Accuracy Sensitive Word-Length Selection for Algorithm Optimization

Suhrid Ashok Wadekar
CENG 98-28

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213-740-4476)
August 1998

ACCURACY SENSITIVE WORD–LENGTH SELECTION FOR ALGORITHM OPTIMIZATION

by

Suhrid Ashok Wadekar

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Electrical Engineering)

August 1998

Dedication

This dissertation is dedicated to my mother Sou. Sanjeevanee Ashok Wadekar.

Acknowledgments

I could not have completed this research without the guidance, support, friendship, and love of many. I am very delighted to express my sincere gratitude. First, I would like to thank my advisor Prof. Alice Parker. From the very beginning, she trusted me and guided me as I explored my own ideas. Along the way, she trained me to think deeper and wider, and to investigate not only the solutions, but also the problems that push the technology forward. I am truly grateful to Dr. Parker for providing me this invaluable knowledge. Dr. Parker is also very kind and considerate in solving problems outside the research domain. It has been a privilege to be her advisee.

I would also like to thank Prof. Peter Beerel for his interest and assistance in my research. He always had time for me and on countless occasions he has entertained my questions about algorithms and design automation. I would like to thank the members of my qualifying examination and final defense committee, Prof. Sandeep Gupta, Prof. George Papavassilopoulos, Prof. Aristides Requicha, and Prof. Dennis McLeod for their comments, suggestions, and encouragement. I also wish to thank Prof. C.P. Ravikumar. Our discussions helped me understand my rough, fresh ideas better, and develop them into this research project.

My wife Swapna has been a constant source of love, encouragement, and inspiration during this work. She was by my side in all the problems, and assisted me in every single way while simultaneously completing her own doctoral research. I thank her very sincerely for her love and support. She is my very best friend! I also thank my parents Ashok and Sou. Sanjeevanee Wadekar. From my childhood they have taught me the importance of learning. They have sacrificed their desires so that I would get all the opportunities to pursue my studies. I am forever indebted to them. My parents–in–law Sudhir and Sou. Mugdha Gokhale have been very supportive, and patient during my studies, and I thank them for their love and trust in me. I thank my sister Subhagya for her best wishes, and

for constantly reminding me to have some fun while staying focused on my work. I also thank my brother–in–law Salil for his love and best wishes.

A number of friends and colleagues have been very helpful during this research. I would like to thank Dong–Hyun Heo, and Diogenes Silva for all their help, and also Yosef Gavriel Tirat–Gefen, Sami Habib, Rohini Montenegro, Mary Zittercob, and Bill Bates for their friendship and assistance. My friends Ivan Hom, Arani Sinha, Shivanand Bhajekar, and Dr. Tai with whom I worked as a teaching assistant are among many others who have made my years at USC pleasant.

This research is funded in part by the Advanced Research Projects Agency and monitored by the Federal Bureau of Investigation under the Grant J–FBI–94–161. The information reported here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred. It is also funded in part by the National Science Foundation under the Grant GER–9023979. I would like to thank these organizations for their support.

Contents

List Of Tables

List Of Figures

Abstract

In typical hardware implementations of an arithmetic–intensive algorithm, designers must determine the word lengths of resources such as adders, multipliers, and registers. This dissertation presents algorithmic level theory and optimization techniques to select distinct word lengths for each computation which meet the desired accuracy and minimize the design cost for the given performance constraints. The reduction in cost is possible by avoiding unnecessary bit–level computations that do not contribute significantly to the accuracy of the final results. *Thus we have introduced a new optimization variable, computation accuracy, into data–path synthesis.* Using these distinct word lengths during high–level synthesis, highly efficient designs in terms of area, power consumption, and cost can be produced. The high efficiency is achieved by simultaneously taking advantage of two relationships: one between the area of an individual resource and its word length, and the other a trade off between word lengths of operations in an algorithm and numerical accuracy of the final result. While some computations in the algorithm may require high precision, others may require low precision, and hence relatively fewer bits. Therefore, using resources of smaller word lengths to execute the low–precision operations, the design cost can be reduced. However, resources with small word lengths cannot directly execute operations requiring large word lengths. Therefore, the choice of word lengths affects resource sharing, and in turn the design cost. Thus a straightforward selection of the minimum word lengths that meet the accuracy requirement does not necessarily result in a low–cost design.

The theory we developed forms the basis for techniques to analyze the flow of computations and produce analytical models of computation accuracy, and implementation cost in terms of word lengths of algorithm variables and resources. Using these models, resource usage, and operation and resource word lengths are identified simultaneously while satisfying the desired accuracy and performance requirements. Our results show on an average, a 30% reduction in functional–resource area using distinct word lengths as opposed to use of a single optimized word length for the entire algorithm.

Chapter 1

Introduction

With new VLSI technologies it is easy to build circuits for applications such as secure mobile communication using spread spectrum techniques, robots for automobile navigation, and satellite guidance control. In such applications, a large number of computations are performed that require varying levels of accuracy. In traditional software implementations, such operations are performed either in single, double or extended double precision. However, in hardware implementations, word lengths of the operands and results, intermediate as well as final, need not be fixed throughout the hardware. They could vary over a vast range. By appropriately choosing the adequate word length for each computation, significant savings in the design cost can be achieved. In resource–dominated circuits cost minimization is intuitive because smaller functional operators can be used for operations requiring less precision. As a result, the total number of transistors in the circuit decreases and in effect, the size, power consumption and delay of the circuit reduce. In the case of low–cost circuits, where a small number of resources are shared by many operations, a resource with the largest required word length must be used to execute many operations even though some operations using the resource do not require all of the bits available. Word length optimizations are still important because the cost of resource sharing (registers, multiplexers and the interconnect) depends on the number of significant bits in each computation. Furthermore, advanced control schemes may be implemented to prevent the unused bits from switching in a large resource. This implies savings in power consumption and delay. Accurate modeling of the error introduced in a computation by a specific choice of word length and its impact on the cost of the implementation are two important aspects of the word length optimization problem.

1.1 Effect of word lengths on computation accuracy and design cost

The numerical accuracy of the final results in a sequence of computations is less sensitive to the precision of some computations compared to the others. Only the computations having a significant impact on the accuracy and reliability of the final results need be implemented using high–precision arithmetic while the other operations can be implemented with relatively less precision [41]. In a finite–precision representation the accuracy of a computation is proportional to the word lengths of its operands. Consequently, resources with larger word lengths may be necessary for some computations, but those with fewer bits may be adequate for computations that have a smaller impact on the accuracy of the final results. The size of a resource is also proportional to its word length. Resources of specific and distinct word lengths may be implemented easily using *module generators*. Thus the use of smaller resources is likely to reduce the design cost. However, it also restricts resource sharing between operations because operations requiring large word lengths cannot be implemented by resources of smaller word lengths without the use of additional hardware and control.

The major decisions in data–path synthesis are often related to the usage of functional resources. Optimization techniques are commonly used during data–path synthesis to maximize the resource sharing and to improve the quality of the circuits in terms of cost as well as performance [11]. Standard arithmetic operations in the circuit specification are typically implemented using resources of a common predetermined word length so as to meet the desired level of accuracy in the final results. Since the word lengths of operations and resources are not treated as variables, the optimization techniques used in data–path synthesis cannot take advantage of the variation in the individual resource area with word length. *The principal objectives of selecting resources with different word lengths are to guarantee a certain accuracy in the final results, to ensure that resource sharing is not adversely affected, and to reduce redundant bit–level computations – thereby minimizing the cost of the design in terms of area, delay, energy and power.*

We examine the impact of word–length optimizations at the chip and system levels. Our word–length selection technique operates on the algorithm and is independent of the optimization techniques used during data–path synthesis. Therefore, design cost can be

minimized using the word lengths selected by our technique in conjunction with any automated or manual data path synthesis procedure.

1.1.1 Sources of computation error in hardware

Typically truncation/quantization errors are introduced in the primary inputs of a system due to finite word length representation. In application–specific hardware, implementations using fixed–point representation of numbers are usually preferred over those using floating–point representation because the former are faster and smaller [1,24,30,64]. For typical arithmetic operations, the word length of the result is greater than that of the inputs. For example, addition of two N bit numbers generates an $N+1$ bit sum. Multiplication of an M–bit and an N–bit bit number generates an $(M+N)$ bit product. When several interdependent operations exist in a computation, the word lengths of intermediate results could be significantly larger than those of the primary inputs, if all the bits are always preserved. However, the least–significant bits of the results are commonly truncated or rounded off, leading to additional error in the computation. Depending on the nature of the flow and types of operations involved in a computation, errors in the intermediate results could contribute significantly or negligibly to the error in the final results [2,18,36].

1.1.2 Word length optimization

In the past, the word–length optimization problem has been researched in the design of specific algorithms such as sine, cosine, and reciprocal computation [23,27,50,63]. Word–length optimizations are also implemented at the architectural level [16,24,52,53,57,59]. In the case of a hardware design for a specific algorithm, the numerical properties of the algorithm are analyzed and taken advantage of in order to optimize the data path. In contrast to these manual methods, we perform word–length optimization automatically for any given algorithm. The numerical robustness of the algorithm is implicitly contained in the error analysis. In research reported at the architecture level, optimizations related to resource usage have already been made and the focus of the reported methods is on minimizing the number of bits [16, 24, 52, 53, 57, 59]. The size and power consumption of a resource are proportional to its input/output word lengths. Hence, optimizations leading to an architecture are affected by word length optimizations. Moreover, the interconnect

cost and resource–sharing cost are also dependent on word lengths. The motivation behind our word–length optimization is to minimize the implementation cost, not merely the number of bits. Another significant difference between our method and the architecture–level methods is that in the latter case, error is observed through exhaustive simulation. We use an analytical model of the error function expressed in word lengths. By analyzing the contribution of error in all the intermediate results to the error in all the final results, we determine the maximum number of bits to be truncated from each intermediate result such that the worst case final error is less than the maximum permissible value. These truncations directly reduce the number of bit–level computations and the cost of the implementation in consequence. From a system–level perspective, eliminating only a few bits in inter-chip communication, or in the values stored in a look–up table might allow the designer to reduce the overall system cost significantly. *Thus, the optimal choice of input/output word lengths of functional resources that minimizes the implementation cost considering resource sharing is important.*

In large systems, thousands of computations are performed in tasks such as calculation of the Riccati gain for navigation control [34].[1] As stated earlier, some of these computations may require less precision compared to the others. Thus, using functional units with smaller word lengths for such operations will reduce the system cost. This problem is loosely similar to the module selection problem [11] where different modules of a function type compute the function using different methods. Depending on the method selected, each module has a distinct area–delay characteristic. Relatively faster and larger modules are used in the critical path and relatively slower but smaller ones are used in non–critical paths. An important distinction between the module selection and word–length selection problem is that in the former, all candidate modules to be used in the implementation and their word lengths are known *a priori*. Even if the most generalized case where modules of distinct word lengths are considered,[2] the word length of each module is known. The word–length optimization problem is to determine the optimum number of resource word lengths of each type, and the corresponding word lengths. In other words, the optimum

[1] In control systems, 15 or more variables are typically observed for parameter estimation. The size of the corresponding Hamiltonian matrix is 30 or more. It is well–known that many matrix operations involve $O(N^2)$ or $O(N^3)$ computations where N is the size of the matrix.

[2] The module selection problem reported in the literature [8, 11, 21] does not consider word length as a variable. However in the most general formulation, different modules of a function type may have distinct word lengths as well as different methods of function computation.

module set to be used in the implementation itself is to be determined. Also, in the module selection problem any module of the same function type is able to execute any operation of that type. For example, all additions could be performed by ripple carry as well as carry look–ahead adders. When operators of different word lengths are used this is not true. A 16–bit adder could be used for 10–bit addition but the converse is not true (at least not without use of additional control and hardware). Thus resource sharing is constrained. Moreover, the size of many operators increases superlinearly or even quadratically with the number of bits. The carry look–ahead adder, Booth multiplier, and arithmetic shifter are a few examples of this observation. Using many small resources of a function type and a fewer large ones implies that the high–precision operations of the same type in a task are executed more sequentially compared to those requiring less precision. If the timing constraints are satisfied, such assignment could lead to implementations with reduced cost due to the nonlinear relation of operator size to word length. The cost of resource sharing i.e., the cost of registers, multiplexers, control, and wiring, also depends on the word lengths of the associated values and is significant in large designs. Thus, the word–length optimization problem is strongly correlated with the traditional resource utilization optimization problem. In the next section, we illustrate the importance of considering the word–length optimization problem at the algorithmic level, prior to architecture synthesis.

1.2 Motivation for algorithm–level word–length optimization

Without the knowledge of word lengths before synthesis, all resources of a certain type are assumed to have a common pre–determined word length. Therefore, high level design decisions cannot take advantage of variation in the area of individual resources according to their word lengths. This is illustrated in the following small example. Figure 1.1 shows two scheduled graphs of an algorithm that performs four multiplications. The corresponding architectures are also shown. Both architectures require two multipliers, and without the knowledge of word lengths of resources *prior to synthesis* both multipliers are assumed to have the same area. Thus design (b) cannot be claimed to be better than design (a). Assume that the word–length optimization and numerical analysis before synthesis has revealed that the optimum word length of the first two multiplications (shown as

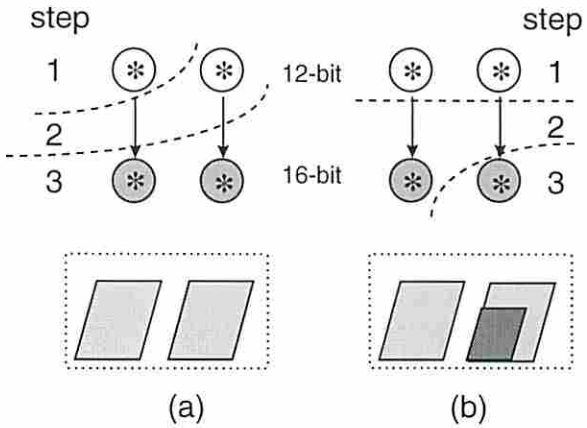


Figure 1.1: Word length optimization after architecture synthesis

unshaded circles in Figure 1.1) is 12 bits. Now if both 12 and 16–bit multipliers are considered, one 12–bit multiplier can be used in place of a 16–bit multiplier. This is possible if and only if the two 16–bit multiplications are assigned to the same multiplier using the schedule in design (b). Once again, the high–level synthesis decision that determines the operation to resource assignment (*binding*) cannot make the optimization illustrated above without the knowledge of word lengths. From this exercise we observe that, if the two 16–bit multiplications are assigned to different multipliers, either by selecting schedule (a), or by choosing the alternate binding in design (b), word length optimization at the architecture level cannot reduce the word length of any multiplier. Considering word length selection before synthesis however, it is possible to reach a low–cost design using one 16–bit and one 12–bit multiplier. This reinforces our claim that knowledge of resource word lengths prior to data–path synthesis, and their use in high–level synthesis significantly increases the possibility of reaching a low–cost design.

The above example illustrates another important aspect of optimal word–length selection. Consider the case where the computation shown in Figure 1.1 is to be completed in 5 major cycles. In this case, it is sufficient to use only one 16–bit multiplier. However, the numerical analysis may reveal that the desired accuracy is also achieved if all four multiplications are performed using 15 bits. Then, using one 15–bit multiplier is a better choice. This demonstrates that the optimum operation word lengths required to achieve a specified accuracy cannot be determined independently of the optimum resource word lengths and resource sharing. In the next section, we introduce the optimal word–length selection

problem at the algorithmic level, and present an overview of the optimization procedure that makes it feasible to select operation and resource word lengths prior to synthesis.

1.3 Overview of word–length selection at the algorithmic level

Our overall objective is to minimize the cost of hardware implementation of an algorithm while keeping the computation error below its maximum permissible value and meeting performance constraints. There are two steps required to achieve this goal. The first step is to model the error in the final result of an algorithm in terms of word lengths of the variables in the algorithm. We show that the error model is a nonlinear function. The second step is to minimize the implementation cost function by selecting resource word lengths. The error model is used to verify whether the selected word lengths satisfy the accuracy requirement. The feasible design space of the hardware implementation of large and small tasks is vast and discrete under the given area, timing, energy, power, and accuracy constraints. For a given algorithm several implementations are feasible ranging from the fastest to the slowest. The resource requirement for each implementation is different. If resources of different word lengths are to be used, the designer must answer two questions:

1. How many distinct resource word lengths denoted ψ^t should be used by each function type t in the algorithm?[3] i.e. what is the word–length set size?

2. What word lengths will minimize the implementation cost for a desired level of accuracy? i.e. what are the values of the members of the word–length set?

The answer to the second question obviously depends on the answer to the first one. However, the answer to the first question is not independent of the word lengths to be used. This is because the number of distinct resource word lengths directly depends on the number of resources used in the implementation. The number of resources is selected to minimize the implementation cost, which depends on the resource word lengths. The two questions above jointly represent the word–length optimization problem at the algorithm level. Figure 1.2 gives an overview of our solution procedure. The first steps in solving these problems are

[3]There could be a different number for each function type t in the algorithm.

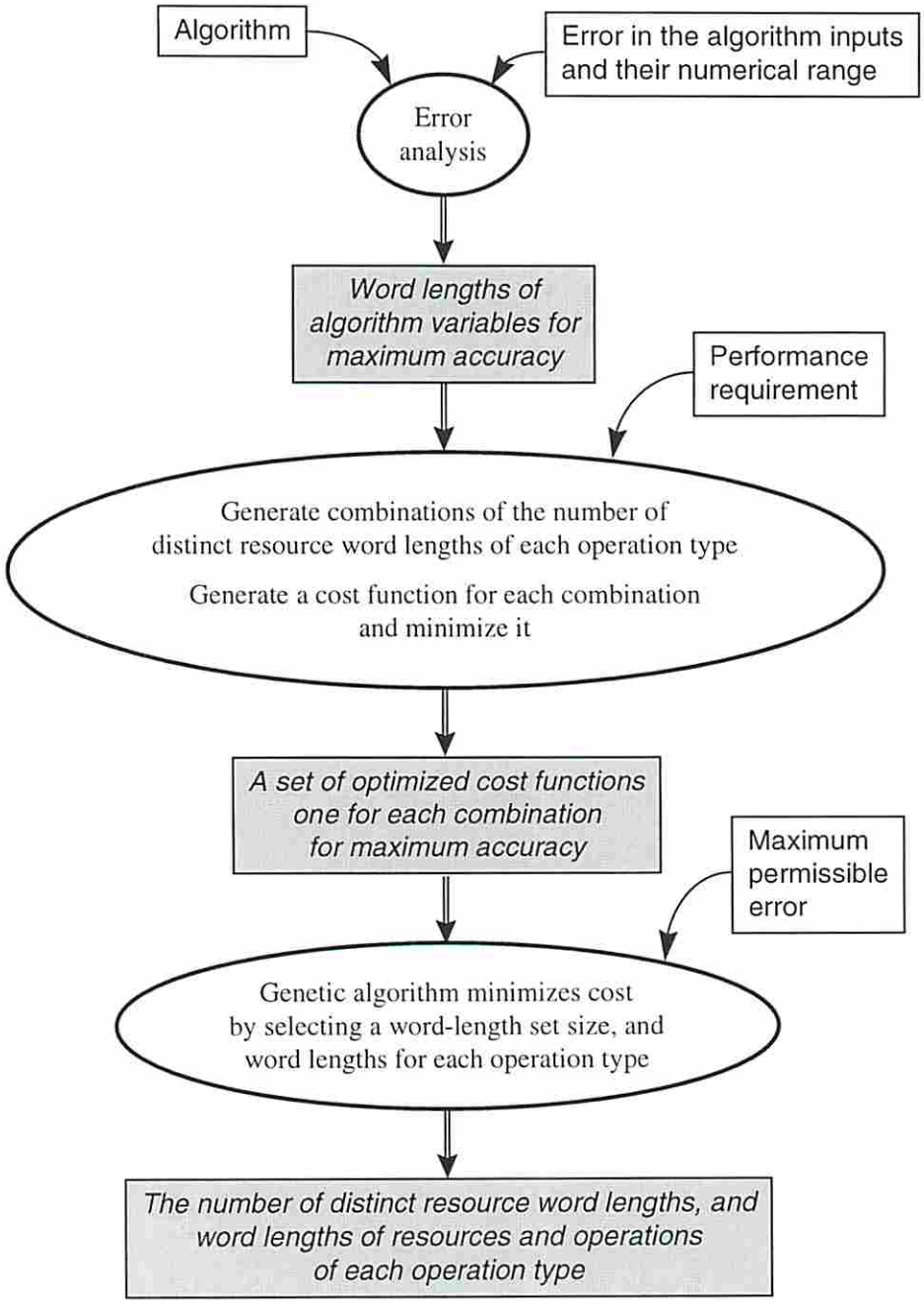


Figure 1.2: Major steps in word–length selection at the algorithmic level

(i) What is the range of ψ^t, the number of different resource word lengths selected to be used by each function type?

(ii) For each vector $\vec{\psi}$ formed by a unique combination of values of ψ^t (within the above range) for each function type, what is the optimized cost function for the implementation?

Figure 1.2 shows that the error analysis is used to determine the smallest word lengths for each operation that yield the maximum achievable accuracy in the final result of the algorithm, for the given error environment [61]. The error analysis also determines the range of ψ^t for each function type t. As shown in the overview, all unique combinations of $\vec{\psi}$ are explored. A cost function in terms of the resource requirement, and resource word lengths[4] is formed corresponding to each $\vec{\psi}$. Initially the resource word lengths determined from the error analysis that yield the maximum achievable accuracy are selected. The resource requirement is determined according to the performance requirement, and by taking into consideration the use of ψ^t different word lengths for each type t. Then the cost function is optimized by minimizing the use of larger word length resources in the implementation of operations that do not require high precision. Thus only a few large word length resources become necessary, which leads to cost reduction. This important optimization step is defined as *clustering*. Now we have a set of optimized cost functions, each function corresponding to a unique $\vec{\psi}$, that guarantees the maximum accuracy for the given error environment. We also have an error constraint equation expressed in terms of word lengths. If the required accuracy is smaller than the maximum accuracy, each cost function can be further minimized by manipulating the word lengths, while satisfying the error constraint. In this thesis we show that this is a nonlinear optimization problem. As seen in Figure 1.2, a genetic algorithm is used to obtain a set of word lengths for each $\vec{\psi}$ such that the corresponding cost function is minimized. From this set of minimized cost functions we select the one that yields the minimum cost by predicting the cost of a design obtained with the associated word lengths. The corresponding $\vec{\psi}$ answers the first question posed in the word–length optimization problem, and the corresponding word length set resulting in the minimum–cost implementation is the answer to the second question posed in that problem.

[4] As stated earlier, the area (cost) of an individual resource is a function of its word length.

