# Optimization of Fixed-Point Circuits Represented by Taylor Series and Real-Valued Polynomials Including Analysis of Precision and Range

**Yu Pang**

Department of Electrical and Computer Engineering
McGill University

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical Engineering

March, 2010

## Acknowledgements

I would especially like to thank my supervisors, Dr. Radecka and Dr. Zilic who give me an opportunity to do this research in which I am really interested, and make this thesis possible in the Department of Computing and Electrical Engineering at McGill University. I sincerely cannot help expressing how I should credit this thesis to their support and guidance.

Very special thanks are also due to my dad and mum and my wife, Mrs Aolei Cui, and all of my friends in IML lab for their great support and encouragement in whole remarkable days.

Many thanks to everybody who ever gave me help and support.

**To Dad and Mum**

**&**

**Aolei Cui**

# Table of Contents

# List of Tables

# List of Figures

# Abstract

In this thesis, our research focuses on fixed-point arithmetic circuits. Fixed-point representation is important in low power Application-Specific Integrated Circuits (ASICs) and in Programmable Logic Devices (PLDs). There are two aspects of the data representation problem: the precision problem and the range problem. Both of these are addressed in this thesis. We use the new technique based on Arithmetic Transform (AT) which is a canonical and efficient representation for digital circuits to avoid the disadvantages of past methods, and design an efficient algorithm which can compose detached modules to obtain the overall AT for a complex circuit.

First the precision problem is processed. The typical imprecise circuits expressed in terms of Taylor series are addressed in our research. Imprecise factors including finite terms and input quantization are analyzed by AT, and algorithms are designed to verify and optimize imprecise circuits in terms of different constraints. A hybrid method performs range analysis to handle the range problem and allocates the smallest integer bit-widths. Having devised the individual methods for precision and range analysis, we then combine the two together to find the optimized implementation. Furthermore, we extend the method to analyze floating-point circuits and feedback circuits.

The proposed algorithms in the thesis overcome disadvantages of past explorations. They are more flexible in processing both Taylor series and multivariate polynomials and obtain more precise results, resulting in better implementations under various constraints.

# Résumé

Dans ce manuscrit, notre recherche se concentre sur les circuits de l'arithmétique à virgule fixe. La représentation à virgule fixe est un facteur important dans les applications d'une faible consommation pour les ASICs (Application Specific Integrated Circuit) ainsi que les circuits logiques programmables (PLD). Au point de la représentation des données, généralement, il y a deux aspects de problèmes dont la précision et la gamme. Dans ce manuscrit, nous adressons principalement à ces deux éléments. Une nouvelle technique basée sur une transformée arithmétique (AT) est utilisée. Ceci est une représentation canonique et efficace pour les circuits numériques qui permet d'éviter les inconvénients des méthodes passées et de concevoir un nouvel algorithme efficace afin de composer des modules détachés en obtenant une AT le plus générale pour les circuits complexes.

Un travail préliminaire sur le problème de précision est effectué. Les circuits imprécis généraux s'expriment en termes d'une série de Taylor a été mis en œuvre dans notre recherche. Y compris des facteurs imprécis tels que les termes finis, la quantification d'entrée qui est analysée par AT ainsi que les algorithmes qui sont conçus pour vérifier et optimiser les circuits imprécis en termes de contraintes différentes. Une méthode d'une façon hybride est effectuée afin de traiter le problème de la gamme et d'allouer un entier le plus petit de bit-widths. Mise au point sur les différentes méthodes pour la précision et l'analyse de la gamme, nous combinons les deux ensembles afin de trouver une implémentation optimisée. En outre, nous étendons la méthode pour analyser des circuits en virgule flottante et les circuits de rétroaction.

Les algorithmes proposés dans ce manuscrit est de surmonter les inconvénients des explorations passées. Ces algorithmes sont plus flexibles dans le traitement de la série de Taylor et des polynômes à plusieurs variables. Ceux-ci nous permettent d'obtenir les résultats plus précis ainsi d'entraîner les meilleures implémentations sous diverses contraintes.

# Chapter 1

# Introduction

*In this chapter, we first introduce the design flow for most common Integrated Circuits (ICs) and then describe verification approaches that include simulation, emulation and formal verification. Then, we state the research goals of thesis aiming at providing the solutions addressing the following three aspects of fixed-point circuit design: transform composition of a complex circuit, optimization of imprecise circuits, and range analysis.*

## 1.1 Circuit Design Flows

With the development of modern material and production techniques, integrated circuits (ICs) reached a level of complexity beyond imagination of even a few years ago. In terms of Moore's law, the number of transistors doubled every 18 months. For example, Intel's Itanium II processor contains more than $10^9$ transistors. Designing such complex circuits is a great challenge. The level of difficulties is lifted even higher by the restrictions on time-to-market. Hence, a systematic approach to design ICs is a must. Figure 1.1 outlines one of more commonly adopted approaches.

An idea for a new product originates usually from market analysis of customer needs. Then a team led by product managers describes in form of a specification the new design requirements. Once the specification is well formulated, the design process starts usually from behavioral modeling. As a result, initial algorithms are represented in hardware description languages (HDLs) like VHDL or Verilog, or even in higher abstraction languages, like SystemC. The correctness of the design refinement at this stage is checked by the comparison to the specification.

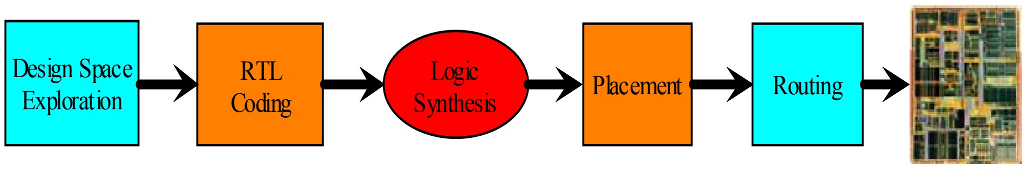


**Figure 1.1: A typical ASIC design flow**

After the behavioral model is verified, engineers generally partition the whole design into smaller and more refined blocks. Whenever possible, such blocks are often represented in terms of intellectual property (IP) cores, while HDL is used to design remaining elements at RTL coding. Once the design functionality and estimated performance satisfy the specification, the circuit is ready to be synthesized. This stage, performed automatically, often needs human intervention is terms of manual modifications necessarily such as design and insertion of boundary scan and built-in-self-test (BIST). After satisfying constraints such as timing, area and power, etc, a layout is conceived for fabrication.

## 1.2 Verification Approaches

Verification is a necessary procedure aiming to check whether the design is correct. It can be also viewed as a quality process that is used to evaluate whether or not a product, service, or system complies with a regulation, specification, or conditions imposed at the start of a development phase. In practice, verification is rarely fully completed while a given circuit is never stated fully correct since in common practice verification only shows the error presence rather than its absence.

Since errors found late in a design process can be potentially very costly, as shown in Figure 1.2, early detection is obviously critical. Hence, verification is performed at each stage of design development, from logic design, through scaling-up, to production, permeating almost all steps in ASIC design. Figure 1.3 illustrates a complete design flow for the development of an ASIC with verification.

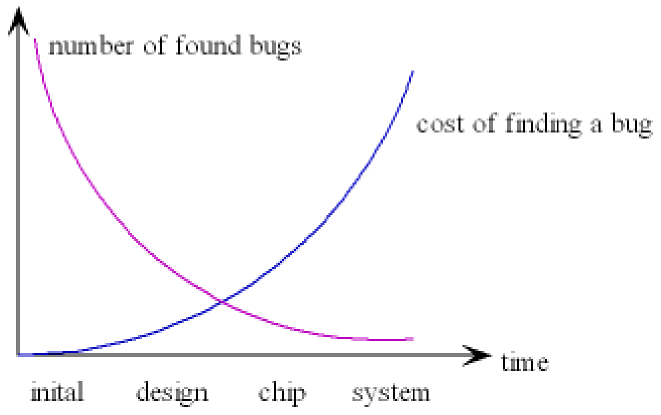


**Figure 1.2: Comparison of detection time and cost of design errors**

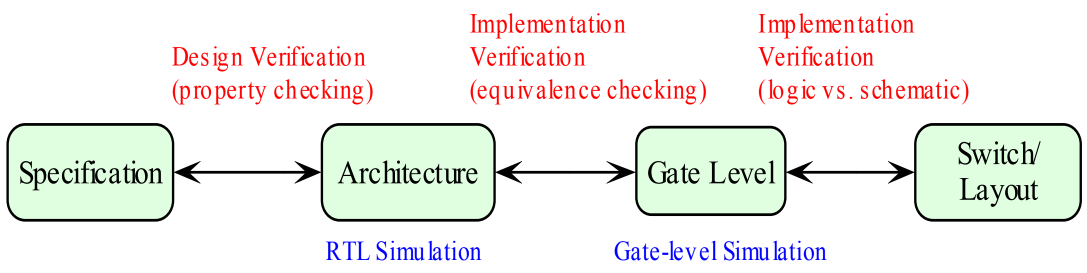

Design Verification (property checking)
Implementation Verification (equivalence checking)
Implementation Verification (logic vs. schematic)
Specification
Architecture
Gate Level
Switch/ Layout
RTL Simulation
Gate-level Simulation

**Figure 1.3: Design and implementation verification**

In general, it is estimated, that product-developing groups often spend beyond 70% of the overall design time and cost on checking the correctness of their design [157]. The graph in Figure 1.4 describes a breakdown of the effort spent in each step and Figure 1.5 shows different aspects of verification.

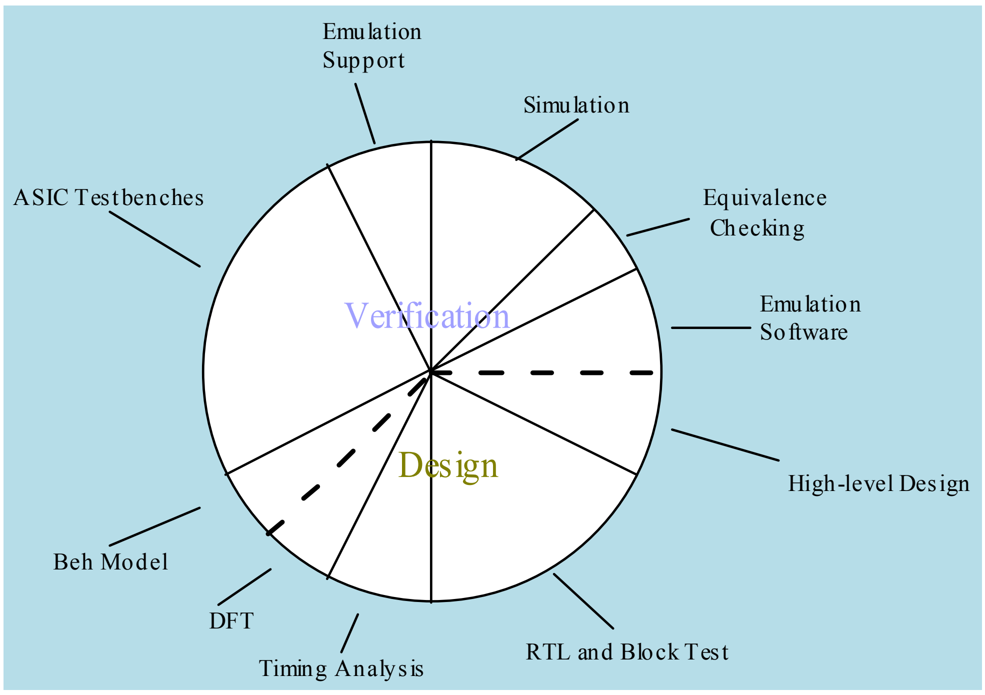


**Figure 1.4: Breakdown of effort**

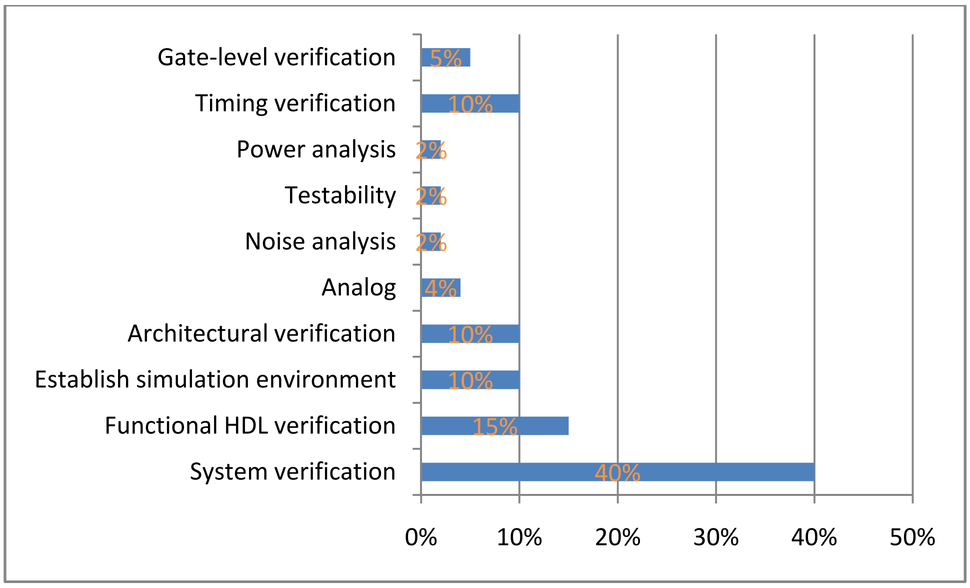


**Figure 1.5: Different aspects of verification**

From the above figure, it can be seen that time spent on verification at various stages of a design process is significant. Hence, engineers need a fast method to achieve the goal. The mainstream verification processes can be divided into three categories: simulation, emulation and formal methods.

## 1.2.1 Simulation-based Verification

Simulation is a process in which a given design is exercised by a certain set of inputs [150]. Its idea is straightforward to comprehend, and the aim is to produce a set of test vectors (stimuli) used to check the design correctness. These test sets are called testbenches (set of input vectors, expected outputs, environment constraints, etc.). More precisely, based on the module response, which is compared to the specification, the correctness of the design is assessed. Simulation can be used throughout the whole development process. Figure 1.6 describes the idea.

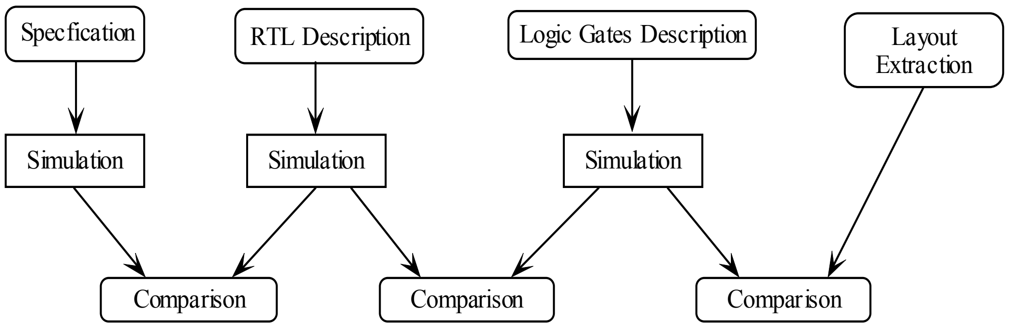


**Figure 1.6: Simulation in the development procedure**

Although the simulation method has obviously strong points, such as simplicity and easy testbench programming, there are some shortcomings we should note. First, sometimes it is not feasible to simulate all input sequences to completely verify a design. Suppose we want to test a 32-bit adder in this case - there are $2^{64}$ combinations. If it requires 1 test/us, it will take $10^{12}$ years to simulate that many vectors. Secondly, result comparison is often incomplete and it is difficult to compare results from different models and simulators. If the system grows larger, the number of possible states grows exponentially with increased number of possible event combinations. Furthermore, simulation can be effective to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

### 1.2.2 Emulation

Hardware emulation is a process that uses a piece of hardware, typically a special purpose emulation system, to imitate the behavior of a hardware system under design. As a special case, in-circuit emulation is very fast as it is performs a working target system in place of a yet-to-be-built chip, so the whole system can be debugged with live data.

High end hardware emulators provide a debugging environment with many features that can be found in logic simulators, and in some cases they even surpass their debugging capabilities [151]:

- The users can set a breakpoint and terminate the emulation process to inspect the design state, interact with the design, and resume emulation. The emulator always stops on cycle boundaries.
- The users can watch all signal or memory contents in the design without probes before the run. While visibility is provided for past time events, an emulator can access the backward time steps which may be limited in some cases by the depth of the emulator’s trace memory.
- The users can even back up time (if they save checkpoints) and re-run.

### 1.2.3 Formal Verification

Formal verification is a process of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain property using formal methods of mathematics. It can be used for verifying systems such as cryptographic protocols, combinational circuits, digital circuits with internal memory, and software expressed as source code [155].

A formal proof is necessary to verify systems based on an abstract mathematical model and the correspondence between the mathematical model and the nature of the system known by construction. Then formal verification is the process of constructing a proof that a target system will behave in accordance with its specification. Basis of formal methods, which distinguish them from simulations are:

- Formal reasoning is used to prove that an implementation satisfies a specification,

- Correctness of a formally verified hardware design holds regardless of input values,
- Exhaustive exploration of all possible behaviors is conducted,
- A counter-example (proof) is presented if the property is incorrect while if correct, all behaviors are verified;

Figure 1.7 describes the formal verification model. A verifier is utilized to check whether the system model matches the system specification. If so, the verifier sends signal of correctness; if not, the verifier gives a counterexample.

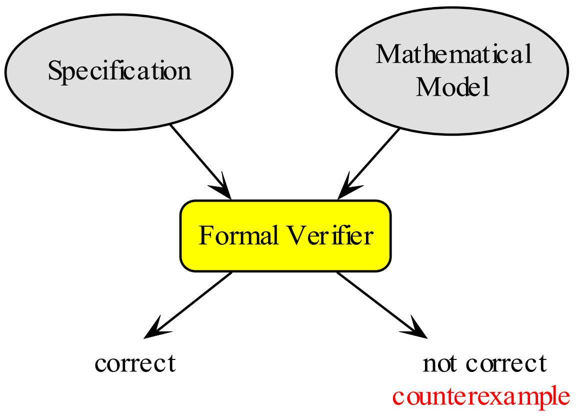


**Figure 1.7: The process model of formal verification**

Further on, formal verification schemes have many advantages:

- Complete with respect to a property,
- Avoid generating expected output sequences,
- Helpful to detect and trace errors.

Since formal verification is based on model methods which are applied when a circuit description is given by propositional temporal logic, the three most widely model-based methods are *equivalence checking*, *model checking* and *theorem proving*. Equivalence checking formally proves that two representations of a circuit design exhibit exactly the same behavior. Generally, a wide range of possible definitions of functional equivalence covers comparisons between different levels of abstraction.

- Sequential equivalence checking considers machine equivalence, which defines two synchronous design specifications functionally equivalent if they generate exactly the same sequence of output signals for all valid sequences of input signals clock by clock.
- A more general problem than equivalence checking is used to compare the

functions specified for the instruction set architecture (ISA) with a register transfer level (RTL) implementation, ensuring that the both models executing any program will cause an identical update of the memory contents.

- A system design flow requires comparison between a transaction level model (TLM) and its corresponding RTL specification. The interest in this mode of checking increases in a system-on-a-chip (SoC) design environment.

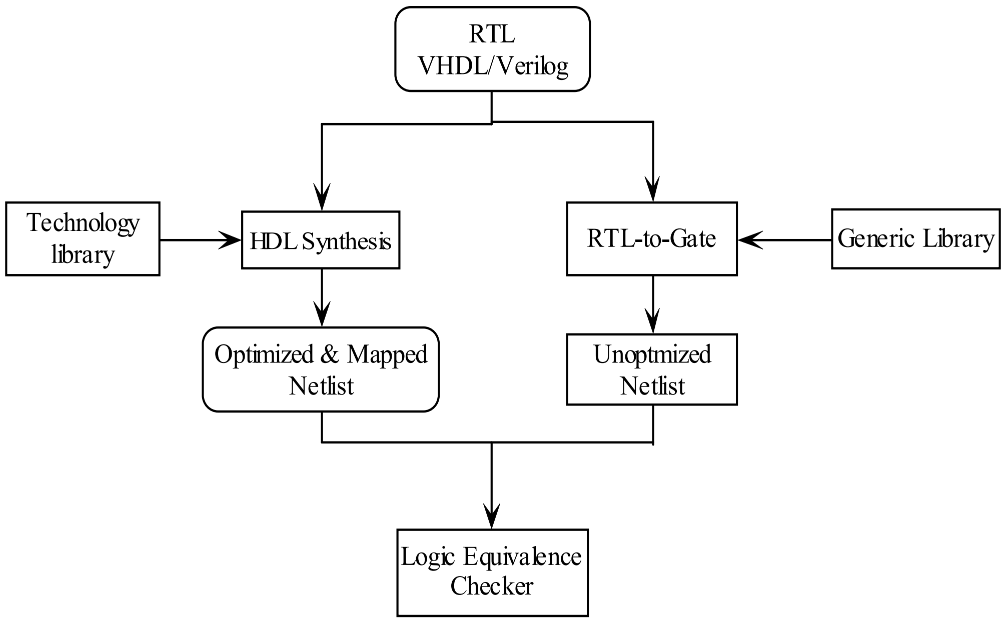


**Figure 1.8: RTL-to-gate equivalence checking**

Figure 1.8 illustrates the case of verification whether the RTL design and the modified netlist are equivalent. Because post-process often includes activities such as insertion of scan chain and some modifications, all these activities can not change the original function so equivalence checking can solve the problem.

Given a model of a system, model checking is a process of automatic test whether this model meets a given specification. The system can be hardware or software, and the specification generally contains safety requirements such as critical states that may possibly crash the system.

The system model and the specification must be described in some precise mathematical language in order to solve such a problem algorithmically. The specification is formulated using a suitable language, and the verification process checks whether a given structure satisfies a given logical formula. The general concept can be applied to all kinds of logics and suitable structures. A simple model-checking problem is to verify whether a given structure satisfies a given formula in the propositional logic and it is useful to check circuit properties such as safety and liveness property. Model checking has characteristics:

- Searches the entire solution space, for possibly infinite duration
- Responds with YES or NO (if it terminates)
- Increasingly used in industry
- Can be automated for smaller blocks or when applied earlier in the flow

Figure 1.9 illustrates the basic idea of model checking.

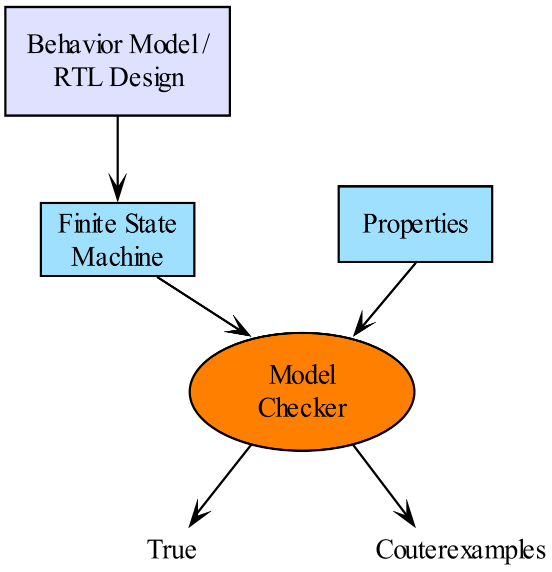


**Figure 1.9: Idea of model checking**

From above figures, we see that although model checking and simulation can both verify RTL description, simulation relies on the testbenches, while model checking relies on mathematical reasoning represented by properties and constraints. Figure 1.10 describes their difference.

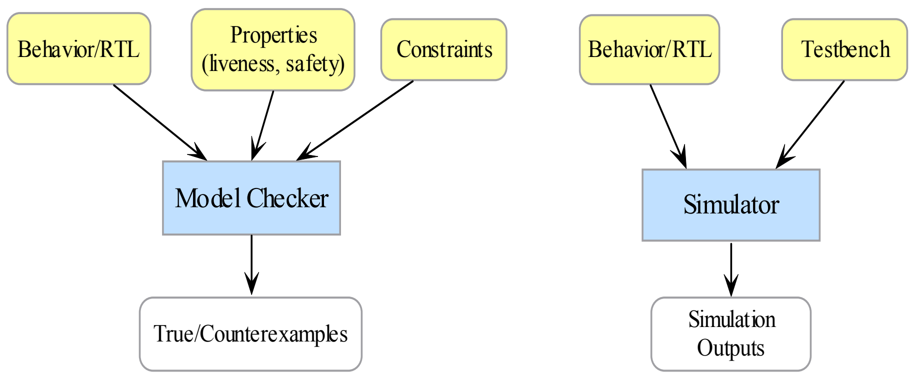


**Figure 1.10: Comparison of model checking and simulation**

Theorem proving decides whether a conjecture is a *logical consequence* of a set of statements (the *axioms* and *hypotheses*), which is used to prove that an implementation fits a specification by mathematical reasoning. The implementation and the specification are both expressed as formulas in a formal logic, and the necessary relationship - logical equivalence or logical implication - is described as a theorem to be proven within the context of a proof calculus. A proof system comprises a set of axioms and interface rules such as simplification, induction, rewriting. Authors in [159] describe how to express PSL's syntax and semantics in the PVS theorem prover and prove the correctness of a set of rewrite rules.

| **Formal Verification Tools** | | | | |
|---|---|---|---|---|
| Supplier | Tool Name | Class of Tool | HDL | Design Level |
| Commercial Tools | | | | |
| Synopsys | Formality | Euqiv. Checking | VHDL/Verilog | RTL/Gate |
| Cadence | Affirma | Euqiv. Checking | VHDL/Verilog | RTL/Gate |
| Cadence | FormalCheck | Model Checking | VHDL/Verilog | RTL |
| IBM | RuleBase | Model Checking | VHDL | RTL |
| Abstract Hardware | Lambda | Theorem Proving | VHDL/Verilog | RTL/Gate |
| Public Domain Tools | | | | |
| CMU | SMV | Model. Checking | Own Language | RTL |
| Berkely | VIS | Model/Equ. Check | Verilog | RTL |
| Cambridge | HOL | Theorem Proving | SML | Universal |

**Figure 1.11: Comparison of formal verification tools**

Figure 1.11 lists some typical tools. Although a variety of tools have been developed to perform formal verification, simulation is still a predominant method in verification because of the advantages of simple operation and relatively straightforward task of writing of testbenches.

## 1.3 Introduction of Fixed-Point Arithmetic

Fixed-point arithmetic is of importance in low power designs, embedded systems and PLDs. Although floating-point data with single or double precision can construct algorithms more accurately, generally for signal processing algorithms such as FFT and DCT initiated from real values, significant processor overhead is required to perform floating-point calculations resulting from the lack of hardware based floating-point support. This disadvantage confines the effective iteration of an algorithm. In order to improve mathematical throughput or increase the execution rate, calculations can be performed by fixed-point representations which require a virtual decimal place in between two bit locations for a given length of data [133].

The labeling convention of the representation is as follows:

$$Q\ [IB]\ .\ [FB] \qquad (1\text{-}1)$$

where *IB* = # of integer bits and *FB* = # of fractional bits.

Total number of bits used to represent the fixed-point number is yielded by the addition of integer bits *IB* and factional bits *FB*. The sum of *IB*+*FB* is known as the wordlength (WL) and this sum often corresponds to variable widths supported on a given processor. The fixed-point format includes two sections of integer and fractional content for the purpose of exploration.

### 1.3.1 Fixed-Point Range – Integer Portion

A fixed-point number is viewed as two distinct parts, the *integer content* and the *fractional content*. The integer range sets the number of *IB*, Eqn. (1-1), required to represent the integer portion of the number. *IB* itself can only hold integer values because of the binary nature of a bit. Two different methods of calculating the number of integer bits match two types of numbers, unsigned and signed.

## *A) Unsigned Integers*

The Equation (1-2) describes the unsigned integer by the minimum and maximum of any *IB* number.

$$0 \le r \le 2^{IB} - 1 \qquad (1\text{-}2)$$

*IB* can be obtained by solving the required number as:

$$IB \ge [\log_2(r+1)]$$

where *r* is the floating-point variable being ranged. The square bracket is the ceiling function.

***Example 1.1:*** *Consider an unsigned variable r = 4.346:*

$$IB = [\log_2(4.346+1)] = [2.43] = 3$$

*Three bits are required for the integer portion of r.*

## *B) Signed Integers*

The previous equations cannot represent signed variables. The changed following equation denotes the definition for the integer contents of signed numbers ($\pm r$):

$$-2^{IB-1} \le r \le 2^{IB-1} - 1$$

Please note that the signed integer type is asymmetrical about zero. For instance, a signed 8-bit value ranges from -128 to 127. By solving for the negative constraint of the equation:

$$-2^{IB-1} \le r$$

we get: $IB \ge [\log_2(-r)] + 1$

By solving for the positive constraint: $r \le 2^{IB-1} - 1$

we get: $IB \ge [\log_2(r+1)] + 1$

***Example 1.2:*** *If $r_{min}$ = -2 and $r_{max}$ = 2,*

$$IB|_{r_{\min}} \ge [\log_2(-r_{\min})] + 1 = [\log_2 2] + 1 = 2$$

$$IB\,|_{r_{\max}} \geq [\log_2(r_{\max}+1)]+1 = [\log_2 3]+1 = 3$$

*IB must be 3 bits to satisfy the two constraints concurrently.*

In the case of signed data type, the positive constraint is tighter than the negative constraint because of the asymmetry. It is common for users to define variable magnitude constraints that are symmetric about zero (for instance, $-5 \leq r \leq 5$). The computation for *IB* can be generated uniformly by the equation:

$$IB = [\log_2(\max(abs[r_{\min,}r_{\max}])+1)]+1$$

***Example 1.3:*** *Compute a signed variable* $-4.43 \leq r \leq 4.43$,

$$IB = [\log_2(\max(abs[-4.43,4.43)+1)]+1 = [\log_2 5.43]+1 = [2.45]+1 = 4$$

## 1.3.2 Fixed-Point Resolution – Fractional Portion

The number of *FB* sets the resolution for a fixed-point variable. The resolution $\varepsilon$ of a fixed-point number is given by the following equation [134]:

$$\varepsilon = \frac{1}{2^{FB}}$$

Therefore the number of *FB* required by a particular resolution is defined as:

$$FB = [\log_2 \frac{1}{\varepsilon}]$$

***Example 1.4:*** *A signed variable r= -3.2782,* $\varepsilon \leq 0.0001$,

$$FB = [\log_2 \frac{1}{0.0001}] = [\log_2 10000] = [13.288] = 14$$

The resolution is limited for a given wordlength and dynamic range of a variable. The WL of the variable must be increased to provide this resolution if a higher resolution is needed for a given range [134].

## 1.3.3 Range & Resolution

The integer and fractional parts of the number for a fixed WL consist of the full range and resolution. The combined range and resolution for an unsigned fixed-point number is defined by [133]:

$$0 \le r \le (2^{IB} - 1)\,|_{\varepsilon = 2^{-FB}}$$

The combined range and resolution for a signed fixed-point number is defined as [133]:

$$-2^{IB-1} \le r \le (2^{IB-1} - 2^{-FB})\,|_{\varepsilon=2^{-FB}}$$

The integer and fractional bits are combined together and used to determine a standard WL that is large enough to hold all integer and fractional bits as:

$$WL_{required} \ge IB + FB$$

A representation *U(IB, FB)* where *IB + FB = N* for unsigned format is denoted to calculate the value of a fixed-point format. For an unsigned format, in the *U(IB, FB)* representation, the $n^{th}$ bit, counting from right to left and beginning at 0, has a weight of $2^n / 2^{FB} = 2^{n-FB}$. Please notice that if *n = FB* the weight is 1. The value of a particular N-bit binary number *x* in a *U(IB, FB)* representation is given by the expression [134]:

$$x = (1/2^b)\sum_{n=0}^{N-1} 2^n x_n$$

where $x_n$ is the bit *n* of *x*. The range representation is from 0 to $(2^N\text{-}1)\,/2^{FB} = 2^{IB} - 2^{-FB}$. For instance, the 8-bit unsigned fixed-point representation *U(5,3)* has the form

$$b_4b_3b_2b_1b_0\,.\,b_{-1}b_{-2}b_{-3}$$

where the bit $b^k$ has a weight of $2^k$. Since *FB* is 3, the binary point is to the right of the third bit from the right (counting from zero), and hence the number has five integer bits and 3 fractional bits. This representation has a range of from 0 to $2^5 - 2^{-3} = 32 - 0.125 = 31.875$.

***Example 1.5:*** *U(6,2). This number has 6+2=8 bits and the range is from 0 to* $2^6 - 1/2^2 = 63.75$. *The value 4Bh (0100, 1011b) is:*

$$(1/2^2)\,(2^0 + 2^1 + 2^3 + 2^6) = 18.75$$

Consider an N-bit binary word *x* as *U(N,0)*. The one's complement of *x* is defined to be an operation that inverts every bit of the original value *x*. This can be performed in the *U(N,0)* representation by subtracting $x$ from $2^N$-1. That is, if we denote the one's complement of $x$ as $\tilde{x}$, then:

$$\tilde{x} = 2^N - 1 - x$$

The two's complement of $x$, denoted as $\hat{x}$, is determined by taking one's complement of $x$ and then adding one:

$$\hat{x} = \tilde{x} + 1 = 2^N - x$$

***Example 1.6:*** *The one's complement of the U(8,0) number 05h (0000,0101) by hex representation is FAh (1111, 1010). The two's complement of the U(8,0) number 05h (0000,0101) is FBh (1111, 1011).*

Considering signed two's complement fixed-point representation, we denote such a representation *A(IB,FB)* that *IB = N-FB-1*. The following expression gives the value of a specific N-bit binary number $x$ in an *A(IB, FB)* representation:

$$x = (1/2^{FB})[-2^{N-1}x_{N-1} + \sum_{n=0}^{N-2} 2^n x_n]$$

Notice that the number of bits in the magnitude of the sum above has one less bit than the equivalent prior unsigned fixed-point representation. These bits are the *N-1* least significant bits because the most significant bit in a signed two's complement number is often referred to as the sign bit.

***Example 1.7:*** *A(11, 2). This number has 11+2+1=14 bits and the range is from* $-2^{11} = -2048$ *to* $+2^{11} - 1/4 = 2047.75$.

Fundamental rules of fixed-point arithmetic are listed as follows [134].

- **Unsigned wordlength**: the number of bits required to represent *U(IB, FB)* is *IB+FB*.
- **Signed wordlength**: the number of bits required to represent *A(IB, FB)* is *IB+FB+1*.
- **Unsigned range**: The range of *U(IB, FB)* is $0 \le x \le 2^{IB} - 2^{-FB}$.
- **Signed range:** The range of *A(IB, FB)* is $-2^{IB} \le x \le 2^{IB} - 2^{-FB}$.
- **Addition operands**: Two binary numbers must keep the same scale in order to be added. That is, *X(a, b) + Y(c, d)* is only valid if *X=Y* (either

both *A* or both *U*) and *a =c* and *b= d*.

- **Addition result**: The scale of the sum of two binary numbers scaled $x(a, b)$ is $x(a+1,b)$, the sum of two *N*-bit numbers requires *N*+1bits.
- **Unsigned multiplication**: $U(IB_1, FB_1) * U(IB_2, FB_2) = U(IB_1 + IB_2, FB_1+ FB_2)$.
- **Signed multiplication**: $A(IB_1, FB_1) * A(IB_2, FB_2) = U(IB_1 + IB_2+1, FB_1+ FB_2)$.

## 1.4 Thesis Goal and Contributions

The investigation of fixed-point representation includes two problems: range and precision. In our research, we try to explore the two problems concurrently, and propose new methods for verifying and optimizing fixed-point circuits.

### 1.4.1 Composition of AT and Extensions

The main technique in our exploration is Arithmetic Transform (AT), which is defined in the spectral domain. The exploration of the function description in a *spectral domain* aims at elevating the classical problems with the *Boolean function domain* where a *truth table* is used. Each entry to the table describes precisely the behavior of the function at a single point, and bears no relation to the function behavior in the other points of the domain. For some applications this is satisfactory, however, other like circuit verification would benefit much more if partial information about the whole function could be included in a function value at each point of its domain. In fact, it is possible to give an alternate representation of a function where the information about the function is much more global in nature. This alternate representation is in the *spectral domain*, where a number of function properties are much more easily deduced than in the Boolean domain. However, it must be stressed that the overall information content of a given function is identical regardless of the domain considered (functional or spectral), and data in one domain can be uniquely recreated from the data in the other. In spite of that, the meaning of the function parameters at each individual point of the two domains is dissimilar. In particular, the discrete nature of the data in the functional domain will generally be replaced by data in the spectral domain, which is global in nature, being influenced by the complete functional performance of the circuit or network under consideration. Therefore finding the spectral transform of the circuit is an important step to verification [56].

A straightforward way to compute the AT requires a multiplication with a matrix of size that is exponential in number of primary inputs. This is clearly an impractical proposition. Other methods, such as conversion from diagrams, usually focus on the whole circuit [92]. If a complex circuit comprises many smaller modules, it is hard to get its transform directly, and then the methods mentioned are invalid [94].

A complex circuit generally consists of modules such as adders, multipliers and similar, for which the transforms are easily obtained. If we can take advantage of the relatively simpler transforms to form the transform of the complex circuits, the gain would be significant. It was shown earlier [70] that AT could be composed out of transforms of circuit blocks by help of several extensions to AT, and we extend that work by constructing efficient algorithms and transform representations. In addition, since the AT representation only contains primary inputs and outputs, if engineers know the overall transform of the complex circuit in advance, compared to the compositional AT representation, they should be identical, and hence the composition procedure can perform equivalence checking. Therefore the process of constructing AT composition becomes very important. Because basic AT cannot represent sequential circuits, extensions are necessary for the purpose of the composition.

In this thesis, we explore AT and its extensions proposed by Zilic and Radecka [70] [158] then develop several subroutines to compose the detached transforms of smaller modules which exist within a bigger circuit, and finally integrate these subroutines into a fast algorithm for the construction of AT and its extensions.

## 1.4.2 Imprecise Circuits

Here we focus our attention on a large category of circuits which cannot be exactly represented. We will refer to these as *imprecise circuits*, as implementations do not match specifications exactly since they are only realized approximately. When dealing with arithmetic circuits, the imprecision of these circuits creates added complexity for the design and verification phase. In such cases, implementations realize intended specifications only to the certain degree of precision, adding yet another dimension to the already complex process of design verification. Also it is not compulsory to require them to be identical as some imprecision reason should and could be tolerated. While verifying arithmetic circuits, if the error measured as a difference (imprecision) between them is within an acceptable range, the implementation is deemed suitable to the specification. Mathematical forms of expressing imprecision are related to the type of implemented designs. For example, for arithmetic circuits, the error can be described in some arithmetic form, and is therefore referred as an *arithmetic error*. Figure 1.12 denotes the basic idea of imprecise circuits. The solid line represents the specification, and the dotted lines represent the implementations. The implementations approximate the specification but not exactly overlap.

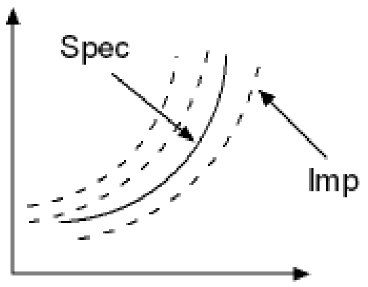


**Figure 1.12: The basic idea of imprecise circuits**

Mathematical forms of expressing imprecision are related to the type of implemented designs. For example, for arithmetic circuits, the error can be described in some arithmetic form, and is therefore referred as an *arithmetic error*.

The current verification methods, such as equivalence checking cannot be applied: in some cases, many output bit values may differ, while the implementation might still be considered correct if the difference of the specification and the implementation is within a given arithmetic precision. Consider, for example, the representation of value 1.0. It is approximation 0.111… can be made arbitrarily precise by increasing the wordlength, yet all the bits are incorrect. On the other hand, the change of a single, most significant bit can change the arithmetic value by 100%.

Further, when verifying the precision, we must explore yet another problem dimension, i.e., the imprecision for the whole domain of definition. In the thesis, we address the problem by the following two aspects.

### *(A) Component Comparison*

The functionality of many circuits, particularly signal processing ones, can be described or approximated by polynomials. For instance, many algorithms use a common arithmetic function such as *sin(X)*. This function, being a real-type and infinite, cannot be realized precisely, and hence some kind of approximation is needed, like, for example, the following one:

$$X - X^3/3! + X^5/5! - X^7/7! \ldots.$$

Here $X$ is within the range [-1, 1] for convergence.

In many cases the implementation of the specification function, like the above is not build from scratch. More realistic problem is to realize the function by, for example, using only 6 terms and 16-bit inputs approximation, where there is an existing module to implement sin($X$) by 5 terms and 12-bit. The existing implementation can be used, as long as the difference between the requirement and the library element is not beyond the given error bound. However, to minimize the error of such a substitution, the Taylor terms and bit-width must be both optimized.

We will approach the Taylor terms and input bit-width optimization simultaneously, and try to provide a uniform platform, which is easily operated and applied. Our goal is to match and verify the precision of real DSP/arithmetic modules such as DCT. For this purpose, we present a method for matching imprecise datapath circuits expressed by Taylor series and extend it to handle word-level polynomials. Such representations are selected based on the fact that Taylor expansions provide a representation of arithmetic functions, which not only can be made arbitrarily close to the desired (specified) function, but also give an elegant solution to the verification of imprecise designs. Therefore, we devise a flexible tool based on *Arithmetic Transforms* that can assist engineers to compute imprecision between two implementations easily.

Figure 1.13 describes two components with difference. If their imprecision, that is, the maximum error, is smaller than the given error bound, the two components can be substituted by each other. This problem is solved in section 6.1.

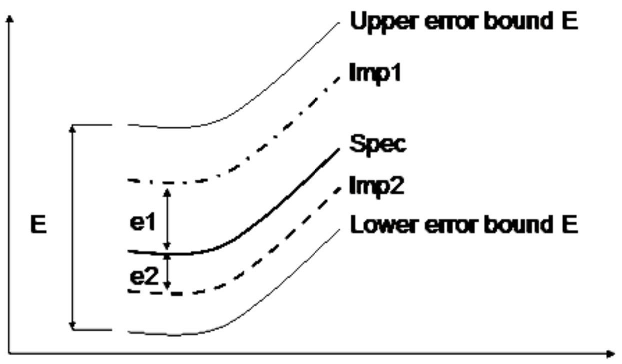


**Figure 1.13: Comparison of two implementations**

## *(B) Precision Verification and Optimization*

From the design perspective, however, the imprecision can provide yet another optimization resource, similar in nature to the notion of “don't cares” in logic synthesis. In particular, as implementations do not need to match specifications exactly, one can search for the least expensive implementation within the allowed imprecision.

Given an implementation with a group of parameters such as Taylor terms and input bit-width, engineers have interest to know what difference between the implementation and the specification. So we need to develop fast algorithms to compute the imprecision and verify whether the implementation fits the specification according to the given error bound.

An approximate implementation is required to realize a real-valued function such as *sin*(*X*) by fixed-point circuits. Traditionally, one mostly relies on simulation-based, or *dynamic* methods, to analyze the imprecision between the specification and the implementation. In essence, one has to explore the whole domain the function definition, with many precision parameters investigated concurrently to get the imprecision. We propose a new method in terms of Arithmetic Transform (AT) to analyze these parameters *statically,* to ascertain whether the existing implementation is suitable to the specification. Please note that many satisfying implementations can fulfill one specification, and it is very much worth finding the implementation with the smallest hardware cost. In Figure 1.14, the three dotted lines represent three implementations which all satisfy the specification represented by the solid line, but only one implementation has the smallest area. How to find out this optimized implementation is attractive in practical engineering.

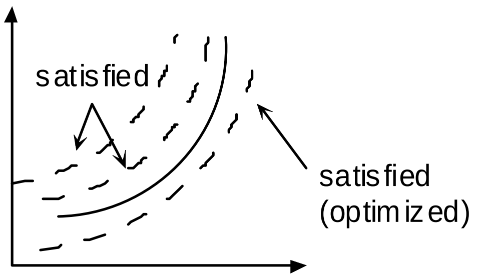


**Figure 1.14: Optimized implementation with the smallest area**

In the thesis we try to analyze the factors generating imprecision such as function approximation and finite bit-widths, and develop a series of algorithms to process imprecise circuits included comparison, verification and optimization. This problem is solved in section 6.2 – 6.4.

## 1.4.3 Range Analysis

Range analysis is a significant step in RTL synthesis which directly influences cost and performance. This topic is always hot and attractive to engineers. Traditional methods have obvious disadvantages of low efficiency and coarse bounds, which lead to infeasibility and additional bits for data representation. In order to overcome these disadvantages, we propose a new method to calculate ranges for each intermediate output and the final output in the datapath. This method can obtain exact ranges and allocate the smallest integer bit-widths for the datapath, so the optimized implementation with the smallest hardware area can be achieved. This problem is solved in Chapter 7.

### 1.4.4 Exploration of Fixed-Point Circuits

After investigating the precision and the range separately, we explore the fixed-point representation with both integer bit-width (IB) and fractional bit-width (FB). The case is more complex and the most important problem is how to determine the fractional bit-width in the datapath and estimate the error. Based on the above analysis, we propose an efficient method to allocate appropriate IB and FB for the inputs and all outputs in the datapath in order to obtain the optimized implementation.

As blind spots in past explorations, circuits with feedbacks – such as IIR filters – are of importance. We analyze feedback circuits and propose algorithms to detect stability and find ranges. Furthermore, sequential circuits are investigated and the process of fixed-point representation is extended to floating-point representation. These problems are solved in Chapter 8.

### 1.4.5 Contributions

On the whole, the main contributions of the thesis are in:

- designing an algorithm to obtain the spectral transform for a complex circuit
- proposing algorithms to verify and optimized imprecise circuits
- proposing an algorithm to calculate ranges of a datapath
- conceiving an algorithm to find the optimized fixed-point implementation with integer and fractional bit-widths
- designing an algorithm to explore imprecise arithmetic circuits with feedback.

# Chapter 2

# Background

*In this chapter, we review function representations including truth tables, Shanon expansion and polynomial representation. We pay special tribune to decision diagrams, as they play an important role in many classical verification methods. Most commonly used diagrams include OBDDs, MTBDDs, BMDs and TEDs. Finally, as usual methods to handle imprecise circuits rely on dynamic analysis and affine arithmetic, we conclude this chapter with the introduction of the mathematical background of these methods.*

With VLSI (Very Large Scale Integration) technologies and the design techniques developing rapidly, microchips are utilized prevalently in many areas of human activities. The integration density increases fast beyond billions of transistors bringing forward a problem: how to build a right system to fit requirements? Thus hardware verification theory emerges as an important element of the overall design process. There were many corresponding explorations in past decades. In this chapter we will review some typical theoretical background dealing with function representations and verification.

## 2.1 Function Representations

Digital combinational circuits rely on the repreentation of *Boolean functions*, either by means of *computation* or *evaluation processes*. *Truth tables* belong to the first group, while *decision diagrams* belong to the second one.

### 2.1.1 Truth Table

A *truth table* is a mathematical table used in logic — specifically in connection with Boolean algebra, Boolean functions, and propositional calculus — to compute the functional values of logical expressions on each of their functional arguments, that is, on each combination of values taken by their logical (input) variables. In particular, truth tables can be used to tell whether a propositional expression is true for all legitimate input values, that is, logically valid.

***Example 2.1:*** *The truth table of the 2-bit unsigned adder with inputs $x = x_1x_0$ and $y = y_1y_0$, and output $z = z_2z_1z_0$ is presented below.*

| $x_1x_0y_1y_0$ | $z_2z_1z_0$ |
|---|---|
| 0 0 0 0 | 0 0 0 |
| 0 0 0 1 | 0 0 1 |
| 0 0 1 0 | 0 1 0 |
| 0 0 1 1 | 0 1 1 |

| | |
|---|---|
| 0 1 0 0 | 0 0 1 |
| 0 1 0 1 | 0 1 0 |
| 0 1 1 0 | 0 1 1 |
| 0 1 1 1 | 1 0 0 |
| 1 0 0 0 | 0 1 0 |
| 1 0 0 1 | 0 1 1 |
| 1 0 1 0 | 1 0 0 |
| 1 0 1 1 | 1 0 1 |
| 1 1 0 0 | 0 1 1 |
| 1 1 0 1 | 1 0 0 |
| 1 1 1 0 | 1 0 1 |
| 1 1 1 1 | 1 1 0 |

Truth tables are useful in many synthesis applications, as well, as verification due to their canonical property. In fact, equivalence checking of two Boolean functions can be done by comparing truth tables of corresponding functions.

A truth table has $2^N$ rows for an $N$-input function, hence the size and time complexity are always exponential in the number of primary inputs. Consequently, the truth table as a binary function representation is impratical for verificaiton of even modertate size circuits.

### 2.1.2 Shannon Expansion

In mathematics, Shannon expansion is a method by which a Boolean function can be represented by the sum of two sub-functions (co-factors) of the original. It provides a way for deriving a Boolean function recursively.

**Definition 2.1:** *The cofactor of a Boolean function $f(x_0, x_2, ..., x_i, ..., x_{n-1})$ with respect to variable $x_i$ is $f_{x_i} = f(x_0, x_1, ..., 1, ..., x_{n-1})$. Similarly, the cofactor with respect to variable $\overline{x_i}$ is $f_{\overline{x_i}} = f(x_0, x_1, ..., 0, ..., x_{n-1})$.*

Each Boolean function can be represented by its cofactors through Shannon expansion.

**Theorem 2.1:** *A Boolean function $f : B^n \to B$ can be represented as*

$$f_{x_i} = f(x_1, x_2, ..., x_i, x_n) = x_i \cdot f_{x_i} + \overline{x_i} \cdot f_{\overline{x_i}} = (x_i + f_{\overline{x_i}}) \cdot (\overline{x_i} + f_{x_i})$$

One way of representing the Shannon's expansion is by means of a multiplexer selects between the two cofactors, depending on the value of a splitting variable $x_i$.

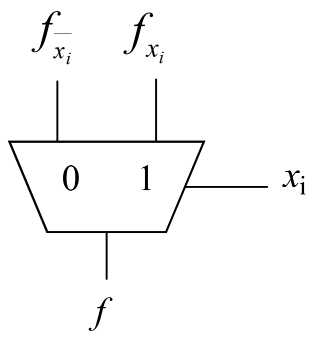


**Figure 2.1: Shannon expansion in variable $x_i$**

***Example 2.2:*** *Given a function of $f = xyz + xy'z + x'y'z + x'y'z'$, we can re-write the function in terms of any two variables — namely, a variable and its complement: $f = xg_x + x'g'_x$. Simply apply the distributive theorem to the function about x: $f = x'(y'z + y'z' + yz) + x(yz + y'z)$. Now we have expanded the function f about the variable x. The work [154] describes a method based on Shannon expansion for low- power and testable circuit synthesis.*

## 2.1.3 Polynomial Representation

Positive and negative Davio expansions are other two expressions of Boolean functions by means of cofactors and the XOR operation.

**Definition 2.2:** *The positive Davio expansion of a Boolean function $f(x_0, x_2, \ldots, x_i, \ldots, x_{n-1})$ with respect to variable $x_i$ is:*

$$f = f(x_0, x_1, \ldots, x_i, \ldots x_{n-1}) = f_{\overline{x_i}} \oplus x_i \cdot (f_{\overline{x_i}} \oplus f_{x_i})$$

*Similarly, the begative Davio expansion is:*

$$f = f(x_0, x_1, \ldots, x_i, \ldots x_{n-1}) = f_{x_i} \oplus \overline{x_i} \cdot (f_{\overline{x_i}} \oplus f_{x_i})$$

The two representations adopt XOR operations over two cofactors. They are useful for polynomial expressions and decision diagrams representations.

If all variables are decomposed by positive Davio expansion, another canonical representation of Boolean functions is obtained as Reed-Muller transform [4], [5], [6]. RM transform is used in technology mapping by symmetry detection, which will be introduced in section 3.1.2.

### 2.1.4 Boolean Satisfiability

Boolean Satisfiability (SAT) is often used as the underlying model for a significant and increasing number of applications in electronic design automation (EDA) as well as in many other fields of computer science and engineering. Satisfiability determines whether the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to TRUE. Another importance is to determine whether no presence of such assignments would imply that the function expressed by the formula is identically FALSE for all possible variable assignments. In this latter case, we say that the function is unsatisfiable, or else it is satisfiable [152] .

The SAT is a decision problem in complexity theory, whose instance is a Boolean expression written using operations of AND, OR, NOT, variables, and parentheses. The question is that given the expression, whether some assignment of *TRUE* and *FALSE* values to the variables will make the entire expression true. In particular, satisfiability searches are most often applied to Boolean functions represented as product of sums. The search for a function variables assignment, which would make all the clauses true, is proven to be NP-Complete [152].

