decision is to use one DSP to compute the x^2 and one DSP to compute x^4, which is equivalent to 18 bit wordlength in each step, the overall precision will drop by at least 5 bits. However, to gain back the evaluation error, one must use additional four DSPs to compute the powers of x when the wordlength increases. This does not only increase the overhead for parallelism but also reduce the performance gain as the latency increases substantially.

Multiplier	Without Optimization	After Rounding	After Truncation
x^2	5	4	3
x^4	5	4	3
a_5x	9	2	2
q_5x^4	9	4	3
a_3x	9	4	3
q_3x^2	9	6	5
a_1x	9	6	5

Table 7.5: DSP count for each multiplication for 5^{th} degree polynomial evaluator using Estrin's method.

Table 7.5 summarizes the DSP savings for the faithful rounding and truncation without losing on accuracy after verified by Gappa. In the table, q_i is the notation for $a_ix - a_{i-1}$. Figure 7.5 shows the fixed point evaluator using Estrin's method. The multipliers and squarers are mainly built using cascaded chains after optimization to operand wordlength of DSP. Noted that for computing square of x, the proposed squarer is used to further optimizing the utilization. In the case where guard bits are not able to fit into the DSP data width, similar "tiling like" multipliers and small multiplier built in LUTs are used instead in multiplication of q_5x^4 and a_3x.

Apparently, similar dilemma between precision and data path wordlength exists in 4^{th} degree polynomial evaluator, which the equation used in Estrin's method is,

$$f(x) = a_4 \cdot x^4 + (a_3x + a_2) \cdot x^2 + (a_1x + a_0) \qquad \text{(Eq. 7.2)}$$

The new evaluator is built according to the flow and verified the Gappa tool using the same coefficients. The DSP saving is provided in Table 7.6. Limited to the space, the details of the 4^{th} degree evaluator is not shown here.

7.3 Polynomial Evaluator using Proposed Method

The proposed method requires a more complex design generator because of the new coefficients required. Since the time to generate the evaluator is negligible, we maintain

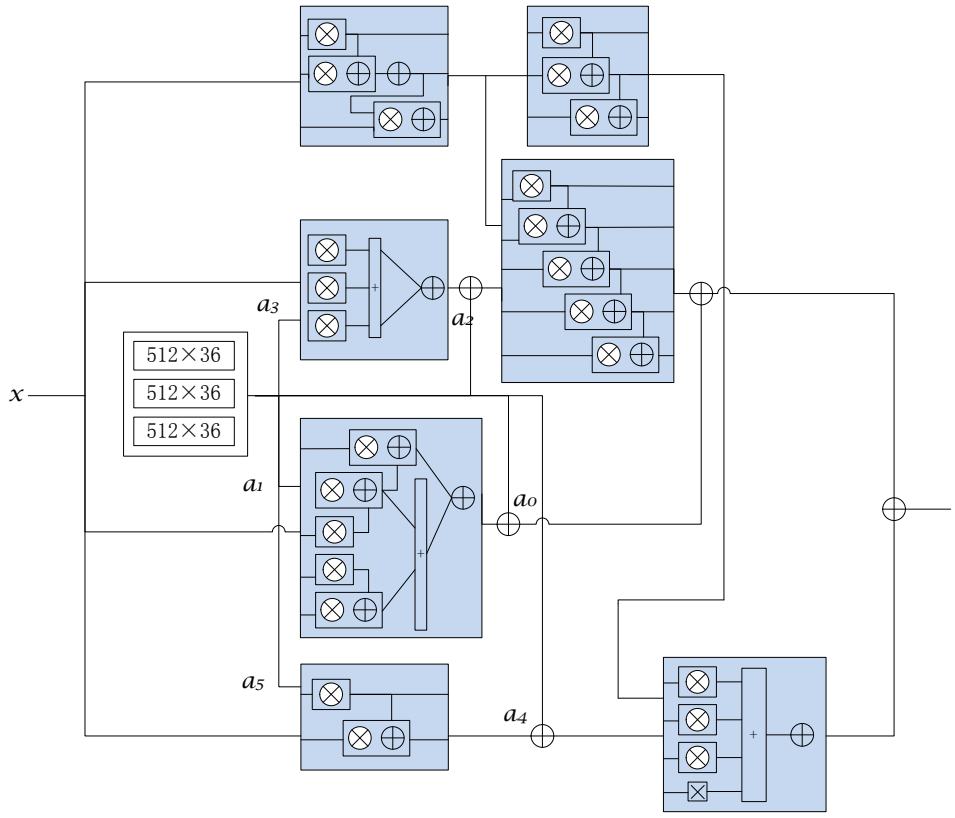

Figure 7.5: Schematic of 5^{th} degree polynomial evaluator using Estrin's method.

Multiplier	Without Optimization	After Rounding	After Truncation
x^2	5	4	3
x^4	5	4	2
a_4x^4	9	2	1
a_3x	9	4	4
q_3x^2	9	4	4
a_1x	9	6	5

Table 7.6: DSP count for each multiplication for 4^{th} degree polynomial evaluator using Estrin's method.

full precision when calculating the new coefficients and only truncate them last when needed. Compared to the Estrin's method, both x and the new coefficients are rounded to the sweet spot. With the help of Gappa tool, we develop the following flow described in Figure 7.6 to derive the wordlength for each arithmetic step. The flow is very similar as the flow in Estrin's method. The new coefficients will be rounded according to precision, BRAM size and DSP data path wordlength. The trail will begin with rounding the coefficients to meet the same error bound as Horner's Rule. If the constraint is not met, the precision of coefficients has to be increased. This will follow by an optimization to further reduce the size of coefficients which does not lead to accuracy loss to reduce the Block RAM utilization. After that, if the coefficients are not able to fit well with the DSP block operand size, trials on alternating the rounding positions are performed to optimize the DSP block usage.

For the 5^{th} degree polynomial, four new coefficients are to be computed as shown in (Eq. 5.13) to (Eq. 5.15) and the flow will consider rounding for all of them. However, it was found that the flow could not converge as it has system limitation reaching the same precision as Horner's Rule with similar rounding position of the coefficients, for the reason been stated in Chapter 5. Therefore, alternative equation (Eq. 5.16) is used instead. The source code for Gappa tool to veify the design is shown in Listing A.4 in the appendix. Noted that definition of original coefficients and the error computation of Horner's Rule are identical in Lising A.1 and they are not shown here. In Listing A.4 in the appendix, line 2-5 defines the full precision value of the new coefficients and their corespondent fixed-point version are calculated in line 9-20. Similar as Estrin's method, the proof is going to compute the error bound for both Horner's rule and proposed method and a hint is given to suggest both equations have the same full precision value. Listing A.5 in the appendix shows the final result of the verification from Gappa after finishing the iteration.