This process can be iterated for all feasible execution delays and pipeline initiation rates. Thus optimized word lengths can be obtained for all the predicted implementations. Figure 1.3 emphasizes the role of behavioral–level word–length optimizations in the overall design process. A synthesis tool that incorporates resources of different word lengths

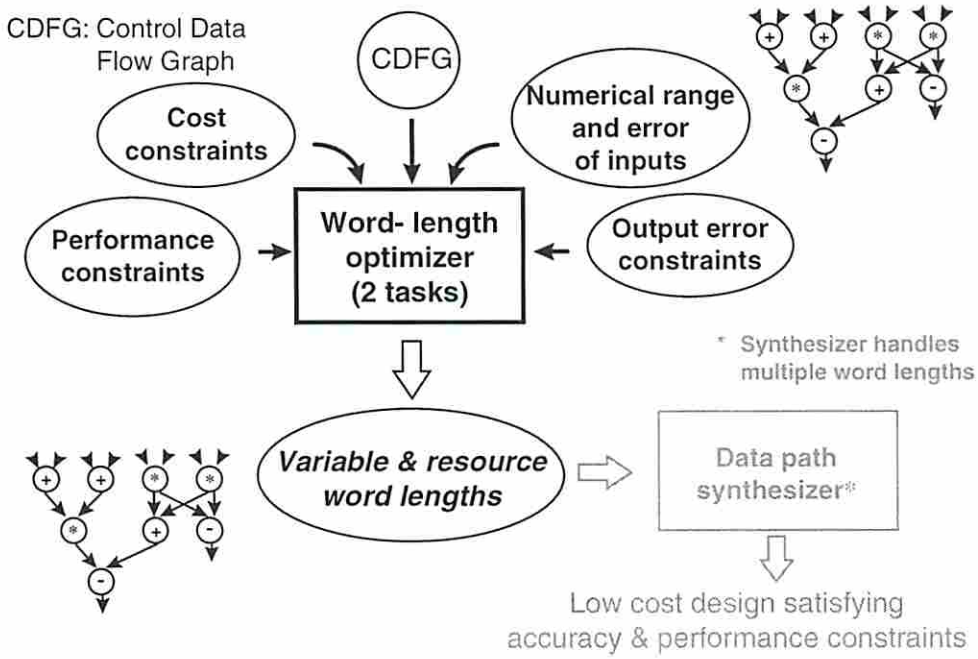


Figure 1.3: Behavioral–level word length optimizations and the overall design process

can explore the design space and make high–level decisions regarding the number of resources and their usage such that the implementation is likely to meet the specified performance and accuracy constraints with minimized cost.

1.4 Organization

In the next chapter we present a brief overview of previous work in the area of selecting minimum word lengths. A comprehensive problem formulation is given in Chapter 3. A mathematical model to represent the worst–case computation error in simple flow graphs[5] and a tool developed to analyze the error are described in Chapter 4. Then, the two–step cost optimization through word–length selection is given in Chapters 5 and 6. In Chapter 5,

[5] A DFG with no unbounded data–dependent loops and conditional branches is assumed.

we describe clustering and the derivation of the optimized cost function for maximum accuracy. As shown in Section 1.2, for maximum reduction in the design cost it is important to consider word–length optimization at the behavioral level. Since the cost function requires the number of resources of different word lengths, we also present a new behavioral prediction technique involving multiple word length use in this chapter. A genetic algorithm for selecting the optimized word–length set that minimizes the overall implementation cost, while satisfying the accuracy and performance requirements, is described in Chapter 6. The word–length set comprises the number of distinct resource word lengths of each operation type, and the word lengths of operations and resources. The results of reduction in resource area using resources of multiple word lengths optimized for the desired accuracy are given in Chapter 7. Conclusions and contributions of our research, and future research directions based on multiple word length use are given in Chapter 8.

Chapter 2

Related work

The effect of word lengths on the accuracy of computation and the cost of design has been studied for various reasons in the past. These efforts can be mainly classified as (a) work related to error analysis of application–specific hardware, (b) experience with design for specific arithmetic computation, and (c) architecture level minimization of the number of bits in the design. We present an overview of this work, and comment on its application to our research in this chapter.

2.1 Related work in analysis of computation error

In the work related to error analysis of application–specific hardware, the emphasis was to quantify the error introduced in the final results output by a digital system due to finite–precision arithmetic, in terms of input error. Largely, the effects of truncation and round off errors were studied. This work mainly considered synthesized hardware implementations of common signal processing applications (such as digital filters [5, 65] and image coding [33]), and digital communication applications (such as Adaptive differential PCM [53] and Spread spectrum receivers [9]). The input error was typically assumed to be additive white Gaussian noise and was specified using mean and variance. Then, mean and standard deviation of the output error were derived using statistical analysis. The techniques described here are less useful during circuit design because it starts with the knowledge of the circuit. It can be used to estimate the circuit behavior and quality of outputs when the environment in which it is operated is specified using statistical parameters.

2.2 Design for specific arithmetic computation

From the circuit design perspective, efforts were initially made to determine the minimum hardware required to implement a specific computation. Wadekar *et al.* implemented high speed chips to compute nonlinear functions such as reciprocal, square–root and trigonometric functions for IEEE single and double precision standards [23]. This implementation produced error in the least significant bit and was improved by Schulte and Schwartzlander [50]. Similar implementation of exponential functions was reported by Kantabutra [27]. In these designs, several implementation algorithms were explored and their numerical properties were analyzed. The most robust algorithm with acceptable speed was selected. Then, the word lengths of resources such as memory and arithmetic operators were determined using numerical analysis. These algorithms had only a few operations. The corresponding designs were also small and used only 4 or 5 functional resources; therefore, the design size was estimated quickly. Also, the numerical analysis was performed manually. The analytical model of computation error in our research was motivated by the effort reported previously [23,27,50]. However, we consider the general class of arithmetic–intensive algorithms involving hundreds or thousands of computations, and different performance and accuracy requirements.

2.3 Architecture level word length minimization

From the design automation perspective, methods were developed to minimize the number of bits in the design. Here again, the minimization starts with a known architecture. Techniques to find the smallest word lengths for a given implementation of discrete cosine transform (DCT) were reported by Uramato *et al.* [59]. Similar methods for discrete wavelet transform (DWT), and block filters were reported by Grzeszczak *et al.* [16], and Jang and Kim [24], respectively. All the techniques listed above employ exhaustive simulation of the architecture where error in the final result is computed, and verified against constraints in each simulation step. Sung and Kum propose a simulation–based word length optimization technique for signal processing systems at the architecture level [57].

These architecture–level techniques have three limitations. First, the objective of optimization is to minimize the number of bits. In our research we minimize the overall implementation cost which is related to, but not the same as the number of bits. Second, error in

the final results is observed through simulations. In large algorithms having several inputs, exhaustive simulations are impractical. Hence, accuracy of results cannot be guaranteed. We perform numerical analysis of the algorithm and model the worst–case error as a function of word lengths. The numerical analysis comprises range analysis and error analysis. For example, if two M–bit numbers A and B are multiplied, a $2M$–bit product C is generated. If the magnitude of A is always less than 1, then the magnitude of C is always less than B. From this range analysis we conclude that the number of integer bits required in C is no more than that in B. Assuming L bits must be truncated from the product, we first omit the unnecessary integer bits. This truncation does not change the value of C and no error is introduced. However, if more bits must be truncated after omitting all unnecessary integer bits, the least significant bits of C are truncated. The error introduced is a function of M and L, the word lengths of the operands of multiplication, and the number of bits truncated in the product. Naturally, if C is used in other computations the error in C propagates. The numerical analysis described in Chapters 3 and 4 allows us to make the best use of available word lengths, and to express the error analytically in terms of word lengths of algorithm variables. For a given choice of word lengths, the worst–case error is then readily computed using the analytical expression. Thus, the accuracy of results can always be guaranteed using our analytical technique. Finally, methods described in the literature start with an architecture obtained by ignoring the effect of word length on resource size, which in turn affects high–level decisions already taken while determining the architecture. We consider the word length optimization problem prior to architectural synthesis, where cost minimization is achieved by incorporating multiple word length use.

Chapter 3

Problem formulation

Given an algorithm, the set of operations to be performed and the connectivity of those operations are known. Each operation has one or more inputs and outputs. These are the variables (or constants) in the algorithm. A word length is associated with each primary input/output and internal variable. The desired accuracy of the computation is specified in terms of the tolerable error. The implementation of the algorithm is a set of resources[1] and interconnects. Input/output word lengths are also associated with each resource. *The word length optimization problem is to select the word lengths of variables in the algorithm and resources in the implementation such that the implementation cost (total area) is minimized while preserving the desired level of accuracy, and the desired performance.* Although the set of operations to be performed is known, the resource set and the word–length set are open sets. The number of resources, the size of the word–length set (i.e., the number of different word lengths to be used) and the elements of the set (i.e., the word lengths) are all unknowns. Note, that the number of required resources depends on the desired performance. Since resources of different word lengths are used, operations requiring larger word lengths cannot be implemented by resources of smaller word lengths. In effect, the number of resources needed also depends on the word length set. The cost of the implementation comprises resource and interconnect costs. These costs are proportional to the associated word lengths. Finally, the numerical accuracy of the result of the algorithm is also proportional to the number of bits preserved in each computation hence,

[1] In the problem formulation, the cost of functional resources and related registers and multiplexers is taken into account. For simplicity however, we primarily considered the functional resources in the development and implementation of the optimization techniques described in Chapters 4, 5, and 6. In general, word–length selection must also be performed for related registers and multiplexers, whose cost must be taken into account, as described in this chapter.

the word lengths of all variables. *The word length optimization problem is then a problem of determining (a) the optimum size of the word length set, (b) the optimum values of its elements, i.e. the word lengths of resources and variables, and (c) the optimum number of resources of each word length.* We formulate this problem by expressing the computation error constraint and the implementation cost as functions of word lengths. Then, appropriate optimization techniques can be used to select the best word lengths.

In this chapter we introduce an abstract model of the joint assignment of operations, word lengths and resources. The model is based on the final architecture–level design and not the synthesis methods, hence it is valid for any design style. Our comprehensive cost model includes the functional resource cost, register and multiplexer cost, and interconnect cost. The resource cost depends on resource word lengths while other costs depend on word lengths assigned to the variables in the algorithm. Therefore, the comprehensive cost model is developed using the model of the joint assignment of operations, word lengths and resources described above. The control cost is dominated by the number of resources and the performance, and is only weakly correlated with the word lengths. In Section 3.1 we develop an analytical model of error in the final result. The comprehensive cost model is described in Section 3.2, and the word length optimization problem is formulated in Section 3.3. The problem complexity is discussed in Section 3.4. An overview of our approach is presented in Section 3.5.

3.1 Analytical error model

In hardware or software implementations of an algorithm, there are three main sources of computation error: namely (i) finite–precision representation of external as well as internal operands, (ii) practical limitations of function evaluation, and (iii) error propagation. The error introduced due to a finite–precision representation is commonly known as *truncation* or *round off* error. Finite precision implies that the number of bits (digits) used is limited. The original number may be represented in infinite or finite precision. In the former case, error is always introduced due to finite–precision representation, and is inversely proportional to the number of bits preserved, i.e. the word length of the operand. In the latter case, error is introduced if the original word length is greater than that of the new representation. Second, some computations such as evaluation of e^x using the *Maclaurin*

series $\sum_{n=0}^{\infty} \frac{x^n}{n!}$, require an infinite number of terms to be evaluated to get the correct result. Since this is impractical in an implementation, a finite number of terms are evaluated and an error is allowed which is the second source of error. Third, if the input operands of an operation are not exact, the result computed also contains an error. This is the error propagated from the preceeding operations that generated the input operands with error to the present operation, and is the third source of error. More error may be introduced into the present result due to truncation of some of its least–significant bits.

In this section we model error in the final result of an algorithm as a function of word lengths of its variables. Given the specific choice of word lengths, we are then able to determine the level of accuracy in the final result. In the following analysis, a fixed–point representation is assumed as it is more common in application–specific hardware implementations compared to a floating–point representation [30].

3.1.1 Model of truncation error

We begin by defining two types of errors in numbers namely, *under–approximation error*, and *over–approximation error*.

Definition 3.1.1 Let $\tilde{v}$ be the actual value of a variable V. Let v be the represented value of V. If $v < \tilde{v}$, the error in representation is called *under–approximation error*. If $v > \tilde{v}$, the error is called *over–approximation error*.

$\tilde{v}$ and v are related as

$$\tilde{v} = \begin{cases} v + \delta^+ & \delta^+ \geq 0 \quad \text{is the under–approximation error} \\ v + \delta^- & \delta^- < 0 \quad \text{is the over–approximation error} \end{cases} \tag{3.1}$$

Definition 3.1.2 The error introduced by w–bit representation of a binary (unsigned or 2's complement) number represented in $\widetilde{\mathrm{w}}$–bits, by omitting $(\widetilde{\mathrm{w}} - \mathrm{w})$ least significant bits, where $\mathrm{w} < \widetilde{\mathrm{w}}$, is defined as *truncation error*.

Lemma 3.1.1 *Truncation error is always a positive number that must be added to the value of a variable to nullify the effect of truncation* (except when all the bits in a number are truncated).

Proof Let $\tilde{x}$ be represented in $\widetilde{\mathrm{w}}$ bits of which $\tilde{k}$ are the fractional bits. Let x be represented in w bits of which k are the fractional bits. Then,

$$k = \tilde{k} - (\widetilde{\mathrm{w}} - \mathrm{w}) \tag{3.2}$$

From Definition 3.1.2, the worst case truncation error in x is given by

$$\begin{aligned} \delta_x &= 2^{-(k+1)} + 2^{-(k+2)} + \cdots + 2^{-\tilde{k}} \\ &> 0 \end{aligned} \tag{3.3}$$

In the best case, $\delta_x = 0$ where the coefficients of all the weights beyond 2^{-k} are 0. Thus, the truncation error is always a positive number. Moreover, the value of $\tilde{x}$ is given by the summation

$$\begin{aligned} -a_{\widetilde{\mathrm{w}}-\tilde{k}-1}\, 2^{(\widetilde{\mathrm{w}}-\tilde{k}-1)} + \sum_{i=\widetilde{\mathrm{w}}-\tilde{k}-2}^{-\tilde{k}} a_i\, 2^i &= \Big[-a_{\widetilde{\mathrm{w}}-\tilde{k}-1}\, 2^{(\widetilde{\mathrm{w}}-\tilde{k}-1)} + \sum_{i=\widetilde{\mathrm{w}}-\tilde{k}-2}^{-k} a_i\, 2^i \Big] \\ &\quad + \sum_{j=-(k+1)}^{-\tilde{k}} a_j\, 2^j \\ &= x + \delta_x \end{aligned} \tag{3.4}$$

where a_i are the coefficients in the 2's complement representation of $\tilde{x}$. From Equation (3.4) it is observed that the truncation error is added to the truncated result to obtain the original value before truncation. □

This result is also valid for unsigned binary numbers, where the weight of the most significant bit is also positive. For numbers represented in 2's complement notation, the case where all bits are omitted may invalidate Lemma 3.1.1; however, this extreme case is of no practical use.

Lemma 3.1.2 *The worst case error introduced by truncating the $m = \tilde{l} - l$ least significant bits of a binary number is $(2^{-l} - 2^{-\tilde{l}})$, where the weight of the least significant bit of the number before truncation is $2^{-\tilde{l}}$, and 2^l after the truncation.*

Proof From Equation (3.3) we have

$$\begin{aligned}
\delta_x &\leq 2^{-(l+1)} + 2^{-(l+2)} + \cdots + 2^{-\tilde{l}} \\
&\leq 2^{-(l+1)} + 2^{-(l+2)} + \cdots + 2^{-(l+m)} \\
&\leq 2^{-(l+m)} \left[2^{m-1} + 2^{m-2} + \cdots + 1\right] \\
&\leq 2^{-(l+m)} \frac{2^m - 1}{2 - 1} \\
&\leq 2^{-l} \left[1 - 2^{-m}\right] \\
\delta_x &\leq 2^{-l} - 2^{-\tilde{l}}
\end{aligned}$$

□

If the number before truncation were represented using infinite precision, i.e. $\tilde{l} = \infty$, then $\delta_x \leq 2^{-l}$. Lemma 3.1.2 makes no assumptions regarding the values of l and $\tilde{l}$ except $\tilde{l} \geq l$. Hence, it is applicable to binary numbers with or without fractional bits, and is valid when fractional as well as integer bits are truncated. For example, if a number x belongs to the set of integers before truncation, the weight of its least significant bit is 1. The corresponding $\tilde{l} = 0$. If two least significant bits are truncated, $m = 2$ and $l = -2$. The worst case truncation error is indeed 3, as obtained from Lemma 3.1.2. In the strict sense, 2^{-l} is the weight of the least significant bit after truncation, which may be an integer as well as fractional bit. However, a few fractional bits are preserved in typical computations. Equation (3.2) gives us the relation between the number of fractional bits and m, i.e. the number of bits truncated. Combining this equation and the result from Lemma 3.1.2 we have

$$\delta_x \leq 2^{-k} \left[1 - 2^{-(\widetilde{\mathrm{w}} - \mathrm{w})}\right] \tag{3.5}$$

Now, the worst case truncation error in a number is expressed as sum of values that are powers of 2, where the exponents are linear functions of the total and fractional word lengths of the number.

3.1.2 Model of error propagation

As stated earlier, error in a partial result is propagated through the flow of computations when that result is an input of some succeeding computation. Without much loss in generality we consider an arbitrary arithmetic operation f, with two inputs and one output. Let the represented values of the two input variables U and V be u and v, respectively. Let their correct values be $\tilde{u}$ and $\tilde{v}$, and the respective errors be δ_u and δ_v such that

$$\tilde{u} = u + \delta_u \tag{3.6}$$

$$\tilde{v} = v + \delta_v \tag{3.7}$$

The errors δ_u and δ_v may be under– or over–approximation errors. If the computation of f does not introduce any error, then the error in the result due to error propagation is given by Taylor series expansion of a function of two variables as

$$\begin{aligned} f(\tilde{u},\tilde{v}) - f(u,v) = \frac{\partial f}{\partial u}(\delta_u) \; + \; \frac{\partial f}{\partial v}(\delta_v) \; + \; \frac{1}{2!}\left[\frac{\partial^2 f}{\partial u^2}(\delta_u)^2 \right. \\ \left. + \; 2\,\frac{\partial^2 f}{\partial u\,\partial v}(\delta_u\,\delta_v) \; + \; \frac{\partial^2 f}{\partial v^2}(\delta_v)^2\right] \; + \; \cdots \end{aligned} \tag{3.8}$$

In the equation above notice that the partial derivative terms evaluated at specific values U and V are constants. In order to account for the worst case error these values are selected from the respective numerical range of U and V such that the magnitude of the propagation error given by the equation above is maximum. If δ_u and δ_v are truncation errors from preceeding results, from Equation (3.5) we observe that the error in the present result is a weighted sum of powers of 2, where the exponents are linear functions of total and fractional word lengths of previous results. This is illustrated in the following simple example.

3.1.2.1 An example of error propagation

Let $f(u,v) = 2u + v^2$, and let the numerical range of U and V be $[1, 100]$ and $[-5, 5]$, respectively. Further, assume that both $\tilde{u}$ and $\tilde{v}$ are represented using 2 fractional bits, and

u and v are represented using only 1 fractional bit. Then, the worst case δ_u and δ_v are written using Lemma 3.1.2 as

$$\delta_u = \delta_v = 2^{-1} - 2^{-2} = 2^{-2} \tag{3.9}$$

From Equation (3.8) the propagation error in f is given by

$$f(\tilde{u}, \tilde{v}) - f(u, v) = 2\,\delta_u \; + \; 2\,v\,\delta_v \; + \; (\delta_v)^2 \tag{3.10}$$

The above error equation is a function of V. For its value $v \in [(\delta_v/2), 5]$ the error is zero or positive and represents the under–approximation propagation error. Its magnitude is maximum when $v = 5$. For $v \in [-5, (\delta_v/2))$ the error is negative, thus representing the over–approximation error. The worst–case over–approximation error is obtained when $v = -5$. Substituting from Equation (3.9), the worst case under– and over–approximation errors in f are

$$\delta_f^+ \;=\; 2 \cdot (2^{-2}) + 2 \cdot 5 \cdot (2^{-2}) + (2^{-2})^2 \tag{3.11}$$

$$\delta_f^- \;=\; 2 \cdot (2^{-2}) + 2 \cdot (-5) \cdot (2^{-2}) + (2^{-2})^2 \tag{3.12}$$

3.1.3 Computation error in algorithms

Let $\mathcal{V}$ be the total number of variables (and constants) in the algorithm $\mathcal{G}$. Let w_i and k_i be the total and fractional word lengths of the i–th variable. From Lemma 3.1.2 and Equation 3.8, we express the error in the final result of the algorithm as a function of word lengths of its variables as

$$error\,(\mathcal{G}) = \sum_{i=1}^{\mathcal{V}} \alpha_i\, 2^{a_i\, w_i} \; + \; \beta_i\, 2^{b_i\, k_i} \; + \; \gamma \tag{3.13}$$

where $\alpha_i, a_i, \beta_i, b_i$, and γ are constants. The desired level of accuracy is preserved for a given maximum permissible error ϵ if

$$error\,(\mathcal{G}) \;\leq\; \epsilon \tag{3.14}$$

which is a nonlinear constraint equation.

3.2 Formulation of the cost function

The joint assignment of resources, operations, and word lengths is shown in Figure 3.1. For simplicity, two commonly used function types, addition and multiplication, are shown.

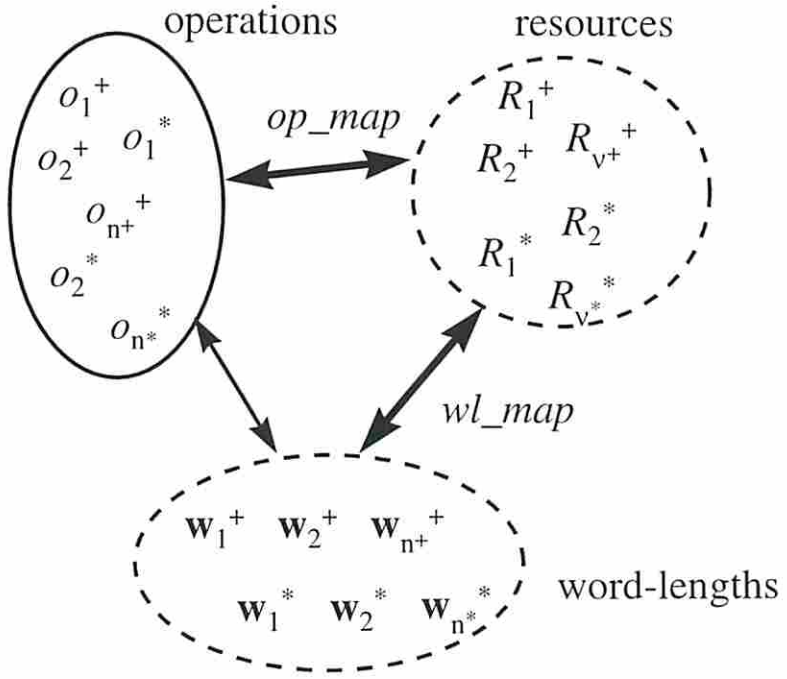


Figure 3.1: Set representation of operation, resources and word lengths mapping

However, the formulation is generalized for an arbitrary number of function types. In algorithms represented as simple flow graphs, all computation operations are fixed and known. Therefore, the set of these operations is a closed set as shown in Figure 3.1. The set of resources is open because the number of resources required depends on the cost and performance constraints. In a given architecture, one or more operations are implemented by each resource. The assignment of operations to resources is given by the function *op_map* defined in Table 3.1. The set of word lengths used in the implementation is also open because it depends on the resource set, and operation–resource mapping. Word lengths are assigned to operations as well as resources in the final design. These assignments are shown by edges between the word-length set, and operation and resource sets. The latter assignment given by the function *wl_map* is also defined in Table 3.1. The cost of a design comprises several parameters such as design size, die cost, cost of designing the circuit, fabrication and packaging cost, and energy/power consumption. Many of these cost parameters are closely related to the design area, hence, in the cost function considered in

this dissertation, the overall circuit area is assumed to represent the design cost. A different cost model to express the energy/power consumption of a circuit in terms of word lengths of its resources is described in Appendix A. This model is based on our system level energy/power prediction technique reported earlier [62]. We now derive a cost (area) function in terms of the resource set and the word–length set. The area of the implemented design is readily obtained from this function when the two sets are known. Table 3.1 summarizes the notation used in the derivation. Note that Greek symbols represent resource attributes and Roman symbols represent operation attributes.

