Data-path Optimization in High-level Synthesis

高位合成におけるデータパス最適化に関する研究

2006年2月

早稲田大学大学院情報生産システム研究科
情報生産システム工学専攻　高位検証技術研究

土井 伸洋

Data-path Optimization in High-level Synthesis

Nobuhiro DOI

Abstract

High-level synthesis is a novel method to generate a RT-level hardware description automatically from a high-level language such as C, and is used at recent digital circuit design. However, there is a big gap between RT-level hardware descriptions and high-level language ones, and various optimization techniques such as pipelining, parallelizing and bit-length optimization are applied to fill the gap at preprocessing stage of high-level synthesis. For image/sound processing algorithm including many single/double-precision floating-point operations, we usually convert such float/double operations to integer/fixed-point operations to improve the area and speed in hardware implementation. The conversion is a key to reduce area but is a rather tedious work for designers.

This thesis introduces two automatic bit-length optimization methods on floating to fixed-point conversion for high-level synthesis. The first method is based on heuristic resource allocation. The bit-length of fractional part is determined according to the acceptable error specified by a designer. The method also uses static error analysis based on interval arithmetics, so the optimization speed is much faster than the traditional simulation based approach. In the second method, non-linear programming technique is newly introduced to obtain optimum bit-length. The technique improves the balancing between computation accuracy and total bit-length of variables and operation units.

Experimental result shows that both optimization methods are successful in reducing bit-length of variables with short time. The non-linear programming method finds better solutions than the heuristic resource allocation based method, but designers easily cooperate with the heuristic resource allocation method to optimize bit-length with specified constraints for medium size circuits.

Finally, effectiveness of proposed methods and future works are discussed.

Keywords:

HDL, High-level Synthesis, Compiler, Bit-length Optimization, Non-linear programming

高位合成におけるデータパス最適化に関する研究

土井 伸洋

内容梗概

高位合成は, C 言語を始めとする高級言語からレジスタトランスファレベルのハードウェア記述を自動的に合成する技術であり, デジタル回路設計の現場において使われることが多くなってきている. しかし, 高級言語とハードウェア記述の間には抽象度において大きな開きがあるため, さまざまなハードウェア向け最適化技術を適用することでこの差を解消する必要がある. とくにパイプライン化や自動並列化, そして本研究で取り扱うビット長の最適化などが重要である. 通常, 音声や動画像を扱うアルゴリズムの多くが浮動小数点演算 (単精度/倍精度) を含んでいるが, コストや動作速度という理由から固定小数点演算として実現されることが多い. この変換は設計者が手動で行なうことが多く, 演算誤差の問題から非常に困難な作業となっている.

本研究では浮動小数点演算から固定小数点演算への変換を自動的に行なうための手法について述べる. はじめにビットのアロケーションにヒューリスティックを利用したビット長最適化手法について述べる. この方式では, 設計者によって与えられる許容誤差をもとにレジスタや演算器の小数部ビット長を算出する. ソースプログラムの誤差解析においては区間演算に基づき静的に解析を行なうため, 従来ビット長最適化の主流であったシミュレーションベースの方法と比較し非常に高速である. つぎに, ビット長最適化問題を非線形計画法を用いて定式化し, SQP 法 (逐次二次計画法) の適用により最適なビット長を決定する手法について述べる. この手法においては, 定式化により, 演算精度やコストのバランスを容易にとることができるという利点がある.

提案アルゴリズムの有効性を確かめるためにいくつかの実験を行ない, 両方式ともに短時間でビット長最適化を実現できることを確認した. 非線形計画法を用いた最適化手法は精度の面でヒューリスティックベースの手法をうわまわっているが, 中規模の回路に対し人手で最適化を行なう場合には, 制約の入れやすさの点でヒューリスティックな手法が適している.

また, データパス最適化に関する今後の課題について述べる.

キーワード:

HDL, 高位合成, コンパイラ, ビット長最適化, 非線形計画法

Contents

Chapter 1

Introduction

IC (Integrated Circuit) or LSI (Large Scale IC) is now used in everything from airplanes to celler phones. The fabrication technology of ICs is rapidly improved, and hundred millions of transistors can be integrated on one chip. Then, not only simple signal processing, but also complex system such as Mpeg encoding/decoding can be implemented. However, it requires much time and designer resources to design a LSI because of the complexity of the circuit. Figure 1.1 is a roadmap about design productivity [1], where the line O is the number of transistors which can be integrated on a chip, and the line G is the number of transistors which can be designed by a designer per year. It can be seen that the gap between these lines grows rapidly. The gap is called as the design productivity gap and is expected to reach the dangerous zone where "Size of ICs to be designed will soon reach levels impossible to be handled with the design paradigm as it is known today".

In order to manipulate large number of modules and reduce design time, various design automation technologies have been developed. Layout automation is a technology to automatize the placement of transistors and the routing lines among transistors. By this, we need not worry about the real position and routing information and just need to consider the abstracted position and the connectivity. The circuits at the abstract-level is called gate-level circuits or netlists.

Logic synthesis is another automation technology to generate a gate-level circuit from a register transfer level (RT-level) description. Designers give a RT-level circuit description written in Hardware Description Language (HDL) and specifies the constraints such as

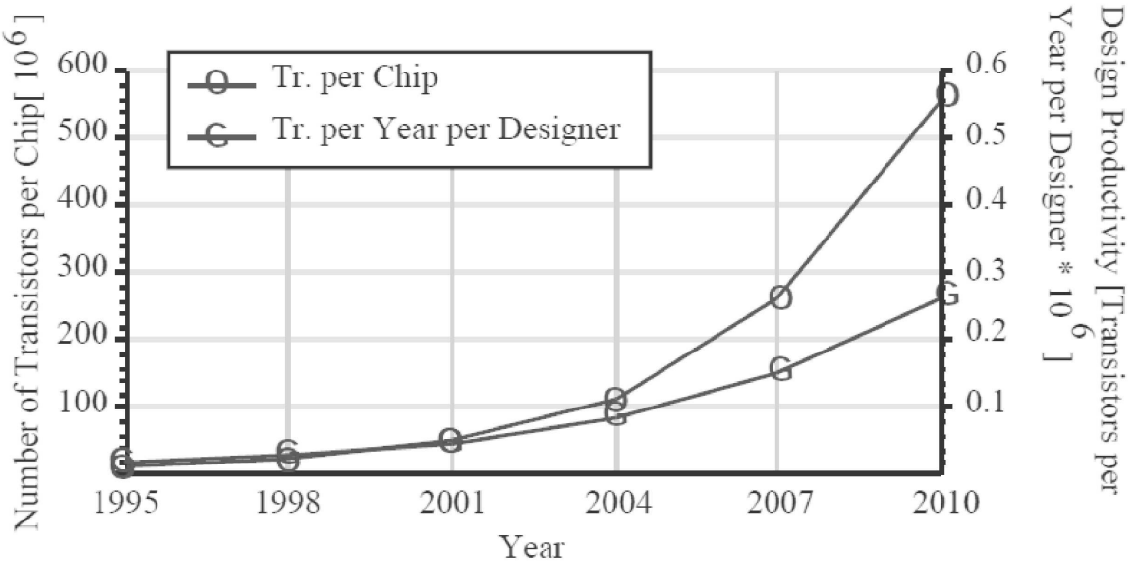

Figure 1.1: SIA road map : Design. [1]

the area and timing for the synthesis tool. In the RT-level description, designers describe operations on registers not only at bit-level but also at word-level to manipulate multiple bit registers. The logic synthesizer applies various optimization methods, which have been developed late 1980's, and generates a gate-level circuit called netlist. At present, Design Compiler [2] from Synopsys and BuildGates [3] from Cadence Design Systems are used as defacto standard logic synthesis tools.

However, the number of transistors on recent LSI is too large to design using RT-level description, and product lifecycles have become shorter and shorter. Then, another design methodology which enables circuit design in higher abstraction level have been required. The design methodology called *High-level synthesis* is one answer to overcome this problem.

High-level synthesis (or *Behavior synthesis*)[4] is synthesis technology for hardware design too. A synthesizable RT-level circuit description is generated automatically from a behavior description. In general, behavior description is written in high-level language such as C. Each behavior is written in serial, and the detail of architecture such as the

number of registers/function units are not defined in a source code. The high-level synthesis tool applies *scheduling*, *register allocation*, *parallelizing* and various optimizations, then generates control/data-path circuits in RT-level. Designers can not only design digital circuits in high-level language, but also obtain different architectures without changing a source code. As surveyed in [5], several researches about high-level synthesis system have been reported [6, 7, 8, 9], and industrial tools are also promoted from some companies [10, 11, 12].

Figure 1.2 shows digital circuit design using high-level synthesis. Design automation technologies (High-level synthesis, Logic synthesis, Layout CAD) can be applied after a designer completes a circuit design in behavior level. In the manual optimization phase, some optimizations which cannot be handled by a high-level synthesis tool are applied. *Bit-length optimization* is difficult but important optimization, and it should be performed by human in current design flow. It is the task to determine minimum bit-length of registers/function units for the implementation. For example, a flag variable declared as an integer (= 32-bit) should be converted to 1-bit register for the area reduction. A modern multimedia application (mpeg, mp3, etc..) is designed with full-precision floating-point arithmetic, so it should be re-designed as a hardware oriented algorithm in which fixed-point arithmetic is used instead of floating-point one. The manual bit-length optimization is very time-consuming and rarely optimum [13].

This thesis provides automatic bit-length optimization techniques for high-level synthesis with floating-point to fixed-point conversion. One of the most difficult and time-consuming tasks for the conversion from an algorithm to a behavior description is bit-length optimization. To automatize this task, we propose two methods, the optimization based on heuristic resource allocation and the exact optimization based on non-linear programming technique. Topics discussed in the thesis is presented in Figure 1.2.

In Chapter 2, the overview of our high-level synthesis system is introduced. A simple high-level synthesis example is given for the explanation. Then, features of target platforms, ASIC (Application Specific Integrated Circuit) and FPGA (Field Programmable Gate Array), are summarized.

In Chapter 3, problems on floating-point to fixed-point conversion are discussed. The

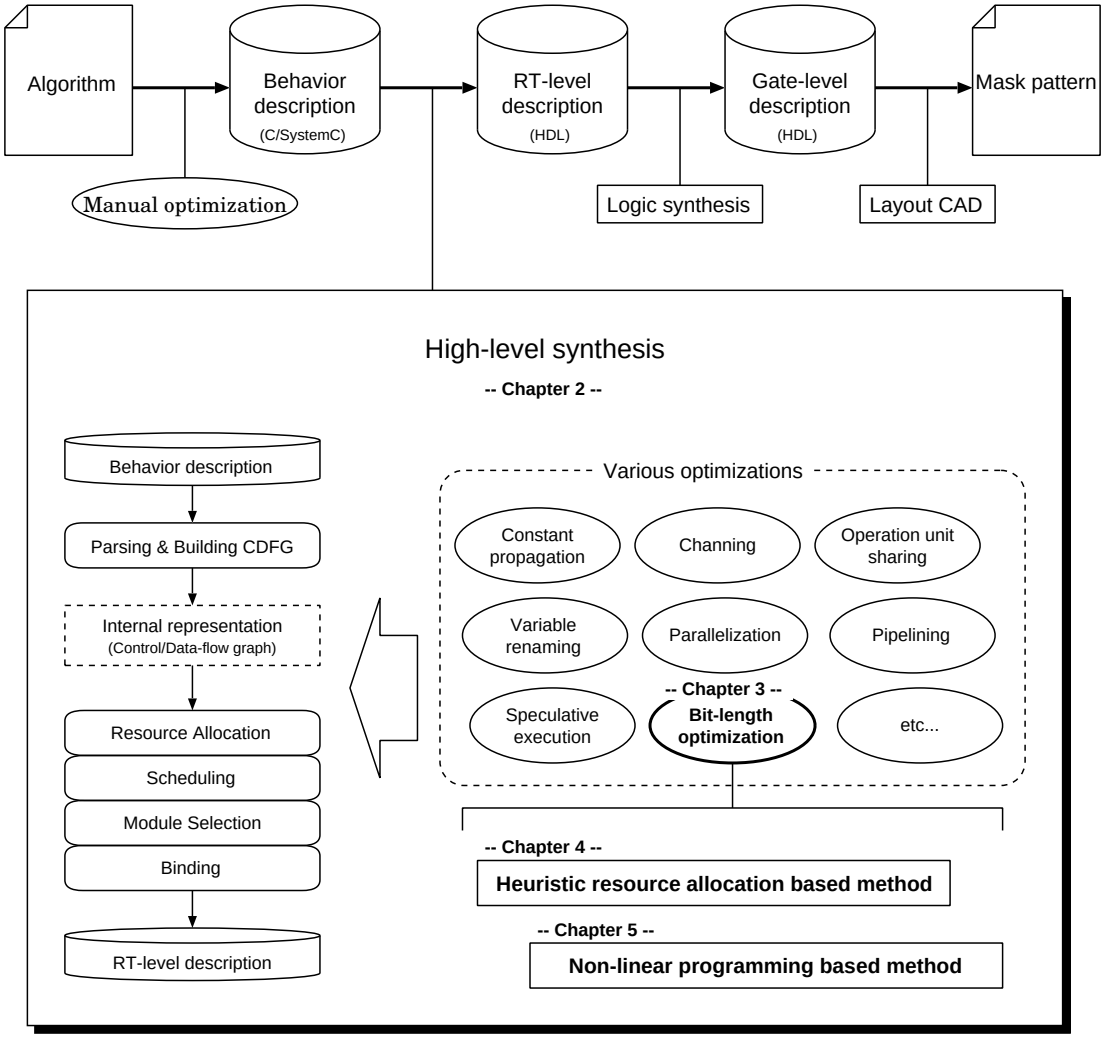

Figure 1.2: Digital circuit design using high-level synthesis

floating-point arithmetics enables real arithmetic on a digital computer. However, function units for floating-point arithmetics are large and their size are unacceptable in some case. So, fixed-point conversion is required for the hardware implementation.

In Chapter 4, bit-length optimization method based on heuristic resource allocation is proposed. In this method, static error analysis is used instead of the simulation base error analysis which takes much time. Computation error of each operation is estimated, and it is propagated to output statistically. Then, bit-length of variables are determined based

on an acceptable error. The method is much faster than the simulation base approach and the resulted bit-length guarantees computation accuracy for all input patterns.

In Chapter 5, exact optimization method based on non-linear programming technique is proposed. In this method, bit-length optimization problem is formalized as a non-linear problem and a general non-linear programming solver is applied. This approach is more general compared with linear-programming base approach because a source program can be modeled directly if it includes multiplications. By using formalization, various constrains such as maximum bit-length of function units or unit sharing can be handled in easy.

In Chapter 6, effectiveness of proposal methods and future works are discussed.

Chapter 2

High-level Synthesis from C Programs

2.1 Overview

High-level synthesis is one of design automation technologies which reduces design time and cost. A RT-level circuit description is automatically generated from a behavior description written in high-level language such as C or extended C according to the timing and area constraints. A generated circuit consists of data-path and control units. The data-path unit is described as networks of registers and function units, and the control unit, which is usually represented with a finite state machine (FSM), controls that.

High-level synthesis is useful for the architecture exploration too. In [14], an evaluation methodology using high-level language is reported. They models a M-JPEG system with SPADE architecture description language, evaluate it, and decide the architecture that satisfy the demands. The evaluation speed with a high-level language is much faster than that with a RT-level description language, so various architectures can be checked in short time. Furthermore, designers don't describe the architecture detail in the circuit design using high-level synthesis. The synthesis tool generates different architectures according to given constraints, and designers evaluate them and select one.

Many researches have been reported in last two decades and several high-level synthesis systems are produced [6, 7, 8, 9]. A high-level synthesis system MIMOLA [6] is used for the design of digital processors. The synthesizer of MIMOLA accepts a PASCAL-

like high-level description as a specification and produces a register transfer description. The system includes a retargetable compiler for the generation of processor microcodes. HAL system [7] is automatic data-path synthesis system. In the system, a novel "load balancing" technique, which reduces the concurrency of similar operations, is introduced. Stanford University CAD group have developed Olympus synthesis system [8]. The system includes behavioral, structural, and logic synthesis tools, and provides technology mapping and simulation. In the system, a behavior description written in a hardware description language called HardwareC is given, and outputs a netlist. SPARK [9] is developed at University of California, and one of the successful high-level synthesis frameworks. It takes a behavior ANSI-C code as a input, and outputs synthesizable RT-level VHDL. In the SPARK system, parallelizing compiler is used to enhance instruction-level parallelism, and various transformations such as speculative code motions are introduced. By using these techniques, the system achieved 70% improvements in performance on some multimedia applications without any increase in the overall area and critical path.