***Example 2.3:*** *After converting Boolean equations from Example 2.1 into product-of-sums, we obtain the following set of clauses:*

$$f(x_1,x_0,y_1,y_0)=\begin{bmatrix} z_2 \\ z_1 \\ z_0 \end{bmatrix}=\begin{bmatrix} (x_1+x_0)(y_1+y_0)(x_1+y_1)(x_1+y_0)(x_0+y_1) \\ (x_1+y_1+y_0)(x_1+x_0+y_1)(\bar{x}_1+\bar{y}_1+y_0)(\bar{x}_1+x_0+\bar{y}_1)(\bar{x}_1+\bar{x}_0+y_1+\bar{y}_0)(x_1+\bar{x}_0+\bar{y}_1+\bar{y}_0) \\ (x_0+y_0)(\bar{x}_0+\bar{y}_0) \end{bmatrix}$$

*The set of input assignments satisfying the above equations is empty. This fact is easy to verify by checking the multiplier truth table, which holds no input* $(x_1, x_0, y_1, y_0)$ *assignment resulting in all the output bits* $(z_2, z_1, z_0)$ *being equal to one.*

## 2.2 Decision Diagrams

Decision diagrams are the binary function representations that explore evaluation process. They do not need to compute the response of input stimuli and evaluate a function based on a set of binary-valued decisions.

### 2.2.1 Binary Decision Diagrams

Binary decision diagram (BDD) [7] was already introduced in 1959 as a data structure that is used to represent a Boolean function. Furthermore, under the name of Branching Programs they were intensively studied in theoretical computer science. Within the following years the importance of BDDs for VLSI CAD was realized by several groups, and an increasing number of BDD algorithms and successful applications were reported.

On a more abstract level, BDDs can be considered as a compressed representation of sets or relations. Unlike other compressed representations, operations are performed directly on the compressed representation, i.e. without decompression. BDDs are based on the Shannon expansion. Generally, bit-level decision diagrams are constructed in terms of one of the three Boolean function decompositions:

$$\textit{Shannon:}\quad f = \overline{x_i}\cdot f_{\overline{x_i}} \oplus x_i \cdot f_{x_i}$$

$$\textit{positive Davio:}\quad f = f_{\overline{x_i}} \oplus x_i \cdot (f_{x_i} \oplus f_{\overline{x_i}})$$

$$\textit{negative Davio:}\quad f = f_{x_i} \oplus \overline{x_i} \cdot (f_{x_i} \oplus f_{\overline{x_i}})$$

**Definition 2.3:** "*A Decision Diagram (DD) over a set of Boolean variables $X_n$ and a non-empty terminal set T is a connected, directed acyclic graph G=(V, E) with exactly one root and the following properties:*

- *A vertex in V is either a non-terminal or a terminal vertex.*
- *Each non-terminal vertex v is labeled with a variable from $X_n$, called the index index(v) of v and has exactly two successors in V , denoted by low(v), high(v).*
- *Each terminal vertex v is labeled with a value $value(v) \in T$ and has no successors." [7]*

***Example 2.4:*** *Consider Decision Diagrams in Figure 2.2 and 2.3. The graph in Figure 2.2 represents a complete tree that by definition is also a complete and ordered DD. The DD in Figure 2.3 is also ordered, but not complete. Since both DDs are ordered they are also free.*

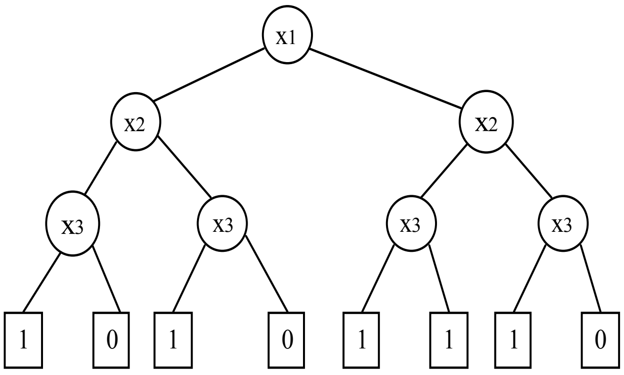


**Figure 2.2: Complete and ordered DD**

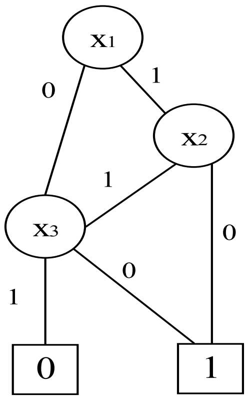


**Figure 2.3: Ordered DD**

**Definition 2.4:** "*A BDD is a DD over $X_n$ and terminal set T={0, 1}. If the BDD has a root vertex v, then the BDD represents a Boolean function $f_v$ defined as follows:*

1. *If v is a terminal vertex and value(v)=1 (value(v)=0), then $f_v=1$ ($f_v = 0$).*
2. *If v is a non-terminal vertex and index(v)= $x_i$, then $f_v$ is the function*

$$f_v(x_1,...,x_n) = \overline{x_i} \cdot f_{low(v)}(x_1,...,x_n) + x_i \cdot f_{high(v)}(x_1,...,x_n).$$

*$f_{low(v)}$ ($f_{high(v)}$) denotes the function represented by low(v) (high(v))." [7]*

## 2.2.2 Reduced Ordered Binary Decision Diagrams

BDDs have obvious limitations because of exponential sizes which confine applications. Some extensions have been proposed to overcome these limitations. Recently, (especially in the area of verification) DDs have also been used to represent Pseudo-Boolean functions, i.e., function of the form $f$ : $B^n \to Z$. The simplest extension of BDDs, ROBDDs (Reduced Ordered Binary Decision Diagrams), has two restrictions:

- Appearance of the variable keeps in the same order along each path from the root to a terminal.
- No isomorphic sub-trees or redundant vertices exist.

**Definition: 2.5:** "*Let $\pi$ be a total order on the set of variables $x_1,\ldots x_n$. An ordered binary decision diagram (OBDD) with respect to the variable order $\pi$ is a directed acyclic graph with exactly one root which satisfies the following properties:*

- *There are exactly two nodes without outgoing edges. These two nodes are labeled by the constants 1 and 0, respectively, and are called sinks.*
- *Each non-sink node is labeled by a variable $x_i$, and has two outgoing edges, which are labeled by 1 and 0, respectively. These edges are called the 1-edge and the 0-edge, respectively.*
- *The order, in which the variable appear on a path in the graph, is consistent with the variable order $\pi$, i.e., for each edge leading from a node labeled by $x_i$ to a node labeled by $x_j$ it holds $x_i <_\pi x_j$." [7]*

An OBDD is a read-once branching program with an additional ordering restriction on the variables. The computation path of an input $a = (a_1 ,\ldots, a_n) \in B^n$ is the path from the root to a sink in the OBDD which is defined by the input. More precisely, the computation path begins in the root, and in each node labeled by $x_i$ the path follows the edge with label $a_i$.

***Example 2.5:*** *Let $\pi$ be the variable order $x_1 < x_2 < x_3$. Figure 2.4 illustrates two OBDD representations of the function $f(x_1, x_2, x_3) = x_1x_2 + \overline{x_3}x_1\overline{x_2}$ with respect to the order $\pi$.*

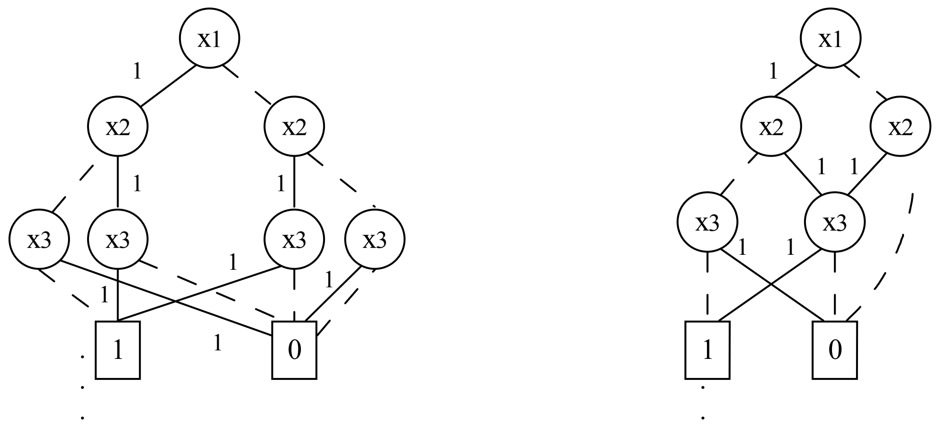


**Figure 2.4: Two OBDDs of Example 2.5**

**Definition 2.6: "***Two OBDDS of $P_1$ and $P_2$ are isomorphic if there is a bijective mapping $\phi$ from the set of nodes of $P_1$ to the set of nodes of $P_2$ such that, for each node v, the two nodes v and $\phi(v)$ are sinks with identical labels, that means var(v)=var($\phi(v)$), $\phi(high(v)) = high(\phi(v))$, $\phi(low(v)) = low(\phi(v))$. An OBDD is called reduced if*

1. *it does not contain a node v with high(v) = low(v), and*
2. *there does not exist a pair of nodes u, v such that the sub-OBDDs rooted in u and v are isomorphic."[7]*

***Example 2.6:*** *Consider a Boolean function f = $x_1x_2x_3 + x_4x_5x_6 + \ldots + x_{n-2}x_{n-1}x_n$. The ROBDD $G_1$ for f with variable ordering $x_1$, $x_2 \ldots x_{n-1}$, $x_n$ is given in* Figure 2.5*. The size of the corresponding graph is given by $|G_1| = n$. Since f depends on all n variables the ROBDD has optimal size.*

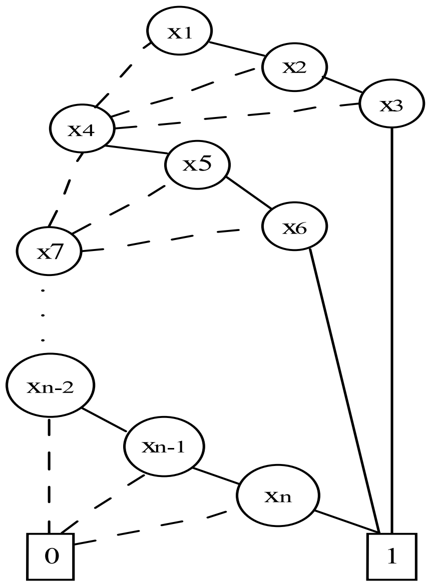


**Figure 2.5: An example of ROBDD**

## 2.2.3 Multi-Terminal BDDs

Another extension of BDDs to aim on handling word-level values is to introduce non-Boolean terminals, i.e, to allow integers in terminal nodes. The resulting DDs are called Multi-Terminal BDDs (MTBDDs) [8] if in each node an (integer-valued) Shannon decomposition is carried out.

***Example 2.7:*** *A MTBDD for function $f=3x_1+x_2$ is given in Figure 2.6.*

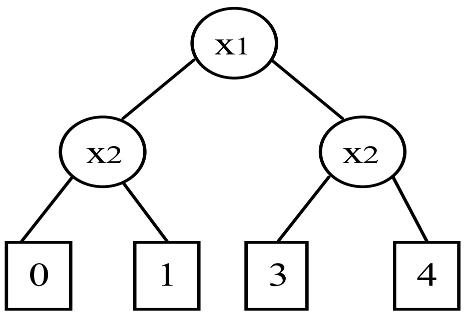


**Figure 2.6: MTBDD for *$f=3x_1+x_2$***

## 2.2.4 Binary Moment Diagrams

Binary Moment Diagrams (*BMDs) [9] [10] [11], which belong to the class of word-level decision diagrams, are generalizations of the BDD to linear functions over domains such as Boolean, but also to integers or to real numbers. They can deal with Boolean functions with complexity comparable to BDDs, but also some functions that are dealt with very inefficiently in a BDD are handled easily by BMD, most notably multiplication. The most important properties of BMD is that, like with BDDs, each function has exactly one canonical representation, and many operations can be efficiently performed on these representations. The main features that differentiate BMDs from BDDs are using linear diagrams instead of pointwise diagrams, and having weighted edges. No node may have all decision parts equivalent to 0 (links to such nodes should be replaced by links to their always part). No edge may have weight zero (all such edges should be replaced by direct links to 0). Weights of the edges should be coprime. Without this rule or some equivalent of it, it would be possible for a function to have many representations, for example 4*x*+4 could be represented as 4*(1+*x*) or 1*(4+4*x*).

*BMDs are particularly effective for representing digital systems at the word level, where sets of binary signals are interpreted as encoding integer (fixed point) or rational (floating point) values. Common integer and floating point encodings have efficient representations as *BMDs, as do operations such as addition and multiplication. *BMDs can also represent Boolean functions as a special case, with size comparable to BDDs .

***Example 2.8:*** *A *BMD for the fractional coding (3 bits) is illustrated as:*

$$f_{enc}(x_3, x_2, x_1) := [x_3, x_2, x_1] := \sum_{i=1}^{3} 2^{-i} x_i$$

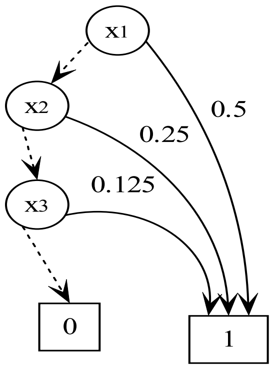


**Figure 2.7: *BMD for unsigned fractional encoding**

Edge weighting leads to a much more concise representation of a function. As an illustration, Figure 2.7 describes the representations of *BMD for the same function.

## 2.2.5 Taylor Expansion Diagrams

A new type of diagram, Taylor Expansion Diagram (TED) [12] – [15], has been developed to solve the problem of word-level computation, such as *A[0:n-1]+B[0:n-1],* requiring the decomposition of the function with respect to each bit-level variable *A[0],…,A[n-1],B[0],…,B[n-1]*. It is unnecessary to expand the word-level variables when treating them as algebraic symbols. Figure 2.8 depicts the decomposition with respect to the word-level variables A and B. If we group the nodes corresponding to the individual bits of these variables, we can abstract the integer variables and use them directly in the design. The figure describes the idea of symbolic abstraction of variables from bit-level components [12].

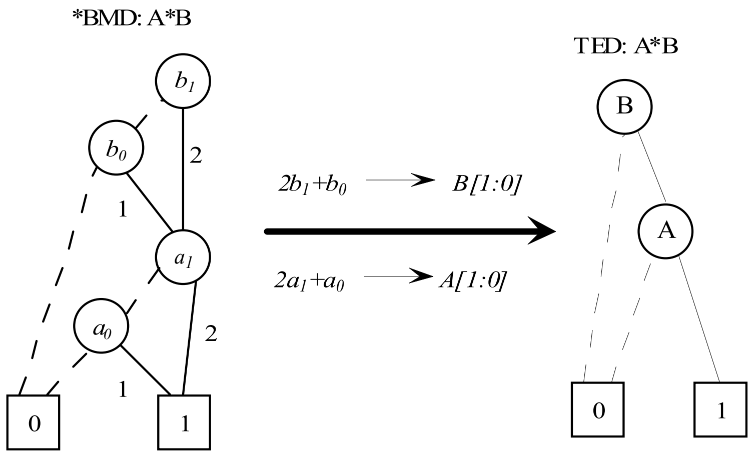

*BMD: A*B
$b_1$
$b_0$
2
1
$a_1$
$a_0$
2
1
0
1
$2b_1+b_0 \longrightarrow B[1:0]$
$2a_1+a_0 \longrightarrow A[1:0]$
TED: A*B
B
A
0
1

**Figure 2.8: Abstraction of bit-level variables into algebraic symbols**

Assume a regular algebra (R, *, +) over real numbers R with integer coefficients on a real differentiable function *f(x,y,…)*. Using the Taylor series expansion with respect to a variable *x*, the function *f* can be represented as [14]:

$$f(x, y...) = f(x = 0, y...) + xf'(x = 0, y,...) + \frac{1}{2}x^2 f''(x = 0, y...). + ..$$

where *f'(x=0, y…)*, *f''(x=0, y…)*,etc., are first, second, and higher order derivatives of *f* with respect to *x*. The derivatives of evaluated at *x*=0 are independent of variable *x*, and can be further decomposed w.r.t. the remaining variables, one variable at a time. The resulting recursive decomposition can be represented by a decomposition diagram called the Taylor Expansion Diagram.

**Definition 2.9:** **"***The Taylor Expansion Diagram, is a directed acyclic graph ($\phi$, V, E, T), representing a multi-variable polynomial expression $\phi$. V is the set of nodes and E is the set of directed edges connecting the nodes. T is the set of terminal nodes. Every node $v \in V$ has an index var(v) which identifies the decomposing variable. The variable of the TED are ordered. The function at node v is determined by the taylor series expansion at the point var(v)=0. The edge emanating from a node v point to its children nodes which correspond to the derivative of the function with respect to the variable var(v). The out-degree of a terminal node $v \in T$ is 0. The function computed at a terminal node is an integer constant c."[14]*

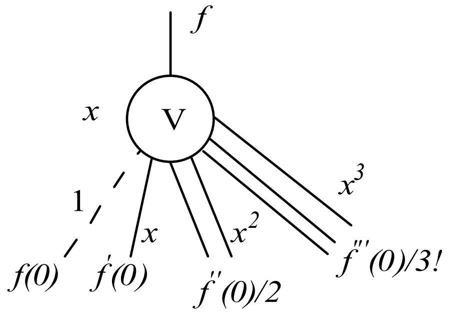


**Figure 2.9: A decomposition node in a TED [12]**

The decomposition is applied recursively to the subsequent children nodes. The $k^{th}$ derivative of a function *f* rooted at node *v* with *var(v)=x* is referred to as a *k-child* of *v; f(x=0)* is a *0-child,* $f'(x=0)$ is a *1-child,* $\frac{1}{2!}f''(x=0)$ is a *2-child*, etc. Notice the implicative terms associated with each arc: $x^0=1$ for the 0-edge, $x^1=x$ for the 1-edge, $x^2$ for the 2-edge, etc.

TEDs are a new canonical, graph-based representation for arithmetic expressions, which can be exploited to facilitate equivalence checking of high-level specifications of digital designs in terms of the compactness and the canonicity properties. TEDs handle algebraic variables as real numbers. Figure 2.10 shows an example of TED representation for a simple algebraic expression.

Note the additive and multiplicative weights assigned to the edges. The computation of the derivatives, and hence the children of *f*, performed recursively, is trivial for polynomial functions.

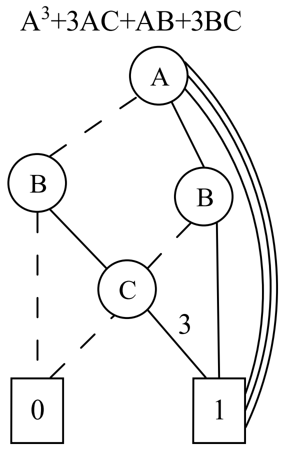


**Figure 2.10: An example of an expression represented with TED**

## 2.2.6 Disadvantages of Decision Diagrams

The canonicity and ease of composition that OBDDs and MTBDD provide make them ideal for matching small combinational circuits. In order to handle complex circuits such as multiplication, the potentially exponential size of BDD structures makes comparison of BDDs time consuming and memory intensive. BMDs and TEDs manipulate the complex circuits by easing the requirement of memory and time. They have been used to verify the functionality of linear circuits [141]. However, they can only yield information on whether or not an implementation matches a specification exactly, but offer no path for quantifying the degree to which the two offer. Therefore, if two functions are similar but not exactly equal, BMDs and TEDs structures may implement drastically different arithmetic functions, while two very different diagrams may implement the same mathematical operation with different degrees of precision. Also, BMDs and TEDs are unsuitable for use in non-linear functions because of the resulting exponential complexity in the worst case [77], and hence decision diagrams are not suitable to be used to explore imprecise circuits.

## 2.3 Dynamic Analysis

Decision diagrams are explored in formal verification as a part of equivalence and model checking, but they have no ability to process the fixed-point representation. The usual method to handle fixed-point designs is through the dynamic analysis which uses appointed vectors as specific inputs. The major elements include the tested circuit and a group of vectors. A testbench represents stimuli to the circuit under verification. The results of the circuit simulations with the stimuli indicate whether the implementation is suitable for the specification. The simple idea makes it prevalently used. In fact, historically, dynamic analysis is the oldest technique to verify digital designs. The major draw back of this class of methods is the requirement to enumerate all possible input values in order to verify a circuit in 100%.

The exhaustive test vectors are usually infeasible for dynamic analysis because of huge execution time. A practical testing method requires as few vectors as possible to cover as many faults as possible, so the technique of test generation has been developed. ATPG (Automatic Test Pattern Generation) is a technology to distinguish between the correct circuit behavior and the faulty circuit behavior caused by defects. Obviously, the processed objects are precise designs and it is difficult to handle or optimize imprecise designs by these methods. Varieties of explorations adopt dynamic analysis and avoid exhaustive vectors to optimize imprecise designs, which are introduced next.

Authors in [18] - [25] rely on the straightforward technique to get optimization of a bit-width. In [19] *Kung et al.* develop a combined *word-level* (WL) optimization and high-level synthesis algorithm to minimize the hardware implementation cost and significantly reduce the optimization time. Their algorithm initially finds the WL sensitivity or minimum WL of each signal throughout fixed-point simulations of a signal flow graph. Then it performs the WL high-level synthesis where signals having the similar WL sensitivity are assigned to the same functional unit. Finally, the algorithm conducts the final WL optimization by iteratively modifying the WLs of the synthesized hardware model. Figure 2.11 [19] depicts the design flow of optimization.

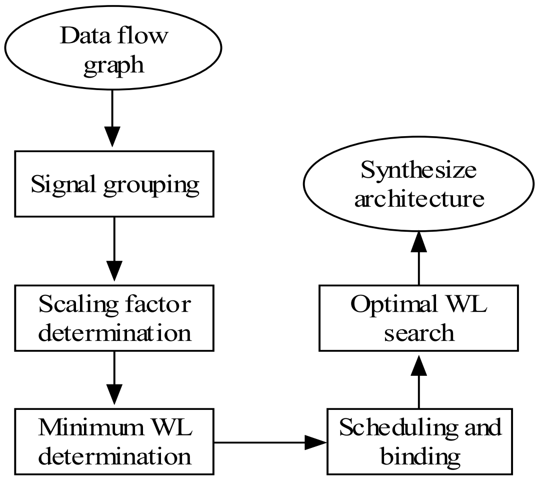


**Figure 2.11: Design flow of the architecture-level WL optimization [19]**

*Willems and Bursgens* [20] present a tool that allows an automated, interactive transformation from floating-point ANSI-C into a bit-true specification. The tool quantizes the input value and analyzes quantization effects on an algorithmic level. Then it invokes the simulation-based fixed-point algorithm to target the described specification. The main disadvantage of the above method is that it requires a large set of input vectors, and hence a long simulation time is unavoidable.

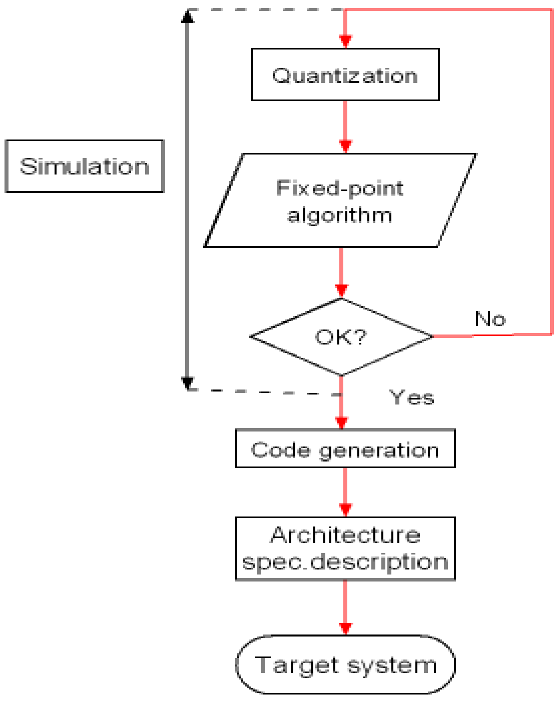


**Figure 2.12: The tool flow of the method in [20]**

*Gaffar et al.* [21] offer a uniform treatment for bit-width optimization of fixed-point designs. They utilize automatic differentiation to compute the sensitivities of outputs to the bit-width of the various operands in the design. This sensitivity analysis enables to explore and compare fixed-point and floating-point implementation for a particular design. As a result they can automate the selection of the optimal number representation for each variable in a design to optimize area and performance. Figure 2.13 describes its design flow.

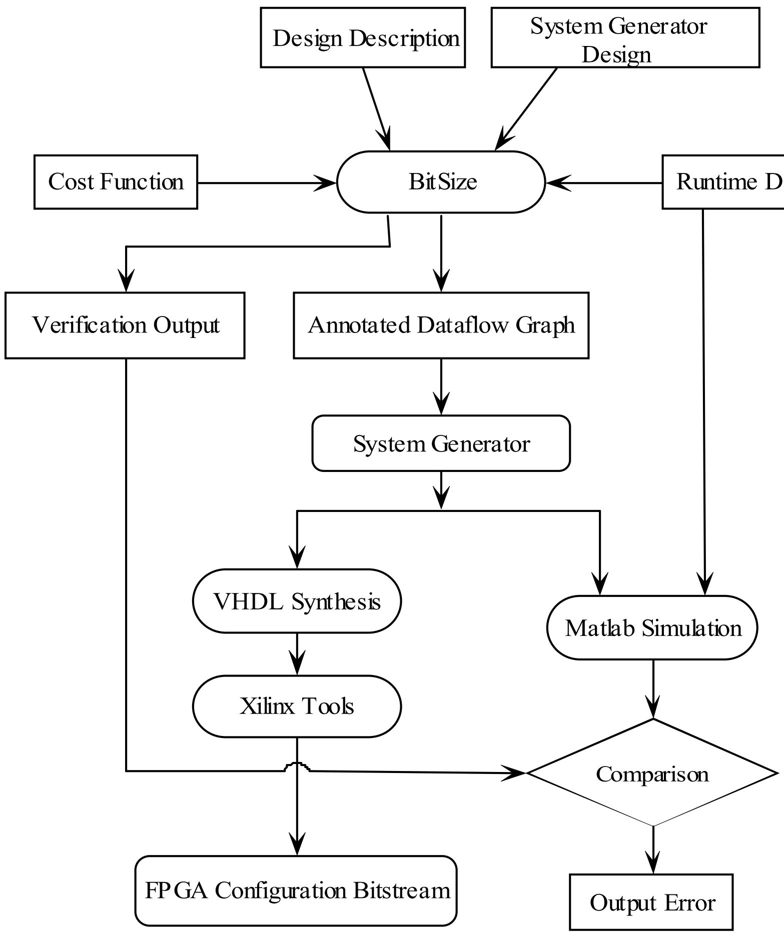


**Figure 2.13: The design flow of dynamic analysis in [21]**

*C. Shi et al.* [22] set up a statistical model to estimate hardware resource in terms of perturbation theory. A tool that automates the floating-point to fixed-point conversion (FCC) process for digital signal system is described based on a simulation tool, Simulink. The tool automatically optimizes fixed-point data types of arithmetic operators, including overflow modes, integer word lengths, fractional word lengths, and the number systems. The approach is based on statistical modeling, hardware resource estimation and global optimization based on an initial structural system description.

*Nayak et al.* [23] propose a precision analysis algorithm to determine the minimum number of bits required by an integer variable, and present a framework to generate an efficient hardware for signal processing applications. Their range optimization relies on data range propagation, while precisions are analyzed and optimized by the DFG which is an acyclic graph representation of a circuit. A memory packing algorithm is proposed to generate faster hardware requiring less execution time. Figure 2.14 illustrates the framework.

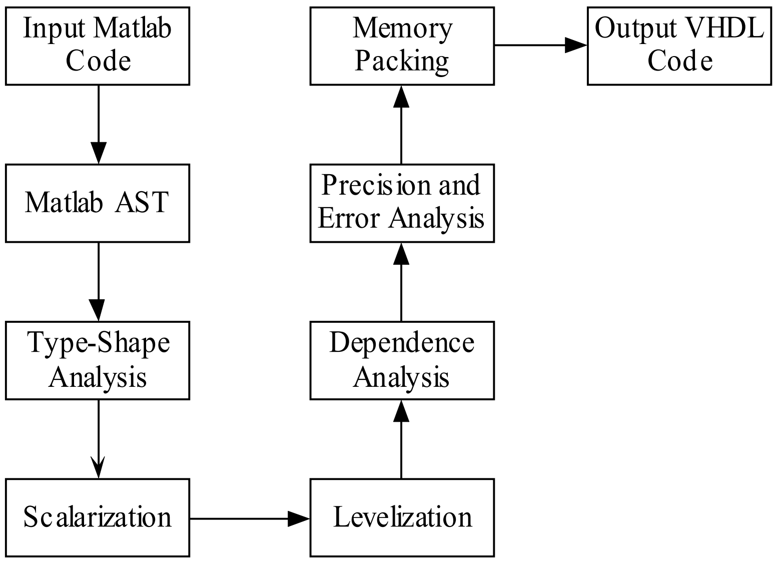


**Figure 2.14: Overview of the synthesis framework in [23]**

Though dynamic analysis provides bit-widths closer to the optimal set for those particular stimuli, it is not a perfect solution since a large set of stimuli signals is required to analyze a design with sufficient confidence. This possibly leads to prohibitively long simulation time without guarantees for alternative input stimuli encountered in practice. Hence, often not only low efficiency of the overall process can be encountered, but the above methods can become infeasible for some cases. Therefore, other methods should be explored.

## 2.4 Static Analysis

Static analysis such as interval arithmetic and affine arithmetic can avoid tedious simulation. This section introduces static methods to handle fixed-point circuits represented by polynomials.

### 2.4.1 Interval Arithmetic

In mathematics, a (real) interval is defined as a set of real numbers with the property that any number that lies between two numbers in the set is also included in the set. For example, the set of all numbers *x* from the interval [0,1] include 0 and 1, as well as all real numbers between them. Interval arithmetic (IA) is a method developed by mathematicians in 1950s and 1960s as an approach to putting bounds on rounding errors in mathematical computation. Among many contributors, we distinguish *Hansen,* who in [26] introduced basic ides of interval arithmetic and *Kearfott,* who in [27] presented some important applications of interval computations. In general, the advances in interval arithmetic led to the development of numerical methods that yield very reliable results.

Where classical arithmetic defines operations on individual numbers, interval arithmetic defines a set of operations on intervals. An operation <OP> on two intervals is defined as:

$$[x_1, x_2] < OP > [y_1, y_2] = \{x < OP > y \mid x \in [x_1, x_2], y \in [y_1, y_2]\}$$

The operand <OP> can, for example, represent addition or multiplication. For practical applications the above notation can be simplified to:

Addition: $[x_1, x_2] + [y_1, y_2] = [x_1 + y_1, x_2 + y_2]$

Subtraction: $[x_1, x_2] - [y_1, y_2] = [x_1 - y_1, x_2 - y_2]$

Multiplication:

$$[x_1, x_2] * [y_1, y_2] = [\min(x_1y_1, x_1y_2, x_2y_1, x_2y_2), \max(x_1y_1, x_1y_2, x_2y_1, x_2y_2)]$$

Division:

$$[x_1, x_2] / [y_1, y_2] = [x_1, x_2] * (1/[y_1, y_2]) ,$$

where $1/[y_1, y_2] = [1/y_1, 1/y_2]$ if $0 \notin [y_1, y_2]$

With the help of these definitions, it is already possible to calculate the range of simple functions, such as $f(a,b,x) = ax+b$. If, for example $a$ = [1,2], $b$ = [5,7] and $x$ = [2,3], it is clear that

$$f(a,b,x) = ([1,2]*[2,3]) + [5,7] = [1*2, 2*3] + [5,7] = [7,13]$$

Interval methods can also apply to functions which do not just use simple arithmetic, and we must also use other basic functions for redefining intervals as known *monotonicity properties*. The range of values is easy to determine for *monotonic functions* in one variable. If $f : R \rightarrow R$ is monotonically rising or falling in the interval $y_1$, $y_2 \in [x_1, x_2]$, then one of the following inequalities applies for all values in the interval such that $y_1 \leq y_2$ :

$$f(y_1) \leq f(y_2) \quad \text{or} \quad f(y_1) \geq f(y_2)$$

The range corresponding to the interval $[y_1, y_2] \subseteq [x_1, x_2]$ can be calculated by applying the function to the endpoints $y_1$ and $y_2$:

$$f([y_1, y_2]) = [min\{f(y_1), f(y_2)\}, max\{f(y_1), f(y_2)\}]$$

Using the above equation, the following basic features for interval functions can easily be defined:

- **Exponential function:** $a^{[x_1,x_2]} = [a^{x_1}, a^{x_2}]$ $\quad a \geq 1$,
- **Logarithm:** $Log_a^{[x_1,x_2]} = [Log_a^{x_1}, Log_a^{x_2}]$ for positive intervals $[x_1, x_2]$ and $a > 1$,
- **Odd powers:** $[x_1, x_2]^n = [x_1^n, x_2^n]$ for odd $n \subseteq N$.

The methods of classical numerical analysis cannot be transferred one-to-one into interval-valued algorithms, as dependencies between numerical values are usually not taken into account.

In order to work effectively in a real-life implementation, intervals must be compatible with floating point computing. The earlier operations were based on exact arithmetic, but in general fast numerical solution methods may not be available. The range of values of the function $f(x,y) = x + y$ for $x \in [0.1, 0.8]$ and $y \in [0.06, 0.08]$ are for example [0.16, 0.88]. Where the same calculation is done with single digit precision, the result would normally be [0.2, 0.9]. But $[0.16, 0.88] \notin [0.2, 0.9]$, so this approach would contradict the basic principles of interval arithmetic, as a part of the domain of $f([0.1, 0.8], [0.06, 0.08])$ would be lost. Instead, it is the outward rounded solution [0.1, 0.9] which is used.

The required *external rounding* for interval arithmetic can thus be achieved by changing the rounding settings of the processor in the calculation of the upper limit and lower limit. Alternatively, an appropriate small interval [$\varepsilon_1, \varepsilon_2$] can be added.

Interval arithmetic is used in association with error analysis to control rounding errors arising from each calculation. The advantage of interval arithmetic is that after each operation there is an interval which reliably includes the true result. The distance between the interval boundaries gives the current calculation of rounding errors directly:

$$\mathit{Error} = \mathrm{abs}(a - b) \text{ for a given interval } [a,b].$$

### 2.4.2 Affine Arithmetic

*Affine arithmetic* (AA) is a model for numerical analysis introduced first by Stolfi and Figueiredo, [32] [33]. In AA, the quantities of interest are represented as *affine combinations* (affine forms) of certain primitive variables, which stand for sources of uncertainty in the data or approximations made during the computation. It is meant to be an improvement on interval analysis (IA).

In affine arithmetic, each input or computed quantity $\hat{x}$ is represented by a formula:

$$\hat{x} = x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \ldots + x_n\varepsilon_n$$

where $x_0$, $x_1$, ... $x_n$ are floating-point numbers and $\varepsilon_1, \varepsilon_2 \ldots \varepsilon_n$ are symbolic variables whose values are only known to lie in the range [-1,+1]. We call $x_0$ the central value of the affine form $\hat{x}$; the coefficients $x_i$ are its partial deviations, and the $\varepsilon_i$ are the noise symbols. Thus, for example, a quantity $\hat{x}$ which is known to lie in the range [3,7] can be represented by the affine form $\hat{x} = 5 + 2\varepsilon_k$.

The key feature of AA is that the same symbol $\varepsilon_i$ may contribute to the uncertainty of two or more quantities (inputs, outputs, or intermediate results) $\hat{x}$ and $\hat{y}$ arising in the evaluation of an expression. The noise symbols can be shared which indicates some partial dependency between the underlying quantities $x$ and $y$, determined by the corresponding coefficients $x_i$ and $y_i$. Note that the signs of these coefficients are not meaningful in themselves, because the sign of $\varepsilon_i$ is arbitrary; but the relative sign of $x_i$ and $y_i$ defines the direction of the correlation. For example, suppose that the quantities $x$ and $y$ are represented by the affine forms:

$$\hat{x} = 17 - 3\varepsilon_1 + 2\varepsilon_3 + 4\varepsilon_4 \qquad \hat{y} = 9 - \varepsilon_1 + \varepsilon_2 - 2\varepsilon_4$$

From this data, $x$ lies in the interval $\hat{x}$= [8, 26] and $y$ lies in $\hat{y}$ = [5, 13], i.e., the pair ($x$, $y$) lies in the grey rectangle of Figure 2.16; however, since the two affine forms include the same noise variables $\varepsilon_1$ and $\varepsilon_4$ with non-zero coefficients, they are not entirely independent of each other. In fact, the pair ($x$, $y$) lies in the dark grey region of Figure 2. 15, which is the set of all possible values of ($\hat{x}, \hat{y}$) when the noise variables $\varepsilon_1$, .. $\varepsilon_4$ are independently. This set is the joint range of the forms $\hat{x}$ and $\hat{y}$, denoted <$\hat{x}$, $\hat{y}$>.

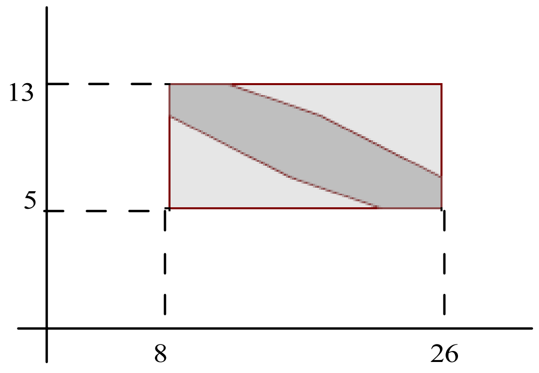


**Figure 2. 15: Joint range ($\hat{x}, \hat{y}$) of two partially dependent quantities as implied by their affine forms**

In order to evaluate a formula with AA, we need to replace each elementary operation $z \leftarrow f(x, y)$ on real quantities $x$ and $y$ by a corresponding procedure $\hat{z} \leftarrow f(\hat{x}, \hat{y})$, which uses affine forms of those quantities and returns an affine form for the result $z$. By definition, there are:

$$\hat{x} = x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \ldots + x_n\varepsilon_n$$

$$\hat{y} = y_0 + y_1\varepsilon_1 + y_2\varepsilon_2 + \ldots + y_n\varepsilon_n$$

Therefore, the result $\hat{z}$ is a function of the unknown variables $\varepsilon_i$ as:

$$\hat{z} = f(\hat{x}, \hat{y}) = f(x_0 + x_1\varepsilon_1 + \ldots x_n\varepsilon_n, y_0 + y_1\varepsilon_1 + \ldots y_n\varepsilon_n)$$

***Example 2.10:*** *Consider the multiplication of two affine forms* $\hat{z} \leftarrow \hat{x}\hat{y}$, *where*

$\hat{x} = 20 - 4\varepsilon_1 + 3\varepsilon_2$ *and* $\hat{y} = 30 + 2\varepsilon_1 + \varepsilon_3$. *Please notice that the operands are partially correlated through the shared noise symbol* $\varepsilon_1$. *The product of* $\hat{x}\hat{y}$ *is:*

$$\hat{z} = \hat{x}\hat{y} = 600 - 80\varepsilon_1 + 90\varepsilon_2 + 20\varepsilon_3 - 8\varepsilon_1^2 - 4\varepsilon_1\varepsilon_3 + 6\varepsilon_1\varepsilon_2 + 3\varepsilon_2\varepsilon_3$$
$$= 600 - 80\varepsilon_1 + 90\varepsilon_2 + 20\varepsilon_3 - 8\varepsilon_4 - 4\varepsilon_5 + 6\varepsilon_6 + 3\varepsilon_7$$

*Using the form of* $\hat{z}$, *we can estimate the range of* $\hat{z}$ *is [389, 811]. The actual range of* $\hat{x}\hat{y}$ *is [403, 756], so the obtained range by AA is (811-389) / (756 – 403) = 1.2 times wider than the exact range. If using IA for comparison, z = [13, 27] * [27, 33] = [351, 891], that is (891 – 351) / (756 – 403) = 1.53 times wider than the exact range. The reason is AA can partly process the correlation between* $\hat{x}$ *and* $\hat{y}$ *implied by the shared symbol* $\varepsilon_1$. *The correlated terms* $-120\,\varepsilon_1$ *and* $+40\,\varepsilon_1$ *nearly cancel out in the AA computation, but are added with the same sign in the IA computation.*

*C.Fang et al.* [39] [40] take advantage of affine arithmetic modeling to analyze range and precision from fixed-point implementations of DSP algorithms. The resulting numerical error estimates are comparable to detailed statistical simulation, but achieve speedups of four to five orders of magnitude by avoiding actual bittrue simulation. Authors in [41] [43] propose an approach that optimizes the bit-widths of fixed-point and floating-point designs. Range analysis depends on a combined affine and interval arithmetic approach to reduce the number of bits. Precision analysis involves a coarse-grain and fine-grain analysis. The best representation in fixed-point or floating-point is then chosen based on the range, precision and latency. Figure 2.16 illustrates the methodology.

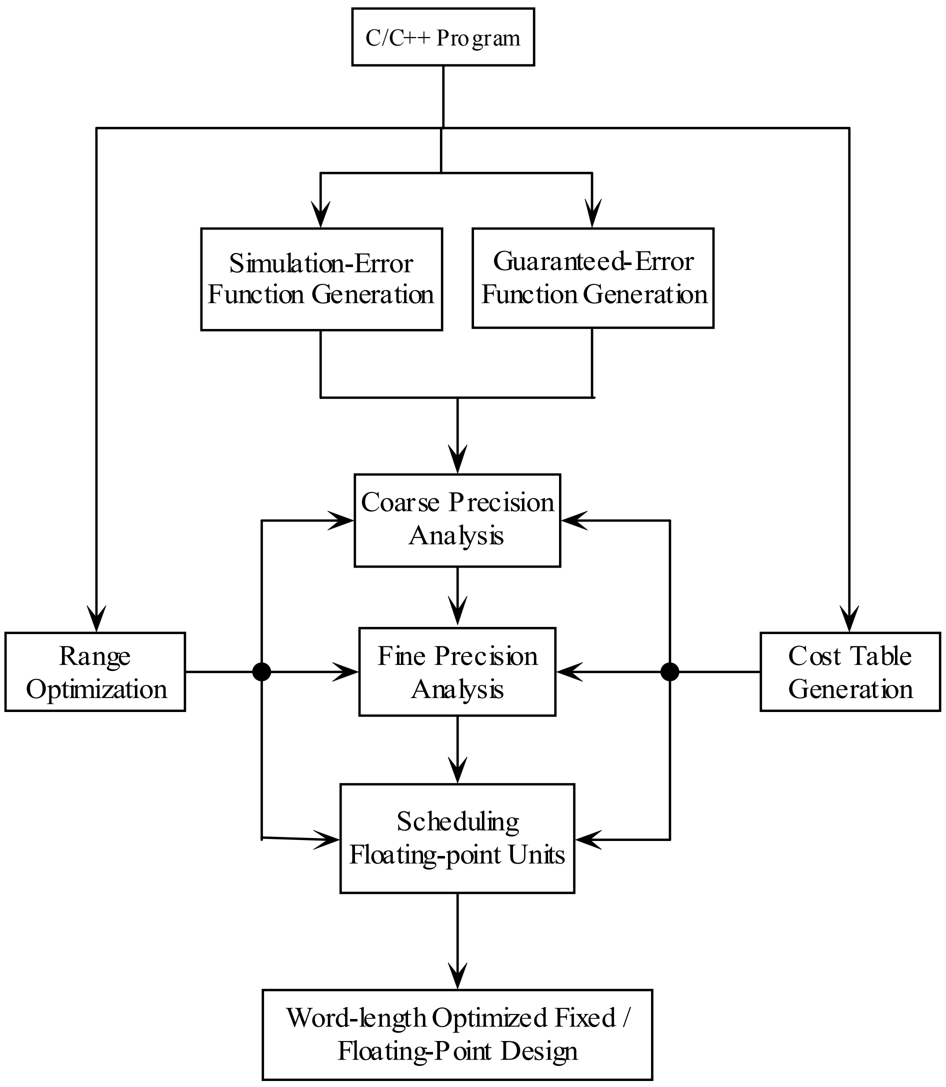

C/C++ Program
Simulation-Error Function Generation
Guaranteed-Error Function Generation
Coarse Precision Analysis
Range Optimization
Fine Precision Analysis
Cost Table Generation
Scheduling Floating-point Units
Word-length Optimized Fixed / Floating-Point Design

**Figure 2.16: An outline of the methodology in [41]**

The algorithm starts from generating cost and error functions and then analyzes range. The next stage is precision analysis. A coarse-grain analysis produces uniform bit-widths. These results are then refined to produce non-uniform bit-widths. The last stage is floating-point scheduling before the source code is reconstructed to a given C/C++ design.

Authors in [42] use AA to investigate bit-width due to truncated and rounded data, and explore hardware area and delay in FPGA on the condition of different bit-width. Figure 2.17 introduces the tool of static analysis.

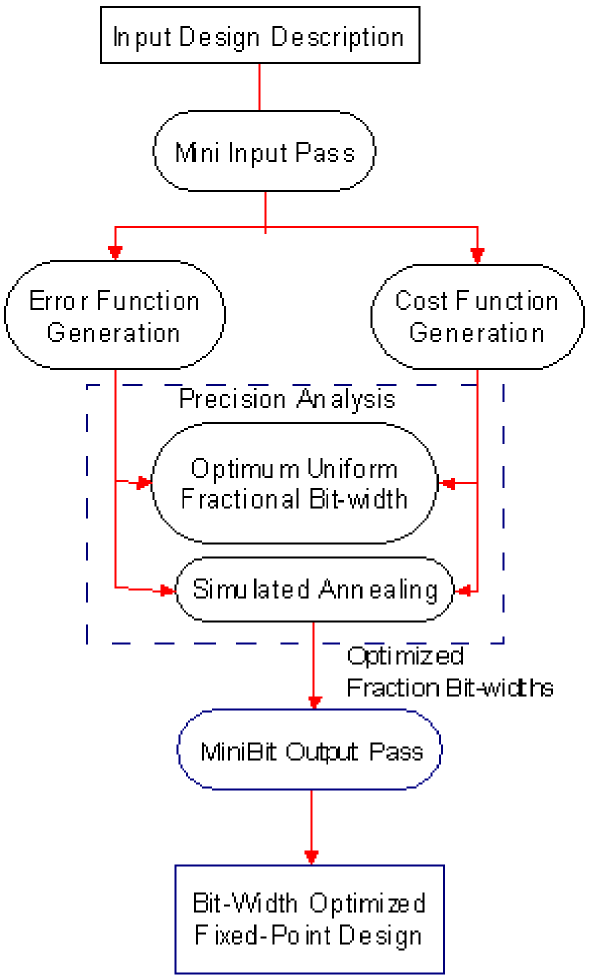

Input Design Description
Mini Input Pass
Error Function Generation
Cost Function Generation
Precision Analysis
Optimum Uniform Fractional Bit-width
Simulated Annealing
Optimized Fraction Bit-widths
MiniBit Output Pass
Bit-Width Optimized Fixed-Point Design

**Figure 2.17: The tool of static analysis in [42]**

The algorithm generates error function and cost function respectively to optimize the fractional bit-width. An optimized fixed-point design is obtained by the precision analysis. *Osborne et al.* [45] extend the work in [42] to propose a tool, LengthFinder, for optimizing wordlengths of hardware designs with fixed-point arithmetic based on analytical error models that guarantee accuracy. The tool adopts a multi-stage approach, with four novel features. First, the code is analyzed and loops are selected to instrument, so information about the number of iterations can be extracted to generate more accurate results. Second, aggressive heuristics are used to produce non-uniform wordlengths rapidly while meeting requirements from the guaranteed error functions. Third, a method which is capable of reducing the search space is developed for data-partitioning with a variable wordlength reduction. Fourth, a genetic algorithm with selective-crossover and high mutation probability is applied to obtain near-optimal results.

In [93], authors set up models for error source dependence. In these models, the dependence is approximated by linear functions (AA) or by general polynomials (Taylor series methods), which are proved optimal. They also describe that the optimal way to decrease the excessive bit-width is to use implicit polynomial dependence.

Affine arithmetic is potentially useful in every numeric problem where one needs guaranteed enclosures to smooth functions, such as solving systems of non-linear equations, analyzing dynamical systems, integrating functions differential equations, etc. Additionally, AA has many applications in areas such as computer graphics, optimization and curve drawing in [35], [36], [37], [38]. Here it is used to handle range analysis and bit-width optimization.

## 2.5 Alternate Methods

*Constantinides et al.* [46] present an approach to the wordlength allocation and optimization for linear DSP systems. The tool `Synoptix` [47] - an optimization technique targeting linear time-invariant digital signal processing systems using a novel resource binding technique is proposed. It is based on saturation arithmetic to perform the range of bit-width optimizations and allows the user to tradeoff implementation area for arithmetic error at system outputs.

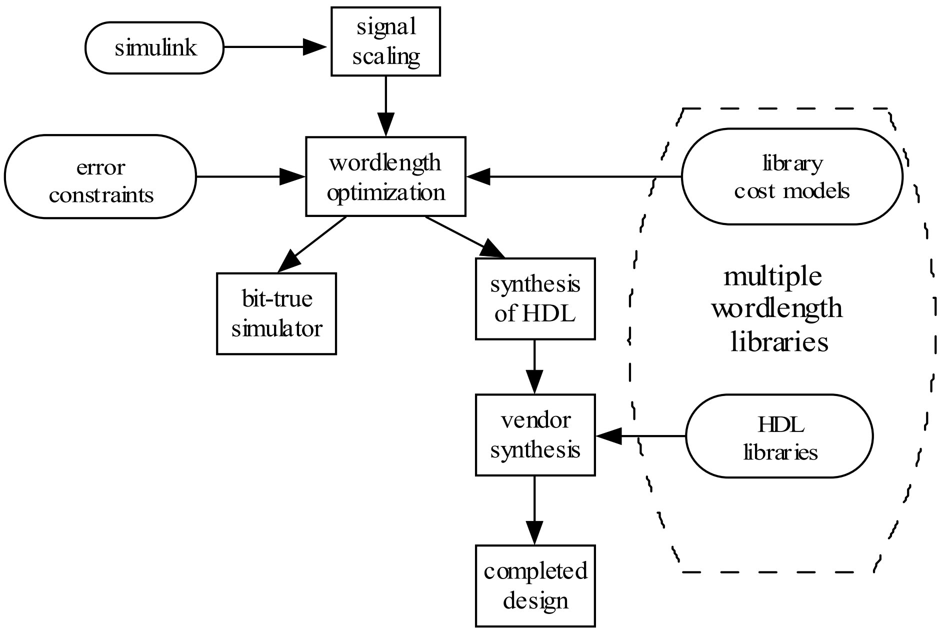


**Figure 2.18: Synoptix design flow in [47]**

Figure 2.18 describes the tool flow. The input to `Synoptix` is a `Simulink` block diagram, and the output is a structural description in VHDL. Third-party tools are then used to perform the low-level logic synthesis, placement, and routing of the designs.

*Kinsman and Nicolici* [55] introduce the theory of SAT-Modulo (SMT) to explore ranges. SMT first uses the coarse bounds obtained by IA, and then refines them by inserting constraints. More precise bounds than AA can be obtained, so determine smaller bit-widths for an implementation. Based on the scheme, an SMT engine can be used to prove/disprove validity of a bound on a given expression by checking for satisfiability.

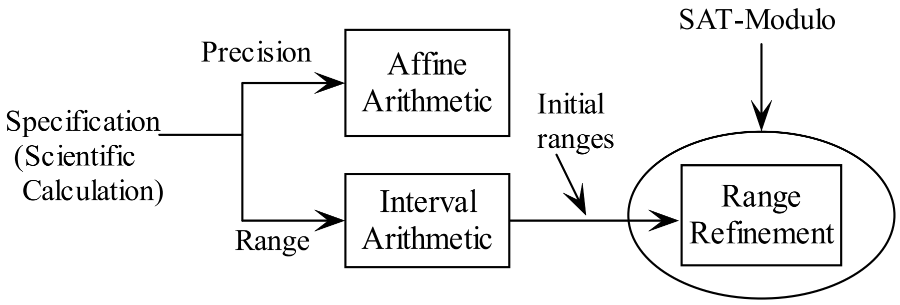


**Figure 2.19: Flow of SMT technique in [55]**

*Ahmadi and Zwolinski* [54] address the bit-width assignment in hardware implementation in the context of high-level synthesis. They introduce a *symbolic noise analysis* (SNA) to surpass the pessimism of IA, which is based on modeling of the error bounds by an assumed probability distribution function over a known range. In comparison to SNA which assumes the error distributions more localized, IA is pessimistic by assuming the uniform distribution. The proposed method is used in combination with models of power consumption, circuit area and delay. Results demonstrate a considerable saving in costs when these optimizations are applied.