Table 3.1: Notation for the cost function

Symbol	Interpretation	Comments
$\mathcal{G}$	Control data flow graph (CDFG) representation of the algorithm	
n	Total number of arithmetic operations in the graph	
z	Number of function types in the graph	
n^t	Number of operations of type t	$t \in [1, z]$
o_i^t	i–th operation of type t	$i \in [1, n^t]$
ν^t	Number of resources of type t	$\nu^t \leq n^t$
R_k^t	k–th resource of type t	$k \in [1, \nu^t]$
O_k^t	Set of operations mapped to R_k^t	
n_k^t	Number of operations in O_k^t	$\sum_{k=1}^{\nu^t} n_k^t = n^t$
$op\_map\ (i, k)$	0–1 mapping function between operations and resources	
$\mathbf{w}_i^t$	A word–length vector consisting of input/output word lengths of o_i^t	$i \in [1, n^t]$
$\mathbf{w}_{k_l}^t$	Word–length vector associated with operation o_l^t in O_k^t	$l \in [1, n_k^t]$
$\mathcal{W}_k^t$	Set of word–length vectors assigned to operations in O_k^t	
ψ^t	Number of distinct word lengths (vectors) used by resources of type t	$\psi^t \leq \nu^t$
$\boldsymbol{\omega}_r^t$	A word–length vector consisting of	$r \in [1, \psi^t]$

Symbol	Interpretation	Comments
	input/output word lengths of R_k^t	
$wl\_map\ (k, r)$	0–1 mapping function between word lengths and resources	
Ω^t	Set of word–length vectors assigned to resources of type t	

3.2.1 Resource area

We first derive an expression for the area of adders in the design. Summation of such expressions over all function types yields the total functional resource area. In this we assume that resources of one type are not used to implement operations of other types. Let *area* (R_k^+) denote the area of k–th adder. Then,

$$A^+ = \sum_{k=1}^{\nu^+} area(R_k^+) \tag{3.15}$$

is the area of all adders in the design. The adder R_k^+ is shared by additions in the set O_k^+. Therefore, the input/output word lengths of the adder must be as large as the corresponding largest word length assigned to the operands of additions in O_k^+. Thus,

$$\boldsymbol{\omega}_r^+ = \max_l \left[\mathbf{w}_{k_l}^+\right] \qquad \forall\, l = 1 \ldots n_k^+ \tag{3.16}$$

Equation (3.16) implies that word lengths given by the r–th vector in Ω^+ are assigned to R_k^+, in other words, $wl\_map\ (k, r) = 1$. Note that one or more adders in the design may share the same word length[2]. Hence, the cardinality of Ω^+, denoted ψ^+, can be less than or equal to the number of adders, denoted ν^+. This *one–to–one* or *many–to–one* relation is modeled by the *wl_map* function as follows:

$$wl\_map\ (k, r) = \begin{cases} 1 & \text{if } k\text{–th adder uses the } r\text{–th word length vector,} \\ 0 & \text{otherwise.} \end{cases} \tag{3.17}$$

[2]In this thesis, the term word length when associated with operations or resources represents the word length vector.

The area of an adder is a function f^+ of its word lengths[3]. Thus,

$$area(R_k^+) = \sum_{r=1}^{\psi^+} wl\_map\ (k,r) \cdot f^+(\omega_r^+) \tag{3.18}$$

Now, we can write an expression for the area of adders used in the design as

$$A^+ = \sum_{k=1}^{\nu^+} \sum_{r=1}^{\psi^+} wl\_map\ (k,r) \cdot f^+(\omega_r^+) \tag{3.19}$$

Finally, the area of resources of all types, used in the implementation of algorithm $\mathcal{G}$ is given by

$$A^{functional\text{–}resource} = \sum_{t=1}^{z} \sum_{k=1}^{\nu^t} \sum_{r=1}^{\psi^t} wl\_map\ (k,r) \cdot f^t(\omega_r^t) \tag{3.20}$$

This equation indirectly depends on the word lengths of variables in $\mathcal{G}$, through Equation (3.16)[4]. Functional–resource area is also given in the alternate form as

$$A^{functional\text{–}resource} = \sum_{t=1}^{z} \sum_{r=1}^{\psi^t} \nu_r^t \times f^t(\omega_r^t) \tag{3.21}$$

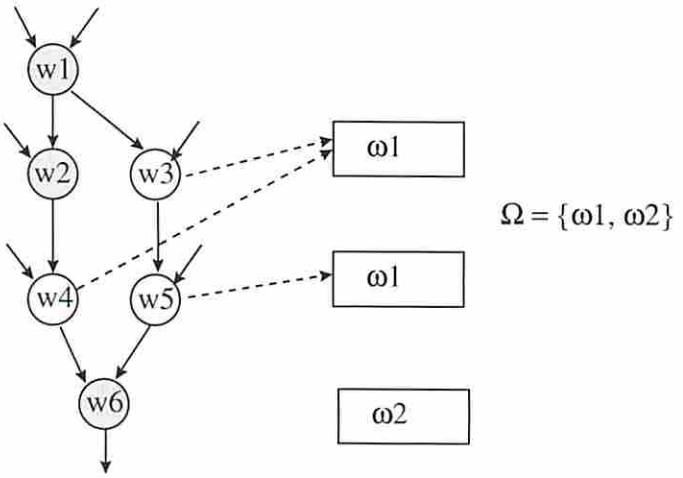
where ν_r^t is the number of resources of r–th word length of type t.

Figure 3.2 shows a typical example, with only one type of operation considered for simplicity. The number of operations in the graph $n = n^1 = 6$, and the number of resources used $\nu = 3$. However, the number of distinct word lengths used ψ is 2. The functions *op_map* and *wl_map* are also shown in the figure as matrices. For this example, Equation (3.20) becomes

$$\begin{aligned} A^{functional\text{–}resource} \;=\; & f(\omega_1) + 0 && k = 1 \\ & + \quad f(\omega_1) + 0 && k = 2 \\ & + \quad 0 + f(\omega_2) && k = 3 \end{aligned} \tag{3.22}$$

[3] Adders having inputs of different word lengths are uncommon. However, this formulation is generalized for any type of functional resources.

[4] In general, equations similar to Equation (3.16) are written for each operation type t.



op_map

k ↓ \ i →	1	2	3	4	5	6
1	0	0	1	1	0	0
2	0	0	0	0	1	0
3	1	1	0	0	0	1

wl_map

k ↓ \ r →	1	2
1	1	0
2	1	0
3	0	1

Figure 3.2: A typical operation, resources and word lengths mapping

3.2.2 Multiplexer and bus driver area

When a resource is shared by two or more operations, the inputs of those operations must be multiplexed. This is done using a multiplexer tree or tristate buffers driving a shared bus. In order for a functional resource to share M inputs, $\lceil \frac{M-1}{m-1} \rceil$ m–way multiplexers or M bus drivers are needed. We first derive an expression for the number of operations sharing a resource, and use it to obtain the total multiplexer or bus driver area. We derive the area equations for addition, and extend the final expression for all function types in the graph.

The mapping function of adders and additions is

$$op\_map\ (k,i) = \begin{cases} 1 & \text{if the } k\text{–th adder implements } i\text{–th addition } o_i^+ \\ 0 & \text{otherwise.} \end{cases} \tag{3.23}$$

Thus, the number of additions sharing the k–th adder, and the corresponding number of m–way ω–bit multiplexers are given by

$$n_k^+ = \sum_{i=1}^{n^+} op\_map\ (k, i) \tag{3.24}$$

$$\#\text{ muxes} = \lceil \frac{n_k^+ - 1}{m - 1} \rceil \tag{3.25}$$

The area of each of these multiplexers is proportional to the word lengths of the associated variables. Consider the set of multiplexers X_k^+, used by R_k^t. There are $\lceil \frac{n_k^+ - 1}{m-1} \rceil$ multiplexers and $m \times \lceil \frac{n_k^+ - 1}{m-1} \rceil$ ω–bit nets in this set. We define a mapping function $mux\_map\ (x, e_j)$, for a multiplexer $x \in X_k^+$, and a net $e_j \in X_k^+$, such that

$$mux\_map\ (x, e_j) = \begin{cases} 1 & \text{if net } e_j \text{ is an input of mux } x, \\ 0 & \text{otherwise.} \end{cases} \tag{3.26}$$

The word length of an m–way multiplexer denoted ω_x, equals the largest of the word lengths of its m inputs. Therefore

$$\boldsymbol{\omega}_x = \max\left[\mathbf{w}e_{j_1}, \ldots, \mathbf{w}e_{j_m}\right] \tag{3.27}$$

where $\mathbf{w}e_{j_a}$ is the word length of the net e_{j_a}, and the nets e_{j_1} through e_{j_m} are inputs to multiplexer x. Incorporating this conjunction into the equation above we have

$$\boldsymbol{\omega}_x = \max\left[\mathbf{w}e_{j_1} \wedge \{mux\_map\ (x, e_{j_1})\}, \ldots, \mathbf{w}e_{j_m} \wedge \{mux\_map\ (x, e_{j_m})\}\right],$$
$$\forall\ e_{j_1}, \ldots, e_{j_m} = 1 \ldots m \times \lceil \frac{n_k^+ - 1}{m - 1} \rceil \tag{3.28}$$

Of the total $\left(m \times \lceil \frac{n_k^+ - 1}{m-1} \rceil\right)$ ω–bit nets, word lengths of n_k^+ nets are determined by the word lengths of variables in O_k^+. The word lengths of the remaining nets are determined by Equation (3.28). Now, the area of multiplexers required by R_k^+ is given by

$$A_k^{mux^+} = \sum_{x=1}^{\lceil \frac{n_k^+ - 1}{m-1} \rceil} f^{mux}(\boldsymbol{\omega}_x) \tag{3.29}$$

The area of all drivers multiplexing variables of the set O_k^+ on a shared bus is proportional to word length vectors of operations in the set, and is given by

$$A^{driver_k^+} = \left(\lceil \frac{n_k^+ - 1}{n_k^+} \rceil \right) \times \sum_{i=1}^{n^+} op\_map\ (k,i) \cdot f^{driver}\ (\mathbf{w_i^+}) \tag{3.30}$$

It is likely that certain operations have dedicated resources (for example, operation w5 in Figure 3.2). A driver is not needed for such cases where n_k^t is 1. Then the term $\lceil \frac{n_k^+ - 1}{n_k^+} \rceil$ in Equation (3.30) becomes 0, and the driver area corresponding to that resource is appropriately 0. For values of $n_k^+ > 1$, the value of $\lceil \frac{n_k^+ - 1}{n_k^+} \rceil$ is always 1.

Depending on the type of functional resource being shared, one or more sets of variables are shared. For example, multiplexers used in additions must select both operands needed for addition. One way to model this is to scale the number of multiplexers (or drivers) by the number of input operands. We assume that functions f^{mux} and f^{driver} incorporate the necessary scaling. Finally, the total area of multiplexers and drivers in the design is obtained by summing Equations (3.29) and (3.30) over all resources of all function types in the graph. Thus we have

$$A^{mux} = \sum_{t=1}^{z} \sum_{k=1}^{\nu^t} \sum_{x=1}^{\lceil \frac{n_k^t - 1}{m-1} \rceil} f^{mux}(\boldsymbol{\omega}_x) \tag{3.31}$$

$$A^{driver} = \sum_{t=1}^{z} \sum_{k=1}^{\nu^t} \left(\lceil \frac{n_k^t - 1}{n_k^t} \rceil \right) \times \sum_{i=1}^{n^t} op\_map\ (k,i) \cdot f^{driver}\ (\mathbf{w_i^t}) \tag{3.32}$$

3.2.3 Register area

The Boolean function that assigns registers to variables, denoted $reg\_var\_map\ (p,v)$, is defined as

$$reg\_var\_map\ (p,v) = \begin{cases} 1 & \text{if } v\text{–th variable is stored in the } p\text{–th register.} \\ 0 & \text{otherwise.} \end{cases} \tag{3.33}$$

The word length of the p–th register, denoted ω_p, is given by

$$\omega_p = \max_v \left[reg\_var\_map\ (p,v) \cdot \mathbf{w}_v \right] \qquad \forall\ v = 1 \ldots m \tag{3.34}$$

where w_v is the word length of the v–th variable, and m is the number of variables (and constants) in the algorithm, previously defined in Section 3.1.3. If there are ν^{reg} registers in the design the total register area is given by

$$A^{reg} = \sum_{p=1}^{\nu^{reg}} f^{reg}\left(\omega_p\right) \tag{3.35}$$

3.2.4 Interconnect area

The nets in a chip may originate from four different sources: namely, input ports, functional resources, multiplexers and registers. This implies that the number of nets is the sum of the number of output bits of modules in each of the above categories. Let π be the number of input ports, each of word length ω_{ip}. We also define a function *out_bits* $(\boldsymbol{\omega})$, which gives the total number of output bits in the vector $\boldsymbol{\omega}$. Now, the total number of nets in the design is written as

$$\begin{aligned}
\nu^{nets} = & \sum_{i=1}^{\pi} \omega_{ip} + \sum_{p=1}^{\nu^{reg}} out_bits\left(\omega_p\right) \\
& + \sum_{t=1}^{z} \sum_{k=1}^{\nu^t} \left\{ \sum_{r=1}^{\psi^t} wl_map(k,r) \cdot out_bits\left(\boldsymbol{\omega}_r^t\right) \right. \\
& + \sum_{x=1}^{\lceil \frac{n_k^t - 1}{m-1} \rceil} out_bits\left(\boldsymbol{\omega}_x\right) \\
& \left. + \left(d_k \cdot \lceil \frac{n_k^t - 1}{n_k^t} \rceil\right) \times \sum_{i=1}^{n^t} op_map\left(k,i\right) \cdot out_bits\left(\mathbf{w}_\mathbf{i}^\mathbf{t}\right) \right\}
\end{aligned} \tag{3.36}$$

The first and second summation terms in the equation above give the number of nets originating from the input ports and registers, respectively. The third summation comprises three terms. The first term gives the number of nets originating from the functional resources. The second and third terms give the number of nets starting from multiplexers and drivers, respectively. The new Boolean variable d_k is "1" if R_k^t is connected to a shared bus. The word length vectors ω_p, $\boldsymbol{\omega}_r^t$, and $\boldsymbol{\omega}_x$ are given by Equations (3.34), (3.16), and (3.28),

respectively. If the average net length is $\bar{l}$, and the average net width is $\bar{w}$[5], the total net area is written as

$$A^{nets} = \nu^{nets} \cdot \bar{l} \cdot \bar{w} \tag{3.37}$$

3.3 Word–length optimization problem formulation

The total design area is obtained by summing areas of all the constituent components.[6]

$$A^{design} = A^{functional-resource} + A^{mux} + A^{driver} + A^{reg} + A^{net} \tag{3.38}$$

Terms on the right hand side in the equation above are defined by Equations (3.20), (3.31), (3.32), (3.35), and (3.36). In these equations, the unknowns to be solved are

- $\mathcal{W}$ word lengths of the algorithm variables,
- ν the number of resources used,
- Ω word lengths of functional resources, and
- ψ the number of different word lengths used by functional resources.

The latter two unknowns depend on $\mathcal{W}$, ν, and the mapping function *op_map*. The number of resources ν, and the mapping function directly affect the desired performance; hence, Ω also directly affects the performance. The choice of word lengths of variables in $\mathcal{G}$, i.e. the elements of $\mathcal{W}$, affects Ω, which in turn affects the cost [ref. Equations (3.20) and (3.38)]. It is obvious that Ω, ν, and ψ cannot be chosen independently of $\mathcal{W}$. Hence, *all four unknowns stated above must be chosen simultaneously* to achieve the design with minimum cost, desired performance, and desired accuracy.

[5]The width of the net also includes the space between adjacent nets.

[6]This total does not include unused area, which has an approximate monotonic relationship with the design area, or control area, which is not directly related to word lengths.

The functions $\mathcal{F}$, relating area and word lengths of resources of various types are non-linear in general. The carry look–ahead adder and multiplier are two examples of this observation. For a given algorithm $\mathcal{G}$ and performance requirement $\boldsymbol{\tau}$, we can abstractly state the cost of implementation $\mathcal{C}$ as

$$\mathcal{C} \;=\; \mathcal{F}\,(\,\mathcal{W},\;\nu,\;\Omega,\;\psi\,) \tag{3.39}$$

Now we formulate the *optimal word–length selection problem for data path optimization* as

$$\begin{gathered} \text{Given } \mathcal{G},\, \boldsymbol{\tau}, \text{ and } \boldsymbol{\epsilon} \\ \textbf{select} \quad \mathcal{W}, \nu, \Omega \quad \text{such that} \\ \mathcal{C} \;=\; \mathcal{F}\,(\,\mathcal{W},\;\nu,\;\Omega,\;\psi\,) \quad \textbf{is minimum}, \\ \textit{error}\;(\mathcal{G}, \mathcal{W}) \leq \boldsymbol{\epsilon}, \quad \text{and} \quad \textit{performance}\;(\mathcal{G}, \nu) \leq \boldsymbol{\tau} \end{gathered} \tag{3.40}$$

3.3.1 Cost of system level components

The implementation of a complete system may comprise more than one chip on an MCM or PC board. On–chip or off–chip memory modules may also be used. The cost of inter–chip communication, especially in terms of energy and power consumption, is significant, and is directly proportional to the number of bits involved. Similarly, the size of memory is directly proportional to the word lengths of the variables stored. Trade offs, such as using larger computational resources (with higher precision on chip), and in effect fewer interconnects between chips are feasible. Inclusion of cost functions of these (and other) components in the overall system–level cost function allows optimum word length selection for system–level designs.

3.4 Problem complexity

The actual area of the implementation can be determined only after allocation and binding of functional resources have been performed and layout generated. Layout generation involves placement of functional and other resources and routing, which are well–known difficult problems. Additionally, resource allocation for minimum area under performance requirements is also believed to be an intractable problem [11]. Therefore selection of the

optimum $\mathcal{W}$, Ω, and ν is an extremely difficult problem to solve. We now consider two constrained versions of the problem and discuss their complexity.

3.4.1 Minimum functional resource area with known word lengths

Since our objective is to determine the optimal operation and resource word lengths, if we assume that they are known, our problem is solved. Assume we know the optimum set of word length vectors assigned to each type of resource and operation. What we demonstrate here is that even so, finding the optimal joint assignment of operations, resources, and word lengths (shown in Figure 3.1 on page 22), that only minimizes the functional resource area, is still a difficult problem. Under the assumption that the optimum Ω is known, we have a set of modules for each function type t where each member of the module set has a distinct area–accuracy characteristic. Since we also know the optimum $\mathcal{W}$, we can easily identify the subset of operations that can be implemented by any module. Now, the problem of determining the optimum ν^t for each t is equivalent to the *generalized module selection* problem. The generalized module selection problem was defined by Jain [21]. Notice, that operations requiring small word lengths can choose between resources of many different word lengths. On the other hand, operations requiring large word lengths have a relatively small choice. Unlike the *module–set selection* technique, also proposed by Jain *et al.* [22], different resources of the same type may have different word lengths. Therefore, the problem of selecting the optimal number of resources of each word length in Ω is a generalized module selection problem. This problem is believed to be combinatorial [8, 21]. It was modeled as a concurrent resource allocation and binding problem and the ILP formulation was given [11]. This approach to determine the optimum ν^t may be very slow or even impractical for large problems.

3.4.2 Word length selection in scheduled data flow graphs

Another way to constrain the word length selection problem is to assume that the data flow graph is scheduled and the optimal operation word lengths are known. In the extreme situation, each operation may be implemented by a dedicated resource of word length the same as that of the operation. Then, $\Omega \subseteq \mathcal{W}$. From the scheduled sequence of operations we have a *compatibility graph* [54]. We can use the *set inclusion model* proposed by Springer

and Thomas [55] to represent the additional hardware required to implement an operation of large word length using a resource of small word length. For example, the cost of an 8–bit addition implemented by a ripple–carry adder is the cost of the set comprising an 8–bit ripple–carry adder. The cost of two 8–bit and one 10–bit additions sharing a resource is the cost of the earlier set plus the cost of another set comprising a 2–bit adder, multiplexers, registers and wiring. It is difficult to determine the exact cost of multiplexers, registers and wiring. However, assuming that the determination of this cost is feasible, the minimum area solution to word length selection can be obtained by solving the weighted clique partitioning problem [55]. The selected sets represent the optimal resource word lengths. While this approach can yield the exact solution for scheduled graphs, the weighted clique partitioning problem was shown to be *NP–hard* [11, 55]. This does not prove that the design problem discussed by Springer and Thomas is *NP–hard*. However, it is believed to be *NP–hard* in the general case. Moreover, this technique applies only when the optimal operation word lengths are known. Recall from Equation (3.40) that $\mathcal{W}$ is one of the unknowns to be optimized in our problem.

We have shown that the constrained versions of the word length selection problem are difficult to solve. While we cannot claim that the unconstrained word–length selection problem is *NP–hard* or *NP–complete*, it is very likely that it is intractable. An overview of our approach to obtain the optimized solution is given next.

3.5 Overview of the approach

Figure 3.3 shows the overall procedure of word–length selection for algorithm optimization. We first analyze the CDFG of the given algorithm and obtain an error constraint in the form of Equation (3.13). We assume that the operating numerical range of each primary input and the corresponding maximum error is specified. In addition, constraints on the word lengths of primary inputs are also taken into consideration. Then, we determine the tight upper–bound word length of each operation in the graph to achieve the maximum accuracy. The error analysis also determines the maximum ψ for each function type. Then, for each possible $\vec{\psi}$ formed by a unique combination of ψ^t for each function type t, an optimized cost function in terms of tight upper–bound word lengths is generated from clustering. In this function the use of resources of multiple word lengths is optimized. A genetic

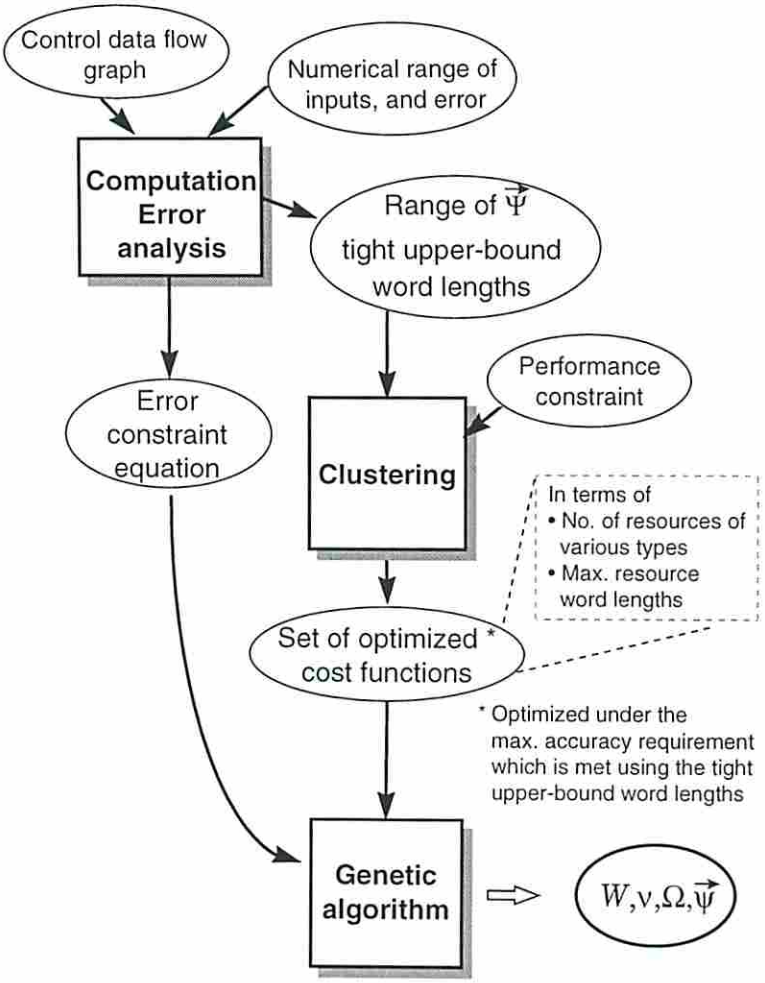


Figure 3.3: Word–length selection for algorithm optimization

algorithm is used to further minimize the set of cost functions while satisfying the accuracy requirements imposed through the error constraint equation. The solution that yields the minimum cost gives the best ψ for each type, the corresponding ν, and also the word lengths of all variables and resources. This procedure may be repeated for various performance constraints.