Some industrial tools are also promoted from some companies [10, 11, 12]. "Cyber" high-level synthesis system is promoted by NEC. It generate a RT-level circuit description from a behavior description language (BDL), which is their original extended C language. BDL has some extension types (input and output port declarations, bit-width, etc...) and hardware-oriented operations (bit extraction, reduction or/and). The behavior synthesizer Cyber receives a source code written in BDL, applies hardware-oriented optimization and generates a synthesizable RT-level circuit description. They synthesis a NIC chip for PC (300-states, 150K gates) with Cyber. "Forte Design Systems - Cynthesizer" generates an optimized RT-level implementation from a behavior description written in SystemC [15]. Starting from un-timed high-level SystemC models, Forte's synthesizer builds timed RT-level hardware implementations based on an external set of directives or constraints (clock speed, latency, operation units, etc...) specified by a designer. The synthesis tool "Mentor -Catapult C Synthesis" accepts an untimed C++ source code. It provides feedback for each architecture modification, and the partial solutions are saved for re-exploration. So, the tool allows a single source code representation to be used to drive multiple implementations.

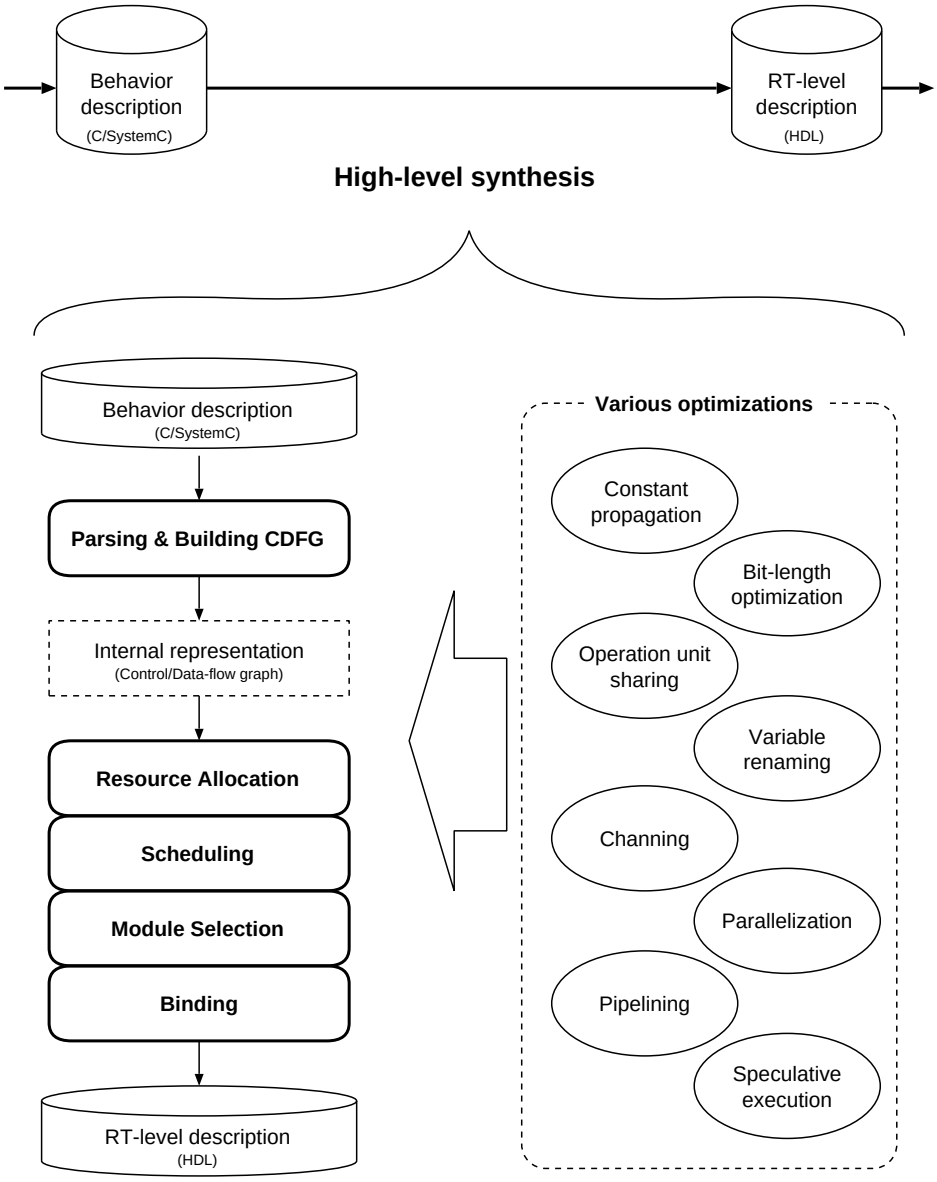

Figure 2.1: Hardware generation from a behavior description

2.2 Design Flow using High-level Synthesis

We have been developing a high-level synthesis system with bit-length optimization. The synthesis system consists of several sub-processes and optimizations as shown in Figure 2.1. Each sub-processes are summarized as follows:

- **Parsing & Building CDFG**

  A source code written in high-level language is parsed, and converted into a internal representation such as Control/Data-flow graph.

- **Resource Allocation**

  The synthesis tool evaluates a source code and estimates the number and kind of resources (adder, multiplier, etc...) required for the implementation.

- **Scheduling**

  The execution timing of operations are determined according to the resource and timing constraints.

- **Module Selection**

  The resource type is selected from the resource library that an operation executes on. For example, an addition can be executed on an adder, ALU or MAC unit.

- **Binding**

  Operations are mapped to specific function units, variables to registers, and data/control transfers to interconnection components.

In each sub-processes, additional optimizations are usually applied. For example, two operations can be executed in parallel if there are no control/data dependencies. It is *parallelization* technique and applied in the scheduling process. After these processes, a synthesizable circuit description is generated in RT-level.

2.2.1 Simple High-level Synthesis Example

We show a simple high-level synthesis example using pseudo code (Figure 2.2-(a)) [16]. The source code includes 4-inputs, 2-outputs and 2 internal variables. Note that several data-types and various control structures exist in the practical design.

At first, the synthesis tool parses a source code and convert it into an internal representation. Figure 2.2-(b) is a *Data-Flow Graph* (DFG) of the source code, where the DFG is

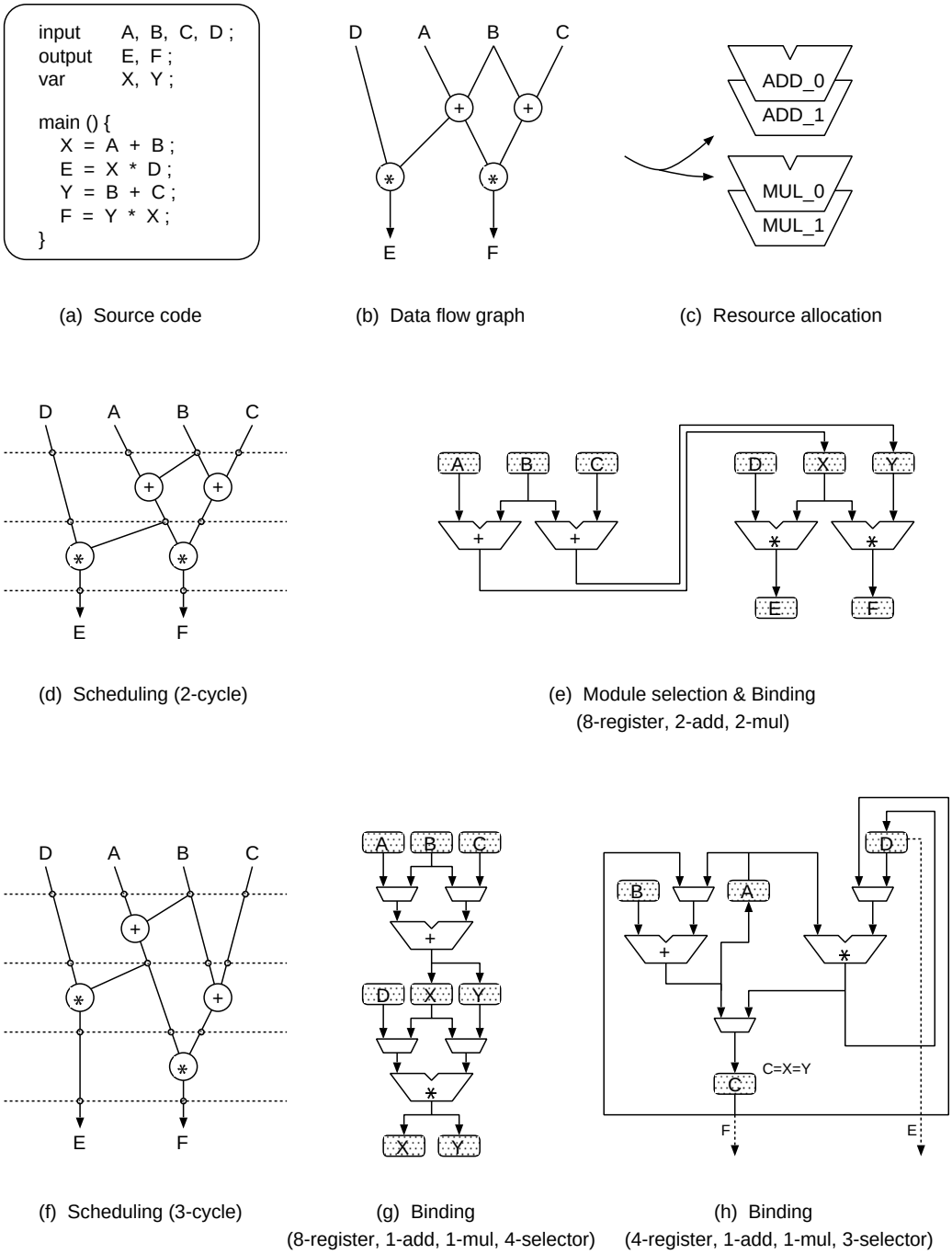

Figure 2.2: Simple high-level synthesis example

a typical representation model [17, 18]. It contains edges and nodes representing values and hardware operations such as adders and multiplications.

Next, a DFG is evaluated and the number of resources is determined. In this example, 2-adders and 2-multipliers are required (Figure 2.2-(c)).

Then, operations are scheduled. In the scheduling process, the synthesis tool determines the clock cycle in which each operation in the design executes under the timing constraint. Figure 2.2-(d) shows a 2-cycle schedule. Two additions are executed at first step, then two multiplications are executed in second step.

After that, data-path/control circuits are constructed. Each operation is mapped on specified function units. Figure 2.2-(e) is a data-path circuit, where 8-registers, 2-adders and 2-multipliers are used. Note that synthesis tool should chooses one resource type from resource library under various constraints such as speed or area in a practical example (Module selection). In the example, an Ripple Carry Adder, Carry Look-Ahead Adder and other types of adder are available for an addition operation, and synthesis tool chooses one of them.

Figure 2.2-(f)(g)(h) shows other implementations derived from single source code. The schedule in Figure 2.2-(f) is 3-cycle, and it requires less hardware resources. The implementation result is shown in Figure 2.2-(g), where 2-additions and 2-multiplications are mapped on one adder unit and one multiplication unit respectively. By applying *register sharing*, we can get a simpler data-path circuit as shown in Figure 2.2-(h).

Each sub-processes are NP-complete and deeply dependent on each other. So, many representation models, scheduling algorithms, binding algorithms and optimization techniques are proposed for the practical application.

Several representation models are proposed in [19, 20, 21]. A *Control-Data Flow Graph* proposed in [19] unifies representations of control parts and data-flow parts. The *Behavior Network Graph* [20] is an RT-level/Gate-level network representing all possible schedules that a behavior specification can assume. In this model, some special logic gates called *State-Value Node*, *Register-Value Node* and *Current-Value Node* are introduced. The *Extended Time Petri Net* [21] is an another representation model for high-level synthesis.

The ETPN is derived from Petri net theory [22], and used as the intermediate representation model for the high-level synthesis system called CAMAD [23].

Scheduling algorithm has great effect on the performance of synthesized circuit. In the traditional high-level synthesis system, *ASAP* (as soon as possible) and *ALAP* (as last as possible) scheduling algorithms are used [6]. These algorithms just schedule all the operations as soon as possible (or as last as possible). *Force-directed scheduling* [24] developed for HAL system [7] uses operation's mobility as scheduling heuristics. The mobility represents a time slack of each operation, and it is defined as the difference between the ASAP and ALAP start time of operation. In the *list scheduling*, operations are scheduled based on its priorities computed from control and data dependencies [25]. The formal approach is also proposed [26]. This paper presents an integer linear programming (ILP) model for the scheduling problem. When the register-to-register communication cost is high, task duplication technique is useful to improve scheduling [27].

Resource allocation and binding techniques have been explored in the past, too. The optimization goal of this processes is reducing hardware cost such as number of registers and function units. Tseng et al. [28] use clique partitioning heuristics to find a clique cover for a module allocation graph. The register allocation algorithm called *Left Edge algorithm* is proposed by F.J.Kurdahi et al. [29]. Variables whose lifetime do not concurrent each other are mapped on one register.

In order to improve the performance of synthesized circuits, various optimization techniques are propose. *Pipelining* is a typical one [30], and *tree height reduction* [31], re-timing [32] and software pipelining [33] are well known techniques. Recently, several research groups focus on the optimization of conditional branch and loops [34, 35]. The paper [34] describes *Loop folding* for the minimization of iteration time, and *Speculation* and *Operation duplication* across a merge node are presented in [35].

2.3 Hardware Platform

In general, two target platforms are available for hardware implementation, ASIC (Application Specific Integrated Circuit) and FPGA (Field Programmable Gate Array). Main

features of them are summarized as follows:

- ASIC

  An AISC is an integrated circuit customized for special use such as Mpeg-encoding, Ciphering and so on. Designers construct a circuit with redesigned logic cells (AND gates, OR gates, registers, etc...) known as *standard cells*. In the ASIC design, logic cells and wires can be placed freely, so designers can devise architectures and achieve high-performance. However, manufacturing cost is high and design time is long.

- FPGA

  A FPGA is a programmable device. Many logic elements called look-up table (LUT) which can be mapped simple binary function are integrated on one chip. Designers describe a circuit in HDL, and it is mapped automatically on a FPGA with a mapping tool . Design turnaround is only a few hours, however wiring resources are limited.

Currently, hardware emulators using FPGAs are developed for design prototyping. By using emulators and high-level synthesis tools, the architecture evaluation is available earlier in the design phase.

In order to achieve high-performance and small area, synthesis tools apply different optimizations according to the target platform. For example, sharing of adder units is effective for the circuit design on a ASIC. However, this optimization is senseless for FPGA based design since the number of logic elements for the mapping of one adder unit is almost same as that of one multiplexer.

Chapter 3

Floating-point to Fixed-point Conversion

3.1 Motivation

In the hardware design, bit-length of variables and function units are closely related to the area and speed of the circuit, so a lot of effort is spent to optimize it. For example, a flag variable may be declared as an integer (= 32-bit) in algorithm level, then it must be converted to 1-bit register in RT-level. The bit-length optimization techniques and tools for integer variables have been proposed in [36, 37].

Currently, multimedia algorithms include not only integer operations, but also floating-point operations which enables real-arithmetics on the hardware. Although a floating-point arithmetics offers both a wide dynamic range and high-precision computation, it requires many transistors and much power. So, its application is highly limited for mobile use. In many cases, these costly floating-point operations are converted to low-cost ones (i.e. fixed-point operations) for hardware implementations.