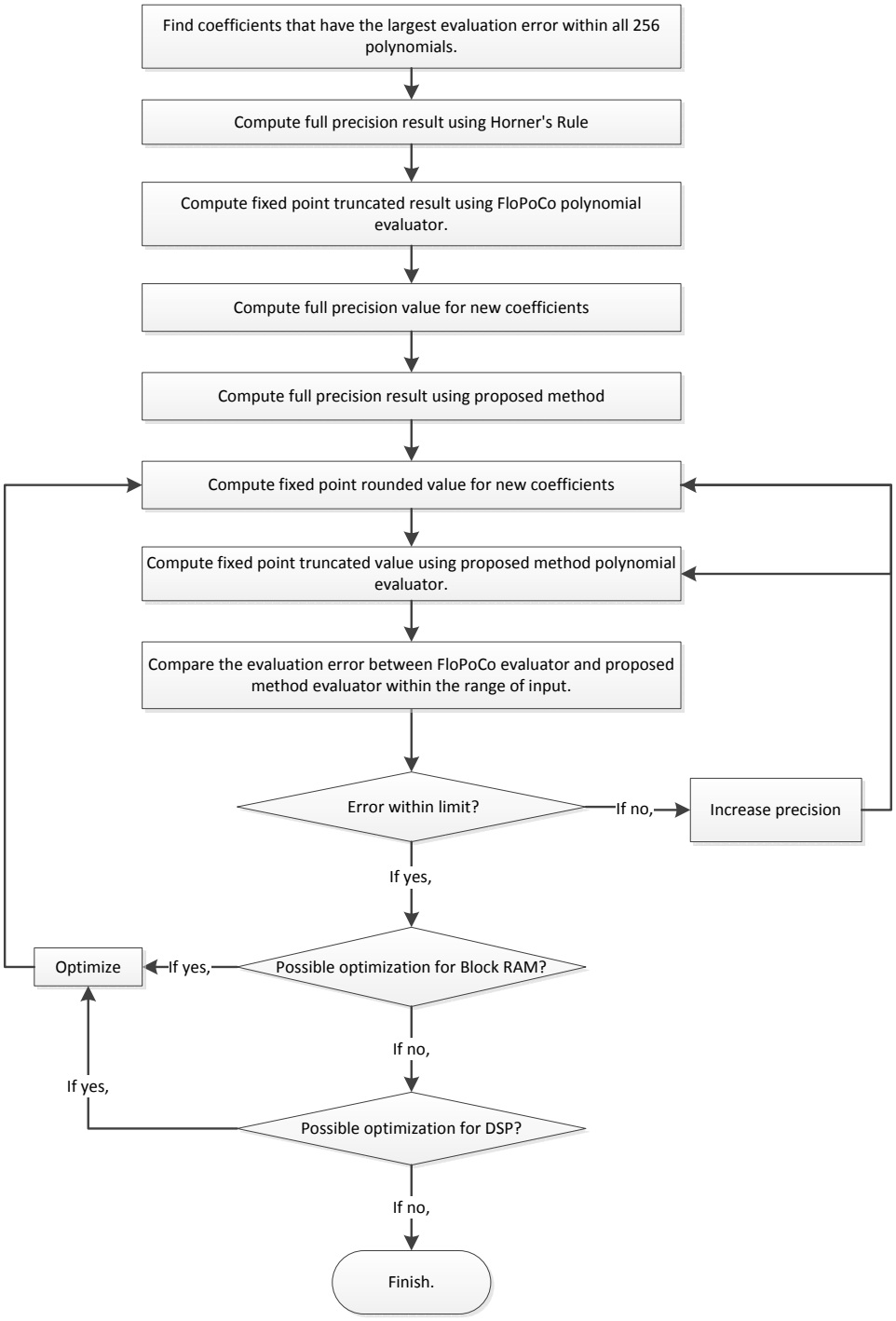

Figure 7.6: Optimization flow for proposed method.

Coefficient	Wordlength
m_5	40
m_4	53
n_5	46
n_4	55
a_5	16
a_4	28

Table 7.7: Coefficient table for 5^{th} degree polynomial evaluator using proposed method.

Table 7.7 provides the new coefficients size after rounding. The new coefficients have 244 significant bits which matches the amount in the FloPoCo design, thus the architecture of coefficient table could be reused.

Multiplier	Without Optimization	After Rounding	After Truncation
x^2	3	3	2
r_5^2	3	3	2
$(r_5^2)a_5$	9	3	2
r_4^2	5	5	4
$(r_4^2)a_4$	9	4	3
a_1x	9	6	5

Table 7.8: DSP count for each multiplication for 5^{th} degree polynomial evaluator using proposed method.

Table 7.8 presents the DSP savings for the faithful rounding and truncation without losing on accuracy after verified by Gappa. In the Table 7.8, r_i represents $(x^2 + m_i)$ and the third and fifth row indicates the multiplication of the two numbers, which the first operand in bracket is from the previous row. The total number of DSP count after optimization is equivalent to the Horner's Rule and fewer than Estrin's method. With the help of proposed squarer, the fourth row in Table 7.8 reduces the DSP count to 5, and is further truncated to 4. Similar as Estrin's method, most of the multiplier/squarer utilize the cascaded chains after the optimization, or post-adders are used to perform part of the partial product reduction by using C input of the DSP. However, even the size of the coefficients are determined by ourselves, it is interesting to see that the optimization