Chapter 4

Computation error analysis

A systematic approach to model the worst–case error in computations is explained in this chapter. The error is introduced due to truncation of the initial and intermediate operands, and error propagation. We restrict the analysis to simple data–flow graphs with no unbounded data–dependent loops. Section 4.1 describes the error analysis for addition and subtraction. A similar analysis for multiplication is described in Section 4.2. In Section 4.3 we show how closed–form expressions of error in the final results of a simple data–flow graph are obtained using the analysis in the previous sections. The computation of the minimum worst–case error in an algorithm, given its environment (i.e., the primary input word lengths and error) is described in Section 4.4. The knowledge of operation word lengths that yield the maximum accuracy is important because increasing the word lengths beyond these upper bounds does not improve the accuracy of the algorithm. An automatic tool was developed for error analysis. Its functions are described in Section 4.5.

4.1 Analysis of error in addition and subtraction

4.1.1 Basic analysis of error in addition

Recall from Definition 3.1.1 on page 17 that $\tilde{v}$ is the actual or intended value of a variable V. The represented value is v. When two variables X and Y are added, the intended result is $(\tilde{x} + \tilde{y})$. However, the adder produces the sum $(x + y)$.

Definition 4.1.1 The error developed in an arithmetic operation resulting from the errors in the input operands is defined as *calculation error*.

From Equation (3.1) on page 17, $\tilde{x}$ and $\tilde{y}$ are written as

$$\begin{aligned} \tilde{x} &= x + \delta_x \quad \text{where } \delta_x \in [\delta_x^-, \delta_x^+] \\ \tilde{y} &= y + \delta_y \quad \text{where } \delta_y \in [\delta_y^-, \delta_y^+] \end{aligned} \tag{4.1}$$

The corresponding calculation error in addition is given by

$$\epsilon_{add} = (\tilde{x} + \tilde{y}) - (x + y) = \begin{cases} (\delta_x^+ + \delta_y^+) \\ (\delta_x^+ + \delta_y^-) \\ (\delta_x^- + \delta_y^+) \\ (\delta_x^- + \delta_y^-) \end{cases} \tag{4.2}$$

The worst–case under and over approximation calculation errors are given by

$$\epsilon_{add}^+ = (\delta_x^+ + \delta_y^+) \tag{4.3}$$

$$\epsilon_{add}^- = (\delta_x^- + \delta_y^-) \tag{4.4}$$

If some of the less–significant bits of the sum are truncated, *truncation error* is introduced. The upper–bound error corresponding to truncation of $l = k - m$ least significant bits denoted $\epsilon^{trunc}(m)$ is obtained from Lemma 3.1.2 on page 18. Here k and m are the number of fractional bits in the sum, before and after truncation respectively.

$$\epsilon^{trunc}(m) = \begin{cases} 2^{-m} - 2^{-k} & \text{if} \quad m < k \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

If the word length assigned to the sum is greater than the number of bits needed, two possibilities shown in Figure 4.1 exist. If sign extension is implemented the number of fractional

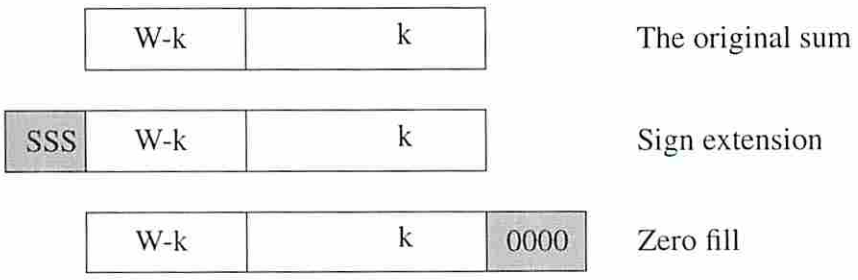


Figure 4.1: Sign extension and zero fill applied to the sum.

bits in the new sum remains to be k. The truncation error in this case is zero. Alternatively, extra fractional bits are added to the sum and are zero filled. In this case also, the truncation error is zero. According to Lemma 3.1.1 on page 17, $\epsilon^{trunc}(m)$ is always positive. The total worst–case under– and over–approximation errors in addition are

$$\delta^{+}_{add} = \epsilon^{+}_{add} + \epsilon^{trunc}(m) \tag{4.6}$$

$$\delta^{-}_{add} = \begin{cases} \epsilon^{-}_{add} + \epsilon^{trunc}(m) & \text{if} \quad (\epsilon^{-}_{add} + \epsilon^{trunc}(m)) < 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.7}$$

Notice that the final error is expressed as a function of the input and output word lengths[1].

4.1.2 Manipulations to avoid unnecessary errors in addition

In fixed point addition (or subtraction) it is required that the locations of the binary points in both the inputs are aligned. It is possible that the fractional and/or total number of bits in the two operands are different. In that situation, circuit designers apply *zero fill* or *sign extension*. The choice depends on which method reduces the final error. In our error analysis, we first determine the correct binary–point adjustment method and derive the worst–case error equation accordingly. Thus, the bounds are guaranteed to be tight bounds. Next, we explain the selection procedure using examples.

Let us consider addition of two variables X and Y. Let their respective total word lengths, integer word lengths and fractional word lengths be denoted $\mathrm{w}_x, \mathrm{w}_y$, $\mathrm{i}_x, \mathrm{i}_y$, k_x, and k_y. Naturally $\mathrm{w}_x = \mathrm{i}_x + k_x$ and $\mathrm{w}_y = \mathrm{i}_y + k_y$. For simplicity let us further assume that $\mathrm{w}_x < \mathrm{w}_y$. Analysis for the case where $\mathrm{w}_x > \mathrm{w}_y$ is symmetric. Finally we assume that the number of bits in the adder denoted $\mathrm{w} \geq \mathrm{w}_y$. Hence, sign extension or zero fill is applied to X.

Case 1: $k_x = k_y$

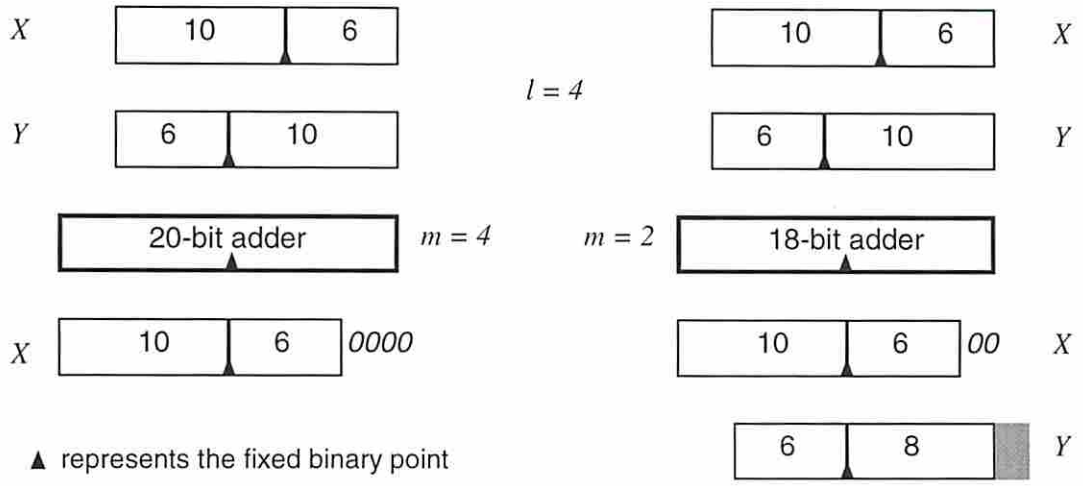
Equations (4.6) and (4.7) apply with no change.

Case 2: $k_x < k_y$

Let $l = k_y - k_x$. We first attempt to *left shift* X with a zero fill. The complete

[1] In most applications it is common to truncate the fractional bits only. However, the equations apply to integer bit truncations as well.

shift of l bits is feasible if $l \leq (\mathrm{w} - \mathrm{w}_x)$. Equations (4.6) and (4.7) still apply without any change in that case. Otherwise, left shift X by $m = (\mathrm{w} - \mathrm{w}_x)$ bits and truncate $(l - m)$ least significant bits of Y. Examples of both the possibilities are shown in Figure 4.2.



a) Zero fill when adder is of sufficient size

b) Zero fill and truncation when adder is too small for operands

Figure 4.2: Binary point alignment when $k_x < k_y$

When Y must be truncated, δ_y^+ and δ_y^- change as

$$\delta_y^{+new} = \delta_y^{+old} + (2^{-k_y+l-m} - 2^{-k_y}) \tag{4.8}$$

$$\delta_y^{-new} = \min\,(0,\, \delta_y^{-old} + (2^{-k_y+l-m} - 2^{-k_y})) \tag{4.9}$$

Equations (4.6) and (4.7) change accordingly.

Case 3: $k_x > k_y$

Let $l = k_x - k_y$ and $m = \mathrm{w} - \mathrm{w}_y$. If $m \geq l$, left shift Y by l bits. Equations (4.6) and (4.7) apply with no change. Otherwise, if $m > 0$, left shift Y by m bits and truncate the $(l - m)$ least significant bits of X. These two possibilities are shown in Figure 4.3.

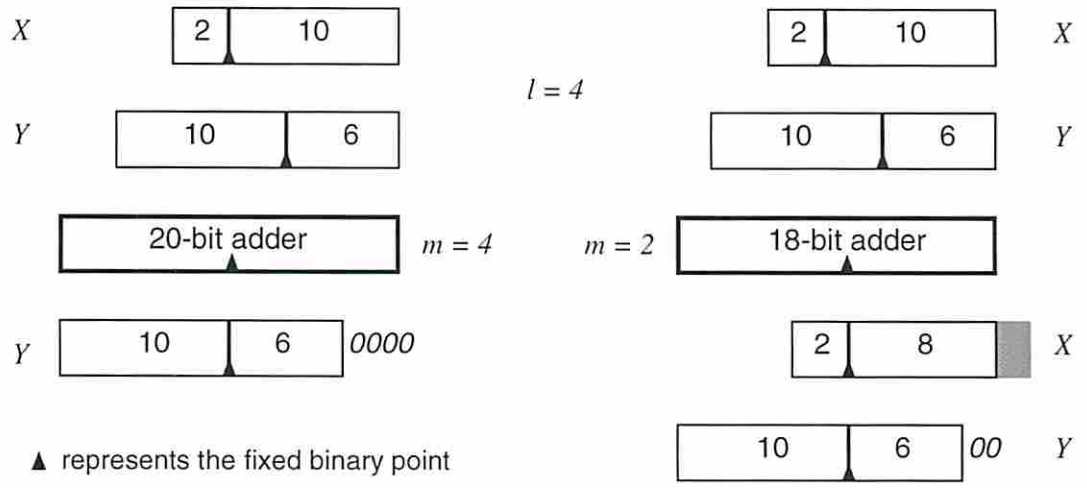


Figure 4.3: Binary point alignment when $k_x > k_y$

Then, δ_x^+ and δ_x^- change as

$$\delta_x^+ = \delta_x^+ + (2^{-k_x+l-m} - 2^{-k_x}) \tag{4.10}$$

$$\delta_x^- = \min\,(0,\, \delta_x^- + (2^{-k_x+l-m} - 2^{-k_x})) \tag{4.11}$$

and, Equations (4.6) and (4.7) change accordingly.

THEOREM 4.1.1 *The specific choice of shifting bits of inputs X and Y guarantees that their binary points are aligned, and that the additional truncation error introduced is minimum.*

Proof Consider the two possibilities in Case 2, where $k_y = k_x + l$. If X is shifted left by l bits, the new X has $k_x + l = k_y$ fractional bits and binary points are aligned. If X is shifted left by m bits, then Y is shifted right by $(l - m)$ bits. The new X and Y have $(k_x + m)$ and $(k_y - l + m) = (k_x + m)$ bits and their binary points are aligned. Analysis for Case 3 is symmetric where both X and Y either have $(k_y + l)$ or $(k_y + m)$ fractional bits.

Zero fill increases the number of fractional bits however, the corresponding truncation error does not decrease because the value of the variable is unchanged. Consider Case 2 where $(l - m)$ least significant bits of Y are truncated. Instead, if only $(l - m - k)$ bits

are truncated then for the binary points to align, X must be left shifted by $m + k$ places. Consequently, the total number of bits in X denoted w'_x is given by

$$\begin{aligned} \mathrm{w}'_x &= \mathrm{i}_x + k_x + m + k \\ &= \mathrm{w} + k \\ &> \mathrm{w} \qquad \forall\, k > 0 \end{aligned}$$

The new X cannot be an operand of the given adder of w bits. Therefore, the minimum number of bits that must be truncated in Y is $(l - m)$ and the corresponding error is minimum. Analysis for Case 3 is symmetric. □

4.1.3 Basic analysis of error in subtraction

It is common to characterize subtraction as addition when both operands are allowed to be positive and negative. Unlike addition however, subtraction is not commutative. $X - Y \neq Y - X$. Hence, we modify Equations (4.2), (4.3), and (4.4) for calculation error in the subtraction $X - Y$ as

$$\epsilon_{sub} = (\tilde{x} - \tilde{y}) - (x - y) = \begin{cases} (\delta_x^+ - \delta_y^+) \\ (\delta_x^+ - \delta_y^-) \\ (\delta_x^- - \delta_y^+) \\ (\delta_x^- - \delta_y^-) \end{cases} \tag{4.12}$$

The worst–case under and over approximation calculation errors are given by

$$\epsilon_{sub}^+ = (\delta_x^+ - \delta_y^-) \tag{4.13}$$

$$\epsilon_{sub}^- = (\delta_x^- - \delta_y^+) \tag{4.14}$$

Equations for the subtraction $(Y - X)$ are symmetric. The error corresponding to truncation of the result is independent of the arithmetic operation therefore, Equation (4.5) is

applicable without any change. The total worst–case under– and over– approximation errors in subtraction are

$$\delta^{+}_{sub} = \epsilon^{+}_{sub} + \epsilon^{trunc}(m) \tag{4.15}$$

$$\delta^{-}_{sub} = \min\left(0,\ (\epsilon^{-}_{sub} + \epsilon^{trunc}(m))\right) \tag{4.16}$$

Finally, the discussion in Section 4.1.2 on operand shifts necessary before addition/ subtraction is directly applicable to subtraction as well. In the error analysis we assume the optimum shifts for the given operand and resource word lengths, and model the minimum error accordingly. This allows the best use of available word lengths.

4.2 Analysis of error in multiplication

The error expressions for the results of addition and subtraction are independent of the numerical ranges of the input operands. In case of multiplication however, the worst–case calculation error depends significantly on these ranges. Calculation error in multiplication is given by

$$\epsilon_{mult} = (\tilde{x} * \tilde{y}) - (x * y) = \max \left\{ \begin{array}{lcl} (x \cdot \delta^{+}_{y} + y \cdot \delta^{+}_{x} + \delta^{+}_{x} \cdot \delta^{+}_{y}) & \triangleq & \epsilon_1 \\ (x \cdot \delta^{+}_{y} + y \cdot \delta^{-}_{x} + \delta^{-}_{x} \cdot \delta^{+}_{y}) & \triangleq & \epsilon_2 \\ (x \cdot \delta^{-}_{y} + y \cdot \delta^{+}_{x} + \delta^{+}_{x} \cdot \delta^{-}_{y}) & \triangleq & \epsilon_3 \\ (x \cdot \delta^{-}_{y} + y \cdot \delta^{-}_{x} + \delta^{-}_{x} \cdot \delta^{-}_{y}) & \triangleq & \epsilon_4 \end{array} \right\} \tag{4.17}$$

In order to obtain a closed form expression for the worst case ϵ_{mult} we need to determine the maximum and the minimum values of ϵ_1, ϵ_2, ϵ_3, and ϵ_4.

$$\max(\epsilon_1) = \max \left\{ \begin{array}{l} \overline{x} \cdot \delta^{+}_{y} + \overline{y} \cdot \delta^{+}_{x} \\ \overline{x} \cdot \delta^{+}_{y} + \underline{y} \cdot \delta^{+}_{x} \\ \underline{x} \cdot \delta^{+}_{y} + \overline{y} \cdot \delta^{+}_{x} \\ \underline{x} \cdot \delta^{+}_{y} + \underline{y} \cdot \delta^{+}_{x} \end{array} \right\} + \delta^{+}_{x} \cdot \delta^{+}_{y} \tag{4.18}$$

where $x \in [\underline{x},\ \overline{x}]$, and $y \in [\underline{y},\ \overline{y}]$.

$\min\ (\epsilon_1)$ is obtained by replacing '$\max\{\cdot\}$' in Equation (4.18) with '$\min\{\cdot\}$'. Expressions for maximum and minimum ϵ_2, ϵ_3, and ϵ_4 are obtained by substituting for δ^{+}_{x}

and δ_y^+ according to Equation (4.17). Then, the worst case under and over approximation calculation errors in multiplication are

$$\epsilon_{mult}^+ = \max\left(0, \max\left\{\max\left(\epsilon_1\right), \max\left(\epsilon_2\right), \max\left(\epsilon_3\right), \max\left(\epsilon_4\right)\right\}\right) \tag{4.19}$$

$$\epsilon_{mult}^- = \min\left(0, \min\left\{\min\left(\epsilon_1\right), \min\left(\epsilon_2\right), \min\left(\epsilon_3\right), \min\left(\epsilon_4\right)\right\}\right) \tag{4.20}$$

We have shown the analysis of truncation error for addition in the previous section. While the analysis for multiplication is identical, truncation is significant in this case. As mentioned before, the product of M– and N–bit 2's complement numbers consists of $(M+N)$ bits. Potentially, the number of bits in the product is twice the number of bits in an input. If the permissible output word length is comparable to that of the inputs, it is likely that almost the entire least–significant half of the bits are truncated. The positions of the binary points of the inputs need not be aligned for multiplication. Let the number of fractional bits in inputs X and Y be k_x and k_y, respectively. Then, the truncation error corresponding to $l = k_x + k_y - m$ least significant bits denoted $\epsilon^{trunc}(m)$ is given by

$$\epsilon^{trunc}(m) = \begin{cases} 2^{-m} - 2^{-(k_x+k_y)} & \text{if} \quad m < k_x + k_y \\ 0 & \text{otherwise} \end{cases} \tag{4.21}$$

The total worst–case under– and over– approximation errors in multiplication are

$$\delta_{mult}^+ = \epsilon_{mult}^+ + \epsilon^{trunc}(m) \tag{4.22}$$

$$\delta_{mult}^- = \begin{cases} \epsilon_{mult}^- + \epsilon^{trunc}(m) & \text{if} \quad (\epsilon_{mult}^- + \epsilon^{trunc}(m)) < 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.23}$$

Note that the final error is expressed as a function of the input and output word lengths and also the limits of the numerical ranges of the inputs.

4.3 Error analysis in simple data flow graphs

4.3.1 Analysis of error in the computation of other functions

In complex numerical computations, other functions such as division, reciprocal, logarithm, square-root and sine are also used. Therefore, it is important to be able to model the computation error in these functions. One way to determine the worst–case error is to perform analysis similar to that carried out for addition or multiplication. For example, the worst–case calculation error produced by the sine function is

$$\begin{aligned}
\epsilon_{sin} &= \sin(\tilde{x}) - \sin(x) \\
&= \sin(x + \delta_x) - \sin(x) \\
&= (\sin(x) \cdot \cos(\delta_x) + \cos(x) \cdot \sin(\delta_x)) - \sin(x) \\
&\approx (\sin(x) \cdot \sqrt{1 - \delta_x^2} + \cos(x) \cdot \delta_x) - \sin(x) \\
&\approx \sin(x)(\sqrt{1 - \delta_x^2} - 1) + \delta_x \cdot \cos(x)
\end{aligned} \tag{4.24}$$

In this equation δ_x is assumed to be a small fraction of x. The worst–case value of ϵ_{sin} is δ_x and occurs when x approaches zero. On the other hand, hardware and software implementations of nonlinear, elementary functions typically comprise additions, subtractions and multiplications [2] [13,23,27,50,51,63]. These more complex operations are expressed using data–flow graphs involving only additions, subtractions and multiplications. Our method described below to analyze the computation error in the final outputs of a simple DFG can be used directly to analyze errors in the above mentioned and other similar operations. This approach has the following two advantages:

1. When deriving equations similar to Equation (4.24) we assume that the pure computation of the function does not introduce any error. In other words, the error in the computation is zero if the input error is zero and all the output bits are preserved. This assumption is true in the case of addition and multiplication but not valid in the case of elementary functions. Error introduced by the computation depends on the algorithm used. For example, a Taylor series expansion using 6 significant terms or 100 significant terms would yield a different computation error. Since equations like

[2] Typically one or more table look-up operations are required and the associated error due to finite word length of the memory is modeled as truncation (or round off) error.

Equation (4.24) have no knowledge of the implementation algorithm, the actual error in computation is likely to be greater than the worst case bounds determined by the equations.

2. Many different algorithms exist for the computation of a nonlinear function. A different DFG is associated with each algorithm and accordingly, the error analysis varies although the function and its inputs are identical. Error analysis using simple DFGs allows the designer to

   (a) model the error corresponding to the implementation or,

   (b) to choose an implementation with the desired cost/error characteristic.

4.3.2 Analysis of error in the final outputs of a simple DFG

In this section we consider two simple examples and demonstrate how closed–form expressions for the error in the final results are obtained. The method is applicable to all DFGs with no unbounded data–dependent loops.