Fang et al. compares three implementation schemes on an inverse discrete cosine transform (IDCT) circuit design example [38]. In this paper, they implement an 8-point IDCT circuit based on the following three ways:

- IEEE standard floating-point unit (32-bit),
- Customized floating-point unit (15-bit),
- Fixed-point unit (up to 24-bit, not fine tuned),

Table 3.1: Comparison of three implementations of IDCT [38].

Implementation	Area(um^2)	Delay(ns)	Power(mw)
IEEE FP	926810	111	1369
Customized FP	216236	46.75	143
Fixed-point	106598	36.11	110

∗ STMicroelectronics 0.18um technology library

∗ Results on gate-level

where all circuits guarantee specified accuracy. Table 3.1 shows the implementation results. The implementation based on fixed-point arithmetics achieves small area, small delay and low power.

However, the floating-point to fixed-point conversion task is not easy because of the computation error. Designers should determine bit-length of variables and function units which is small but guarantees specified accuracy. In order to automatize this task, many approaches have been proposed and we also focus to this topic.

In Section 3.2 and 3.3, we briefly give the basics of floating-point arithmetics and fixed-point arithmetics respectively. In Section 3.4, we introduce related works and our methods.

3.2 Manipulation of Floating-point Variables and Operations

On the digital computer, every numbers are represented with finite wordlength. Since it is impossible to represent real values exactly, various representation methods using approximation have been developed. Floating-point arithmetics is one of them and the most common approach. It enables both a wide dynamic range for extremely large numbers and high precision for very small numbers.

The typical format of a floating-point number is specified by 3-tuple $\langle \boldsymbol{sign}, \boldsymbol{exponent}, \boldsymbol{significand} \rangle$, where

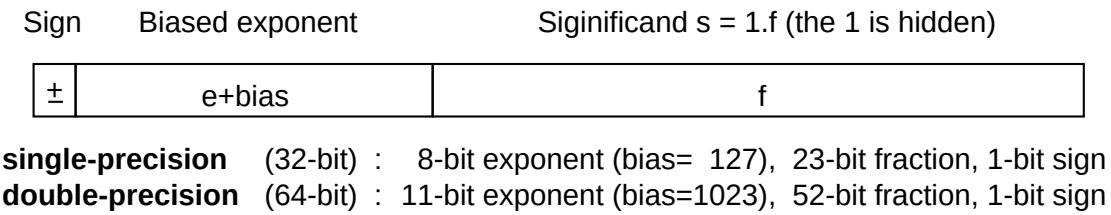

Figure 3.1: IEEE-754 standard floating-point format

sign	number sign (positive or negative),
exponent	the parameter for dynamic range specification,
significand	fractional part.

Then, one floating-point value x is specified as follows:

$$x = \pm 2^{\text{exponent}} \times \text{significand}$$

IEEE-754 standard floating-point formats are used (Figure 3.1) as the default. The standard formats *single-precision* (32-bit) and *double-precision* (64-bit) are correspond to float and double in C/C++ programs.

An addition/subtraction of floating-point numbers consists of a addition for aligned significands and extra handling of sings, exponents, alignment preshift, normalization postshift and special values (0,$\pm\infty$,etc.). A floating-point multiplication/division are relatively simpler since shift operations for alignment are not needed. Many embedded processors and DSPs do not include a floating-point unit due to its unacceptable hardware cost. In this case, fixed-point arithmetics is implemented in software level.

3.3 Fixed-point Numbers

Fixed-point arithmetics is another representation method for real values. The format is the same as integer representation one, except the binary point. Although the dynamic range and the resolution is limited, operations of fixed-point numbers can be executed on an integer unit.

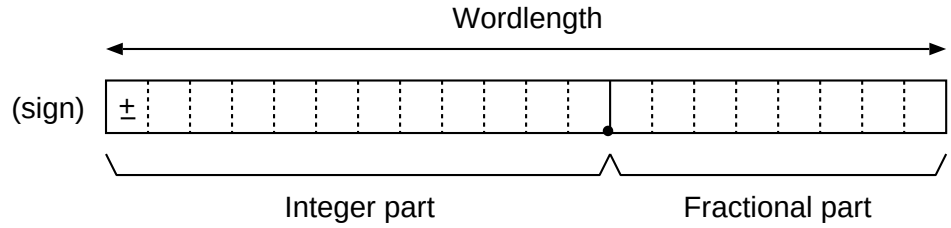

Figure 3.2: Fixed-point format.

The typical format of fixed-point numbers is specified by a 3-tuple $\langle \boldsymbol{sign}, \boldsymbol{wl}, \boldsymbol{iwl} \rangle$, where

$\boldsymbol{sign}$	2's complement representation,
$\boldsymbol{wl}$	word length (the number of bits),
$\boldsymbol{iwl}$	integer word length.

In addition, the wordlength of the fractional part is described as $\boldsymbol{fwl}$ ($wl = iwl + fwl$) (Fig.3.2). In the figure, the specified data type consists of 12-bit integer part and 8-bit fractional part, and so it defines the dynamic range from 2^{-11} to 2^{11} and the resolution of 2^{-8}.

When converting one fixed-point data type instance $\langle \boldsymbol{s_1}, \boldsymbol{wl_1}, \boldsymbol{iwl_1} \rangle$ into another different instance $\langle \boldsymbol{s_2}, \boldsymbol{wl_2}, \boldsymbol{iwl_2} \rangle$, bit-length of fractional part should be cut down in some cases. In the following, *Truncation* is used as the default policy since it requires less bit width.

3.4 Conversion to Fixed-point Numbers and Errors

As described in Section 4.1, the bit-length optimization task is not easy but necessary for the implementation of high-performance and low-power circuit. The most difficult problem in the optimization is computation errors.

We show a simple example of this problem in Figure 3.3. In the example, one real value 0.3 is represented with a floating-point format and fixed-point formats. At the algorithm design, designers pay little attention to dynamic range and precision, because

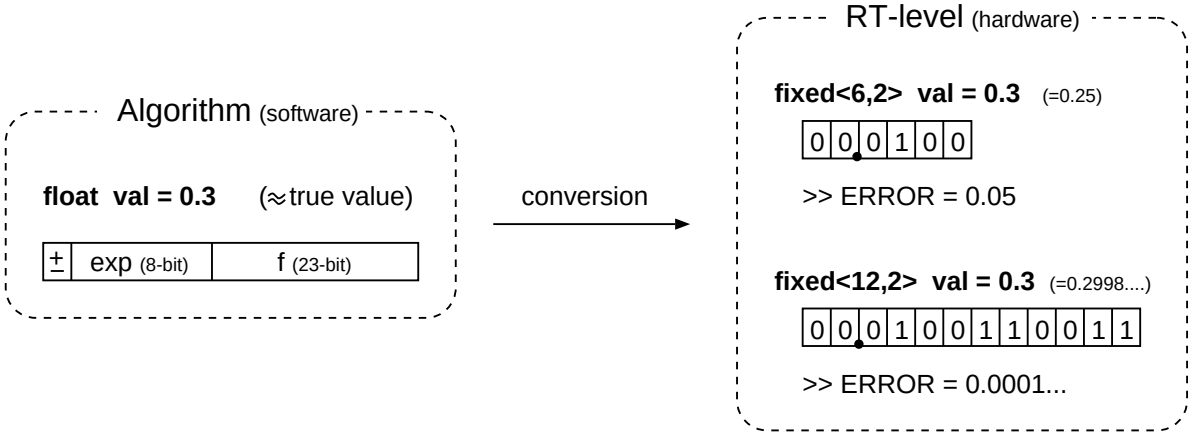

Figure 3.3: Errors after the conversion to fixed-point numbers.

IEEE standard floating-point format provides enough accuracy for practical applications. One real value 0.3 represented with the floating-point format almost equals to a true value. However, fixed-point numbers have some error. The upper one has 2-bit integer part and 4-bit fractional part. The real value 0.3 represented with this format is smaller than that by 0.05. The lower one has some errors too. If the algorithm requires output in high-precision, the fractional part must be longer.

Therefore, designers should determine the bit-length of fractional part in careful. The conversion task is time consuming and error-prone, so many techniques have been proposed to automatize it.

In the FRIDGE project [39, 40], a framework for the conversion is proposed to check the correctness of the specified bit-length of variables using the simulation. Though the static analysis method is partially used to speed up the checking process, a designer still should specify the word length of each register and function unit that is tedious task.

A translator from C programs with floating-point operations to DSP programs with only integer operations was developed by K. Kim et al. [41]. In the translation, several properties of DSPs such as the fixed wordlength unit and operation set are used to reduce the search space for which simulated annealing handles. These properties are too restrictive on the ASIC/FPGA design.

S.Kim et al. proposed a numerical analysis method for IDCT architectures [42], where bit-length of coefficients and adder units are calculated using variance matrix. The method in [43], which is applied for digital filter design, also uses numerical analysis method. It propagates and analyzes errors on the Z transfer function of IIR and FIR. These methods are for application-specific problems and more general frameworks is required.

Static analysis methods [44, 45, 46] are practical one and applicable for general problem. They compute maximum error of each operation in a program, and estimate wordlength of function units which guarantees the specified accuracy. This type of method do the 1-pass estimation using program analysis, so the processing time is far small compared with simulation based approaches. George A. Constantinides et al. presents an approach to the wordlength allocation and optimization problem for linear digital signal processing systems in [44]. The synthesis system "Synoptix" using this approach can receive a Matlab-Simlink[47] block diagram and outputs a structural description in VHDL. In the optimization, floating-point operations are converted to fixed-point ones by using mixed integer programming technique. They also propose enhanced method [45] which is applicable for non-linear systems. The approximation technique based on Tailor expansion is used in the method.

In the [46], a static error analysis technique for the code generation of DSP applications is introduced. The smart interval method *Affine Arithmetic* [48] is used instead of the traditional interval method *Interval Arithmetic* [49] at the error analysis.

We proposed two automatic conversion methods based on static error analysis, **Heuristic resource allocation based method** and **Non-linear programming based method**. Figure 3.4 shows the optimization flow. Two kinds of analysis, *Value range analysis* and *Error analysis*, are applied on a Control/Data-flow graph. The range of each variable is estimated in "Value range analysis" phase, and computational error is estimated in "Error analysis" phase. Then, bit-length optimization algorithms are applied.

Heuristic resource allocation based method (in Chapter 4) employs back propagation technique. The total budget for acceptable error is specified by a designer, then it is rationed to each variable in reverrse order.

Non-linear programming based method (in Chapter 5) formalizes a bit-length optimiza-

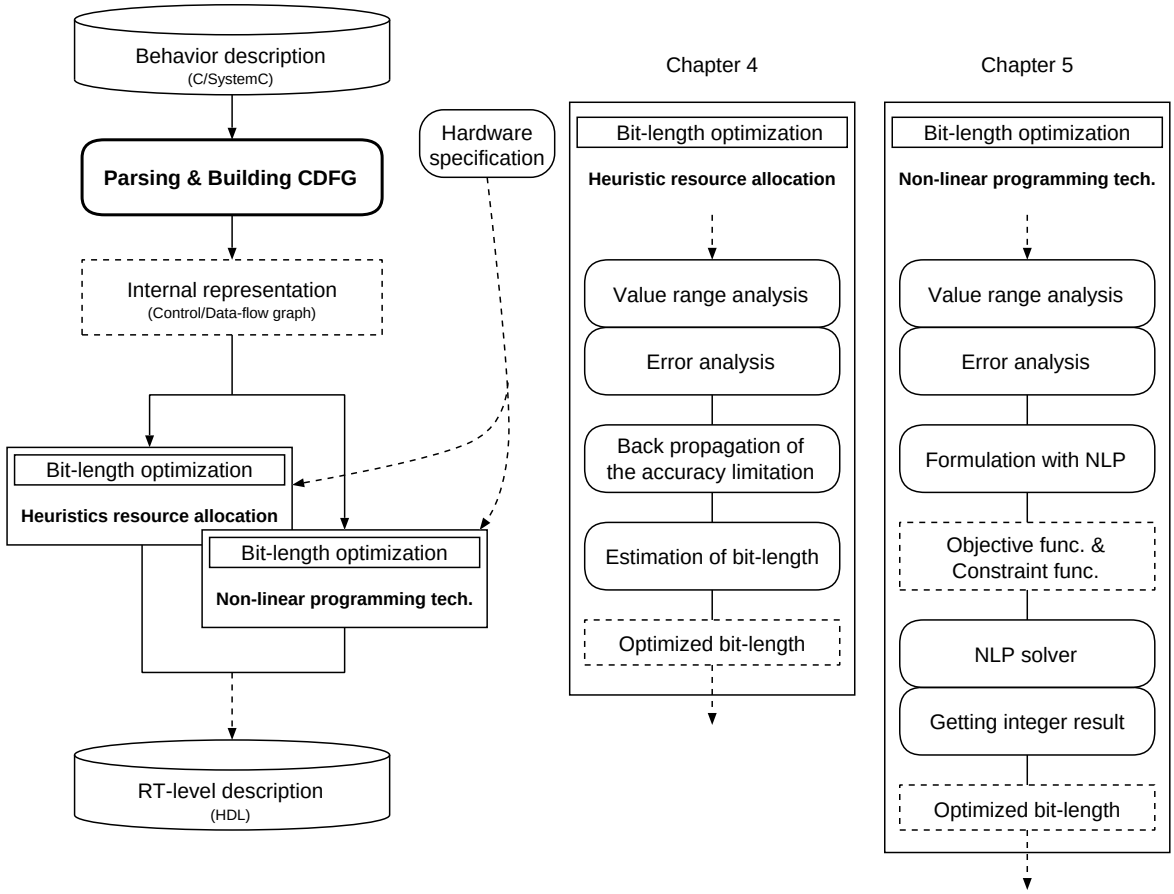

Figure 3.4: Bit-length Optimization Flow.

tion problem with a non-linear programming technique. After the formalization, general non-linear solver can be applied to find bit-length of variables and function units. Since the result of non-linear problem is real value, the extra rounding operation is needed.

Then, a behavior description based on fixed-point arithmetics, which is customized for accurate computation with less hardware, is outputed. The high-level synthesis tool generates a RT-level code from an optimized source code.

The bit-length optimization problem is summarized as follows:

Inputs :

- Control/Data Flow Graph
- Hardware specification that says known bit-length of input/output variables and operation units, acceptable error, maximum hardware area and so on.

Outputs :

- Optimized bit-length of variables and operation units which enable accurate computation, but satisfy area constraints.

Chapter 4

Bit-length Optimization Based on Heuristic Resource Allocation

This chapter presents bit-length optimization method based on heuristic resource allocation. The method analyzes computation errors statistically, and propagate it from input variables to output variables. Then, bit-length of variables are computed from acceptable errors. The budget for acceptable errors of outputs are rationed to each variables according to computation errors. Bit-length of each variable is estimated from each acceptable error.

In this chapter, we describe our error model for the analysis, and explain about optimization algorithm. The optimization algorithm uses back propagation technique that determine the acceptable error for each variable. After that, we shows the optimization results based on our algorithm.

4.1 Error Models

Floating-point operations in an original algorithm are converted to fixed-point ones for speed and area optimizations of synthesized circuits. Since computation errors are generated due to the finite fractional wordlength of the fixed-point format, the bit-length of variables should be long enough for accurate computation. In this section, we present the error model used for the analysis.

Fixed-point variables $X_{\mathrm{i}}(\mathrm{i} = 0, 1, \cdots)$ in a program contain two kinds of errors. One is

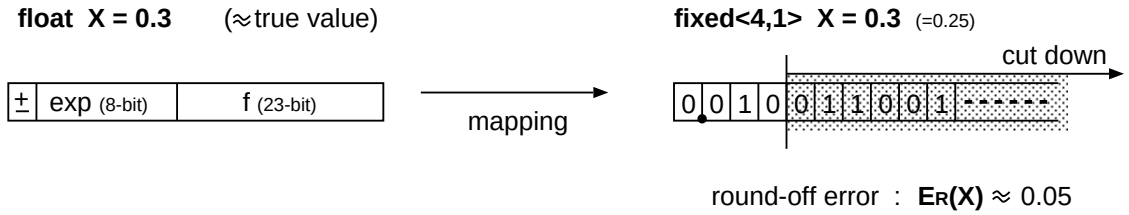

Figure 4.1: Round-off error.

a *round-off error* and the other is a *propagation error*.