## 2.6 Conclusions

In this chapter, we introduced the usual Boolean function representations such as decision diagrams. Although decision diagrams such as TEDs are suitable to equivalence checking and model checking, they cannot be applied to imprecise circuits or to bit-width optimization. Dynamic analysis is a common method and many explorations are based on it, but its low efficiency confines its applications. Static analysis has been developed to overcome this limitation. IA is the usual method of finding ranges and AA is a derivation which can calculate more precise ranges than IA.

These explorations only get one optimization of bit-width such as [42] or hardware area such as [51]. Another disadvantage is that they do not consider the function approximation so they are not capable of investigating these factors concurrently. In our research, we overcame this disadvantage and simultaneously processed bit-widths and various constraints as well as approximations for Taylor series and real-valued polynomials.

# Chapter 3

# Compositions of AT and Extensions

*Arithmetic Transform (AT) must be extended to represent combinational circuits and sequential circuits efficiently. We state past methods of calculating AT coefficients, and then address the use of AT and its extensions to express word-level quantities and sequential elements. Since a circuit transform can express properties of the circuit distinctly and help engineers to penetrate its essence straightforwardly, obtaining an overall transform by symbolic compositions of individual blocks' transforms becomes most significant. For the purpose of running time and memory, the best algorithm is proposed for a compositional verification of the complex datapath.*

## 3.1 Introduction of Spectral Transforms

As a main method exploring the fixed-point circuits in our research, Arithmetic Transform (AT) is a spectral representation different with Boolean representations. So we introduce the spectral domain and the basic AT definition at first in this Chapter.

### 3.1.1 Spectral Domain

It is common to use the product and sum operators of the Boolean algebra together with negation to define such functions — for example, $f(x_1, x_2, x_3) = x_1\overline{x_2}x_3 + \overline{x_1}x_2\overline{x_3}$. The use of Boolean algebra for the manipulation and analysis of switching circuits is well known. Part of the problem with the definition in the Boolean domain is that each of the entries in the truth table for *f* tells us precisely the behavior of the function at a single point but nothing of its behavior for any other points. It is possible to give an alternate representation of a function where the information about the function is much more global in nature. This alternate representation is in the spectral domain, and a number of properties are much more easily deduced in the spectral domain than in the Boolean one [56]. Spectral techniques are very powerful tools for logic functions to express the principle of linearity and superposition.

The basic idea of the spectral domain and how to get there is illustrated in Figure 3.1. In order to avoid losing information, the transform should be reversed, that is, we can move to and from the spectral domain without any loss of information.

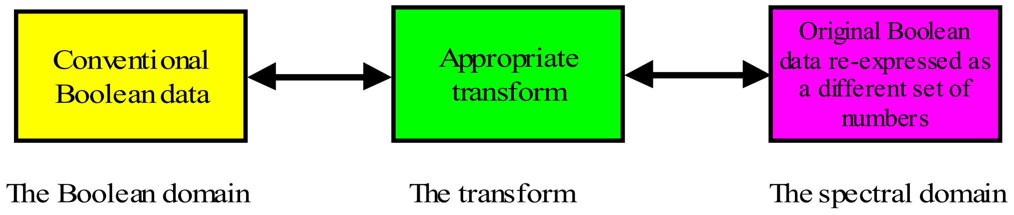

Conventional Boolean data
Appropriate transform
Original Boolean data re-expressed as a different set of numbers
The Boolean domain
The transform
The spectral domain

**Figure 3.1: The spectral transform**

The information content in the functional and spectral domains will be identical, and the data in either domain is uniquely recreatable from the data in the other, but the meaning of the parameters in the two domains will be dissimilar. In particular, the discrete nature of the data in the function domain will be generally influenced by the complete functional performance of the circuit or network under consideration. The following section outlines several usual spectral transforms.

## 3.1.2 Various Transforms

### *A) Reed-Muller Transform*

**Definition 3.1:** *In matrix notation, positive polarity Reed-Muller (PPRM) expressions for functions in GF(2) are given by:*

$$RM(f) = R_n F$$

*where F is the truth table for the Boolean function f and*

$$R_n = \begin{bmatrix} R_{n-1} & 0 \\ R_{n-1} & R_{n-1} \end{bmatrix}, \qquad R_0 = 1 \tag{3-1}$$

***Example 3.1:*** *Consider a function $f(x_0, x_1, x_2) = x_1x_2+x_0$, i.e., $F = [0, 1, 0, 1, 0, 1, 1, 1]^T$ . Using the Eqn. 3-1, coefficients of Reed-Muller transform are calculated as:*

$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \\ r_7 \end{bmatrix} = \begin{bmatrix} 1&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 1&0&1&0&0&0&0&0 \\ 1&1&1&1&0&0&0&0 \\ 1&0&0&0&1&0&0&0 \\ 1&1&0&0&1&1&0&0 \\ 1&0&1&0&1&0&1&0 \\ 1&1&1&1&1&1&1&1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

*Thus $RM(f) = x_0 \oplus x_1x_2 \oplus x_0x_1x_2$*

### *B) Fixed-Polarity Reed-Muller Transform*

The fixed polarity Reed-Muller (FPRM) transform is derived from the negative Davio expansion together with the positive Davio expansion (no need for the same variable). These transforms are characterized by the polarity vectors $H = (h_1, \dots, h_n) \in \{0, 1\}^n$, whose $i^{th}$ coordinate $h_i = 1$ shows that the corresponding variable is represented by the negative literal $\overline{x_i}$ in the polynomial representation for a given function *f* [57].

For a given polarity vector *H*, the FPRM polynomial is given in the matrix notation by:

$$FPRM(f) = (\prod_{i=1}^{n}[1 \;\; x_i^{h_i}])(\prod_{i=1}^{n}[R^{h_i}(1)])F$$

where

$$x_i^{h_i} = \begin{cases} x_i, h_i = 0 \\ \overline{x_i}, h_i = 1 \end{cases} \qquad R^{h_i}(1) = \begin{cases} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, h_i = 0 \\ \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}, h_i = 1 \end{cases}$$

***Example 3.2:*** *Figure 3.2 [57] shows the Reed-Muller transform matrix for n = 3 and the polarity vector H = (0, 1, 0).*

$$R^{(0,1,0)}(3) = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

**Figure 3.2: Reed-Muller matrix for n = 3 and the polarity vector H = (010)**

*The indices of columns in R(010)(3) are defined as ($i_1 \oplus h_1$, $i_2 \oplus h_2$, $i_3 \oplus h_3$) compared to the positive polarity (H = (0, 0, 0)) Reed-Muller matrix R(3). So the original output order (0, 1, 2, 3, 4, 5, 6, 7) changes to (2, 3, 0, 1, 6, 7, 4, 5). With this matrix, for a function f given by the truth-vector F =[1, 0, 0, 1, 0, 1, 1, 1]$^T$, the Reed-Muller expansion for H = (0, 1, 0) is given by*

$$FPRM(f) = x_0 \oplus \overline{x_1} \oplus x_2 \oplus x_2x_0 \oplus x_2\overline{x_1}\,x_0$$

## *C) Walsh Transform*

The Walsh functions [57] [59] [60] [61] are a closed set of two-valued orthogonal functions, given by

$$Wal(j,k) = \prod_{\eta=0}^{n-1} \{(-1)^{\{k_{n-\eta}+k_{n-1-\eta}\} j_n}\}$$

Where $j_\eta$ , $k_\eta$ are determined by the binary expansions of *j, k* respectively, *j, k* $\in$ 0 to $2^n$-1, where

$j = \{j^{n-1}2^{n-1} + j^{n-2}2^{n-2} + \ldots + j^0 2^0\}$ $\qquad$ $k = \{k^{n-1}2^{n-1} + k^{n-2}2^{n-2} + \ldots + k^0 2^0\}$

The Walsh transform is a complete orthogonal square matrix, with row and column entries $\in$ {+1, -1} and with a recursive structure as follows:

$$W_n = \begin{bmatrix} W_{n-1} & W_{n-1} \\ W_{n-1} & -W_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes W_{n-1}$$

where $\otimes$ denotes the Kronecker product operator. The transform is given by $W(f) = W_n F$.

## *D) Fixed-Polarity Walsh Transform*

For a given polarity vector $H = (h_1, \ldots, h_n)$ the fixed polarity Walsh polynomial is given in the matrix notation by [57]:

$$FPW(f) = 2^{-n}(\prod_{i=1}^{n}[1 \;\; 1-2x_i^{h_i}])(\prod_{i=1}^{n}[(-1)^{h_i} \;\; (-1)^{\overline{h_i}}])F$$

## *E) Kronecker Transform*

**Definition 3.2:** *For a function f, the Kronecker spectrum is defined as:*

$$K(f) = K_n F$$

*where* $K_n = \begin{bmatrix} 0 & K_{n-1} \\ K_{n-1} & K_{n-1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \otimes K_{n-1}$

Figure 3.3 shows the Kronecker transform matrix K(3):

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

**Figure 3.3: A Kronecker transform matrix for n = 3**

## *F) Haar Transform*

The orthogonal Haar functions [56] may be defined as follows, where *k* is taken over the continuous interval 0 to 1:

$$H_0^0(k) = +1.0$$

$$H_i^q(k) = |\sqrt{2}\,|^{i-1}\ (+1.0) \quad \text{for } \frac{q}{2^{i-1}} \le k < \frac{q+\frac{1}{2}}{2^{i-1}}$$

$$= |\sqrt{2}\,|^{i-1}\ (-1.0) \quad \text{for } \frac{q+\frac{1}{2}}{2^{i-1}} \le k < \frac{q+1}{2^{i-1}}$$

where $i = 1, 2, \ldots, n$ and $q = 0, 1, \ldots, 2^{i-1}-1$.

The sequentially ordered discrete Haar functions for $n = 3$ are shown in Figure 3.4.

$$H_s(3) = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} \\ 2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & -2 \end{bmatrix}$$

**Figure 3.4: Sequentially ordered Haar functions for n = 3**

## 3.2 Arithmetic Transform

### 3.2.1 Basic Definition

We adopt *Arithmetic Transform* that is defined in the spectral domain as our main method to analyze imprecise factors and compute imprecision. Traditional methods are hard to determine the maximum error on the condition of the Taylor word-level input, but AT can decompose word-level variables into bit-level quantity to avoid the disadvantage and represent the error function essentially. AT has been proved to be suitable for precision verification and optimization by precision constraints, so here we use it to analyze imprecision of Taylor series.

AT is a canonical polynomial representing uniquely multi-input and multi-output Boolean functions $f : B^n \to B^m$. Multi-output can be grouped to form a word-level (integer) number $w$ to obtain an AT description in a form of a single polynomial, leading to a pseudo Boolean function $f : B^n \to w$. Therefore, the AT representation has Boolean inputs and a word-level output.

**Definition 3.3:** *The* ***Arithmetic Transform (AT)*** *[62] is a polynomial representing a pseudo Boolean function $f : B^n \to w$ using an arithmetic operation "+", word-level coefficients $c_{i_1 i_2 \ldots i_n}$, binary inputs $x_1, x_2, x_n$ and binary exponents $i_1, i_2 .. i_n$:*

$$AT(f) = \sum_{i_1=0}^{1} \sum_{i_2=0}^{1} \ldots \sum_{i_n=0}^{1} c_{i_1 i_2 \ldots i_n} x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n}$$

The matrix multiplication is most frequently used to determine AT of a given function. In this method, the set of AT coefficients $C = \{c_{i_1 i_2 \ldots i_n}\}$ are obtained by multiplying the $2^n \times 2^n$ matrix $T_n$ by a $2^n \times 1$ vector of function values (truth table of $f$): $C = T_n \times f$ where the transform matrix $T_n$ is defined recursively:

$$. T_n = \begin{bmatrix} T_{n-1} & 0 \\ -T_{n-1} & T_{n-1} \end{bmatrix} \qquad T_0 = 1 \qquad (3\text{-}2)$$

AT generates a word-level output and it is encoded by binary weights addition. A word-level encoding is explicitly expressed by the number norm function $|\ |: B^m_{\cdot} \rightarrow W_{\cdot}$, defining a Boolean vector interpretation in the word-level domain. Table 3.1 [70] gives a summary of common integer and fractional number norms for a vector of Boolean values $x_i$.

| Word | Number Norm $\lvert x \rvert$ | | |
|---|---|---|---|
| | Unsigned | Sign Extended | 2's Complement |
| Integer | $\sum_{i=0}^{n-1} x_i 2^i$ | $(1-2x_{n-1})\sum_{i=0}^{n-2} x_i 2^i$ | $\sum_{i=0}^{n-2} x_i 2^i - x_{n-1} 2^{n-1}$ |
| Fractional | $\sum_{i=0}^{n-1} x_i 2^{-i}$ | $(1-2x_0)\sum_{i=1}^{n-1} x_i 2^{-i}$ | $-x_0 + \sum_{i=1}^{n-1} x_i 2^{-i}$ |
| Fixed Point | $\sum_{i=0}^{n-1} x_i 2^{i-m}$ | $(1-2x_0)\sum_{i=1}^{n-1} x_i 2^{i-m}$ | $\sum_{i=1}^{n-1} x_i 2^{i-m} - x_0 2^{m-n}$ |

**Table 3.1: Norm functions for common word encodings**

***Example 3.3:*** *Consider the following Boolean function, where ($x_2$, $x_1$, $x_0$) are bit-level variables, and output variables are grouped to form an integer at Boolean domain. Arithmetic Transforms can be obtained using the function truth table:*

| | |
|---|---|
| 000 | 2 |
| 001 | -11 |
| 010 | 8 |
| 011 | 2 |
| 100 | 18 |
| 101 | -14 |
| 110 | 17 |
| 111 | -8 |

$$AT(F) = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & -1 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 \\ -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 \end{vmatrix} * \begin{bmatrix} 2 \\ -11 \\ 8 \\ 2 \\ 18 \\ -14 \\ 17 \\ -8 \end{bmatrix} = \begin{bmatrix} 2 \\ -13 \\ 6 \\ 7 \\ 16 \\ -19 \\ 7 \\ 0 \end{bmatrix}$$

*Hence* $AT = 2 - 13x_0 + 6x_1 + 7x_1x_0 + 16x_2 - 19x_2x_0 + 7x_2x_1$

Arithmetic polynomials are used for efficient representation and calculation of multi-output functions $f_k$ , $f_{k-1}$, . . . , $f_0$ represented as integer-valued functions $f(z)$ via the mapping [57]:

$$f(Z) = \sum_{i=0}^{k} 2^i f_i$$

***Example 3.4:*** *Consider a system of functions:*

$$(f_2(x_2, x_1, x_0), f_1(x_2, x_1, x_0), f_0(x_2, x_1, x_0))$$

*where* $f_0(x_2, x_1, x_0) = x_2(x_0 + x_1)$

$f_1(x_2, x_1, x_0) = x_2x_0 \oplus x_1$

$f_2(x_2, x_1, x_0) = x_1 + x_2x_0$

*A matrix F whose columns are truth-vectors of $f_2$, $f_1$, and $f_0$, with their values interpreted as integers is used:*

$$F = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} = [F_2, F_1, F_0]$$

*An integer valued representation for $f_2$, $f_1$, and $f_0$ is obtained as $f = 2^2 f_2 + 2f_1 + f_0$, i.e,*

$$\begin{bmatrix} 0 \\ 0 \\ 3 \\ 3 \\ 3 \\ 4 \\ 6 \\ 7 \end{bmatrix} = 2^2 \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} + 2 \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} = 4F_2 + 2F_1 + F_0$$

*Now, the arithmetic spectrum of $F = [0, 0, 3, 3, 3, 4, 6, 7]^T$ is*

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & -1 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 \\ -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 6 \\ 6 \\ 0 \\ 7 \\ 7 \\ 5 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 6 \\ 0 \\ 0 \\ 7 \\ 1 \\ -9 \end{bmatrix} \begin{matrix} \leftrightarrow & 1 \\ \leftrightarrow & x_0 \\ \leftrightarrow & x_1 \\ \leftrightarrow & x_1 x_0 \\ \leftrightarrow & x_2 \\ \leftrightarrow & x_2 x_0 \\ \leftrightarrow & x_2 x_1 \\ \leftrightarrow & x_2 x_1 x_0 \end{matrix}$$

*Therefore, f is represented as the arithmetic polynomial*

$$f(z) = 6x_1 + 7x_2x_0 + x_2x_1 - 9\, x_2x\, x_0$$

*From the linearity of the arithmetic transform, this polynomial can be generated as the sum of the arithmetic polynomials for $f_1$, $f_2$, $f_3$.*

## 3.2.2 Utilization of Spectral Techniques

Spectral techniques have been applied for circuit synthesis, verification and testing by many researches. *Clarke et al.* [64] describe how to calculate concise representations of the Walsh transform for a Boolean function with huge variables. The technique is applied for Boolean technology mapping and obtains a speed up for matching case.

*Klaus* [65] develops a new method based on AT for the derivation of fault signatures for the detection of faults in single-output combinational networks. The signatures do not require exhaustive testing so they provide substantially less work than syndrome testing or the verification of Rademacher-Walsh spectral coefficients. Two counters are used to test spectral coefficients in [65] as the following figure.

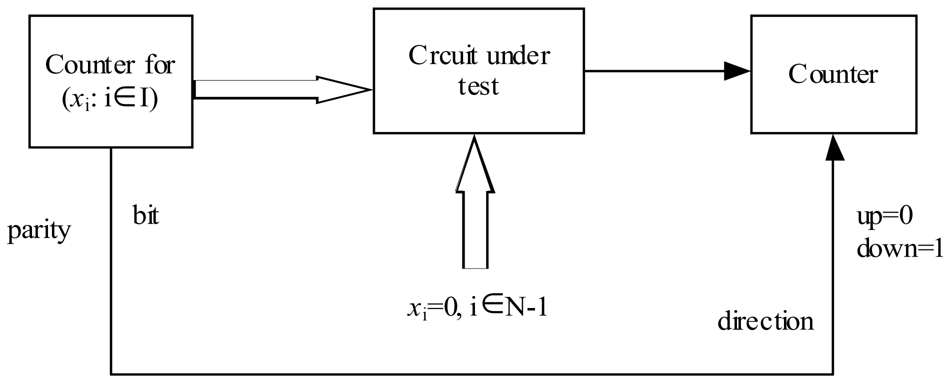


**Figure 3.5: The spectral coefficient $a_i$ test structure in [65]**

*Lui et al.* [66] use spectral signature testing methods for the model of multiple stuck-at faults. The testability condition for multiple-input faults is established and a minimal spanning signature (MSS) is defined to cover all these faults. A MSS contains a single spectral coefficient to detect over 99% of all input and internal multiple faults. The approach can obtain a complete signature for all multiple faults in any irredundant combinational network with small numbers of fan-outs and the possible overhead being an extra control input.

*Miller* and *Muzio* [67] describe a method for the derivation of fault signatures for certain classes or irredundant combinational networks. These signatures consist of a set of values derived from the network. Any stuck-at fault causes at least one of the values to change. The signatures provide complete fault detection for all single stuck-at faults.

*Radecka et al.* [68] exploit the algebraic properties of the AT that are used in the compact graph-based representations of arithmetic circuits. Verification time can be shortened under assumption of corrupting a bounded number of transform coefficients. Bounds are derived for a number of test vectors and the vectors successfully verify arithmetic circuits under a class of error models derived from proposed basic design error classes including single stuck-at faults.

In [135], authors describe a methodology for simulation-based verification in the presence of a fault model. The authors propose an implicit fault model that is based on the AT representation of a circuit and design faults. The proposed approach has the advantage of compatibility with formal verification and manufacturing testing methods. Errors can be modeled implicitly, and such an implicit error model is given by AT of a difference between the correct and faulty circuits. Since a fault is treated as a quantity added to the circuit output, the behavior $\tilde{f}$ of the faulty circuit is represented as a sum of the correct output and the error function *e*, that is, $\tilde{f} = f + e$. The relation:

$$AT(\tilde{f}) = AT(f) + AT(e)$$

is satisfied. The size of the error is measured in terms of the number of non-zero spectral coefficients in AT of the error *e*, that is, *AT(e)*. Based on the linearity feature, black-box verification can be performed without any knowledge of a circuit structure and implementation, as it is performed through design interfaces without accessing directly any of internal states.

## 3.2.3 Calculation of AT Coefficients

The definition of AT has been introduced. The usual method relies on matrix multiplication, which needs huge computation of multiplication and addition, so it is always inefficient. Past explorations investigate some other methods to calculate AT coefficients.

*Folkowski* and *Chang* [92] develop an algorithm to calculate the AT of the Boolean function from its OBDD representation. The method of decomposition of arithmetic spectral coefficients in terms of the cofactors of Boolean functions that resembles Shannon decomposition has been introduced. A new algorithm to synthesize OBDD from arithmetic spectrum is described.

Authors in [94] introduce a fast algorithm to generate AT. In that paper, different properties and ways of calculation for multi-polarity generalized arithmetic and adding transforms have been presented. Mutual relationships among spectra of different polarities have been discussed and the possibility to generate spectrum of an arbitrary polarity directly from the known spectrum of some polarity has been indicated. The following figure illustrates the fast algorithm.

*Krenz et al.* [95] present a fast algorithm for evaluating the arithmetic transform of a Boolean function based on its circuit representation. Unlike previous algorithms requiring an orthogonal and non-redundant representation or a single BDD, a new algorithm is proposed to partition the evaluation based on the dominator relations of the circuit graph. The dominators simplify intermediate evaluation steps greatly. So the algorithm can process larger circuits.

*Whitley et al.* [96] use representations of decision diagrams to calculate spectral coefficients by graph-based algorithms which produce Walsh, Arithmetic and Reed-Muller transforms for multi-output functions. *Thornton et al.* [97] propose matrix based techniques to calculate direct transformations amongst Walsh, Haar, Arithmetic and Reed-Muller spectral domains. They implement the fast transforms directly on decision diagrams.

*Moraga et al.* [98] introduce new diagrams based on AT, that is, arithmetic transform decision diagrams (ACDDs) which are the integer counterparts of the functional decision diagrams (FDDs). The paper describes how to construct the diagrams by the structure of arithmetic transform spectrum of Boolean functions. Example 3.6 shows an ACDD for a Boolean function.

***Example 3.5:*** Figure 3.6 *shows the ACDD for functions of n = 3 variables. Figure 3.7 shows the reduced ACDD for the Boolean function:*

$$f(x_1, x_2, x_3) = 3 - 2x_1 - x_2 + 4x_1x_2 + x_1x_3 + 2x_2x_3$$

*The constant nodes represent the arithmetic spectrum of f given by $A_f = [3\ 1\ 2\ 4\ 3\ 2\ 4\ 7]^T$.*

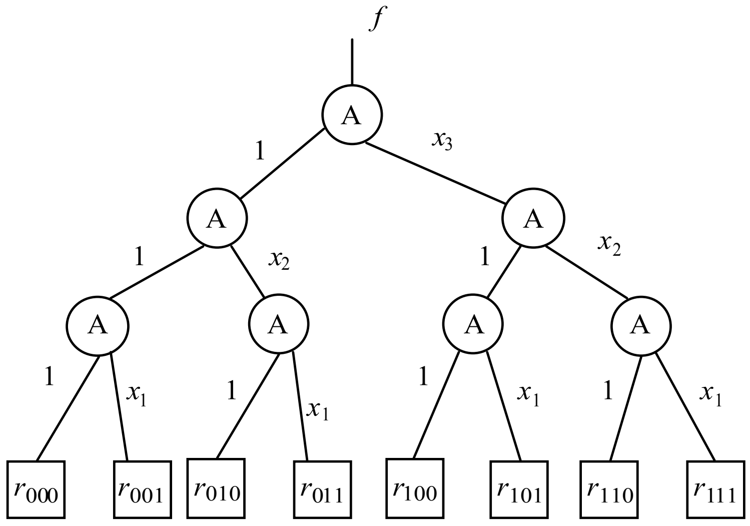


**Figure 3.6: ACDD for n=3**

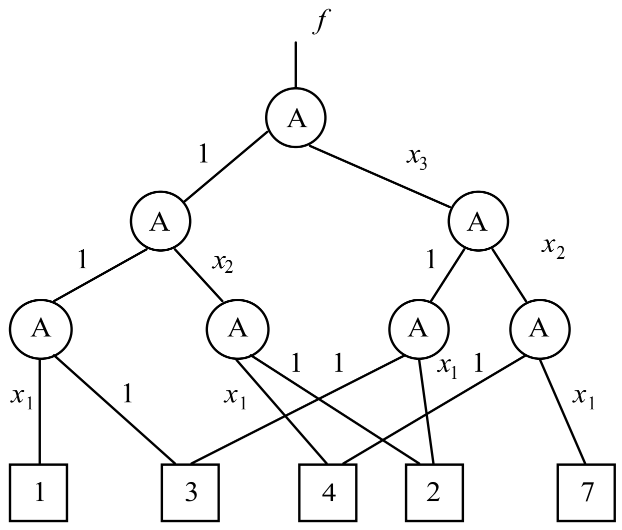


**Figure 3.7: ACDD of f in Example 3.6**

*Cintra et al.* [99] propose a unified theory for AT of a variety of discrete trigonometric transforms. Interpolation process is required and determines the transform. Authors also introduce a new algorithm to calculate the discrete Hartley transform by AT.

Past explorations calculate AT coefficients directly in spite of using matrix multiplication or starting from OBDDs or other function representations. The direct way sometimes leads to low efficiency especially for larger circuits. We design a new method to calculate AT in this chapter which is an indirect way by composing detached blocks in the circuit. First three extensions of AT are introduced.

## 3.3 Extensions of the Arithmetic Transform

Consider a circuit consisting of two blocks B1 and B2 in Figure 3.8. The composition of the two ATs: *P=AT(B1) and Q=AT(B2)* require the binary encoding, that is from the conversion of the word-level output *P* of the first AT into the bit-level values *T*, acceptable as inputs to the second AT [69].

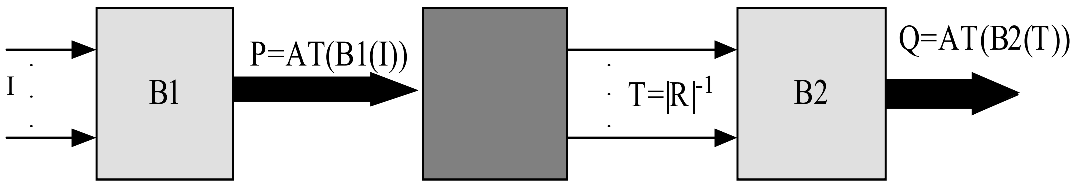


**Figure 3.8: Binary encoding use for compositions of ATs**

Instead of closed-form expression for binary encoding, the integer-to-binary conversion algorithm is applied to the AT polynomial to obtain $|w|^{-1}$. AT extensions should accept both word- and bit-level inputs because of no simple form of $AT(|w|^{-1})$.

The majority of digital circuits subject to verification are complex designs composed out of many smaller sub-blocks. AT can still be used to represent such designs, however in order to facilitate the compositions of ATs describing individual blocks (some of them may be sequential) we need to derive extensions to the basic AT. *Radecka and Zilic* [70] has proposed three extensions to represent complex combinational and sequential circuits. Here a summary introduces them shortly.

## 3.3.1 Mixed Arithmetic Transform

The first extension (MAT) facilitates the compositions of two or more AT blocks. The introduction of MAT is dictated by the incompatibility of inputs and outputs accepted and generated by AT. Note, that ATs in their original forms accept inputs as only binary variables, while for the compositions of ATs some of the inputs may be binary as well as word-level.

**Definition 3.4:** The ***Mixed AT (MAT)*** [69] *is a polynomial representing the function* $f : B^m \times w^k \to w$ *which uses an arithmetic "+" operation, word-level coefficients* $c_{i_1 i_2 \dots i_n}$, *binary* $x_1, x_2, \dots, x_m$ *and word-level* $w_1, w_2 \dots w_k$ *inputs as well as binary exponents* $i_1, i_2, \dots, i_n$ *and* $e_1, e_2, \dots, e_k$:

$$MAT\,(f) = \sum_{i_1=0}^{1} \dots \sum_{i_n=0}^{1} \sum_{e_1=0}^{1} \dots \sum_{e_k=0}^{1} c_{i_1 \dots i_n} x_1^{i_1} \dots x_m^{i_m} c_{e_1 \dots e_k} w_1^{e_1} \dots w_k^{e_k} \tag{3-3}$$

Eqn. (3-4) can be used to calculate the coefficients of a MAT, which is expanded around binary input variables, and treat word-level input quantities unassigned as symbols:

$$c(w_1, w_2 ... w_k) = T_n * f \qquad (3\text{-}4)$$

***Example 3.6:*** *Consider the MAT of a function f=3a+b, where "a" and "b" are 2-bit unsigned integers. We treat $a=a_1a_0$ as a bit vector, and "b" as a single word-level quantity. We obtain the truth table:*

$$f = [b\ 3+b\ 6+b\ 9+b]^T$$

*from which the AT transform application generates:*

$$MAT(f) = T * f = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} b \\ 3+b \\ 6+b \\ 9+b \end{bmatrix} = \begin{bmatrix} b \\ 3 \\ 6 \\ 0 \end{bmatrix}$$

*The resulting polynomial is $F(a_1a_0) = b+3a_0+6a_1$*

The size of the matrix $T_n$ is shrunk from 16*16 to 4*4 by treating the input *b* as word-level values. Therefore, the above example denotes that a MAT allows a compact way of generating AT.

A block represented by MAT can always be converted to the AT with polynomial size increase in wordlength *m*. A MAT is of importance for composing ATs by means of its word-level input variables, rather than for representing all functions. A function should be expressed explicitly in terms of designated word-level inputs.

## 3.3.2 Sequential AT Extensions

Since AT and MAT have no ability to represent sequential circuits, as there is no notion of time provided by these transforms, two extensions are introduced to allow variables to change over time to facilitate sequential implementations. We refer to such variables as *timed variables*.

**Definition 3.5:** ***The Timed variable*** *"v[n]" is a variable "v" to which a time tag "[n]" is assigned to indicate that the function generating the value of "v" changes with time instance "n"[70].*

Timed variables are used to abstract away the clock in the sequential implementation. A timed function *f[n]* represents the value of *f* in the $n^{th}$ clock period. The function *f[n]* is executed in a finite number of clock cycles.

***Example 3.7:*** *A timed equation of a memory element such as a flip-flop whose content is reloaded every clock cycle is defined as [70]:*

$$m_{out}[n] = m_{in}[n-1]$$

**Definition 3.6:** *The* ***AT Sequential*** **(*ATS*)** *is the Arithmetic Transform AT(f)[n] of timed function "f" at time instance "n", while the* ***MAT Sequential*** **(*MATS*)** *is analogously MAT(f)[n] of a timed function with word- and bit-level inputs [70].*

***Example 3.8:*** *Consider a standard flip-flop with input "D", reset signal "reset" and an enable signal "En" – all bit type is represented by ATS [70]:*

$$ATS\ (f)[n] = (1 - reset\ )(E_n * D + (1 - E_n) * f[n-1])$$

In fact, if intermediate variables generated by sequential elements are word-level quantities, the only appropriate sequential transform is an ATS.

The MATS of a sequential function "*f*" can be obtained from the MAT of the combinational part of "*f*" by the replacement of each MAT input that is generated by a memory element with its defining MATS. MATS have two forms. A type I MATS presents a case where the timed output variable *f* is expressed only in terms of timed input values, and a type II MATS describes a recurrence equation, where a symbol of a considered function *f* appears on both sides of a definition. The circuit behavior at a given time instance can be obtained through solving the recurrence equation analytically and symbolically by tools such as Maple or Mathmatica.

***Example 3.9:*** *In Figure 3.9(a), block A1 represents an N-bit adder. In the* $n^{th}$ *step, one summand is taken from primary inputs, while the other is supplied from multiplication of a constant and the register storing the values of the previous n-1 additions. The register has been initially reset.*

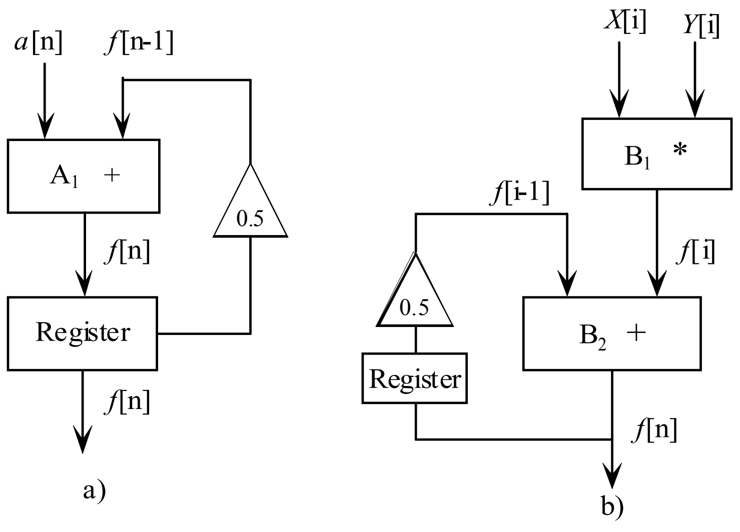


**Figure 3.9: Add- and Multiply-Accumulate Loops**

*The MATS of this loop is obtained by considering the register input f[n], with the value given by the recurrence:*

*MATS(f)[n]=a[n]+0.5*MATS(f)[n-1],          MATS(f)[0]=0*

*Its solution is:* $MATS(f)[n] = \sum_{i=1}^{n} 0.5^{n-i} a[i]$

*Then block B1 in Figure 3.9(b) represents an N*N-bit multiplier, and block B2 is a (2N+1)-bit adder creating a multiply-and-accumulate loop. The MATS results from the previously derived MAT transforms of its individual blocks. The inputs to the MAC loop at the time instance "i" are the N-bit binary vectors x[i] and y[i], and the output f[i] is a binary of size (2N+1). The ATS (all inputs are bits) of the multiplier B1 is defined for inputs at time instance "i":*

$$ATS(B1 = x * y)[i] = f[i] = \sum_{k=0}^{N} x_k[i]2^k * \sum_{k=0}^{N} y_k[i]2^k = a[i]$$

*The recurrence solution of the loop transform is:*

$$MATS\ (f)[n] = \sum_{i=1}^{n} 0.5^{n-i} a[i] = \sum_{i=1}^{n} 0.5^{n-i} (\sum_{k=0}^{N} x_k[i]2^k * \sum_{k=0}^{N} y_k[i]2^k)$$

Table 3.2 [70] clearly enumerates all definitions of transforms.

| Transform | Definition |
|---|---|
| AT | $AT\ (f) = \sum_{i_1=0}^{1}\sum_{i_2=0}^{1}\cdots\sum_{i_n=0}^{1} c_{i_1 i_2 \ldots i_n} x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n}$ |
| MAT | $MAT\ (f) = \sum_{i_1=0}^{1}\cdots\sum_{i_n=0}^{1}\sum_{e_1=0}^{1}\cdots\sum_{e_k=0}^{1} c_{i_1 \ldots i_n} x_1^{i_1} \ldots x_n^{i_n} c_{e_1 \ldots e_k} w_1^{e_1} \ldots w_k^{e_k}$ |
| ATS | AT transform *ATS(f)[n]* of a timed function *f* at a time instance *n* |
| MATS | MAT transform *MATS(f)[n]* of a timed function *f* at a time instance *n* |

**Table 3.2: Definitions of the AT and its extensions**

## 3.4 Composition Subroutines

After describing each design sub-block in terms of corresponding MAT, MATS or ATS, the overall AT can be constructed. Some of the approaches to the AT compositions focus on transferring ATs into decision diagrams [92]. However, due to their limitations, they are inadequate for many complex cases. In addition, factors such as running time and space are significant for these schemes. In this section we propose several subroutines to manage the complexity of constructing AT and its extensions.

### 3.4.1 Composition of AT and MAT

Composition of MAT and AT blocks can get a combinational circuit transform. While word-level variables are substituted by their AT polynomials, the overall circuit transform comes from the replacements and the Boolean algebra law $x_i^n = x_i \quad (n \neq 0)$. A block downstream must be represented by a MAT or an AT. Throughout the composition procedure, lots of intermediate terms would be generated and they should be combined for simplification, so running time and spaces are crucial factors that need attention. A best algorithm gets a tradeoff between them.

The following observation is a key to facilitating the combination of polynomial terms that become isomorphic by applying Boolean algebra rules to polynomials. A single, easy-to-calculate integer parameter referred to as an *index* of the term will be sufficient for finding isomorphic terms. We say that the index of the term is the integer encoded characteristic function of its variable indices. For instance, the index for the term $x_3^2 x_1 x_0^2$ is computed as $2^3+2^1+2^0 = 11$, and it is identical to the index of the term $x_3 x_1^3 x_0$ . Thus, the two terms are isomorphic terms and should be combined.

```
Compose_AT_MAT (AT_poly, MAT_poly)
{
1.  for (p=0; p<MAT_poly.term_num; p++)
    {
2.      for (i=0; i<MAT_term.wordvarnum; i++)
        {
3.          if (word_var[i] = AT_poly)
            {
4.              inter_term = Substitute (MAT_term, AT_poly);
5.              inter_term = Norm (inter_term);
6.              Store (inter_term, inter_poly);
            }
        }
7.      if (i = MAT_term.WordVarNum)
            Store (MAT_term, inter_poly);
    }
8.  Set_index (inter_poly);
9.  for (p=0; p<inter_poly.term-1; p++)
    {
10.     Adjust_term_position( term[p], term[p+1]);
11.     if (term[p].index = term[p+1].index)
            term[p].coeff += term[p+1].coeff);
    }
12. final_poly = inter_poly;   return final_poly;
}
```

**Figure 3.10: Algorithm of MAT and AT composition**

Figure 3.10 elaborates the subroutine in detail. The algorithm loops all terms in the MAT polynomial and searches whether the terms comprise the word-level variable represented by the AT polynomial. If so, the variable is expanded to form new terms; if not, the MAT terms are stored in an intermediate polynomial directly; the procedures are described in Step 1 - 7. After the loop is finished, an intermediate polynomial is obtained and all terms' indices are computed in Step 8. The algorithm then sorts terms with smaller indices forward, and if two terms have identical indices, the algorithm adds their coefficients. Ultimately, the composition polynomial is obtained, as reflected by Step 9 - 12. If the algorithm sorts and combines terms after each expansion procedure, it might be costly, so an intermediate polynomial is essential to cut computation time. Therefore, the procedures of adjustment and combination occur after all expansions are accomplished.

***Example 3.10:*** *Steps for composition of MAT and AT. Assume two modules with three primary inputs* $(x_2, x_1, x_0)$.

$$AT(f_1) = 1 + 2x_0 + x_1 - 4x_1x_0$$

$$MAT(f_2) = 2 - 3w_0 - 5x_1 + x_2 - 6w_0x_2 + 4x_2x_1$$

*A main loop begins with the first MAT term, a constant "2", until it reaches the last term "*$4x_2x_1$*". Since the first term of MAT does not contain the word-level number* $w_0$*, it is stored in an intermediate polynomial directly. The second term of MAT comprises the word-level variable, using* $w_0=AT(f_1)$ *as a substitute for expansion in this term. After simplification, the expanded terms are stored in the intermediate polynomial. When the loop is finished, an intermediate AT polynomial is obtained:*

$inter\_poly = 2 - 3 - 6x_0 - 9x_1 + 12x_1x_0 - 5x_1 + x_2 - 6x_2 - 12x_2x_0 - 18x_2x_1 + 24x_2x_1x_0 + 4x_2x_1$

*and the indices of the expanded terms are:*

*(0, 0, 1, 2, 3, 2, 4, 4, 5, 6, 7, 6)*

*Through position adjustment, the sequence sort orderly:*

*(0, 0, 1, 2, 2, 3, 4, 4, 5, 6, 6, 7)*

*Now, the intermediate polynomial changes:*

$inter\_poly = 2 - 3 - 6x_0 - 9x_1 - 5x_1 + 12x_1x_0 + x_2 - 6x_2 - 12x_2x_0 - 18x_2x_1 + 4x_2x_1 + 24x_2x_1x_0$

*Terms "2" and "-3", "*$x_2$*" and "*$-6x_2$*", "*$-18x_2x_1$*" and "*$4x_2x_1$*" are combined, and the overall AT polynomial is generated:*

$$AT(f) = -1 - 6x_0 + 14x_1 + 12x_1x_0 - 5x_2 - 12x_2x_0 - 14x_2x_1 + 24x_2x_1x_0$$

## 3.4.2 Composition of ATS and MATS

ATS and MATS have time tags, so the subroutine has a distinct step to process the tags. The difference is denoted in Figure 3.11. The returning polynomial is an ATS polynomial or a MATS polynomial.

Step 4 adds time tags of the word-level variables in the MATS polynomial to the ATS polynomial and then expands the MATS term. If two identical bit-level variables in an expanded term have same time tags, they must be combined，for instance, a term of $5x_0[n-2]x_1[n-1]x_1[n-1]$ is simplified as $5x_0[n-2]x_1[n-1]$. This procedure is described in Step 7. After the intermediate polynomial is generated, if two terms have identical indices, and corresponding variables in the two terms also have same time tags, the algorithm combines their coefficients. Step 13 - 15 elaborate the procedure.

```
Compose_ATS_MATS (ATS_poly, MATS_poly)
{
1.   for (p=0; p<MAT_poly.term_num; p++)
     {
2.       for (i=0; i<MATS_term.wordvarnum; i++)
         {
3.           if (word_var[i] = ATS_poly)
             {
4.               Add_time(word_var[i].tag, ATS_poly);
5.               inter_term=Substitute( MATS_term,ATS_Poly);
6.               for (k=0; k<inter_term.varnum-1; k++)
                 {
7.                   if ( var[k].index = var[k+1].index && var[k].tag = var[k+1].tag)
                         Norm( inter_term);
                 }
8.               Store (inter_term, inter_poly);
             }
         }
9.       if (i = MATS_term.wordvarnum)
             Store (MATS_term, inter_poly);.
     }
10.  Set_index (inter_poly);
11.  for (p=0; p<inter_poly.term_num-1; p++)
     {
12.      Adjust_term_position(term[p], term[p+1]);
13.      if (term[p].index = term[p+1].index )
         {
14.          if term[p].var[k].tag!=term[p+1].var[k].tag)
                 term[p].coeff += term[p+1].coeff);
         }
     }
15. final_poly = inter_poly;    return final_poly;
}
```

**Figure 3.11: Algorithm of MATS and ATS composition**

***Example 3.11:*** *Steps for composition of MATS and ATS.*

$ATS(f_1) = 1 + 2x_0[n-1] + 3x_1[n-1]$

$MATS(f_2) = w_0 - w_1[n-2] - 4w_0[n-1]x_0[n-2]$

*MATS includes two word-level variables $w_0$ and $w_1$, and $w_0 = ATS(f_1)$, therefore the overall transform is a MATS polynomial. A loop begins with the first MATS term $w_0$ and it contains the ATS output variable $w_0$, so it is substituted by $ATS(f_1)$ and expanded terms are stored in an intermediate polynomial. The second term comprises another word-level variable so it does not need expansion. The last term has a word-level variable with a time tag and it is accumulated to ATS tags, since two $x_0$ variables have same tags "2", they are combined.*

$inter\_poly = 1 + 2x_0[n-1] + 3x_1[n-1] - w_1[n-2] - 4x_0[n-2] - 8x_0[n-2] - 12x_0[n-2]x_1[n-2]$

*Through position adjustment and combination of isomorphic terms, the overall transform is generated:*

$MATS(f) = 1 - w_1[n-2] + 2x_0[n-1] - 12x_0[n-2] + 3x_1[n-1] - 12x_0[n-2]x_1[n-2]$

The other two subroutines, Composition of ATS and MAT, and Composition of AT and MATS, are similar to the mentioned subroutines. They are omitted here.

## 3.5 Overall Composition Algorithm

Each block represented by a corresponding transform is as a node defined by a data structure to describe its properties to facilitate composition of detached blocks. The suitable structure definition is:

```
{   unsigned long type;        unsigned long type_index;
    unsigned long level;       unsigned long in_word_num;
    char *in_index;            char out_index;   }
```

The parameter *type* indicates which the transform type is corresponding to AT, ATS, MAT or MATS; *type_index* evaluates its index inside nodes which have same type with this node; *level* determines its depth in the constructed diagram, and blocks with primary inputs are always set “0”; *in_word_num* indicates the number of input word-level variables, *in_index* stores indices of input word-level variables and *out_index* stores the index of its output word-level variable. Figure 3.12 outlines steps to compose modules to get an overall transform.

```
1. for (i=0; i<node_num; i++)
       Set_property (node[i]);
2. for (i=0; i<node_num; i++)
3. {   if (node[i].type = 2 or 3)        // MAT or MATS
4.     {   for (j=0; j<node_num; j++)
5.         {     if (node[j].out_word_index = node[i].in_word_index)
                 {
6.                   if (node[i].level<node[j].level+1)
                         node[i].level=node[j].level+1;
                 }
           }
       }
   }
7. current_level = 1;
8. for (i=0; i<node_num; i++)
   {
9.     if (node[i].level = current_level)
       {
10.        for (j=0; j<node_num; j++)
           {
11.            if (node[j].out_index = node[i].in_index)
               {
                   new_node = Subroutine(node[i], node[j];
                   Set_property ( new_node);
               }
           }
       }
12.    current_level++;
   }
```

**Figure 3.12: The overall composition algorithm**

The most important issue confirming the parameter *level* of each node at the block-level netlist is dedicated in Step 2 - 6. The “level” parameter builds a hierarchy to designate a composition path. The composition procedure always begins from AT or ATS with primary inputs, and they are set to level “0”. While it goes forward according to the current level, and encounters a block which has an identical level with the current level, the algorithm invokes a

corresponding subroutine in terms of the block's type, eventually the overall transform of the circuit is achieved, and please note this transform with primary inputs does not contain any intermediate variables, so
the final transform is AT or ATS.

***Example 3.12:*** *Consider a circuit consisting of four nodes with four primary input bits as Figure 3.13. Each word-level output is assigned to a different index. By the composition algorithm, we get its overall transform.*

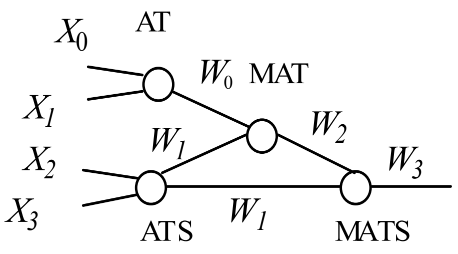


**Figure 3.13: A circuit with 4 modules**

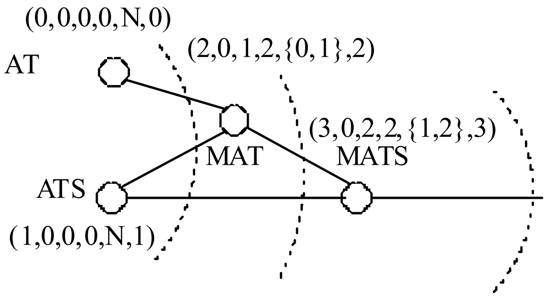


**Figure 3.14: Node properties**

*First, each node properties are labeled through step 1 - 6 in Figure 3.14. N represents NULL and the MATS node has the largest level "2".*

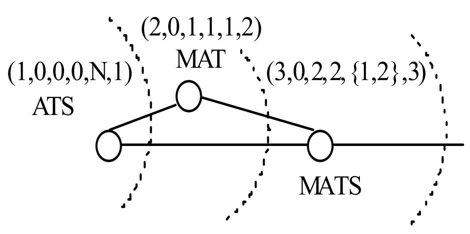


**Figure 3.15: Composing the MAT and the AT nodes**

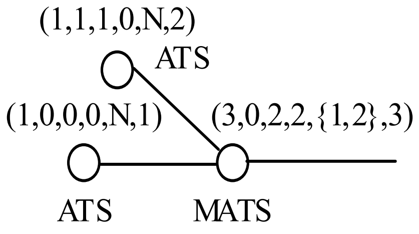


**Figure 3.16: Composing the MAT and the ATS nodes**

*A parameter current_level is set to "1" at the beginning, and the algorithm searches which nodes has a level the same as the current_level. It is the AT node in this case and its out word-level variable is one of the input variables in the MAT node. The algorithm calls Compose_MAT_AT function and since the MAT node has two different word-level variables, it generates a new MAT mode as in Figure 3.15. Next, the algorithm finds that the ATS output variable is another input variable of the MAT node. Therefore, it calls the subroutine Compose_MAT_ATS and gets a new ATS node in Figure 3.16.*

*While no other nodes have same level, the parameter current_level is increased by 1, to become 2. The algorithm matches it with the MATS node, and then the subroutine of Compose_MATS_ATS can be invoked.*

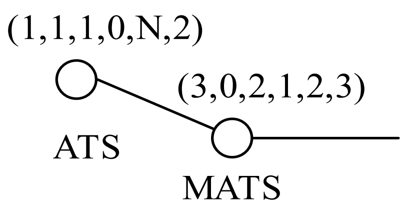


**Figure 3.17: Composing the MATS and the ATS nodes**

*Finally, an ATS polynomial is obtained through the composition of the new MATS node and the remaining ATS node.*

From the example, one can notice that the algorithm follows a fixed order determined by the parameter “level” to compose block representations. Its logic is easy to follow, to implement simply for arbitrary topologies and even transforms.

## 3.6 Experimental Results

In this section, the composition algorithm in Figure 3.12 is verified by several benchmarks such as ALU, CSA and MAC.

### 3.6.1 ALU Circuit Implementation

Arithmetic Logic Unit (ALU) is a necessary block at microchips. It takes charge of data operations, including arithmetic, logic and relation operations, and stores results in memory. Figure 3.18 illustrates a typical ALU model. The AT of an adder is:

$$AT(f_1) = \sum_{i=0}^{N-1} (2^i x_i + 2^i y_i)$$

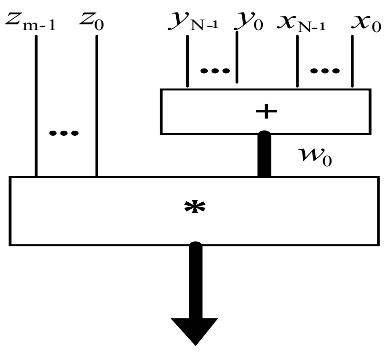


**Figure 3.18: An ALU model**

Inputs of a multiplier consist of bit-level variables and a word-level variable which is from the output of the adder, so the multiplier has MAT form:

$$MAT\ (f_2) = w_0 * \sum_{k=0}^{m-1} 2^k z_k$$

| Adder Inputs | Multiplier Inputs | Adder Terms | Multiplier Terms | AT Terms | Time [s] |
|---|---|---|---|---|---|
| 12 | 7 | 12 | 7 | 84 | 0.875 |
| 14 | 8 | 14 | 8 | 112 | 1.672 |
| 16 | 9 | 16 | 9 | 144 | 3.834 |
| 24 | 13 | 24 | 13 | 312 | 13.4 |
| 32 | 17 | 32 | 17 | 544 | 34.3 |

**Table 3.3: Results for the ALU transform**

Table 3.3 gives parameters of the adder and multiplier inputs and gets the number of their transform terms based on given input variables. It reveals the overall transform terms number after composition.

## 3.6.2 CSA Circuit Implementation

Carry-Select Adder (CSA) is a common implementation of adders, which computes alternative results in parallel and subsequently selects the correct results with single or multiple stage hierarchical techniques. The carry-select adder increases its area requirements for purpose of enhancing its speed performance. In carry-select adders both sum and carry bits are calculated for the two alternatives: input carry “0” and “1”. Once the carry-in is delivered, the correct computation is chosen by a multiplexer to generate a desired output. Therefore waiting for the carry-in to calculate the sum is avoidable, and the sum is correctly generated as soon as the carry-in gets there. The obvious advantage is that CSA largely reduces time of computing the sum. Two adders share 8-bit inputs variables and have different input carry. The adder transform is:

$$AT(f_1) = \sum_{i=0}^{N-1} 2^i x_i + \sum_{i=0}^{N-1} 2^i y_i + carry$$

The multiplexer transform is:

$$MAT(f_2) = (1-c)w_0 + cw_1$$

Here $c$ is a bit-level variable and ($w_0$, $w_1$) are word-level variables from outputs of the two adders. The concept is illustrated in Figure 3.19.