4.3.2.1 Example: Two additions

The first example is Two additions. Consider the DFG shown in Figure 4.4. All the values

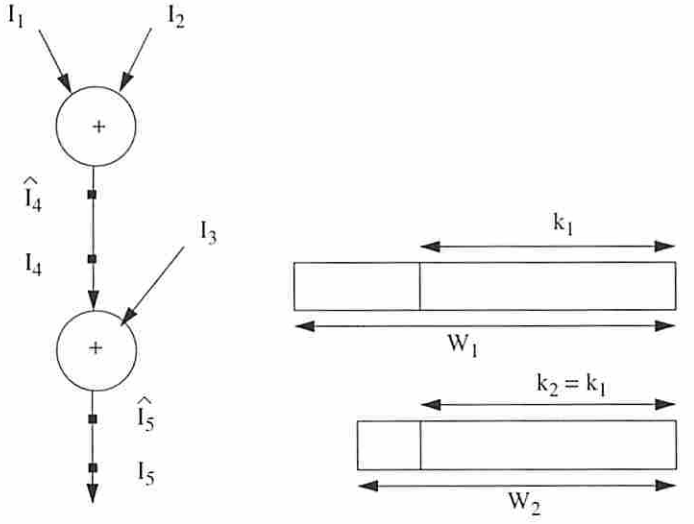


Figure 4.4: Two additions DFG

in the graph are represented as fixed–point 2's complement binary numbers. The word lengths associated with each variable I_j are represented as

$$\mathrm{W}_{I_j} \triangleq \mathrm{w}_j : k_j$$

where w_j is the total word length associated with I_j, and k_j is the number of fractional bits. I_1, I_2, and I_3 are the primary inputs. $\widehat{I_4}$, I_4 and $\widehat{I_5}$ are the intermediate results, and I_5 is the final output of the DFG. $\widehat{I_j}$ is a result obtained by preserving all the output bits from the preceeding operation. The corresponding I_j is the result after truncation. I_5 is the final output of the DFG.

While the overall objective is to determine the optimal values of w_j, the designer may impose certain constraints for arbitrary reasons[3]. If any such constraints are known *a priori*, they are taken into consideration in the analysis. For example, in the present DFG we assume $k_1 = k_2$, and $k_3 = k_4$. Otherwise, arithmetic shifts must be performed on the inputs I_1 and/or I_2 as described in Section 4.1.2.

Word–lengths of $\widehat{I_4}$ and I_4 are

$$\mathrm{W}_{\widehat{I_4}} \triangleq \max(\mathrm{w}_1, \mathrm{w}_2) + 1 : k_1 \tag{4.25}$$

$$\begin{aligned} \mathrm{W}_{I_4} &\triangleq \mathrm{w}_4 : (k_1 - (\max(\mathrm{w}_1, \mathrm{w}_2) + 1 - \mathrm{w}_4)) \\ &\triangleq \mathrm{w}_4 : k_4 \end{aligned} \tag{4.26}$$

where the total word length of I_4, denoted w_4, is unknown. In order to write expressions for the word lengths of $\widehat{I_5}$ and I_5 we assume that $k_3 = k_4$. Thus,

$$\mathrm{W}_{\widehat{I_5}} \triangleq \max(\mathrm{w}_3, \mathrm{w}_4) + 1 : k_4 \tag{4.27}$$

$$\begin{aligned} \mathrm{W}_{I_5} &\triangleq \mathrm{w}_5 : (k_4 - (\max(\mathrm{w}_3, \mathrm{w}_4) + 1 - \mathrm{w}_5)) \\ &\triangleq \mathrm{w}_5 : k_5 \end{aligned} \tag{4.28}$$

where the total word length of I_5 denoted w_5 is also an unknown.

[3]For example, the word lengths may be required to be even numbers, or word lengths greater than a certain value may not be allowed.

Either truncation, calculation or both types of errors are associated with all the values in the flow graph. The corresponding under–approximation error expressions are

$$\epsilon^+(I_1) = 0 + [\delta_1^+] \tag{4.29}$$

$$\epsilon^+(I_2) = 0 + [\delta_2^+] \tag{4.30}$$

$$\epsilon^+(\widehat{I_4}) = [\delta_1^+ + \delta_2^+] + 0 \tag{4.31}$$

$$\epsilon^+(I_4) = [\delta_1^+ + \delta_2^+] + [\max(0, (2^{-k_4} - 2^{-k_1}))]$$

substituting for k_4 from Equation (4.26),

$$= [\delta_1^+ + \delta_2^+] + [\max(0, (2^{-k_1}(2^{(\max(w_1,w_2)+1-w_4)} - 1)))] \tag{4.32}$$

$$\epsilon^+(I_3) = 0 + [\delta_3^+] \tag{4.33}$$

$$\epsilon^+(\widehat{I_5}) = [\delta_1^+ + \delta_2^+ + \max(0, (2^{-k_4} - 2^{-k_1})) + \delta_3^+] + 0 \tag{4.34}$$

$$\epsilon^+(I_5) = [\delta_1^+ + \delta_2^+ + \delta_3^+ + \max(0, (2^{-k_4} - 2^{-k_1}))] + [\max(0, (2^{-k_5} - 2^{-k_4}))]$$

substituting for k_5, and k_4 from Equations (4.26) and (4.28) respectively,

$$= [\delta_1^+ + \delta_2^+ + \delta_3^+ + \max(0, (2^{-k_1}(2^{(\max(w_1,w_2)+1-w_4)} - 1)))] + [\max(0, (2^{(k_1-(\max(w_1,w_2)+1-w_4))} \times (2^{(\max(w_3,w_4)+1-w_5)} - 1)))] \tag{4.35}$$

The worst case under and over approximation errors in the primary inputs are denoted δ_i^+, and δ_i^- respectively. Their values are known *a priori*. In all the expressions, the term in the first square brackets corresponds to the calculation error and that in the second brackets to the truncation error.

4.3.2.2 Example: Three multiplications

Figure 4.5 shows the flow graph for the example Three multiplications. The notation used here is the same as that used in the previous example. I_1, I_2, I_3, and I_4 are the primary inputs, $\widehat{I_5}$, I_5, $\widehat{I_6}$, I_6, and $\widehat{I_7}$ are the intermediate results and I_7 is the final result. We also

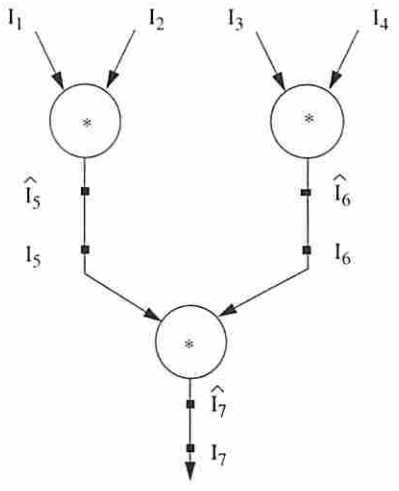


Figure 4.5: Three multiplications DFG

assume the following constraints that $w_5 < w_1 + w_2$, $w_6 < w_3 + w_4$, and $w_7 < w_5 + w_6$. The word lengths of all the results in the flow graph are

$$W_{\hat{I}_5} \triangleq w_1 + w_2 : k_1 + k_2 \tag{4.36}$$

$$W_{I_5} \triangleq w_5 : k_1 + k_2 - w_1 - w_2 + w_5 \tag{4.37}$$

$$\triangleq w_5 : k_5$$

$$W_{\hat{I}_6} \triangleq w_3 + w_4 : k_3 + k_4 \tag{4.38}$$

$$W_{I_6} \triangleq w_6 : k_3 + k_4 - w_3 - w_4 + w_6 \tag{4.39}$$

$$\triangleq w_6 : k_6$$

$$W_{\hat{I}_7} \triangleq w_5 + w_6 : k_5 + k_6 \tag{4.40}$$

$$W_{I_7} \triangleq w_7 : k_5 + k_6 - w_5 - w_6 + w_7 \tag{4.41}$$

$$\triangleq w_7 : k_7$$

The under–approximation error expressions for all the values are obtained by applying Equations (4.17), (4.18), (4.19), and (4.21). These expressions are complicated because of the "max" and "min" operators in the basic equations. While all the cases considered in the basic equations must be evaluated in practice, here we make simplifying assumptions to obtain expressions that are more readable. First we assume that the over–approximation

errors δ^- in all the primary inputs are zero. Therefore, we evaluate ϵ_1 only from Equation (4.17). Next, we assume that the primary inputs are always positive. Consequently, Equations (4.18) and (4.19) reduce to

$$\epsilon_1 = \overline{x} \cdot \delta_y^+ + \overline{y} \cdot \delta_x^+ + \delta_x^+ \cdot \delta_y^+ \triangleq \epsilon_{mult} \tag{4.42}$$

where X and Y are the two input operands of the multiplication, and $\overline{x}$ and $\overline{y}$ are their maximum values respectively. Finally, the previously stated assumptions $\mathrm{w}_5 < \mathrm{w}_1 + \mathrm{w}_2$, $\mathrm{w}_6 < \mathrm{w}_3 + \mathrm{w}_4$, and $\mathrm{w}_7 < \mathrm{w}_5 + \mathrm{w}_6$ imply that the truncation error given by Equation (4.21) is strictly positive. Now, the under approximation error expressions for all the values in Three multiplications are

$$\epsilon^+(I_1) = 0 + [\delta_1^+] \tag{4.43}$$

$$\epsilon^+(I_2) = 0 + [\delta_2^+] \tag{4.44}$$

$$\epsilon^+(\widehat{I_5}) = [\overline{x_1} \cdot \delta_2^+ + \overline{x_2} \cdot \delta_1^+ + \delta_1^+ \cdot \delta_2^+] + 0 \tag{4.45}$$

$$\epsilon^+(I_5) = [\overline{x_1} \cdot \delta_2^+ + \overline{x_2} \cdot \delta_1^+ + \delta_1^+ \cdot \delta_2^+] + [2^{-k_5} - 2^{-k_1-k_2}]$$

substituting for k_5 from Equation (4.37),

$$= [\overline{x_1} \cdot \delta_2^+ + \overline{x_2} \cdot \delta_1^+ + \delta_1^+ \cdot \delta_2^+] + [2^{-k1-k2} \cdot (2^{\mathrm{w}_1+\mathrm{w}_2-\mathrm{w}_5} - 1)] \tag{4.46}$$

$$\triangleq \delta_5^+$$

$$\epsilon^+(I_3) = 0 + [\delta_3^+] \tag{4.47}$$

$$\epsilon^+(I_4) = 0 + [\delta_4^+] \tag{4.48}$$

$$\epsilon^+(\widehat{I_6}) = [\overline{x_3} \cdot \delta_4^+ + \overline{x_4} \cdot \delta_3^+ + \delta_3^+ \cdot \delta_4^+] + 0 \tag{4.49}$$

$$\epsilon^+(I_6) = [\overline{x_3} \cdot \delta_4^+ + \overline{x_4} \cdot \delta_3^+ + \delta_3^+ \cdot \delta_4^+] + [2^{-k_6} - 2^{-k_3-k_4}]$$

substituting for k_6 from Equation (4.39),

$$= [\overline{x_3} \cdot \delta_4^+ + \overline{x_4} \cdot \delta_3^+ + \delta_3^+ \cdot \delta_4^+] + [2^{-k3-k4} \cdot (2^{\mathrm{w}_3+\mathrm{w}_4-\mathrm{w}_6} - 1)] \tag{4.50}$$

$$\triangleq \delta_6^+$$

$$\epsilon^+(\widehat{I_7}) = [\overline{x_5} \cdot \delta_6^+ + \overline{x_6} \cdot \delta_5^+ + \delta_5^+ \cdot \delta_6^+] + 0$$

$$= \left[(\overline{x_1} \cdot \overline{x_2}) \cdot \delta_6^+ + (\overline{x_3} \cdot \overline{x_4}) \cdot \delta_5^+ + \delta_5^+ \cdot \delta_6^+\right] + 0 \tag{4.51}$$

$$\epsilon^+(I_7) = \left[(\overline{x_1} \cdot \overline{x_2}) \cdot \delta_6^+ + (\overline{x_3} \cdot \overline{x_4}) \cdot \delta_5^+ + \delta_5^+ \cdot \delta_6^+\right] + \left[2^{-k_7} - 2^{-k_5-k_6}\right]$$

substituting for k_5, k_6, and k_7 from Equations (4.37), (4.39), and (4.41) respectively,

$$= \left[(\overline{x_1} \cdot \overline{x_2}) \cdot \delta_6^+ + (\overline{x_3} \cdot \overline{x_4}) \cdot \delta_5^+ + \delta_5^+ \cdot \delta_6^+\right] + \left[2^{(w_1+w_2+w_3+w_4)-(k_1+k_2+k_3+k_4)-w_5-w_6} \cdot (2^{w_5+w_6-w_7} - 1)\right] \quad (4.52)$$

4.3.2.3 Generalized form of the error equation of the final result of a DFG

Observe from Equations (4.35) and (4.52) that the total worst–case error is expressed in terms of

1. the range limits of the primary inputs (constants since the values of the range limits are known *a priori*),
2. the worst case under and over approximation errors in the primary inputs (constants, also known *a priori*), and
3. the word length and the number of fractional bits in all the values in the DFG (the unknowns to be optimized).

Now we write the generalized form of the total worst–case error expression of the final output of a simple DFG as

$$\delta_{DFG} = \sum_i \alpha_i \, 2^{a_i w_i} + \sum_i \beta_i \, 2^{b_i k_i} + \gamma \qquad i = 1, 2, \cdots V. \quad (4.53)$$

where V is the total number of values in the DFG and α_i, a_i, β_i, b_i, and γ are known constants. Notice that neither calculation nor truncation error equations are exponential functions of input error. Therefore, all exponents in Equation (4.53) are linear combinations of the total and fractional word lengths. Product terms such as $(w_i \cdot w_j)$, $(k_i \cdot k_j)$, and $(w_i \cdot k_j)$ do not appear. The error expression, however, is nonlinear.

4.3.2.4 Control structures in flow graphs

When an 'if–then–else' structure is used, alternate sequences of computations or paths exist in the CDFG. The worst–case error corresponding to each path is modeled separately.

Typically, the path that generates the maximum error is used in further modeling. However, if different kinds of operations are performed along alternate paths, word lengths of different kinds of functional operators need to be optimized. In some algorithms the result of a computation is shared by two or more successor operations. An intermediate 'distribute' node is then used as shown in Figure 4.6. There is no calculation error involved

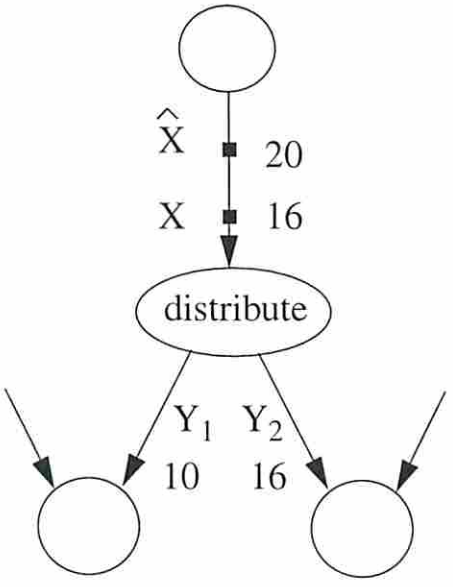


Figure 4.6: 'Distribute' node in flow graphs and associated error

with this node, however, additional truncation error may be introduced. This possibility exists when an input with word length smaller than that of the result is sufficient for one of the operations sharing the result. In Figure 4.6 for example, $\mathrm{w}_{Y_1} < \mathrm{w}_{Y_2}$ and $\mathrm{w}_{Y_2} = \mathrm{w}_X$. The total worst case error in Y_1 is then given by

$$\delta^+_{Y_1} = \delta^+_X + \epsilon^{trunc}(l) \tag{4.54}$$

$$\delta^-_{Y_1} = \min\left(0,\, \delta^-_X + \epsilon^{trunc}(l)\right) \tag{4.55}$$

where δ^+_X and δ^-_X are the worst–case under– and over–approximation errors in X, respectively. The number of bits preserved in Y_1 is l, and $\epsilon^{trunc}(l)$ is the error introduced due to truncation of the remaining least–significant bits of Y_1.

4.4 Tight upper bound on word lengths for minimum worst–case error

In the previous sections we have shown how closed form expressions of error in the final result can be derived in terms of word lengths of variables in the algorithm. Given a set of word lengths, from Equation 4.53 we can determine the worst–case error in the final result of the algorithm. If all the bits in each operand are preserved (i.e. no truncations are allowed) the corresponding worst–case error in the final result is the minimum worst–case error. In spite of preserving all bits, the computation error is not necessarily zero. This is because the primary inputs of the computation may carry error from the external environment. For example, digital filter inputs may have quantization error from A/D converters. This error propagates through the computations and the corresponding error in each operand assuming adequate word lengths is its minimum worst–case error. If the error in all primary inputs is zero the minimum worst–case error is also zero. Preserving all bits yields the maximum achievable accuracy of an algorithm for the specified environment. For each operand the corresponding word length is an upper bound because assigning a larger word length to any operand does not affect the computational accuracy of the algorithm. Hence it is essential in error analysis to determine the minimum worst–case error and the word lengths of all operands in the algorithm that guarantee it. The upper–bound word lengths can be naively computed using rules of elementary binary arithmetic. For example, the sum of two N–bit numbers contains $(N + 1)$ bits. The product of an M–bit and an N–bit numbers contains $(M + N)$ bits. However, bounds thus obtained do not take into account the numerical range of the partial and final results of a computation. This is illustrated in the following example.

Consider the summation $S = \sum_{i=1}^{256} A_i$, where A_i are 8–bit numbers in the range $[0, 1]$. In a sequential algorithm, based on a naive analysis, the first sum has 9 bits, and the last one i.e. S requires 263 bits. However, from the range analysis we determine that each A_i has 1 integer and 7 fractional bits in unsigned representation. The maximum value of S is 256 requiring only 9 integer bits. Thus, the tight upper bound on the word length of S is 16 as opposed to 263. In error analysis we first perform the numerical range analysis to determine the numerical range of all intermediate operands and final results by analyzing the sequence of arithmetic functions represented by the CDFG. Then for each operand,

we determine the minimum number of integer bits needed to represent its range. Finally, the tight upper–bound word length is determined by considering the fractional bits and no truncations. Consider a computation involving two primary inputs a and b, let $x = a + b$ and the final result $z = x^2$. Table 4.1 shows the numerical ranges and the tight upper–bound word lengths of a, b, x, and z.

Table 4.1: Numerical range of variables and the tight upper–bound word lengths

Variable	Range	no. of integer bits	no. of fractional bits	total word length
a	[0, 1]	1	13	14
b	[0, 1000]	10	13	23
$x = a + b$	[0, 1001]	10	13	23
$z = x^2$	[0, 1002001]	19	26	45

When the tight upper–bound word lengths for all operations in the algorithm are determined, there may be dozens of word lengths required for each operation type. Since all bits are always preserved, word lengths of some operands could also be very large. For example, a 1-D DCT algorithm requires 25 distinct word lengths for multiplication ranging from 8 bits through 244 bits. Cost considerations may force the designer to limit the word lengths as well as the number of distinct word lengths with acceptable computation error.

4.5 Automatic tool for error analysis

An automatic tool was developed to perform error analysis. Its functions are listed below.

i. First the tight upper–bound word lengths are determined by performing range analysis.

ii. By counting the number of distinct tight upper–bound word lengths of all the operations of a type t, the maximum number of distinct resource word lengths denoted $\langle \max . \psi^t \rangle$ is determined. At least one word length must always be used by resources of any type. Therefore, the range of ψ is $[1, \langle \max . \psi^t \rangle]$.

iii. For the given values of operand word lengths, the error analysis tool internally generates an error expression in the form of Equation (4.53) and computes the numerical values of the worst case under and over approximation errors.

iv. If the maximum permissible error in the final result is specified, the tool determines whether or not the specified word length set can be used in the implementation by comparing the two error values. The maximum permissible error may be specified as a constant or a fraction of the maximum numerical value of the final result.

The error analysis tool implements the analytical model of computation error described in this chapter and can replace exhaustive–simulation–based estimates of the worst–case error suggested in previous work.

Chapter 5

Clustering

It was shown in Section 3.4 that simultaneous identification of the optimum word lengths of variables and resources, along with the optimum number of resources is likely to be an extremely difficult problem. In order to obtain a good solution that minimizes the implementation cost, we introduced a two–step optimization process. In this chapter, we describe the first optimization step, clustering, that yields a set of optimized cost functions corresponding to the tight upper–bound word lengths. As previously described in Section 3.3, the final word length selection is performed during the second optimization step, which is implemented using a genetic algorithm.

The organization of this chapter is as follows. In Section 5.1 we describe the word–length optimization problem and explain the scope of clustering. Motivation for clustering is given in Section 5.2. In Section 5.3 we introduce the clustering problem and present methods to solve it. In that section we present a dynamic programming solution to place operations in the algorithm into a specified number of clusters according to their word length compatibility. The next step in determining the optimized cost function involves prediction of the number of resources of different word lengths. Previous resource prediction methods are reviewed in Section 5.4. A new behavioral resource prediction method for multiple word length use is described in Section 5.5. Both nonpipelined and pipelined implementation styles are considered in the clustering process.

5.1 Background

In Section 3.3 we derived the comprehensive cost function given by Equation (3.38). The cost function can also be expressed in the form of Equation (3.21), reproduced here for convenience.

$$\textit{design area} = \sum_{t=1}^{z} \sum_{k=1}^{\psi^t} \nu_k^t \times f^t\left(\omega_k^t\right) \tag{5.1}$$

where

$$\begin{aligned} z &= \text{no. of operation types} \\ \psi^t &= \text{no. of distinct word lengths of resources of type } t \\ \omega_k^t &= \text{value of the } k\text{–th word length of type } t \\ \nu_k^t &= \text{no. of resources of word length } \omega_k^t \\ f^t(\omega) &= \text{function that gives the area of a resource of type } t \text{ and word length } \omega. \end{aligned}$$

A vector $\vec{\psi}$ is defined as

$$\vec{\psi} \triangleq [\psi^1, \psi^2, \ldots \psi^z] \tag{5.2}$$

There are $\prod_{t=1}^{z} \langle \max . \psi^t \rangle$ unique combinations of $\vec{\psi}$ where $\langle \max . \psi^t \rangle$ is the maximum number of distinct word lengths of resources of type t that can be used in the implementation. Its value is determined by error analysis, as shown in Section 4.4, or it may be specified by the designer. Corresponding to each $\vec{\psi}$, a cost function exists in the form of Equation (5.1), thus forming a set of cost functions of cardinality $\prod_{t=1}^{z} \langle \max . \psi^t \rangle$. Each of these cost functions can be optimized by appropriately selecting the number of resources ν_k^t, and the values of resource word lengths ω_k^t for all $k \in [1, \psi^t]$, for each operation type t. The number of resources of each selected word length of each type depends on (a) ψ^t, (b) ω_k^t (c) the number of operations of type t, n^t, (d) the optimal word lengths of operations, and (e) the performance requirement. Note that the optimal word lengths of operations must also be obtained in order to optimize the cost functions.

In clustering we assume the requirement of maximum achievable accuracy introduced in Section 4.4. Then the necessary operation word lengths are the tight upper–bound word

lengths. We also consider all possible combinations of $\vec{\psi}$ corresponding to the types of functional/arithmetic operations. In a typical algorithm only a few arithmetic functions are used such as addition, square root, and multiplication. Therefore the size of the cost function set is not too large. Then, for a given ψ^t we must select ω_k^t and ν_k^t such that the corresponding cost function is minimized. Since all $\vec{\psi}$ are considered, we obtain a set of optimized cost functions. In Theorem 5.2.1 we show that the choice of ω_k^t in clustering is limited only to the tight upper–bound word lengths.