Round-off error

A round-off error $E_R(X)$ occurs when a floating-point value ($\approx$ a true value) is mapped on a fixed-point variable (in Figure 4.1). The fractional part is not infinite, so lower bits are cut down after the mapping. The cut down value is the difference between a floating-point value and fixed-point one, and it is called round-off error. For example, the fractional part of the variable X is 3-bits in Figure 4.1. The cut down value is 0.05, so $E_R(X) = 0.05$. Note that rounding is used as the default operation at the mapping.

Propagation error

A propagation error $E_P(X)$ originates in errors of input variables or constants representing real number and is passed via operations. Let $\Delta X_{\mathrm{src1}}, \Delta X_{\mathrm{src2}}$ be total amount of errors of source variables $X_{\mathrm{src1}}, X_{\mathrm{src2}}$, and $E_P(X_{\mathrm{dst}})$ be a propagation error of a destination variable X_{dsr}. On the addition $X_{\mathrm{dsr}} \leftarrow X_{\mathrm{src1}} + X_{\mathrm{src2}}$, not only values, but also errors are propagated to a the destination variable (in Figure 4.2). In this case, the propagation error $E_P(X_{\mathrm{dst}})$ can be computed as $\Delta X_{\mathrm{src1}} + \Delta X_{\mathrm{src2}}$. The propagation error for other operations can be computed in the same way.

Therefore, the error of a variable is modeled as follow:

$$\Delta X \;=\; E_R(X) \;+\; E_P(X).$$

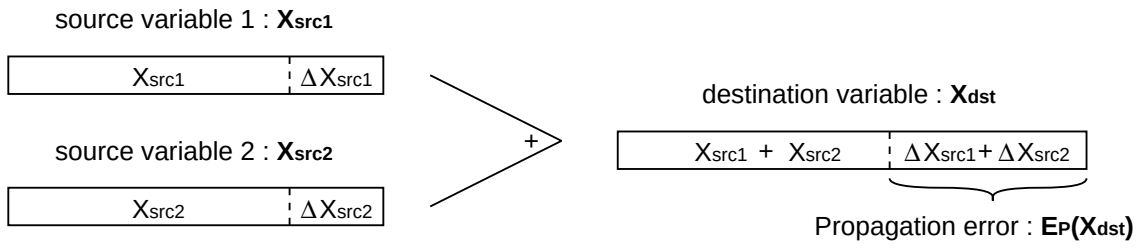

Figure 4.2: Propagation error.

Note that $\Delta X = E_R(X)$ for all input variables and constants because they have no propagation errors.

4.1.1 ε Parameters

Our algorithm based on heuristic resource allocation rations acceptable error to variables according to the amount of its error, and estimates bit-length. However, the error of each variable can not be computed, because it depends on the fractional wordlength of the variable which we want to know. So, we present all errors of variables relatively by using **ε parameters**.

ε parameters are the unit values for the error analysis. All errors of variables are represented with this parameters. ε is a value range expressed as follows:

$$\varepsilon_{\min} \quad < \quad E_R(X_{\mathrm{i}}) \quad \leq \quad \varepsilon_{\max} \ , \quad (X_{\mathrm{i}} \in X_{\mathrm{input}} \cup X_{\mathrm{const}})$$

where $\varepsilon_{\min}$ is the minimum value of the round-off errors of the inputs and the constants, and $\varepsilon_{\max}$ is their maximum value. Each round-off error $E_R(X_{\mathrm{i}})$ of an input or constant is bounded by the uniform values $\varepsilon_{\min}$ and $\varepsilon_{\max}$.

We analyze the propagation errors using these parameterized ranges. In some cases, we use ε for denoting the worst case error in the range. Figure 4.3 shows simple example. The variable X is an input, and its error is bounded by $\varepsilon_{\min}$ and $\varepsilon_{\max}$. The worst case of $\Delta X (= E_R(X))$ is estimated as $\varepsilon_{\max}$. Then, the worst case of the doubled value $2 \cdot X$ is estimated as $2 \cdot \varepsilon_{\max}$, and the one of $50 \cdot X$ is $50 \cdot \varepsilon_{\max}$.

Furthermore, we define $\lceil \ \rceil_\varepsilon$ and $\lfloor \ \rfloor_\varepsilon$. The former indicates the upper bound of ΔX

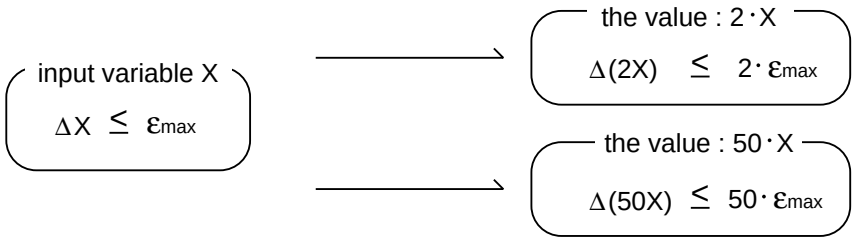

Figure 4.3: The representation of worst case error using ε.

and the latter indicates the lower bound. For example, the worst case errors of an real constant is bounded like,

$$\lceil \Delta X_{\text{const}} \rceil_{\varepsilon} = \varepsilon_{\max}$$

$$\lfloor \Delta X_{\text{const}} \rfloor_{\varepsilon} = \varepsilon_{\min}$$

Note that all errors are denoted using ε like $k \cdot \varepsilon$ (k is a real value).

When the fractional wordlength of input variables or constants is known, a designer can specify round-off error individually.

4.2 Value Range Analysis and Error Analysis

This section presents two pre-analysis for the estimation of fractional wordlength of variables. The first analysis is *Value Range Analysis*. The value range which a variable can take is estimated via the program analysis. The result of this analysis is passed to the next step *Error Analysis*. In this step, the error of each variable is computed by using error propagation technique. After the pre-analysis, bit-length of variables are determined.

4.2.1 Value Range Analysis

Many techniques for analyzing programs and tracing the range of variables have been developed [36, 49], where the interval arithmetic is usually used. We adopt this approach and introduce a mechanism to analyze round-off errors and propagation errors. In the interval arithmetic, each variable has the range represented by the upper and lower bounds,

and a program is executed symbolically using the range. We use X.max as the upper bound of X and X.min as the lower bound of X.

Arithmetic Operations

Each operation in a program is described as a *Three Address Code*. One three address code is described as $X_{\mathrm{dst}} \leftarrow X_{\mathrm{src1}} \circ X_{\mathrm{src2}}$, where $\circ$ is an operation. The result of addition, subtraction, and multiplication increases monotonically according to the increase in the values of source operands. Thus, the upper and lower bounds of X_{dst} can be calculated from the known upper and lower bounds of X_{src1} and X_{src2} as follows:

$$X_{\mathrm{dst}}.\mathrm{max} = \mathrm{Max} \left\{ \begin{array}{lcl} X_{\mathrm{src1}}.\mathrm{max} & \circ & X_{\mathrm{src2}}.\mathrm{max}, \\ X_{\mathrm{src1}}.\mathrm{max} & \circ & X_{\mathrm{src2}}.\mathrm{min}, \\ X_{\mathrm{src1}}.\mathrm{min} & \circ & X_{\mathrm{src2}}.\mathrm{max}, \\ X_{\mathrm{src1}}.\mathrm{min} & \circ & X_{\mathrm{src2}}.\mathrm{min} \end{array} \right\},$$

$$X_{\mathrm{dst}}.\mathrm{min} = \mathrm{Min} \left\{ \begin{array}{lcl} X_{\mathrm{src1}}.\mathrm{max} & \circ & X_{\mathrm{src2}}.\mathrm{max}, \\ X_{\mathrm{src1}}.\mathrm{max} & \circ & X_{\mathrm{src2}}.\mathrm{min}, \\ X_{\mathrm{src1}}.\mathrm{min} & \circ & X_{\mathrm{src2}}.\mathrm{max}, \\ X_{\mathrm{src1}}.\mathrm{min} & \circ & X_{\mathrm{src2}}.\mathrm{min} \end{array} \right\}.$$

The upper and lower bounds of the result of the division can be calculated in the same way, when the range of the divisor does not include 0. If the divisor can take a value of 0, we redefine the maximum and minimum values as follows:

[Case 1]: If the fractional word length of the divisor X_{src2} is n,

$$X_{\mathrm{src2}}.\mathrm{max} = 2^{-n}, \quad X_{\mathrm{src2}}.\mathrm{min} = -2^{-n}.$$

[Case 2]: If the fractional word length of the divisor X_{src2} is not known,

$$X_{\mathrm{src2}}.\mathrm{max} = \varepsilon_{\mathrm{min}}, \quad X_{\mathrm{src2}}.\mathrm{min} = -\varepsilon_{\mathrm{min}}.$$

The definition follows from the fact that X_{dst} takes the maximum value when X_{src1} takes the maximum value and X_{src2} takes the minimum value in the division.

Conditional structures

For a conditional structure (i.e. `if(), then{}, else{}`), we cannot decide which branch is to be taken at compile time. So, both the 'then' part and 'else' part are analyzed and the worst case is adopted.

Let Btrue(X) and Bfalse(X) be the results of the analysis of the 'then' part and of the 'else' part, respectively. The final X after the conditional statement is calculated as follows:

$$\begin{aligned} X.\max &= \mathrm{Max}\{\mathrm{Btrue}(X).\max, \mathrm{Bfalse}(X).\max\}, \\ X.\min &= \mathrm{Min}\{\mathrm{Btrue}(X).\min, \mathrm{Bfalse}(X).\min\}. \end{aligned}$$

Loop structures

Loop structures are categorized into following three types:

1. the number of the loop iteration is known,
2. the number of the loop iteration is not known, but the maximum number of the iteration is known, and
3. there is no information on the number of the iteration.

When a loop structure belongs to type 1 or type 2, it is possible to unfold the loop. We apply our method by unfolding all loop structures. It is impossible to apply our analysis method to type 3 loop structures directly (an extension of the fixed point calculation [36] should be devised in the future).

Let the upper and lower bounds of X after the t-th iteration be $X_{\mathrm{t}}.\max$ and $X_{\mathrm{t}}.\min$, respectively. The upper and lower bounds of the variable X_{dst} after the t-th iteration can be calculated as follows:

$$\begin{aligned} X_{\mathrm{dst}}.\max &= \mathop{\mathrm{Max}}_{1 \le s \le t}\{X_{\mathrm{s}}.\max\}, \\ X_{\mathrm{dst}}.\min &= \mathop{\mathrm{Min}}_{1 \le s \le t}\{X_{\mathrm{s}}.\min\}, \end{aligned}$$

We denote the procedure which calculates upper and lower bounds of X_{dst} from source operands as $\mathrm{RangeMax}(X_{\mathrm{src1}}, \circ, X_{\mathrm{src2}})$ and $\mathrm{RangeMin}(X_{\mathrm{src1}}, \circ, X_{\mathrm{src2}})$.

4.2.2 Error Analysis

Errors of source variables are propagated to the destination variables as mentioned before. For example, if $X_{\mathrm{dst}} = X_{\mathrm{src1}} + X_{\mathrm{src2}}$ and both ΔX_{src1} and ΔX_{src2} are less than ε, $E_P(X_{\mathrm{dsc}})$ is $\Delta X_{\mathrm{src1}} + \Delta X_{\mathrm{src2}}$ and can be estimated as $2 \cdot \varepsilon$ in the worst case.

The amount of errors depends on the operation and the result of the value range analysis. The computation of the propagation of errors is summarized in Table 4.1, where $X.|\mathrm{max}|$ and $X.|\mathrm{min}|$ are defined as follows:

$$
\begin{aligned}
X.|\mathrm{max}| &= \mathrm{Max}\{\mathrm{abs}(X.\mathrm{max}), \mathrm{abs}(X.\mathrm{min})\}, \\
X.|\mathrm{min}| &= \mathrm{Min}\{\mathrm{abs}(X.\mathrm{max}), \mathrm{abs}(X.\mathrm{min})\}.
\end{aligned}
$$

Note that abs() is the absolute function and that $X.|\mathrm{max}| \geq 0$ and $X.|\mathrm{min}| \geq 0$. $X.|\mathrm{max}|$ and $X.|\mathrm{min}|$ are used to estimate the worst case errors.

For division, the propagation error is estimated based on the following formula.

$$
\begin{aligned}
\Delta X_{\mathrm{dst}} &= \frac{X_{\mathrm{src1}}.|\mathrm{max}| + \Delta X_{\mathrm{src1}}}{X_{\mathrm{src2}}.|\mathrm{min}| - \Delta X_{\mathrm{src2}}} - \frac{X_{\mathrm{src1}}.|\mathrm{max}|}{X_{\mathrm{src2}}.|\mathrm{min}|} \\
&= \frac{X_{\mathrm{src1}}.|\mathrm{max}| \cdot \Delta X_{\mathrm{src2}} + X_{\mathrm{src2}}.|\mathrm{min}| \cdot \Delta X_{\mathrm{src1}}}{X_{\mathrm{src2}}.|\mathrm{min}| \cdot (X_{\mathrm{src2}}.|\mathrm{min}| - \Delta X_{\mathrm{src2}})} .
\end{aligned}
$$

By replacing ΔX_{src1} and ΔX_{src2} with $\lceil X_{\mathrm{src2}} \rceil_\varepsilon$ and $\lfloor X_{\mathrm{src2}} \rfloor_\varepsilon$, we obtain the following upper bound:

$$
\begin{aligned}
\Delta X_{\mathrm{dst}} &= \frac{X_{\mathrm{src1}}.|\mathrm{max}| \cdot \Delta X_{\mathrm{src2}} + X_{\mathrm{src2}}.|\mathrm{min}| \cdot \Delta X_{\mathrm{src1}}}{X_{\mathrm{src2}}.|\mathrm{min}| \cdot (X_{\mathrm{src2}}.|\mathrm{min}| - \Delta X_{\mathrm{src2}})} \\
&\leq \frac{X_{\mathrm{src1}}.|\mathrm{max}| \cdot \lceil X_{\mathrm{src2}} \rceil_\varepsilon + X_{\mathrm{src2}}.|\mathrm{min}| \cdot \lceil X_{\mathrm{src1}} \rceil_\varepsilon}{X_{\mathrm{src2}}.|\mathrm{min}| \cdot (X_{\mathrm{src2}}.|\mathrm{min}| - \lfloor X_{\mathrm{src2}} \rfloor_\varepsilon)} .
\end{aligned}
$$

The error propagation function for $X_{\mathrm{src1}} - X_{\mathrm{src2}}$ is not subtract operation, but add operation. All errors are accumulated on positive side by this formalization, since it is required for the bound of worst case errors.

Table 4.1: Error propagation from sources and an operation to a destination

Operation	Magnitude of the propagation error
$X_{\mathrm{src1}} \pm X_{\mathrm{src2}}$	$E_P(X_{\mathrm{dst}}) = \lceil \Delta X_{\mathrm{src1}} \rceil_\varepsilon + \lceil \Delta X_{\mathrm{src2}} \rceil_\varepsilon$
$X_{\mathrm{src1}} \times X_{\mathrm{src2}}$	$E_P(X_{\mathrm{dst}}) = X_{\mathrm{src1}}.\lvert\max\rvert \cdot \lceil \Delta X_{\mathrm{src2}} \rceil_\varepsilon + X_{\mathrm{src2}}.\lvert\max\rvert \cdot \lceil \Delta X_{\mathrm{src1}} \rceil_\varepsilon + \lceil \Delta X_{\mathrm{src1}} \rceil_\varepsilon \cdot \lceil \Delta X_{\mathrm{src2}} \rceil_\varepsilon$
$X_{\mathrm{src1}} \div X_{\mathrm{src2}}$	$E_P(X_{\mathrm{dst}}) = \dfrac{X_{\mathrm{src1}}.\lvert\max\rvert \cdot \lceil X_{\mathrm{src2}} \rceil_\varepsilon + X_{\mathrm{src2}}.\lvert\min\rvert \cdot \lceil X_{\mathrm{src1}} \rceil_\varepsilon}{X_{\mathrm{src2}}.\lvert\min\rvert \cdot (X_{\mathrm{src2}}.\lvert\min\rvert - \lfloor X_{\mathrm{src2}} \rfloor_\varepsilon)}$

We denote the propagation error from the source variables X_{src1} and X_{src2} with their errors and the operation $\circ$ as Propagate$(X_{\mathrm{src1}}, \Delta X_{\mathrm{src1}}, X_{\mathrm{src2}}, \Delta X_{\mathrm{src2}}, \circ)$.