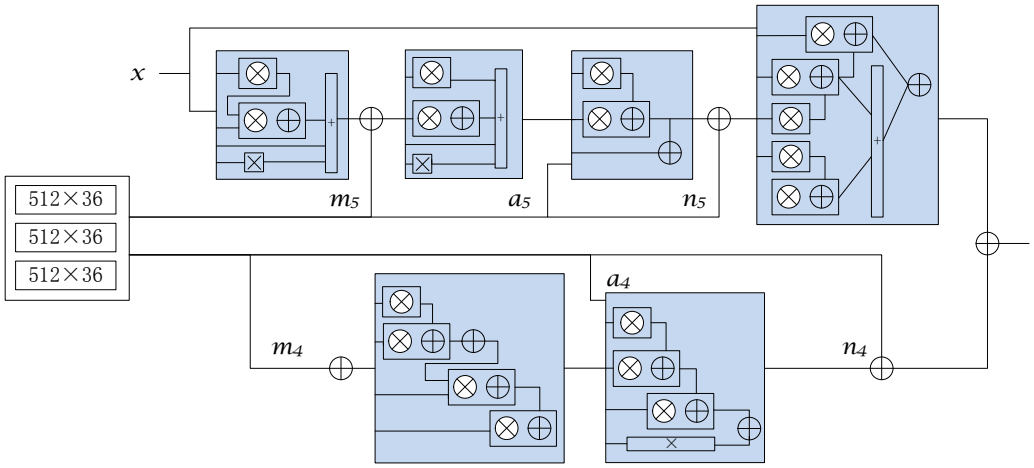

Figure 7.7: Schematic of 5^{th} degree polynomial evaluator using proposed method.

of mapping to DSP48E1 does not always improve the design further. In this case r_5^2 and $(r_4^2)a_4$ are not able to fit well into the DSP48E1 primitive after the truncation. Therefore small multipliers are needed after most part of the multiplication being mapped to DSP48E1 and they are implemented using adders/compressors in the LUTs to reduce the DSP count. The overall schematic is illustrated in Figure 7.7.

Coefficient	Wordlength
m_4	51
n_4	59
a_4	22
a_3	31
a_1	48

Table 7.9: Coefficient table for 4^{th} degree polynomial evaluator using proposed method.

For the same 4^{th} degree polynomial, the proposed method uses (Eq. 7.3) to compute the equation. It is based on alternative equation and similar to (Eq. 5.16) for 5_{th} degree

Multiplier	Without Optimization	After Rounding	After Truncation
x^2	3	3	2
r_4^2	3	3	2
$(r_4^2)a_4$	9	3	2
r_4^2	5	5	4
$(r_4^2)a_4$	9	4	3
a_1x	9	6	5

Table 7.10: DSP count for each multiplication for 4^{th} Degree Polynomial Evaluator using Proposed Method.

polynomial.

$$f(x) = a_4(x^2 + m_4)^2 + n_4 + x \cdot (a_3x^2 + a_1) \quad \text{(Eq. 7.3)}$$

where,

$$m_4 = \frac{a_2}{2a_4} \quad \text{(Eq. 7.4)}$$

$$n_4 = a_0 - \frac{a_2^2}{4a_4} \quad \text{(Eq. 7.5)}$$

Using the same process, new coefficients are derived with faithful rounding, which are shown in Table 7.9 and the amount of DSP used after truncation is listed in Table 7.10. Limited to the space, the schematic of the 4^{th} degree evaluator is not shown here.

7.4 Implementation Results

After building the schematic with macros, both Estrin's method and proposed method designs have been implemented in RTL and synthesized, placed and routed on targeted FPGA. The results are presented blow.

	LUT	DSP	BRAM
FloPoCo	901	18	3
Estrin's	350	24	3
Proposed	375	18	3

Table 7.11: Hardware resource for 5^{th} degree polynomial evaluator.

	LUT	DSP	BRAM
FloPoCo	665	14	3
Estrin's	189	19	3
Proposed	297	15	3

Table 7.12: Hardware resource for 4^{th} degree polynomial evaluator.

Table 7.11 and 7.12 summarize the hardware cost for both polynomial evaluators for 5^{th} degree and 4^{th} degree polynomial. As all the three methods use similar amount of bit for the coefficients, the number of BRAM used to store the coefficients are the same. The proposed method uses same amount of DSP in 5^{th} degree polynomial evaluator and one more in 4^{th} degree one. By contrast, Estrin's method uses 6 or 33% more in 5^{th} degree evaluator and 5 or 35% more in 4^{th} degree one. This is contributed by both precision requirement and the complexity of the algorithm. FloPoCo design has utilized many "tiling like" multipliers which consists of post compressors and adders and they lead to relatively high LUT cost. Both Estrin's method and proposed method are optimized to implement more cascaded chains in order to reduce the LUT usage, therefore they use less LUTs compared to FloPoCo design.

To evaluate overall hardware area cost among three methods, Table 7.13 shows the equivalent number of LUT where one DSP has the same weight as 149 LUT in the targeted FPGA. It is clearly that the proposed method uses the least amount of hardware resources, which is 11.9% less compared to the FloPoCo design in 5^{th} degree polynomial. In 4^{th} degree polynomial, it still has an advantage of 5% less. By contrast, Estrin's method uses 14.1% and 11.5% more hardware resources respectively compared to the FloPoCo design.

Table 7.14 and 7.15 present the performance of all the three evaluators. Estrin's method shows its advantage in terms of latency as it has higher level of parallelism than the Horner's Rule, which is 48.6% faster in 5^{th} degree polynomial and 44.8% faster in 4^{th} degree polynomial. Proposed method has the same latency as the Estrin's method

	5^{th} Degree	4^{th} Degree
FloPoCo	4433.5	3412.5
Estrin's	5060	3917.75
Proposed	3907.5	3240.75

Table 7.13: Equivalent LUT count for polynomial evaluators.

in 4^{th} degree polynomial, while three stages, or 7.6% longer than Estrin's method in 5^{th} degree polynomial. This is mainly due to the use of alternative equation, which one more multiplication stage is added in the critical path. Compare to the maximum frequency of DSP, FloPoCo design has a frequency drop of 26%, where the other two drop around 16%. Further pipeline on top of the FloPoFo default pipline strategy could increase the frequency until nearly maximum frequency of the DSP block, while it is not discussed here.

	Latency	Frequency
FloPoCo	37	334MHz
Estrin's	19	378MHz
Proposed	22	380MHz

Table 7.14: Performance for 5^{th} degree polynomial evaluator.

	Latency	Frequency
FloPoCo	29	332MHz
Estrin's	16	384MHz
Proposed	16	381MHz

Table 7.15: Performance for 4^{th} degree polynomial evaluator.

In summary, proposed method shows large advantages over non-parallel method based on Horner's Rule in both hardware cost and speed. It also shows hardware cost reduction over Estrin's method in the same level of parallelism. The result thus proves our hypothesis stated in Chapter 5.