5.2 Motivation

5.2.1 Clusters of operations

Since all combinations of $\vec{\psi}$ are considered, we analyze the effect of all ψ^t on the corresponding cost functions. Let the number of operations of type t be n^t. Since ψ^t distinct resource word lengths are used there are at least ψ^t resources. In general, more than one resource of a particular word length may be used; therefore, the number of resources $\nu^t \geq \psi^t$. Each resource must execute at least one operation. Therefore, n^t operations are partitioned into ψ^t non–empty sets or *clusters*. At least one or more resources are assigned to each cluster such that the word lengths of resources assigned to the same cluster are identical, and the word lengths of resources assigned to different clusters are distinct. Thus there are exactly ψ^t word lengths of type t. Let the number of operations in the k–th cluster be n_k^t, and the number of resources assigned to the k–th cluster be ν_k^t where $k \in [1, \psi^t]$. All n_k^t operations in the cluster are implemented by ν_k^t resources assigned to the cluster.

The following theorem shows that members of resource word length set Ω may be selected from the set of operation word lengths $\mathcal{W} \equiv \{ \mathrm{w}_1, \mathrm{w}_2, \cdots \mathrm{w}_{n^t} \}$.

THEOREM 5.2.1 *The set of resource word lengths Ω is contained in the set of operation word lengths $\mathcal{W}$.*

Proof Consider an element $\omega_i \in \Omega$ such t hat $\omega_i \ni \mathcal{W}$. Let w_p be the largest element of $\mathcal{W}$ smaller than ω_i, such that the corresponding operation o_p is implemented by a resource of word length ω_i. Therefore, all operations implemented by resources of word length ω_i, require word lengths less than or equal to w_p. Furthermore, no operation requiring word

length greater than w_p is implemented by any resource of word length ω_i. Since operation o_p requires w_p bits, the word length w_p is *necessary*. Also, $(\omega_i - \mathrm{w}_p)$ bits of all resources of word length ω_i are always unused because operations requiring word lengths greater than w_p are not implemented by these resources. Thus, the word length w_p is *sufficient*. This observation is valid for any i in the range $[1, \psi^t]$, and for any type t. Therefore ω_i can be replaced with w_p, and $\Omega \subseteq \mathcal{W}$. □

From this theorem we observe that the word lengths of resources assigned to a cluster must be equal to the word length of at least one operation in the cluster. Since an operation of a larger word length cannot be directly executed by a resource of smaller word length, we define *cluster word length* as follows:

Definition 5.2.1 The *cluster word length* is the word length of resource(s) assigned to the cluster, which is equal to the maximum of the word lengths of operations in the cluster.

Let the cluster word length of the k–th cluster be ω_k^t. An example of 3–clustering is shown in Figure 5.1. The dashed lines indicate the scope of each resource. For example, resource of word length ω_3 can implement all operations.

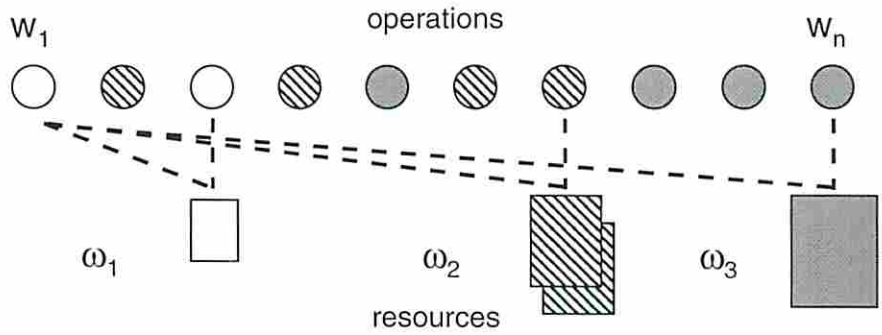


Figure 5.1: Unconstrained formation of clusters

5.2.2 Complexity of cost function minimization

The cost function corresponding to a given $\vec{\psi}$ is minimized by solving the problem

$$\text{Select} \quad \omega_1^t, \omega_2^t, \cdots \omega_{\psi^t}^t \quad \forall t = 1, \ldots z \quad \text{such that}$$

$$\sum_{t=1}^{z} \sum_{k=1}^{\psi^t} \widetilde{\nu_k^t} \times f^t(\omega_k^t) \quad \text{is minimum} \tag{5.3}$$

where $\widetilde{\nu_k^t}$ is the optimum number of resources assigned to the k–th cluster, and $f^t(\omega)$ is the area of a resource of type t and word length ω.

Given n^t and ψ^t, there are exponentially many ways to partition n^t operations into ψ^t non–empty clusters. Each way must be explored, and clustering that results in the minimum cost is the optimal clustering. For every partitioning $\widetilde{\nu_k^t}$ must be determined repeatedly as it depends on the operations in the k–th cluster. Since resources assigned to the k–th cluster exclusively implement operations in the k–th cluster, we can assume that there are $\sum_{t=1}^{z} \psi^t$ different types of operations. Then each $\widetilde{\nu_k^t}$ can be determined by solving the problem of minimum resource allocation under performance constraints. This problem is believed to be intractable [11]. Moreover, there are $\sum_{t=1}^{z} \psi^t$ distinct word lengths for each instance of partitioning that can be varied to minimize the cost function. Finally, this procedure must be iterated over all $\vec{\psi}$ in order to determine the minimum cost function. The corresponding $\vec{\psi}$ represents the optimal ψ^t for each type t, and the optimal clustering. The word lengths associated with the minimum cost function are the best operation and resource word lengths. Determining the minimum cost function is believed to be computationally highly intensive. *The objective of clustering is to quickly estimate clusters and their resource requirements that are most likely to minimize the design area.* In the following sections we describe clustering and resource prediction.

5.3 Word–length compatible clustering

In high–level synthesis, the design area is minimized by maximizing the resource utilization. This is achieved by assigning sequential operations of the same type to a common resource. Such sequential operations are detected by analyzing the *scheduling compatibility* of operations [11]. Likewise, to make the best use of resources of multiple word lengths, we introduce word–length compatibility.

Definition 5.3.1 *Word–length compatibility* between two operations is defined as the inverse of the Hamming distance between the word length vectors of the two operations.

In other words, if the Hamming distance between the word length vectors of two operations is small, they posses strong word length compatibility. We cluster operations by exploring their word–length compatibility only. Given a CDFG or sequencing graph we only know scheduling compatibility of operations. From this, a conflict graph can be constructed

which indicates potentially but not necessarily concurrent operations. By taking advantage of the scheduling compatibility at a later stage we minimize the resource requirement. Recall from Section 5.1 that in clustering we assume the requirement of maximum achievable accuracy in the algorithm. Therefore the operation word lengths considered in determining word–length compatibility are the tight upper–bound word lengths. Operations of a function type are ordered according to their tight upper–bound word lengths and then clustered in that order. Each cluster is then defined by its characteristic word length.

Definition 5.3.2 The *characteristic word length* of a cluster is the maximum of the tight upper–bound word lengths of operations in the cluster.

From Definition 5.2.1 the initial word length of one or more resources assigned to a cluster is the same as the characteristic word length of the cluster. Without knowledge of the scheduling compatibility, all operations in a cluster are implemented by resources assigned to the cluster. This ensures that an operation in the k–th cluster of word length w_j is most likely implemented by a resource of word length ω_k. Since clusters are ordered, the word length w_j is most upward compatible to word length ω_k. Only upward compatible resource word lengths are considered because resources of a smaller word length cannot directly implement an operation of larger word length. We now conclude that only the operations that require large word lengths are initially assigned to resources of large word lengths. Thus, in the implementation we have large word length resources that are essential. This leads to minimization of the design area, especially when the area–word length relation $f^t(\omega)$ is superlinear.

The likelihood of this clustering solution being as close as possible to the optimal clustering solution described in Section 5.2.2 is increased by following a two–step process. First, we must select characteristic word lengths in such a way that the overall cost is minimized. Second, if one or more resources assigned to a cluster of larger characteristic word length are idle during the execution of the algorithm, they must be used to implement operations from other smaller characteristic word length clusters, in order to decrease the resource requirements in those clusters. This directive is seemingly contradictory to the strongest word–length compatible resource assignment stated above. However, in such an assignment the number of large word length resources is not increased. Instead their utilization is increased and in turn the number of resources of some smaller word length is

reduced. This inter–cluster resource sharing exploits the scheduling compatibility of operations that was previously ignored, and is described in Section 5.5.3. Behavioral prediction of resources assigned to a cluster ignoring inter–cluster resource sharing, described in Section 5.5, explores the scheduling compatibility of operations within a cluster. We now describe the selection of characteristic word lengths from an ordered set.

5.3.1 Selection of characteristic word lengths

The task of partitioning an ordered set of n^t elements into ψ^t non–empty subsets is equivalent to selecting $(\psi^t - 1)$ places of cutting the set. There are $(n^t - 1)$ ways to select the first cut position, $(n^t - 2)$ ways to select the second and finally, $(n^t - \psi^t + 1)$ ways to select the $(\psi^t - 1)$–th cut position. However, the clusters formed are identical for a given choice of $(\psi^t - 1)$ cut places, regardless of the order of their selection. Thus, the total number of ways of forming ordered clusters for a given operation type t is given by

$$X^t = \frac{(n^t - 1)!}{(\psi^t - 1)! \cdot (n^t - \psi^t)!} \tag{5.4}$$

There are $\prod_{t=1}^{z} X^t$ total ways of selecting the characteristic word lengths of all function type for a given $\vec{\psi}$. This is a large number. Corresponding to each way, we form a cost function in the form of Equation (5.3). Since the strongest word length compatibility requirement is enforced in operation to resource assignment, n_k^t operations in the k–th cluster do not share resources from any other cluster. Then minimizing the cost function is equivalent to determining minimum resource allocation under performance constraints. In this problem the number of operation types is $\sum_{t=1}^{z} \psi^t$. This problem is believed to be intractable, but an ILP formulation is available [11]. Note that all $\prod_{t=1}^{z} X^t$ cost functions must be minimized. From these, the one that yields the minimum cost corresponds to the best way of selecting the characteristic word lengths or clusters.

5.3.1.1 Ordered clustering using normalized estimated resource cost

The ordered clustering problem described above is difficult because $\widetilde{\nu_k^t}$ must be determined. This problem is further simplified by replacing $\widetilde{\nu_k^t}$ with lower bounds of ν_k^t. The number

of operations in the k–th cluster of type t is denoted n_k^t. For single–cycle architectures, the lower bound of ν_k^t for non–pipelined as well as pipelined designs is obtained [26, 45]:

$$\langle \text{ lower bound} \cdot \nu_k^t \rangle = \lceil \frac{n_k^t}{\tau} \rceil \tag{5.5}$$

where τ is a performance requirement expressed in terms of the number of major cycles. For nonpipelined implementations τ represents the execution delay, and for pipeline implementations it represents the initiation interval (latency). The objective function in Equation (5.3) is re–written as

$$\text{minimize} \quad \sum_{t=1}^{z} \sum_{k=1}^{\psi^t} \lceil \frac{n_k^t}{\tau} \rceil \times f^t(\omega_k^t) \tag{5.6}$$

Since τ is a constant for a given target design, the objective function stated above is equivalent to the objective function

$$\sum_{t=1}^{z} \left(\text{minimize} \sum_{k=1}^{\psi^t} n_k^t \times f^t(\omega_k^t) \right) \tag{5.7}$$

The term $n_k^t \times f^t(\omega_k^t)$ represents the normalized estimated cost of resources allocated to the k–th cluster of function type t. Clustering is independent of performance constraints and function types for the given $\vec{\psi}$. However, the best $\vec{\psi}$, i.e. the best set of numbers of distinct word lengths to be used by each function type, depends on the required performance. Therefore, all possibilities of $\vec{\psi}$ are evaluated while predicting the optimized design. In the ordered clustering problem, n_k^t are determined by cluster boundaries only, which are determined by the selected characteristic word lengths. *The clustering problem is now reduced to the problem of selecting characteristic word lengths that minimize the normalized estimated functional–resource area.* There are still exponentially many ways to form ordered clusters but a dynamic programming solution to identify the clustering that satisfies Equation (5.7) exists, and is described next.

5.3.1.2 Cluster formation using dynamic programming

In the following discussion, we avoid the use of superscript t representing the function type, for simplicity of notation. We first show that the number of candidate clusters that can be used in ordered cluster formation is $O(n^2)$.

The normalized estimated cost of a cluster comprising ordered operations o_p through o_q is denoted $C_{p..q}$, and is given by

$$C_{p..q} = n_{p..q} \cdot f(\mathbf{w}_q) \tag{5.8}$$

where $n_{p..q}$ is the number of operations in the cluster and the characteristic word length of the cluster $\omega = \mathbf{w}_q$ by Theorem 5.2.1. The first and the last clusters are anchored because the first cluster must always start with the operation requiring the smallest word length, i.e. o_1. Similarly, o_n, the operation requiring the largest word length must be the last operation in ψ–th cluster. In other words, one boundary of these two clusters is always fixed while the other is flexible. Both boundaries of the other $(\psi - 2)$ clusters are flexible. For the first cluster, the second boundary may lie between operations o_1 and o_2, or o_2 and o_3, and so on but, may not exceed $o_{n-\psi+1}$. This is because, $(\psi - 1)$ non–empty clusters must be formed after the first cluster, which requires at least $(\psi - 1)$ elements. Extending this analysis to the second cluster, its first boundary coincides with the second boundary of the first cluster. Then, the second boundary may lie between operations o_2 and o_3, and so on but, may not exceed $o_{n-\psi+2}$. This is because $(\psi - 2)$ non–empty clusters must be formed after the second cluster. Figure 5.2 shows different formations of clusters 1, 2, and ψ.

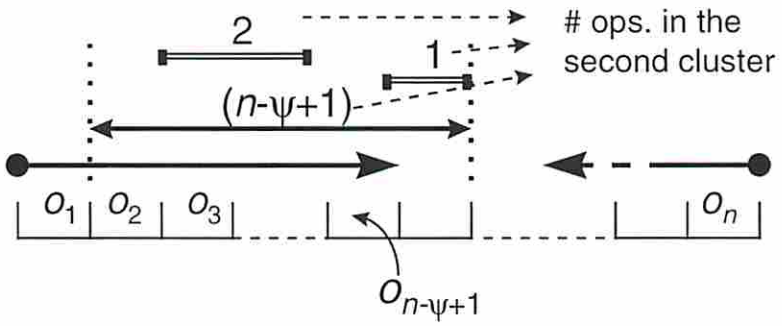


Figure 5.2: Boundaries of clusters

Generalizing this observation for the k–th unanchored cluster, we state that its first boundary may lie at least between o_{k-1} and o_k, and its second boundary may not exceed

$o_{n-\psi+k}$. Starting from o_k, there are $(n-\psi+1)$ distinct k–th clusters. These are listed as $(o_k..o_k)$, $(o_k..o_{k+1})$, and so on up to $(o_k..o_{k+n-\psi})$. Starting from o_{k+1}, there are $(n-\psi)$ distinct k–th clusters and finally, starting from $o_{n-\psi+k}$ there is only one k–th cluster. Thus, we have

$$\text{Total \# distinct } k\text{–th clusters} = \sum_{j=1}^{n-\psi+1} j = \frac{(n-\psi+1)(n-\psi+2)}{2} \tag{5.9}$$

Notice, that $(\psi-1)$ non–empty clusters must be formed before the last cluster, which require at least $(\psi-1)$ elements. Therefore, the first boundary of ψ–th cluster cannot lie before $o_{\psi-1}$. Thus, there are $(n-\psi+1)$ different first and last clusters. Now, the total number of candidate clusters that must be evaluated to select ψ clusters is given by

$$\begin{matrix}\text{Number of candidates for} \\ \psi \text{ non–empty clusters}\end{matrix} = 2\cdot(n-\psi+1) + (\psi-2)\frac{(n-\psi+1)(n-\psi+2)}{2} \tag{5.10}$$

This number is $O(n^2)$.

Optimal substructure property

Optimal substructure and overlapping subproblems are two important properties of problems that must be examined in order to apply a dynamic programming solution. A problem exhibits *optimal substructure* if an optimal solution to the problem contains within it optimal solutions to subproblems [10]. The optimal substructure of the lower bound ordered clustering problem is as follows. Assume that the optimal solution includes a cut position after o_k. Then, elements o_1 through o_k may be partitioned into 1,2, or up to $(\psi-1)$ clusters. Correspondingly, elements o_{k+1} through o_n may be partitioned into $(\psi-1)$, $(\psi-2)$, or down to 1 clusters. Let r be the optimum number of clusters formed in o_1 through o_k, and s be the optimum number of clusters formed in o_{k+1} through o_n, such that $(r+s)=\psi$. The original problem of partitioning n elements into ψ subsets is now divided as partitioning k elements into r subsets and $(n-k)$ elements into $(\psi-r)$ subsets. Greedy algorithms also take advantage of the optimal substructure property. However, in this problem the greedy

choice fails because the optimal o_k cannot be determined before optimal solutions to subproblems are obtained. In other words, the solution to the problem of forming ψ' clusters is not necessarily a part of the solution to ψ–clustering problem for an arbitrary $\psi' < \psi$.

Overlapping subproblems

Dynamic–programming solutions typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed. When a recursive algorithm revisits the same problem over and over again, the optimization problem is said to have *overlapping subproblems* [10]. For a given optimal cut position after o_k, r ranges from 1 through $(\psi - 1)$ in the worst case. The optimal value of r is not known before all possibilities are examined. However, observe from Equation (5.8) that the normalized estimated cost of a cluster depends only on its boundaries and is independent of other clusters. Therefore, the cost of a cluster is computed only once although it may be used in multiple formations. For example, two ways of clustering 20 nodes are shown below:

$$\begin{array}{lllll} (o_1..o_4), & (o_5..o_7), & (\mathbf{o_8..o_{11}}), & (o_{12}..o_{15}), & (o_{16}..o_{20}) \\ (o_1..o_3), & (o_4..o_7), & (\mathbf{o_8..o_{11}}), & (o_{12}..o_{17}), & (o_{18}..o_{20}) \end{array} \qquad \text{and,}$$

One way could be better than the other in terms of the normalized estimated cost but, $C_{8..11}$ is the same in either solution. From Equation (5.10) we know that the number of candidate clusters from which ψ clusters may be chosen is $O(n^2)$. Evaluation of Equation (5.8) requires a subtraction and computation of $f(\omega)$, which takes $O(1)$ time. Hence, costs of all candidate clusters can be evaluated in polynomial time, $O(n^2)$.

Dynamic programming solution to select clusters

We define a Cluster Graph ($\mathcal{CG}$) of depth ψ as follows: At depth 0, there is a source node. At each successive depth k, the nodes represent k–th cluster candidates. For example, at depth 3 the nodes represent candidate clusters $(o_3..o_3)$, $(o_3..o_4)$, and so on up to $(o_3..o_{3+n-\psi})$. Thus, the total number of nodes in $\mathcal{CG}$ equals the number of candidate clusters given by Equation (5.10) plus 1. Since clusters are ordered, and exactly ψ clusters are to be selected, *forward* edges are drawn between nodes at adjacent depths. Specifically, forward edges originate only from nodes at depth $(k - 1)$ and terminate at nodes at

depth k, for all k in the range $[1, \psi]$. There are no backward edges, edges between nodes at the same depth, or edges between nodes at non–adjacent depths. Additionally, any edge originating at a node that represents a cluster $(o_x..o_y)$ must terminate on nodes representing clusters starting from operation o_{y+1}. This property of $\mathcal{CG}$ ensures that boundaries of adjacent clusters coincide. The weight of an edge is the normalized estimated cost of its destination cluster. Here we take advantage of the overlapping subproblems property because a candidate cluster node may have one or more incident edges, each representing a different clustering however, the weight assigned to each incident edge is identical.

THEOREM 5.3.1 *The shortest path in the clustering graph represents the ordered clustering that minimizes the normalized estimated implementation cost*[1].

Proof Any path from the source node to any node at ψ–th depth has exactly ψ edges; therefore, each path in the $\mathcal{CG}$ represents the formation of exactly ψ clusters. Nodes along any of these paths represent the corresponding cluster boundaries expressed as the first and last operations in the cluster. The weight of the k–th edge in a path is the normalized estimated cost of the k–th cluster. Thus, the weight of a path equals the normalized estimated implementation cost corresponding to clustering represented by the path. Recall from Equation (5.7) that for a function type, the normalized estimated cost of implementation is the sum of normalized estimated costs of all clusters. Finally, the shortest path weight equals the minimum normalized estimated implementation cost, and the path yields the optimized ordered clustering solution. □

$\mathcal{CG}$ is acyclic, as there are no backward edges. Hence the DAG shortest paths algorithm can be used to determine the optimal clustering [10]. Examples of 4–way and 3–way cluster formation are shown in Figure 5.3. Although the DAG shortest paths algorithm employees greedy choice, the final clustering solution is obtained only after computing the shortest paths to all candidate clusters at the ψ–th depth. There are $(n - \psi + 1)$ such paths and the one with least weight is selected. For this reason, the solution to the clustering problem is not greedy.

[1] As stated earlier, for simplicity, our cost model for this step includes functional resource area only. Simple modification to the cost model could be made by including estimates of storage, switching and wiring, without any changes to the technique we describe here.