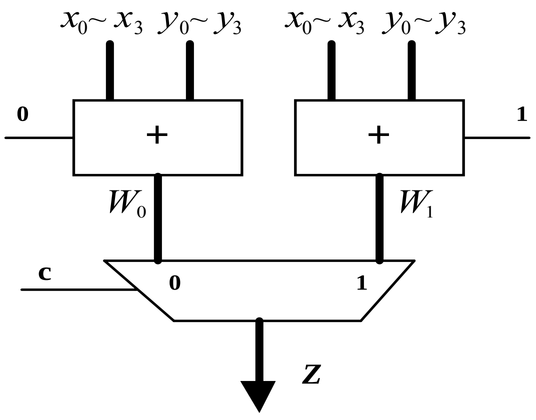


**Figure 3.19: 4-bit carry select adder**

Since the MUX transform has two word-level variables, an intermediate MAT polynomial is generated for convenience to incorporate one word-level variable. The seventh column of Table 3.4 indicates the space requirements.

| Inputs | Adder Terms | MUX Terms | Inter Terms | AT Terms | Time (s) | Space (MB) |
|---|---|---|---|---|---|---|
| 24 | 25 | 3 | 49 | 25 | 0.1 | 0.02 |
| 32 | 33 | 3 | 65 | 33 | 0.18 | 0.036 |
| 40 | 41 | 3 | 81 | 41 | 0.26 | 0.058 |
| 48 | 49 | 3 | 97 | 49 | 0.35 | 0.073 |
| 56 | 57 | 3 | 113 | 57 | 0.44 | 0.092 |
| 64 | 65 | 3 | 129 | 65 | 0.53 | 1.2 |

**Table 3.4: Results of CSA transforms**

It is apparent that even when the number of input bits becomes large, the running time and space requirement remain modest. The program provides an effective interface to process sparse coefficients which comprise lots of “0” values. Hence, the time is dominated by the number of non-zero AT terms, rather than being possibly exponential function of the number of input bits. We observe that additional speedup can be obtained by relying on the equivalence checking of the individual blocks, before the module is incorporated in larger netlist. As inclusion of AT of individual blocks is less costly than the construction by a netlist traversal of those blocks.

## 3.6.3 MAC Transform

The AT specification of a MAC circuit from Figure 3.20 can be determined by combining AT, MAT, and MATS components. The unit is built using shift registers, a multiplier, and an adder-register loop.

The expression of a MAC is shown below:

$$f[n] = \sum_{i=0}^{n-1} \left(\sum_{k=0}^{N-1} 2^k x_k[i] * \sum_{k=0}^{N-1} 2^k y_k[i]\right)$$

The equation should be solved at a time instance *n* to obtain the MAC transform. For example, for *n*=8 and *N*=2, the ATS of the multiplier is:

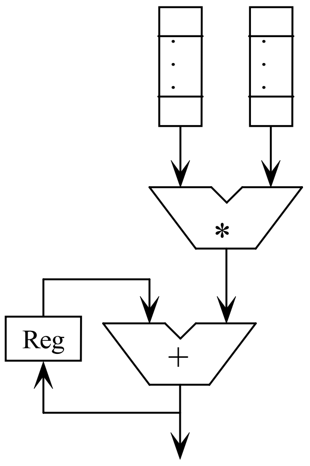


**Figure 3.20: Implementation of a MAC**

$$ATS(mul[k]) = x_0[8-k]y_0[8-k] + 2x_1[8-k]y_0[8-k] + 2x_0[8-k]y_1[8-k] + 4x_1[8-k]y_1[8-k]$$

The overall ATS is given by followed equation:

$$ATS(f) = \sum_{k=1}^{8} x_0[8-k]y_0[8-k] + \sum_{k=1}^{8} 2x_0[8-k]y_1[8-k] +$$

$$\sum_{k=1}^{8} 2x_1[8-k]y_0[8-k] + \sum_{k=1}^{8} 4x_1[8-k]y_1[8-k]$$

| Word Size | Time Instance | AT Terms | Time(s) | Space (MB) |
|---|---|---|---|---|
| 8 | 4 | 256 | 0.137 | 0.085 |
| 8 | 8 | 512 | 0.465 | 0.14 |
| 8 | 16 | 1024 | 1.28 | 0.26 |
| 16 | 4 | 1024 | 1.459 | 0.28 |
| 16 | 8 | 2048 | 3.251 | 0.46 |
| 16 | 16 | 4096 | 6.874 | 0.91 |
| 32 | 16 | 16384 | 25.43 | 3.82 |
| 32 | 32 | 32768 | 55.8 | 7.46 |
| 32 | 64 | 65536 | 132.9 | 15.8 |

**Table 3.5: Results of MAC transforms**

Table 3.5 displays results of the MAC implementation. Column 1 and 2 denote its word-level variable size and time instance value. Even though the AT terms grows exponentially with word size, the computation time and space are satisfied.

## 3.6.4 Implementation of Hilbert Transform

Hilbert transform is a useful mathematical tool to describe the complex envelope of a real-valued carrier modulated signal. The definition of the Hilbert transform is as follows:

$$\hat{s}(t) = (h * s)(t) = \int_{-\infty}^{\infty} s(\tau)h(t-\tau)d\tau = \frac{1}{\pi}\int_{-\infty}^{\infty} \frac{s(\tau)}{(t-\tau)}d\tau$$

where $h(t) = \frac{1}{\pi t}$.

The Hilbert transform has a frequency response given by the Fourier Transform:

$$H(w) = F\{h\}(w) = -i * \operatorname{sgn}(w)$$

where

$$\operatorname{sgn}(\omega) = \begin{cases} 1, & \text{for } \omega > 0, \\ 0, & \text{for } \omega = 0, \\ -1, & \text{for } \omega < 0, \end{cases}$$

The Hilbert transform has the effect of shifting the negative frequency components of $s(t)$ by +90 degrees and the positive components by -90 degrees. Generally, FIR is a good realization of Hilbert Transform. Figure 3.21 gives a FIR structure.

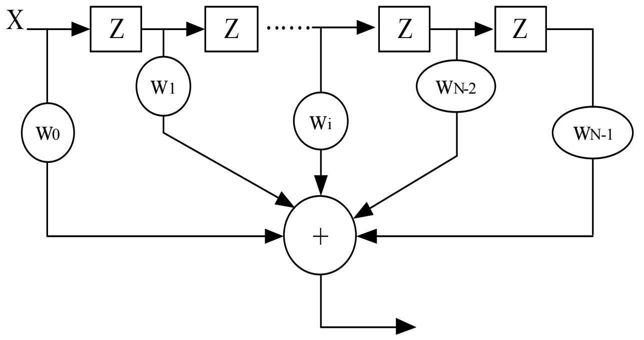


**Figure 3.21: A FIR model to realize Hilbert transform**

The timed register equation is:

$$m_{out}[n] = m_{in}[n-1]$$

The MAT of adder is:

$$MAT\ (f) = \sum_{i=0}^{N-1} X_i$$

where $X_i$ is a word-level input from each tap output.

| Taps | Word Size | ATS Terms | Time(s) | Space(MB) |
|---|---|---|---|---|
| 32 | 16 | 512 | 0.21 | 0.56 |
| 32 | 32 | 1024 | 0.39 | 1.3` |
| 32 | 64 | 2048 | 0.72 | 2.53 |
| 64 | 16 | 1024 | 0.53 | 1.22 |
| 64 | 32 | 2048 | 0.98 | 2.54 |
| 64 | 64 | 4096 | 1.87 | 5.1 |
| 128 | 16 | 2048 | 0.78 | 2.55 |
| 128 | 32 | 4096 | 1.98 | 5.23 |
| 128 | 64 | 8192 | 4.05 | 10.68 |

**Table 3.6: Results of Hilbert transforms**

The FIR implementation has a structure that is easily represented by ATS. Furthermore, the task of equivalence checking or the verification of imprecise implementations can facilitate to verify whether the implementation fits the specification.

## 3.7 Conclusions

AT is the most important representation in our research, so in this chapter the spectral techniques and the basic definition of AT were introduced. Although AT can represent an arithmetic circuit compactly, it has limitations. The proposed three extensions for representing combinational and sequential circuits were outlined. Getting the circuit transform is significant for verification. Direct computation sometimes requires too much time for these processes. We proposed a topological method of composing the transforms of detached blocks to facilitate the calculation, so it is easy to obtain the overall transform for a complex circuit. The experiments proved its high efficiency.

# Chapter 4

# Basic Algortihms

*Imprecise circuit specifications such as Taylor series complicate the process of design and verification. We adopt a spectral technique, Arithmetic Transform (AT), to process the imprecise circuits. In this chapter, three basic algorithms based on AT are described which convert polynomials and search for the maximum absolute value. These are fundamental algorithms for the verification and optimization in following chapters.*

The fixed-point representation problem includes two facets, the precision problem and the range problem. Beginning in this chapter, we explore the precision problem. First, the typical imprecise representation is introduced.

## 4.1 Taylor Series

In mathematics, the **Taylor series** is a representation of a function as an infinite sum of terms calculated from the values of its derivatives at a single point. Let *f(X)* be a real and differentiable function corresponding to an algebraic expression. The variables are real numbers with usual field operations (+,*) over real numbers R.

**Definition 4.1:** *The function can be represented as* ***Taylor series*** *using a variable X and an initial constant* $X_0$.

$$f(X)=\sum_{n=0}^{\infty}\frac{1}{n!}(X-X_0)^n f^{(n)}(X_0)$$

$$= \quad f(X_0)+Xf^{'}(X_0)+\frac{X^2}{2}f^{''}(X_0)...+\frac{(X-X_0)^n}{n!}f^{(n)}(X_0)+R_n(X)$$

*where* $f^{'}(X), f^{''}(X)$*, etc, are first, second and higher derivatives of f(X), and* $R_n(X)$ *is a Lagrange remainder.*

The error R is bounded, using point $\xi$ in the interval *I*, as:

$$R_n(X)=\frac{f^{(n+1)}(\xi)}{(n+1)!}(X-X_0)^{n+1} \qquad (4\text{-}1)$$

Taylor series can be used to calculate the value of an entire function in every point, if the value of the function, and of all of its derivatives, are known at a single point. Uses of the Taylor series for entire functions include:

- The partial sums (the Taylor polynomials) of the series can be used as approximations of the entire function. These approximations are good if sufficiently many terms are included.
- The series representation simplifies many mathematical proofs.

If this series converges for every *x* in the interval (*a* − *r*, *a* + *r*) and the sum is equal to *f*(*x*), then the function *f*(*x*) is analytic in the interval (*a* − *r*, *a* + *r*). If this is true for any *r* then the function is an entire function. One normally uses estimation for the remainder term of Taylor's theorem to check whether the series converges towards *f*(*x*). A function is analytic iff it can be represented as a power series; the coefficients in that power series are then necessarily the ones given in the above Taylor series formula.

Many transcendental arithmetic functions such as sin(*X*) and log(*X*) are realized through Taylor series. For example, Taylor series of sin(*X*) is:

$$\sin(X) = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$$

Naturally, any hardware realization implements finite terms of Taylor series, which invariably would lead to an error. Imprecision further comes from a finite-word representation of real numbers. The precision analysis is therefore necessary to make use of the fixed-point number representation, which is attractive in balancing complexity, cost and energy consumption.

Both of the above approximations cause the implementation imprecision error. The first case from truncation of Taylor terms is easy to evaluate. The remainder $R_n(X)$ has an explicit expression and can be estimated without actually computing. The most common estimation is based on bounding the absolute value of the $n^{th}$ order derivative on the entire interval that contains the intermediate point $\xi$. While estimating the derivative on a given interval, it is not necessary to find the exact maximum of a function, for most cases trying to find some upper bound is not too rough. Therefore our emphasis concentrates on the error due to finite wordlength. Arithmetic Transform (AT) is used to investigate the imprecision.

## 4.2 Algorithm for AT Conversion by Taylor Series

Many arithmetic functions can be represented as (infinite) Taylor series, however their hardware realization inevitably leads to imprecision due to the restrictions regarding the finite number of terms to be implemented. Any imprecision of the implementation causes a circuit to behave differently with the assumed specification. Nevertheless, known imprecision cannot be treated as unintended errors committed during the design process. Therefore, we accept the design to be fault free, if its behavior differs from specification within assumed error interval. We convert the Taylor series specification/design representation into a corresponding AT to evaluate the error upper bound of the implementation. This step is needed in order to integrate the verification of the imprecisely implemented blocks into the overall verification scheme proposed in this work, and based on the Arithmetic Transform data representation.

AT is canonical, and will be used to directly represent approximation and imprecision errors coming from the finite Taylor series function representations. The correspondence between Taylor and AT representation is illustrated by the following lemma.

**Lemma 4.1**: *Consider a finite Taylor polynomial around $X_0=0$ where the variable X will be represented as an m-bit unsigned fractional number. By denoting $f_0^{(i)}=f^{(i)}(X_0)$, we have:*

$$f(X) = f_0 + Xf_0' + \frac{X^2}{2!}f_0'' + \cdots + \frac{X^{n-1}}{(n-1)!}f_0^{(n-1)}.$$

*The AT of f(X) is expanded from the Taylor polynomial as:*

$$AT[f(X)] = f[AT(X)] =$$

$$f_0 + f_0'(\sum_{i=0}^{m-1}2^{-(i+1)}x_i) + \frac{f_0''}{2!}(\sum_{i=0}^{m-1}2^{-(i+1)}x_i)^2 \cdots + \frac{f_0^{(n-1)}}{(n-1)!}(\sum_{i=0}^{m-1}2^{-(i+1)}x_i)^{n-1}$$

*Proof: The transform of an m-bit unsigned fractional number X is $AT(X)=\sum_{i=0}^{m-1}2^{-(i+1)}x_i$. Since AT is linear, that is, $AT(f_1+f_2) = AT(f_1)+AT(f_2)$ and $AT(C*f) = C*AT(f)$, where C is a constant, we can obtain:*

$$AT[f(X)] = AT\,(f_0 + Xf_0' + \frac{X^2}{2!}f_0'' \cdots + \frac{X^{n-1}}{(n-1)!}f_0^{(n-1)})$$

$$= AT(f_0) + AT(Xf_0') + AT(\frac{X^2}{2!}f_0'') \cdots + AT(\frac{X^{n-1}}{(n-1)!}f_0^{(n-1)})$$

$$= f_0 + f_0' AT(X) + \frac{f_0''}{2!} AT(X^2) ... + \frac{f_0^{(n-1)}}{(n-1)!} AT(X^{n-1})$$

$$= f_0 + f_0' (\sum_{i=0}^{m-1} 2^{-(i+1)} x_i) + \frac{f_0''}{2!} (\sum_{i=0}^{m-1} 2^{-(i+1)} x_i)^2 \cdots + \frac{f_0^{(n-1)}}{(n-1)!} (\sum_{i=0}^{m-1} 2^{-(i+1)} x_i)^{n-1}$$

$$= f[AT(X)] \qquad \square$$

Lemma 4.1 denotes that *AT*[*f*(*X*)] results from substituting expanded bit-level variables for the word-level variable *X* in *f*(*X*). By combining coefficients of isomorphic terms in the expanded polynomial, the AT representation in Def. 3.3 is obtained, thus leading to the conversion of Taylor expansions to AT.

While Lemma 4.1 might seem to lead to a simple realization of the conversion between Taylor and AT, in reality the process could be time- and memory-consuming. To evaluate the imprecision error using AT, the specification should be translated into AT as well. In this section we describe the conversion of Taylor series into AT by expansion from Lemma 4.1. A straightforward method for generating *AT*[*f*(*X*)] replaces each monomial in Taylor series *f*(*X*) by its defining AT, followed by the *consolidation* of AT terms. Although the overall conversion procedure is conceptually simple, the expansion of the real-valued quantities from Taylor series into word-level AT terms can lead to a large intermediate polynomial, similar to what is known to happen in symbolic computing.

By the rule that Boolean algebra $x_i^n$ equals $x_i$, lots of expanded terms are identical and they should be combined to form a simplified AT polynomial. A straightforward method multiplies each factor recursively, and gets an intermediate polynomials, then simplifies it by using the Boolean rule, so the AT polynomial is achieved. Although the procedure is easy to comprehend, complexity in the calculation comes from large Taylor degrees and bits number which leads to a large size of the intermediate polynomial since it comprise a great many expanded terms.

For example, with degree k=7 and input bits N=16, the number of intermediate terms increases to over 2000000. Consequently, storage and grouping of the same terms are major hurdles and result in low efficiency. We now show how to perform conversion into AT polynomial that handles efficiently the intermediate data swell.

### 4.2.1 Expansion Formula

The key problem in converting Taylor series into an AT polynomial is the calculation of the corresponding AT terms $(\sum_{i=0}^{N-1} 2^i x_i)^k$. Assume $N \le k$, and the above sum can be obtained as:

$$(\sum_{i=0}^{N-1} 2^i x_i)^k = \sum_{i=0}^{N-1} (2^i x_i)^k + C_k^p (2x_1)^p x_0^{k-p} + C_k^p C_{k-p}^q (4x_2)^p (2x_1)^q x_0^{k-p-q} \ldots \qquad (4\text{-}2)$$

where $C_k^m$ is defined as $C_k^m = \begin{pmatrix} k \\ m \end{pmatrix}$. Based on Eqn. (4-2), we find that the intermediate coefficients of the isomorphic terms must be combined to simplify the obtained AT. The structure of equation will be explored to reveal the possibility to derive an efficient conversion algorithm. In particular, the following property is used for efficient grouping of common terms.

**Property 4.1:** *For AT raised to the exponent k, Eqn. (4-2), the sum of the individual variable's exponent is k for each term.*
*Proof: The calculation of the sum requires k-1 multiplication, where all bit-level variables in a single factor have a fixed component '1". Through each multiplication procedure, the term's exponent augments one and its beginning exponent is also one, so finally the total exponent is k-1+1= k.*

**Property 4.2:** *If an AT term has p variables, the largest exponent which a variable can obtain is the Taylor degree k subtracting variables number p plus 1, and the least exponent is 1 in all expanded isomorphic terms.*
*Proof: If a variable appears in an AT term, that's easy to know it has an exponent "1" at the lowest. In terms of Property 4.1, the summed exponent of the p variables is Taylor degree k, while other p-1 variables all have a least exponent "1", the variable can get the largest exponent, etc., k-p+1.*

Towards that goal, some definitions are in place. An integer coefficient multiplying expanded terms is named "***weight***". For example, in the expanded term $2560x_2^3x_1x_0$, "2560" is its weight. We refer to final terms after combination as "***AT terms***". Next, ***msv*** and ***lsv*** represent most significant and least significant variables, respectively, in an AT term. For instance, for the AT term $x_2x_1x_0$, $x_2$ is msv and $x_0$ is lsv; for the AT term $x_1x_0$, $x_1$ is msv and $x_0$ is lsv. The algorithm requires two computation steps: one gets forms of isomorphic terms, which is most important to determine performance; the other calculates the weight of each expanded term.

## 4.2.2 Isomorphic AT Terms Combination

The following example describes the expanded terms.

***Example 4.1***: *Given three input bits ($x_2$, $x_1$, $x_0$) and Taylor degree k=5, the expansion is:*

$$
\begin{aligned}
(\sum_{i=0}^{2} 2^i x_i)^5 &= x_0^5 + (2x_1)^5 + C_5^4(2x_1)^4x_0 + C_5^3(2x_1)^3x_0^2 + C_5^2(2x_1)^2x_0^3 \\
&+ C_5^1(2x_1)x_0^4 + (4x_2)^5 + C_5^4(4x_2)^4x_0 + C_5^3(4x_2)^3x_0^2 + C_5^2(4x_2)^2x_0^3 \\
&+ C_5^1(4x_2)x_0^4 + C_5^4(4x_2)^4(2x_1) + C_5^3(4x_2)^3(2x_1)^2 + C_5^2(4x_2)^2(2x_1)^3 \\
&+ C_5^1(4x_2)(2x_1)^4 + C_5^3C_2^1(4x_2)^3(2x_1)x_0 + C_5^2C_3^2(4x_2)^2(2x_1)^2x_0 \\
&+ C_5^2C_3^1(4x_2)^2(2x_1)x_0^2 + C_5^1C_4^3(4x_2)(2x_1)^3x_0 + C_5^1C_4^2(4x_2)(2x_1)^2x_0^2 + C_5^1C_4^1(4x_2)(2x_1)x_0^3
\end{aligned}
$$

One can easily see that the degree of every bit-level variable amounts to *k* in each expanded term due to the property, etc., the summed degree of $x_2^3x_1x_0$ is 3+1+1=5. There are $2^N$-1=7 AT terms as ($x_0$, $x_1$, $x_1x_0$, $x_2$, $x_2x_0$, $x_2x_1$, $x_2x_1x_0$). The isomorphic terms for the AT term $x_2x_1x_0$ in the expanded equation is obtained as:

$$2560x_2^3x_1x_0, 1920x_2^2x_1^2x_0, 960x_2^2x_1x_0^2, 640x_2x_1^2x_0, 480x_2x_1^2x_0^2, 160x_2x_1x_0^3$$

Now we show how to get all isomorphic terms for an arbitrary AT term such as $x_2x_1x_0$ under a specific Taylor degree. A tuple (m,o,p) expresses variable degrees of $x_2$, $x_1$ and $x_0$. At beginning msv $x_2$ is set to the largest degree "3", and degrees of $x_1$ and $x_0$ are "1" according to Property 4.1 and 4.2. The first degree representation is (3,1,1) and after that a next degree representation is computed. Beginning from lsv $x_0$, preceding variables are searched until one variable with the degree larger than “1” is discovered. In the case considered here, such a variable is $x_2$. Therefore its degree decreases one and the degree of the succedent variable increases one. After this iteration the degree representation is changed to (2,2,1). The computation process continues until lsv $x_0$ is set to the largest degree 3, and degrees of other two variables are both 1. At this time, the degree representation turns into (1,1,3). Transformation of the degree sequence is:

$$(3,1,1)\rightarrow(2,2,1)\rightarrow(2,1,2)\rightarrow(1,3,1)\rightarrow(1,2,2)\rightarrow(1,1,3)$$

Here, the sequence determines the movement order of degree representations, and guarantees them not to be repeated or missed. Also it makes an easy implementation by a program.

### 4.2.3 Weights of Expanded Terms

Next we calculate terms’ weights. They are obtained by an input binary weight multiplying a combination constant. For example, in the case of an expanded term $C_5^2C_3^2(4x_2)^2(2x_1)^2x_0$, the input binary weight equals to $4^2*2^2*1=64$, and the combination constant is $C_5^2C_3^2=30$. Using variable indices simplifies the computational process of the input binary weight, so the problem reduces to getting the combination constant. The terms number of the combination constant is $N$-1 (result of the last $N^{\text{th}}$ term is always 1, so it is neglected). According to Equation (3-3), the first term is $C_k^p$, where $k$ is the total degree (5 in considered case), and $p$ is the degree of first variable $x_2$ (equals to 2 in the example). The second term is $C_{k-p}^q$, where $q$ is the degree of second variable $x_1$ (equals 2 in the example). The procedure continues until it reaches the last variable. Since each variable degree is known from the previous sequence in advance, it becomes easy to compute.

### 4.2.4 Other Discussion

Above we assume that the bits number is lower than the Taylor degree; if not, etc., $N>k$, the circumstance would be more complicated. For instance $N$=4 and $k$=2,

$$
\begin{aligned}
(\sum_{i=0}^{3} 2^i x_i)^2 &= x_0^2 + (2x_1)^2 + C_2^1(2x_1)x_0 + (4x_2)^2 + C_2^1(4x_2)x_0 \\
&+ C_2^1(4x_2)(2x_1) + (8x_3)^2 + C_2^1(8x_3)x_0 + C_2^1(8x_3)(2x_1) + C_2^1(8x_3)(4x_2)
\end{aligned}
$$

There are no terms with 3 and 4 variables, so the algorithm only needs a little amendment — terms which have the variable number beyond the Taylor degree would be neglected. In this example, the neglected terms are $x_2x_1x_0$, $x_3x_2x_1$, $x_3x_2x_0$, $x_3x_1x_0$ and $x_3x_2x_1x_0$.

Integrating these two cases, Property 4.3 counts how many AT terms from Taylor conversion.

**Property 4.3:** *The number of AT terms is determined by the bits number N and the highest Taylor degree k. If N<k, the terms number equals $2^N$-1; if not, it is $\sum_{g=1}^{k} C_N^g$. Please note if the constant f($X_0$) is not zero in Taylor series, the number needs to add 1.*

The situation of $X_0$=0 in Taylor Series has been elaborated. $X_0$ must not be 0 at some functions such as log($X$) and $(1/X)^n$. $Y$ replaces $X_0$ to avoid confusion with the binary bit $x_0$ to explore it.

***Example 4.2:*** *Given three input bits and Taylor degree k=3, Y is not zero value, the expansion is:*

$$
\begin{aligned}
(\sum_{i=0}^{2} 2^i x_i - Y)^3 &= -Y^3 + x_0^3 - C_3^2 x_0^2 Y + C_3^1 x_0 Y^2 + (2x_1)^3 - C_3^2(2x_1)^2 Y + C_3^1(2x_1)Y^2 + C_3^2(2x_1)^2 x_0 \\
&+ C_3^1(2x_1)x_0^2 - C_3^1C_2^1(2x_1)x_0Y + (4x_2)^3 - C_3^2(4x_2)^2Y + C_3^1(4x_2)Y^2 + C_3^2(4x_2)^2x_0 + C_3^1(4x_2)x_0^2 \\
&- C_3^1C_2^1(4x_2)x_0Y + C_3^2(4x_2)^2(2x_1) + C_3^1(4x_2)(2x_1)^2 - C_3^1C_2^1(4x_2)(2x_1)Y + C_3^1C_2^1(4x_2)(2x_1)x_0
\end{aligned}
$$

$Y$ is regarded as a variable and expanded in terms of Equation (4-2) although it is a constant in fact. $x_0^3$, $C_3^1x_0Y^2$ and $-C_3^2x_0^2Y$ represent the same AT term $x_0$ thus they should be combined. The difference in comparison with a true variable (not a constant) is that its exponent can be permitted to set "0" whereas an ordinary bit-level variable has a smallest exponent "1" in terms of Property 4.2. Therefore, the algorithm needs to be revised: if $Y$ is not 0, let the exponent of $Y$ change from 0 to the largest to get weights of expanded terms. For example, $Y$ changes its degree from 0 to 2 in the AT term $x_0$ and from 0 to 1 in the term $x_2x_1$.

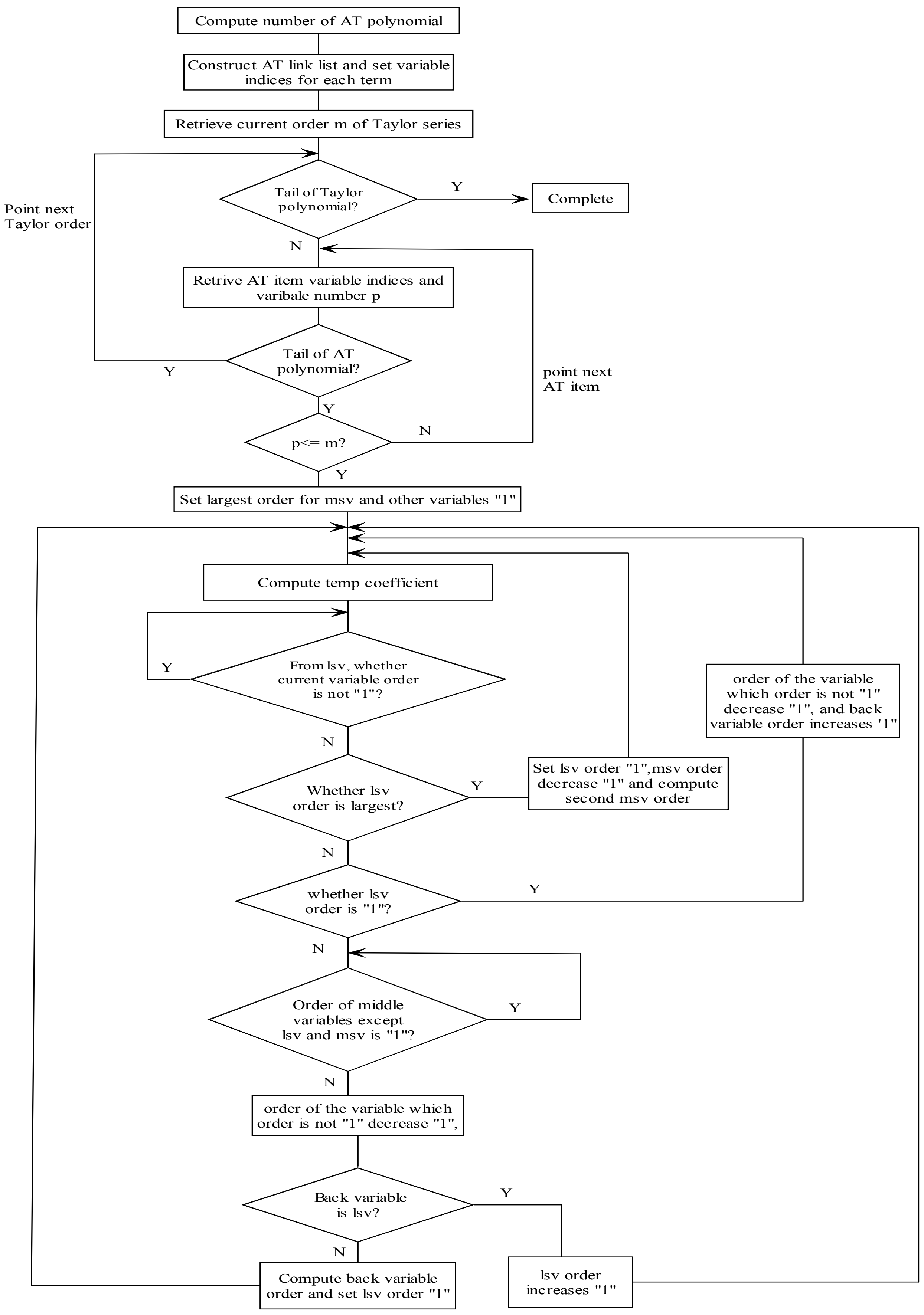

Compute number of AT polynomial
Construct AT link list and set variable indices for each term
Retrieve current order m of Taylor series
Tail of Taylor polynomial?
Y
Complete
N
Point next Taylor order
Retrive AT item variable indices and varibale number p
Tail of AT polynomial?
Y
point next AT item
Y
p<= m?
N
Y
Set largest order for msv and other variables "1"
Compute temp coefficient
Y
From lsv, whether current variable order is not "1"?
N
order of the variable which order is not "1" decrease "1", and back variable order increases '1"
Whether lsv order is largest?
Y
Set lsv order "1",msv order decrease "1" and compute second msv order
N
whether lsv order is "1"?
Y
N
Order of middle variables except lsv and msv is "1"?
Y
N
order of the variable which order is not "1" decrease "1",
Back variable is lsv?
Y
N
Compute back variable order and set lsv order "1"
lsv order increases "1"

**Figure 4.1: Algorithm of converting Taylor series to AT**

### 4.2.5 Flow of Conversion Algorithm

Equation (4-2) establishes the algorithm foundation. The algorithm first computes how many AT terms will be according to Property 4.3 and creates an AT linked list to allocate their variable indices, then commences a main loop. Within each loop procedure, the algorithm retrieves a Taylor degree from Taylor series and starts an inner loop to point the AT link list, which indicates the first AT term at beginning. Based on the retrieved Taylor degree, isomorphic forms and their weights for the indicated AT term are fast computed due to Property 4.1 and 4.2, the weights addition is a temporary coefficient for the AT term under the specific Taylor degree. While the pointer has moved to the last AT term, a new procedure of the main loop occurs to retrieve a next Taylor degree and the pointer resets to the first AT term. When the algorithm finishes the main loop, AT coefficients can be obtained eventually by summation of all corresponding temporary coefficients. Figure 4.1 outlines the algorithm in detail. We observe that the algorithm does not generate any intermediate polynomials to store expanded terms explicitly, therefore, the algorithm avoids expending huge memory and running time.

## 4.3 Processing Multivariate Polynomials

The conversion of Taylor series to AT has been solved above. However, Taylor series only comprises one word-level variable – work in [84] gave examples for verification and the limitation was similar to Taylor series, that is, the benchmarks only consisted of one word-level variable. This case restricts further applications since many circuits are represented by polynomials included beyond one word-level variable or mixed with bit-level variables such as a multiplexer. Emergence of the fast more realistic conversion algorithm above makes it possible to conquer the problem for cases. In addition, a significant advantage is polynomial data structures are often represented by decision diagrams like BMDs and TEDs, which stand for bit- and word-level variables, respectively. These diagrams can be transferred to

ATs easily, therefore a bridge is generated between decision diagrams and the imprecision model to overcome their weakness to do component matching. The conversion algorithm mentioned above is unable to process the more difficult case. The algorithm is revised to deal with several word-level variables to overcome this limitation.

For an AT term, we define its *index*, which is unique for each term. The index will facilitate the combination of isomorphic terms in an intermediate polynomial.

**Definition 4.2:** *Let the term consist of p bit-level literals $b_{p-1} \ldots b_0$. Let every bit $b_r$ belong to the word-level variable $W_r$, that is $m_r$-bit wide. Then, the term index of the AT term is defined as:*

$$\text{term.index} = \sum_{r=0}^{p-1} 2^{\left(b_r + \sum_{q=0}^{Wr-1} m_q\right)} \tag{4-3}$$

***Example 4.3:*** *Consider AT over three word-level variables X, Y and Z consisting of 3, 4 and 3 bits, respectively. Let X be the least significant variable indexed as "0", and Z be the most significant variables indexed as "2". For the three bit-level literal term $z_2z_1x_0$, the word-level variables to which the respective literals belong, are ($W_2$, $W_1$, $W_0$) = (2, 2, 0). The index of the term is obtained as the sum of the three literal indices. First, the computation for $x_0$ produces its index $2^0=1$, since $b_0$ is 0 and $W_0$ is 0. Then, $z_1$ contributes $2^{1+(3+4)}=256$, since $b_1$ is 1 and $W_1$ is 2, so $m_0+m_1 = 3+4=7$. Finally, $z_2$ produces $2^{2+(3+4)}=512$, because $b_2$ is 2 and $W_2$ is 2. Therefore, the term index for the AT term $z_2z_1x_0$ is 512+256+1=769.*

It is evident that this case incurs more complexity. Figure 4.2 describes the algorithm to produce AT over multiple word-level variables from a real-valued polynomial. The algorithm first generates AT for each monomial, and then performs additions of the isomorphic intermediate monomials, leading to the final transform. The function `Expand_Term` expands a single word-level polynomial term into its AT. The subroutine `Convert_Univar_AT` introduced in Figure 4.1 obtains ATs for all word-level variables in the term.

Then, the subroutine `Multiply_AT` multiplies the resulting univariate AT into the multivariate AT. Note that `Multiply_AT` follows the conversion of a word-level variable that reduces the number of terms. Hence, the size of resulting AT can be kept under control by avoiding storing expanded terms. In each iteration, the algorithm adjusts term indices and combines isomorphic terms. Each AT term input to the `Multiply_AT` is assigned a unique index from Definition 4.2, which guarantees linear ordering among terms.

```
Convert_Multivar_AT(f, term_num, bit_num )
{    for (i=0; i< term_num; i++)
   {    temp_AT = Expand _Term (bit_num);
        sum_AT = Add_AT (sum_AT, temp_AT);      }
     final_AT = sum_AT;          return final_AT;
}
Expand _Term (bit_num)
{    for (p=0; p<word_var_num; p++)
   {    AT_poly[p]=Convert_Univar_AT (f, term_num, bit_num);
        product_AT= Multiply_AT(AT_poly[p], AT_poly[p-1]); }
     Set_index (product_AT);      return product_AT;
}
Add_AT (augend_AT, addend_AT)
{      While (!augend_AT.tail && !addend_AT.tail( ) )
       {    if ( augend_AT.term.index < addend_AT.term.index)
                Copy_AT_term (sum_AT.term, augend_AT.term);
         else if (augend_AT.term.index> addend_AT.term.index )
                 Copy_AT_term (sum_AT.term, addend_AT.term);
         else {    Copy_AT_term(sum_AT.term, augend_AT.term);
                   sum_AT.term.coeff = augend_AT.term.coeff + addend_AT.term.coeff; }
       }
     Delete (augend_AT, addend_AT);    return sum_AT;    }
}
Multiply_AT (multiplicand_AT, multiplicator_AT)
{    while (!multiplicand_AT.tail)
     {    while (!multiplicator_AT.tail)
        {    product_AT.term.coeff = multiplicand_AT.term.coeff
                                   * multiplicator_AT.term.coeff;
             for (p=0; p<cand_bit_num; p++)
                 product_index[p] = cand_index[p];
            for (p=cand_bit_num; p<product_bit_num; p++)
                   product_index[p]=cator_index[p-cand_bit_num]; }
     }
     return product_AT; }
```

**Figure 4.2: Algorithm for converting a multivariate polynomial**

The function `Add_AT` adds two AT polynomials in a canonical way. In this procedure, the isomorphic term combination and the term ordering by index occur concurrently. When comparing indices of terms, the AT term with a smaller index is moved forward in the ordered list. If two terms have identical indices, they are isomorphic, and hence their coefficients are accumulated.

***Example 4.4:*** *Consider a polynomial that has two word-level variables consisting of (2, 3) bits.*

$$F(X, Y) = 2X^3Y + X^2Y^2$$

*This polynomial has two terms. The algorithm loops them and expands them to two AT polynomials. In the first term $2X^3Y$, expansions of $X^3$ and $Y$ are:*

$$AT(X^3) = (2x_1+x_0)^3 = x_0 + 8x_1 + 18x_1x_0 \qquad AT(Y) = 4y_2 + 2y_1 + y_0$$

*This term transform is multiplied by the two sub-AT polynomials:*

$$AT(2X^3Y) = 2y_0x_0 + 16y_0x_1 + 36y_0x_1x_0 + 4y_1x_0 + 32y_1x_1 + 72y_1x_1x_0 + 8y_2x_0 + 64y_2x_1 + 144y_2x_1x_0$$

*The individual AT term index is: (5, 6, 7, 9, 10, 11, 17, 18, 19)*

*In the second term, expansions of $X^2$ and $Y^2$ are:*

$$AT(X^2) = x_0 + 4x_1 + 4x_1x_0 \qquad AT(Y^2) = y_0 + 4y_1 + 4y_1y_0 + 16y_2 + 8y_2y_0 + 16y_2y_1$$

*Their multiplication is the transform of $X^2Y^2$:*

$$AT(X^2Y^2) = y_0x_0 + 4y_0x_1 + 4y_0x_1x_0 + 4y_1x_0 + 16y_1x_1 + 16y_1x_1x_0 + 4y_1y_0x_0 + 16y_1y_0x_1 + 16y_1y_0x_1x_0 + 16y_2x_0 + 64y_2x_1 + 64y_2x_1x_0 + 8y_2y_0x_0 + 32y_2y_0x_1 + 32y_2y_0x_1x_0 + 16y_2y_1x_0 + 64y_2y_1x_1 + 64y_2y_1x_1x_0$$

*Its index is: (5, 6, 7, 9, 10, 11, 13, 14, 15, 17, 18, 19, 21, 22, 23, 25, 26, 27 )*

*The addition subroutine is invoked to compute the transform of $2X^3Y + X^2Y^2$ in terms of their indices:*

$$AT(2X^3Y + X^2Y^2) = 3y_0x_0 + 20y_0x_1 + 40y_0x_1x_0 + 8y_1x_0 + 48y_1x_1 + 88y_1x_1x_0 + 16y_1y_0x_1 + 16y_1y_0x_1x_0 + 24y_2x_0 + 128y_2x_1 + 208y_2x_1x_0 + 8y_2y_0x_0 + 32y_2y_0x_1 + 32y_2y_0x_1x_0 + 16y_2y_1x_0 + 64y_2y_1x_1 + 64y_2y_1x_1x_0$$

Because polynomial multiplications described as the subroutine `Multiply_AT` take place after conversion of a word-level variable, the result AT size can be controlled and avoid storing expanded terms, also in each loop procedure the algorithm adjusts terms position and combines isomorphic terms, and releases memory in time, so it disperses computation time then reduces total complexity. Therefore, the algorithm keeps good performance even though there are a number of word-level variables.

## 4.4 Imprecision Searching Algorithm

Saving costs and speeding up a design are so important to engineers, whenever available, they benefit from reusing a previously designed module. However, these modules usually do not match specifications so they are only approximations. If discrepancy (imprecision) is within an acceptable boundary, it could be chosen. The approximations come from various aspects and this paper concentrates on restrict input space and finite realization of Taylor series. Therefore, a good solution to find difference between specifications and implementations is significant.

A static method for range and precision analysis was used in [43], where circuits described by Verilog were assessed for FPGA implementations. This solution did not provide a uniform platform and it depended on tools of simulation annealing which are often inefficient. In this paper we explore the suitability of Arithmetic Transform in the representation of the imprecise blocks and make up their deficiency.

### 4.4.1. Basic Definitions of the Algorithm

Related definitions are introduced to describe the imprecision searching algorithm comprehensibly.

A straightforward approach tries every input value to compute its error AT. The procedure requires $2^N$ calculation because of total $2^N$ possible inputs. Experiments indicate that such an approach would require an infeasible amount of time, and therefore a fast algorithm is necessary. In this work we propose such an improved algorithm.

For each input variable $x_i$, we say that $S_i$ is a sum of coefficients multiplying terms with $x_i$. The *most positive variable (mpv)* is the variable $x_j$ where the sum $S_j$ is largest. An *upper bound ubcoef* of AT polynomial is by summing all coefficients that are positive and the coefficient $c_{00\ldots00}$ that contributes an offset for all input assignments. Such a bound is calculated as:

$$ub_{coef} = \sum_{c>0} c_{i_1} c_{i_2} .. c_{i_n} + c_{00\ldots00}$$

The algorithm checks whether there are the input assignments to be made without the search to avoid calling the main search loop unnecessarily. Such a preprocessing step is used at each call of the search routine.

a) Assign $x_i$ =1 if coefficients of the AT monomials with $x_i$ present are all positive (or zero).

b) Assign $x_i$ =0 if coefficients of the AT monomials with $x_i$ present are all negative (or zero).

## 4.4.2. Branch-and-Bound Searching Algorithm

```
Search_max (AT_poly)
{ const = Remove_constant(AT_poly);
  var_index = Mpv(AT_poly);
  rev_AT_poly = Reverse(AT_poly);          rev_var_index = Mpv(rev_AT_poly);
  value_0 = Decompose(AT_poly, var_index);
  value_1 = Decompose(AT_poly, rev_var_index);
  value_2 = Decompose(rev_AT_poly, var_index);
  value_3 = Decompose(rev_AT_poly, rev_var_index);
    max_value = Max(value_0, value_1);    |min_value| = Max(value_2, value_3);
    mismatch = Max ( |max_value+const|, |min_value+const|; }
Decompose(AT_poly, mpv)
{   for (i=0; i<var_num; i++)
    {   flag = Preprocess(AT_poly, mpv[i]);
        if (flag = 1)
        {   $AT_1 = AT(f)_{x_i=1}$ , ub_1 = Ub($AT_1$);
            $AT_0 = AT(f)_{x_i=0}$ ,  ub_0 = Ub($AT_0$);
            if (ub_1> ub_0)      $AT = AT_1$;
            else      $AT = AT_0$;
        }
        Delete_var (mpv[i]);       var_num--;
        for (i=0; i<var_num; i++)
        {   flag = Preprocess(AT_poly, mpv[i]);
            if (flag = 0)
              Delete_var (mpv[i]);      var_num--;
        }
    }
}
Preprocess (AT_poly, $x_i$)
{   if (all  $c_{x_i}$ > 0)    val = 1;
  else if (all  $c_{x_i}$ < 0)    val = 0;    else    return 1;
  $AT = AT(f)_{x_i=val}$;      return 0;
}
```

**Figure 4.3: Searching the maximum absolute value in AT**

The algorithm first removes the constant in the polynomial if it exists, and gets the mpv sequence as the order of decomposition variables, and then the reversed AT polynomial and the reversed mpv sequence are obtained easily.

A subroutine `Decompose` is invoked to compute the maximum value and the minimum value due to the two AT polynomials and two sequences. The preprocessing step deals with a variable to explore whether it can be evaluated directly by probing into its coefficients; if not, the algorithm chooses a path which has a larger upper bound. Figure 4.3 describes the branch searching algorithm in detail.

***Example 4.5:*** *Consider the following AT polynomial:*

$$AT(f) = -2 + x_0 - 3x_1x_0 + 3x_2 + 3x_2x_1 - 4x_3x_1 - 2x_3x_2x_0 + 5x_3x_2x_1$$

*Figure 4.4 illustrates all the steps taken to compute the maximum absolute value. First remove the constant and get a new AT polynomial:*

$AT(f)' = x_0 - 3x_1x_0 + 3x_2 + 3x_2x_1 - 4x_3x_1 - 2x_3x_2x_0 + 5x_3x_2x_1$

$S_0=-4$, $S_1=1$, $S_2=9$, $S_3=-1$, *so the mpv sequence is* $(x_2, x_1, x_3, x_0)$. *The reversed polynomial is:* $AT(f)'' = -x_0 + 3x_1x_0 - 3x_2 - 3x_2x_1 + 4x_3x_1 + 2x_3x_2x_0 - 5x_3x_2x_1$

*The reversed mpv sequence is* $(x_0, x_3, x_1, x_2)$.

*First* $AT(f)'$ *is searched by the order of the mpv sequence, due to the ubcoef value,* $x_2$ *and* $x_1$ *are set to 1, here the decomposed polynomial is* $3 - x_0 + 4x_1 - 3x_1x_0$*, then the algorithm finds coefficients of all terms with variable* $x_0$ *present are negative, so* $x_0$ *is preprocessed to 0; and it continues to preprocess* $x_3 = 1$*, finally a constant value_0 = 7 is obtained; the procedure is displayed by a) in Figure 4.4. Using the reversed mpv sequence upon* $AT(f)'$*, the obtained constant is value_1 = 3, showed by b), so the maximum value of the AT polynomial without the constant "-2" is:*

$$max\_value = max\ (value\_0, value\_1) = 7.$$

*Decompose* $AT(f)''$ *by the mpv and the reversed mpv sequences respectively, showed by c) and d), value_2 = value_3 = 6, so the minimum value of the AT polynomial without the constant "-2" is:*

$$min\_value = max\ (value\_0, value\_1) * -1 = -6.$$

*Eventually the maximum mismatch is computed as:*

$$\max(|\max\_\,value - 2|, |\min\_\,value - 2|) = \max(|7-2|, |-6-2|) = 8$$

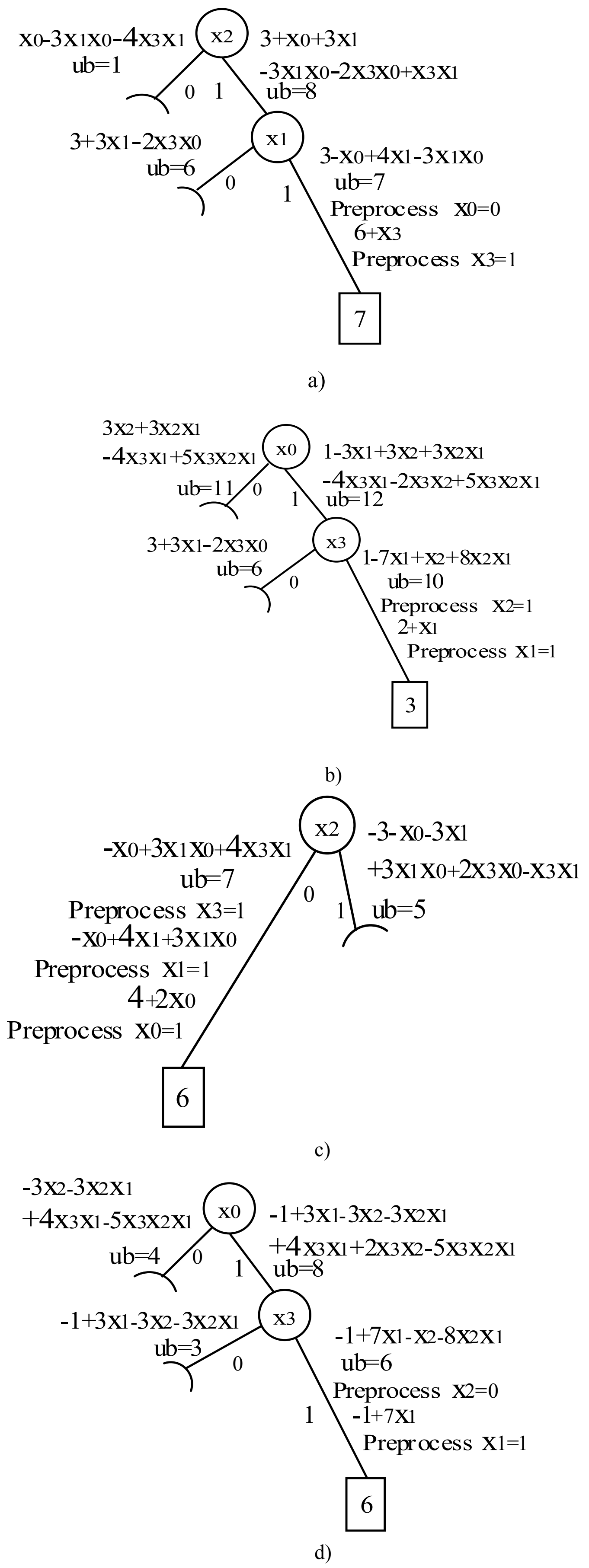


**Figure 4.4: Performing the imprecision algorithm in Example 4.5**

Compared to the searching algorithm in [70] and [85], the predominance of the algorithm improvement stands to reason. It recursively seeks the variables which can be preprocessed in a decomposition procedure. If successful, complexity is minified much since the computation avoids decomposing the variable and directly sets its value, and then the residual polynomial is simplified. For example, only one node, $x_2$, is searched to determine its value in c), and other three variables are preprocessed, therefore time and space requirements are diminished.

## 4.5 Experimental Results

The conversion algorithm is a basic algorithm for verification and optimization of imprecise circuits because of its huge impact on performance. Here we mainly aim the benchmarks of Taylor series. All experiments are done on an Intel Celeron 2.4GHz CPU with 1G main memory under Linux.

$X_0 = 0$

| Function | Taylor degree | Bits | AT terms | Expanded terms | Run time (s) | Memory (MB) |
|---|---|---|---|---|---|---|
| sin(x) | 7 | 31 | 3572223 | 10625591 | 586.593 | 156 |
| sin(x) | 9 | 26 | 5658536 | 55962920 | 179.171 | 247 |
| sin(x) | 11 | 24 | 7036529 | 316283264 | 921.218 | 293 |
| sin(x) | 13 | 20 | 988115 | 409609664 | 1167.58 | 59 |
| exp(x) | 10 | 24 | 4540386 | 131128139 | 371.266 | 239 |
| exp(x) | 12 | 22 | 3096514 | 548354039 | 1633.36 | 182 |
| exp(x) | 14 | 18 | 261156 | 471435599 | 1497.81 | 3 |
| exp(x) | 14 | 20 | 1026876 | 1391975639 | 4222.25 | 59 |
| exp(x)*sin(x) | 10 | 24 | 4540385 | 123221864 | 314.703 | 254 |
| exp(x)*sin(x) | 13 | 20 | 988115 | 429816984 | 1445.19 | 88 |
| exp(x)*sin(x) | 15 | 16 | 65534 | 282662144 | 985.703 | 18 |

*$X_0$ = 0.5*

| Function | Taylor degree | Bits | AT terms | Expanded terms | Run time (s) | Memory (MB) |
|---|---|---|---|---|---|---|
| sin(x) | 7 | 31 | 3572224 | 13002888 | 873.437 | 163 |
| sin(x) | 9 | 24 | 2579130 | 41317895 | 158.125 | 159 |
| sin(x) | 11 | 20 | 784626 | 95629666 | 269.093 | 43 |
| sin(x) | 13 | 20 | 988116 | 668795865 | 2286.89 | 49 |
| exp(x) | 10 | 24 | 4540386 | 183578305 | 509.89 | 156 |
| exp(x)*sin(x) | 10 | 24 | 4540386 | 173039772 | 625.171 | 150 |

**Table 4.1: Performance of Taylor series conversion**

Table 4.1 shows results of the algorithm described by Figure 4.1. The two sub-tables correspond to “0” and “0.5” values of $X_0$ respectively. Column 2 and 3 list the highest degree and input bits. Column 4 and 5 show final AT terms and expanded isomorphic terms.

From the table, the conversion algorithm is feasible even though Taylor degree and input variables are very large. The performance of time and space are satisfied, and the AT terms only occupy around 5% - 20% of isomorphic terms. So combining these terms to form AT terms will spend huge processing time, but the algorithm can handle it easily. During experiments, we find this algorithm has been always the fastest algorithm.

## 4.6 Conclusions

Taylor series is a typical imprecise representation with function approximation and finite wordlengths, so it is our main research object that we adopt AT. In order to utilize AT technique, we propose several algorithms which can convert Taylor series to AT and search for its maximum absolute value. These algorithms can handle not only Taylor series but also real-valued polynomials with multiple variables, and are fundamental to the future verification and optimization, so they can cover a majority of applications.