Rounding of temporal variables

By repetition of operations such as multiplication, the required fractional wordlength of temporal variables become longer. Rounding is effective to reduce the size of registers and data paths, while the amount of errors increases. The rounding is acceptable as long as the final accuracy is sufficient.

Figure 4.4 shows a multiplication example ($X_{\mathrm{tmp}} = X_{\mathrm{src1}} \times X_{\mathrm{src2}}$). Since the fractional wordlength of X_{src1} and X_{src2} are n and m respectively, the *fwl* of the operation result X_{tmp} becomes $n + m$. To reduce the size of the register, the fractional part of X_{tmp} is shrunk into 3-bits in the example, and the round-off error ($= \frac{1}{2} \times 2^{-3}$) is additionally introduced.

4.3 Back Propagation of Accuracy Limitation and Estimation of Bit-length

After the pre-analysis steps, estimated errors of variables based on ε is obtained. Bit-length of variables are decided according to them and a hardware specification called *accuracy limitation*. Accuracy limitation indicates acceptable error for output variables,

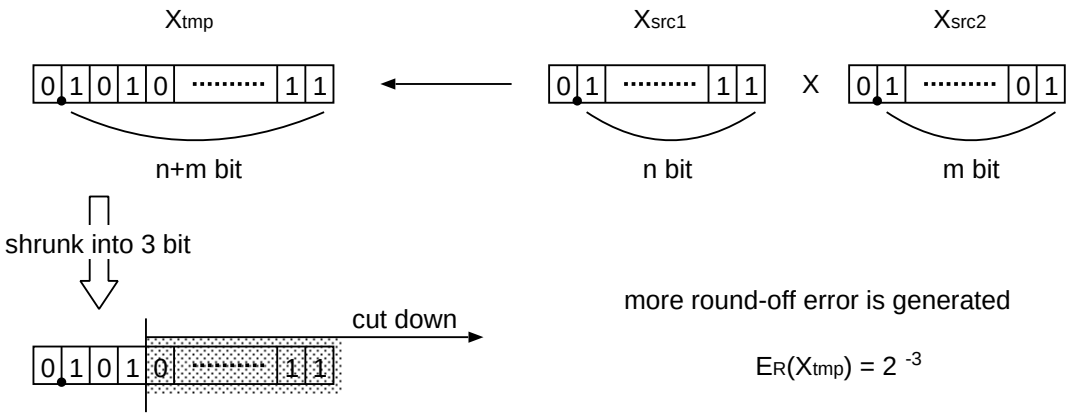

Figure 4.4: The reduction of fractional wordlength for the temporal variable.

and it is a budget for the computation error. We denote it as $\mathrm{Lim}(X_{\mathrm{output}})$. Our algorithm propagates the budget from outputs to inputs in reverse order, and rations it to variables. Then, a fractional wordlength of each variable is computed.

This section describes last two steps of bit-length optimization, 'Back propagation of the accuracy limitation' and 'Estimation of bit-length'.

4.3.1 Back Propagation of Accuracy Limitation

The accuracy limitation of an out put variable $\mathrm{Lim}(X_{\mathrm{output}})$, which is the budget for computation error, is propagated in reverse order. Figure 4.5 is a simple example for an addition operation ($X_{\mathrm{dst}} = X_{\mathrm{src1}} + X_{\mathrm{src2}}$). The budget for the destination variable is 0.5, and it is divided into the budget for $E_P(X_{\mathrm{dst}})$ and the one of $E_R(X_{\mathrm{dst}})$. In this case, 0.125 is allocated to the round-off error and 0.375 remains for the propagation error. That specifies the fractional wordlength of X_{dst} should be long enough on which the worst case error is less than 0.125. The remainder is propagated to source variables as accuracy limitations. $E_P(X_{\mathrm{dst}})$ is $2 \cdot \varepsilon$ and both $E_P(X_{\mathrm{src1}})$ and $E_P(X_{\mathrm{src2}})$ are ε. Thus, $E_P(X_{\mathrm{dst}})$ is divided equally, so both $\mathrm{Lim}(X_{\mathrm{src1}})$ and $\mathrm{Lim}(X_{\mathrm{src2}})$ are $\frac{0.375}{2} = 0.1875$.

There are several methods to determine $E_R(X_{\mathrm{dst}})$, and we use a heuristic method based on the depth of the operation in a DFG. A propagation error is generated when an operation is applied to variables. It becomes large according to the number of operations,

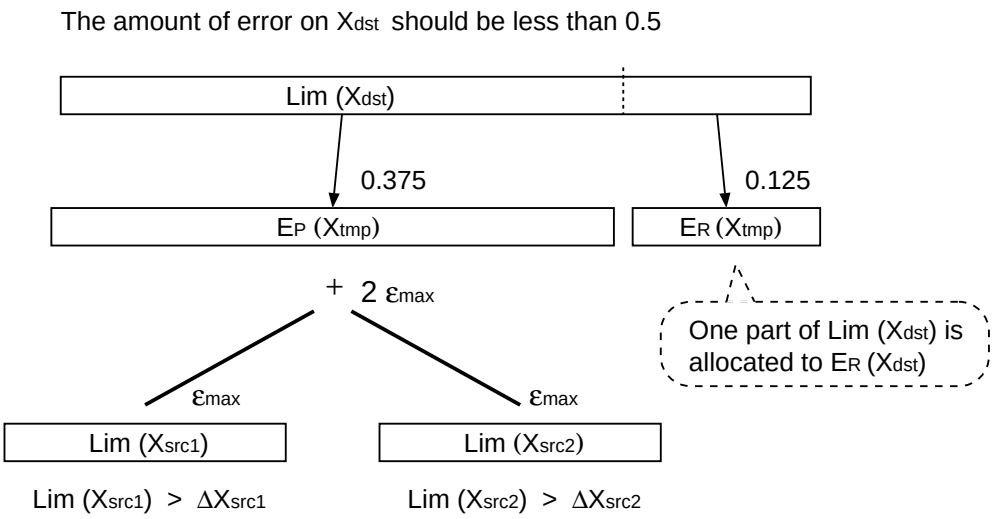

Figure 4.5: Accuracy limitation is propagated in the top-down manner.

and the round-off error becomes relatively small. So, $E_R(X_{\mathrm{dst}})$ is decided as follows:

$$E_R(X_{\mathrm{dst}}) = 2^{-k}$$
$$\text{where } k \text{ is the smallest number}$$
$$\text{such that } 2^{-k} \leq \frac{1}{(\text{depth of data flow})}$$

The outline of this procedure Allocate($\Delta X_{\mathrm{dst}}, \Delta X_{\mathrm{src1}}, \Delta X_{\mathrm{src2}}, \mathrm{Lim}(X_{\mathrm{dst}})$) is summarized as follows:

Input:

- $\Delta X_{\mathrm{dst}}, \Delta X_{\mathrm{src1}}, \Delta X_{\mathrm{src2}}$: error information,
- $\mathrm{Lim}(X_{\mathrm{dst}})$: an accuracy limitation of the destination variable.

Output:

- $E_R(X_{\mathrm{dst}})$: a round-off error of X_{dst},
- $\mathrm{Lim}(X_{\mathrm{src1}}), \mathrm{Lim}(X_{\mathrm{src2}})$: accuracy limitations of source variables.

Calculation of accuracy limitations:

1. $E_R(X_{\mathrm{dst}})$ is decided based on the depth of the data-flow.

2. $E_P(X_{\mathrm{dst}})$ is calculated from $\mathrm{Lim}(X_{\mathrm{dst}})$ and $E_R(X_{\mathrm{dst}})$:

$$E_P(X_{\mathrm{dst}}) = \mathrm{Lim}(X_{\mathrm{dst}}) - E_R(X_{\mathrm{dst}}).$$

3. $\mathrm{Lim}(X_{\mathrm{src1}})$ and $\mathrm{Lim}(X_{\mathrm{src2}})$ are calculated from the rate of ΔX_{dst}, ΔX_{src1} and ΔX_{src2}.

$$\begin{aligned} \mathrm{Lim}(X_{\mathrm{src1}}) &= \frac{\Delta X_{\mathrm{src1}}}{\Delta X_{\mathrm{dst}}} \cdot E_P(X_{\mathrm{dst}}), \\ \mathrm{Lim}(X_{\mathrm{src2}}) &= \frac{\Delta X_{\mathrm{src2}}}{\Delta X_{\mathrm{dst}}} \cdot E_P(X_{\mathrm{dst}}). \end{aligned}$$

   where ΔX_{dst}, ΔX_{src1} and ΔX_{src2} are the estimated errors in the 'Error analysis' step.

4.3.2 Estimation of Bit-length

According to the above three steps, the upper and lower bounds and the round-off errors of all variables are computed from the given accuracy limitations. Bit-length of the integer part and the fractional part for each variable is decided respectively based on these informations

The integer wordlength of a variable X is obtained from the range of its integer part. For example, if $X.\mathrm{max} = 255.0$, $X.\mathrm{min} = 0.0$ and $\mathrm{Lim}(X) = 0.1$, then 8 bits are sufficient to represent any value for the integer part. If the value range includes negative values, 2's complement representation is used, and an additional bit is supplied.

The fractional wordlength is estimated from the round-off error. It is obtained as the minimum k satisfying

$$\frac{1}{2} \cdot 2^{-k} \leq \mathrm{Lim}(X).$$

As for X in the above example, the fractional wordlength becomes 3 bits because $\frac{1}{2} \cdot 2^{-3} \leq 0.1 < \frac{1}{2} \cdot 2^{-2}$. Note that if X is an not input variable or a constant, fractional wordlength may be long because $\mathrm{Lim}(X)$ should be remained partly for the propagation error.

4.4 Implementation and Evaluation

4.4.1 Outline of the Optimization Algorithm

Our optimization algorithm is summarized in Figure 4.6. This algorithm analyzes a Control/Data-flow graph and calculates the fractional wordlength of each variable based on the accuracy information.

At the steps 2 and step 3, DFGs are analyzed in order. The basic block at the beginning of a program is analyzed first. Then, the accuracy limitation is propagated in reverse order as denoted in the above section.

Figure 4.7 is an example given to explain the behavior of our optimization algorithm. This program contains three real constants, receives three integers, and returns Y. The accuracy information is also given. Each row includes 'name of variable', 'input or output', 'variable type', 'upper bound', 'lower bound' and the information about accuracy. The last one indicates 'round-off error' for an input, 'accuracy limitation' for an output. One can see that three integer variables (red, green, blue) take values between 0 to 255 and these variables do not have errors. The accuracy limitation of Y is specified as 0.5.

After the value range analysis, the error ΔY is estimated as at most $765 \cdot \varepsilon_{\max}$, and it should be suppressed under 0.5 (= Lim(Y)). At the root node, Lim(Y) is divided to $E_R(Y)(= 0.125)$ and $E_P(Y)(= 0.375)$. Then, $E_P(Y)$ is propagated to source variables in top-down sequence. The acceptable error of each variable is determined in the back propagation step, and the fractional wordlength is computed based on that. Finally, the total number of bits is estimated as 46 bits, that is the sum of fractional wordlength of variables and constants. The total number of bits becomes 80 bits if we do not use rounding.

4.4.2 Experimental Results

We have implemented the optimization algorithm with C++ language (3000 lines) and SUIF library [50], and applied it to six sample programs: Color space conversion, Sharping filter, FIR filter, 8×8 DCT, Robot arm control and Ray tracing. Table 4.2 shows the properties of the programs.

AI : Accuracy information
$\mathrm{DFG_i}$: One basic block ($\mathrm{i} = 0, 1, ..., n-1$), consist of set of $\mathrm{TA_i}$
$\mathrm{TA_i}$: One three address code ($\mathrm{i} = 0, 1, ..., m-1$)
X_{CDFG} : all variables in a Control/Data-flow graph

BitLength_Optimize (CDFG, AI)
begin
/* initialization */
foreach ($X_{\mathrm{i}} \in X_{CDFG}$)
if information such as range or error is specified by AI
define X_{i}.max, X_{i}.min, ΔX_{i} or $\mathrm{Lim}(X_{\mathrm{i}})$;
end foreach;

/* Pre-definition of round-off error using ε parameter */
foreach ($X_{\mathrm{i}} \in X_{CDFG}$)
if ($X_{\mathrm{i}} \in \{X_{\mathrm{input}}, X_{\mathrm{const}}\} \wedge$ wordlength of X is not known)
/* Round-off errors are assumed with ε parameter */
$E_R(X_{\mathrm{i}}) = \varepsilon$;
end foreach;

/* Step. 1 (Value range analysis)*/
/* Sequentially from head DFG ($\mathrm{DFG_0}$) */
foreach ($\mathrm{DFG_i} \in$ CDFG)
/* Sequentially from head TA ($\mathrm{TA_0}$) */
/* $\mathrm{TA}_i = \{R_{\mathrm{dst}}, \circ, R_{\mathrm{src1}}, R_{\mathrm{src2}}\}$ */
foreach ($\mathrm{TA_j} \in \mathrm{DFG_i}$)
$X_{\mathrm{dst}}.\mathrm{max} = \mathrm{RangeMax}(X_{\mathrm{src1}}, \circ, X_{\mathrm{src2}})$;
$X_{\mathrm{dst}}.\mathrm{min} = \mathrm{RangeMin}(X_{\mathrm{src1}}, \circ, X_{\mathrm{src2}})$;
end foreach;
end foreach;

/* Step. 2 (Error analysis) */
/* Sequentially from head DFG ($\mathrm{DFG_0}$) */
foreach ($\mathrm{DFG_i} \in$ CDFG)
/* Sequentially from head TA ($\mathrm{TA_0}$) */
foreach ($\mathrm{TA_j} \in \mathrm{DFG_i}$)
$E_P(X_{\mathrm{dst}}) = \mathrm{Propagate}(X_{\mathrm{src1}}, \Delta X_{\mathrm{src1}}, X_{\mathrm{src2}}, \Delta X_{\mathrm{src2}}, \circ)$;
end foreach;
end foreach;

```
    /* Step. 3 (Back propagation of accuracy limitation) */
    /* Sequentially from tail DFG (DFG_{n-1}) */
    foreach (DFG_i ∈ CDFG)
       /* Sequentially from tail TA (TA_{m-1}) */
       foreach (TA_j ∈ DFG_i)
          Allocate(ΔX_dst, ΔX_src1, ΔX_src2, Lim(X_dst))
       end foreach;
    end foreach;

    /* Step. 4 (Estimation of bit-length) */
    foreach (X_i ∈ X_CDFG)
          /* Decide iwl and fwl */
          Estimate iwl and fwl of X_i based on X_i.max, X_i.min and Lim(X_i)
    end foreach;
end
```

Figure 4.6: Outline of the optimization algorithm

Table 4.2: Source programs for the experiments

	line	unfold	#var	number of inputs
RGB2YCrCb	10	10	8	3 (8-bit×3)
Sharping filter	25	25	12	9 (12-bit×9)
FIR (11 point)	42	1094	32	1 (16-bit×1)
8×8 DCT	68	11001	29	64 (16-bit×64)
Robot arm control	173	487	41	6 (6-bit×4, 8-bit×2)
Ray tracing	213	2013	67	8 (8-bit×7, 9-bit×1)

We compared the results of our algorithm with two approaches based on the simulation. 'Binary based search' is a heuristic method to decide the word length of all variables using binary search. For example, if the initial word length is 32 and the optimum one is 12, then

Source Program

```
funcY(){
   int red, green, blue;
   float tmp0,tmp1,tmp2,tmp3,Y;

   tmp0 = 0.29900 * red
   tmp1 = 0.58700 * green
   tmp2 = 0.11400 * blue
   tmp3 = tmp0 + tmp1
   Y = tmp2 + tmp3
}
```

Accuracy Information

red	IN	int	255	0	0
green	IN	int	255	0	0
blue	IN	int	255	0	0
Y	OUT	double	255	0	0.5

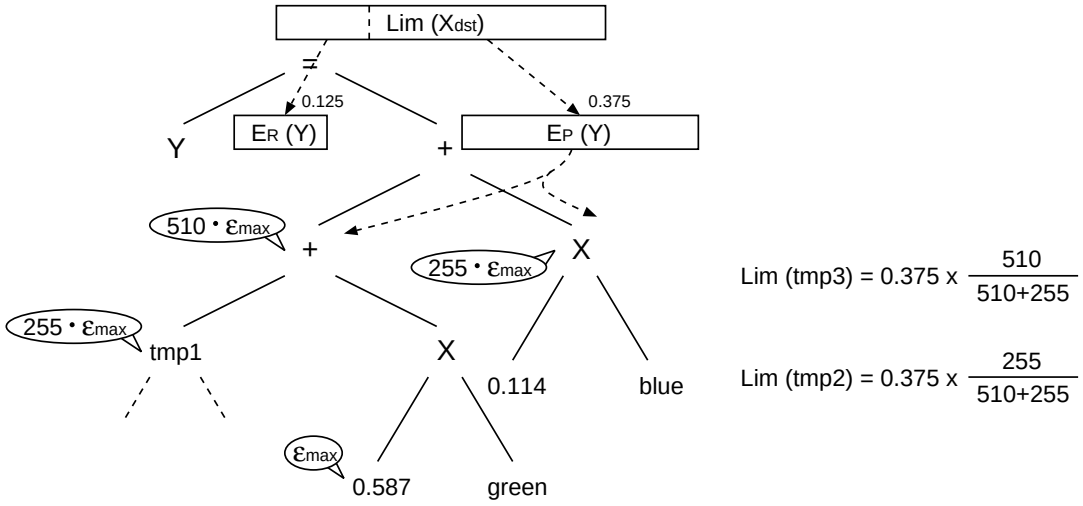

Figure 4.7: Color space conversion from RGB to YCrCb

we can obtain the optimum value by searching 32, 16(=32/2), 8(=16/2), and 12(=8+(16-8/2)). We should check the acceptability of each wordlength using the simulation. After the search, we perform the adjustment by reducing the length of each variable one by one in the predefined order.