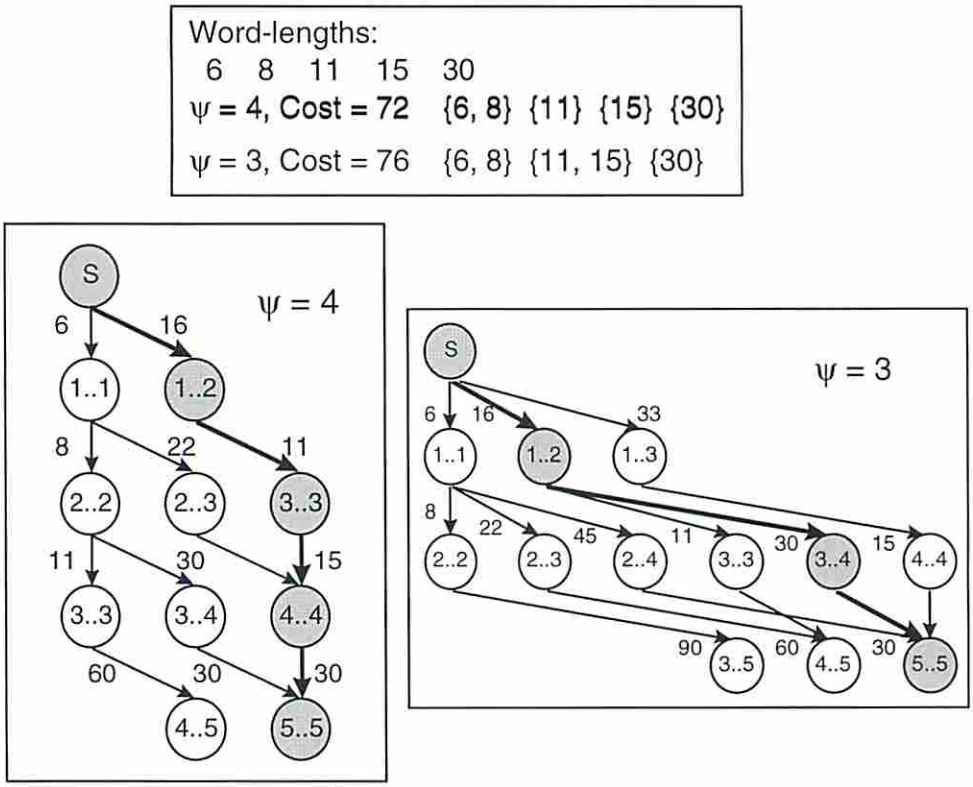


Figure 5.3: 3–way and 4–way cluster formation

5.3.1.3 Complexity analysis of clustering

In the previous section we have shown that the number of nodes in the clustering graph is $O(n^2)$. We now show that the number of edges in the $\mathcal{CG}$ is $O(n^2\psi)$. Since edges exist only between nodes at adjacent depths, we first determine the number of edges originating at depth k. We assume that candidate clusters at depths k and $(k+1)$ are unanchored. A generalized example is shown in Figure 5.4. Let the absolute cluster boundaries at depth k be o_p and o_q. Although the k–th cluster has flexible boundaries, absolute boundaries represent operations beyond which k–th cluster formation is not feasible, as described in the previous section. The absolute cluster boundaries at depth $(k+1)$ are o_{p+1} and o_{q+1}. There are $(n-\psi+1)$ nodes representing clusters starting from o_{p+1}. Therefore, there are $(n-\psi+1)$ edges originating from the node at depth k representing the cluster $(o_p..o_p)$. This is because, under the assumption of ordered clustering, if the k–th cluster ends with operation o_p, the $(k+1)$–th cluster must start from operation o_{p+1}. Similarly, there are $(n-\psi)$ nodes representing clusters starting from o_{p+2}. Therefore, there are $(n-\psi)$ edges originating from each node at depth k representing clusters ending with o_{p+1}. Since there

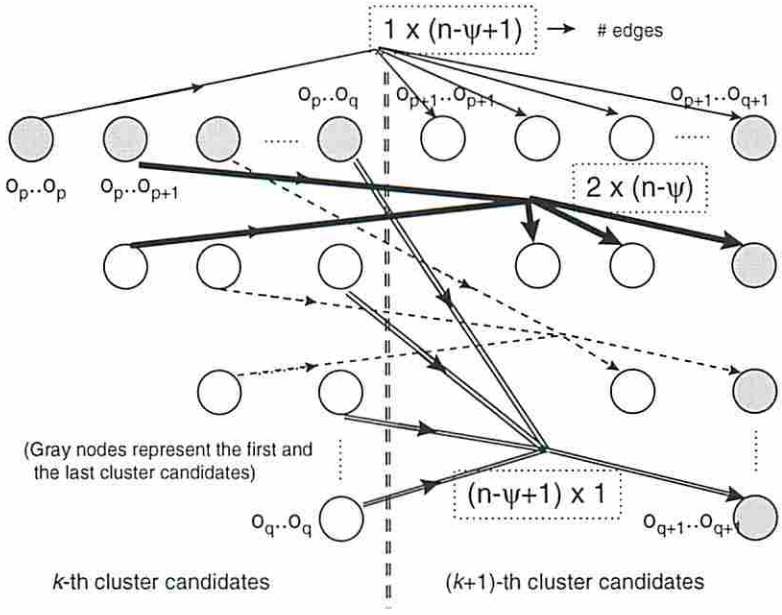


Figure 5.4: Edges in clustering graph

are 2 such nodes namely, $(o_p..o_{p+1})$ and $(o_{p+1}..o_{p+1})$ at depth k, the corresponding number of edges is $2 \cdot (n - \psi)$. Generalizing this observation we can write the number of edges originating at the k–th depth as

$$\begin{aligned} e_k &= 1 \cdot (n - \psi + 1) + 2 \cdot (n - \psi) + \cdots + (n - \psi + 1) \cdot 1 \\ &= \sum_{j=1}^{n-\psi+1} (n - \psi + 2 - j) \cdot j \end{aligned} \tag{5.11}$$

Observe from Equation (5.11) that the number of edges originating at depth k is $O(n^2)$. There are exactly $(n - \psi + 1)$ edges originating from the dummy source node, and $\frac{1}{2}(n-\psi+1)(-\psi+2)$ edges originating from anchored nodes at depths 1 and $(\psi-1)$. Since there are ψ depths in $\mathcal{CG}$, the asymptotic number of edges is $O(n^2\psi)$. It was shown elsewhere [10] that the run time complexity of the DAG shortest paths algorithm is $O(V + E)$, where V is the number of nodes in the DAG, and E is the number of edges. Thus, the time required to find the shortest paths from the source node to each node at ψ–th depth in $\mathcal{CG}$ is $O(n^2\psi)$. There are $(n - \psi + 1)$ such paths and the clustering solution is obtained by selecting the path of minimum weight. Therefore, the required run time of ordered clustering that minimizes the normalized estimated functional–resource cost is $O(n^3\psi)$.

The next step in clustering to optimize the cost function is to explore the scheduling compatibility that was ignored previously. For this, we predict the number of resources assigned to each cluster that minimize the cost function. In the next section we discuss some previous work in behavioral resource prediction. In Section 5.5 a novel method to predict resources of multiple word lengths is given.

5.4 Review of related prediction methods

Estimating the number of functional resources is critical in area/delay predictions. Mathematical models to estimate that number, and methods to determine its bounds were reported [6, 19, 22, 26, 44]. These techniques consider different types of resources, however, their word lengths are assumed fixed to the word length of the operation. One may use the existing techniques by listing multiple subtypes of resources of the same function type, each subtype corresponding to a different word length. For example, mult8, and mult12 for 8 and 12–bit multiplications respectively. Techniques described in these publications [6, 19, 22, 26, 44] assume that functions of a type are implemented only by the resources of that type. Under this assumption, the predictor will ignore the fact that an 8–bit multiplication may be implemented by any of 8, 12, 16, and 32–bit multipliers. Therefore, the lower bound on the number of resources given by these methods is not a tight bound. Our procedure determines the word–length compatibility of operations and also estimates timing conflicts. It is then able to share large resources between large and small operations. Another important feature of our procedure is that, unlike other methods, we bias small operations given the superlinear or quadratic nature of the cost function. In the implementation, using many small resources and only a few large ones of a function type may lead to significant reduction in area. Our resource estimation procedure can be biased to predict such designs. The third distinction is that other techniques predict a value or bounds of the number of resources. The corresponding estimates of area are also values or a distribution. This is feasible in their case because the area–delay characteristic of the library modules is fixed. In our predictions, these characteristics are functions of resource word lengths that are the unknowns to be optimized. Therefore, our procedure generates cost functions in terms of word lengths as opposed to values or a distribution of values.

5.5 Prediction of the number of resources required in a cluster

After the operations of a type are clustered, the next step is to predict the number of resources required in each cluster, so that a cost function can be computed. We consider both nonpipelined and pipelined implementations and present a two–step method to determine the lower bound on the number of resources required for a given execution delay or latency. In the first step described here, the minimum number of resources required in each cluster is determined by ignoring inter–cluster resource sharing.

Assumptions

We only consider algorithms that can be expressed as DAGs. For implementation, we assume a *single–cycle architecture* [26], although our techniques can be extended for other styles.

Definition 5.5.1 The *single–cycle architecture* model is a scheduling model in which all operations are assumed to be combinational and to take one time step to execute. The clock cycle time has to be longer than any operation delay to ensure the correct execution of the specification.

We also assume that the execution delay and latency constraints are specified in terms of the number of steps. The minimum number of steps required to execute the algorithm is obtained from *ASAP* scheduling [11]. All subsequent analysis in this chapter is assumed to be for a given function type t. Hence, for simplicity of notation, we omit the superscript t. Note that resource prediction must be performed for each type of operation in a given algorithm.

5.5.1 Nonpipelined implementation

Let the number of operations in the k–th cluster be n_k. Let the total number of steps permitted for the execution of $\mathcal{G}$ be T. If T^{ASAP} is the number of steps required by the *ASAP* schedule, then $T \geq T^{ASAP}$. Let T_k be the number of steps over which operations in the

k–th cluster are distributed. Note that operations in different clusters may be concurrent. Therefore,

$$\sum_{k=1}^{\psi} T_k \geq T^{ASAP} \tag{5.12}$$

Definition 5.5.2 The *utilization* of resources assigned to cluster k in non–pipelined design is defined as

$$u_k = \frac{n_k}{\nu_k \times T_k} \tag{5.13}$$

where ν_k is the number of resources assigned to the k–th cluster.

Since utilization of any resource can not exceed 100%, the lower bound on ν_k is written as

$$\nu_k \geq \lceil \frac{n_k}{T_k} \rceil \tag{5.14}$$

The task of predicting the minimum number of resources required in the k–th cluster is now transformed into determining T_k, the number of steps over which operations in the cluster are distributed. From the *ASAP* schedule of $\mathcal{G}$, one can determine the earliest time step in which at least one operation in a given cluster can begin. Similarly, from the *ALAP* schedule, one can determine the latest time step in which at least one operation in the cluster can execute. The difference between these latest and earliest times steps for the k–th cluster is the absolute maximum value of T_k. An example is shown in Figure 5.5 for the 3x3 matrix inverse algorithm. T_k represents the collective *slack* of operations in cluster k. From Equation (5.14) it is clear that for large T_k, the resource requirement is small. Let $[t_k^b, t_k^e]$ denote the interval during which operations in the k–th cluster are executed. Then T_k is given by

$$T_k = t_k^e - t_k^b + 1 \tag{5.15}$$

If T_k is maximized, some successor operations in cluster k' may be delayed. According to the *ALAP* schedule, the maximum t_k^e will be limited so that the maximum $t_{k'}^e$ does not change. However, the value of $t_{k'}^b$ may increase and the corresponding value of $T_{k'}$ will be reduced. Consequently, the resource requirement in cluster k' will increase. Consider the

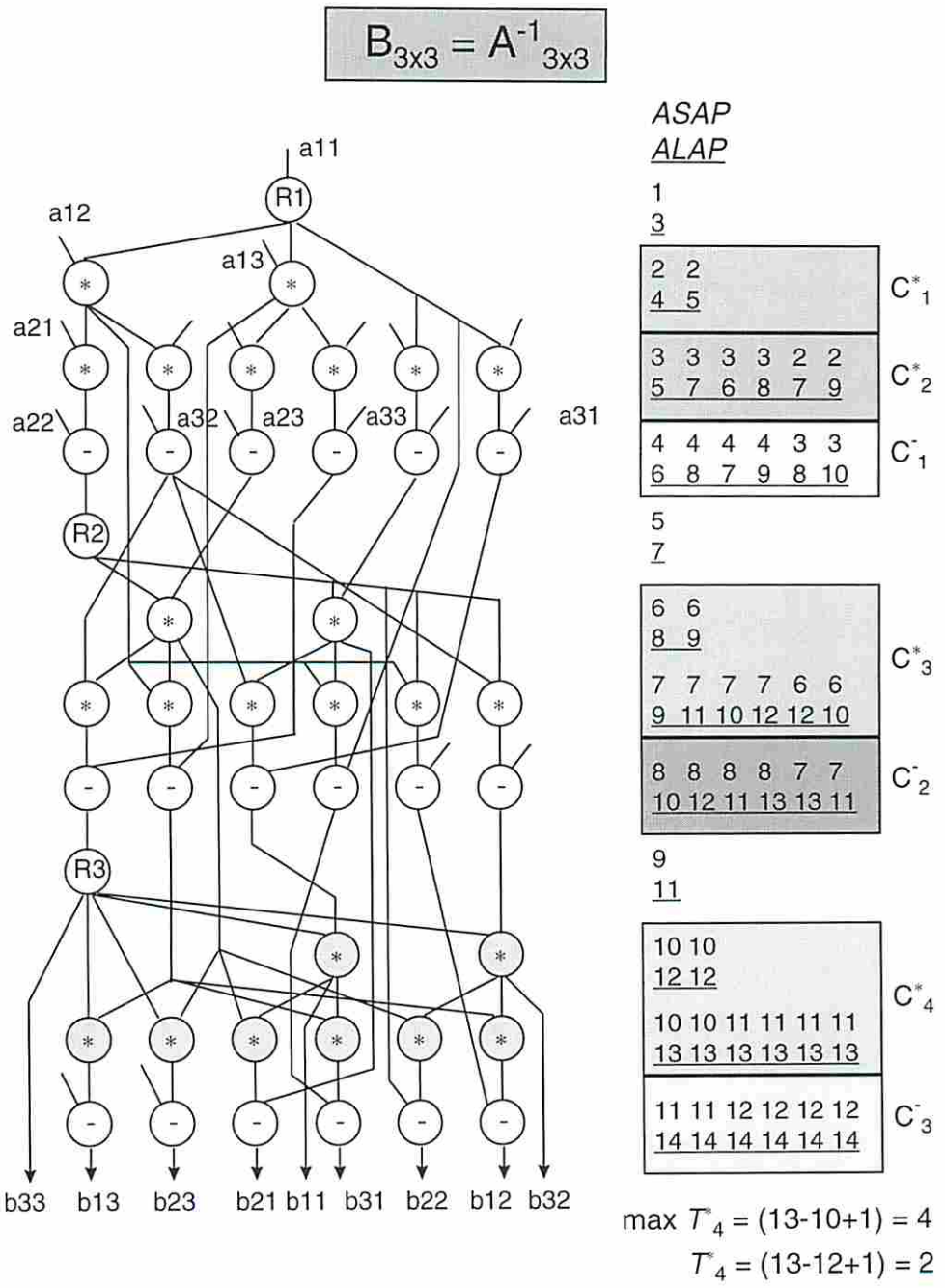


Figure 5.5: Changes in cluster execution intervals

3x3 matrix inverse example in Figure 5.5. If R3 is delayed until step 11, T_4^* is reduced from its absolute maximum value 4, to 2. Then, 4 multipliers are required as opposed to 2 for that cluster. Thus, the interval of execution for a cluster in the graph must be determined by considering the timing requirements of all other operations in the graph.

The average slack of a cluster is the average of individual slacks of all nodes in the cluster. Let $t_{k_i}^S$ and $t_{k_i}^L$ be the time steps of the i–th operation in the cluster, corresponding to *ASAP* and *ALAP* schedules. Then the average slack of cluster k is given by

$$\langle \text{avg} \cdot T_k \rangle = \frac{1}{n_k} \sum_{i=1}^{n_k} \left(t_{k_i}^L - t_{k_i}^S + 1 \right) \tag{5.16}$$

$$= \frac{1}{n_k} \sum_{i=1}^{n_k} t_{k_i}^L - \frac{1}{n_k} \sum_{i=1}^{n_k} t_{k_i}^S + 1 \tag{5.17}$$

For a single–cycle architecture, $t_{k_i}^L$, corresponding to the i–th operation in the k–th cluster, o_{k_i} is given by

$$t_{k_i}^L = T - \text{max. no. of successor nodes of } o_{k_i} \tag{5.18}$$

This is because each successor node requires one step. Taking the average over all nodes in the cluster, on both sides of Equation (5.18) we have

$$\frac{1}{n_k} \sum_{i=1}^{n_k} t_{k_i}^L = T - \frac{1}{n_k} \sum_{i=1}^{n_k} \text{max. no. of successor nodes of } o_{k_i}$$

$$= T - TC_k \tag{5.19}$$

Substituting Equation (5.19) in (5.17) we get

$$\langle \text{avg} \cdot T_k \rangle = T - TC_k - \frac{1}{n_k} \sum_{i=1}^{n_k} t_{k_i}^S + 1 \tag{5.20}$$

A large estimate of T_k is obtained by replacing the average earliest step in the cluster, given by the summation term in Equation (5.20), with the earliest step in which any operation in cluster k can start. The latter is given by

$$t_k^S = \min_i \{ t_{k_i}^S \} \qquad \forall i = 1 \ldots n_k \tag{5.21}$$

Similarly, a small estimate of T_k is obtained by replacing the average earliest step with the latest step in which at least one operation in cluster k must start, which is given by

$$t_k^L = \min_i \{t_{k_i}^L\} \qquad \forall\, i = 1 \ldots n_k \tag{5.22}$$

Now, we write large and small estimates of T_k as

$$\langle \text{large} \cdot T_k \rangle = T - TC_k - t_k^S + 1 \tag{5.23}$$

$$\langle \text{small} \cdot T_k \rangle = T - TC_k - t_k^L + 1 \tag{5.24}$$

The resource area is directly proportional to the number of resources and superlinearly proportional to resource word length, i.e. the characteristic word length of the cluster. Since we know the relative ordering of characteristic word lengths of clusters, we perform a biased allocation of resources to minimize area. Short execution intervals and more resources are expected to be used by clusters with small characteristic word lengths. Conversely, longer execution intervals and fewer resources are desirable for clusters with large characteristic word lengths. In Equations (5.23) and (5.24), we change the starting step of the interval in order to change its length. A biasing parameter $\beta_k \in [0, 1]$ is used to vary the starting step according to the characteristic word length of the cluster. Clusters with small characteristic word lengths have small β_k. Clusters with large characteristic word lengths have large β_k. Finally, the execution interval of a cluster can be written by using β_k to combine Equations (5.23) and (5.24) as

$$T_k = T - TC_k - [\beta_k \cdot t_k^S + (1 - \beta_k) \cdot t_k^L] + 1 \tag{5.25}$$

and a lower bound on the number of resources for the k–th cluster is given by

$$\nu_k = \lceil \frac{n_k}{T - TC_k - [\beta_k \cdot t_k^S + (1 - \beta_k) \cdot t_k^L] + 1} \rceil \tag{5.26}$$

5.5.2 Pipelined implementation

The prediction of lower bound on the number of resources needed in each cluster for pipelined designs is relatively straight forward. Unlike the nonpipelined case, the execution

interval of all clusters is the same as the initiation rate. Let L be the latency (i.e. initiation rate). From the definition of resource utilization in pipelined designs [26] we have

Definition 5.5.3 The *utilization* of resources assigned to cluster k in pipelined designs is defined as

$$u_k = \frac{n_k}{\nu_k \times L} \tag{5.27}$$

where ν_k is the number of resources assigned to the k–th cluster.

The lower bound of the the number of resources required in a pipelined design was given by Jain *et al.* [22]. Using their result we write the lower bound on ν_k as

$$\nu_k = \lceil \frac{n_k}{L} \rceil \tag{5.28}$$

5.5.3 Sharing resources between clusters

Resources assigned to a cluster of large characteristic word length may be used to implement operations in a cluster of smaller characteristic word length, if execution intervals of the operations sharing the resources do not overlap. Then for any cluster k' sharing resources from another cluster k, the number of resources required is reduced further and a tight lower bound is obtained as

$$\nu_{k'} = \max \{0, (\nu_{k'} - \nu_k)\} \tag{5.29}$$

This expression prohibits a negative number of resources. If new $\nu_{k'}$ equals 0, no resources of the corresponding word length are needed. In effect, the number of distinct word lengths to be used ψ, is reduced by 1. The resource sharing possibilities are explored pairwise among all clusters. If the exact number of resources were known, an ILP solution for optimal concurrent binding and scheduling could be used that satisfies the timing constraints and maximizes inter–cluster resource sharing. However, this is a difficult problem to solve and only estimates of the number of resources are available. Hence, the execution interval of each cluster is estimated using a heuristic similar to force–directed scheduling proposed by Paulin and Knight [46]. Using the predicted number of resources in each cluster as resource constraints, an *ASAP* schedule for the graph is obtained. Operations with the least

mobility and a large number of successor nodes are given scheduling priority. The first step during which an operation may be scheduled is determined by resource availability and operation sequencing in the graph. This constraint is loosely similar to the notion of *self–force* in force directed scheduling. Priority enforced by mobility and the number of successor nodes is analogous to the notion of *predecessor/successor force*. Once scheduling is performed, the cluster interval is obtained as

$$\begin{aligned} \text{Cluster interval } \mathrm{i}_k &= [t^{Pb}, t^{Pe}] \\ \text{where } t^{Pb} &= \min_j \{t_j^{Sp}\} \quad \forall\, j = 1 \cdots n_k \\ t^{Pb} &= \max_j \{t_j^{Sp}\} \quad \forall\, j = 1 \cdots n_k \\ t_j^{Sp} &= \text{Predicted } \textit{ASAP} \text{ schedule step of } o_j \in O_k \end{aligned} \tag{5.30}$$

A procedure PREDRESNUM to predict lower bounds on the number of resources required in each cluster for nonpipelined designs is given in Figure 5.6. Figure 5.7 describes a procedure SHARERES to share resources between clusters and procedure PREDEXECINT shown in Figure 5.8 determines the execution interval of a cluster.

5.5.4 Analysis of resource prediction procedures

The run time complexity of ASAP and ALAP procedures is $O(ne)$, where e is the number of edges in $\mathcal{G}$. The main loop in PREDRESNUM is iterated ψ times, and time taken by each step in the loop is $O(n_k)$. In the worst case, $n_k = n$. Therefore, total run time complexity of PREDRESNUM is $O(ne)$. The run time complexity of PREDEXECINT is also $O(ne)$. Finally in SHARERES, procedure PREDEXECINT is called ψ times. Pairwise identification of non–overlapping intervals takes $O(\psi^2)$ time. Assuming that the clusters are arranged in increasing order of their characteristic word length, the cascaded reverse loops in SHARERES prioritize the possibility of sharing resources between clusters of higher ranks. This enables reduction in the number of large word length resources before small ones. The overall run time complexity of SHARERES is $O(\psi\, ne)$.

```
PREDRESNUM(O, ψ, T)
ASAP(O)
ALAP(O, T)
    /* ASAP and ALAP scheduling to determine slack */
do k := 1 to ψ
        /* Estimate the average no. of steps needed after
           the operations in the k–th cluster have been performed */
   TC_k ← avg (t^L_{k_i}, n_k)
        /* The earliest step in which any operation in cluster k can start */
   t^S_k ← min (t^S_{k_i}, n_k)
        /* The latest step in which at least one operation in cluster k must start */
   t^L_k ← min (t^L_{k_i}, n_k)
        /* The estimate of the no. of steps used by the operations in cluster k */
   T_k ← T − TC_k − [β_k · t^S_k + (1 − β_k) · t^L_k] + 1
   ν_k ← ⌈n_k / T_k⌉
end
SHARERES(O, ψ)
```

Figure 5.6: A procedure to predict lower bounds on the number of required resources

```
SHARERES(O, ψ)
do k := 1 to ψ
   I_k ← PREDEXECINT(k)
        /* Determine the execution interval of the operations in cluster k */
end
ψ' ← ψ
     /* It is assumed that the clusters are sorted by their characteristic word lengths
        so that resources of large word lengths are shared by
        operations requiring smaller word lengths */
do k := ψ downto 1
   do l := ψ − 1 downto 1
           /* Consider a pair of candidate clusters. The most downward word–length
              compatible cluster is given priority in resource sharing */
      if I_k ⋂ I_l = ∅
           /* Share and update the number of resources between the two clusters */
        then ν_l ← max(0, (ν_l − ν_k))
                /* Reduce the number of resources required in the cluster of
                   smaller word length */
             I_k ← I_k ⋃ I_l
             I_l ← I_k
                /* The operations in the two clusters share resources.
                   Update the execution intervals of both clusters so as to
                   avoid a conflict if any of these two clusters shares
                   its resources with a different cluster */
             if ν_l = 0
               then ψ' ← ψ' − 1
                       /* No resources of a word length assigned to this cluster
                          are required. Hence, decrease the number of distinct
                          resource word lengths used */
             endif
      endif
   end
end
ψ ← ψ'          /* Update ψ if necessary */
```

Figure 5.7: A procedure to share resources between clusters

PREDEXECINT(k)
while $O_k \neq \emptyset$ **do**
    Select a set of vertices $\hat{O}_k$ whose predecessors are all scheduled
    Sort $\hat{O}_k$ by mobility
    Sort $\hat{O}_k$ by no. of successor nodes
    $o_j \leftarrow$ EXTRACTTOP $\hat{O}_k$
        /* Of the operations having all predecessor operations scheduled o_j has the least mobility and the maximum number of successor operations */
    $t_j^S \leftarrow \max_{\text{Edge from } o_i \text{ to } o_j \text{ in } \mathcal{G}} t_i^S + 1$ /* Schedule o_j */
    $O_k \leftarrow O_k - o_j$
end
$t^{Pb} \leftarrow$ FINDMIN(t_j^S)
    /* The estimated earliest step in which any operation in cluster k is scheduled */
$t^{Pe} \leftarrow$ FINDMAX(t_j^S)
    /* The estimated latest step in which any operation in cluster k is scheduled */
$I_k \leftarrow [t^{Pb}, t^{Pe}]$
return (I_k)

Figure 5.8: A procedure to determine the execution interval of a cluster

Chapter 6

Word–length selection using genetic algorithms

In Chapter 4, we discussed a systematic approach to determine the tight upper–bound word lengths of all the variables in an algorithm, given the error, and numerical range of each of its primary inputs. Recall from Section 4.4 that the use of these word lengths yields the maximum accuracy in the given environment. In the previous chapter we developed a method to generate a set of optimized cost functions, where each function in the set corresponds to a unique $\vec{\psi}$, the vector comprising the number of distinct resource word lengths of each operation type in the algorithm. These cost functions use the tight upper–bound word lengths, and thus guarantee the maximum accuracy. The optimization in clustering reduces the functional–resource area by taking advantage of the word–length compatibility of operations as well as the scheduling compatibility. We now minimize all the cost functions in the set further, by selecting smaller word lengths that satisfy the accuracy requirement, using a genetic algorithm. The function resulting in the minimal cost after word–length reduction determines the optimized number of distinct resource word lengths for each type, and the optimized word–length set. The chapter is organized as follows: The use of classical optimization techniques in word–length selection is discussed in Section 6.1. In this section, we also motivate the use of a genetic algorithm in our problem. An overview of genetic algorithms is given in Section 6.2. In Section 6.3, we develop a genetic algorithm for word–length selection. Finally, the overall word–length selection procedure using techniques described in this and the previous chapters is given in Section 6.4.