# Chapter 5 Analysis of Precision Parameters

*Arithmetic circuits such as these realizing Taylor series-based algorithms incorporate many generalizations leading to imprecision. In order to design and verify imprecise circuits, the first step is to analyze these factors carefully. Traditional methods have difficulty to represent the factors mathematically. In this chapter we describe the imprecise arithmetic computations, and then utilize AT to analyze imprecise parameters in a polynomial, and estimate how much error is caused by each parameter.*

## 5.1 Imprecise Arithmetic Computations

Major causes of imprecision in an implementation come from two aspects. One is the approximations of the specifications in hardware realization and the other is using finite wordlength to represent an infinite length of specification data. For example, real fractional numbers are usually realized by finite size registers which are regarded as fixed-point data representations. *Radecka* and *Zilic* [70] introduced the fundamental idea based on AT representations.

**Definition 5.1:** *The error is a numerical difference between the results required by the specification and the quantity obtained in the implementation. The* **unit in the last place (ULP)** *used to evaluate the error is the least significant bit for binary encoding of a given number.*

The function approximation is an inexact implementation regardless of the precision while the precision is the total bit number used to represent the fixed-point circuit. Although there might be some other causes of imprecision in ASIC implementations, the above two reasons are the focal points in this work.

### 5.1.1 Finite Wordlength

Using finite precision to represent infinite length real numbers is performed by truncation and rounding. Output bit-width is always restricted so it is unavoidable to cause imprecision. The following example explores data truncation and rounding.

***Example 5.1:*** *A circuit has four N-bit unsigned fractional inputs: "a", "b", "c" and "d" to perform the operation ab+cd. The output result has 2N-1 bits :*

$$AT(ab+cd)=\sum_{k=1}^{N}a_k 2^{-k} * \sum_{k=1}^{N}b_k 2^{-k} + \sum_{k=1}^{N}c_k 2^{-k} * \sum_{k=1}^{N}d_k 2^{-k}$$

*If the result of the implementation is restricted to N most significant bits of the original expression, two cases would be considered:*

*a) Rounding to the nearest value causes the error bounded to half of the ULP, i.e., $2^{-(N+1)}$.*

*b) When truncating to "N" bits, the error is bounded by one ULP, which is $2^{-N}$.*

Explicit representation of output values is required for the precision verification because the precision on a per-bit basis is not reasonable. A simple example can describe the situation that even though all output bits are incorrect, the imprecision is arbitrarily small. For instance, if the exact *N*-bit result is 100….0, and the approximation is 011…1, then all bits are incorrect; the error is one ULP, however, which for large *N* becomes negligible.

## 5.1.2 Arithmetic Transforms and Imprecise Datapaths

AT has a property of linearity which can be directly applicable to verification of imprecise circuits. The transform of an imprecise circuit, i.e, *IAT(f)*, can be represented as a linear superposition of the specified AT form *SpecAT(f)* and the error *e*. Generally, error accumulation makes that various errors throughout the circuit can be observed at outputs and expressed by the error *e* and fault-free AT representation of *SpecAT(f)*:

$$SpecAT(f) = IAT(f) + ErrAT(f) \tag{5-1}$$

The error AT polynomial (*ErrAT*) is determined by a series of imprecision sources, which may be caused by function approximations, or size restrictions of intermediate data of an implementation.

**Definition 5.2**: "*The AT error polynomial (**ErrAT**) is a difference polynomial between Arithmetic Transforms of specification (**SpecAT**) and its corresponding implementation (**IAT**)" [70].*

***Example 5.2:** A circuit calculates the product a*b with 8-bit for each variable and disregards all partial products needed for obtaining 8 least significant bits. This approximation will save half the circuit area, but causing the AT error*:

$$ErrAT(f) = SpecAT(a*b) - IAT(a*b) = \sum_{i=1}^{8} a_i \sum_{j=1}^{8} b_i 2^{-i-j} - \sum_{i=2}^{9} \sum_{j=1}^{i-1} a_{i-j} b_i 2^{-i}$$

$$= \sum_{j=2}^{8} \sum_{i=6-j}^{8} a_i b_i 2^{-i-j}$$

*After summation, we obtain that the worst case error is bounded by $(6*2^8+2)/2^{11}$, which is $O(2^{-6})$.*

Since AT has the linear property, if a module within a circuit has an error, this error can be peeled off from the transform of the module, the following equation describes it:

$$AT(f+e) = AT(f) + AT(e) \tag{5-2}$$

The arithmetic transform of the erroneous module equals the addition of the transform of the good module and the error transform. The property makes it easy to analyze the effect caused by errors.

Once the overall AT is constructed for an imprecise circuit, the maximum allowable value of an error polynomial (*ErrAT*) can be determined. When an input/output size of an implementation differs from that of specification, the precision of the implementation, expressed in terms of acceptable error bounds is a required parameter. Only then we can state that the implementation (*IAT*) is in agreement with the specification (*SpecAT*) within a precision error bound $\varepsilon$. In consequence, the maximum absolute value of *ErrAT* must accord with the inequality [69]:

$$\max(ErrAT) = \max\left|SpecAT - IAT\right| \leq \varepsilon \tag{5-3}$$

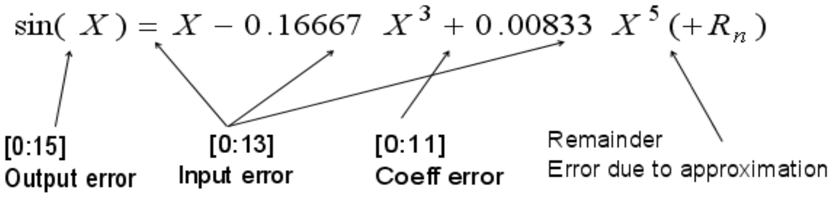
The maximum absolute error can be calculated by the branch-and-bound searching algorithm introduced in Chapter 4. If *SpecAT* is imprecise itself and represents a function $f$ up to an absolute precision of $\delta$, the following inequality [69] holds:

$$\begin{aligned}&\max\ |IAT(X) - AT(f(X))|\\ &\leq \max\left|IAT(X) - SpecAT(X)\right| + \max\ |SpecAT(X) - AT(f(X))| \leq \varepsilon + \delta\end{aligned} \tag{5-4}$$

While the value $\delta$ is known, Eqn. 5-4 can be used to verify the imprecision between *SpecAT* and *IAT*.

## 5.2 Function Approximation Error

Determining the set of parameters needed to achieve a circuit of the allowed imprecision is a challenge that is in part due to the difficulties with the precision analysis. The traditional method of using simulations over various values of the parameters is costly and not guaranteed to produce the optimal result. We next analyze the arithmetic precision parameters due to all approximations and finite bit widths in the implementations of real-valued specifications such as Taylor series in Figure 5.1. In summing the imprecision, we will repeatedly use the triangle inequality.



$\sin(X) = X - 0.16667\ X^3 + 0.00833\ X^5 (+R_n)$
[0:15]
Output error
[0:13]
Input error
[0:11]
Coeff error
Remainder
Error due to approximation

**Figure 5.1: Imprecision due to the combined sources**

In implementing real-valued functions by arithmetic circuits, an algorithm might be employed to *approximate*, rather than exactly implement the function. For instance, when using *n* Taylor terms to represent a transcendental function, the approximation error is provably bounded by a remainder $R_n(X)$, Eqn. (4-1). Hence, for a function given in interval *I*, this truncation error bound $e_t$ is:

$$e_t = \max_{X \in I} |R_n(X)| \tag{5-5}$$

***Example 5.3:*** *Consider the following function f(X) = cos(X). In interval [-1, 1], its Taylor approximation around* $X_0$=0 *with 3 terms is:*

$$Taylor\ (\cos X) = 1 - \frac{1}{2}X^2 + \frac{1}{24}X^4,$$

*Now we can estimate its error bound:*

$$e_t = \max\ |R_5(X)| = \max\ |\frac{1}{120}\sin\xi * X^5| \le |\frac{1}{120}\sin 1| = 0.007$$

Given the desired error bound *E*, it is easy to find the appropriate number of Taylor terms *n* as a largest integer for $e_t < E$. Such a finite truncation of Taylor series will have the least number of terms that result in an acceptable imprecision over the given interval *I*. Please note from Eqn. (4-1) that instead of finding the exact maximum of the *(n+1)*st derivative on *I*, using an upper bound might suffice.

## 5.3 Input Bit-width and Quantization Error

In fixed-point implementations, a bit vector represents the real-valued input variable *X*, so the input quantization due to finite bit-width affects the final result. An insufficiently precise result can be caused by using too few bits, and we hence try to find an appropriate bit-width resulting in the acceptable overall error.

### 5.3.1 Effect of Finite Input Bit-width – Interval Analysis

An argument of a real-valued function is potentially infinitely precise. Such a theoretical value $X_{th}$ is instead replaced by the quantized input value *X* in function calculation. The classical interval analysis [26]-[31] is expressed in terms of AT as follows. Let *FB* represent Fractional Bits. The input range is divided into uniform $2^{FB}$ intervals, so the difference between two consecutive intervals is $2^{-FB}$. The point representing $X_{th}$ is between two quantized values, as in Figure 5.2. The relationship between $X_{th}$ and *X* is then:

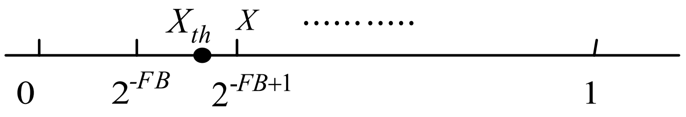


**Figure 5.2: Value description of $X_{th}$ and X**

$$|X_{th} - X| \le 2^{-(FB+1)} \quad \Rightarrow \quad X - 2^{-(FB+1)} \le X_{th} \le X + 2^{-(FB+1)} \tag{5-6}$$

Hence, by replacing $X_{th}$ by *m* fractional bits of *X* in accordance with Eqn. (5-6), we get the expressions for the theoretical $f_{th}$ and quantized *f* function values (given $X_0 = 0$):

$$f_{th} = \sum_{i=0}^{n} \frac{f^{i}(X_0)}{i!}[(\sum_{k=0}^{m-1} 2^{-(k+1)} x_k) \pm 2^{-m-1}]^{i}$$

$$= \sum_{i=0}^{n} C_i [(\sum_{k=0}^{m-1} 2^{-(k+1)} x_k) \pm 2^{-m-1}]^i \quad (5\text{-}7)$$

$$f = \sum_{i=0}^{n} C_i (\sum_{k=0}^{m-1} 2^{-(k+1)} x_k)^i \quad (5\text{-}8)$$

where $C_i$ is a Taylor coefficient that equals to $\frac{f^i(X_0)}{i!}$.

We represent $f_{\mathrm{th}}$ and $f$ by AT polynomials $AT(f_{\mathrm{th}})$ and $AT(f)$ to efficiently search over binary inputs, obtained from Eqn. (5-7) and (5-8), respectively. The conversion algorithm introduced in Figure 4.1 is designed to deal efficiently with the intermediate terms swell when the number of Taylor series terms and the bit-widths increase. The error polynomial $AT(f_{ei})$ is then a difference between $AT(f_{\mathrm{th}})$ and $AT(f)$:

$$AT(f_{e_i}) = AT(\sum_{i=0}^{n} C_i [(\sum_{k=0}^{m-1} 2^{-(k+1)} x_k) \pm 2^{-m-1}]^i) - AT(\sum_{i=0}^{n} C_i (\sum_{k=0}^{m-1} 2^{-(k+1)} x_k)^i) \quad (5\text{-}9)$$

This AT formulation of the interval analysis assumptions allows us to obtain a bound $e_i$ on the effects of input quantization of half an *ulp* to the output precision. The maximum absolute value of $AT(f_{ei})$ in Eqn. (5-9) gives the error bound $e_i$. While a straightforward approach requires $2^m$ polynomial evaluations, $e_i$ can be obtained by the efficient branch-and-bound searching algorithm tuned for this application.

The interval method is represented by the Eqn. (5-9) which considers the worst case, and applies the algorithms for Taylor conversion and imprecision searching. Figure 5.3 shows the AT usage of interval analysis to estimate error of input quantization.

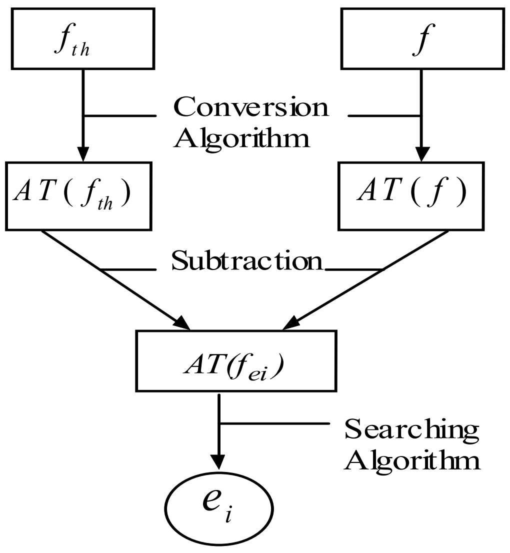


**Figure 5.3: Computation of input quantization error**

## 5.3.2 Tight-bound Interval Scheme

The interval analysis unavoidably overestimates the error bound and gets a coarse result. We now propose a *tight-bound* interval scheme, which employs a more precise specification with larger input bit-width, to obtain tighter error bounds.

For example, assume that *m*=8 bits is used to represent fractional number. Let *f* and $f_{\text{th}}$ represent the quantized function and the theoretical function, respectively. For interval analysis:

$$| f - f_{hp} | = \varepsilon_I = \Theta(2^{-8})$$

We improve precision analysis by the tight-bound method. For this, we use another, finer quantized function representation with, say *t*=17 bits, labeled by $f_{hp}$ and get:

$$| f - f_{hp} | = \varepsilon_{hp}$$

The error in the higher-precision specification alone is estimated by the interval analysis as:

$$| f_{hp} - f_{th} | \le \varepsilon_{I\_TB} = \Theta(2^{-17})$$

From the triangle inequality, it follows that:

$$| f - f_{th} | \le | f - f_{hp} | + | f_{hp} - f_{th} | = \varepsilon_{hp} + \varepsilon_{I\_TB} \qquad (5\text{-}10)$$

In other words, the tight bound analysis uses the exact knowledge of the mismatch to a more precise specification, to which a significantly smaller residual error by interval analysis is added, which allows us to get a tighter error bound.

Please note that the second, larger bit-width function is used here only for analysis purposes, and will not increase the cost. Actually, due to the tighter bounds, the tight-bound interval analysis can lead to a sufficiently precise implementation with less bits used in the implementation. For example, instead of $m$=8, it might suffice to have only bit-width of 7, as the tight-bound comparison with the 17-bit implementation will arrive to the imprecision not worse to that with $m$=8 bits, obtained by the straightforward interval analysis. The scheme for tight-bound interval based on AT technique combines Figure 5.3 and the inequality (5-10) to obtain the suitable bit-width.

## 5.4 Quantization of Coefficients and Output

The finite-word representation of real-valued constants such as coefficients of Taylor expansions causes *coefficient quantization*. If $q$ stands for the coefficient bit-width, then the value of the theoretical (infinite precision) coefficient $C_{th}$ and its word-level representation $C$ are related as follows:

$$C - 2^{-(q+1)} \le C_{th} \le C + 2^{-(q+1)} .$$

Using this inequality to replace $C_{th}$, the expression of $f_{\text{th}}$ becomes:

$$f_{th} = \sum_{i=0}^{n} (c_i \pm 2^{-q-1}) * [(\sum_{k=0}^{m-1} 2^{-(k+1)} x_k)] \qquad (5\text{-}11)$$

The error function $f_{\text{ec}}$ is defined as the difference between $f_{\text{th}}$ and $f$, while the error polynomial $AT_{ec}$ is its transform:

$$AT_{e_c} = AT(\pm 2^{-q-1} \sum_{i=0}^{n} (\sum_{k=0}^{m-1} (2^{-(k+1)} x_k))^i) \qquad (5\text{-}12)$$

The tight-bound analysis can also be applied to explore coefficient bit-widths. The *maximum error* $e_{\text{c}}$ is again computed by the branch searching algorithm combined with the tight-bound scheme over this AT polynomial.

Finally, if the output bit-width is $o$, the bound on the *output quantization* error $e_o$ is $2^{-O-1}$. With $e_t$, $e_i$ and $e_c$ determined, the upper bound of $e_o$ is $e_o = 2^{-O-1} = E - e_t - e_i - e_c$. Hence, $o$ is given as: $o = -\log_2(E - e_t - e_i - e_c) + 1$. Since $e_o$ can be obtained easily and the output bit-width does not affect on internal hardware structure, it is omitted from further considerations below.

## 5.5 Conclusions

Imprecise circuits generally contain many imprecise factors leading to error generation. Here we focus to analyze Taylor series which has four imprecise factors as function approximation, quantization of input bit-width, coefficient bit-width and output bit-width. We use AT and construct mathematical expressions for each factor to facilitate analysis. These expressions are fundamental to future verification and optimization.

# Chapter 6

# Algorithms for Precision Verification and Optimization

*In this chapter, we propose an algorithm to compare two similar, but not exact components. A verification algorithm is then introduced to check whether the implementation satisfies the error bound. A sequential method is designed to find a feasible implementation to satisfy the error bound. In order to single out the best implementations under different constraints, such as area, delay, and fixed bit-width, an optimization algorithm is described. Finally, we integrate these algorithms into a package to handle imprecise circuits.*

## 6.1 Component Comparison

We will now outline our method of finding imprecision between two implementations. If the imprecision between the two components is restricted by the given error bound, they can be substituted by each other, so an investigation of their difference can assist in finding the implementation at a reduced cost. Figure 6.1 illustrates two implementations, *Imp1* and *Imp2*, differing from the specification *Spec* by errors *e1* and *e2* respectively. If errors *e1* and *e2* are within the allowed error bound *E*, then these two implementations are acceptable and can be substituted for each other.

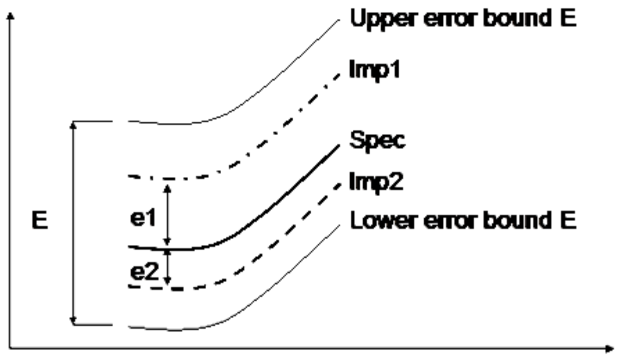


**Figure 6.1: Comparison of two implementations**

The problem description is as follows.

| | |
|---|---|
| ***Problem 6.1***: Computing difference of two implementations | |
| *Inputs*: | $f_1(X)$, $n_1$, $m_1$, $f_2(X)$, $n_2$, $m_2$ |
| *Output*: | *imprecision* |

The error AT polynomial ($AT_e$) introduced in Definition 5.2 is a difference between specified and implemented AT polynomials, and its maximum absolute value is the maximum mismatch which denotes difference between the specification and the implementation.

The use of the algorithm within a realistic tool for comparing precision among two different implementations of real-valued functions is shown in Figure 6.2. The interface file describes two implementations of Taylor series or real-valued polynomials and the bit-widths of corresponding variables. The front-end and the parser hide the details needed to deal with AT. The conversion algorithm converts the two implementations of real-valued functions into two AT polynomials, while the error AT is obtained by subtracting the two polynomials. Then the imprecision is obtained by the searching algorithm introduced in Chapter 4.

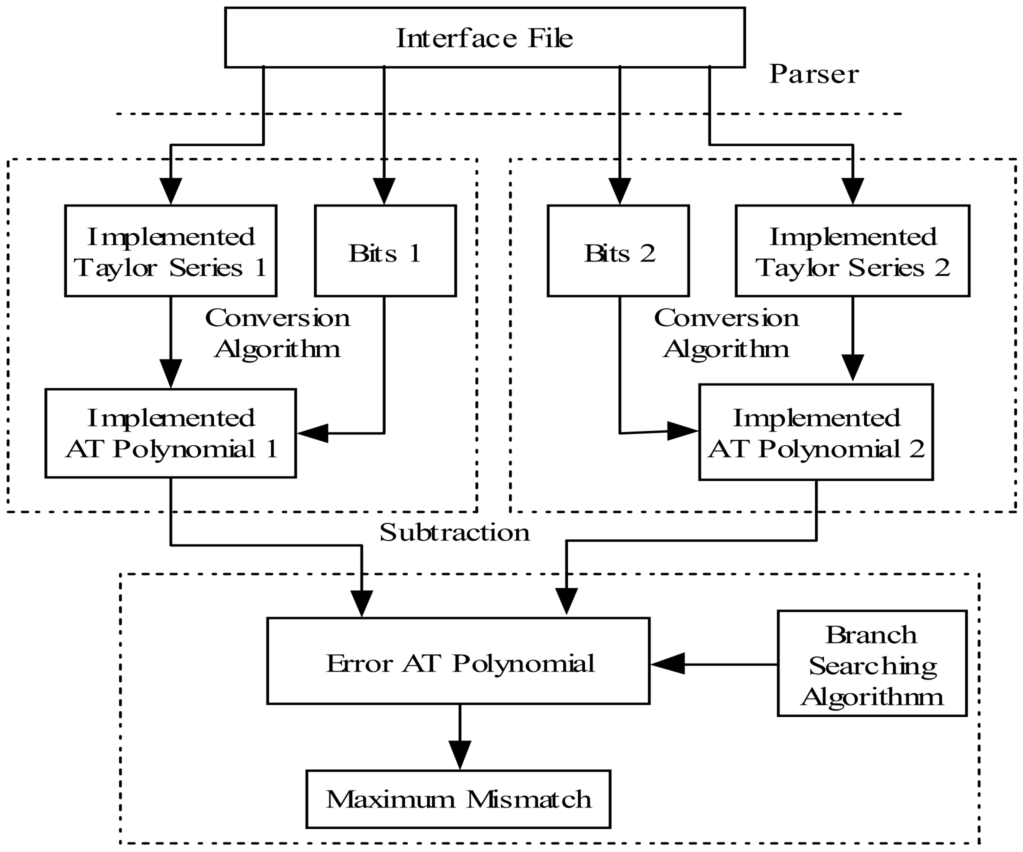

Interface File
Parser
Implemented Taylor Series 1
Bits 1
Conversion Algorithm
Implemented AT Polynomial 1
Bits 2
Implemented Taylor Series 2
Conversion Algorithm
Implemented AT Polynomial 2
Subtraction
Error AT Polynomial
Branch Searching Algorithnm
Maximum Mismatch

**Figure 6.2: Algorithm of computing imprecision between two implementations of Taylor series**

## 6.2 Verification of Implementations

Given an implementation, the imprecision between the specification and the implementation determines whether the implementation can fit the specification, so it becomes necessary to calculate the imprecision coming from the four sources described in Figure 5.1. The problem description is as follows.

***Problem 6.2**: Verifying an implementation*

*Inputs*: $f(X), E, n, m, q, o$

*Judgment*: $e_t + e_i + e_c + e_o < E$

*Outputs*: *Satisfied? (Yes or No)*

The given implementation includes the number of Taylor terms, quantization bits of the inputs, coefficients and output. Calculating the imprecision can be achieved by adding the values of $e_t$, $e_i$, $e_c$ and $e_o$. If the imprecision is beyond the error bound, the implementation does not satisfy the specification. It is helpful to evaluate the validity of the implementation.

```
Check_Imp ($f, E, n, m, q, o$)
1. {  if ($e_t \geq E$)   return false;
2.    $e_i$ = Get_input_error ($f, n, m$);
3.    if ($e_t + e_i \geq E$)   return false;
4.    $e_c$ = Get_coeff_error ($f, n, m, q$);
5.    if ($e_t + e_i + e_c \geq E$)   return false;
6.    $e_o = 2^{-O-1}$;
7.    if ($e_t + e_i + e_c + e_o \geq E$)   return false;
      else   return true;
   }
Get_input_error ($f, n, m$)
{   AT_theoretical = Convert_AT ($f, n, m, 2^{-m-1}$);
    AT_real = Convert_AT ($f, n, m$);
    error_AT = AT_theoretical - AT_real;
    $e_i$ = Search_imprecision (error_AT);
    return $e_i$;
}
Get_coeff_error ($f, n, m, q$)
{   $AT_{ec}$ = Convert_AT ($f, n, m, 2^{-q-1}$);
    $e_c$ = Search_imprecision ($AT_{ec}$);     return $e_c$;
}
```

**Figure 6.3: Algorithm of verifying the implementation**

Figure 6.3 describes an algorithm that checks an implementation by computing each type of error. The result indicates whether the implementation is suitable to the specification through the confirmation of a relationship between the imprecision and the given error bound. The algorithm concurrently investigates function approximation and bit-widths. It handles not only Taylor series but also any real-valued specifications without approximations, and so has wide applications.

## 6.3 Finding a Feasible Implementation

As distinct from the above section, our goal here is to explain how to design a satisfying implementation to restrict the imprecision within the error bound if given a specification represented by Taylor series expanded around *Xo* and the error bound. We now solve the problem of finding a *feasible implementation*, so that the error in the given interval *I* is smaller than *E*.

***Problem 6.3**: Feasible Precision Parameters*

*Inputs*: $f(X), X_0, I, E$

*Constraint*: $imprecision < E, \quad \forall X \in I$

*Outputs*: $n, m, q$

The algorithm in Figure 6.4 applies sequential selection of parameters such that the total imprecision is smaller than *E*. The symbols *n*, *m* and *q* represent the Taylor terms, input bit-width and coefficients bit-width respectively. Since all the error causes can be made arbitrarily small by increasing *n*, *m* or *q*, we can investigate them in any order. As the Taylor approximation error, Eqn. (4-1), is independent of bit-widths, while the errors caused by the bit-widths rely on the exact number of Taylor terms, $e_t$ is investigated first (Step 1), and *n* is selected such that the imprecision due to approximation is smaller than *E*. In Steps 2 and 3, we find input and coefficient bit-widths *m* and *q* using triangle inequality in order to obtain the required precision.

This algorithm always terminates with a feasible implementation, because each of the three steps can determine an arbitrarily small error. Although one can apportion the percentage of *E* for each step, this is potentially wasteful. Since the first source of error is relatively small in comparison to the whole error bound, the distance to *E* will leave room for subsequent quantization values of errors $e_i$ and $e_c$ without needing very long bit-widths *m* and *q*.

```
1. Determine Taylor terms
   {   assume $n$ terms and obtain $e_t$;
       while ($e_t \geq E$)       {   $n$++;   obtain $e_t$;   }
   }
2. Determine input bit-width
   { assume input bit-width $m$;
       for ( )
       {   $AT(f)_{th}$ = Convert_Taylor_AT ($f_{th}, n, m$);
           $AT(f)$ = Convert_Taylor_AT ($f, n, m$);
           $e_i$ = Imprecision_Searching ($AT(f_{th}$ - $AT(f)$));
           if ($e_i \geq E$-$e_t$)     $m$++;     else   break;   }
   }
3. Determine coefficient bit-length
   {   assume bit-width of coefficients $q$;
       while ( )
       {     $AT_{ec}$ = Convert_Taylor_AT($f_{ec}$);
             $e_c$ = Imprecision_Searching ($AT_{ec}$);
             if ($e_c \geq E$-$e_t$- $e_i$)   $q$++;     else   break;   }
   }
```

**Figure 6.4: A sequential method of fitting the error bound**

The method is applicable to Taylor series, but also to any real-valued polynomial specifications. Please note that when some (input or output) bit-widths are fixed because of other modules, those steps are skipped. This scheme achieves a tighter match than the traditional error bounding techniques as its exact searches for the worst-case imprecision account for the interplay between multiple imprecision causes. Although this algorithm can stand on its own, its immediate application is as a pre-selection stage of the full precision optimization algorithm, which is presented next.

## 6.4 Designing Optimized Implementations with Constraints

Although the algorithms in Figure 6.3 and 6.4 compute the precision automatically and indicate whether the implementation is feasible to the error bound, it cannot give information to optimize the implementation. Because the satisfying implementation is not the best one possible in different constraints, it is necessary to develop an algorithm to allow for a flexible distribution of imprecision due to the error sources. In this section, we demonstrate an automated way to find the precision parameters (bit widths, approximation schemes) of the minimum cost determined by constraints.

## 6.4.1 AT Size as a Cost Function

While it is impossible to know precise area data before mapping a circuit by a concrete technology, we do not need to know the exact area as long as the different alternatives can be compared realistically. In our case, the area increases monotonically in both *n* and *m*. More Taylor terms (*n*) require more stages in hardware, which raises inputs to higher exponents. Similarly, longer bit-width (*m*) requires more arithmetic circuitry. As the number of AT polynomial terms |AT(*f*)| exhibits the same tendency, we use it as the cost function to be minimized. The size of AT is obtained by directly expanding the *n*-term Taylor polynomial over *m*-bit input words. One can show that:

$$\left|AT\ (f_{n,m})\right| = \sum_{i=1}^{n} \binom{m}{i} \tag{6-1}$$

## 6.4.2 Error Sensitivity

We recall first that the Taylor series representation comes with a provable bound on the error due to the truncation of the Taylor terms *n*, given by Eqn. (4-1). This bound can be readily used during the precision searches, when different values of *n* are explored. Further, we can readily access the information on error sensitivity due to the input bit width *m*.

Traditionally, sensitivity [21] is defined as Eqn. (6-2) and Figure 6.5 to describe the influence that a small change Δ*X* of *X* has on the output *Y*:

$$\Delta Y \approx f'(X)\Delta X \tag{6-2}$$

where *f'(X)* is the derivative of *f(X)*.

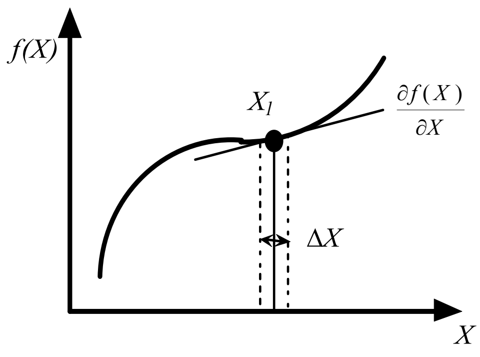


**Figure 6.5: The basic idea of sensitivity [21]**

In order to use sensitivity to investigate the input quantization error and find the suitable input bit-width, we re-define the sensitivity.

**Definition 6.1:** *The* ***sensitivity*** *is a numerical value to describe the influence that a small change of X has on the output Y in condition of the worst case:*

$$\Delta Y = AT(f'(X))_{max} * 2^{-m-1} \tag{6-3}$$

The sensitivity reflects the output change in terms of tiny input turbulence. It has the same essence as the representations of Eqn. (5-7) and (5-8), so sensitivity can be used as a substitution. The performance bottleneck in determining the optimized implementation is that the procedure must repeat itself to invoke the conversion algorithm when searching different Taylor terms and input bits. In each flow, this requires invoking the conversion algorithm twice, and subtracting two AT polynomials as Eqn. (5-7) and (5-8) to get the input error in order to confirm whether the input bit-width is satisfied. Of course the complex procedure will consume a lot of time and memory. However, if using sensitivity, as long as *f'(X)* is converted to *AT(f'(X))* and the branch-bound algorithm is used to find the maximum value to match the worst case, the sensitivity can be calculated by its multiplication with $\Delta X$. Here $\Delta X$ is $2^{-m-1}$, i.e., half of the *ulp*. We can see this procedure only invokes the conversion algorithm one time to transform *f'(X)* into *AT(f'(X))*. The advantage is very obvious. When the sensitivity is obtained, combined with the input error bound, it is easy to conclude the suitable input bit-width.

Similarly, the search for an appropriate bit-width of the Taylor coefficients $C_i$ is guided through the corresponding sensitivity, readily calculated using Taylor series, the conversion algorithm and the searching algorithm.

## 6.4.3 Constraint of the Smallest Area

### *A) Optimized Parameters for Taylor Series*

In some cases, there is no limitation for Taylor terms and input bit-width, so engineers can adjust the parameters to achieve an error-satisfied circuit with the smallest area. Consider the following problem, where the total imprecision due to the disparate causes and the cost are obtained through AT.

***Problem 6.4***: *Finding optimized Taylor terms and input bit-width to get the smallest area*

*Inputs*: $f(X)$, $X_0$, $I$, $E$

*Constraints*: *imprecision* $< E$, $\forall X \in I$

*Outputs*: $n$, $m$

The goal is to get a satisfying implementation with the minimum AT size which represents the smallest area. The constraint which restricts the imprecision must be smaller than the error bound. Since coefficients and output bit-width have much less effect of area, we mainly focus on the number of Taylor terms and input bit-width. In deriving a more thorough search scheme, we need the ability to concurrently explore multiple precision parameters.

Figure 6.6 describes the algorithm optimizing the number of Taylor terms and the input bit-width. A pair ($n$, $m$) is referred to as a *node*, representing a combination of a number of Taylor terms ($n$) and an input bit-width ($m$) used in each step of the search. In the first iteration, the algorithm gets the smallest number of Taylor terms for the given error bound, and obtains input bit-width by sensitivity computation (Steps 1 to 5). It is sufficient to consecutively increase the set of Taylor terms used to explore the search space, while simultaneously exploring the alternative input bit-widths (Steps 6 and 7). If the new node can satisfy the error bound $E$, the newly computed number of Taylor terms is assumed, and the algorithm continues to decrease input bit-width until the current node breaks the bound. When it happens, the algorithm backtracks to the previous node and stores it (steps 9 and 10). The procedure is repeated until the change of bit-widths is exhausted, while $e_i > E$ (step 8).

```
Design_min_Taylor_area (f, E)
{
1.   while ($e_t > E$)    { ++n;    $e_t$ = Get_Taylor_error (n)   }
2.   AT_derivative = Convert_Univar_AT($f'$, n, $m_0$);
3.   sensitivity=Search_Imprecision (AT_derivative)* $2^{-m_0-1}$;
4.   ini_m = $m_0$ – log[($E$- $e_t$) / sensitivity];
5.   Store_node (n, ini_m);      m = ini_m;
     while
6.   {   $e_t$ = Get_Taylor_error (++n);
7.       $e_i$ = Get_input_error (f, n, --m);
8.       if ($e_i < E$)
         {   while ($e_i < E - e_t$)   $e_i$ = Get_input_error (f, n, --m);
9.               if (++m != ini_m)
                 {   Store_node (n, m);   ini_m = m;
10.                  Tight_interval (node);    }
         }
11.      else   break;
     }
12.  best_node = Compare_AT_size (nodes);
13.  ($e_c$, q) = Get_coeff_bit (E, $e_t$, $e_i$) ;
14.  $o = -\log_2 (E - e_t - e_i - e_c) + 1$;
     return best_node;
}
Get_input_error (f, n, m)              // Using Eqn. (6-3)
{   AT_derivative = Convert_Univar_AT($f'$, n, m);
    max_val = Search_Imprecision (AT_derivative);
    $e_i$ = max_val * $2^{-m-1}$;        return   $e_i$;
}
Compare_AT_size (nodes)
{   for (i=0; i<nodes_num; i++)
        AT_size[i] = Get_AT (node[i](n), node[i](m) );
    Sort (AT_size);     return the node with smallest AT_size;
}
Get_AT (n, m)
{ for (i=1; i<=n; i++)   AT_num += Choose (m, i); }
```

**Figure 6.6: Algorithm of finding the optimized implementation with smallest area**

Since Taylor series cannot be compared directly, it is necessary to use AT for comparison because of the easy computation of Eqn. (6-1), so in the above procedure the conversion algorithm is invoked to achieve that goal. The searching algorithm helps to find the quantization error represented by AT polynomials. A subroutine `Compare_AT_size` is called into action to compare the AT size of each stored node, and selects the one with the smallest

AT representation. In fact, while the algorithm begins with the largest $e_t$ value (within the total bound $E$) – initially $e_i$ is smallest, but in subsequent steps $e_t$ shrinks while $e_i$ grows until $e_i$ becomes the largest value – the procedure explores the search space, eliminating nodes that will have larger AT than already obtained solutions. Finally, the bit-width of coefficients is calculated using the notion of sensitivity, while the output bit-width $o$ is determined using the expression $o = -log_2 (E- e_t - e_i - e_c) + 1$ (Step 13 and 14). Note that at this point all the error parameters in the above equation can be determined using the optimal values of $n$, $m$ and $q$.

The algorithm provides a branch-and-bound exploration of the space of all potential optimized nodes. When the error bound $E$ is exceeded, the complete subtree of the search tree is safely abandoned. Further, the search is guided by the sensitivity function, as a heuristic to speed up the search. At each node, the error $e_i$ from Eqn. (6-3) is computed in the subroutine `Get_input_error`, which uses the sensitivity definition. The transform of the first order derivative of $f(X)$ is obtained in terms of the Taylor terms $n$ and input bit-width $m$. Then, the branch searching algorithm is invoked to get its maximum mismatch, so the sensitivity is calculated through the multiplication of the maximum mismatch and $\Delta X$, i.e., $2^{-m-1}$. As a result, the conversion algorithm is invoked only once to get AT of $f^{'}(X)$, while the use of Eqn. (5-7) to (5-9) would activate the algorithm twice. The following example illustrates the use of the precision optimization algorithm.

***Example 6.1***: *Consider an implementation of sin(x) represented by Taylor series. Due to the given error bound 0.0002, the algorithm finds the least number of Taylor terms to be 4, and the corresponding input bit-width to be 14 on the condition of the Taylor terms. Therefore, the initial node is (4,14).*

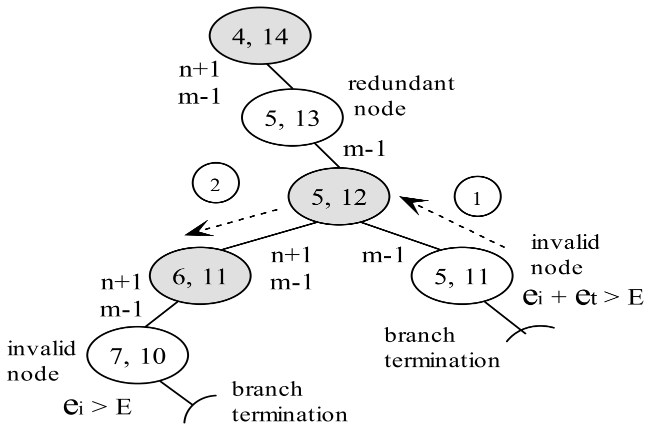


**Figure 6.7: Search of optimized parameters in Example 6.1**

*The algorithm adds then one Taylor term and cuts one input bit at the same time, hence generating a new node (5, 13). By using the sensitivity, $e_i$ is estimated fast, and as this node satisfies the error bound, input bits are decreased again. However, when the node reached (5, 11), the error addition of $e_t$ and $e_i$ is beyond the bound but $e_i$ is smaller than the bound, and the algorithm backtracks to the previous node (5, 12). The node (5, 13) is redundant because its AT terms number is obviously larger than the node (5, 12), and the node (5, 11) is an invalid node. The procedure is repeated with Taylor terms increased to 6 giving the node (6, 11) which satisfies the bound. The input error $e_i$ of the next node (7, 10) breaks through the error bound so it is an invalid node, which means the smallest input bit-width is 11 regardless of the increase in the number of Taylor terms, so the algorithm stops.*

*Figure 6.7 indicates three nodes (4, 14), (5, 12) and (6, 11) that satisfy the given error bound. The procedure Compare_AT_size is then called to select the node with the smallest AT size, so the node (6, 11) is the optimized parameters for Taylor terms and input bit-width.*

From this example, we see that starting from an initial feasible implementation, the algorithm proceeds with generating nodes of improved parameters, and then checks whether such new nodes are within the error bound. In each search step, the sensitivity is used to accelerate calculation of the input quantization error, drastically improving the performance. When the error bound is exceeded, the backtracking technique returns the previously determined feasible solutions, and no solution will be missed.

### *B) Optimized parameters for multivariate polynomials*

The above section proposes an algorithm that is limited to Taylor series of only one word-level variable. Since many real-valued polynomials comprise word-level variables beyond one, the optimization algorithm needs an extension to process it. An algorithm is now presented to handle cases of specifications given over several word-level variables.

```
Design_best_poly_imp (f, E)
{
1.     for (i=0; i<word_var_num; i++)
2.     {     AT_th = Convert_AT(f, i, 0);
3.           AT_real = Convert_AT(f, i, $2^{-m_0-1}$);
4.           error_AT = AT_th – AT_real;
5.           sens [i]=Search_imprecision (error_AT);
       }
6.      ini_bit = Get_ini_node (sensitivity);
7.      final_bit = Get_final_node (sensitivity);
8.      for (i=word_var_num-1; i>=0; i--)
9.      {     ini_bit[i]++;    ini_bit[i+1]--;
          for (m=word_var_num-1; m>=i; m--)
10.           {    stop_error = Compute_input_error (sens, ini_bit);
12.                $e_i$[0] = pow(2, init_bit[0]-$m_0$) * sens[i];
13.                if ($e_i$[0] = stop_error)
                        break;
               else { while ($e_i$ < E)      init_bit[0]--;
                        Store (nodes);    Tight_interval (node);    }
           }
14.           if (ini_bit = final_bit)
                  break;
       }
15.     Irredundant (nodes);
16.    optimized_bit = Compare_AT (nodes);
}
```

**Figure 6.8: Algorithm for finding optimized parameters for real-valued polynomials over multiple variables**

A set of bit-widths for each variable is referred to as a *node* in Figure 6.8. The algorithm first gets sensitivity for each variable as in Step 1 – 5, and obtains the initial node and final node by using sensitivity as in Step 6 – 7. The initial node makes the first variable determine the minimum bit-width and the final node makes the last variable calculate the minimum bit-width.

Beginning from the initial node, the algorithm shrinks the error generated by the first variable by increasing its bit-width. At the same time, the bit-width of the following variable decreases and this may enlarge the error. The procedure propagates the input error within the error bound from the first variable to the last variable in sequence. When the final node is reached, the loop stops and all possible nodes are traversed as in Step 8 – 14. While all intermediate nodes are obtained, the redundant nodes are deleted in Step 15.

If two nodes only differ in one variable and other variables have the same bit widths, the node which has more bits is identified as the redundant node. For example, if the two nodes have three variables consisting of (12, 13, 12) and (12, 14, 12) bits respectively, one variable is different and the node of (12, 14, 12) is deleted as a redundant node. The optimized bit-widths for variables are selected by comparing AT sizes of obtained nodes and choosing the smallest one as in Step 16.

***Example 6.2:*** *Consider a function F with three word-level variables and the given error bound is 60.*

$$F(X, Y, Z) = 2X^2 + 3YZ - 4Z^3 + XYZ$$

*By using sensitivity the initial node is obtained as (14, 16, 18) which means that the error generated by X has the largest value within the error bound, and the final node is (18, 16, 13) which means that the error generated by Z has the largest value within the error bound. The Figure 6.9 describes the two nodes and the error generated by each variable.*

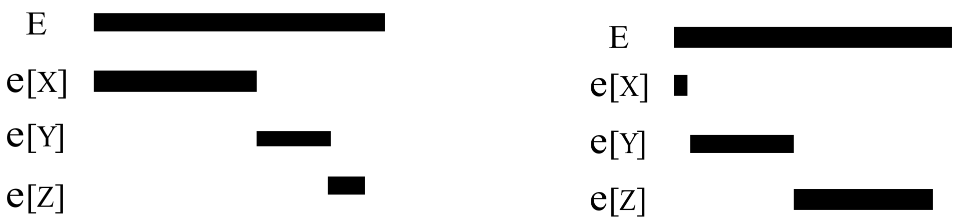


**Figure 6.9: The error of each variable for the initial node and the final node**

*Now the algorithm begins with the initial node to increase bit-width of Y and decrease bit-width of Z, etc., e[Y] is reduced and e[Z] is augmented. The new obtained node is (14, 17, 16) and since the bit-width of Z cannot be cut down any more, the bit-width of X has to be increased to "15" and bit-widths of Y and Z are computed again. Consequently, the node changes to (15, 15, 15). The two nodes are shown in Figure 6.10.*

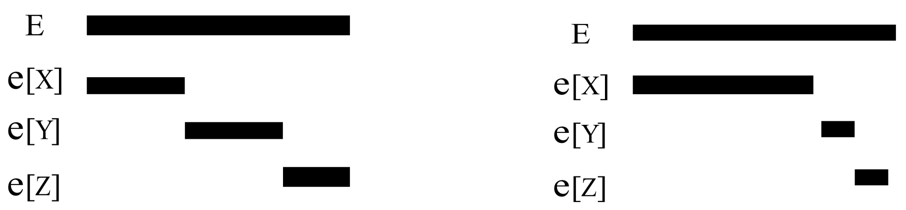


**Figure 6.10: Two intermediate nodes from the initial node**

*The algorithm continues to get intermediate nodes until it reaches the final node. It removes the redundant nodes and creates a search path to represent each node. The chain is described as:*

*(14,16,18)* $\rightarrow$ *(14,17,16)* $\rightarrow$ *(15,15,15)* $\rightarrow$ *(15,16,14)* $\rightarrow$ *(16,14,16)* $\rightarrow$ *(16,15,14)* $\rightarrow$ *(17,14,15)* $\rightarrow$ *(17,17,13)* $\rightarrow$ *(18,16,13)*

*The AT size of each node is calculated and a node with the smallest size is chosen as the optimized node. In this example the optimized node is (16, 15, 14).*

## 6.4.4 Constraint of the Minimum Delay

Some applications often require that the implementation has a minimum delay. Taylor series is implemented by a Horner polynomial evaluation such as the cosine circuit:

$$f(X) = \sum_{i=0}^{n} (-1)^i \frac{X^{2i}}{(2i)!} = 1 + X^2(-\frac{1}{2!} + X^2(\frac{1}{4!} + X^2(...)))$$

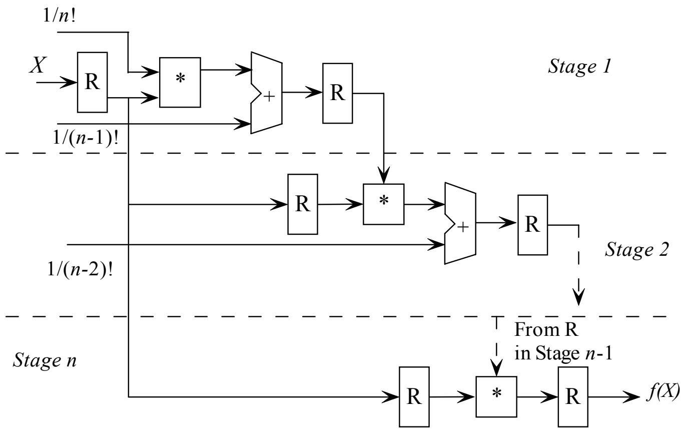


**Figure 6.11: n-stage pipelined circuit**

In Figure 6.11, $n$-terms Taylor series correspond to an $n$-stage circuit represented by a Horner polynomial. Although input bit-width and coefficient bit-width both have effect on delay, it is obvious that the number of Taylor terms has far bigger impact. More terms result in a longer delay, so the minimum delay requires the least Taylor terms and is restricted by the imprecision. The least number of Taylor terms is simple to obtain and the input bit-width can be obtained by using Eqn. (6-3). The problem description is as follows.

***Problem 6.5**: Finding optimized parameters to get the minimum delay*

*Inputs*: $f(X), X_0, I, E$

*Outputs*: $n, m$

*Constraint*: $imprecision < E, \quad \forall X \in I$

*Goal*: *minimum satisfying Taylor terms n*

```
Design_min_delay ($f, E$)
{   while ($e_t < E$)    { --$n$;   $e_t$ = Get_Taylor_error ($n$) };
    $m$ = Initiate ($f, n$) ;
    $e_i$ = Get_input_error ($f, n, m$) ;
    while ($e_i < E - e_t$)
    {   $m$--;
        $e_i$ = Get_input_error ($f, n, m$) ;
    }
    $m$++ ;
    return ($n, m$)
}
```

**Figure 6.12: Algorithm of finding parameters for the minimum delay**

Figure 6.12 describes the algorithm for finding the optimized implementation with the minimum delay. It calculates the least number of Taylor terms to satisfy the inequality $e_t < E$, then decreases the initial input bit-width and keeps the calculation of the input error $e_i$ until $e_i > E - e_t$. So the appropriate input bit-width is obtained.

## 6.4.5 Constraint of Interface Input Bit-width

In some cases the input comes from the output of another module, so the bit-width is determined by that module and it cannot be changed. Figure 6.13 illustrates this situation.

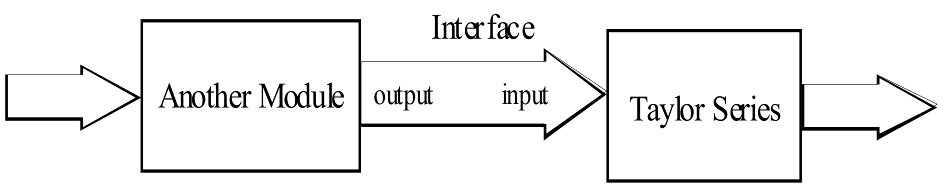


**Figure 6.13: Description of interface input bit-width**

Since the parameter of input bit-width is fixed in this case, only the Taylor terms and coefficient bit-width should be explored to make the imprecision suitable to the error bound. Figure 6.14 describes the algorithm of calculating Taylor terms and coefficients bit-width.

```
Design_fixed_input ($f, E, m$)
1. { while ($e_t < E$)      { --$n$;    $e_t$ = Get_Taylor_error ($n$) };
2.   $e_i$ = Get_input_error ($f, n, m$);
3.   if ($e_i \geq E$ )
          print   "The interface input bit-width is too small to fit the error bound";
4.     else if ($e_t + e_i \geq E$ )
5.     {   while ($e_t + e_i \geq E$ )
6.           {   $e_t$ = Get_Taylor_error (--$n$);
7.               $e_i$ = Get_input_error ($f, n, m$);    }
       }
8.     $e_c = E - e_t - e_i$;
9.     for (i=0 ; i<$m$ ; i++)
           input_val += pow(2, -i-1) ;
10.    for (i=0 ; i<$n$; i++)
           coeff_sen += pow(input_val, i) ;
11.    $q$ = (-log($e_c$ / coeff_sen) / log2) – 1;
       return ($n, m, q$) ;
  }
```

**Figure 6.14: Algorithm of finding parameters for interface input bit-width**

The algorithm first finds the least satisfying Taylor number to make the approximation error $e_t$ smaller than the error bound (Step 1), and calculates the corresponding input error (Step 2). If the error $e_i$ is larger than the error bound, it means that the interface input bit-width is too small to fit the error bound and the algorithm will give the error information (Step 3). If the addition of $e_t$ and $e_i$ is larger than the error bound, which would indicate that the number of Taylor terms is too small, the algorithm increases the number value $n$ and re-calculates its input error (Step 4 – 7) since the number of terms will affect $e_i$ even though the input bit-width is fixed. After the suitable Taylor number $n$ is obtained, the coefficient quantization error $e_c$ is determined, and the algorithm calculates the coefficient bit-width by Eqn. (5-12) corresponding to the worst case (Step 9 - 11).

***Example 6.3:*** *Given an error bound E=2e-4 for exp(X), the interface input bit-width is 13. The algorithm finds the least number of Taylor terms is 6, and gets* $e_t = 1.98e\text{-}4$, $e_i = 1.76e\text{-}4$. *Since* $e_i < E$ *and* $e_t + e_i > E$, *that denote the number of Taylor terms is too small so the algorithm loops to find that the suitable number of Taylor terms is 8. It obtains* $e_t = 2.76e\text{-}6$ *and* $e_i = 1.79e\text{-}4$, *so* $e_c = E - e_t - e_i = 1.82e\text{-}5$. *In order to calculate the coefficient bit-width, Step 9 and 10 execute:*

$$\sum_{i=0}^{7}(\sum_{k=0}^{12}2^{-k-1}x_k)^i = 7.99561$$

*when each* $x_k$ *equals 1 considering the worst case, the equation is* $1.82e\text{-}5 = 2^{-q-1} * 7.99561$ *and the coefficient bit-width q is obtained as 18 bit, so the final obtained parameters are n=8, m=13, q=18.*

## 6.5 Experimental Results

### 6.5.1 Comparison of Two Implementations

#### *(A) Benchmarks*

1) Imprecise Cosine circuit implementation

In ASICs or FPGAs, the pipelined implementation of a cosine circuit represented by finite terms of Taylor series often uses the Horner's polynomial evaluation:

$$f(x) = \sum_{i=0}^{n}(-1)^i\frac{x^{2i}}{(2i)!} = 1 + x^2(-\frac{1}{2!} + x^2(\frac{1}{4!} + x^2(...)))$$

2) B-splines

Uniform cubic B-splines are used for image warping applications. Four B-spline basic functions $B_0$, $B_1$, $B_2$ and $B_3$ are defined by:

$$B_0(u) = -\frac{1}{6}u^3 + \frac{1}{2}u^2 - \frac{1}{2}u + \frac{1}{6} \qquad B_1(u) = \frac{1}{2}u^3 - u^2 + \frac{2}{3}$$

$$B_2(u) = -\frac{1}{2}u^3 + \frac{1}{2}u^2 + \frac{1}{2}u + \frac{1}{6} \qquad B_3(u) = -\frac{1}{6}u^3$$

where $u$= [0, 1]. We use different bits to represent $u$ to implement this design and observe imprecision effects.