'Manual optimization' is a method to optimize the word length manually. The designer analyzes the program, and uses the knowledge to reduce the word length. For each setting,

Table 4.3: Results of color space conversion.

Var name	Manual optimization	Binary-based search	Our algorithm
0.299	10 bit	9 bit	12 bit
0.587	10 bit	7 bit	12 bit
0.114	10 bit	8 bit	12 bit
tmp0	2 bit	8 bit	2 bit
tmp1	2 bit	7 bit	2 bit
tmp2	2 bit	8 bit	2 bit
tmp3	2 bit	8 bit	2 bit
Y	2 bit	8 bit	2 bit
Total Bits	40 bit	63 bit	46 bit

we should check the correctness using the simulation. The designer can use the binary search at the first stage, and try various combinations at the second stage. The result depends on the skill of the designer and the size of the program. For large programs, it is hard to apply the method.

The check using the simulation is the comparison of the computation result of the program using floating variables and that using fixed-point variables. The simulation should be performed for all input patterns for the exact check, but that is difficult since the number of input patterns becomes huge. So we use 1,000,000 random patterns for the simulation if the number of all input patterns is larger than 1,000,000. In the sample programs, we can do the exact check for only the RGB2YCrCb conversion.

Table 4.3 shows the optimization result of the RGB2YCrCb conversion. The table shows the number of bits of the fractional part of each variable/constant. Total bits shows the summation of the number of bits. Note that the number of bits of all variables is almost the same when using the binary based search.

The summary of the experiments is shown in Table 4.4. Total bits are the number of bits of all variables, and time is the CPU time. The time is measured using the `time` command of linux on a Pentium IV 2.4 GHz with 512 MB memory. As mentioned

Table 4.4: Results of bit length optimization

Program Name	Manual optimization		Binary-based search		Our algorithm	
	Total Bits	Time	Total Bits	Time	Total Bits	Time
RGB2YCrCb	40 bit	30 min	63 bit	95.2 sec	46 bit	0.01 sec
Sharping Filter	33 bit*	30 min*	36 bit*	83.6 sec*	44 bit	0.03 sec
FIR filter	153 bit*	45 min*	94 bit*	273.5 sec*	172 bit	0.35 sec
8×8 DCT	—	—	—	—	496 bit	60.54 sec
Robot arm	—	—	—	—	1703 bit	0.89 sec
Ray trace	—	—	—	—	992 bit	4.12 sec

before, the simulation is done for 1,000,000 random patterns for programs other than RGB2YCrCb, and the result is not exact. "*" denotes that. "-" in the table shows that we could not obtain the results within 24 hours. That is because of the repetition of the simulation for 1,00,000 input patterns.

From the table, we can see that the estimation results using our algorithm are comparable with those of the other two methods, and our algorithm is useful for large programs. The result of the FIR filter using the binary based method is much smaller compared to those of the other methods. We have checked the result and found that the result is incorrect. That is due to the simulation with limited random data.

For RGB2YCrCb, we tried to apply the full exhaustive search on the combination of the bit length. The length of each variable can vary 1,2, ..., 12 bits, and there are 8 variables and we should check 12^8 cases. It took about 53 hours to check 12^6 cases, and we gave up. Note that the exhaustive search is prohibitive for such a small problem.

From the results, we can conclude that our algorithm is useful for the estimation of the fractional wordlength variables. Since the algorithm does not require the simulation with huge input patterns, it is very fast. The obtained bit-length might be an over-estimation, but the manual optimization becomes easier by using the result of our algorithm as the initial length.

The hardware cost for the optimization program with fixed-point variables is very small compared to those of initial programs with floating-point variables, since the area and delay of the fixed-point units are far smaller than those of the floating-point units. Therefore, in many cases we need not consider the sharing of fixed-point units (especially when using FPGA). As an example, we compared the hardware cost of optimized and non-optimized ones using RGB2YCrCb. We have synthesized RTL codes with ALTERA Quartus II to EP20K400BC652-1X FPGA. RGB2YCrCb with fixed-point variables uses only 9 logic elements with a 2.4 nsec delay, but that with a floating-point variables requires 3127 logic elements with a 207.6 nsec delay. Note that both circuits are combinational, and the circuit with floating-point variables would be small by sharing the operation units but such sharing is useless after the conversion to fixed-point variables.

4.5 Concluding Remarks

We describe an optimization algorithm of the fractional wordlength of fixed-point variables in the high-level synthesis from C programs including floating-point variables. The algorithm propagates the error information, decides the optimum wordlength, and automatically converts floating-point variables to fixed-point variables.

We have implemented the optimization algorithm and applied it to six sample programs. The results show that our algorithm is effective for obtaining an upper bound of the necessary wordlength. The algorithm does not require the simulation with huge input data, so the execution is very fast and applicable to large programs.

The result of our algorithm might be an over-estimation, but is very useful for prototyping designs and for the initial setting in the simulation-based method.

We have shown a basic idea for the estimation of the fractional wordlength. Refinement of the algorithm would be needed. One of the important problems is the manipulation of the sharing of operation units. As shown in the RGB2YCrCb example, the sharing is useful for the floating-point units since the size is very large, but that has little effect on the fixed-point units. Total optimization including the sharing is very difficult but is an interesting problems. Developing an analysis method for programs including elementary

functions such as sin, cos, exp is another problem.

Chapter 5

Exact Bit-length Optimization Based on Non-linear Programming

This chapter presents exact bit-length optimization method based on non-linear programming. The method formalizes a bit-length optimization problem as a non-linear problem. By the formulation, various constraints can be handled in easy. The results of a formalized non-linear problem are real value, so the extra processing to find integer results is needed.

In this chapter, we show the motivative example first. Then, we describe our error model *positive and negative error model* devised for the formulation. The bit-length optimization is formalized using this error model, and a general non-linear programming solver is applied. We also refer to the operation unit sharing.

5.1 Motivative Example

We use following equation as an example to explain the behavior of the optimization algorithm.

$$\mathsf{Cr} = -0.1684 * \mathsf{red} - 0.3316 * \mathsf{green} + 0.5 * \mathsf{blue}$$

Figure 5.1 is a C program of the equation. The program is a part of the color space conversion from RGB to YCrCb. It includes three inputs (red,green,blue), one output (Cr) and four intermediate variables (tmp0,tmp1,tmp2,tmp3). The program also includes three real constants defined as double precision floating-point number. These constants

Source Program

```
funcCr(){
   int red,green,blue;
   double tmp0,tmp1,tmp2,tmp3,Cr;

   tmp0 = 0.1684 * red;
   tmp1 = 0.3316 * green;
   tmp2 = 0.5 * blue;
   tmp3 = tmp0 + tmp1;
   Cr = tmp2 - tmp3;
}
```

Figure 5.1: Source program of color space conversion

should be converted to fixed-point ones for hardware implementation. Our algorithm finds the minimum fractional wordlength of each variable required for accurate computation.

The following specifications are given as an accuracy information of the program:

- Each input variable (red,green,blue) is 8 bit integer and takes values between 0 to 255. These variables are integers and do not have errors.
- The output variable (Cr) is real value and takes values between 0 to 255. The error in Cr should be less than ± 0.5.

In the next section, *positive and negative error model* is presented which is refined for the non-linear programming formulation. Then, optimization steps including NLP formulation are described.

5.2 Positive and Negative Error Model

The error model used in heuristic resource allocation method (in Chapter 4) is useful for the representation of the worst case. However, the accuracy of error estimation is not good because all errors are accumulated on positive side. So, we refine the error model.

Round-off error

The error model called 'positive and negative error model' captures an error as a value range. Let X be a fixed-point variable and L_X be its fractional wordlength. It contains two kinds of errors, round-off error and propagation error, as described in Chapter 4. The round-off error $E_R(X)$ is in the range from 0 to 2^{-L_X}. Note that *truncation* is used as the default rounding operations for this error model. The error based positive and negative error model is denoted as the pair of values like

$$\Delta X \;=\; \left[\; \Delta^{-}X \;,\; \Delta^{+}X \;\right],$$

where $\Delta^{+}X$ is the worst case error in positive side (= *positive error*) and $\Delta^{-}X$ is the worst case error in negative side (= *negative error*). $\Delta^{+}X$ always takes positive value and $\Delta^{-}X$ always takes negative one. So $E_R(X)$ can be represented like $[\ 0,\ 2^{-L_X}]$. In the case of $E_R(0-X)$, the round-off error can be represented like $[-2^{-L_X},\ 0\ ]$.

Propagation error

The definition of propagation error is the same as old one except that each error is denoted as the range. Let $\Delta X_{\mathrm{src1}}, \Delta X_{\mathrm{src2}}$ be errors' range of source variables $X_{\mathrm{src1}}, X_{\mathrm{src2}}$, and $E_P(X_{\mathrm{dst}})$ be a propagation error of a destination variable X_{dst}. On the addition $X_{\mathrm{dst}} \leftarrow X_{\mathrm{src1}} + X_{\mathrm{src2}}$, $E_P(X_{\mathrm{dst}})$ can be computed as $\Delta X_{\mathrm{src1}} + \Delta X_{\mathrm{src2}}$. Since both ΔX_{src1} and ΔX_{src2} are value ranges, interval arithmetic rules are applies for the computation of the propagation error. The propagation error is also denoted as a value range. In this case, a positive error and a negative error of $E_P(X_{\mathrm{dst}})$ are computed respectively. The worst case on a positive side can be computed as $\Delta X^{+}_{\mathrm{src1}} + \Delta X^{+}_{\mathrm{src2}}$, and $\Delta X^{-}_{\mathrm{src1}} + \Delta X^{-}_{\mathrm{src2}}$ for a negative error. Therefore, the value range of $E_P(X_{\mathrm{dst}})$ is calculated as follow:

$$\begin{aligned} E_P(X_{\mathrm{dst}}) &= \Delta X_{\mathrm{src1}} + \Delta X_{\mathrm{src2}} \\ &= \left[\; \Delta^{-}X_{\mathrm{src1}} + \Delta^{-}X_{\mathrm{src2}},\ \Delta^{+}X_{\mathrm{src1}} + \Delta^{+}X_{\mathrm{src2}} \;\right] \end{aligned}$$

5.3 Value Range Analysis and Error Analysis

A typical C program have not only arithmetic operations, but also control structures such as if-else and for. Then, the program analysis is significant as the pre-analysis for error estimation. The informations about variables and control flows which are obtained by this step are used in the next step 'Error analysis'. Many techniques for program analysis or tracing of a variable range have been developed, where the interval calculation approach is usually used [36, 49].

In the interval calculation, each variable has the range of possible value represented by the lower and upper bounds, and a program is executed symbolically based on the range. For example, one variable X can be represented like $[X.\min, X.\max]$, where $X.\min$ is the lower bound and $X.\max$ is the upper bound.

An expression in a basic block is described as '$X_{\mathrm{dst}} \leftarrow X_{\mathrm{src1}} \circ X_{\mathrm{src2}}$', where X_{src1} and X_{src2} are source operands, $\circ$ is an arithmetic operation and X_{dst} is the result of the operation. The range of the destination variable X_{dst} can be calculated from known range of source variables (X_{src1} and X_{src2}) and an arithmetic operation as shown in [36].

When source variables of the expression (X_{src1} and/or X_{src2}) include some error, the error is propagated to X_{dst}. That is called *propagation error*. The errors of primary inputs and constants are propagated to primary outputs sequentially. We trace the behavior of error propagation and estimate the amount of errors of primary outputs.

In our previous work the algorithm propagates only positive error. Thus, all errors of variables are positive and it makes error estimation inaccurate. In this chapter, the error is denoted as the pair of values and the error in final results can take both of positive and negative value.

Let $E_P(X)$ be the propagation error from source variables and ΔX be the amount of the error of X. In this section, we show the behaviors of error propagation about 'Addition', 'Subtraction' and 'Constant multiplication' as the examples.

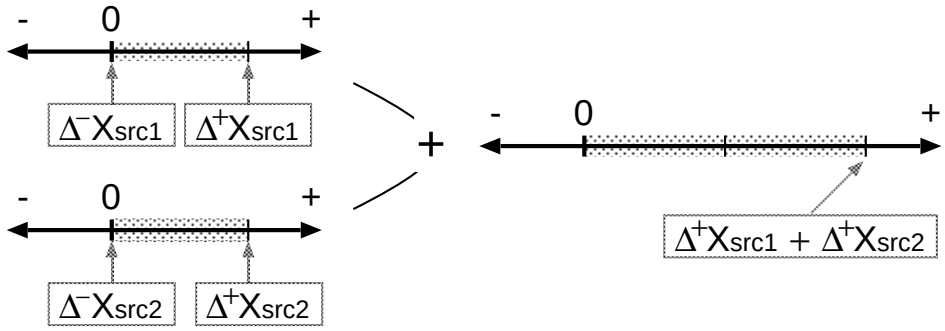

Figure 5.2: Propagation error of addition.

Addition : $X_{\mathbf{dst}} = X_{\mathbf{src1}} + X_{\mathbf{src2}}$

In this case, two positive errors and negative errors are summed up respectively. So, the propagation error of X is calculated as follows:

$$\begin{aligned} E_P(X_{\mathrm{dst}}) &= \Delta X_{\mathrm{src1}} + \Delta X_{\mathrm{src2}} \\ &= \left[\Delta^- X_{\mathrm{src1}} + \Delta^- X_{\mathrm{src2}},\ \Delta^+ X_{\mathrm{src1}} + \Delta^+ X_{\mathrm{src2}} \right] \end{aligned}$$

Subtraction: $X_{\mathbf{dst}} = X_{\mathbf{src1}} - X_{\mathbf{src2}}$

In this case, positive and negative errors of X_{src2} are replaced each other. Then, the propagation error of X is calculated as follows:

$$\begin{aligned} E_P(X_{\mathrm{dst}}) &= \Delta X_{\mathrm{src1}} - \Delta X_{\mathrm{src2}} \\ &= \left[\Delta^- X_{\mathrm{src1}} - \Delta^+ X_{\mathrm{src2}},\ \Delta^+ X_{\mathrm{src1}} - \Delta^- X_{\mathrm{src2}} \right] \end{aligned}$$

Constant multiplication : $X_{\mathbf{dst}} = A * X_{\mathbf{src1}}$

In this case, both of positive and negative errors are multiplied by constant A. So, the propagation error of X is calculated as follows:

$$E_P(X_{\mathrm{dst}}) = \left[A * \Delta^- X_{\mathrm{src1}},\ A * \Delta^+ X_{\mathrm{src1}} \right]$$

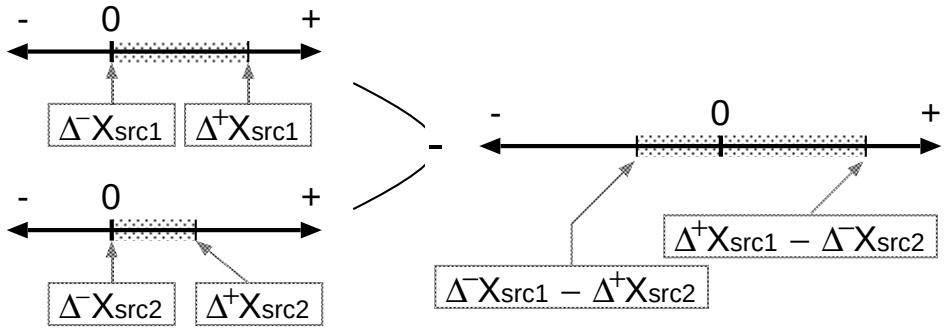

Figure 5.3: Propagation error of subtraction.