Chapter 8

Conclusion

The parallel polynomial evaluation algorithm implementation on FPGA has been analyzed the and a novel algorithm to tackle the problem of large hardware overhead on the conventional algorithms has been proposed. The proposed method shows significant advantages in terms of computation complexity by replacing multiplication with more efficient squaring in theory. The novel algorithm has been implemented on FPGA in real-world applications The results have proved that the novel algorithm is more efficient than both conventional method, Horner's Rule and Estrin's methods. The area reduction is enabled by an efficient squarer circuit developed by us. The novel squarer circuit has shown 21.8% fewer hardware overhead than the design in the literature which uses the least amount of DSP resources in FPGA. Therefore, the overall polynomial evaluator is able to reach 41% latency reduction with 11.9% less area compared to the Horner's Rule and the performance gain can only be reached by Estrin's method consuming larger amount of hardware resources.

What's need beyond this thesis is to automate the design generation process. First of all, a general format of such algorithm shall be developed with implementation details. Secondly, automation of the flows stated in Chapter 7 shall be done, which includes optimization for DSP operand size and Block RAM size. The formal verification of error bound in the new polynomial evaluator over the entire range could be included in

the automation tool as well. This could also possibly enable generation of polynomial on-the-fly in the future.

Further pipelining the design, for both conventional and novel evaluators can be done to meet the maximum speed of DSP and built into the design generator. This could easily improve the throughput of the design substantially.

The novel polynomial evaluation algorithm can be applied to more applications, not only limited to function evaluation. Meanwhile, it will show even larger advantages in floating point system. Tools to auto-generate designs which could convert conventional polynomials evaluation algorithm into novel one automatically is considered in the future.

References

[1] J.-M. Muller, *Elementary functions: algorithms and implementation.* Secaucus, NJ, USA: Birkhauser Boston, Inc., 1997.

[2] L. Rabiner, J. McClellan, and T. Parks, "Fir digital filter design techniques using weighted chebyshev approximation," *Proceedings of the IEEE*, vol. 63, no. 4, pp. 595–610, 1975.

[3] V. S. Miller, "Use of elliptic curves in cryptography," in *Lecture notes in computer sciences; 218 on Advances in cryptology—CRYPTO 85.* New York, NY, USA: Springer-Verlag New York, Inc., 1986, pp. 417–426. [Online]. Available: http://dl.acm.org/citation.cfm?id=18262.25413

[4] F. Ahmadi, I. McLoughlin, and H. Sharifzadeh, "Analysis-by-synthesis method for whisper-speech reconstruction," in *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, 2008, pp. 1280–1283.

[5] H. Vikalo, B. Hassibi, and T. Kailath, "Iterative decoding for mimo channels via modified sphere decoding," *Wireless Communications, IEEE Transactions on*, vol. 3, no. 6, pp. 2299–2311, 2004.

[6] V. Y. Pan, "Methods of computing values of polynomials," *Russ. Math. Surv*, vol. 21, p. 105, 1966.

[7] Y. Muraoka, "Parallelism exposure and exploitation in programs," Ph.D. dissertation, 1971.

[8] I. Munro and M. Paterson, "Optimal algorithms for parallel polynomial evaluation," *J. Comput. Syst. Sci.*, vol. 7, no. 2, pp. 189–198, 1973.

[9] K. Maruyama, "On the parallel evaluation of polynomials," *Computers, IEEE Transactions on*, vol. C-22, no. 1, pp. 2–5, 1973.

[10] L. Li, J. Hu, and T. Nakamura, "A simple parallel algorithm for polynomial evaluation," *SIAM J. Sci. Comput.*, vol. 17, no. 1, pp. 260–262, 1996.

[11] J. Villalba, G. Bandera, M. A. Gonzalez, J. Hormigo, and E. L. Zapata, "Polynomial evaluation on multimedia processors," in *Application-Specific Systems, Architectures and Processors, 2002. Proceedings. The IEEE International Conference on*, pp. 265–274.

[12] G. Estrin, "Organization of computer systems: the fixed plus variable structure computer," pp. 33–40, 1960.

[13] W. S. Dorn, "Generalizations of horner's rule for polynomial evaluation," *IBM J. Res. Dev.*, vol. 6, no. 2, pp. 239–245, 1962.

[14] J. Duprat and J.-M. Muller, "Hardwired polynomial evaluation," *J. Parallel Distrib. Comput.*, vol. 5, no. 3, pp. 291–309, 1988.

[15] W. P. Burleson, "Polynomial evaluation in vlsi using distributed arithmetic," *Circuits and Systems, IEEE Transactions on*, vol. 37, no. 10, pp. 1299–1304, 1990.

[16] D.-U. Lee, A. A. Gaffar, O. Mencer, and W. Luk, "Optimizing hardware function evaluation," *IEEE Trans. Comput.*, vol. 54, no. 12, pp. 1520–1531, 2005.

[17] A. Tisserand, "Hardware reciprocation using degree-3 polynomials but only 1 complete multiplication," in *Circuits and Systems, 2007. MWSCAS 2007. 50th Midwest Symposium on*, pp. 301–304.

[18] F. Haohuan, O. Mencer, and W. Luk, "Optimizing logarithmic arithmetic on fpgas," in *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pp. 163–172.

[19] J. Detrey and F. de Dinechin, "Table-based polynomials for fast hardware function evaluation," in *Application-Specific Systems, Architecture Processors, 2005. ASAP 2005. 16th IEEE International Conference on*, pp. 328–333.

[20] A. Tisserand, "High-performance hardware operators for polynomial evaluation," *Int. J. High Perform. Syst. Archit.*, vol. 1, no. 1, pp. 14–23, 2007.

[21] N. Brisebarre, J. M. Muller, and A. Tisserand, "Sparse-coefficient polynomial approximations for hardware implementations," in *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Eighth Asilomar Conference on*, vol. 1, pp. 532–535 Vol.1.

[22] S. Winograd, "On the number of multiplications required to compute certain functions," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 58, no. 5, pp. 1840–1842, 1967.

[23] J. A. Pineiro, J. D. Bruguera, and J. M. Muller, "Fpga implementation of a faithful polynomial approximation for powering function computation," in *Digital Systems, Design, 2001. Proceedings. Euromicro Symposium on*, pp. 262–269.

[24] M. Wojko and H. ElGindy, "On determining polynomial evaluation structures for fpga based custom computing machines," *in Proc. 4th Australasian Comput. Arch. Conf*, pp. 11–22, 1999.

[25] B. Rachid, S. Stephane, and T. Arnaud, "Function evaluation on fpgas using on-line arithmetic polynomial approximation," in *Circuits and Systems, 2006 IEEE North-East Workshop on*, pp. 21–24.