3) Chebyshev polynomials

Chebyshev filters are analog or digital filters with a steeper roll-off and more passband ripple. The gain response as a function of angular frequency $w$ of the $n$th order low pass filter is:

$$G_n(w) = \frac{1}{\sqrt{1 + \varepsilon^2 T_n^2\left(\frac{w}{w_0}\right)}}$$

Where $\varepsilon$ is the ripple factor and $T_n$ is the Chebyshev polynomial of the $n$th order. Its mathematical characteristics are derived from Chebyshev polynomials. They are a sequence of orthogonal polynomials which are related to de Moivre's formula and which are easily defined recursively. The Chebyshev polynomials of the first kind are defined by the recurrence relation:

$$T_0(X) = 1 \qquad T_1(X) = 1 \qquad T_{n+1}(X) = 2XT_n(X) - T_{n-1}(X)$$

According to the relation, we get:

$$T_8(X) = 128X^8 - 256X^6 + 160X^4 - 32X^2 + 1$$

$$T_9(X) = 256X^9 - 576X^7 + 432X^5 - 120X^3 + 9X$$

4) Implementations of cubic filters

Cubic filters generally have multiple word-level variables, such as the benchmarks from University of Utah [51]. The complicated module contains three word-level variables, and we have to try exhaustive variable combinations if simulation is adopted, but the method of AT can avoid this time-consuming situation. Consider a filter:

$$F(X, Y, Z) = 16384X^4 + Y^4 + 57344Z^4 + 64767XY^3 + 16127Y^2Z^2 + 8965X^3Z + 19275X^2YZ + 51903XYZ + 32768X^2Y + 40960Z^2 + 32768XY^2 + 49152X^2 + 4869Y$$

5) Discrete Cosine Transform (DCT)

DCT is the kernel of JPEG and MPEG. Here the $8 \times 8$ DCT implementation according to is considered. A vector of input data $x_0 \ldots x_7$ can be transformed to DCT coefficients by $y_0 \ldots y_7$. Coefficients $c_0 \ldots c_6$ are fractional numbers within (-0.5, 0.5) and generally approximated by 8 – 16 bits.

$$\begin{bmatrix} y_0 \\ y_2 \\ y_4 \\ y_6 \end{bmatrix} = \begin{bmatrix} c_0 & c_0 & c_0 & c_0 \\ c_2 & c_5 & -c_5 & c_2 \\ c_0 & -c_0 & -c_0 & c_0 \\ c_5 & -c_2 & c_2 & -c_5 \end{bmatrix} \begin{bmatrix} x_0 + x_7 \\ x_1 + x_6 \\ x_2 + x_5 \\ x_3 + x_4 \end{bmatrix}$$

$$\begin{bmatrix} y_1 \\ y_3 \\ y_5 \\ y_7 \end{bmatrix} = \begin{bmatrix} c_1 & c_3 & c_4 & c_6 \\ c_3 & -c_6 & -c_1 & -c_4 \\ c_4 & -c_1 & c_6 & c_0 \\ c_6 & -c_4 & c_3 & -c_1 \end{bmatrix} \begin{bmatrix} x_0 - x_7 \\ x_1 - x_6 \\ x_2 - x_5 \\ x_3 - x_4 \end{bmatrix}$$

6) Box-Muller implementation

Box-Muller algorithm for generating Gaussian random variable is critical to a number of applications such as accurate bit error rate testers. The algorithm uses the following expression:

$$Y(X_1, X_2) = Y_1(X_1) * Y_2(X_2) = \sqrt{-2\ln(X_1)} * \cos(2\pi X_2)$$

We represent it by a finite number of Taylor series terms:

$Y_1(X_1) = \sqrt{-2\ln(X_1)}$ around the point $X_1 = 0.5$ and $Y_2 = \cos(2\pi X_2)$ around $X_2 = 0$.

$$Y_1(X_1) = 1.17741 - 1.6984(X_1 - 0.5) + 0.4733(X_1 - 0.5)^2 - 1.582(X_1 - 0.5)^3$$

$$+ 1.0198(X_1 - 0.5)^4 - 3.3284(X_1 - 0.5)^5 + 2.7848(X_1 - 0.5)^6$$

$$Y_2(X_2) = \sum_{i=0}^{\infty} (-1)^i \frac{(2\pi X_2)^{2i}}{(2i)!}$$

The implementation consists of two Taylor series and two word-level variables. Imprecision in two variables affect each other, so it is difficult to evaluate imprecision and get the optimized implementation by past univariate explorations.

## *(B) Comparison Results*

| Case | Imp Degree 1 | Imp Degree 2 | Imp Bit 1 | Imp Bit 2 | Error AT Terms | Error | Time(s) | Space(MB) |
|---|---|---|---|---|---|---|---|---|
| cos(x) | 8 | 8 | 20 | 16 | 224747 | 1.2e1-5 | 7.98 | 66.6 |
| cos(x) | 8 | 8 | 24 | 20 | 1007676 | 7.52e-7 | 38.84 | 347.3 |
| cos(x) | 10 | 8 | 24 | 20 | 615115 | 2.75e-7 | 44.16 | 71 |
| cos(x) | 10 | 8 | 24 | 24 | 4533805 | 2.76e-7 | 214.9 | 523.5 |
| B-splines | 3 | 3 | 20 | 16 | 654 | 2.86e-5 | 0.375 | 0.38 |
| B-splines | 3 | 3 | 24 | 20 | 974 | 1.79e-6 | 6.2 | 0.46 |
| B-splines | 3 | 3 | 28 | 24 | 1356 | 1.12e-7 | 114.4 | 0.55 |
| Chebyshev | 8 | 8 | 20 | 16 | 224747 | 9.15e-4 | 7.9 | 75.6 |
| Chebyshev | 8 | 8 | 24 | 20 | 1007676 | 5.72e-5 | 38.73 | 347 |
| Chebyshev | 9 | 9 | 20 | 16 | 381267 | 0.0012 | 21.1 | 145 |
| Chebyshev | 9 | 9 | 24 | 20 | 2147220 | 7.24e-5 | 132.6 | 599 |
| Filter | 4 | 4 | (16,16,16) | (16,16,14) | 11549 | 19.39 | 2.13 | 55.2 |
| Filter | 4 | 4 | (20,20,20) | (18,18,18) | 307909 | 3.83 | 23.5 | 221.1 |
| Filter | 4 | 4 | (20,20,20) | (20,18,18) | 68156 | 2.36 | 16 | 144.5 |
| DCT | 1 | 1 | 16 | 8 | 512 | 15.62 | 0.08 | 0.24 |
| DCT | 1 | 1 | 16 | 10 | 512 | 3.86 | 0.11 | 0.27 |
| DCT | 1 | 1 | 16 | 12 | 512 | 0.92 | 0.13 | 0.29 |
| Box-Muller | (5,4) | (4,4) | (10,10) | (8,8) | 219001 | 0.013 | 4.65 | 38.2 |
| Box-Muller | (5,6) | (5,4) | (12,12) | (10,10) | 613567 | 0.0068 | 18.3 | 86.5 |

**Table 6.1: Error and performance of various components on different criteria**

The module in Figure 6.2 is critical for both the conversion and the branch searching algorithms, so it is important to pay special attention to it. The error AT polynomial is derived from two implemented AT polynomials, from which imprecision can be discovered by the brand search. The module has the advantages of being fast and space-efficient, as Table 6.1 shows.

More bits imply that the results are more precise, i.e., the implemented function value is closer to the originally specified data output. However, the precision comes not only at the cost of area, but also the rate of speed and energy consumption. In light of this, choosing an appropriate length to represent coefficients is worth the effort. Table 6.1 displays imprecision based on different degrees and input bits. It is obvious that imprecision decreases in proportion to the increase of the Taylor degree and input bits. Running time is acceptable even for a large number of terms. Hence, this module provides a reliable method of calculating and matching the imprecision of implementations, which will allow the engineers to lower the cost of design. The results also help to obtain an understanding of whether the existing implementations can be reused.

## 6.5.2 Verification of Imprecise Circuits

In this section, the algorithm in Figure 6.3 is verified. In order to cover general applications, two elementary functions represented by Taylor series and three circuits represented by real-valued polynomials are used as benchmarks to assess the effectiveness of the algorithm.

| Case | Error Bound | n | m | q | o | $e_t$ | $e_i$ | $e_c$ | $e_o$ | AT Term | Impre-cision | Satisfied | Time (S) | Mem (MB) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| sin($X$) | 5e-4 | 4 | 12 | 13 | 12 | 2.48e-5 | 1.22e-4 | 2.44e-4 | 1.22e-4 | 3301 | 5.13e-4 | No | 0.78 | 1.63 |
| sin($X$) | 5e-4 | 4 | 15 | 14 | 11 | 2.48e-5 | 1.53e-5 | 1.22e-4 | 2.44e-4 | 16383 | 4.06e-4 | Yes | 2.42 | 6.47 |
| sin($X$) | 2e-4 | 5 | 14 | 15 | 13 | 2.76e-7 | 3.06e-5 | 7.63e-5 | 6.1e-5 | 14912 | 1.68e-4 | Yes | 15.5 | 12.7 |
| sin($X$) | 2e-4 | 4 | 15 | 13 | 14 | 2.48e-5 | 1.53e-5 | 2.44e-4 | 3.05e-5 | 16383 | 3.15e-4 | No | 4.7 | 6.28 |
| exp($X$) | 2e-3 | 6 | 13 | 12 | 12 | 1.98e-4 | 1.66e-4 | 7.32e-4 | 1.22e-4 | 4095 | 1.22e-3 | Yes | 0.47 | 1.11 |
| exp($X$) | 5e-4 | 6 | 14 | 13 | 14 | 1.98e-4 | 8.29e-5 | 3.66e-4 | 3.05e-5 | 6475 | 6.77e-4 | No | 0.54 | 1.77 |
| exp($X$) | 5e-4 | 6 | 16 | 14 | 13 | 1.98e-4 | 2.07e-5 | 1.83e-4 | 6.1e-5 | 14892 | 4.63e-4 | Yes | 0.89 | 3.68 |
| Bspline | 1e-3 | -- | 12 | 12 | 10 | — | 7.12e-5 | 3.66e-4 | 4.88e-4 | 298 | 9.26e-4 | Yes | 0.09 | 0.14 |
| Bspline | 1e-3 | -- | 13 | 10 | 11 | — | 3.56e-5 | 1.46e-3 | 2.44e-4 | 377 | 1.74e-3 | No | 0.13 | 0.19 |
| Cheby | 5e-3 | -- | 14 | — | 8 | — | 6.54e-3 | — | 1.95e-3 | 14912 | 8.49e-3 | No | 5.84 | 5.14 |
| Cheby | 3e-3 | -- | 17 | — | 9 | — | 8.2e-4 | — | 9.77e-4 | 89845 | 1.97e-3 | Yes | 26.2 | 28.3 |
| DCT | 4 | -- | -- | 8 | -- | — | — | 15.71 | — | 512 | 15.71 | No | 0.08 | 0.24 |
| DCT | 4 | -- | -- | 10 | -- | — | — | 3.93 | — | 512 | 3.93 | Yes | 0.11 | 0.27 |
| DCT | 1 | -- | -- | 12 | -- | — | — | 0.98 | — | 512 | 0.98 | Yes | 0.13 | 0.29 |

**Table 6.2: Checking implementations whether to satisfy the error bound in terms of given parameters**

Table 6.2 lists corresponding errors of various functions due to given parameters and indicates whether the implementation is suitable to the specification on the condition of the error bound. Column 11 shows the number of obtained AT terms; Column 12, "Imprecision," is a summation of the four types of errors; time and space requirements are showed in Columns 14 and 15 respectively, which indicates the performance level of the checking algorithm. It is clear that even when the given error bound is small and parameters have a large bit size, our algorithm is fast and efficient in terms of time and memory requirements.

## 6.5.3 Finding Implementations with the Smallest Area

Engineers usually try to find the implementation with the smallest area, which helps to lower costs. In Figures 6.6 and 6.8 we verify the algorithms used to process Taylor series and multivariate polynomials.

### *(A) Performance of Scheme for Optimized Implementations*

Using traditional methods, simulation cannot find the optimized implementations efficiently because all possible parameters should be investigated for all input values. We provide a much better technique than traditional error bounding techniques which select the precision parameters without exhaustive investigation of the interplay between the imprecision sources.

Two elementary functions (*cos*(*x*) and *exp*(*x*)) given by Taylor series, and three circuits (*B-spline, Chebyshev* and DCT) represented by polynomials with one variable are used in Figure 6.6 as benchmarks to assess the effectiveness of our algorithm. In Figure 6.8, two circuits (*cubic filter* and *Box-Muller*) are used to verify the algorithm to find the optimized implementations of real-valued polynomials with multiple input variables.

Column 2 in Table 6.3 gives different error bounds for various functions; Columns 3 – 10 list the obtained parameters and corresponding errors for implementations optimized for the bounds. Columns 11 and 12 show how many nodes are investigated in the whole procedure and the number of obtained AT terms; Column 13 gives the total imprecision, which is always smaller than the given error bound. Time and space requirements are reported in Columns 14 and 15.

| Circuit | Error Bound | $n$ | $m$ | $q$ | $o$ | $e_t$ | $e_i$ | $e_c$ | $e_o$ | Node | AT Terms | Impreci-sion | Time [s] | Mem [MB] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cos(x)/S | 5e-4 | 5 | 13 | 14 | 11 | 2.32e-6 | 5.96e-5 | 1.23e-4 | 2.44e-4 | -- | 7098 | 4.29e-4 | 1.56 | 1.86 |
| **cos(x)/O** | **5e-4** | **5** | **10** | **17** | **17** | **2.32e-6** | **4.77e-4** | **1.52e-5** | **3.82e-6** | **4** | **1012** | **4.98e-4** | **1.33** | **1.52** |
| cos(x)/S | 3e-4 | 5 | 14 | 13 | 15 | 2.76e-6 | 3.03e-5 | 2.46e-4 | 1.53e-5 | -- | 12910 | 2.94e-4 | 2.56 | 3.01 |
| **cos(x)/O** | **3e-4** | **4** | **12** | **18** | **17** | **1.67e-4** | **1.19e-4** | **7.69e-6** | **3.8e-6** | **7** | **2509** | **2.97e-4** | **1.58** | **2.13** |
| exp(x)/S | 3e-4 | 8 | 14 | 15 | 13 | 2.48e-5 | 8.42e-5 | 1.07e-4 | 6.1e-5 | -- | 9908 | 2.77e-4 | 1.98 | 2.34 |
| **exp(x)/O** | **3e-4** | **7** | **14** | **18** | **17** | **1.98e-4** | **8.42e-5** | **1.31e-5** | **3.7e-6** | **6** | **6476** | **2.95e-4** | **2.37** | **2.86** |
| B-spline/S | 7e-4 | -- | 11 | 11 | 15 | -- | 2.45e-4 | 2.43e-4 | 1.5e-5 | -- | 231 | 5.03e-4 | 0.09 | 0.18 |
| **B-spline/O** | **7e-4** | -- | **10** | **12** | **13** | -- | **4.91e-4** | **1.22e-4** | **6.1e-5** | **1** | **175** | **6.74e-4** | **0.08** | **0.11** |
| Cheby/O | 3e-2 | -- | 12 | -- | 7 | -- | 2.57e-2 | -- | 3.91e-3 | 1 | 3797 | 2.96e-2 | 1.42 | 1.53 |
| Cheby/O | 1e-2 | -- | 14 | -- | 8 | -- | 6.54e-3 | -- | 1.95e-3 | 1 | 12911 | 8.49e-3 | 3.84 | 5.14 |
| Cheby/O | 3e-3 | -- | 16 | -- | 9 | -- | 1.64e-3 | -- | 9.77e-4 | 1 | 39203 | 2.62e-3 | 9 | 15.2 |
| DCT/O | 20 | -- | -- | 8 | -- | -- | -- | 15.71 | -- | 1 | 512 | 15.71 | 0.08 | 0.13 |
| DCT/O | 4 | -- | -- | 10 |  | -- | -- | 3.92 | -- | 1 | 512 | 3.92 | 0.11 | 0.14 |
| DCT/O | 1 | -- | -- | 12 | -- | -- | -- | 0.98 | -- | 1 | 512 | 0.98 | 0.13 | 0.15 |
| Filter/S | 50 | (14, 14, 14) |  |  |  | -- | 27.6 | -- | -- | -- | 47865 | 27.6 | 6.7 | 8.9 |
| **Filter/O** | **50** | **(13, 13, 13)** |  |  |  | -- | **49.3** | -- | -- | **21** | **37636** | **49.3** | **11.9** | **25.4** |
| Filter/S | 35 | (15, 14, 15) |  |  |  | -- | 19.5 | -- | -- | -- | 51391 | 19.5 | 9.2 | 12.3 |
| **Filter/O** | **35** | **(13, 14, 14)** |  |  |  | -- | **32.4** | -- | -- | **14** | **45232** | **32.4** | **18.9** | **25.5** |
| Box-Mul/S | 5e-3 | (5,6) | (12,12) | 11 | 8 | 1.3e-3 | 5.8e-4 | 6.6e-4 | 1.95e-3 | -- | 2153903 | 4.5e-3 | 2.68 | 1.58 |
| **Box-Mul/O** | **5e-3** | **(5,6)** | **(11,11)** | **11** | **10** | **1.3e-3** | **2.4e-3** | **6.8e-4** | **4.9e-4** | **13** | **1620432** | **4.9e-3** | **5.22** | **6.87** |
| Box-Mul/S | 1e-3 | (7,6) | (12,13) | 12 | 11 | 4.2e-5 | 2.8e-4 | 3.3e-4 | 2.5e-4 | -- | 9725892 | 9e-4 | 7.46 | 4.92 |
| **Box-Mul/O** | **1e-3** | **(6,6)** | **(12,12)** | **13** | **12** | **3.6e-4** | **3.2e-4** | **1.6e-4** | **1.2e-4** | **17** | **5938969** | **9.6e-4** | **13.3** | **17.6** |

**Table 6.3: Optimized implementations with smallest area and performance for different error bounds**

In comparison to Figure 6.6, we invoke the sequential method introduced in Figure 6.4 to solve Problem 6.3, which is a case of feasible implementation. By considering the precision parameters sequentially, it mimics often applied schemes for setting precision parameters in isolation. The label "/S" in Column 1 indicates that this sequential assignment algorithm is used in Figure 6.4, while the label "/O" points to the area optimization algorithm here. The optimization algorithm traverses more nodes to investigate the real-valued polynomials with multiple variables, such as cubic filter and Box-Muller, than Taylor series. Please notice that no unique group of parameters satisfies the error bound; changing one parameter would affect the others, as in rows 2 and 3, 4 and 5. These rows have different parameters, and all fit the given error bound indicated by Column 2.

It is clear that, even when the given error bound is small and the parameters are large, our algorithm is fast and efficient in memory requirements. It takes advantage of appropriate paths to search and traverse the least valid nodes, which then leads to very good performance. Our method is not only feasible but a highly efficient way to get the best implementation. In many cases the optimization algorithm is faster than the sequential algorithm, which indicates that finding the best implementation is sometimes more efficient than finding a feasible implementation. We are unique in searching for the optimized implementation for a given error bound, while other researches mostly consider area reduction only in terms of wordlengths.

| Case | Precision | Time (s) [25] | Area [25] (Slices) | Time (s) | Area (Slices) |
|---|---|---|---|---|---|
| B-Spline | 8 | 0.12 | 1368 | 0.07 | 1132 |
| | 16 | 0.19 | 2188 | 0.15 | 2056 |
| DCT | 8 | 0.89 | 3598 | 0.08 | 857 |
| | 16 | 0.51 | 5069 | 0.17 | 1481 |
| Degree 4 Polynomial | 8 | 1.9 | 803 | 0.96 | 763 |
| | 16 | 2.0 | 1921 | 1.55 | 1208 |

**Table 6.4: Result comparison with the paper [45]**

Research in [45] utilizes a multi-stage approach to get 8-bit and 16-bit output precision. Its benchmarks are real-valued polynomials where input wordlength is considered – it cannot deal with Taylor series and function approximation. We consider not only the input but coefficients and the output. Table 6.4 compares results with those in [45] [44]. Our algorithm achieves higher speed and smaller area. We also notice that benchmarks in [43] and [45] have lower degrees than ours. We can handle functions with higher degrees, such as Chebyshev polynomials of degree 9. Furthermore, our algorithms are able to process functions with multiple variables. Cubic filter and Box-Muller, which are more difficult for verification and optimization, are used to prove it. We facilitate a more complex exploration of combining as many factors as possible when investigating the imprecision and approximation of the specification.

| Case | $n$ | $m$ | AA | Ours |
|---|---|---|---|---|
| sin($X$) | 3 | 9 | 1.52e-2 | 1.1e-3 |
| sin($X$) | 3 | 11 | 1.52e-2 | 2.7e-4 |
| sin($X$) | 4 | 10 | 1.57e-2 | 5.46e-4 |
| sin($X$) | 4 | 12 | 1.57e-2 | 1.37e-4 |
| sin($X$)*exp($X$) | 4 | 8 | 6.7e-2 | 1.5e-2 |
| sin($X$)*exp($X$) | 4 | 11 | 6.7e-2 | 1.9e-3 |
| sin($X$)*exp($X$) | 5 | 8 | 8.9e-2 | 1.48e-2 |
| sin($X$)*exp($X$) | 5 | 11 | 8.9e-2 | 1.87e-3 |

**Table 6.5: Error comparison of AA and our method**

Table 6.5 compares the errors obtained by AA and our method for the same number of Taylor terms and input bit-widths, listed in Columns 2 and 3. The error obtained by our method is far smaller than that of AA, which is an indicator of better accuracy compared to past explorations.

### *(B) Area of Mapped Optimized Hardware*

While the optimization algorithm produces precision parameters for a minimal size AT polynomial, the exact area of the resulting circuit depends on the technology used in mapping circuits. We perform further experiments with mapping on FPGAs to evaluate the real area impact of the proposed optimization algorithm. In this section we use the Xilinx Virtex-4 XC4VLX100-12 FPGA, with the ISE tool (version 8.1), the same device and tool as in [45], to obtain a fair comparison of the results.

| Circuit | *E* | Taylor Terms | Input [bits] | Coef. [bits] | Area [Slice] | Saving |
|---|---|---|---|---|---|---|
| cos(X)/S | 3e-4 | 5 | 13 | 14 | 1037 | -- |
| cos(X)/I | 3e-4 | 5 | 12 | 15 | 965 | 6.9% |
| **cos(X)/O** | **3e-4** | **4** | **12** | **16** | **746** | **28.1%** |
| exp(X)/S | 3e-4 | 8 | 14 | 15 | 1179 | -- |
| exp(X)/I | 3e-4 | 8 | 14 | 13 | 1136 | 3.6% |
| **exp(X)/O** | 3e-4 | **7** | **14** | **16** | **933** | **20.9%** |
| Cheby/S | 3e-3 | -- | 20 | -- | 1906 | — |
| **Cheby/O** | **3e-3** | -- | **16** | -- | **1439** | **24.5%** |
| DCT/S | 4 | -- | -- | 14 | 1162 | — |
| **DCT/O** | **4** | -- | -- | **10** | **894** | **23.1%** |
| Filter/S | 35 | -- | (15,15,15) | -- | 3036 | 7.6% |
| **Filter/O** | **35** | -- | **(13,14,14)** | -- | **2725** | **17%** |
| Muller/S | 1e-3 | (7,6) | (13, 14) | 13 | 4327 | -- |
| Muller/I | 1e-3 | (7,6) | (12, 11) | 12 | 3986 | 7.9% |
| **Muller/O** | **1e-3** | **(6,6)** | **(12, 12)** | **13** | **3759** | **13.1%** |

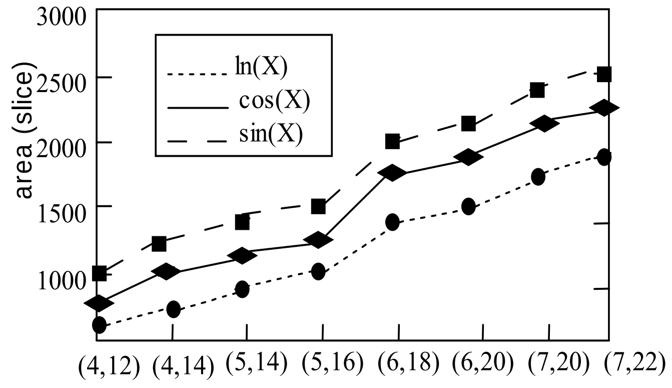
**Table 6.6: Hardware area of optimized circuits**

Table 6.6 compares the area of the FPGA implementations in terms of different parameters. All implementations can satisfy the given error bound *E*, shown in the second column. The rows labeled "/I" use the tight-bound interval method for input bit-width and coefficient bit-width to improve on the sequential algorithm, still labeled with "/S". This new case produces less input and coefficient bits than the sequential algorithm. The rows labeled "/O" invoke the optimization algorithm which contains the tight interval method of this dissertation. The results achieve ~5% area reduction over the optimization algorithm reported in [86] (as "/O"), which uses the plain interval method for transcendental functions such as *cos (X)* and *exp (X)*. The optimization algorithm, in combination with the tight interval method, can save the area by up to 30% over the sequential exploration of individual precision parameters.

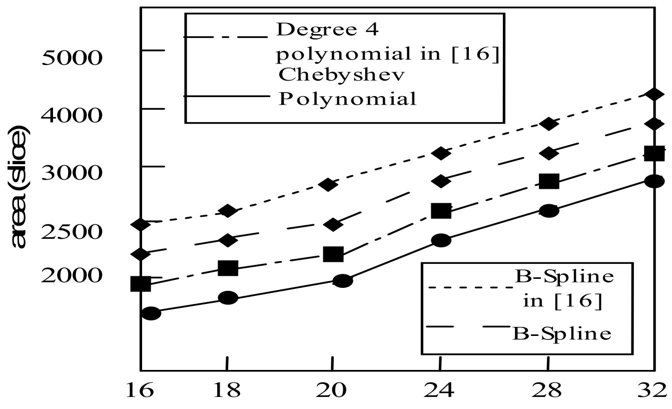
In the case of real-valued polynomials that do not contain function approximation, the optimized algorithm does not benefit from either tight-bound interval method, so the results do not show "/I" for real-valued polynomials.

Finally, Figure 6.15 describes an achievable FPGA hardware area for benchmark circuits using different combinations of Taylor terms and input bits. Such a tabulation facilitates the exploration of trade-offs between precision and complexity. For comparison, Figure 6.15(b) shows B-spline and Chebyshev polynomial results from [45]. Results from our optimization algorithm require less hardware area. When mapped to the same FPGA with the same synthesis tools, our benchmarks – such as B-Splines or the

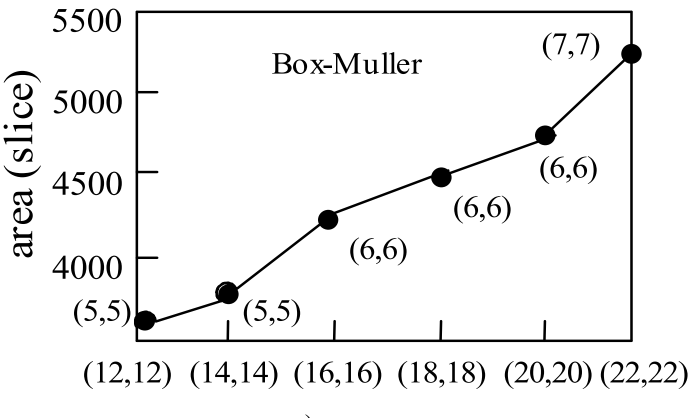
Chebyshev polynomial – reduce the area achieved in [45] by 20% while obtaining the same precision.



a)



b)



c)

**Figure 6.15: Hardware area of Taylor series and real-valued polynomials in different Taylor terms and input bits**

## 6.5.4 Finding Implementations due to Various Constraints

In this section, we verify that the algorithms can handle more constraints in terms of area, delay and interface input, as shown in Figures 6.6, 6.12 and 6.14.

| Case | Error Bound | Constraint | Optimized Parameters | $e_t$ | $e_i$ | $e_c$ | $e_o$ | Impre-cision | AT Term | Time (s) | Mem (MB) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| cos(x) | 5e-4 | fixed input (12) | (4,12,13,14) | 1.67e-4 | 1.19e-4 | 1.83e-4 | 3.05e-5 | 4.99e-4 | 2510 | 0.14 | 0.1 |
| cos(x) | 5e-4 | delay | (4,11,14,18) | 1.67e-4 | 2.39e-4 | 9.14e-5 | 1.91e-6 | 4.99e-4 | 1486 | 0.31 | 0.43 |
| cos(x) | 5e-4 | area | (5,10,17,17) | 2.32e-6 | 4.77e-4 | 1.52e-5 | 3.82e-6 | 4.98e-4 | 1012 | 0.55 | 0.48 |
| exp(x) | 1e-4 | fixed input (17) | (7,17,16,16) | 2.48e-5 | 1.04e-5 | 5.34e-5 | 7.63e-6 | 9.62e-5 | 41225 | 0.56 | 1.41 |
| exp(x) | 1e-4 | delay | (7,15,17,17) | 2.48e-5 | 4.15e-5 | 2.67e-5 | 3.81e-6 | 9.68e-5 | 16383 | 0.41 | 0.95 |
| exp(x) | 1e-4 | area | (8,14,19,17) | 2.76e-6 | 8.29e-5 | 7.63e-6 | 3.81e-6 | 9.71e-5 | 12910 | 1.7 | 0.91 |
| B-spline | 5e-4 | fixed input (11) | (-,11,11,16) | — | 2.45e-4 | 2.43e-4 | 7.63e-5 | 4.95e-4 | 67 | 0.17 | 0.3 |
| B-spline | 5e-4 | area | (-,10,16,19) | — | 4.91e-4 | 7.63e-6 | 9.53e-7 | 4.99e-4 | 56 | 0.14 | 0.13 |
| Cheby | 3e-3 | fixed input (18) | (-,18,-,8) | — | 4.1e-4 | — | 1.95e-3 | 2.36e-3 | 45685 | 11.6 | 19.7 |
| Cheby | 3e-3 | area | (-,16,-,9) | — | 1.64e-3 | — | 9.77e-4 | 2.62e-3 | 39203 | 9 | 15.2 |
| Filter | 50 | fixed input (15,14,-) | (15,14,12) | — | 42.6 | — | — | 42.6 | 42827 | 3.25 | 4.71 |
| Filter | 50 | area | (13,13,13) | — | 49.3 | — | — | 49.3 | 37636 | 11.9 | 25.4 |

**Table 6.7: Optimization of imprecise circuits due to constraints**

The constraints are listed in Column 3, and Column 4 shows calculated optimized parameters; Columns 5 - 8 indicate each error, and the column labeled “Imprecision” (which is smaller than the given error bound) is a summation of the four types of errors; time and space requirements are shown in Columns 11 and 12. The performance indicates the optimization algorithms are highly efficient, while the algorithms can calculate different implementations in terms of the three constraints.

We map the obtained logic to Xilinx Virtex5 FPGAs by their ISE tool. Table 6.8 lists the mapped area and delay for each implementation from Table 6.7. Columns 5 and 6 show the mapped results of delay and area respectively. The implementations in Rows 3 and 6 have the minimum delay, while those in Rows 4 and 7 have the smallest area on the condition of the same error bound. Clearly, the optimized implementations save significant area or delay for circuits compared to other feasible implementations. Less area and less delay means less power dissipation and faster calculation speed, and these are important factors in microchips. This demonstrates the necessity of finding an implementation with the smallest area or delay in real applications.

| Case | *E* | Constraint | Parameter | Delay (ns) | Area (Slices) |
|---|---|---|---|---|---|
| exp(x) | 1e-4 | fixed input | (7,17,16,16) | 11.85 | 1662 |
| exp(x) | 1e-4 | delay | (7,15,17,17) | 9.13 | 1536 |
| exp(x) | 1e-4 | area | (8,14,19,17) | 10.1 | 1389 |
| B-spl | 5e-4 | fixed input | (-,11,11,16) | 6.37 | 422 |
| B-spl | 5e-4 | area | (-,10,16,19) | 5.76 | 396 |
| Cheby | 3e-3 | fixed input | (-,18,-,8) | 13.58 | 1758 |
| Cheby | 3e-3 | area | (-,16,-,9) | 12.23 | 1439 |
| Filter | 50 | fixed input | (15,14,12) | 14.9 | 2646 |
| Filter | 50 | area | (13,13,13) | 13.79 | 2435 |

**Table 6.8: Hardware delay and area for optimized implementations**

## 6.6 Conclusions

We proposed a series of algorithms to handle imprecise circuits in this chapter. A comparison algorithm was described to compute imprecision between two components, and a verification algorithm was then proposed to verify whether a given implementation satisfies the error bound. We determined that a sequential method can find a feasible implementation to fit the given error bound, while optimization algorithms are designed to obtain optimized implementations in terms of different constraints, including the smallest area, minimum delay and interface input bit-width. We saw that these algorithms can process both Taylor series and multivariate polynomials, and cover various applications of imprecise circuits. The experiments used several arithmetic circuits as benchmarks to verify these algorithms and the results were satisfactory.

# Chapter 7

# Range Analysis

*Range analysis is an important task for obtaining the best cost and performance of arithmetic circuits. The traditional methods, either simulation-based or static, have the disadvantages of low efficiency and coarse bounds leading to the use of unnecessary bits. We propose a new method of performing fixed-point range analysis that combines several techniques to efficiently obtain exact ranges.*

## 7.1 Disadvantages of Traditional Methods

In Chapters 5 and 6, we analyzed precision and proposed a series of algorithms to process the design and verification of imprecise circuits. In this chapter we address range and allocate integer bit-widths. Allocating bit-widths in a datapath is a necessary step in the synthesis because of its direct impact on resources and delay. Manual or sub-optimal methods might over- or under-allocate bit-widths. Too few bits will cause overflow, while too many are not cost efficient. Therefore, an automatic way of finding the most appropriate bit-widths is a significant contribution in the high-level synthesis of datapaths.

In obtaining the optimal allocation of bit-widths, the data representation that exposes the variable ranges plays a key role. If we can find the exact ranges for all intermediate variables we can achieve the smallest bit-widths, which will reduce both the circuit area and the delay. Chapter 2 explored past attempts at this. In the range analysis so far, there is a clear separation among the solutions that deal with the quality of the result versus those where the computation time has been the focus, without the explicit possibility to exploit the specifics of a given problem. Dynamic methods and SMT focus on tight ranges, while IA and AA are designed to shorten the calculation time. Figure 7.1 compares the time requirement for each method.

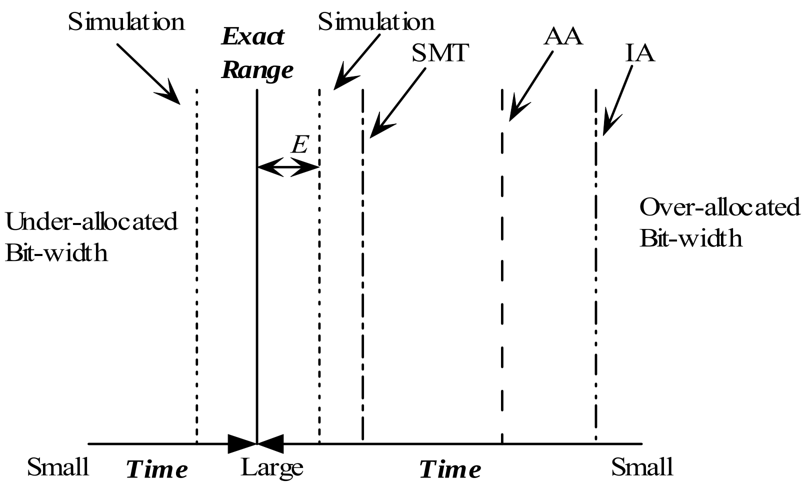


**Figure 7.1: Tradeoff between ranges and calculation times**

The error *E*, defined as the largest difference between the true and the resulting range values, reflects the method accuracy. The goal is to obtain the smallest value of $E$ whilst maintaining the one-sided error, i.e., not underestimating the bit-width. From the figure, SMT, AA and IA may overestimate ranges, which may generate additional bits for data representation.

***Example 7.1: Use of IA and AA in range calculation.*** *Consider the implementation of a function z=ab+c-b with the range of signals as shown in square brackets in Figure 7.2.*

*Using IA is easy to get the ranges for each output. For example, $d_I$ = ab is calculated as [min(-1*4, -1*10, 2*4, 2*10) , max(-1*4, -1*10, 2*4, 2*10)]= [-10, 20]. In AA, an ordinary interval [$x_{min}$, $x_{max}$] for an input variable can be converted into an equivalent affine form* $x_A = x_0 + x_1\varepsilon$ *with*

$$x_0 = \frac{x_{\max} + x_{\min}}{2} \qquad\qquad x_1 = \frac{x_{\max} - x_{\min}}{2} \tag{7-1}$$

*The intermediate signal or the output is represented as a first degree polynomial:*

$$y_A = y_0 + y_1\varepsilon_1 + y_2\varepsilon_2 ... + y_n\varepsilon_n$$

*where $y_0$, $y_1$, ... $y_n$ are floating-point numbers and* $\varepsilon_1, \varepsilon_2 ... \varepsilon_n$ *are symbolic variables whose values are only known to lie in the range[-1,+1].*

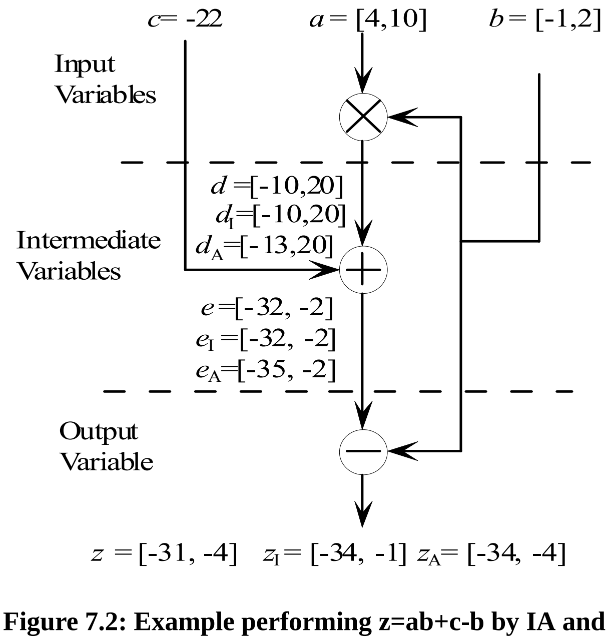


**Figure 7.2: Example performing z=ab+c-b by IA and AA**

*In affine forms, we get:*

$a_A = 7+3\varepsilon_1 \qquad b_A = 0.5+1.5\varepsilon_2 \qquad c_A = -22$

$d_A = a_A b_A = 3.5+1.5\varepsilon_1 + 10.5\varepsilon_2 + 4.5\varepsilon_1\varepsilon_2 = 3.5+1.5\varepsilon_1 + 10.5\varepsilon_2 + 4.5\varepsilon_3$

$e_A = d_A + c_A = -18.5+1.5\varepsilon_1 + 10.5\varepsilon_2 + 4.5\varepsilon_3$

$z_A = e_A - b_A = -19+1.5\varepsilon_1 + 9\varepsilon_2 + 4.5\varepsilon_3$

Figure 7.2 describes the exact ranges and the ranges obtained by IA and AA respectively. We observe that by AA the intermediate variable *e* must be represented by 7 signed integer bits since its range is beyond [-32, 31] by 6 signed integer bits, and the primary output is also using 7 bits; however, 6 bits are enough for the exact ranges to represent *e* and *z* since their ranges are [-32, -2] and [-31, -4]. The reason is as $\varepsilon_1\varepsilon_2 = \varepsilon_3$ in $a_A b_A$, so the term $\varepsilon_1\varepsilon_2$ is dependant of the two variables $\varepsilon_1$ and $\varepsilon_2$, but AA uses a new variable $\varepsilon_3$ as a substitution. This new variable is independent of $\varepsilon_1$ and $\varepsilon_2$, hence AA has to extend the range.

Note that AT can encode intervals, as required in range analysis. It is easy to represent an entire domain, that is, [0, $2^N$-1] for unsigned integers and [-$2^N$, $2^N$-1] for sign extended integers. AT can represent them compactly as $\sum_{k=0}^{N-1} 2^k x_k$ and $(1-2x_{N-1})\sum_{k=0}^{N-2} 2^k x_k$. For example, the expression of $8x_3+4x_2+2x_1+x_0$ represents the entire domain [0, 15]. However, in order to represent the subset of [0, 13], the expression, needs a larger polynomial, $8x_3+4x_2+2x_1+x_0 -14x_3x_2x_1-x_3x_2x_1x_0$. Obviously, the subset generates a much more complex expression, and if there are operations such as multiplication and exponentiation, a number of AT terms will be generated leading to a need for a branch-and-bound search.

Considering the features of AT, Example 7.1 provides useful information for range analysis:

- AA can get the tighter range than IA. For instance, the range of the final output *z* in the datapath obtained by AA is tighter than that of IA.
- IA is not always worse than AA. Observing the intermediate variables "*d*" and "*e*" in Figure 7.2, IA gets the tighter ranges than AA, because there is no correlation existing in the two intermediate outputs *d* = *ab* and *e* = *ab-c*.

Correlation is the concept defined in [42], meaning that if the value of a term in a polynomial changes, the other terms will follow the change. If the polynomial exhibits no correlation, IA is better than AA; otherwise, AA is better.

- AA can represent the arbitrary input range compactly while AT might not, so the input is better to be represented by AA. We note that if the uncertain variable $\varepsilon$ in AA takes an entire range (say normalized to [-1, 1]), AT may easily represent it.
- The worst case is when the unit quantity of range leads to an additional bit. For example, if the exact range of $e$ is [-32, -2] and if the lower bound moves by 1, leading to -33, an additional bit will be generated. Since the intermediate variables cannot obtain the exact range, the datapath propagates the coarse ranges backward to lead the inexact result. Of course, the additional bits are useless and cause unnecessary area and deteriorate the performance.

In terms of the above analysis, we conclude that the advantages of IA, AA and AT are complementary and can be used together, as long as they are employed in suitable conditions. Hence, a hybrid algorithm for the static range analysis and bit-width optimization is appealing. In this chapter, we introduce the methods that try to achieve the exact ranges and the short calculation time concurrently, by tackling every (sub-)problem in a precise, yet efficient way, depending on its nature. We develop a hybrid engine that can get exact ranges while reducing the calculation time as much as possible by analyzing the correlation between the variables, which then lends itself to a selection of a best approach for a given (sub-)problem. The method combines advantages of IA, AA and AT with high efficiency. It is capable of obtaining the exact ranges and allocating the smallest bit-widths to find optimized implementations with the smallest area.

## 7.2 Datapath Analysis

In order to develop the hybrid engine, it is useful to analyze the polynomial representing a datapath. We use Example 7.1 to assist the explanation of the analysis.

### 7.2.1 AA Expressions

The datapath of Example 7.1 has three primary inputs, two intermediate outputs and one primary output. The three primary variables *a*, *b* and *c* are represented by AA in terms of Eqn. (7-1) as:

$$a_A = 7 + 3\varepsilon_1 \qquad b_A = 0.5 + 1.5\varepsilon_2 \qquad c_A = -22$$

The first intermediate variable is $d = ab$. It is easy to confirm that there is no correlation in the polynomial since *a* and *b* are independent, and the two variables only occur once in the polynomial, so the range of *d* can be calculated by IA, that is, [-10, 20]. Although it is simple to get the range of *d*, the AA expression is necessary since in the future the expression may be used. So we get:

$$d_A = a_A b_A = (7+3\varepsilon_1)(0.5+1.5\varepsilon_2) = 3.5 + 1.5\varepsilon_1 + 10.5\varepsilon_2 + 4.5\varepsilon_1\varepsilon_2$$

Then the next intermediate variable in the datapath is $e = ab+c$. By scanning the polynomial, there is also no correlation, so the range of *e* is calculated by IA, that is, [-32, -2]. The AA expression of *e* is:

$$e_A = d_A - c_A = -18.5 + 1.5\varepsilon_1 + 10.5\varepsilon_2 + 4.5\varepsilon_1\varepsilon_2$$

The final step is to determine the range of the primary output $z = ab + c - b$. The polynomial has correlation because the variable *b* occurs two times in the polynomial, so the two terms of "*ab*" and "*-b*" have correlation. The case is much more complex than the cases without correlation. The AA expression of *z* is:

$$z_A = -19 + 1.5\varepsilon_1 + 9\varepsilon_2 + 4.5\varepsilon_1\varepsilon_2$$

## 7.2.2 Determining Quantization Bits of Uncertain Variables

As $\varepsilon_1$ and $\varepsilon_2$ belong to [-1, 1], AT can represent the scope approximately by *m* bits as a signed fractional number, i.e., $(1-2x_0)\sum_{i=1}^{m-1} 2^{-i}x_i$ in Figure 7.3.

| sign | 0.5 | 0.25 | 0.125 |
|---|---|---|---|
| $x_0$ | $x_1$ | $x_2$ | $x_3\ldots$ |

**Figure 7.3: Data format of the signed factional number**

If we can determine the value of *m*, the output is represented compactly and the approximation can be evaluated. So the next step is to choose the appropriate bit-widths for $\varepsilon_1$ and $\varepsilon_2$.

From the Example 7.1, the worst case occurs if the approximation error is beyond 1, when it is possible to generate an additional bit. The uncertainty must be limited to 1 unit to avoid this case, and the inequality becomes:

$$|1.5\vec{\varepsilon}_1|_{err} + |9\vec{\varepsilon}_2|_{err} + |4.5\vec{\varepsilon}_1\vec{\varepsilon}_2|_{err} < 1$$

$\vec{\varepsilon}_1$ and $\vec{\varepsilon}_2$ are quantized uncertain variables to replace $\varepsilon_1$ and $\varepsilon_2$. So there is the inequality:

$$|4.5\vec{\varepsilon}_1\vec{\varepsilon}_2|_{err} < 1 \quad \Rightarrow \quad 4.5[1-(1-2^{-m+1})^2] < 1$$

The reason to choose the term "$4.5\vec{\varepsilon}_1\vec{\varepsilon}_2$" first is because the term has second-order uncertainty while terms such as $1.5\vec{\varepsilon}_1$ and $9\vec{\varepsilon}_2$ have first-order uncertainties. The order of uncertainty for a monomial is defined as the degree summation of uncertain variables in the monomial. The preferential choice of the term with highest order uncertainty is helpful to decrease the calculation complexity. Obviously when all bits in the data format are 1, the fractional number has the largest approximation error $2^{-m+1}$, or else $2^{-m}$ for other values. For instance, in the Figure 7.3 to approximate "1", while $x_1$, $x_2$ and $x_3$ are all 1, the error is $2^{-3} = 0.125$, and in other values the error is $2^{-4} = 0.0625$. While the maximum error is $2^{-m+1}$, the value of $\vec{\varepsilon}_1$ is $1-2^{-m+1}$ and $\vec{\varepsilon}_1\vec{\varepsilon}_2$ equals to $(1-2^{-m+1})^2$. Therefore, the maximum error of the term $4.5\vec{\varepsilon}_1\vec{\varepsilon}_2$ is represented as $4.5[1-(1-2^{-m+1})^2]$. Here we assume that $\vec{\varepsilon}_1$ and $\vec{\varepsilon}_2$ have uniform bit-widths *m*.

By solving the inequality, the value of *m* is 5, that means, $\vec{\varepsilon}_1$ and $\vec{\varepsilon}_2$ both have 5 bits at least to satisfy that the approximation error is restricted to 1 unit. Substituting $\vec{\varepsilon}_1 = \vec{\varepsilon}_2 = 0.9375$ as five bits, the real value is $4.5 * 0.9375^2 = 3.955$.

We conclude that the real maximum error is 4.5 – 3.955 = 0.545 so the left error space is 1-0.545 = 0.455. Then we explore the term $1.5\vec{\varepsilon}_1$. The inequality is $1.5 * 2^{-m+1} < 0.455$. So $\vec{\varepsilon}_1$ must have three bits at least. Considering 5 bits in the term $4.5\vec{\varepsilon}_1\vec{\varepsilon}_2$ and 3 bits in the term $1.5\vec{\varepsilon}_1$, $\vec{\varepsilon}_1$ should be 5 bits to satisfy the two terms at the same time. So we get $1.5\vec{\varepsilon}_1 = 1.5 * 0.9375 = 1.40625$.

The real maximum error for the term $1.5\vec{\varepsilon}_1$ is 1.5-1.40625 = 0.09375 so the left error space is 1-0.545-0.09375=0.36125. The final term $9\vec{\varepsilon}_2$ must satisfy the inequality $9 * 2^{-m+1} < 0.36125$.

The bit-width of $\vec{\varepsilon}_1$ is 6 in the inequality and in combination with the bit-width in the term $4.5\vec{\varepsilon}_1\vec{\varepsilon}_2$, we obtain the final bit-width of $\vec{\varepsilon}_2$ is 6. At last, we determine the two uncertain variables have 5 and 6 bits. The expression of *z* is changed as:

$$z = -19 + 1.5(1-2x_0)\sum_{i=1}^{4} x_i 2^{-i} + 9(1-2y_0)\sum_{i=1}^{5} y_i 2^{-i} + 4.5[(1-2x_0)\sum_{i=1}^{4} x_i 2^{-i}][(1-2y_0)\sum_{i=1}^{5} y_i 2^{-i}]$$

By invoking the conversion algorithm and the branch searching algorithm, the lower bound and the upper bound are -4.7881 and -30.3814. Since the bounds are approximate to the exact bounds, and the absolute values of uncertain variables are smaller, the calculated bounds should be covered by the exact bounds, so we get the exact bounds of the primary output are [-31, -4].

If the term $4.5\vec{\varepsilon}_1\vec{\varepsilon}_2$ is not chosen first, $\vec{\varepsilon}_1$ and $\vec{\varepsilon}_2$ both need 8 signed bits for representations. Although the obtained range of *z* is same, the calculation time increases much more since more quantization bits burden the conversion algorithm and the branch searching algorithm. Hence, the first choice of the term with higher uncertain degree is very significant.

### 7.2.3 Allocating Bit-widths for All Outputs

It is easy to allocate the bit-widths After all intermediate ranges have been obtained. The integer bit-width (IB) is calculated as:

$$IB = [\log_2 (max(|x_{low}|, |x_{upp}| ))] + \alpha \qquad (7\text{-}2)$$

where

$$a = \begin{cases} 1 & mode(\log_2|x_{upp}|, 1) \neq 0 \\ 2 & mode(\log_2|x_{upp}|, 1) \neq 0 \end{cases}$$

In Eqn. (7-2), $x_{low}$ and $x_{upp}$ represent the lower and the upper bound of the obtained range, and the square bracket is the ceiling function. The intermediate outputs and the primary output all have signed 6 bits since their ranges are restricted in the scope [-32, 31]. Compared to AA, *e* and *z* save one bit; compared to IA, the final output range is much tighter.

Our method combines techniques of IA, AA and AT. If the polynomial has no correlation, it adopts IA to calculate the range; if not, using AA gets compact expressions while AT is applied to handle correlation. The step of quantizing the uncertain variables in AA expressions keeps trace to the correlation, hence the accuracy is guaranteed. Therefore, the method avoids their disadvantages and integrates each advantage, and hence it can process the worst case to obtain exact ranges.

## 7.3 Algorithm for Calculating Ranges

Figure 7.4 describes the algorithm to allocate bit-widths in a datapath. It first retrieves the polynomial description, and gets the AA expression for future utilization. If the polynomial has no correlation, IA is used to get the exact range so the bit-width is determined; if not, the uncertain variables are quantized in AA expression, the conversion algorithm is invoked to convert the expression to an AT, and the branch-and-bound searching algorithm finds the upper and the lower bounds. Finally, the bit-width of the output is allocated.