The amount of errors depends on the operation and value range of variables. The computation of the propagated error is summarized in Table 5.1, where the algorithm propagates the error range.

As for division, the propagation error can be drawn as follows:

$$\begin{aligned} X_{\mathrm{dst}} + \Delta X_{\mathrm{dst}} &= \frac{X_{\mathrm{src1}} + \Delta X_{\mathrm{src1}}}{X_{\mathrm{src2}} + \Delta X_{\mathrm{src2}}} \\ \Delta X_{\mathrm{dst}} &= \frac{X_{\mathrm{src1}} + \Delta X_{\mathrm{src1}}}{X_{\mathrm{src2}} + \Delta X_{\mathrm{src2}}} - \frac{X_{\mathrm{src1}}}{X_{\mathrm{src2}}} \\ &\leq \frac{X_{\mathrm{src1}} + \Delta X_{\mathrm{src1}}}{X_{\mathrm{src2}}} - \frac{X_{\mathrm{src1}}}{X_{\mathrm{src2}}} \\ &\leq \frac{\Delta X_{\mathrm{src1}}}{X_{\mathrm{src2}}} \quad . \end{aligned}$$

Note that ΔX_{dst} takes the largest value when X_{src1} takes the maximum value and X_{src2} takes the minimum value. To simplify the equation, ΔX_{src2} in the denominator is ignored. Since ΔX_{src2} is far smaller than X_{src2}, the function *Propagate*() still represents the error in the worst case.

The rounding of temporal variables is also applied. The long fractional wordlength may be cut down. For example, the fractional part of X_{tmp} is reduced to 3 bits. The round-off error is newly introduced and it is denoted as $E_R(X_{\mathrm{tmp}}) = [0,\ 2^{-3}]$.

Table 5.1: Propagation error from sources and an operation to destination.

Operation	Magnitude of the propagation error
$X_{\mathrm{src1}} + X_{\mathrm{src2}}$	$E_P(X_{\mathrm{dst}}) = \Delta X_{\mathrm{src1}} + \Delta X_{\mathrm{src2}}$
$X_{\mathrm{src1}} - X_{\mathrm{src2}}$	$E_P(X_{\mathrm{dst}}) = \Delta X_{\mathrm{src1}} - \Delta X_{\mathrm{src2}}$
$X_{\mathrm{src1}} \times X_{\mathrm{src2}}$	$E_P(X_{\mathrm{dst}}) = X_{\mathrm{src1}} \cdot \Delta X_{\mathrm{src2}} + X_{\mathrm{src2}} \cdot \Delta X_{\mathrm{src1}} + \Delta X_{\mathrm{src1}} \cdot \Delta X_{\mathrm{src2}}$
$X_{\mathrm{src1}} \div X_{\mathrm{src2}}$	$E_P(X_{\mathrm{dst}}) = \dfrac{\Delta X_{\mathrm{src1}}}{X_{\mathrm{src2}}}$ $\left(\text{If the devider takes 0} \quad \begin{smallmatrix} X_{\mathrm{src2}}.\max = & 2^{-L_{\mathrm{src2}}} \\ X_{\mathrm{src2}}.\min = & -2^{-L_{\mathrm{src2}}} \end{smallmatrix}\right)$

5.4 Formulation with Non-linear Programming

After the pre-analysis, positive and negative values of an error in a primary output is computed. These values are represented as a polynomial of $2^{-L_i}(i = 0, 1, \cdots, m-1)$, where L_i is a fractional wordlength of a variable. A general non-linear solver is applied for minimizing the fractional wordlength of variables.

Let $O_j(j = 0, 1, \cdots, n-1)$ be primary outputs and $\mathrm{Lim}(O_j)$ be an accuracy limitation, which is the border value of the error ΔO_j. The estimated error can be described as follows:

$$\begin{aligned} \Delta O_j &= \left[\ \Delta^- O_j\ ,\ \Delta^+ O_j\ \right], \\ &= \left[\ F_0^-(2^{-L_0}, 2^{-L_1}, \cdots, 2^{-L_{m-1}})\ ,\right. \\ &\qquad \left. F_0^+(2^{-L_0}, 2^{-L_1}, \cdots, 2^{-L_{m-1}})\ \right]. \end{aligned}$$

where F_0^- and F_0^+ are the polynomials representing lower and upper bounds. I other word, F_0^- represents the worst case error in negative side and F_0^+ represents the worst-case error in positive side.

It is specified that ΔO_j should not exceed its accuracy limitation $\mathrm{Lim}(O_j)$. Then, the following conditional expressions are drawn:

$$\begin{aligned} -\mathrm{Lim}(O_j) &< \Delta^- O_j = F_j^-(2^{-L_0}, 2^{-L_1}, \cdots, 2^{-L_{m-1}}), \\ \mathrm{Lim}(O_j) &> \Delta^+ O_j = F_j^+(2^{-L_0}, 2^{-L_1}, \cdots, 2^{-L_{m-1}}). \end{aligned}$$

They are considered as conditional functions for the non-linear problem.

The total bits of fractional part is the measure of the hardware area. So, the goal of the minimization of the fractional wordlength is formulated as follows:

$$\text{Minimize} \sum_{i=0}^{m-1} L_i$$

One of non-linear solvers 'SQP method' [51] is applied to solve the problem. As the result of the non-linear problem, we can get $L_i (i = 0, 1, \cdots, n-1)$ in real value. Since the fractional wordlength L_i should be an integer, these values are revaluated.

5.4.1 An Application Example

In this section, we show the behavior of optimization algorithm on the example program.

First, we focus on the errors in three real constants {0.1684, 0.3316, 0.5}. When the constants are represented as fixed-point number having $\{2^{-L_0}, 2^{-L_1}, 2^{-L_2}\}$ fractional wordlength, the round-off errors are represented as

$$\begin{aligned}
\Delta X_{0.1684} &= E_R(X_{0.1684}) = \left[\, 0 \,,\, 2^{-L_0} \right], \\
\Delta X_{0.3316} &= E_R(X_{0.3316}) = \left[\, 0 \,,\, 2^{-L_1} \right], \\
\Delta X_{0.5} &= E_R(X_{0.5}) = \left[\, 0 \,,\, 2^{-L_2} \right].
\end{aligned}$$

Variables tmp0,tmp1,tmp2 are the results of multiplication of primary inputs and constants. Each primary input takes values from 0 to 255. Then, the error of intermediate variables are written as

$$\begin{aligned}
\Delta X_{\text{tmp0}} &= [0, 255] \cdot \Delta X_{0.1684} + \left[\, 0 \,,\, 2^{-L_3} \right], \\
\Delta X_{\text{tmp1}} &= [0, 255] \cdot \Delta X_{0.3316} + \left[\, 0 \,,\, 2^{-L_4} \right], \\
\Delta X_{\text{tmp2}} &= [0, 255] \cdot \Delta X_{0.5} + \left[\, 0 \,,\, 2^{-L_5} \right].
\end{aligned}$$

The error range of tmp3 is calculated from the error range of tmp0 and tmp1, thus the range is given by

$$\Delta X_{\text{tmp3}} = \left[\, 0 \,,\, 255 \cdot (\, 2^{-L_0} + \, 2^{-L_1}) + 2^{-L_3} + 2^{-L_4} \right].$$

The primary output Cr is the result of subtraction of tmp2 and tmp3. Then, the worst-case error in positive field is give by

$$\Delta^{+}Cr \;=\; 255 \cdot 2^{-L_2} + 2^{-L_5}.$$

In the same way, the worst-case error in negative field is given by

$$\Delta^{-}Cr \;=\; -255 \cdot (2^{-L_0} + 2^{-L_1}) - 2^{-L_3} - 2^{-L_4}.$$

The error in the final result should not exceed its accuracy limitation $\mathrm{Lim}(Cr)$, thus the following constrains are drawn:

$$0.5 \;>\; \Delta^{+}Cr,$$

$$-0.5 \;<\; \Delta^{-}Cr.$$

The bit-length optimization problem is reduced to a non-linear problem and the non-linear solver is applied.

5.5 Getting Integer Result

By the formulation, a general non-linear solver can be applied, and the fractional wordlength of each variable is determined. However, the result of non-linear programming is real value in may cases. So, it is re-processed to get integer result.

Following re-processing methods are available:

- **Round-up :** All real values in the result are rounded-up. Though the result is overestimation, it guarantees computation accuracy.
- **Heuristics search :** For each fractional wordlength, rounding-up or truncation is applied, then the correctness is checked. The result which guarantees computation accuracy and occupies least area is selected.

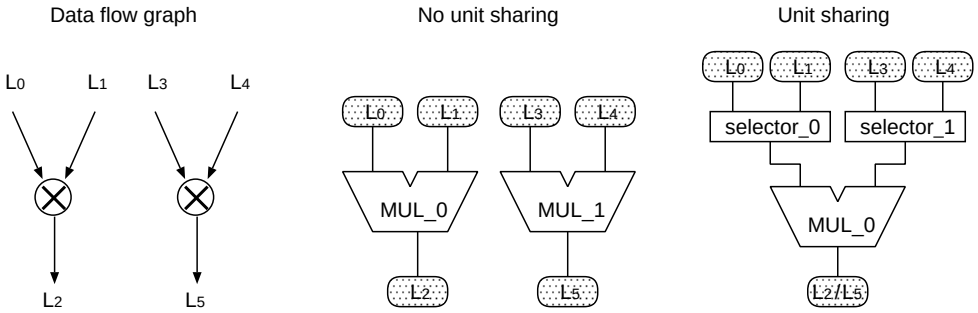

Figure 5.4: Sharing of a multiplier.

5.6 Operation Unit Sharing

In the previous section, we present the optimization flow using non-linear programming technique. The constraints for computation accuracies are specified as equalities and inequalities constraints of non-linear problem. The constraint for sharing can be specified in the same way.

Figure 5.4 is a simple sharing example, where two multiplications are assigned to the same multiplier. For the assignment, fractional wordlength of inputs and outputs should be the same. Thus, the following constrains should be included,

$$\begin{aligned} L_0 &= L_3, \\ L_1 &= L_4, \\ L_2 &= L_5. \end{aligned}$$

The optimization with sharing of operation units can be performed by the non-linear formulation with these constraints.

However, there are two approaches for optimization including sharing of operation units.

- The constraints for sharing are specified as equality constraints of non-linear problem. The optimization results are the final results.
- The bit-length optimization is performed without sharing of operation units. After

the optimization, the algorithm finds operation set those are assigned on the same unit.

In the sharing, it seems to be more effective to share operations with almost the same bit length, so the second approach is promising. There are several works on the second approach as in [52].

Note that we can combine these two method: at first we apply the second method to obtain the bit length without sharing, then we apply the first method to obtain the bit length with sharing.

5.7 Implementation and Evaluation

5.7.1 Implementation Detail

We have implemented the optimization algorithm with C++ language (5000 lines), System C library [15], Antlr parser library[53] and DONLP2 library[54]. Then, we applied it two sample programs: Color space conversion (example program) and FIR filter.

We compared the results of proposal algorithm with those of other two approaches 'Manual optimization' and 'Our previous algorithm'. In the 'Manual optimization', the designer analyzes the program and uses the knowledge to reduce fractional wordlength. The designer check the correctness of optimization result by using exhaustive simulation. It is impractical to use exhaustive simulation for a large program, input patters are limited. 'Our previous algorithm' is the optimization based on heuristic resource allocation as described in Chapter 4. The computation error is analyzed in statistically based on ε parameter, and the fractional wordlength is estimated from acceptable error.

5.7.2 Application to Color Space Conversion

The optimization result of color space conversion is shown in Table 5.2. Note that the correctness of the manual optimization result are verified with exhaustive simulation.

The result shows that the positive and negative error model is more suitable than positive error model for error estimation. The optimization of the constant 0.5 is not

Table 5.2: Results of color space conversion.

Var name	Manual optimization	Previous algorithm	New algorithm
$0.1684(L_0)$	10 bit	12 bit	11 bit
$0.3316(L_1)$	10 bit	12 bit	11 bit
$0.5(L_2)$	2 bit	12 bit	8 bit
$\mathrm{tmp0}(L_3)$	3 bit	4 bit	3 bit
$\mathrm{tmp1}(L_4)$	3 bit	4 bit	3 bit
$\mathrm{tmp2}(L_5)$	3 bit	4 bit	3 bit
$\mathrm{tmp3}(L_6)$	3 bit	4 bit	3 bit
$Cr(L_7)$	3 bit	4 bit	3 bit
Total Bits	37 bit	56 bit	45 bit

sufficient compared with manual optimization. The constant 0.5 does not have error after the conversion to fixed-point number. It is a limitation of our algorithm because bit-length of variables are handled as a parameter.

5.7.3 Application to FIR Filter

FIR filter is often used for sound processing and demand for hardware implementation is high. The key of hardware implementation is fractional wordlength of coefficients. The specification of FIR filter is as follows:

- The filter has 11 taps.
- The inputs are 16 bit integer.
- The output is real value between 0 to 2^{16} and acceptable error is ± 1.0

The optimization result of FIR filter is shown in Table 5.3. The result of our new algorithm is better than manual optimization.

Table 5.3: Results of FIR filter.

	Coefficients	Total bits
Manual optimization	19 bit	269 bit
Previous algorithm	21 bit	291 bit
New algorithm	20 bit	268 bit

5.8 Concluding Remarks

We have proposed an optimization algorithm to estimate the fractional wordlength of variables in the high-level synthesis. The error of outputs are estimated with a new error model called 'positive and negative error model'.

Minimization problem of the fractional wordlength is formulated as a non-linear problem, and solved using a SQP method. We also present constrains for sharing of operation units. Our algorithm does not require the simulation on huge data, and is fast and guarantee the worst case accuracy.

We have implemented the optimization algorithm and that is applied to two sample programs. From the results, we found that the method is useful for the estimation of the fractional wordlength.

This method can be used for prototype development of a hardware module or an embedded system. Designers can estimate the bit length of data paths or the amount of registers easily.

Refinements of the optimization algorithm is needed for practical circuit designs. As a cost function of a non-linear programming, the function representing actual hardware costs should be introduced instead of the sum of fractional wordlength. Re-processing phase for getting integer result is also important topic, and some heuristics may be useful to find better solutions.

Chapter 6

Conclusion

6.1 Summary of Thesis

Design productivity gap is a critical problem in current LSI design, and high-level synthesis is usually used to reduce design time. This thesis provides automatic bit-length optimization methods for high-level synthesis with floating-point to fixed-point conversion. Two optimization methods, 'Heuristic resource allocation method' and 'Non-linear programming based method', have been discussed in the thesis.

The first method is based on heuristic resource allocation. The method analyzes computation errors statistically, and propagates it from input variables to output ones. Then, the budget for acceptable error called 'accuracy limitation' is rationed to variables according to its worst case error. We have implemented the optimization algorithm and applied it to six sample programs. The results show that our algorithm is effective for obtaining an upper bound of the necessary wordlength. The algorithm does not use the simulation with huge input data, so the execution is very fast and applicable to large programs. The result of our algorithm might be an over-estimation, but is very useful for prototyping designs and for the initial setting for further optimization using the simulation-based methods.