[26] N. Anane, H. Bessalah, M. Issad, K. Messaoudi, and M. Anane, "Reconfigurable architecture for elementary functions evaluation," in *Design & Technology of Integrated Systems in Nanoscal Era, 2009. DTIS '09. 4th International Conference on*, pp. 90–94.

[27] K. Chung and L.-S. Kim, "Area-efficient special function unit for mobile vertex processors," *Electronics Letters*, vol. 45, no. 16, pp. 826 –827, 30 2009.

[28] D. D. Donofrio and X. Li, "Enhanced floating-point unit for extended functions," U.S Patent 7 676 535, 2007.

[29] B. Pasca, "Correctly rounded floating-point division for DSP-enabled FPGAs," in *Field Programmable Logic and Applications (FPL)*, Aug. 2012, pp. 249 –254.

[30] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *Design Test of Computers*, vol. 28, no. 4, pp. 18 –27, Jul-Aug 2011.

[31] A. Strollo and D. De Caro, "Booth folding encoding for high performance squarer circuits," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 50, no. 5, pp. 250 – 254, May 2003.

[32] K.-J. Cho and J.-G. Chung, "Parallel squarer design using pre-calculated sums of partial products," *Electronics Letters*, vol. 43, no. 25, pp. 1414 –1416, 6 2007.

[33] J.-T. Yoo, K. F. Smith, and G. Gopalakrishnan, "A fast parallel squarer based on divide-and-conquer," *IEEE Journal of Solid-State Circuits*, vol. 32, pp. 909–912, 1995.

[34] J. Pihl and E. Aas, "A multiplier and squarer generator for high performance DSP applications," in *IEEE 39th Midwest symposium on Circuits and Systems*, vol. 1, Aug. 1996, pp. 109 –112 vol.1.

[35] A. J. Al-Khalili and H. Aiping, "Design of a 32-bit squarer - exploiting addition redundancy," in *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on*, vol. 5, pp. V–325–V–328 vol.5.

[36] F. de Dinechin and B. Pasca, "Large multipliers with fewer DSP blocks," in *Field Programmable Logic and Applications*, Sep. 2009, pp. 250 –255.

[37] S. Gao, N. Chabini, D. Al-Khalili, and P. Langlois, "FPGA-based efficient design approach for large-size two's complement squarers," in *Application-specific Systems, Architectures and Processors*, Jul. 2007, pp. 18 –23.

[38] B. Lee and N. Burgess, "Improved small multiplier based multiplication, squaring and division," in *Field-Programmable Custom Computing Machines*, Apr. 2003, pp. 91 – 97.

[39] F. Curticpean and J. Nittylahti, "Direct digital frequency synthesizers of high spectral purity based on quadratic approximation," in *Electronics, Circuits and Systems, 2002. 9th International Conference on*, vol. 3, pp. 1095–1098 vol.3.

[40] J. C. Bajard, L. Imbert, and G. A. Jullien, "Parallel montgomery multiplication in gf(2¡sup¿k¡/sup¿) using trinomial residue arithmetic," in *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, pp. 164–171.

[41] D. De Caro and A. G. M. Strollo, "High-performance direct digital frequency synthesizers using piecewise-polynomial approximation," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 52, no. 2, pp. 324–337, 2005.

[42] M. Bodrato and A. Zanoni, "Long integers and polynomial evaluation with estrin's scheme," in *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2011 13th International Symposium on*, pp. 39–46.

[43] F. de Dinechin, M. Joldes, and B. Pasca, "Automatic generation of polynomial-based hardware architectures for function evaluation," in *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, pp. 216–222.

[44] X. Inc. (2013) Field programmable gate array (fpga). [Online]. Available: http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm

[45] I. Xilinx, "Virtex-4 fpga user guide," 2008.

[46] ——, "Virtex-6 fpga configurable logic block," 2012.

[47] X. Inc, "Virtex-6 user manual," Xilinx Inc.

[48] C. S. Wallace, "A suggestion for a fast multiplier," *Electronic Computers, IEEE Transactions on*, vol. EC-13, no. 1, pp. 14–17, 1964.

[49] C. R. Baugh and B. A. Wooley, "A two's complement parallel array multiplication algorithm," *Computers, IEEE Transactions on*, vol. C-22, no. 12, pp. 1045–1047, 1973.

[50] H. Parandeh-Afshar and P. Ienne, "Measuring and reducing the performance gap between embedded and soft multipliers on fpgas," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pp. 225–231.

[51] J. Pihl and E. J. Aas, "A multiplier and squarer generator for high performance dsp applications," in *Circuits and Systems, 1996., IEEE 39th Midwest symposium on*, vol. 1, pp. 109–112 vol.1.

[52] A.Karatsuba and Y.Ofman, "Multiplication of multi-digit numbers on automata," *Soviet Physics Doklady 7*, vol. 145, pp. pp. 595–596, 1963.

[53] S. Srinath and K. Compton, "Automatic generation of high-performance multipliers for FPGAs with asymmetric multiplier blocks," in *Field Programmable Gate Arrays*, ser. FPGA '10. New York, NY, USA: ACM, 2010, pp. 51–58. [Online]. Available: http://doi.acm.org/10.1145/1723112.1723123

[54] S. Gao, D. Al-Khalili, and N. Chabini, "Asymmetric large size signed multipliers using embedded blocks in FPGAs," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, May 2011, pp. 271 –277.

[55] R. K. Kolagotla, W. R. Griesbach, and H. R. Srinivas, "Vlsi implementation of 350 mhz 0.35 100 μm 8 bit merged squarer," *Electronics Letters*, vol. 34, no. 1, pp. 47–48, 1998.

[56] P. Montgomery, "Five, six, and seven-term Karatsuba-like formulae," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 362 – 369, Mar. 2005.

[57] J. M. Simkins and B. D. Philofsky, "Structures and methods for implementing ternary adders/subtractors in programmable logic devices," U.S Patent 7 274 211, 2007.

[58] G. Melquiond. (2013) Gappa. [Online]. Available: http://gappa.gforge.inria.fr/

[59] S. Boldo, J.-C. Fillitre, and G. Melquiond, "Combining coq and gappa for certifying floating-point programs," in *Intelligent Computer Mathematics*, ser. Lecture Notes in Computer Science, J. Carette, L. Dixon, C. Coen, and S. Watt, Eds. Springer Berlin Heidelberg, 2009, vol. 5625, pp. 59–74.