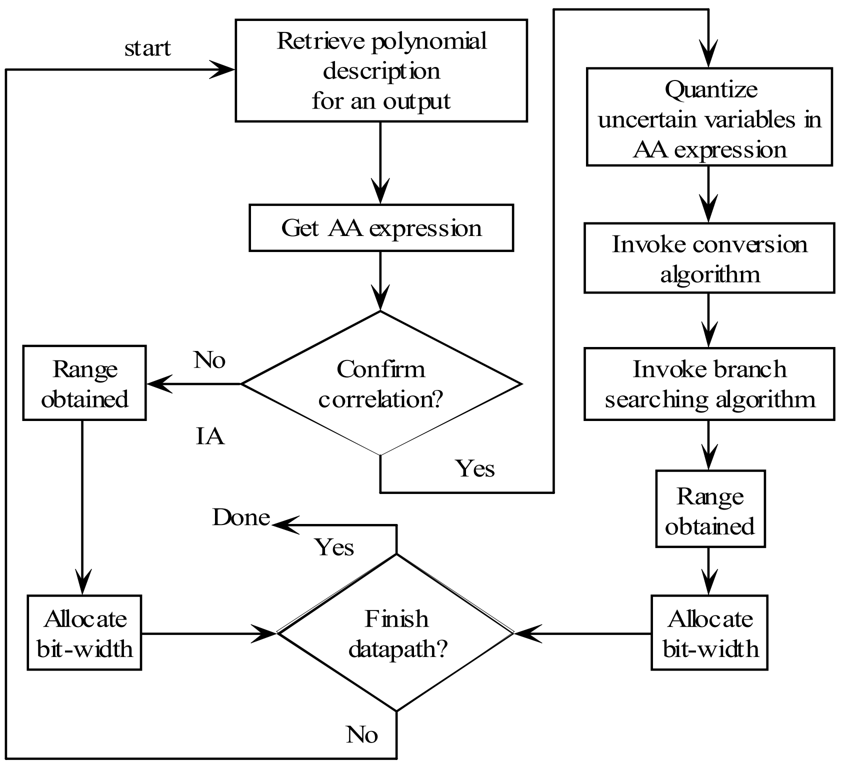

start
Retrieve polynomial description for an output
Get AA expression
Confirm correlation?
No
IA
Range obtained
Yes
Quantize uncertain variables in AA expression
Invoke conversion algorithm
Invoke branch searching algorithm
Range obtained
Allocate bit-width
Done
Yes
Finish datapath?
Allocate bit-width
No

**Figure 7.4: Algorithm of allocating bit-widths**

The two key steps in Figure 7.4 are how to confirm correlation and quantize uncertain variables. Figure 7.5 describes how to check whether a polynomial has some correlation. The symbol *n* represents the number of input variables in the polynomial and the symbol *t[i]* records occurred times of the variable $v_i$. If all variables occur only once, the function clearly exhibits no correlation.

```
Confirm_correlation (f)
{   for (p=0; p<terms_num; p++)      // loop all terms
    { for (i=0; i<n; i++)
         if (variable $v_i$ is present)
           t[i]++;      // count appearances for the variable
    }
    for (i=0; i<n; i++)
    {   if (t[i] >1)     // the polynomial f has correlation
            return corr_flag = 1;
    }
```

**Figure 7.5: Algorithm for confirming correlation**

In Example 7.1, the algorithm scans the intermediate variable *d* and finds that the variables *a* and *b* only occur one time in the polynomial, so no correlation exists; similarly, the variable *b* occurs two times in the expression of *z*, so the polynomial has correlation and IA cannot obtain its range directly. AA and AT are used to process the case. The important step is determining the quantization bits for each uncertain variable. Figure 7.6 describes the subroutine.

The subroutine sorts the terms in the AA expression. The terms with higher uncertain degrees are explored with higher priority. Considering the worst case, the initial error space is set to 1 unit, so the initial bit-widths of uncertain variables can be procured. The error space is reset and the sub-routine continues to handle the next term. After all terms are processed, the final bit-widths of corresponding uncertain variables are the maximum obtained bit-widths.

```
Determine_uncertain (AA_Expre)
{   for (p=0; p<terms_num; p++)
    // loop all terms in AA_Expre
    { if (current_term.degree < next_term.degree)
          Move_forward (next_term);
    }   // sort terms with higher uncertain degrees;
    error_space = 1;
    for (p=0; p<terms_num; p++)
    // loop all sorted terms
    {    $m_p = [1 - \log_2(1-(1-error\_space/term.coeff)^{1/degree})]$;
         error_space = error_space - term.coeff * $[1-(\sum_{i=1}^{m-1} 2^{-i})^{degree}]$;
         store $m_p$ in corresponding uncertain variable $\varepsilon$ ;
    }
    for (i=0; i<uncertain_var_num; i++)
      $q_i$ = max (bit-widths for the uncertain variable $\varepsilon_i$);
    return q;   }
```

**Figure 7.6: Algorithm of determining quantization bit-widths for uncertain variables**

The initial error space limits the deviation of the obtained ranges - the unit value can cause the worst case to make the obtained ranges not equal to the exact ranges. The efficient AT conversion and branch-and-bound searching are instrumental to the high efficiency in performing the range analysis.

## 7.4 Experimental Results

We implement the algorithm by C++. The benchmarks are described by Verilog HDL augmented with the datapath representation and range information. We try several benchmarks to assess its performance. Experiments are done on a 512MB, 2.4GHz Intel Celeron machine under Linux.

### 7.4.1 Filter Polynomial

Image processing applications often use polynomial filter with presentation given by:

$$F = a_1x^4 + a_2x^3 + a_3x^2$$

Here we consider an example as ($X \in [-20, 10]$):

$$F = 4X^4 + 16X^3 + 20X^2$$

The implementation has four intermediate variables.

$$q_1 = X^2 \qquad q_2 = q_1X \qquad q_3 = q_2X$$

$$q_4 = 4q_2 + 16q_3 \qquad z = q_4 + 20q_1$$

| Output | Our Method | | AA | |
|---|---|---|---|---|
| | Range | Bit | Range | Bit |
| $q_1$ | [0, 400] | 9 | [-350, 400] | 10 |
| $q_2$ | [-8000, 1000] | 14 | [-8000, 7750] | 14 |
| $q_3$ | [0, 160000] | 18 | [-158750,160000] | 19 |
| $q_4$ | [-108,512000] | 20 | [-511000,534000] | 21 |
| $z$ | [0, 520000] | 19 | [-511000,542000] | 21 |

**Table 7.1: Comparison with AA for filter polynomial**

## 7.4.2 Dickson Polynomial

Dickson polynomials have important applications in coding and communication areas. The definition for $n$>0 is:

$$D_0(x, a) = 2 \qquad D_n(x, a) = \sum_{p=0}^{[n/2]} \frac{p}{n-p} \binom{n-p}{p} (-a)^p x^{n-2p}$$

The polynomial contains two variables. Here we explore the implementation of the $4^{th}$ order polynomial over real numbers (assume $x \in [-50, 50]$, $a \in [-20, 40]$):

$$D_4(x, a) = x^4 - 4x^2 a + 2a^2$$

The implementation has 5 intermediate variables from $q_1$ to $q_5$:

$$q_1 = x^2 \qquad q_2 = q_1^2 \qquad q_3 = 4q_1 a$$
$$q_4 = 2a^2 \qquad q_5 = q_2 - q_3 \qquad z = q_5 + q_4$$

| Output | Our Method | | AA | | Time (s) | | |
|---|---|---|---|---|---|---|---|
| | Range | Bit | Range | Bit | Ours | Sim | AT |
| $q_1$ | [0,2500] | 12 | [-2500, 2500] | 13 | 0.03 | 0.03 | 0.08 |
| $q_2$ | [0，6250000] | 23 | [-6250000, 6250000] | 24 | 0.04 | 0.14 | 1.56 |
| $q_3$ | [-200000,400000] | 20 | [-400000, 400000] | 20 | 0.06 | 0.2 | 0.25 |
| $q_4$ | [0, 3200] | 12 | [-2800, 3200] | 13 | 0.03 | 0.03 | 0.27 |
| $q_5$ | [-6399,6450000] | 24 | [-6450000, 6450000] | 24 | 1.15 | > 60 | 1.87 |
| $z$ | [-3199,6453200] | 24 | [-6453200, 6453200] | 24 | 1.4 | > 60 | 2.35 |

**Table 7.2: Comparison of our method, AA, improved simulation and AT for Dickson polynomial**

## 7.4.3 Multivariate Datapaths

Here, a datapath is always expressed by a polynomial with multiple variables. The polynomial with 3 integer variables is:

$$F = 30A^2 - 60AB - 40BC$$

Here $A \in [-20, 30]$, $B \in [10, 40]$ and $C \in [-10, 30]$. The case is broken intermediately into:

$$q_1 = 30A^2 \qquad q_2 = 60AB \qquad q_3 = 40BC$$

$q_4 = q_1 - q_2$ $z = q_4 - q_3$

| Output | Our Method | | AA | |
|---|---|---|---|---|
| | Range | Bit | Range | Bit |
| $q_1$ | [0, 27000] | 15 | [-25650, 27000] | 16 |
| $q_2$ | [-48000, 72000] | 18 | [-57000,72000] | 18 |
| $q_3$ | [-16000, 48000] | 17 | [-28000, 48000] | 17 |
| $q_4$ | [-45000, 60000] | 17 | [-82500, 69000] | 18 |
| $z$ | [-93000, 76000] | 18 | [-131500, 97000] | 19 |

**Table 7.3: Comparison with AA for a multivariate datapath**

## 7.4.4 Energy Spectral Density

The benchmark of energy spectral density [55] calculates:

$$\phi(w) = \frac{1}{2\pi} F(w)F^*(w)$$

where *F(w)* is the FFT of discrete signals. The experiments use an 8-point with each of the 8 inputs a complex number in $[-128,128] + [-128,128]i$.

| Output | Our Method | | AA | | SMT | |
|---|---|---|---|---|---|---|
| | Range | Bit | Range | Bit | Range | Bit |
| 0 | [0, 2097152] | 22 | [-1835008, 2097152] | 22 | [-1, 2097153] | 22 |
| 1 | [0, 1984106] | 21 | [-2373666, 2635814] | 23 | [-1, 1984106] | 21 |
| 2 | [0, 1790022] | 21 | [-2269321, 2531463] | 23 | [-1, 1790022] | 21 |
| 3 | [0, 2052757] | 21 | [-2373666, 2635814] | 23 | [-1, 2052757] | 21 |
| 4 | [0, 2097152] | 22 | [-1835008, 2097152] | 22 | [-1, 2097153] | 22 |
| 5 | [0, 1957096] | 21 | [-2373666, 2635814] | 23 | [-1, 1957096] | 21 |
| 6 | [0, 1790023] | 21 | [-2269321, 2531463] | 23 | [-1, 1790023] | 21 |
| 7 | [0, 2029555] | 21 | [-2373666, 2635814] | 23 | [-1, 2029555] | 21 |

**Table 7.4: Our method vs. AA vs. SMT for energy spectral density**

We use the AA method introduced in [42] for comparison. In Table 7.1 to 7.3, the intermediate variables' and the primary outputs' ranges are exact and far tighter than those of AA. Table 7.2 compares execution time with the methods of improved simulation and pure AT. Since the pure AT method generates more terms and spends time in conversion and the search, while the improved simulation has to calculate many points and compare them to found bounds, their execution time is much longer than our method. Table 7.4 compares our results with those obtained by SMT [55]. Using a benchmark from [55], our method can get the exact ranges, while SMT obtains more precise ranges than AA. Regarding the SMT results, since there are negative quantities, the bit-widths could require one additional bit, but as authors estimate that the function will only have positive values, the additional bit is omitted in their reporting. Reported runtime in [55] is on the order of 100s of seconds, while we spend 8.9 seconds for the same benchmark.

### 7.4.5 Area of Optimized Implementations

As the exact area of the resulting circuit depends on the technology used in mapping circuits, we perform further experiments with mapping to FPGAs. We map the circuits to Xilinx Virtex5 FPGAs using ISE tool, version 8.1, to evaluate the real area impact of the proposed algorithm in Table 7.5. Again, the implementations obtained by AA are used as comparison.

| Circuit | Area (Slices) | | | Delay (ns) | | |
|---|---|---|---|---|---|---|
| | Ours | AA | Saving | Ours | AA | Saving |
| Filter | 686 | 772 | 11.1% | 23.5 | 26 | 9.62% |
| Filter | 725 | 805 | 9.96% | 24.6 | 26.9 | 8.55% |
| Filter | 756 | 820 | 7.77% | 25.4 | 27.5 | 7.64% |
| Dickson | 809 | 897 | 9.8% | 31.3 | 33.5 | 6.57% |
| Dickson | 845 | 926 | 8.7% | 32 | 33.9 | 5.6% |
| Dickson | 877 | 948 | 7.5% | 32.4 | 34.1 | 4.99% |
| MultiVar | 532 | 574 | 7.3% | 27.4 | 29.9 | 8.36% |
| MultiVar | 557 | 596 | 6.5% | 27.9 | 30.2 | 7.62% |
| MultiVar | 588 | 623 | 5.6% | 28.7 | 30.7 | 6.51% |

**Table 7.5: Area comparison of our method and AA**

The input bit-widths of the three benchmarks increase, reflecting in the area increase. Column 4 indicates the saving ratio. There are four variables which save bits in the filter benchmark, while another two benchmarks only have three variables, so the filter has larger area saving ratio. With the increase of the input ranges, the saving ratio decreases because the auxiliary area caused by additional bits reduces. Our method can achieve the optimized implementations with area smaller for around 6% - 12%. The delay of implementations is compared in Column 5 - 7. Due to the he smaller bit-widths, we are able to decrease delay around 6%- 10%. Hence, the hybrid method is helpful to both area and delay. The calculation time of AA is close to

1 second while our method requires 3 – 6 seconds. The increase in computation time pays off, as the obtained ranges are far tighter.

## 7.5 Conclusions

Range analysis is an important step in RTL synthesis since it directly impacts cost and performance. Previous methods, including the improved simulation-based techniques, are of low efficiency, while the AA-based methods reach coarse bounds. The coarse ranges may generate unnecessarily additional bits, leading to more costly circuits. In this paper, we propose a new method to calculate ranges statically. It combines techniques of IA, AA and AT to find ranges efficiently, while at the same time the obtained ranges can be exact, hence avoiding the generation of additional bits. The key to our hybrid method is the ability to handle the correlation. Each intermediate output can obtain the smallest satisfying bit-width based on the ranges; therefore, the optimal implementation with the smallest hardware area can be achieved. The experiments indicate that the method is much closer in computation time to the approximate methods such as AA-based rather than more exhaustive SMT-based, while at the same time optimizing the bit-widths, which necessarily leads to the efficient area and delay characteristics obtained by synthesis.

# Chapter 8 Combining Range and Precision

*We discuss fixed-point circuits together with range and precision in this chapter. The important aspects lie in how to allocate appropriate integer and fractional bit-widths, and estimate the error. It is necessary to conduct the mathematic model of the circuit in order to get the optimized implementation. We analyze precision, and propose an algorithm to calculate range and optimize the allocation of fractional bit-width. Furthermore, circuits with feedbacks and floating-point representation are investigated.*

## 8.1 Fixed-Point Representation

We have discussed precision and range corresponding to fractional bit-width (FB) and integer bit-width (IB) respectively in above chapters. A fixed-point representation often has IB and FB concurrently. Figure 8.1 describes the two problems in the fixed-point representation.

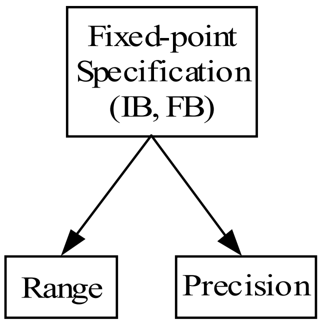


**Figure 8.1: Exploration of the fixed-point representation**

***Example 8.1:*** *A datapath with three primary inputs a, b and c is shown in* Figure 8.2. *The numerical bounds are given in the square brackets.*

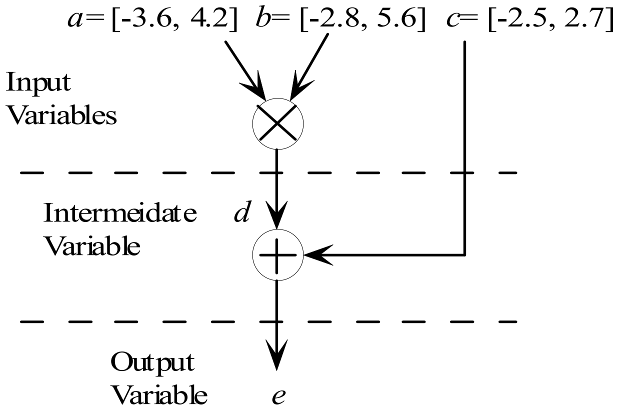


**Figure 8.2: The datapath of Example 8.1**

*The datapath has one intermediate output d and one primary output e where d=ab and e=d+c. All variables need to be represented by the fixed-point format both with IB and FB.*

Important problems in the example are stated as follows:

- How to get the value bounds for all variables?
- How to allocate the bit-widths for all variables included the primary inputs?

- How to estimate the error for the primary output?
- How to get the optimized implementation?

The above four questions are most significant to all fixed-point circuits. Since the primary inputs have FB, we can conclude that the primary output also has FB. In real applications, engineers generally give the error bound to make the maximum difference between the exact value and the true value of the primary output restricted in the bound. The interplay of the four problems results in hardness of analysis. Determination of IBs relies on the values bounds of all variables, while determination of FBs and optimization rely on the error bound.

Past explorations only focus one aspect. For instance, authors in [55] investigate how to get ranges and then allocate IB, but they do not explore the precision so cannot allocate FB. The paper [42] analyzes both range and precision, and allocates IB and FB. But it has no capability to get the optimized implementation with the smallest hardware area. Exploring the four problems concurrently is difficult. In this chapter, we analyze range and precision, and propose an algorithm to allocate IB and FB, and then obtain the optimized implementation.

## 8.2 Analysis of Range and Precision

Now we use the Example 8.1 to help analysis of range and precision. The Chapter 7 has already given the algorithm that combines IA, AA and AT in Figure 7.3 to get the exact ranges. The algorithm represents primary inputs as AA expressions, and then checks whether the polynomial representing the datapath has correlation between monomials. If not, IA is invoked to get ranges; otherwise, it quantizes the uncertain variables, and the algorithms of AT conversion and branch searching are invoked to find ranges. The hybrid method has high efficiency and can get exact ranges to allocate smallest IB for all variables. Using the hybrid method in the Chapter 7, the minimum and the maximum integer parts of the intermediate variable *d* are -21 and 24 respectively, while for the primary output *e* they are -23 and 26. Therefore, the

IBs for the two output are both 6 (included the sign bit).

So the main problem changes to how to analyze precision. It is easy to know the biggest error is $2^{-FB-1}$ if the fractional part has the length of FB.

***Example 8.2:*** *Given the range of [0, 14.95], IB is 4 and let FB be 3. We can evaluate the precision is* $2^{-4} = 0.0625$. *The maximum value "14.95" can be coded to "15" and the error is 0.05. There is a special case. If the range is [0, 15.95], since the IB is only 4, the maximum value "15.95" is coded as 15.875, the error is 0.075 and beyond* $2^{-4}$. *In this case, the reason is that the IB restricts the coding to represent 16, so the largest error is not* $2^{-FB-1}$ *but* $2^{-FB}$.

Generally, we do not consider the special case that the integer part equals $2^N$-1. If it occurs, as long as the IB increases one bit, the special case is cancelled. So we explore the biggest error $2^{-FB-1}$. Let $\tilde{a}$ represent the exact value and $a$ represent the true value. We get:

$$\tilde{a} = a + 2^{-FB_a-1}\varepsilon_1 \qquad\qquad \tilde{b} = b + 2^{-FB_b-1}\varepsilon_2 \qquad (8\text{-}1)$$

where $FB_a$ is the $FB$ of $a$. Hence, the error at $x$ due to finite precision effects is given by

$$E_a = 2^{-FB_a-1}\varepsilon_1$$

For multiplication: $\tilde{d} = \tilde{a}\tilde{b} = ab + a\,E_b + b\,E_a + E_a\,E_b + 2^{-FB_d-1}\varepsilon_3$

$$\Rightarrow E_d = a\,E_b + b\,E_a + E_a\,E_b + 2^{-FB_d-1}\varepsilon_3$$

The primary output:

$$\tilde{e} = \tilde{d} + \tilde{c} = ab + a\,E_b + b\,E_a + E_a\,E_b + 2^{-FB_d-1}\varepsilon_3 + c + E_c + 2^{-FB_e-1}\varepsilon_5$$

$$\Rightarrow E_e = a\,E_b + b\,E_a + E_a\,E_b + 2^{-FB_d-1}\varepsilon_3 + E_c + 2^{-FB_e-1}\varepsilon_5$$

Note that $E_e$ would be at its maximum when the signals $a$ and $b$ are at their absolute maximum, that is, $a$ = 4.2 and $b$ = 5.6. We get the following maximum error at the output $\tilde{e}$:

$$max(E_e) = 4.2 * 2^{-FB_b-1} + 5.6 * 2^{-FB_a-1} + 2^{-FB_a-FB_b-2} + 2^{-FB_d-1} + 2^{-FB_c-1} + 2^{-FB_e-1} \qquad (8\text{-}2)$$

We first assume all variables with uniform FBs, so get:

$$max(E_e) = 6.4 * 2^{-FB} + 0.25 * 4^{-FB} < 0.01$$

Solving the inequality, the FB is 10 which means if all variables have 10 bits for fractional representations, the error of the primary output can be limited in the error bound. However, the uniform FBs do not lead to the optimized implementation. The Chapter 6 introduces how to use the AT size as the cost function to find the optimized implementation with the smallest area. Therefore, we need to represent the datapath by AT.

The sign bit is assumed to be the most significant bit (MSB) of the input vector. Figure 8.3 describes the fixed-point representation of *a.*

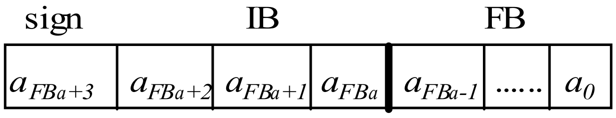


**Figure 8.3: Fixed-point representation of variable a**

Since the range of *a* is [-3.6, 4.2], the IB is 3 and one bit sign, so the AT representation is:

$$AT(a) = (1-2a_{FB_a+3})\sum_{i=0}^{2} 2^i a_{FB_a+i} + \sum_{k=0}^{FB_a-1} 2^{k-FB_a} a_k \qquad (8\text{-}3)$$

The first part in Eqn. (8-3) represents the sign and integer number, while the second part represents the fractional number.

In Chapter 6, we introduce using AT size to indicate the area because AT size is in a good correspondence to the overall circuit area. The datapath is represented by AT as:

$$AT(d) = AT(a) * AT(b)$$

$$= ((1-2a_{FB_a+3})\sum_{i=0}^{2} 2^i a_{FB_a+i} + \sum_{k=0}^{FB_a-1} 2^{k-FB_a} a_k) * ((1-2b_{FB_b+3})\sum_{i=0}^{2} 2^i b_{FB_b+i} + \sum_{k=0}^{FB_b-1} 2^{k-FB_b} b_k)$$

$$AT(e) = AT(d) + AT(c)$$

$$= ((1-2d_{FB_d+5})\sum_{i=0}^{4} 2^i d_{FB_d+i} + \sum_{k=0}^{FB_d-1} 2^{k-FB_d} d_k) + ((1-2c_{FB_c+2})\sum_{i=0}^{1} 2^i c_{FB_c+i} + \sum_{k=0}^{FB_c-1} 2^{k-FB_c} c_k)$$

The AT size of the datapath is calculated by:

$$|AT(f)| = |AT(d)| + |AT(e)| \qquad (8\text{-}4)$$

It requires the smallest $|AT(f)|$ to obtain the optimized implementation with the smallest area. The uniform FBs for all variables are $FB_a = FB_b = FB_c = FB_d = FB_e = 10$ and the maximum error of the primary output is represented as Eqn. (8-2):

$$max(E_e) = 2.1*2^{-FB_b} + 2.8*2^{-FB_a} + 0.25*\ 2^{-FB_a-FB_b} + 0.5*\ 2^{-FB_d}$$
$$+\ 0.5*\ 2^{-FB_c} + 0.5*\ 2^{-FB_e}$$

A searching algorithm is proposed in Figure 6.6. Observing the Eqn. (8-4), $FB_a$ and $FB_b$ have more impact on the AT size than $FB_c$ and $FB_d$. Hence, starting from the uniform FBs, that is, $FB_a = FB_b = FB_c = FB_d = FB_e = 10$, the algorithm first decreases $FB_a$ and computs the AT size, until the calculated error of $e$ is beyond the error bound. Then the algorithm backtracks to search $FB_b$. After all possible implementations are found, the algorithm compares their AT sizes, and the implementation with the smallest AT size is the best one. In Example 8.1, the satisfying sequence is (each value in a bracket represents a variable FB):

(10, 10, 10, 10, 10)$\rightarrow$ (9, 10, 10, 10, 10)$\rightarrow$(8, 11, 11, 12, 13) $\rightarrow$ (8, 11, 11, 13, 12)$\rightarrow$(8, 11, 12, 11, 13)$\rightarrow$(8, 11, 12, 13, 11)$\rightarrow$ (9, 9, 11, 12, 13)$\rightarrow$ (9, 9, 11, 13, 12)$\rightarrow$(9, 9, 12, 11, 13)$\rightarrow$(9, 9, 12, 13, 11)

The above implementations all satisfy the error bound. By calculating their AT sizes, the implementation of (9, 9, 11, 12, 13) has the smallest AT size, so it is the optimized implementation. Finally, the bit-width allocation of the optimized implementation is:

*a (4, 9)     b (4, 9)     c (3, 11)     d (6, 12)     e (6, 13)*

The first value is IB including the sign bit and the second is FB in the bracket.

## 8.3 Algorithm for Finding Optimized Implementations

We propose an algorithm to allocate bit-widths for all variables in the datapath to satisfy the given error bound and get the optimized implementation with the smallest area in terms of the above analysis in this section.

**Problem 8.1**: *Finding the optimized implementation for a fixed-point datapath*

| | |
|---|---|
| Inputs: | *imp, E* |
| Constraints: | *imprecision < E* |
| Outputs: | *bit-widths of all variables* |
| Goal: | *minimum* $\vert AT(f)\vert$ |

The inputs of the algorithm comprise the datapath structure and the error bound. The constraint restricts that the error of the primary output cannot break through the error bound. The AT size of the datapath is used as an indicator to the area, and the optimized implementation demands the smallest size.

```
Design_Opt_Imp (imp, E)
1.  {  IBs = Calculate_Range (imp);
2.     Construct expression $e$ of the primary output;
3.     $FB$ = Uniform_FB ($e$);
4.     Construct expression of AT size $|AT(f)|$;
5.     Determine the searching order $V$;
6.     for (i=0; i< var_num; i++)
7.     {  $e$ = Calculate_error (--$FB_{vi}$);
8.        if ($e < E$)    continue;
9.        else   ++ $FB_{vi}$;
10.       Re-compute FBs of other variables;
11.       $|AT(f)|$ = Calculate_AT_size ($AT(f)$, FBs);
12.       Store (FBs, $|AT(f)|$ );
       }
13.    Compare ($|AT(f)|$ );
14.    return $FB_{opt}$;
    }
```

**Figure 8.4: Algorithm of finding the optimized fixed-point implementation**

Figure 8.4 describes the algorithm. It first invokes the algorithm introduced in Chapter 7 to get exact ranges of all variables, and allocates IBs (Step 1). Then the algorithm constructs the expression of the primary output and gets the uniform FBs (Step 2 and 3). After that, the AT size expression is obtained (Step 4). By analyzing the expression, the algorithm determines the variable searching order (Step 5). A loop begins in Step 6 in terms of the searching order, and decreases the variable FB with highest priority and calculates the error until the error is beyond the error bound (Step 6 - 9). Then, FBs of other variables will be updated (Step 10). The algorithm calculates the AT size, and stores it for the obtained satisfying FBs (Step 11 and 12); while the loop is finished, all AT sizes are compared to find the smallest one, so the optimized FBs are found.

***Example 8.3:*** *Starting from the first group with the uniform FBs (10, 10, 10,* *10, 10), Figure 8.5 describes how to find the satisfying group (8, 11, 11, 12, 13).*

*The $FB_a$ is first decreased to get the group (9, 10, 10, 10, 10) and the calculation of the error is within the error bound, so the new group is satisfying. Then the algorithm continues to cut down $FB_a$ and finds that the group (8, 10, 10, 10, 10) cannot satisfy the error bound. However, the error caused by $FB_a$ is within the error bound, so the algorithm increases $FB_b$ to get the group (8, 11, 11, 10, 10). The new group does not satisfy the error bound, but the error addition caused by $FB_a$ and $FB_b$ is smaller than the bound, then $FB_c$ is increased to form the group (8, 11, 11, 11, 10). The procedure is continued until the group (8, 11, 11, 12, 13) is reached. Since the error is limited in the bound, the group satisfies the error bound. The searching process is repeated until all satisfying groups are found. Figure 8.5 lists all the traversed groups and the satisfying groups are marked by gray color.*

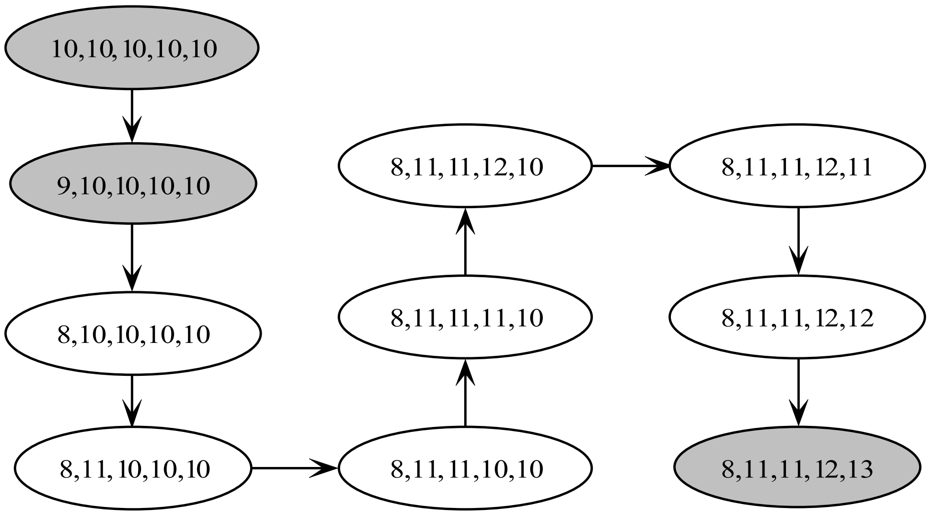


**Figure 8.5: Finding next satisfying FBs**

If the coefficients also have fractional numbers, we can use the same multiplication analysis like $\tilde{d}$ to process precision and search them together with other variables, so in the datapath all fixed-point variables can be allocated appropriate bit-widths to get the optimized implementation.

## 8.4 Discussion of Cost Functions

In above analysis, AT size is a cost function to estimate hardware cost and choose the optimized implementation. There are other cost functions besides AT size. The usual one employs factorization. Given a polynomial to describe the specification, factorization allows us to find the optimized implementation with the smallest area. For example, the polynomial of $c = ab + b^2$ has two word-level variables $a$ and $b$. The direct implementation needs two multipliers and one adder. However, if using factorization method to change the form as $c = b(a+b)$, the implementation only needs one multiplier and one adder. In this example, factorization plays as a cost function to shrink the number of multipliers.

However, factorization has an obvious disadvantage. For example, given two implementations of Taylor series with the first implementation having 5 finite terms and 12-bit inputs, and the other with 6 terms and 10-bit inputs. They both have Horner forms and structures as Figure 6.11. Although the second implementation has one more stage, the input bit-width is smaller, that is, the multiplier size is 10*10 and smaller than the first one with multiplier size of 12*12.

The case generates a problem: which factor has more impact on area, stage or multiplier size? The cost function of factorization (counting the number of multipliers) cannot answer the question because it is too coarse to estimate the cost. That is the reason why we use AT size as a cost function in Taylor series.

More commonly, given a specification represented by a polynomial, it can be minimized by many ways. Factorization is one possibility. However, as there is not a unique answer how to conduct factorization, we must be very careful here, as different approaches may have different multiplier sizes. For instance, using factorization needs a 12*12 multiplier, and another implementation needs two multipliers as 6*6 and 9*9, so the question is how to determine which implementation is better? Of course AT size can solve the problem. So there is a prerequisite to use factorization as a cost function, that is, all implementations must keep same size of multipliers.

It is possible to combine factorization and AT size. Consider an example: $d = ab + b^2 + ac$ with different bit-widths of $a$, $b$ and $c$. There are three implementations:

- Direct implementation with 3 multipliers and 2 adders
- Factorization by *b*: $d = b\,(a+b) + ac$ with 2 multipliers and 2 adders
- Factorization by *a*: $d = a(b+c)+b^2$ with 2 multipliers and 2 adders

The first one may need more area so factorization is possibly leading to the optimized implementation. Consider the latter two implementations. They have same numbers of multipliers and adders. Comparing AT sizes in the two implementations, we can choose the one with smaller AT size. That means, factorization and AT size can play together. In this case factorization is a coarse cost function and then using AT size refines it.

## 8.5 Sequential Fixed-Point Circuits

The above analysis and past explorations of fixed-point representations are based on combinational circuits. Given a datapath with FFs like Figure 8.6, the analysis of range and precision depends on the lengths of the FFs.

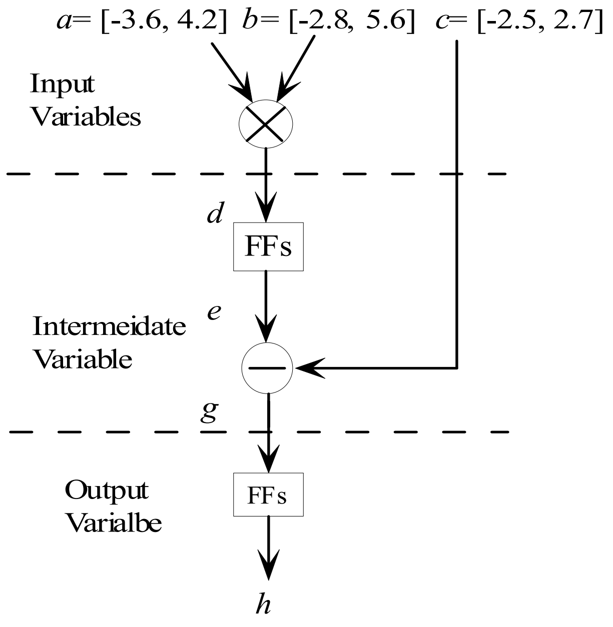


**Figure 8.6: A sequential datapath with FFs**

If the lengths of the FFs equal to the lengths of their inputs, that is, ($d_{IB} = e_{IB}$,

$d_{FB}= e_{FB}$) and ($g_{IB}= h_{IB}$, $g_{FB}= h_{FB}$), the analysis of range and precision is the same as the combinational circuit without FFs. Otherwise, the sequential circuit may cause overflow, and the analysis expression is different with the combinational circuit. For instance, *d* and *e* are two different variables so they have their own precision expressions. Therefore, the analysis of sequential fixed-point designs has no special essence.

## 8.6 Extension to Feedback Datapaths

Past explorations cannot process the datapaths with feedbacks. The usual datapaths with feedbacks are IIR (infinite impulse response) filters which apply to DSP. In this section, we propose algorithms to find ranges of circuits with feedbacks.

### 8.6.1 Delay Units

A feedback circuit always includes delay units that consist of registers, so analyzing the characteristic of delay units is the first step. Figure 8.7 describes the relationship of the input range and the output range.

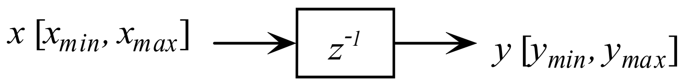


**Figure 8.7: A delay unit with ranges**

Since the delay unit only has the shift operation and cannot change the input value, its output keeps the same range as the input range, that is, $y_{min} = x_{min}$ and $y_{max} = x_{max}$. Here $x_{min}$ and $x_{max}$, $y_{min}$ and $y_{max}$ represent the lower bounds and the upper bounds of the input and the output respectively.

### 8.6.2 FIR Filters

First, we explore FIR (finite impulse response) filters. The impulse response is *finite* because it settles to zero in a finite number of sample intervals. The difference equation of Eqn. (8-5) defines the output of an FIR filter based on the input:

$$y[n] = \sum_{i=0}^{N} h_i x[n-i] \tag{8-5}$$

where *x[n]* is the input signal, $h_i$ are the filter coefficients and *N* is the filter order which are commonly referred to as *taps*. The *Z*-transform of the impulse response yields the transfer function of the FIR filter:

$$H(z) = \sum_{n=0}^{N} h_n z^{-n}$$

FIR filters are inherently stable because all the poles are located within the unit circle. The absence of feedbacks means that any rounding errors are not compounded by summed iterations. The same relative error occurs in each calculation which makes implementation simpler. The main disadvantage of FIR filters is that a lot of taps cause considerably more computation especially when low frequency (relative to the sample rate) cutoffs are needed. Figure 8.8 describes an implementation of the FIR filter with *k+1* taps.

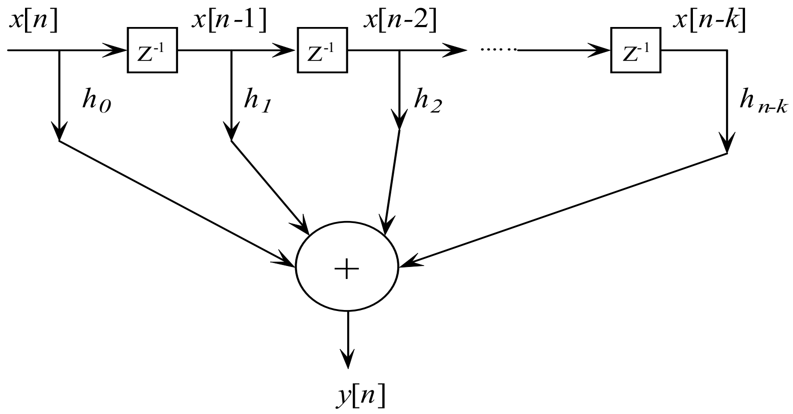


**Figure 8.8: Implementation of the FIR filter with k+1 taps**

Given the range of the input *x*, calculating ranges of intermediate variables and the primary output is easy. The ranges of the intermediate variables are calculated by the multiplication of the coefficients and the range of the primary input, and the range of the primary output is calculated by the addition of intermediate ranges.

***Example 8.4:*** *The following circuit is a FIR filter with three taps. All ranges* *are described in the square brackets.*

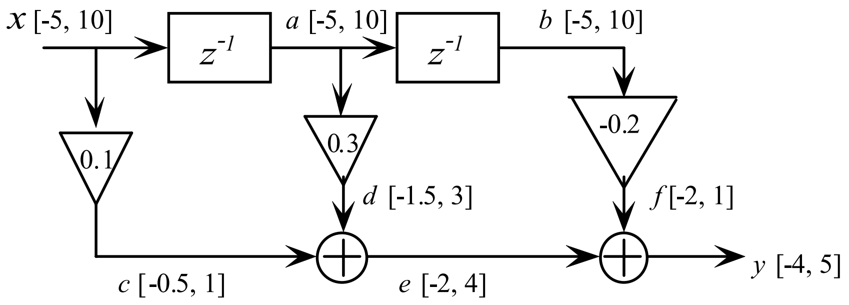


**Figure 8.9: Ranges of a FIR filter**

*In the figure, the delayed variables a and b have the same ranges as the primary input x. The ranges of the intermediate variables c, d and f equal to the range of x multiplying the tap coefficients, and the range of the primary output y equal to the summation of the ranges of e and f.*

## 8.6.3 Linear Feedbacks – IIR Filters

Calculating the ranges of FIR filters without feedbacks is a simpler task compared to the much more complex case of IIR filters. IIR systems have an impulse response function that is non-zero over an infinite length of time. A condensed form of the difference equation is:

$$y[n] = \sum_{i=0}^{R} b_i x[n-i] + \sum_{j=1}^{S} a_j y[n-j] \tag{8-6}$$

where $R$ is the feedforward filter order, and $b_i$ are the feedforward filter coefficients; $S$ is the feedback filter order, and $a_i$ are the feedback filter coefficients. The $Z$-transform of the impulse response yields the transfer function of the IIR filter:

$$H(z) = \frac{\sum_{i=0}^{P} b_i z^{-i}}{1 - \sum_{j=1}^{Q} a_j z^{-j}}$$

The first part in Eqn. (8-6) is the same as Eqn. (8-5) so the ranges are easy to find. We focus on the second part as feedbacks possibly leading to unstability, meaning that the range of the output is not convergent and will increase (or decrease) to infinity (or become infinitesimal).

***Example 8.5:*** *The following circuit has a feedback. The primary input x is limited in the range [-5, 10], and the output z has the expression of z =2(x+ $z^{-1}$). It is obvious that the circuit is unstable since the range of z has no limitation.*

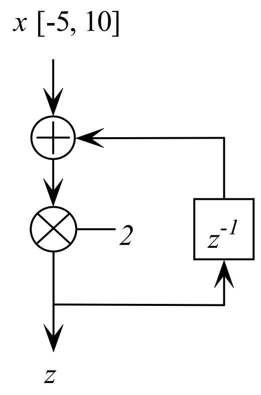


**Figure 8.10: A circuit with a feedback**

*Now we analyze why the circuit is unstable. We assume the circuit is stable and the range of z is [$r_0$, $r_1$] ($r_1 > r_0$). In terms of the above analysis of delay units, $z^{-1}$ has the same range of z and they are considered as same variables since $z^{-1}$ is driven completely by z, so the expression representing the datapath is:*

$$2(2.5+7.5\varepsilon_1+\frac{r_1+r_0}{2}+\frac{r_1-r_0}{2}\varepsilon_2)=\frac{r_1+r_0}{2}+\frac{r_1-r_0}{2}\varepsilon_2$$

$$\Rightarrow \qquad 5+15\varepsilon_1 = -\frac{r_1+r_0}{2}-\frac{r_1-r_0}{2}\varepsilon_2$$

*Since the assumption requires the convergence of z, the parts with certainty and the parts with uncertainty in the left and the right of the above equation should equal respectively:*

$$-\frac{r_1+r_0}{2}=5 \qquad\qquad -\frac{r_1-r_0}{2}=15$$

*By solving the two equations, we obtain $r_1$= -10 and $r_0$=20. The results violate the assumption $r_1 > r_0$ so the circuit is unstable and the output has no convergent range.*

Example 8.5 describes how to explore whether the circuit with linear feedbacks is stable by AA. Now we amend the multiplicand coefficient in the Figure 8.10 to re-calculate the output range.

***Example 8.6:*** *The output z has the expression of $z = 0.25(x+ z^{-1})$.*

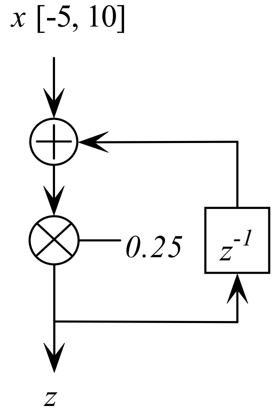


**Figure 8.11: A circuit like Example 8.5 with the different coefficient**

*We get the expression by AA forms:*

$$0.25(2.5+7.5\varepsilon_1+\frac{r_1+r_0}{2}+\frac{r_1-r_0}{2}\varepsilon_2)=\frac{r_1+r_0}{2}+\frac{r_1-r_0}{2}\varepsilon_2$$

$$\Rightarrow\quad \frac{5}{8}+\frac{15}{8}\varepsilon_1=\frac{3(r_1+r_0)}{8}+\frac{3(r_1-r_0)}{8}\varepsilon_2$$

$$\frac{3(r_1+r_0)}{8}=\frac{5}{8}\qquad\qquad \frac{3(r_1-r_0)}{8}=\frac{15}{8}$$

*By solving the equations, we get $r_1 = \frac{10}{3}$ and $r_0 = -\frac{5}{3}$. The results fit the assumption of $r_1 > r_0$ that denotes the circuit is stable. Using this initial range to replace the unknown variable $z^{-1}$ in the polynomial $0.25(x+ z^{-1})$ gets the final output range $[-\frac{5}{3}, \frac{10}{3}]$ which is the same as the initial range. The experiment proves the circuit is convergent to the range.*

Based on the two examples, we propose a method in Figure 8.12 to explore whether IIR is stable and calculate the ranges in the datapath if stable. It uses AA forms to express the implementation, and partitions the forms into parts of certainty and uncertainty after simplification (Step 2 - 4). Here $C_L$ and $C_R$ represent the certainty expressions in the left and the right of the AA form while $U_L$ and $U_R$ are the uncertainty expressions. The initial range is obtained by solving the equations of certainty and uncertainty (Step 5). If the condition $r_1 > r_0$ is satisfied, the initial range replaces the unknown feedback variable and the algorithm re-calculates the final output range (Step 7). Please note that the Step 7 is necessary since the initial range may under-estimate the bounds so it needs refinement.

```
Find_Linear_Range (imp)
{
1.  Assume the range ($r_0$, $r_1$);
2.  AA_form = AA_Express (imp, $r_0$, $r_1$);
3.  Simplify (AA_form);
4.  ({$C_L$, $U_L$}, {$C_R$, $U_R$}) = Partition (AA_form);
5.  ($r_0$, $r_1$) = Solve ($C_L = C_R$, $U_L = U_R$);
6.    if ($r_1 < r_0$)     return "The circuit is unstable!";
    else
7.    {  ($r_0$, $r_1$) = AA_Range (imp, $r_0$, $r_1$);
         return range ($r_0$, $r_1$);
      }
}
AA_Express (imp, $r_0$, $r_1$)
{    Using AA to replace known inputs;
     $AA_{out} = (r_1 + r_0)/2 + \varepsilon (r_1 - r_0)/2$ ;
     Replace all feedback variables with $AA_{out}$;
     return AA_form;
}
AA_Range (imp, $r_0$, $r_1$)
{    loop all uncertain terms in the expression
     {  if (term.coeff < 0)          uncertain_var = -1;
        else    uncertain_var = 1;
        $r_1$ += term.coeff * uncertain_var;
     }
     $r_0 = -r_1 +$ constant;    $r_1$ += constant; return ($r_0$, $r_1$)
}
```

**Figure 8.12: Algorithm of finding ranges of IIR filters**

***Example 8.7:*** *An IIR filter is described in the Figure 8.13. It has two taps with coefficients 0.2 and -0.3.*

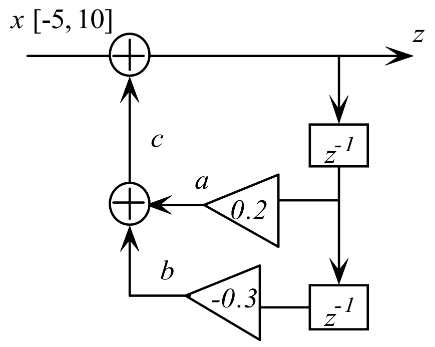


**Figure 8.13: An IIR filter with two taps**

*The expression of the IIR filter is:* $x + 0.2 * z^{-1} - 0.3 * z^{-2} = z$

*Using the AA form of z replaces* $z^{-1}$ *and* $z^{-2}$ *to get the representation in terms of step 2*

*in Figure 8.13:*

$$2.5 + 7.5\varepsilon_1 + 0.2 * (\frac{r_1 + r_0}{2} - \frac{r_1 - r_0}{2}\varepsilon_2) - 0.3 * (\frac{r_1 + r_0}{2} - \frac{r_1 - r_0}{2}\varepsilon_2) = \frac{r_1 + r_0}{2} + \frac{r_1 - r_0}{2}\varepsilon_2$$

$$2.5 + 7.5\varepsilon_1 = \frac{11(r_1 + r_0)}{20} + \frac{11(r_1 - r_0)}{20}\varepsilon_2$$

$$\frac{11(r_1 + r_0)}{20} = 2.5 \qquad \frac{11(r_1 - r_0)}{20} = 7.5$$

*The results are* $r_1 = 9.09$ *and* $r_0 = -4.56$ *so the filter is stable and the output has the convergent range. The initial range replaces the unknown variables* $z^{-1}$ *and* $z^{-2}$ *by the AA form* $2.27 + 6.83\,\varepsilon_2$ *in the expression of* $x + 0.2 * z^{-1} - 0.3 * z^{-2}$*, so the polynomial changes to:*

$$2.5 + 7.5\varepsilon_1 + 0.2 * (2.27 + 6.83\varepsilon_2) - 0.3 * (2.27 + 6.83\varepsilon_2) \Rightarrow 2.273 + 7.5\varepsilon_1 - 0.683\varepsilon_2$$

*The coefficient of the term "*$7.5\varepsilon_1$*" is positive so the algorithm sets* $\varepsilon_1 = 1$ *while sets* $\varepsilon_2 = -1$ *to get* $r_1$*. Based on Step 7, the final output range is re-calculated as [-5.91, 10.46]. The ranges of the two intermediate variables can be calculated by the coefficients of taps as a=[-0.1.82, 2.09], b=[-3.14, 1.77]. The range of the intermediate variable c cannot be calculated directly by the range addition of a and b because the two variables are both driven by z so present correlation leads to a coarse range. Using IA to calculate the range of c by range subtraction of z and x obtains [-0.91, 0.45]. By experiments, the output range is [-5.901, 10.447], after 14 iterations and the experiments prove the correctness of the calculated results.*

***Example 8.8:*** *An IIR filter is described like Example 8.7 in the Figure 8.13. It has two taps with coefficients 0.2 and 0.3.*

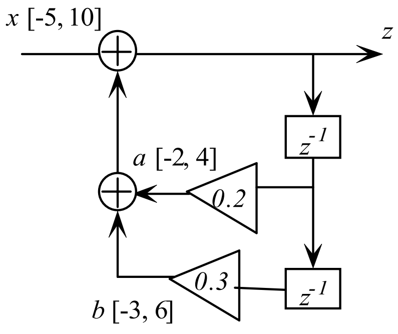


**Figure 8.14: An IIR filter like Example 8.7 with different coefficients**

*The expression of the IIR filter is:* $x + 0.2 * z^{-1} + 0.3 * z^{-2} = z$

*The representation of AA forms is:*

$$2.5 + 7.5\varepsilon_1 + 0.2 * (\frac{r_1 + r_0}{2} - \frac{r_1 - r_0}{2}\varepsilon_2) + 0.3 * (\frac{r_1 + r_0}{2} - \frac{r_1 - r_0}{2}\varepsilon_2) = \frac{r_1 + r_0}{2} + \frac{r_1 - r_0}{2}\varepsilon_2$$

$$\Rightarrow \quad 2.5 + 7.5\varepsilon_1 = \frac{r_1 + r_0}{4} + \frac{r_1 - r_0}{4}\varepsilon_2$$

$$\frac{r_1 + r_0}{4} = 2.5 \qquad\qquad \frac{r_1 - r_0}{4} = 7.5$$

*The results are $r_1 = 20$ and $r_0 = -10$ so the filter is stable and the output has the convergent range. Using the initial range to replace the unknown variables $z^{-1}$ and $z^{-2}$ by the AA form $5 + 15\varepsilon_2$ in the polynomial of $x + 0.2 * z^{-1} + 0.3 * z^{-2}$, and the final output range is re-calculated as [-10, 20] which is the same as the initial range. By experiments, the output range is [-9.96, 19.92], and the two intermediate variables a and b have ranges [-1.98, 3.978] and [-2.96, 5.95] respectively after 13 iterations. The experimental results are quite suitable to the calculated results.*

### 8.6.4 Non-linear Feedbacks

Consider a circuit with a non-linear feedback in the Figure 8.15. The expression is $z = x + (0.25 * z^{-1})^2$.

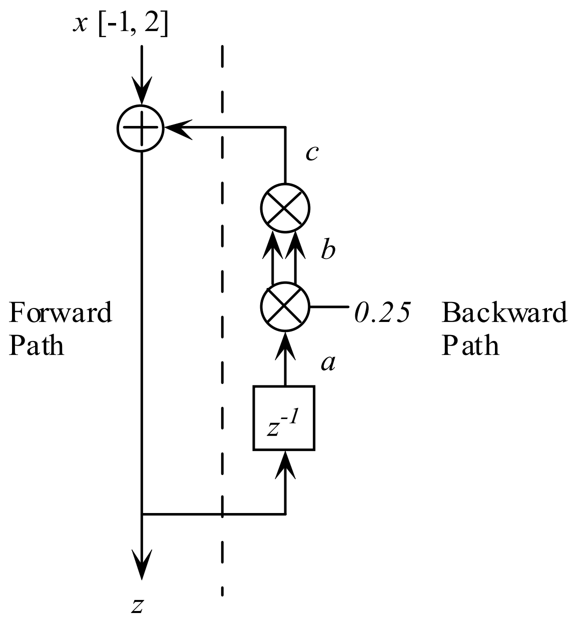


**Figure 8.15: A circuit with a non-linear feedback**

If using the above method processing the non-linear feedback, we obtain:

$$64 = 32r_1 + 32r_0 - (r_1 + r_0)^2$$

$$128 = 32(r_1 - r_0) - 2(r_1^2 - r_0^2) - (r_1 - r_0)^2$$

Obviously solving the two equations is difficult, so we need to develop a new method to handle the circuits with non-linear feedbacks. First we introduce a lemma.