In the second method, non-linear programming technique is newly introduced. The technique improves the balancing between computation accuracy and total bit-length of variables and operation units. Minimization problem of the fractional wordlength is

formulated as a non-linear problem, and solved using a SQP method. We also present constrains for sharing of operation units. Our algorithm does not require the simulation on huge data, and is fast and guarantee the worst case accuracy. We have implemented the optimization algorithm and that is applied to two sample programs. From the results, we found that the method is useful for the estimation of the fractional wordlength. The result also shows that the non-linear programming based method finds better solutions than that of the heuristic resource allocation based method.

6.2 Future Works

We have shown a basic idea for the estimation of the fractional wordlength. Refinement of the algorithm would be needed.

One of the important problems is the unification of scheduling, unit sharing and bit-length optimization. They are closely related each other and have great effects on the performance of synthesized circuits. The constraints such as maximum area and latency should be included too.

The evaluation of the optimization result is another important topic. The optimization result of our algorithm and true optimum result should be compared, and evaluated how the algorithm achieve bit-length optimization.

In the non-linear programming based method, the result in real value is re-processed into an integer result. This task has some effect on final result, so refinements are needed.

Developing an analysis method for programs including elementary functions such as square root, cos, exp is challenging issue.

Acknowledgement

I would like to appreciate Professor Shinji Kimura, my supervisor, for his continuous encouragement and support. He tells me the interest of hardware design and design automation technologies. Working with him has been and will continue to be a source of honor and pride for me.

I wish to thank the member of this thesis reviewing committee Professor Takeshi Yoshimura and Professor Tsutomu Yoshihara for their careful review of this thesis.

I am heartily grateful to Assistant Professor Takashi Horiyama of Graduate School of Kyoto University and Assistant Professor Masaki Nakanishi of Nara Institute of Science and Technology for their constant guidance, encouragement and suggestions throughout this work.

I wish to thank funds from the Japanese Ministry of ECSST via Kitakushu and Fukuoka knowledge-based cluster projects and funds from Advanced Research Institute for Science and Engineering for their strong financial supports.

I have greatly appliciated all my friends, colleagues and faculties at Kimura Lab. of Waseda University, Watanabe Lab. of Nara Institute of Science and Technology and Ishii Lab. of Nagoya Institute of Technology for their helping and owe them a great deal for comfortable and pleasant school life at the laboratory.

Finally, I wish to thank my family for their mental and various supports.

References

[1] Wojciech P. Maly. "SIA Road Map and DESIGN&TEST". *UC Berkeley*, April 1997.

[2] "Synopsys - Design Compiler". http://www.synopsys.com/.

[3] "Cadence Design Systems - BuildGates". http://www.cadence.com/.

[4] Daniel D. Gajski, Nikil D. Dutt, Allen C.-H. Wu, and Steve Y.-L. Lin. *"High-Level Synthes : introduction to chip and system designs"*. Kluwer Academic Publishers, 1992.

[5] Gaurav Singh, Sumit Gupta, Sandeep Shukla, and Rajesh Gupta. *"Parallelizing High-Level Synthesis: A Code Transformational Approach to High-Level Synthesis"*. The CRC Handbook of EDA for IC Design, 2005.

[6] P. Marwedel. "A new synthesis for the MIMOLA software system". In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 271–277, June 1986.

[7] P. G. Paulin and J. P. Knight. "Force-Directed Scheduling in Automated Data Path Synthesis". In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 195–202, July 1987.

[8] G. De Micheli, D. C. Ku, F. Mailhot, and T. Truong. "The Olympus Synthesis System". *IEEE Design & Test*, 7(5):37–53, October 1990.

[9] S. Gupta, N.D. Dutt, R.K. Gupta, and A. Nicolau. "SPARK: A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations". In *Proceedings of the 16th International Conference on VLSI Design*, pages 461–466, January 2003.

[10] Kazutoshi Wakabayashi and Takumi Okamoto. "C-Based SoC Design Flow and EDA Tools: An ASIC and System Vendor Perspective". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19:1507–1522, December 2000.

[11] "FORTE - Cynthesizer". http://www.forteds.com/.

[12] "Mentor Graphics - Catapult C Synthesis". http://www.mentor.com/.

[13] Holger Keding, Markus Willems, Martin Coors, and Heinrich Meyr. "FRIDGE: A Fixed-Point Design and Simulation Environment". In *Proceedings of the conference on Design, Automation and Test in Europe*, pages 429–435, February 1998.

[14] Paul Lieverse, Todor Stefanov, Pieter van der Wolf, and Ed Deprettere. "System level design with spade: an M-JPEG case study". In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design*, pages 31–38, November 2001.

[15] "SystemC". http://www.systemc.org/.

[16] Kazutoshi Wakabayashi. *"Cyber: High Level Synthesis System from Software into ASIC"*. Kluwer Academic Publishers, 1991.

[17] M. C. McFarland. "The Value Trace: A data base for automated digital design". Technical report, Carnegie-Mellon University, Design Research Center, 1978.

[18] R. Camposano and R. M. Tabet. "Design representation for the synthesis of behavioral VHDL models". In *Proceedings 9th International Synposium on Computer Hardware Description Languages and Their Applications*, pages 49–58, June 1989.

[19] A. Orailoglu and D.D. Gajski. "Flow graph representation". In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 503–509, June 1986.

[20] Reinaldo A. Bergamaschi. "Behavioral network graph unifying the domains of high-level and logic synthesis". In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 951–957, June 1999.

[21] Zebo Peng and Krzysztof Kuchcinski. "Automated Transformation of Algorithms into Register-Transfer Level Implementations". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13:150–166, February 1994.

[22] James Lyle Peterson. *"Petri Net Theory and the Modeling of Systems"*. Prentice Hall PTR, 1981.

[23] Zebo Peng. "Synthesis of VLSI systems with the CAMAD design aid". In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 278–284, June 1986.

[24] P. G. Paulin and J. P. Knight. "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–678, June 1989.

[25] H. De Man et al. "Cathedral-II: A silicon compiler for digital signal processing". *IEEE Design & Test*, 3(6):13–26, December 1986.

[26] Hwang T., Lee J., and Hsu Y. "A Formal Approach to the Scheduling Problem in High- Level Synthesis". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(4):464–475, April 1991.

[27] Nobuhiro Doi, Kazunari Sumiyoshi, and Naohiro Ishii. "Task Duplication and Insertion for Scheduling with Communication Costs". *International Journal of Computer and Information Science*, 3(1):73–83, March 2002.

[28] C.J. Tseng and D.P. Siewiorek. Automated synthesis of data paths in digital systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-5:379–395, July 1986.

[29] Fadi J. Kurdahi and Alice C. Parker. "REAL: A program for REgister ALlocation". In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 210–215, June 1987.

[30] N. Park and A. Parker. "Sehwa: A software package for synthesis of pipelines from behavioral specifications". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(3):356–370, March 1988.

[31] Alexandru Nicolau and Roni Potasman. "Incremental Tree Height Reduction For High Level Synthesis". In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 770–774, June 1991.

[32] M.Potkonjak and J.Rabaey. "Optimizing resource utilization using transformations". In *Proceedings of the 1991 IEEE/ACM International Conference on Computer-Aided Design*, pages 88–91, November 1991.

[33] Miodrag Potkonjak and Mani B. Srivastava. "Rephasing: A Transformation Technique for the Manipulation of Timing Constraints". In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 107–112, June 1995.

[34] Y. L. Lin T. F. Lee, Allen C.-H. Wu and D. D. Gajski. "A Transformation-based Method for Loop Folding". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):439–450, April 1994.

[35] K. Wakabayashi and T. Yoshimura. "A Resource Sharing and Control Synthesis Method for Conditional Branches". In *Proceedings of the 1989 IEEE/ACM International Conference on Computer-Aided Design*, pages 62–65, November 1989.

[36] Osamu Ogawa, Kazuyoshi Takagi, Yasufumi Itoh, Shinji Kimura, and Katsumasa Watanabe. "Hardware Synthesis from C Programs with Estimation of Bit Length of Variables". *IEICE Transaction*, E82-A(11):2338–2346, November 1999.

[37] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. "Bitwidth Analysis with Application to Silicon Compilation". In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 108–120, June 2000.

[38] F. Fang, Tsuhan Chen, and Rob A. Rutenbar. "Lightweight Floating-Point Arithmetic: Case Study of Inverse Discrete Cosine Transform". *EURASIP Journal on Signal Processing*, 2002(9):879–892, May 2002.

[39] Markus Willems, Volker Bursgens, Holger Keding, Thorsten Grutker, and Heinrich Meyr. "System Level Fixed-Point Design Based on an Interpolative Approach". In *Proceedings of the 34st ACM/IEEE Design Automation Conference*, pages 293–298, June 1997.

[40] Markus Willems, Volker Bursgens, and Heinrich Meyr. "FRIDGE: Floating-Point Programing of Fixed-Point Digital Signal Processors". In *Proceedings of the International Conference on Signal Processing Applications & Technology*, September 1997.

[41] Ki-Il Kum, Jiyang Kang, and Wonyong Sung. "AUTOSCALER For C: An Optimizing Floating-Point to Integer C Program Converter For Fixed-point Digital Signal Processors". *IEEE Transaction on Circuits and Systems II*, 47(9):840–848, September 2000.

[42] Seehyun Kim and Wonyong Sung. "Fixed-Point Error Analysis and Word Length Optimization of 8x8 IDCT". In *Proceedings of the IEEE Workshop VLSI Signal Processing*, pages 398–407, December 1996.

[43] Daniel Menard and Olivier Sentieys. "Automatic Evaluation of the Accuracy of Fixed-point Algorithms". In *Proceedings of the conference on Design, Automation and Test in Europe*, pages 529–535, March 2002.

[44] George A. Constantinides, Peter Y. K. Cheung, and Wayne Luk. "Wordlength Optimization for Linear Digital Signal Processing". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(10):1432–1442, October 2003.

[45] George A. Constantinides. "Perturbation Analysis for Word-length Optimization". In *Proceddings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 81–90, April 2003.

[46] Claire Fang Fang, Rob A. Rutenbar, Markus Puschel, and Tsuhan Chen. "Toward efficient static analysis of finite-precision effects in DSP applications via affine arithmetic modeling". In *Proceedings of the 40th ACM/IEEE Design Automation Conference*, pages 496–501, June 2003.

[47] "Matlab-Simlink". http://www.mathworks.com/.

[48] L. H. de Figueiredo and J. Stolfi. "Self-validated numerical methods and applications". *Brazilian Mathematics Colloquium monograph, IMPA*, July 2003.

[49] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[50] "The Stanford SUIF Compiler Group". http://suif.stanford.edu/.

[51] T. Ibaraki and M. Fukushima. *"FORTRAN 77 Optimization Programming"(in Japanese)*. Iwanami, 1991.

[52] Suhrid A. Wadekar and Alice C. Parker. "Accuracy Sensitive Word-Length Selection for Algorithm Optimization". In *Proceedings of the International Conference on Computer Design*, pages 54–61, October 1998.

[53] "ANTLR Parser Generator and Transrator". http://www.antlr.org/.

[54] "DONLP2". http://plato.la.asu.edu/donlp2.html.

List of Publications

論文

- Nobuhiro DOI, Takashi HORIYAMA, Masaki NAKANISHI, Shinji KIMURA and Katsumasa WATANABE, "Bit Length Optimization of Fractional Part on Floating to Fixed Point Conversion for High-Level Synthesis," IEICE Transactions on Fundamentals, pp.3184-3191, Vol. E86-A, No. 12, December 2003.
- Nobuhiro Doi, Kazunari Sumiyoshi and Naohiro Ishii, "Task Duplication and Insertion for Scheduling with Communication Costs," International Journal of Computer and Information Science, pp.73–83, Vol. 3, No. 1, March 2002.

国際会議

- Nobuhiro Doi, Takashi Horiyama, Masaki Nakanishi, Shinji Kimura, "An Optimization Method in Floating-point to Fixed-point Conversion using Positive and Negative Error Analysis and Sharing of Operations," In Proceedings of SASIMI 2004, pp.466-471, October 2004.
- Chengnan Jin, Nobuhiro Doi, Hatsukzu Tanaka, Shigeki Imai, Shinji Kimura, "Efficient Hardware Architecture of a New Simple Public-Key Cryptosystem for Real-Time Data Processing," In Proceedings of SASIMI 2004, pp.466-471, October 2004.
- Nobuhiro Doi, Takashi Horiyama, Masaki Nakanishi and Shinji Kimura, "Minimization of Fractional Wordlength on Fixed-Point Conversion for High-Level Synthesis," In Proceedings of ASP-DAC 2004, pp.80-85, January 2004.

- Nobuhiro Doi, Takashi Horiyama, Masaki Nakanishi, Shinji Kimura and Katsumasa Watanabe, “Bit Length Optimization of Fractional Parts on Floating to Fixed Point Conversion for High-Level Synthesis,” In Proceedings of SASIMI 2003, pp.129-136, April 2003.

- Nobuhiro Doi, Kazunari Sumiyoshi and Naohiro Ishii, “Estimation of Earliest Starting Time for Scheduling with Comunication Costs,” In Proceedings of SNPD’01, pp.399-406, August 2001.

講演

- 土井伸洋, 堀山貴史, 中西正樹, 木村晋二, 「ビット長に制約がある場合の実数演算の固定小数点演算化」, DA シンポジウム 2005 論文集, pp.49-54, 2005 年 7 月.

- 土井伸洋, 堀山貴史, 中西正樹, 木村晋二, 「非線形方程式と整数解の探索に基づく高位合成向けビット長最適化」, ETNET2005, 2005 年 3 月.

- 金成男, 土井伸洋, 田中初一, 今井繁規, 木村晋二,「動画像処理向け高速公開鍵暗号 LSI アーキテクチャ」, ETNET2005, 2005 年 3 月.

- 許哲武, 土井伸洋, 木村晋二, 「肌色認識に基づくユーザーインタフェースに関する研究」, 電気関係学会 九州支部大会 07-2P-26, 2004 年 9 月.

- 土井伸洋, 堀山貴史, 中西正樹, 木村晋二, 「負方向への誤差を考慮した高位合成向けビット長最適化手法」, 電子情報通信学会 2004 ソサエティ大会 AS-3-1, 2004 年 9 月.

- Chengnan Jin, Nobuhiro Doi, Shinji Kimura, 「A New Simple Public-Key Cryptosystem LSI for Real-Time Data Processing」, 電子情報通信学会 2004 ソサエティ大会 AS-3-3, 2004 年 9 月.

- 土井伸洋, 堀山貴史, 中西正樹, 木村晋二, 「浮動小数点演算での誤差の増減を考慮した変数ビット長の最適化」, DA シンポジウム 2004 論文集, pp.85-90, 2004 年 7 月.

- 土井伸洋, 堀山貴史, 中西正樹, 木村晋二, 「抽象解釈手法に基づく変数の相互関係解析とそのデータパス最適化への応用」, 信学技報 VLD2004-8, pp.7-12, 2004 年 5 月.
- Nobuhiro Doi, Takashi Horiyama, Masaki Nakanishi and Shinji Kimura, "Minimization of Fractional Wordlength on Fixed-Point Conversion for High-Level Synthesis," In Proceedings of Waseda University System LSI International Workshop, January 2004.
- 土井伸洋, 堀山貴史, 中西正樹, 木村晋二, 「丸目を考慮した浮動小数点数の固定小数点数への自動変換」, DA シンポジウム 2003 論文集, pp.209-214, 2003 年 7 月.
- 土井伸洋, 堀山貴史, 中西正樹, 木村晋二, 「C プログラムからの合成における浮動小数点演算のビット長最適化」, 第 6 回システム LSI ワークショップ論文集, pp.263-266, 2002 年 11 月.
- 土井伸洋, 堀山貴史, 中西正樹, 木村晋二, 「浮動小数点を含む C プログラムからのハードウェア生成におけるビット長最適化」, DA シンポジウム 2002 論文集, pp.119-124, 2002 年 7 月.