[60] F. de Dinechin, C. Q. Lauter, and G. Melquiond, "Assisted verification of elementary functions using gappa," in *Proceedings of the 2006 ACM symposium on Applied computing*, ser. SAC '06. New York, NY, USA: ACM, 2006, pp. 1318–1322. [Online]. Available: http://doi.acm.org/10.1145/1141277.1141584

Appendix A

Supplementary Figures Tables and Source codes

For multiplier has operand larger than 42 bits, three splits are required for implementing on FPGA using cascaded method and this is shown in Figure A.1.

For squarer using cascaded method that has operand larger than 53 bits, Figure A.2 shows the schematic diagram where one more DSP block is needed. However, if it is exact 53 bits, an adder could be used instead of incurring a DSP block, shown in figure A.3.

Table A.1 presents the amount of LUT and DSP for all types of squarers implemented on FPGA.

Figure A.4 shows the diagram of 3:1 compressor using two LUT to improve timing.

Source codes for gappa program have been shown in Listing A.1 to A.5

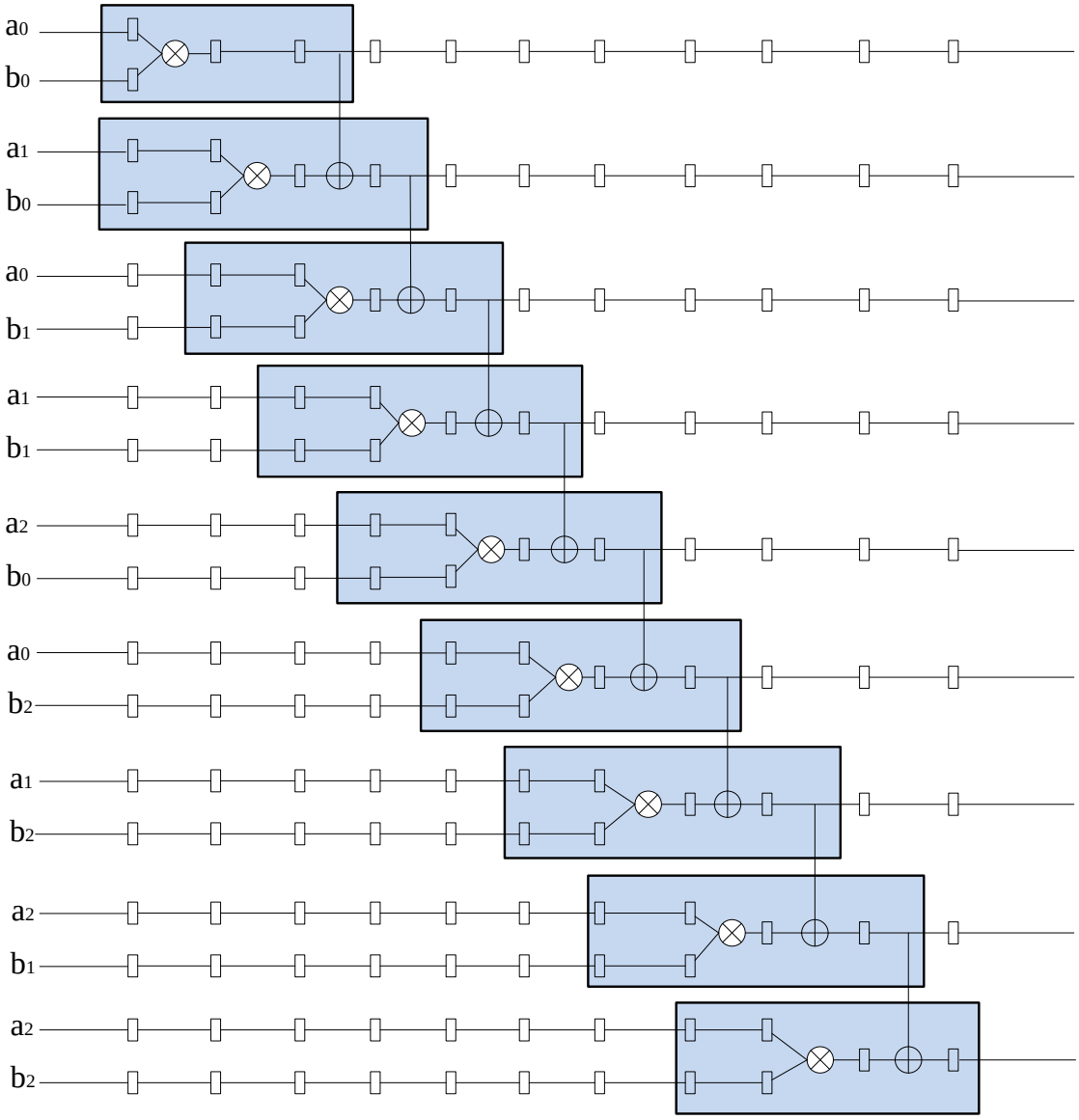

Figure A.1: Schematic of multipliers for three splits input.