**Lemma 8.1:** *If all intermediate variables have convergent ranges, the primary output is also convergent; vice versa, if the primary output has a convergent range, all intermediate variables are convergent.*

*Proof: Convergent ranges of input passing through basic operations of multiplication and addition in a datapath results in a convergent output, that is,*

$$convergence \times convergence \to convergence$$

$$convergence + convergence \to convergence$$

*So traversing the entire datapath creates a convergent primary output.*

In Figure 8.15, we assume the primary output $z$ is convergent. We split the datapath into the forward path and the backward path, and the feedback is included in the backward path. By Lemma 8.1, the variables $a$ and $b$ should be both convergent. The expression of the non-linear variable $c$ is $c = b^2$. Based on the knowledge of power series, when the range of $b$ lies in [-1, 1], the variable $c$ obtains the range [0, 1] and forms a closure space to $b$, that is, $range(c) \subseteq range(b)$, to guarantee convergence of the non-linear feedback. By the addition of $x$ in the forward path, we obtain that the range of $z$ is [-1, 3] labeled as $z_{forward}$. We go back to the variable $a$ from $b$, and conclude that the range of $a$ is [-4, 4], and then we obtain that the range of $z$ is [-4, 4] labeled as $z_{backward}$. The convergence requires the condition of $z_{forward} \subseteq z_{backward}$ because if the condition is violated, the real range of $z$ will increase in each iteration and ultimately reach infinity (or infinitesimal). Now the ranges of $z_{forward}$ and $z_{backward}$ satisfy the condition, we confirm that the circuit is stable.

The different ranges of $z_{forward}$ and $z_{back}$ denote that the obtained ranges are coarse and they need to be refined. Let $z = z_{forward}$ then a loop calculation of $z$ starts. Each loop begins to go through the backward path to get the range of $c$, and then follows the forward path to obtain the new range of $z$. The threshold value "0.01" is set. In two consecutive iterations, if the error of the two obtained ranges is smaller than the threshold, that is, $|z_{new} - z_{old}| < threshold$, the loop calculation is stopped. In this example, after four loops the threshold condition is reached, so finally we get the convergent range of $z$ as [-0.944, 2.341]. Figure 8.16 describes the algorithm to find ranges for circuits with non-linear feedbacks.

The algorithm first splits the datapath to two sub-paths as the forward path and the backward path. The coarse range of the feedback variable is calculated in terms of the non-linear feedback expression. Then two ranges of the output are obtained due to the forward path and the backward path by the subroutine `Calculate_range` introduced in Chapter 7. Comparing the two ranges, if the circuit is stable, the algorithm starts a loop calculation until the error between the two consecutive obtained ranges is limited in the threshold. Therefore, the convergent range of the primary output is found.

```
Find_Nonlinear_Range (imp, threshold, input_range)
{
    (forward_path, backward_path) = Split (imp);
    feedback_range = Converge (feedback_expression);
    $z_{forward}$ = Calculate_range (forward_path, input_range);
    $z_{backward}$ = Calculate_range (backward_path, feedback_range);
    if ( $z_{forward} \not\subset z_{backward}$ )    return "The circuit is not stable.";
    else
    {   $z_{new} = z_{forward}$;
      while (|$z_{new} - z_{old}$|  $\geq$  threshold )
      {   $z_{old} = z_{new}$;
          feedback_range = Calculate_Range (backward_path, $z_{old}$);
        $z_{new}$ = Calculate_Range (forward_path, input_range, feedback_range);
      }
      return $z_{new}$;
}
```

**Figure 8.16: Algorithm of finding ranges of circuits with non-linear feedbacks**

## 8.6.5 Experimental Results

We implement the algorithm in Figure 8.12 in C++. Several benchmarks are sued to assess its performance. Experiments are done on a 512MB, 2.4GHz Intel Celeron machine under Linux. Using the variable $y$ represents the first part in Eqn. (8-5) and the primary output is $z$.

### *A) Butterworth Filters*

Butterworth filters are also known as "maximally flat" filters because they have no passband ripple. They also have a monotonic response in both the stopband and passband. The indicators of ($w_p$, $a_p$, $w_s$, $a_s$) represent passband frequency, amplitude error, stopband frequency and stopband attenuation.

The first Butterworth filter has indicators (0.2$\pi$, 1dB, 0.35$\pi$, 10dB), and the coefficients from smaller orders to larger orders are:

$b$ = (0.0456, 0.1027, 0.0154)    $a$ = (1.9184, -1.6546, 0.6853, -0.1127)

The second Butterworth filter has indicators (0.2$\pi$, 3dB, 0.6$\pi$, 40dB), and the coefficients are:

$b$ = (0.0473, 0.0709, 0.0473, 0.0118)

$a$ = (1.8778, -1.6214, 0.663, -0.1087)

The third Butterworth filter is a bandpass filter which has indicators ((0.3$\pi$-0.4$\pi$), 3dB, (0-0.2$\pi$, 0.5$\pi$), 18dB), and the coefficients are:

$b$ = (-0.042, 0.021)  $a$ = (1.491, -2.848, 1.68, -1.273)

## *B) Chebyshev Filters*

Chebyshev filters are analog or digital filters having a steeper roll-off and more passband ripple or stopband ripple than Butterworth filters.

The first Chebyshev filter has coefficients:

$b$ = (9.055E-5, 0, -0.00027, 0, 0.00027, 0, -9.055E-5)

$a$ = (5.765, -13.899, 17.936, -13.067, 5.095, -0.831)

The second Chebyshev filter corresponds to the indicators (0.2$\pi$, 1dB, 0.3 $\pi$, 15dB) and has the coefficients:

$b$ = (0.0073, 0.011, 0.0073, 0.0018)

$a$ = (1.5548, -2.9809, 2.2925, -0.5507)

## *C) Cauer Filters*

A Cauer filter has a feature of equalized ripple behavior in both the passband and the stopband. The indicators of the Causer filter are (0.1$\pi$, 0.1dB, 0.5$\pi$, 32dB) and the coefficients are given:

$b$ = (-0.724, 0.0984, 0, 0.00027, 0,-9.055E-5)

$a$ = (3.3553, -4.3439, 2.5578, -0.5771)

| IIR | Input Range | Range of $y$ | Output Range $z$ | Time (s) | Memory (MB) |
|---|---|---|---|---|---|
| Butter | [-500, 1000] | [-81.85, 163.7] | [-511.9, 1023.1] | 0.12 | 0.16 |
| Butter | [-2000, 1000] | [-354.6, 177.3] | [-1970.3, 985.3] | 0.15 | 0.19 |
| Butter | [-5000, 10000] | [-210, 105] | [-2100, 4200] | 0.15 | 0.2 |
| Cheby | [-4E+5, 1E+6] | [-504.8, 504.8] | [-504800, 504800] | 0.26 | 0.25 |
| Cheby | [-3000, 2000] | [-82.2, 54.8] | [-120.9, 80.6] | 0.16 | 0.2 |
| Cauer | [-500, 800] | [-500.3, 312.7] | [-63309, 39602] | 0.18 | 0.17 |

**Table 8.1: Performance of the algorithm finding IIR ranges**

Table 8.1 describes the ranges of the benchmarks. Column 2 denotes the input ranges, and the intermediate ranges and the primary ranges are shown in Column 3 and 4. Column 6 describes the real obtained ranges by simulation after iterations whose number is listed in Column 5. Column 7 and 8 indicate the algorithm performance of time and memory. From the table, we can find that the real ranges approximate the calculated ranges very well, and the requirements of time and space are satisfiable. Using simulation will spend huge time by a lot of iterations such as Row 5 and is hard to determine the lower bound and the upper bound. However, the algorithm can complete the job very easily.

## 8.7 Extension to Floating-Point Circuits

If the radix point (decimal point, or, more commonly in computers, binary point) can "float", that is, it can be placed anywhere relative to the significant digits of the number, the representation refers to the term "*floating-point*". Because the position of the radix point is indicated separately in the internal representation, floating-point representation can thus be thought of as a computer realization of scientific notation.

The floating-point representation can support a much wider range of values than the fixed-point representation. For example, a fixed-point representation that has eight decimal digits, with the decimal point assumed to be positioned after the sixth digit, can represent the numbers 123450.67, 87654.32, 2345.00, and so on, whereas a floating-point representation (such as the IEEE 754 decimal32 format) with eight decimal digits could in addition represent 12.3456789, 123.4567, 0.0001234567, 1234567000000000, and so on. The floating-point format requires a little more storage (to encode the position of the radix point), so the floating-representation can achieve greater range at the expense of precision when stored in the same space.

Floating point numbers are used to obtain a dynamic range for representable real numbers without having to scale the operands. Floating point numbers are approximations of real numbers and it is not possible to represent an infinite continum of real data into precisely equivalent floating point value.

Logically, a floating-point number consists of [156]:

- A signed digit string of a given length in a given base (or radix). This is known as the *significand*, or sometimes the mantissa or coefficient. The radix point is implicitly assumed to always lie in a certain position within the *significand* — often just after the most significant digit. The length of the *significand* determines the precision to which numbers can be represented.

- A signed integer exponent is a scale to modify the magnitude of the number.

A floating point number system is completely specified by specifying a suitable base *β*, significand (or mantissa) *M*, and exponent *E*. A floating point number *F* has the value

$$F = M\beta^E$$

*β* is the base of exponent and it is common to all floating point numbers in a system. Commonly the significand is a signed - magnitude fraction. The floating point format consists of a sign bit *S*, *e* bits of an exponent *E,* and *m* bits of an unsigned fraction *M*, as shown below:

| | | |
|---|---|---|
| *S* | Exponent *E* | Unsigned Significand *M* |

The value of such a floating point number is given by:

$$F = (-1)^S M\beta^E$$

The most common representation of exponent is as a biased exponent, according to which $E = E^{true} + bias,$ where bias is a constant and $E^{true}$ is the true value of exponent. The range of $E^{true}$ using the e bits of the exponent field is:

$$-2^{e-1} \leq E^{true} \leq 2^{e-1} - 1$$

The bias is normally selected as the magnitude of the most negative exponent; i.e. $2^{e-1}$, so that

$$0 \leq E \leq 2^e - 1$$

When comparing two exponents, which is required in the floating point addition for example, the sign bits of exponents can be ignored and they can be treated as unsigned numbers. This is an advantage of using biased exponent.

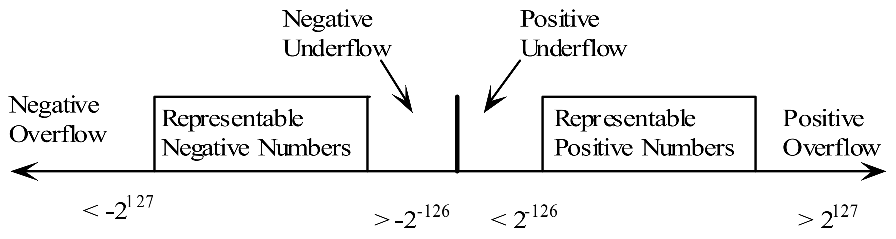


**Figure 8.17: Range of floating point numbers**

Not only cannot all real numbers be expressed exactly, there are whole ranges of numbers that cannot be represented. Consider the real number line as shown in Figure 8.17. The number zero can be represented exactly because it is defined by the standard. The positive numbers that can be represented fall approximately in the range $2^{-126}$ to $2^{+127}$.

Numbers greater than $2^{+127}$ cannot be represented; this is called **positive overflow**. A similar range of negative numbers can be represented. Numbers to the left of that range cannot be represented; this is **negative overflow**.

***Example 8.9:*** *S=0, E=3 bits, M = 4 bits. Then the bias is $2^{E-1}-1=3$. The maximum range is:*

| | | |
|---|---|---|
| *0* | 1 1 1 | 1 1 1 1 |

$$(-1)^0\ 1.1111\ 2^{7-3} = 1.1111\ 2^4 = 11111 = 31_{10}$$

*The minimum range, assuming exponent 000 is reserved for zero.*

| | | |
|---|---|---|
| *0* | 0 0 1 | 0 0 0 0 |

$$(-1)^0\ 1.0000\ 2^{1-3} = 1.0000\ 2^{-2} = 0.01 = 0.25_{10}$$

The precision of floating-point numbers is not like fixed-point numbers which have uniform error as $2^{-FB-1}$. The error in each exponent value is different. Figure 8.18 describes the error with non-uniform distribution for Example 8.9.

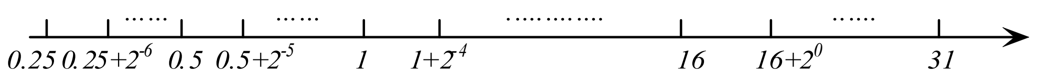


**Figure 8.18: Non-uniform distribution error in floating-point representation**

In the figure, there are $2^M = 16$ values in each exponent interval, and the smallest error is $2^{-7}$, that is, $2^{-(2^{E-1}-1+M)}$ in the left axis, while the largest error is

$2^{2^{E-1}-1-M}$ = $2^{-1}$ in the right axis. We can obtain the expression of each interval error as $2^{c-bias-1-M}$. Here *c* is the coded value of the interval, and *bias* is calculated as $2^{E-1}-1$. For example, the interval "2" includes values from *2* to $2+15*2^{-3}$. "2" is coded as "100" and bias is "011", so they correspond to the values of "4" and "3" respectively. The interval error is calculated as $2^{4-3-1-4}$ = $2^{-4}$.

In terms of the above analysis, we can perform range and precision analysis for floating-point circuits. The range analysis is the same as the Chapter 7, and the hybrid method is also suitable for floating-point circuits to find exact ranges. Precision analysis is a bit different with the fixed-point circuits. Given the input range as $[r_1, r_2]$, Eqn. (8-1) represents the relation between the exact value and the real value for fixed-point circuits. Since the floating-point representation has no uniform distribution error, the coefficient of the uncertain variable must set the largest error value:

$$\tilde{a} = a + 2^{c-bias-1-M}\,\varepsilon$$

Here *c* is chosen the larger coded value in the two intervals of $r_1$ and $r_2$, that is, if $|r_1| > |r_2|$, we choose the interval coded value of $r_1$; if not, we choose the interval coded value of $r_2$.

***Example 8.10:*** *The floating-point representation is as Example 8.9. The input range is a= [-7.5, 13]. Since the absolute values of the lower bound and the upper bound are 7.5 and 13 respectively, we choose the interval value of 13. Because the value "13" is located in the interval of "8", the interval coded value is 110 as c=6, so the coefficient of the uncertain variable is* $2^{c-bias-1-M}$ = $2^{-2}$*. The expression of the exact input value is changed to* $\tilde{a} = a + 2^{-2}\varepsilon$.

After we amend the input expression, the method of performing precision analysis in section 8.2 can also be used for floating-point datapath. So we extend the fixed-point process to the floating-point process.

## 8.8 Conclusions

Fixed-point representations often comprise integer and fractional bit-widths. The problems of exploring fixed-point circuits include range analysis and precision analysis. Since the circuits cannot get the exact fractional numbers, the satisfying implementation must fit the error bound, that is, the maximum error of the primary output is restricted by the bound. In order to find the attractive optimized implementation with the smallest area, it is necessary to obtain ranges and construct the precision models. The AT size plays an indicator to describe area. We propose an algorithm to find the optimized implementation in this chapter. It invokes the algorithm in Chapter 7 to get ranges and allocates IBs, and then calculates uniform FBs. Starting from the FBs, the algorithm searches all satisfying implementations and calculates their AT sizes. The implementation with the smallest AT size is the optimized one that can fit the error bound and have the smallest area.

The circuits with feedbacks are more complex to find ranges like IIR filters. We handle FIR filters without feedbacks only with delay units, and then propose a method to process IIR filters with linear feedbacks. The method can explore whether IIR filters are stable and calculate the ranges if stable. Furthermore, we analyze the circuits with non-linear feedbacks.

Sequential datapaths with FFs are investigated to extend combinational models based on previous chapters. Floating-point representation is different with non-uniform error distribution. We analyze floating-point representation and develop the mathematical models for error distribution, then extend the methods processing fixed-point representation to the floating-point datapath.

# Chapter 9 Conclusions and Future Work

## 9.1 Conclusions

As the complexity of integrated circuit increases rapidly, the challenge of time-to-market arises. In the overall design procedure, verification plays a significant role since it concentrates on most steps from system specification to manufacturing. Verification often requires beyond 70% time and capital in the whole ASIC design process. Because of its importance, engineers are forced to explore verification techniques. Simulation as a main technology has advantages of easy operation but low efficiency is the fatal weakness, so formal verification emerged. Various bit-level and word-level decision diagrams adapt to equivalence checking and model checking.

Fixed-point data format is suitable for a number of implementations of digital circuits. Traditional methods of dealing with imprecise fixed-point circuits have disadvantages in both verification and optimization. In our exploration, we adopt a spectral technique, that is, Arithmetic Transform, to investigate fixed-point circuits. Basic AT only represents combinational circuits, so three transform extensions have been proposed. The total four types of transforms form a complete group to represent complex combinational and sequential circuits, and every circuit can be represented by one type. Because obtaining a circuit transform is a significant step for verification, various spectral transformation methods have been explored. The most straightforward method relies on matrix multiplication, and a fast algorithm has been proposed. These methods all compute the transform directly. We design a new algorithm to obtain transform of a complex circuit by composing transforms of detached blocks in the circuit. It is a method based on traversing the sub-block topology, to provide an efficient way to get the transforms for complex arithmetic circuits.

The fixed-point representation often includes IB and FB. First, we explore them separately. As a big category, imprecise circuits need to be explored carefully. They are different with common circuits because they have a feature that the implementations do not match the specifications exactly, so decision diagrams have no capability to handle them. Many methods have been developed. Dynamic analysis based on simulation is usually used to investigate range and static analysis is applied such as IA and AA to avoid its disadvantage. They primarily handle optimization of input bit-width but do not consider other factors, so AT is introduced in the work to make up the weakness.

We explore imprecise circuits such as ones realizing Taylor series-based algorithms, and construct mathematical expressions for each imprecise factor due to AT representations. A series of algorithms that can process function approximation and bit-widths concurrently and handle Taylor series and real-valued polynomial with multiple variables are designed for verification and optimization due to various constraints.

Imprecise circuits do not confine the utilization of AT. We develop a fast and accuracy-guaranteed method to perform range analysis for arithmetic circuits by mixed techniques. The method can find the maximum value and the minimum value for each intermediate output in the datapath in terms of given input ranges, and allocate the smallest bit-width. Since the method does not extend the range and handles polynomials statically, it can obtain exact ranges, and avoid low efficiency simulation. The obtained smallest bit-widths lead to the optimized implementation with the smallest area.

Finally, we combine range and precision together. In the datapath of fixed-point representation, given the error bound, the most important problem is confirming the bit-widths include IBs and FBs for all variables. The appropriate bit-widths must fit the error bound, and lead to the implementation with the smallest area. We propose an algorithm to solve the problem. It can allocate the smallest IBs, and find non-uniform FBs to satisfy the error bound and obtain the optimized implementation with the smallest area.

## 9.2 Future Work

Exploring range value and component difference are always hot topics. They refer to circuit optimization with smaller area or faster speed and keep attracting engineers. We resolve the problem for fixed-point circuits and obtain good results. In the future, we will continue to explore optimized implementations in different constraints, and extend the method to process floating-point circuits and more subtle error models will be investigated.

In the area of formal verification, the derived transformations for compositional verification encourage verification for Intellectual Property (IP) cores. It would be helpful if a set of appropriate benchmarks are devised to quantify the quality of such methods. Additionally, although highly promising, AT might not be the only transformation that is appropriate for the formal verification applications presented in this thesis. The greatest opportunities in verification lie in the combination of the two approaches: simulation-based and formal. A study of suitable data structures and their concrete implementations would complement the research presented here.

# References

[1] Jacob Abraham; “Hardware Verification - Application of formal techniques to chip designs”, University of Texas

[2] Evans, A.; Silburt, A.; Vrckovnik, G.; Brown, T.; Dufresne, M.; Hall, G.; Tung Ho; Ying Liu; **“**Functional verification of large ASICs”, *Design Automation Conference, 1998. Proceedings*, 15-19 Jun 1998 Page(s):650 – 655

[3] S. Tahar; Slides of “Formal Verification”, Concordia University.

[4] Zilic, Z.; Vranesic, Z.G.; **“**Reed-Muller forms for incompletely specified functions via sparse polynomial interpolation”, *Multiple-Valued Logic, 1995. Proceedings., 25th International Symposium,* 23-25 May 1995 Page(s):36 – 43

[5] I. L. Zhegalkin, “Arithmetization of Symbolic Logic - Part One”, *Matematicheskii Sbornik*, 35(1), pp. 311-373, 1928, (in Russian with French summary).

[6] B.J. Falkowski, “A Note on the Polynomial Form of Boolean Functions and Related Topics”, *IEEE Transactions on Computers*, 48(8), pp.860-864, August 1999.

[7] Rolf Drechsler and Bernd Becker, “*Binary Decision Diagrams: Theory and Implementation”,* Kluwer Academic Publishers, 1998

[8] E.Clarke, M.Fujita, P.McGeer, K.L.McMillan, J.Yang and X.Zhao. *Multi terminal binary decision diagrams: An efficient data structure for matrix representation. In Int'l Workshop on Logic Synth.*, pages P6a:1-15, 1993

[9] R. P. Bryant and Y. A. Chen, “Verification of Arithmetic circuits with Binary Moment Diagrams”, *Proc. of 32nd Design Automation Conference*, pp. 535-541, 1995.

[10] K. Hamaguchi, A. Morita and S. Yajima, “Efficient Construction of Binary Moment Diagrams for Verification of Arithmetic Circuits”, In *Proc. ICCAD*, pp.78-82, 1995.

[11] R. Drechsler, B. Becker and S. Ruppertz, “The K*BMD: A Verification Data Structure”**,** *IEEE Design and Test of Computers*,Vol. 14, No. 2, pp. 51-59, April-June 1997

[12] Ciesielski, M.; Kalla, P.; Zhihong Zeng and Rouzeyre. B, “Taylor

Expansion diagrams: a new representation for RTL verification”**,** *High-Level Design Validation and Test Workshop, 2001. Proceedings. Sixth IEEE International,* 7-9 Nov. 2001 Page(s):70 – 75

[13] M. Ciesielski, P. Kalla, Z. Zeng and B. Rouzeyre, “Taylor Expansion Diagrams: a Compact, Canonical Representation with Applications to Symbolic Verification”, *Proc. Design Automation & Test in Europe, DATE-2002*, pp. 285-289, March 2002

[14] Ciesielski, M.; Priyank Kalla; Askar, S.; “Taylor Expansion Diagrams: A Canonical Representation for Verification of Data Flow Designs”, Computers, IEEE Transactions on, Volume 55, Issue 9, Sept. 2006 Page(s):1188 - 1201

[15] M. Ciesielski, P. Kalla, Z. Zeng and B. Rouzeyre, “Taylor Expansion Diagrams: a Compact, Canonical Representation with Applications to Symbolic Verification”, *Proc. Design Automation & Test in Europe, DATE-2002*, pp. 285-289, March 2002.

[16] Becker, B.; Drechsler, R.; Enders, R.; **“**On the representational power of bit-level and word-level decision diagrams”**,** *Design Automation Conference 1997. Proceedings of the ASP-DAC '97. Asia and South Pacific*, 28-31 Jan. 1997 Page(s):461 – 467

[17] B. Alizadeh, M. Fujita, “Modular-HED: A Canonical Decision Diagram for Modular Equivalence Verification of Polynomial Functions”, *in the fifth Workshop on Constraints in Formal Verification (CFV),* pp. 22-40, 2008.

[18] S. Kim and W. Sung, “Fixed-point error analysis and word length optimization of 8 × 8 IDCT,” *IEEE Trans. Circuits Syst. Video Tech.*, Vol. 8, No. 8, Dec. 1998, pp. 935–940.

[19] K. Kum and W. Sung, “Combined wordlength optimization and highlevel synthesis of digital signal processing systems,” *IEEE Trans. CAD* Vol. 20, No. 8, Aug. 2001, pp. 921–930.

[20] M. Willems, V. Bürgens, H. Keding, T. Grötker and H. Meyr, “System Level fixed-point design based on an interpolative approach,” *Proc. Design Autom. Conf.* 1997, pp. 293–298.

[21] A. Gaffar, O. Mencer, W. Luk, and P. Cheung, “Unifying bit-width optimisation for fixed-point and floating-point designs,” in *Proc. IEEE*

*Symp. Field-Programmable Custom Comput. Mach*, FCCM 2004, pp. 79–88.

[22] C. Shi and R. Brodersen, “Automated fixed-point data-type optimization tool for signal processing and communication systems,” *Proc. Design Automation Conf.* 2004, pp. 478–483.

[23] A. Nayak, M. Haldar, A. Choudhary, P. Banerjee, “Precision and error analysis of Matlab applications during automated synthesis for FPGAs,” *Proc. DATE*, 2001, pp. 722–728

[24] W. Sung and K. I. Kum, “Simulation-based wordlength optimization Method for fixed-point digital signal processing systems,” *IEEE Trans. Signal Processing,* vol. 43, pp. 3087–3090, Dec. 1995.

[25] S. Roy and P. Banerjee; “An algorithm for trading off quantization error with hardware resources for MATLAB-based FPGA design”, *IEEE Transactions on Computers,* 54(7), July 2005.

[26] E.R. Hansen, A generalized interval arithmetic, in “Interval Mathematics” (K. Nickel, ed.), Lecture Notes in Computer Science 29, pp. 7–18, Springer, 1975.

[27] R.B. Kearfott and V. Kreinovich, eds., *Applications of Interval Computations* (Kluwer, Dordrecht, 1996).

[28] R. Baker Kearfott, Algorithm 763: INTERVAL ARITHMETIC — A Fortran 90 module for an interval data type, ACM Transactions on Mathematical Software, 22, No. 4 (1996), 385–392.

[29] R. Moore, *Interval Analysis*. Englewood Cliffs, NJ: Prentice- Hall, 1966.

[30] W. Barth, R. Lieger, and M. Schindler. Ray tracing general parametric surfaces using interval arithmetic. The Visual Computer, 10, No. 7 (1994), 363–371.

[31] K. Ichida and Y. Fujii, An interval arithmetic method for global optimization, Computing, 23 (1979), 85–97.

[32] J. Stolfi, L.H. de Figueiredo, “An Introduction to Affine Arithmetic”, *TEMA Tend. Mat. Apl. Comput.,* 4, No. 3 (2003), 297-312.

[33] L.H. de Figueiredo and J. Stolfi, Affine arithmetic: Concepts and applications, Numerical Algorithms, (2004), to appear.

[34] J. Stolfi and L. de Figueiredo. Self-Validated Numerical Methods and Applications. Institute for Pure and Applied Mathematics (IMPA), Rio

de Janeiro, 1997.

[35] J.L.D. Comba and J. Stolfi, Affine arithmetic and its applications to computer graphics, in “Anais do VI Simp´osio Brasileiro de Computa¸c˜ao Gr´aficae Processamento de Imagens (SIBGRAPI’93)”, pp. 9–18, Recife (Brazil), October, 1993.

[36] A. Bowyer, R. Martin, H. Shou and I. Voiculescu, Affine intervals in a CSG geometric modeller, in “Proc. Uncertainty in Geometric Computations”, pp. 1–14. Kluwer Academic Publishers, July, 2001.

[37] F. Messine, Extentions of affine arithmetic: Application to unconstrained global optimization, Journal of Universal Computer Science, 8, No. 11 (2002), 992–1015.

[38] Q. Zhang and R.R. Martin, Polynomial evaluation using affine arithmetic for curve drawing, in “Proc. of Eurographics UK 2000 Conference”, pp. 49–56, 2000.

[39] C. Fang, R. Rutenbar, and T. Chen, “Fast, accurate static analysis for fixed-point finite-precision effects in DSP designs,” in *Proc. ACM/IEEE Int. Conf. Comput.-Aided Des.*, 2003, pp. 275–282.

[40] C. Fang, R. Rutenbar, M. Püschel, and T. Chen, “Toward efficient static analysis of finite-precision effects in DSP applications via affine arithmetic modeling,” in *Proc. ACM/IEEE Design Automation Conf.*, 2003, pp. 496–501.

[41] W. G. Osborne, R. C. C. Cheung, J. G. F. Coutinho, and W. Luk. “Automatic accuracy guaranteed bit-width optimization for fixed and floating-point systems”. *In Field- Programmable Logic and Applications. 17th International Conference*, FPL 2007, August 2007.

[42] D.-U. Lee, A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. Constantinides, “Accuracy-Guaranteed Bit-Width Optimization”, *IEEE Trans. CAD,* Vol. 25, No. 10, Oct. 2006, pp. 1990 –2000.

[43] D.-U. Lee, J.D. Villasenor, “A Bit-Width Optimization Methodology for Polynomial-Based Function Evaluation”, *IEEE Trans. Computers, Vol.56, No.4, Apr.07 pp. 567 – 571.*

[44] D. Lee, A. Abdul Gaffar, O. Mencer, and W. Luk, “MiniBit: Bit-Width Optimization via Affine Arithmetic,” Proc. ACM/IEEE Design

Automation Conf., pp. 837-840, 2005.

[45] W.G. Osborne, J. Coutinho, R. Cheung, W. Luk, O. Mencer, “Instrumented Multi-Stage Word-Length Optimization”, *Proc. Field-Programmable Technology*， Dec. 2007, pp. 89 – 96

[46] G. Constantinides and G. Woeginger, “The complexity of multiple wordlength assignment,” *Appl. Math. Lett.*, vol. 15, no. 2, pp. 137–140, 2001.

[47] G. Constantinides, P. Cheung, and W. Luk, “Wordlength optimization for linear digital signal processing,” *IEEE Trans. on CAD* vol. 22, no. 10, pp. 1432–1442, Oct. 2003.

[48] G. Constantinides, P. Cheung, and W. Luk, “Optimum wordlength location,” in *Proc. IEEE Symp. Field Program. Custom Comput. Machines*, 2002, pp. 219–228.

[49] G. Constantinides, “Perturbation analysis for word-length optimization,” in *Proc. IEEE Symp. Field-Program. Custom Comput. Machines*, 2003, pp. 81–90.

[50] CONSTANTINIDES, G., CHEUNG, P., AND LUK, W. 2001. Heuristic datapath allocation for multiple wordlength systems. In *Proceedings of the Design Automation and Test in Europe (DATE)* (Munich).

[51] S. Gopalakrishnan and P. Kalla, “Optimization of polynomial datapaths using finite ring algebra”, ACM Transactions on Design Automation of Electronic Systems (TODAES)**,** Volume 12 Issue 4, Sep.2007.

[52] N. Shekhar, P. Kalla, F. Enescu, “Equivalence Verification of Polynomial Datapath with Multiple Word-Length Operands”, *in Proc. of Design Automation and Test in Europe (DATE)*, pp. 824-829, 2006.

[53] N. Shekhar, P. Kalla, F. Enescu, and S. Gopalakrishnan, “Equivalence Verification of Polynomial Datapaths with Fixed- Size Bit-Vectors using Finite Ring Algebra”, in Intl. Conf. on Computer- Aided Design, ICCAD, 2005.

[54] A. Ahmadi and M. Zwolinski; “Symbolic noise analysis approach to computational hardware optimization”, *DAC 2008. 45th ACM/IEEE,* 8-13 June 2008 Page(s):391 – 396

[55] Kinsman, A.B.; Nicolici, N.; “Finite Precision bit-width allocation

using SAT-Modulo Theory", *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE '09.* 20-24 April 2009 Page(s): 1106 – 1111

[56] S.L.Hurst, D.M.Miller and J.C.Muzio, *Spectral Techniques in Digital Logic,* Academic Press, 1985

[57] Radomir S. Stankovic; Jaakko T. Astola; "Spectral Interpretation of Decision Diagrams", Springer, 2003

[58] Radomir S. Stankovic, Tsutomu Sasao; "A Discussion on the History of Research in Arithmetic and Reed–Muller Expressions", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol.20, no.9, September 2001

[59] Falkowski, B.J; "Calculation of Rademacher-Walsh spectral coefficients for systems of completely and incompletely specified Boolean functions", *Circuits and Systems, 1993., ISCAS'93, 1993 IEEE International Symposium,* 3-6 May 1993 Page(s):1698 - 1701 vol.3

[60] Falkowski, B.J.; Chip-Hong Chang, "Efficient algorithms for the calculation of Walsh spectrum from OBDD and synthesis of OBDD from Walsh spectrum for incompletely specified Boolean functions", *Circuits and Systems, 1994., Proceedings of the 37th Midwest Symposium,* Volume 1, 3-5 Aug. 1994 Page(s):393 - 396 vol.1

[61] Falkowski, B.J.; Perkowski, M.A; "Walsh type transforms for completely and incompletely specified multiple-valued input binary functions", *Multiple-Valued Logic, 1990., Proceedings of the Twentieth International Symposium*, 23-25 May 1990 Page(s):75 – 82

[62] K. Radecka and Z. Zilic, "Specifying and Verifying Imprecise Circuits by Arithmetic Transforms", *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pp. 128-131, 2002.

[63] Chip-Hong Chang; Falkowski, B.J.; "Operations on Boolean functions and variables in spectral domain of arithmetic transform", *Circuits and Systems, 1996. ISCAS '96., 'Connecting the World'., 1996 IEEE International Symposium*, Volume 4, 12-15 May 1996 Page(s):400 - 403 vol.4

[64] Clarke, E.M.; McMillan, K.L.; Zhao, X.; Fujita, M.; Yang, J.; "Spectral

Transforms for Large Boolean Functions with Applications”, *Design Automation, 1993. 30th Conference*, 14-18 June 1993 Page(s):54 – 60

[65] K.D. Heidtmann, “Arithmetic spectrum applied to fault detection for combinational networks”, *IEEE Trans. On Comput.,* vol.40, no.3, pp. 320-324, March 1991

[66] P.K. Lui and J.C. Muzio, “Spectral signature testing of multiple stuck-at faults in irredundant combinational networks,” *IEEE Trans. Comput.*, vol. C-35, pp. 1088-1092, Dec. 1986

[67] J. C. Muzio and D. M. Miller, "Spectral fault signatures for internally unate combinational networks," IEEE Trans. Comput., vol. C-32, pp. 1058-1062, Nov. 1983.

[68] K. Radecka and Z. Zilic; “Using Arithmetic Transform for Verification of Datapath Circuits via Error Modeling”, VLSI Test Symposium, 2000. Proceedings. 18th IEEE 30 April-4 May 2000 Page(s):271 - 277

[69] Kartarzyna Radecka’s Ph.D Thesis

[70] K. Radecka and Z. Zilic, “Arithmetic Transforms for Compositions of Sequential and Imprecise Datapaths”, *Computer- Aided Design of Integrated Circuits and Systems, IEEE Transactions on,* Volume 25, Issue 7, July 2006 Page(s):1382 – 1391

[71] K. Radecka and Z. Zilic, “Arithmetic Transforms for Verifying Compositions of Sequential Datapaths”, *Proc. IEEE international Symposium on Computer Design*, pp. 348-358, 2001.

[72] Z. Zhou and W. Burleson, “Equivalence Checking of Datapaths Based on Canonical Arithmetic Expressions”, *Proceedings of 32nd Design Automation Conference,* pp. 546-551, San Francisco, 1995

[73] Kuo-Hua Wang; TingTing Hwang , **“**Boolean matching for Incompletely specified functions”**,** *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions*, 23-25 May 1995 Page(s):36 – 43

[74] Purwar, S.; **“**Polynomial representation of spectral coefficients”, *Electronics Letters*, Volume 28, Issue 15, 16 July 1992 Page(s):1412 – 1413

[75] Keim, M.; Martin, M.; Becker, B.; Drechsler, R.; Molitor, P.;

"Polynomial formal verification of multipliers", *VLSI Test Symposium, 1997., 15th IEEE*, 27 April-1 May 1997 Pp:150 – 155

[76] J. Smith and G. De Micheli，" Polynomial methods for component matching and verification", *Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers. 1998 IEEE/ACM International Conference*, 8-12 Nov 1998 Page(s):678 – 685

[77] J. Smith and G. De Micheli, "Polynomial Circuit Models for Component Matching in High-level Synthesis", *IEEE Transactions on VLSI*, vol. 9, no. 6, pp. 783-800, Dec. 2001.

[78] D. W. Currie, A. J. Hu, S. Rajan and M. Fujita, "Automatic Formal Verification of DSP Software", *Proceedings of 37th ACM/IEEE Design Automation Conference*, pp. 130-135, 2000.

[79] D. Knuth, "*The Art of Computer Programming*," Addison-Wesley, 1998.

[80] R. Cmar, L. Rijnders, P. Schaumont, S. Vernalde, and I. Bolsens, "A methodology and design environment for DSP ASIC fixed point refinement," in *Proc. ACM/IEEE Design Automation Test Eur. Conf.*, 1999, pp. 271–276.

[81] D. Menard and O. Sentieys, "Automatic evaluation of the accuracy of fixed-point algorithms," in *Proc. ACM/IEEE Design Automation Test Eur. Conf.*, 2002, pp. 1530–1591.

[82] S. Wadekar and A. Parker, "Accuracy sensitive word-length selection for algorithm optimization," in *Proc. IEEE Int. Conf. Comput. Des.*, 1998, pp. 54–61.

[83] W. G. Osborne, R. C. C. Cheung, J. G. F. Coutinho, and W. Luk. "Automatic accuracy guaranteed bit-width optimization for fixed and floating-point systems". *In Field- Programmable Logic and Applications. 17th International Conference, FPL 2007,* August 2007.

[84] J. Smith and G. De Micheli, "Polynomial methods for allocating complex components", In *Proc. Design, Automation and Test in Europe*, DATE, pp. 217 –222, 1999.

[85] Yu Pang, Katarzyna Radecka and Zeljko Zilic, "Arithmetic Transforms of Imprecise Datapaths by Taylor Series Conversion", *ICECS-2006,* pp. 696-699, Dec. 2006.

[86] Yu Pang; Radecka, K.; “Optimizing imprecise fixed-point arithmetic circuits specified by Taylor Series through Arithmetic Transform”, *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, 8-13 June 2008 Page(s):397 - 402

[87] Yu Pang; Radecka, K.; Zilic, Z.; “Verification of Fixed-Point Circuits Specified by Taylor Series Using Arithmetic Transform”, *Circuits and Systems and TAISA Conference, 2008. NEWCAS- TAISA 2008. 2008 Joint 6th International IEEE Northeast Workshop on*, 22-25 June 2008 Page(s):261 – 264

[88] Gok, M.; Schulte, M.J.; Arnold, M.G.; “Integer multipliers with overflow detection”**,** *Computers, IEEE Transactions on*, Volume 55, Issue 8, Aug. 2006 Page(s):1062 - 1066

[89] Schuite, M.J.; Balzola, P.I.; Akkas, A.; Brocato, R.W; “Integer multiplication with overflow detection or saturation”**,** *Computers, IEEE Transactions on*, Volume 49, Issue 7, July 2000 Page(s):681 - 691

[90] Landauro, A.; Lienard, J.; “On Overflow Detection and Correction in Digital Filters”**,** *Computers, IEEE Transactions on*, Volume C-24, Issue 12, Dec. 1975 Page(s):1226 - 1228

[91] P.D. Pai and A. Tran, “Overflow Detection in Multioperand Addition”, *Int'l J.Electronics*, vol. 73, no. 3, pp. 461-469, Sept. 1992.

[92] Falkowski, B.J.; Chip-Hong Chang; “Efficient algorithms for the calculation of arithmetic spectrum from OBDD and synthesis of OBDD from arithmetic spectrum for incompletely specified Boolean functions”**,** *Circuits and Systems, 1994. ISCAS '94., 1994 IEEE International Symposium on*, Volume 1, 30 May-2 June 1994 Page(s):197 - 200

[93] N.S. Nedialkov, V. Kreinovich, S.A. Starks, “Interval Arithmetic, Affine Arithmetic, Taylor Series Methods: Why, what next?”, Numerical Algorithms, vol.37, no. 1-4, pp. 325-336, 2004

[94] Falkowski, B.J.; Chip-Hong Chang; “Fast generalized arithmetic and adding transforms”, *Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95., IFIP International Conference on Hardware Description Languages; IFIP International Conference on Very Large Scale Integration., Asian and South Pacific*, 29 Aug.-1 Sept. 1995 Page(s):723 - 728

[95] Rene Krenz, Elena Dubrova, Andreas Kuehlmann; "Circuit-based Evaluation of the Arithmetic Transform of Boolean Functions", *Int. Workshop on Logic Synthesis*, 2002

[96] Whitney J. Townsend, Mitchell A. Thornton, Rolf Drechsler, D. Michael Miller; "Computing Walsh, Arithmetic, and Reed-Muller Spectral Decision Diagrams Using Graph Transformations", *Proceedings of the 12th ACM Great Lakes symposium on VLSI,* New York, New York, USA, pp. 178 – 183, 2002

[97] M. A. Thornton, D. M. Miller, R. Drechsler; "Transformations Amongst the Walsh, Haar, Arithmetic and Reed-Muller Spectral Domains", *International Workshop on Applications of the Reed-Muller Expansion in Circuit Design (RMW),* August 10-11, 2001, pp. 215-225

[98] C. Moraga, T. Sasao, and R. Stankovic; "A Unifying Approach to Edge-valued and Arithmetic Transform Decision Diagrams", *Automation and Remote Control,* Vol. 63, No. 1, 2002, pp. 125–138.

[99] Cintra, R.; de Oliveira, H.; "How to interpolate in arithmetic transform algorithms", *Acoustics, Speech, and Signal Processing, 2002. Proceedings. (ICASSP '02). IEEE International Conference on*, Volume 4, 13-17 May 2002 Page(s):IV-4169 vol.4

[100] Yu Pang; Radecka, K.; Zilic, Z.; "Fast Algorithms for Compositions of Arithmetic Transforms and Their Extensions", *Circuits and Systems and TAISA Conference, 2008. NEWCAS-TAISA 2008. 2008 Joint 6th International IEEE Northeast Workshop on*, 22-25 June 2008 Page(s):314 – 317

[101] Pang, Yu; Radecka, Katarzyna; Zilic, Zeljko; "Algorithms for Compositions of Arithmetic Transforms and Their Extensions", Electronics, Circuits and Systems, 2006. ICECS '06. 13th IEEE International Conference on 10-13 Dec. 2006 Page(s):379 – 382

[102] A.Madisetti and A.N.Willson, Jr., "A 100 MHz 2-D 8x8 DCT/IDCT processor for HDTV applications", *IEEE Trans, Circuits Syst. Video Technol.*, vol.5, no.2, pp. 158-165, Apr. 1995.

[103] B. Alizadeh and M. Fujita, "A Canonical and Compact Hybrid Word-Boolean Representation as a Formal Model for Hardware/ Software Co-designs", *in the fourth Workshop on Constraints in*

*Formal Verification (CFV),* pp. 15-29, 2007.

[104] D. Brand, “Incremental Synthesis”, in *Proc. of International Conference on Computer-Aided Design (ICCAD)*, pp. 14-18, 1994.

[105] M. Fujita, T. Kakuda, Y. Matsunaga, “Redesign and Automatic Error Correction of Combinational Circuits”, *in Proc. of the IFIP TC10/WG10.5 Workshop on Logic and Architecture Synthesis*, pp. 253-262, 1990.

[106] M. Kubo, M. Fujita, “Debug Algorithm for Arithmetic Circuits on FPGAs”, *International Conference on Field-Programmable Technology, (FPT)*, pp. 236-242

[107] D. Stoffel, W. Kunz, “Verification of Integer Multipliers on the Arithmetic Bit Level,” in *Proc. of International Conference on Computer-Aided Design (ICCAD)*, pp. 183-189, 2001.

[108] O. Sarbishei, B. Alizadeh and M. Fujita, “Polynomial Datapath Optimization Using Partitioning and Compensation Heuristics”, *Accepted to appear in Proc. of International Design Automation Conference (DAC),* 2009.

[109] T. Stanion, “Implicit Verification of Structually Dissimilar Arithmetic Circuits”, *in Proc. of IEEE International Conference on Computer Desgin (ICCD)*, pp. 46-50, 1999.

[110] M. J. Schulte and E. E. Swartzlander, Jr., “Hardware designs for exactly rounded elementary functions,” *IEEE Trans. Comput.*, vol. 43, no. 8, pp. 964–973, Aug. 1994.

[111] A. Mallik, D. Sinha, P. Banerjee, and H. Zhou. Low-power optimization by smart bit-width allocation in a SystemCbased ASIC design environment. *IEEE Transactions on Computer- Aided Design of Integrated Circuits and Systems,* 26(3):447-455, March 2007.

[112] A. Peymandoust and G. DeMicheli, “Application of Symbolic Computer Algebra in High-Level Data-Flow Synthesis”, IEEE Trans. CAD, vol. 22, pp. 1154–11656, 2003.

[113] D. Cyrluk, O. Moller and H. Rues, “An Efficient Decision Procedure for the Theory of Fixed-Size Bitvectors”, In *Proc. of LNCS, Computer Aided Verification*, vol 1254, 1997.

[114] S. Kim, K. Kum, and W. Sung, “Fixed-point Optimization Utility for C

and C++ Based Digital Signal Processing Programs," in *Workshop on VLSI and Signal Processing* '95, (Osaka), pp. 197-206, Nov. 1995.

[115] G. De Micheli, *Synthesis and optimization of digital circuits.* McGraw-Hill, 1994.

[116] F. Catthoor, J. Vandewalle, and H. De Man, "Simulated annealing based optimization of coefficient and data word-lengths in digital filters," *Int. J. Circuit Theory Applicat.*, vol. 16, pp. 371–390, Sep. 1988.

[117] J.-I. Choi, H.-S. Jun, and S.-Y. Hwang, "Efficient hardware optimization algorithm for fixed point digital signal processing ASIC design," *Inst. Elect. Eng. Electron. Lett.*, vol. 32, no. 11, pp. 992–994, May 1996.

[118] Y. C. Lim and S. R. Parker, "Finite word-length FIR filter design using integer programming over a discrete coefficient space," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-30, pp. 661–664, Aug. 1982.

[119] D. Menard, and O. Sentieys, "A methodology for evaluating the precision of fixed-point systems," *Proc. IEEE Int. Conf on Acoust., Speech, and Signal Processing,* vol. 3,2002. pp. 3152-3155.

[120] M. A. Cantin, Y. Savaria, and P. Lavoie, "A comparison of automatic word length optimization procedures," *Proc.IEEEInt. Sym. Circs. andSys,* 2002, vol. 2, pp. 612 -615.

[121] C. Shi, and R. W. Brodersen, "A perturbation theory on statistical quantization, effects in fixed-point DSP with nonstationary inputs," *IEEE Int. Sym. Circs. and Sys,* 2004.

[122] C. Shi, and R. W. Brodersen, "Floating-point to fixed-point conversion with decisipn-errors due to quantization," *IEEE Int. Conf on Acoust., Speech, and Signal Processing,* 2004.

[123] B. Lee and N. Burgess, "Some Approximations on Taylor-Series Function Approximation on FPGA," Proc. Asilomar Conf. Circuits, Systems, and Computers, vol. 2, pp. 2198-2202, 2003.

[124] A. Tzidon, I. Berger and Y.M. Yoeli, "A practical approach to fault Detection in combinational circuits", *IEEE Trans. Comput.,* vol. C-27, pp. 968-971, Oct. 1978

[125] H. Choi and W. P. Burleson, “Search-based wordlength optimization for VLSI/DSP synthesis,” in *Proc. VLSI Signal Processing*, La Jolla, CA, 1994, pp. 198–207.

[126] M.-A. Cantin, Y. Savaria, D. Prodanos, and P. Lavoie, “An automatic word length determination method,” in *Proc. IEEE Int. Symp. Circuits and Systems (ISCAS)*, Sydney, Astralia, 2001, vol. 5, pp. 53–56.

[127] M. Berz and G. Hoffstätter, Computation and application of Taylor polynomials with interval remainder bounds, Reliable Comput. 4 (1998) 83–97.

[128] A. Hosangadi, F. Fallah, and R. Kastner, “Energy Efficient Hrdware Synthesis of Polynomial Expressions”, in Int’l. Conf. on VLSI Design, pp. pp. 653–658, 2005.

[129] Xing X W, Jong C C. “Using symbolic computer algebra for subexpression factorization and subexpression decomposition in high-level synthesis”. *Proceedings of the IEEE International Symposium on Circuits and Systems* (*ISCAS’ 05*), Kobe, 2005, pp.5645-5648.

[130] Hosangadi A, Fallah F, Kastner R. “Optimizing polynomial expressions by algebraic factorization and common subexpression elimination”. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 2006; 25(10), pp.2012-2022.

[131] M. Gok, M.J. Schulte, P.I. Balzola, and R.W. Brocato, “Efficient Integer Multiplication Overflow Detection Circuits,” *Proc. 35th Asilomar Conf. Signals, Systems, and Computers*, pp. 1661-1665, 2001.

[132] Y.H. Cha, G.Y. Cho, H.H. Choi, and H.B. Song, “N Bit Result Integer Multiplier with Overflow Detector,” *IEE Electronic Letters*, vol. 37, pp. 940-942, July 2001.

[133] Erick L. Oberstar, “Fixed-Point Representation & Fractional Math”, Aug. 2007

[134] Randy Yates, “Fixed-Point Arithmetic: An Introduction”, Jul. 2009

[135] K. Radecka and Z. Zilic, "Verification by Error Modeling: Using Testing Techniques for Hardware Verification" , Kluwer Academic Publishers, 2003.

[136] A. Veneris and M. Abadir, “Design Error Diagnosis and Correction via Test Vector Simulation”, *IEEE Transactions of CAD of Integrated Circuits and Systems*, 18(12), pp. 1803-1816, 1999.

[137] Synopsys Inc, “*Co-centric Fixed Point Designer Datasheet*”, 2002.

[138] M. Huhn, K. Schneider, Th. Kropf and G. Logothetis, “Verifying Imprecisely Working Arithmetic Circuits”, *Proc. Design Automation and Test Europe*, pp. 65-69,1999.

[139] T. Damarla, and M. Karpovsky, “Fault Detection in Combinational Networks by Reed-Muller Transform”, *IEEE Transactions on Computers*, 38(6), pp. 788-797, Jun.1989.

[140] G. Even and W. J. Paul, “On the Design of IEEE Compliant Floating Point Units”, *IEEE Trans. Computers*, Vol. 49, No. 5, pp. 398-413, May 2000.

[141] Y. A. Chen and R. Bryant, “ACV: An arithmetic circuit verifier,” in *Proc. ACM/IEEE Int. Conf. Computer-Aided Design*, San Jose, CA, Nov. 1996, pp. 361–365.

[142] A. J. Al-Khalili; Slides of “Digital Systems Designs and Synthesis”, Concordia University.

[143] M. Huhn, K. Schneider, Th. Kropf and G. Logothetis, “Verifying Imprecisely Working Arithmetic Circuits”, *Proc. Design Automation and Test Europe*, pp. 65-69,1999.

[144] L. Entrena and K-T. Cheng, “Combinational and Sequential Logic Optimization by Redundancy Addition and Removal”, *IEEE Transactions on CAD*, 14(7), pp. 909-916, Jul. 1995.

[145] T. Damarla, “Generalized Transforms for Multiple Valued Circuits and their Fault Detection”, *IEEE Transactions on Computers*, 41(9), pp. 1101-1109, Sep. 1992.

[146] T. Kropf, “Introduction to Formal Hardware Verification”, New York, *Springer*, 1999

[147] T. Larrabee, “Test Pattern Generation using Boolean Satisfiability”, *IEEE Transactions on CAD of Integrated Circuits and Systems*, 11(1), pp. 4-15, Jan. 1992.

[148] C. Lee, “Representation of Switching Circuits by Binary-Decision Programs”, *Bell Systems Technical Journal*, vol. 38, pp. 985-999, July

1959.

[149] Z. Zilic and Z. G. Vranesic, “A Deterministic Multivariate Interpolation Algorithm for Small Finite Fields”, To appear in *IEEE Transactions on Computers*, Sep. 2002.

[150] http://en.wikipedia.org/wiki/Simulation#Engineering.2C_technology _or_ process_simulation

[151] http://en.wikipedia.org/wiki/Hardware_emulation

[152] http://en.wikipedia.org/wiki/Satisfiability_problem

[153] SoftJin Infotech Private Limited; “Enabling RTL-to-gate equivalence checking”.

[154] Swaroop Ghosh, Swarup Bhunia, Kaushik Roy; “Low-Power and Testable Circuit Synthesis Using Shannon Decomposition”, *ACM Transactions on Design Automation of Electronic Systems*, 2007vol.12(no.4)

[155] http://en.wikipedia.org/wiki/Formal_verification

[156] http://en.wikipedia.org/wiki/Floating_point

[157] M. Boule and Z. Zilic, *"Generating Hardware Assertion Checkers for Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring"* , Springer, 2008. ISBN: 978-1-4020-8585-7

[158] K. Radecka and Z. Zilic, *"Verification by Error Modeling: Using Testing Techniques for Hardware Verification"* , Kluwer Academic Publishers, 2003. ISBN: 978-1-4020-7652-7

[159] Morin-Allory, K.; Boule, M.; Borrione, D.; Zlic, Z.; “Proving and disproving assertion rewrite rules with automated theorem provers”, HLDVT '08. IEEE International, 19-21 Nov. 2008, pp: 56 - 63