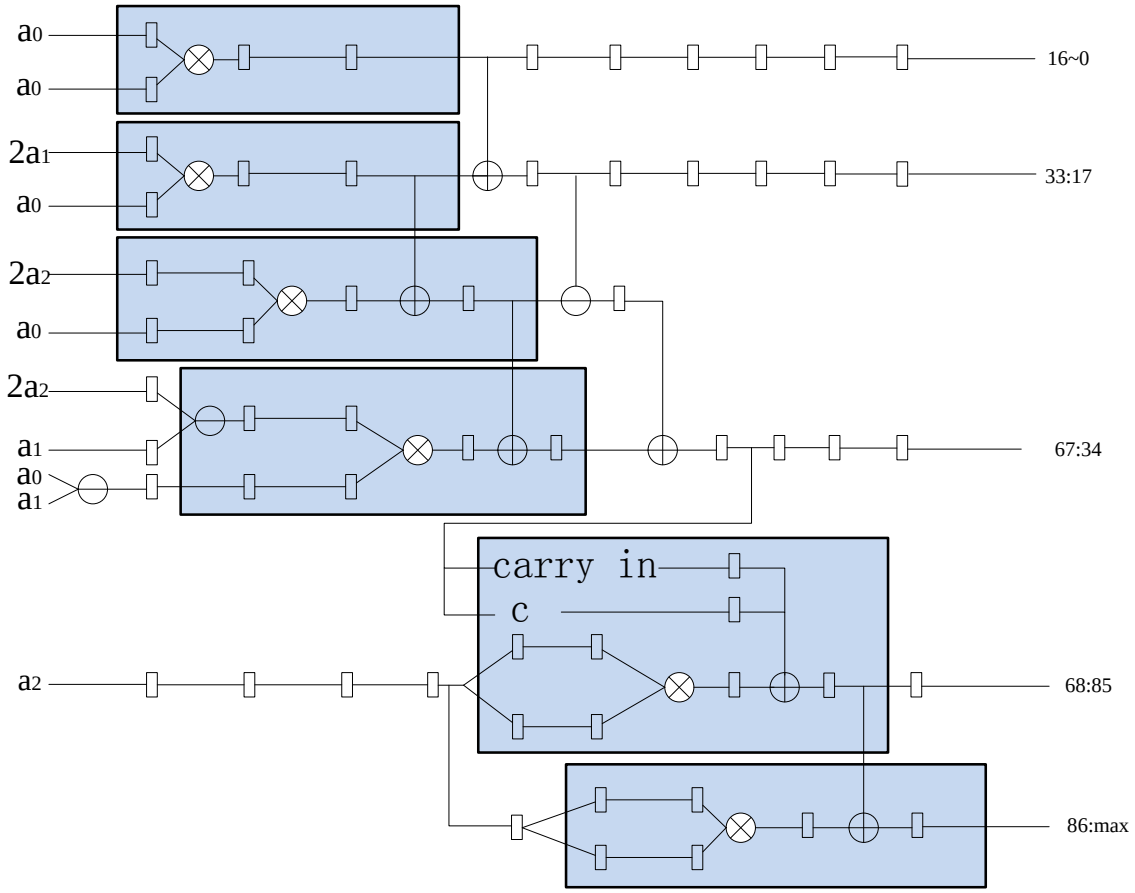

Figure A.2: Schematic of squarers for input larger than 53 bits.

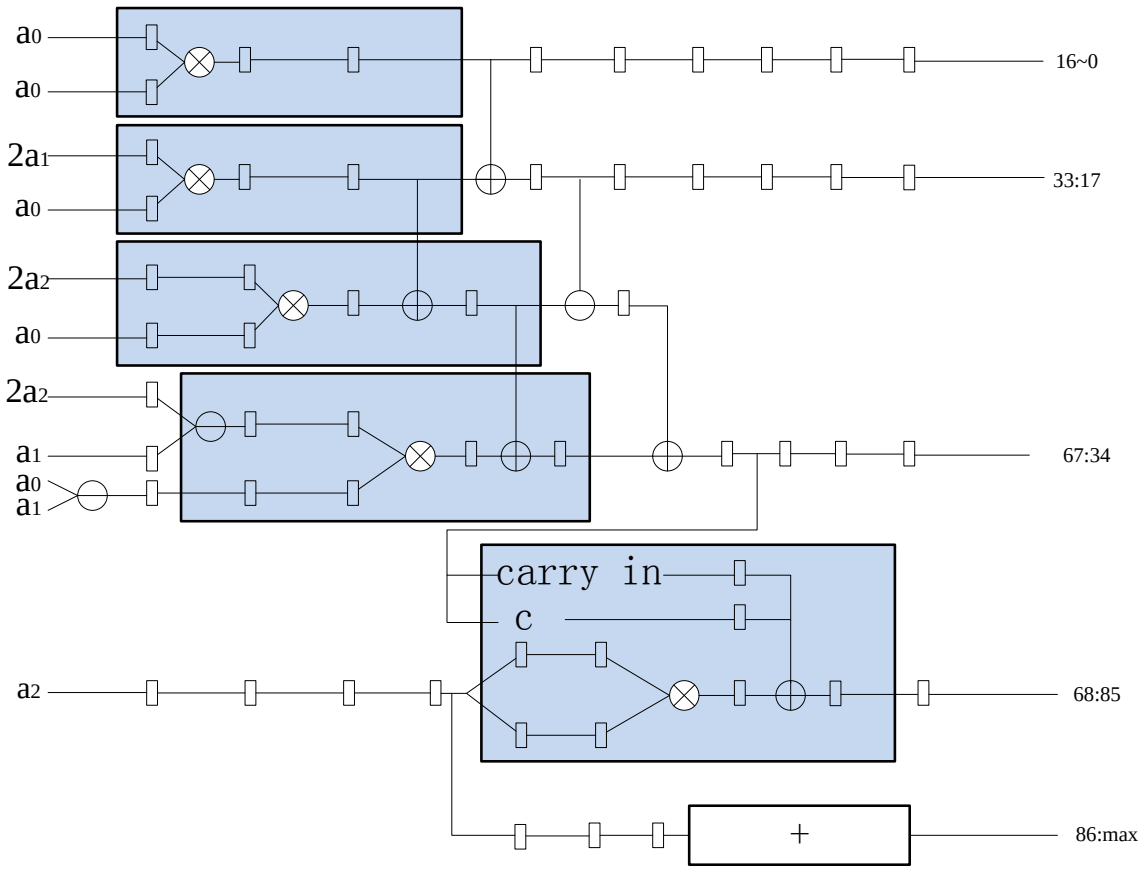

Figure A.3: Schematic of 53×53 bit squarer.

	LUT				DSP			
	multiplier	cascaded/flopoco	tiling	proposed	multiplier	cascaded/flopoco	tiling	proposed
42	0	0	N/A	107	5	6	N/A	5
43	0	0	127	109	9	6	5	5
44	0	0	131	111	9	6	5	5
45	0	0	138	113	9	6	5	5
46	0	0	147	115	9	6	5	5
47	0	0	158	117	9	6	5	5
48	0	0	172	119	9	6	5	5
49	0	0	183	121	9	6	5	5
50	0	0	202	123	9	6	5	5
51	0	0	211	125	9	6	5	5
52	0	0	216	127	9	6	5	5
53	0	0	220	172	10	7	5	5
54	0	0	N/A	152	10	7	N/A	6
55	0	0	N/A	154	10	7	N/A	6
56	0	0	N/A	156	10	7	N/A	6
57	0	0	N/A	158	10	7	N/A	6
58	0	0	N/A	160	10	7	N/A	6
59	0	0	N/A	219	10	10	N/A	8
60	0	0	N/A	221	16	10	N/A	8
61	0	0	N/A	223	16	10	N/A	8
62	0	0	N/A	226	16	10	N/A	8
63	0	0	N/A	228	16	10	N/A	8
64	0	0	N/A	230	16	10	N/A	8

Table A.1: Post place and route hardware resource for all types of squarers.

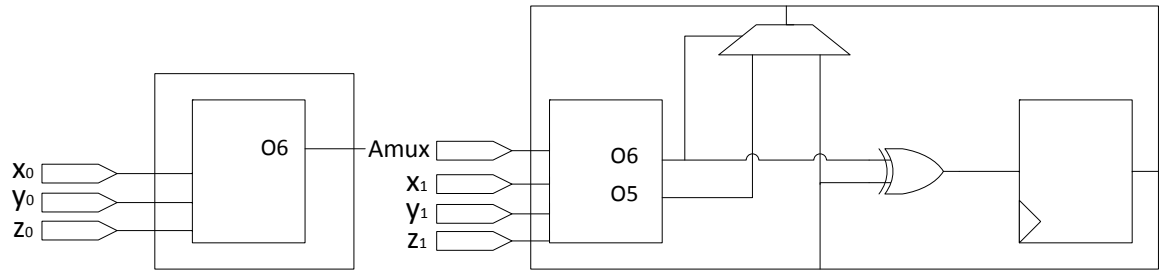

Figure A.4: Schematic of 3:1 compressor using two LUTs.

```
#coefficients from Sollya
a5=-0.11075496673583984375;
a4=4.05604408681392669677734375;
a3=0.19020896436995826661586761474609375;
a2=-4.93145830230196224874816834926605224609375;
a1=-0.11563330119172121612791670486330986022949218750;
a0=0.99932238458834951599918383635667851194739341735839;

#Horner's Rule ideal
Mp=(((((a5*Mx+a4)*Mx+a3)*Mx+a2)*Mx+a1)*Mx+a0);

#Horner's Rule truncate
a_5 = fixed<-26, dn> (a5);
a_4 = fixed<-28, dn> (a4);
a_3 = fixed<-41, dn> (a3);
a_2 = fixed<-42, dn> (a2);
a_1 = fixed<-54, dn> (a1);
a_0 = fixed<-57, dn> (a0);

x_32 = fixed<-32, dn> (Mx);
x_42 = fixed<-42, dn> (Mx);
x_52 = fixed<-52, dn> (Mx);

p5 fixed<-33, dn> = (a_5*x_32);
ps5 = p5+a_4;
p4 fixed<-37, dn> = (ps5*x_42);
ps4 = p4+a_3;
p3 fixed<-48, dn> = (ps4*x_52);
ps3 = p3+a_2;
p2 fixed<-50, dn> = (ps3*x_52);
ps2 = p2+a_1;
p1 fixed<-63, dn> = (ps2*x_52);
p fixed<-52, dn> = p1+a_0;
```

Listing A.1: Source code to define evaluation error for Horner's Rule.

```
#Estrin ideal
Mq=Mx*Mx*Mx*Mx*(a5*Mx+a4)+Mx*Mx*(a3*Mx+a2)+(a1*Mx+a0);

#Estrin truncate
x_f = fixed<-39,dn> (Mx);
x_sq_f fixed<-52,dn> = x_52*x_52;
x_sq_h = fixed<-52,dn> (x_sq_f);
x_q fixed<-52,dn> = x_sq_h*x_sq_h;

q0 fixed<-52,dn> = a_5*x_f;
q0s = q0+a_4;
q5 fixed<-52,dn> = x_q*q0s;

q2 fixed<-52,dn> =a_3*x_f;
qs2 = q2+a_2;
q3 fixed<-52,dn> = x_sq_f*qs2;

q4 fixed<-52,dn> =a_1*x_52;
qs4 = q4+a_0;

q=q5+q3+qs4;

#proof
{Mx in [0,0.00390625] -> |Mp-p| in ? /\ |Mp-q| in ? }

#hint
Mp -> Mq;
Mx*Mx*Mx*Mx -> Mx*Mx*(Mx*Mx);
```

Listing A.2: Source code to compute evaluation error for Estrin's method.

```
Results for Mx in [0, 0.00390625]:
|Mp - p| in [0, 75620484737189591b-108 {2.33024e-16, 2^(-51.9304)}]
|Mp - q| in [0, 70643622084607b-98 {2.22912e-16, 2^(-51.9944)}]
```

Listing A.3: Verification of evaluation error for Estrin's method.

```
#Proposed method ideal
Mm5 = a3/(2*a5);
Mm4 = a2/(2*a4);
Mn5 = a1-a3*a3/(4*a5);
Mn4 = a0-a2*a2/(4*a4);
Mr = ((Mx*Mx+Mm5)*(Mx*Mx+Mm5)*a5+Mn5)*Mx+((Mx*Mx+Mm4)*(Mx*Mx+Mm4)*a4+Mn4)
    ;

#Proposed method truncate
m5 fixed<-40,dn> = a3/(2*a5);
m4 fixed<-54,dn> = a2/(2*a4);

n5 fixed<-50,dn> = a1-a3*a3/(4*a5);
n4 fixed<-56,dn> = a0-a2*a2/(4*a4);

x_50 = fixed<-50,dn> (Mx);
x_sq fixed<-43,dn> = x_50*x_50;
x_sq_f fixed<-56,dn> = x_50*x_50;

a5_new = fixed<-20, dn> (a5);
a4_new = fixed<-26, dn> (a4);

rs1 = x_sq+m5;
r1 fixed<-43,dn> = rs1*rs1;
r2 fixed<-45,dn> = r1*a5_new;
rs2 = r2+n5;
r3 fixed<-54,dn> = rs2*x_52;

rs4 = x_sq_f+m4;
r4 fixed<-54,dn> = rs4*rs4;
r5 fixed<-56,dn> = r4*a4_new;
rs5 = r5+n4;

r=r3+rs5;

#proof
{Mx in [0,0.00390625] -> |Mp-p| in ? /\ |Mp-r| in ? }

#hint
Mp -> Mr;
```

Listing A.4: Source code to compute evaluation error for propose method.

```
1 Results for Mx in [0, 0.00390625]:
2 |Mp - p| in [0, 75620484737189591b-108 {2.33024e-16, 2^(-51.9304)}]
3 |Mp - r| in [0, 54718734920450031 5b-111 {2.10769e-16, 2^(-52.0752)}]
```

Listing A.5: Verification of evaluation error for proposed method.