6.1 Word–length selection using classical optimization techniques

In Sections 3.1.3, and 3.3 we showed that the computational error constraint equation, and the cost function in terms of word lengths are both nonlinear. Therefore we consider classical nonlinear optimization techniques such as basic descent, gradient projection [37] and modified Newton's method [42]. These techniques have two drawbacks. First, they assume that the cost and constraint functions are continuous differentiable. It is however obvious that word lengths must be natural numbers. Hence, both cost and error–constraint equations are discrete. This problem can be solved by assuming word lengths to be real numbers. When the optimal solution is obtained after convergence, the real numbers can be rounded off to the next integer. The error constraint equation violates this assumption in yet another way, because truncation error does not always change with the word length. For example, when the word length allocated to the result of a computation is greater than the maximum number of bits generated by that computation, the truncation error is always 0. The error constraint equation can be made continuous differentiable by using *slack variables* described by Mullins *et al.* [42]. The second problem is direction control. In the gradient projection method this problem is solved by introducing a *tolerance factor*, and throughout the procedure the constraints are satisfied only to within the tolerance factor [37]. Solutions generated by this method may sometimes be unacceptable. For example, if the designer–specified constraint requires all word lengths to be less than or equal to 120 bits, then a solution involving a 121–bit adder is not acceptable. Modified Newton's method also has the potential danger of losing the direction of descent, especially when the initial solution is far from the optimal solution [37,58]. Randomized search techniques have also been used in nonlinear optimizations [48]. However, we know that computation error monotonically decreases, while implementation cost monotonically increases with word lengths. Therefore, we use genetic algorithms for word–length selection, where randomized search is efficiently enhanced by taking advantage of the monotonic properties of the computation–error and implementation–cost equations.

6.2 Overview of genetic algorithms

Genetic algorithms (GA) are characterized by the natural process of evolution [15, 40]. They start with the initial population of solutions represented as *chromosomes*. A chromosome comprises *genes* where each gene represents a specific attribute of the solution. The *fitness* of a chromosome is a measure of the quality of the solution it represents, in terms of various optimization parameters of the solution. A more fit chromosome suggests a better solution. In GA, the population is evolved: the relatively fit solutions reproduce, while the relatively inferior solutions die [40]. To select better solutions, a suitable fitness function is used to evaluate the fitness of each solution in the population. Then the solutions are ranked by their fitness. The evolution process continues until a solution with desirable fitness (quality) is found. This mechanism does not guarantee the optimal solution. However, carefully designed fitness functions can yield good, feasible solutions quickly. The evolution process involves two operations namely, *mutation* and *crossover*. Mutation arbitrarily alters one or more genes of a randomly selected chromosome [40]. The intuition behind the mutation operation is to improve the fitness of existing solutions in the population. While the goal of GA is to maximize the fitness of a solution, in real–life situations it is often necessary for the solution to meet certain other requirements. In other words, the maximization of fitness is constrained. A mutation may produce a chromosome representing a solution that violates one or more requirements. In this event, that mutation must be discarded. Crossover combines features of two chromosomes to form two similar offsprings by swapping genes of the parent chromosomes. The intuition behind the crossover operation is to exchange information between different potential solutions [40]. We now describe how the elements of the genetic optimization technique are applied to the word–length selection problem.

6.3 Genetic algorithm for word–length selection

6.3.1 Chromosome representation

Corresponding to each type of operation (such as addition, subtraction, and multiplication) in the given algorithm, the final solution of the word–length selection problem includes

1. the number of distinct word lengths (ψ^t) used by resources of type t,

2. the resource word lengths,

3. the number of resources of each word length,

4. the function $f^t(\omega)$ that gives the area of a resource of type t and word length ω, and

5. the operation word lengths.

A chromosome (shown in Figure 6.1), represents the final solution. The chromosome consists of one gene corresponding to each operation type in the algorithm. Each gene holds a

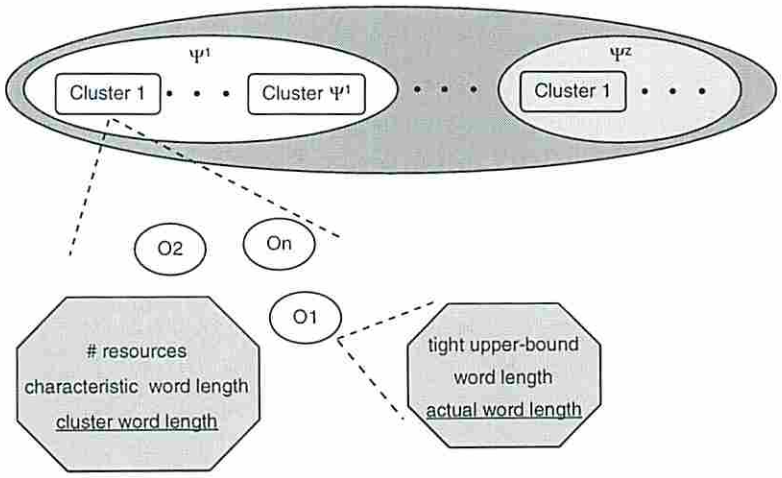


Figure 6.1: Chromosome representation for word–length selection

value, or a set of values corresponding to the solution attributes listed above. For the given algorithm, the required performance τ, and a specific value of ψ^t, the number of resources of each word length is fixed. However, it directly affects the cost (area) of the implementation, hence is a part of the gene structure. The underlined members in the genes are changed to decrease the cost of an implementation represented by a particular chromosome.

6.3.2 Fitness of a chromosome

GA attempts to maximize the fitness of the final solution. The objective of word–length selection is to minimize the implementation area. Hence, we define fitness as

$$\textit{fitness} \triangleq -1 \times \textit{implementation area} \tag{6.1}$$

$$= -\sum_{t=1}^{z} \sum_{k=1}^{\psi^t} \nu_k^t \times f^t\left(\omega_k^t\right) \tag{6.2}$$

where

$$
\begin{aligned}
z &= \text{no. of operation types} \\
\psi^t &= \text{no. of distinct word lengths of resources of type } t \\
\omega_k^t &= k\text{–th word length of type } t \\
\nu_k^t &= \text{no. of resources of word length } \omega_k^t, \text{ and} \\
f^t(\omega) &= \text{function that gives the area of a resource of type } t \text{ and word length } \omega.
\end{aligned}
$$

If the operation type t in the above equation is limited only to functional or arithmetic operations, the implementation area represents only the functional–resource area given by Equation (3.21) on page 25. However, t may incorporate other types of resources such as registers, multiplexers, drivers, input/output pads, and even the interconnect, so as to represent the overall circuit area. We reported a system level energy/power prediction technique [62] where the energy/power consumption is characterized by the size of resources. Since the size of a resource is proportional to its word length, the cost function given above can be readily modeled to represent energy/power consumption using our formula [62], as described in Appendix A. Thus a low power/low energy circuit can be obtained by selecting the appropriate word lengths. The fitness function used in the GA is the negative cost function given by Equation (6.2). Thus the most fit chromosome represents the minimized–area solution. For the scope of this thesis we limit z to represent the functional resources only.

6.3.3 Initial population

The maximum number of distinct resource word lengths for each function type, previously denoted $\langle \max . \psi^t \rangle$, is either specified by the designer, or obtained using our error analysis tool described in Section 4.5. The vector $\vec{\psi}$ comprising ψ^t, the number of distinct resource word lengths for each operation type t, was defined in Equation (5.2) on page 56. We reproduce the definition for convenience:

$$\vec{\psi} \triangleq [\psi^1, \psi^2, \ldots \psi^z] \tag{6.3}$$

where z is the number of operation types. A feasible solution exists corresponding to each $\vec{\psi}$ obtained by varying ψ^t in the range $[1, \langle \max.\psi^t \rangle]$, for all types. There are $\prod_{t=1}^{z} \langle \max.\psi^t \rangle$ combinations of $\vec{\psi}$, and the solution corresponding to each combination is a member of the initial population. The optimal solution is obtained by selecting the best number of distinct resource word lengths for each function type, and the corresponding optimal $\vec{\psi}$ is contained in the initial population.

6.3.4 Mutation

In the mutation operation we select a chromosome at random, and select one of its genes for mutation at random. In the selected gene there are ψ clusters. We select one cluster at random, and change the cluster word length[1]. The cluster word length is either increased or decreased by a random choice governed by two probabilities namely, the probability of increasing cluster word lengths denoted $\mathcal{P}^+(g)$, and the probability of decreasing cluster word lengths denoted $\mathcal{P}^-(g)$. The parameter g is the generation index. Initially, $\mathcal{P}^+(g)$ is very low and $\mathcal{P}^-(g)$ is high. As shortly explained, for later generations, $\mathcal{P}^+(g)$ is increased and $\mathcal{P}^-(g)$ is reduced accordingly. For $g = 0$, i.e. the initial population $\mathcal{P}^+(0) = 0$, and $\mathcal{P}^-(0) = 1$. Since the initial population comprises solutions using the tight upper–bound word lengths, increasing the cluster word length cannot improve the accuracy of the final result [61]. On the other hand, reducing the cluster word length implies reduction in the area of resources associated with the selected cluster, and in turn reduction in the overall circuit area. Hence, $\mathcal{P}^-(g)$ for small g is high. For the selected cluster we change the cluster word length by a small percentage, and consider the following two possibilities:

Case 1: cluster word length is reduced

By the definition of cluster word length, the word length of at least one operation (say o_x) must be equal to the cluster word length, and all other operations in the cluster must have word lengths smaller than or equal to the cluster word length. After reducing the cluster word length, we reduce if necessary, the word lengths of operations in the cluster to meet this requirement. The word length of o_x will always be reduced. As a result, the least significant bits of some operands may be truncated. This may introduce truncation error in those operands, which will

[1] Definition 5.2.1 on page 58.

propagate through subsequent computations. Correspondingly the worst–case error in the final result may increase.

Case 2: cluster word length is increased

(a) *The new cluster word length is greater than its characteristic word length*[2]: From the definition of characteristic word length, the new cluster word length is greater than the tight upper–bound word lengths of all operations in the cluster. Therefore, implementing any operation in the cluster using the new cluster word length does not improve the accuracy of the final result [61]. On the other hand, the area of resources in this cluster will increase, thereby increasing the overall circuit cost. Hence, this mutation is discarded.

(b) *The new cluster word length is smaller than or equal to its characteristic word length*:

The word lengths of some operations in the cluster that were reduced in the previous generations may be increased. This may reduce the worst–case error in the final result. In this step, the word–length of any operation is not increased beyond its tight upper–bound word length, as it does not improve the accuracy of the final result. By the definition of characteristic word length, the tight–upper bound word length of at least one operation in the cluster (say o_y) is greater than or equal to the new cluster word length. This is because the new cluster word length is smaller than or equal to the characteristic word length. Therefore, the word length of o_y is always increased.

Now the mutation of the gene is complete and a mutated chromosome representing a new solution is created in the population. The word length of at least one operation o_x or o_y, is is different from its previous value. Hence, the worst–case error of the final result may have changed. We obtain its value from Equation (3.13) on page 21. If it is less than or equal to the maximum permissible computation error specified by the designer, the error constraint equation (3.14) is satisfied, and the mutation represents a valid solution. Otherwise, the error is too large and the solution is invalid. However, an offspring of this chromosome from a later generation may represent a valid solution that satisfies Equation (3.14). This possibility exists when

[2]Definition 5.3.2 on page 60.

(a) The selected cluster in this gene in this chromosome is selected again. This time the cluster word length is increased, thereby reducing the error in the final result

(b) A different cluster in this gene in the same chromosome is selected, and its word length is increased. Thus the word lengths of one or more operations of the same type, other than those in this cluster are increased. This causes the worst–case error in the final result to reduce, or

(c) A cluster in a different gene in this chromosome is selected, and its word length is increased. Thus the word lengths of one or more operations of a different type are increased. This causes the worst–case error in the final result to reduce.

Situations (b) and (c) exploit the trade off between the relative precision of operations of the same or different types in a sequence of computation. This also explains why chromosomes representing invalid solutions are not discarded immediately. Furthermore, the situations described above explain the advantage of increasing the probability of increasing cluster word lengths $\mathcal{P}^{+}(g)$ in later generations.

6.3.5 Crossover

In crossover, two similar chromosomes are formed by swapping sets of genes of two parent chromosomes, hoping that at least one child will have genes that improve its fitness. In the word–length selection problem, crossover diversifies the population by swapping ψ of several operation types from a pair of parent chromosomes as shown in Figure 6.2. The number next to the function type is the number of distinct resource word lengths of that type. Consider two partitions of $\vec{\psi}$ given by Equation (6.3), namely, $\vec{\psi_A}$ and $\vec{\psi_B}$. Without the loss of generality assume that $\vec{\psi_A}$ and $\vec{\psi_B}$ are given by

$$\vec{\psi_A} = [\psi_1, \ldots, \psi_m] \tag{6.4}$$

$$\vec{\psi_B} = [\psi_{m+1}, \ldots, \psi_z] \qquad \text{for some constant } m \in [2, (z-1)] \tag{6.5}$$

Recall that each chromosome consists of a gene of each operation type 1 through z, and in a gene of type t the number of distinct resource word lengths is given by ψ^t. Consider two chromosomes x and y where $\vec{\psi_A}$ in the first chromosome is denoted $\langle x.\vec{\psi_A}\rangle$, and in the second $\langle y.\vec{\psi_A}\rangle$. Similarly, $\vec{\psi_B}$ in the two chromosomes are denoted $\langle x.\vec{\psi_B}\rangle$ and $\langle y.\vec{\psi_B}\rangle$

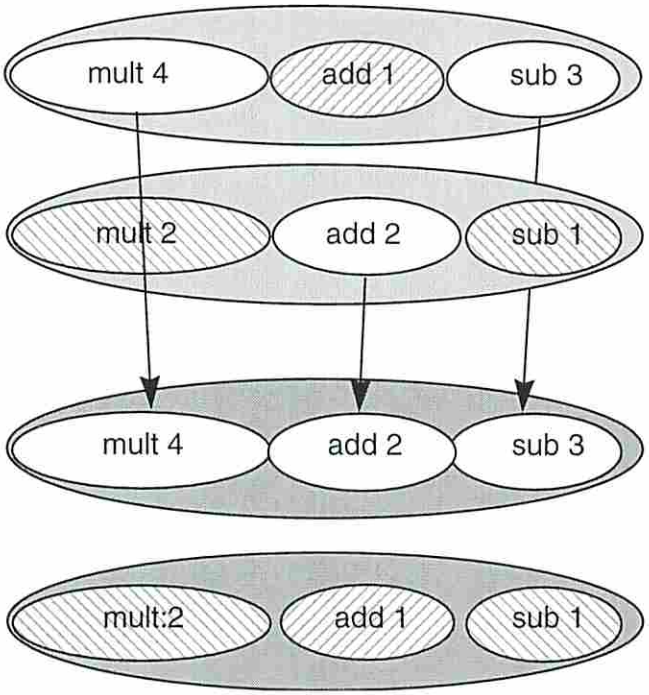


Figure 6.2: An example of crossover operation

respectively. Assume that $\langle x.\vec{\psi_A}\rangle$ and $\langle y.\vec{\psi_A}\rangle$ are distinct, and similarly $\langle x.\vec{\psi_B}\rangle$ and $\langle y.\vec{\psi_B}\rangle$ are distinct. When the two sets of genes corresponding to operation types 1 through m and $(m+1)$ through z are swapped two new chromosomes denoted u and v are formed. The vectors $\vec{\psi}$ corresponding to these chromosomes are given by

$$\langle u.\vec{\psi}\rangle = [\langle x.\vec{\psi_A}\rangle, \langle y.\vec{\psi_B}\rangle] \tag{6.6}$$

$$\langle v.\vec{\psi}\rangle = [\langle y.\vec{\psi_A}\rangle, \langle x.\vec{\psi_B}\rangle] \tag{6.7}$$

Thus each new solution gets the number of distinct resource word lengths for some operation types from the first chromosome. The number of distinct resource word lengths for the remaining operation types is swapped from the other parent solution. Assume that solution x contains the optimal value of ψ^p for some $p \in [1, m]$, and solution y contains the optimal value of ψ^q for some $q \in [(m+1), z]$. Then, solution u contains the optimal values of the number of distinct resource word lengths for both operation types p and q as a result of crossover.

6.3.6 Gene repair

In a mutation, the word length of a randomly selected cluster is changed. Accordingly, the word lengths of one or more operations in the cluster may change. This could change the tight upper–bound word lengths of some of the successor operations. For example, consider the sequence of two multiplications and an addition shown in Figure 6.3. As the

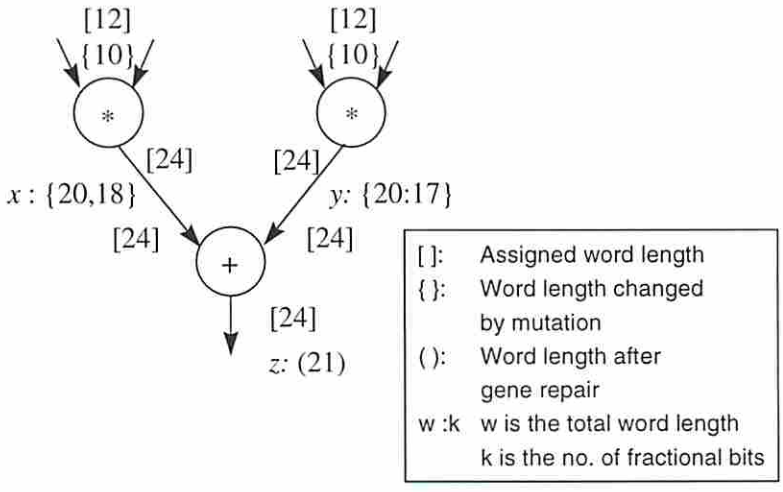


Figure 6.3: An example of gene repair operation

multiplication input word lengths are changed from 12 bits to 10 bits, each product has only 20 bits which are forwarded to the addition. Now, the adder input word length of 24 bits is unnecessary. Assume that in product x there are 18 fractional bits and in product y there are 17 fractional bits. In order to align the binary point the second operand of the adder can be shifted left using one of the available 24 bits. Now, the sum z has 18 bits and the maximum number of bits in z, assuming 3 integer bits, is 21, as opposed to the previous assignment of 24 bits. If this particular configuration represents a valid solution, i.e. the error is acceptable, the 24–bit adder can be replaced with a 21–bit adder without introducing any additional error.

In gene repair, such changes in the tight upper–bound word lengths of operations are detected and cluster word lengths are updated if necessary. This change in the tight upper–bound word length, however, is only specific to a chromosome that is being evaluated. The original tight upper–bound word lengths of all the variables corresponding to the environment of the algorithm remain unchanged. It is not always necessary that the tight upper–bound word lengths temporarily decrease. They may increase as well, as long as they do not exceed their original value. If the word lengths decrease however, potentially one or

more cluster word lengths may decrease, resulting in a fitness improvement of the chromosome being evaluated.

6.3.7 Selection

There are two important issues in the evolution process of the genetic search: population diversity and selective pressure. Strong selective pressure may force early convergence to a local optimum solution. In contrast, the search is ineffective if the population is too diverse in terms of the desired qualities, and the selective pressure is weak [40, page 56]. Several modifications of the cumulative–probability based basic selection [40, page 32] have been reported [4,25]. In the word–length optimization problem, the final selection of the optimal chromosome is constrained such that the word lengths in the optimal chromosome must satisfy the computation accuracy requirement. In order to meet this requirement we employ a simple selection method by applying the monotonic property of the computation–error and implementation–cost equations. By the monotonic property we imply that increasing the word length of an operand does not increase the worst–case computation error in the final result. Similarly, the use of a resource of a larger word length, without changing the allocation and binding, does not reduce the implementation area. We now describe the selection process in detail.

The selection of chromosomes that are more fit to reproduce occurs only when the number of chromosomes in the population is exactly equal to *max_population_size*, a parameter of the GA. Otherwise, more chromosomes are generated through crossover and mutation. First we rank all the chromosomes by their fitness. Then we determine the maximum of the worst–case errors corresponding to each solution in the first half. After comparing its value with the tolerable error (ϵ), two possibilities exist:

Case 1: The maximum of the worst–case errors in the first half is smaller than or equal to the tolerable error:

There are (*max_population_size* / 2) relatively better fitness (low cost) solutions that all satisfy the accuracy constraint. Chromosomes reproduced from this half are very likely to represent good solutions. Some solutions in the other half may also satisfy the accuracy constraint however, these are relatively high cost solutions, and are hence discarded.

Case 2: The maximum of the worst–case errors in the first half is larger than the tolerable error

The chromosomes are now ranked in the increasing order of error. Then, we determine the first chromosome that violates the accuracy requirement. Let the index of this chromosome according to error rank be ρ. Starting from the ρ–th chromosome the remaining (*max_population_size* - ρ) chromosomes are sorted again in the order of their fitness. Naturally, all these chromosomes represent solutions that violate the accuracy constraint. Therefore, in the new sorted (by fitness) order, the latter half represents solutions that do not meet the accuracy requirement, and are high in cost. Hence, these solutions are discarded.

The next evolutions begins with (*max_population_size* / 2) chromosomes when Case 1 applies, and (ρ+ ((*max_population_size* - ρ) / 2)) chromosomes when Case 2 applies. This selection mechanism falls into the category of deterministic, generational, rank–based, and elitist selection. It is deterministic because the chromosomes to be discarded are not selected randomly; generational because children may replace parent chromosomes only during selection and not on the fly. In this approach we take into consideration the relative position of a chromosome according to its fitness, and not the actual value of the fitness; hence, it is a rank–based approach. Finally, there is no distinction between parent and children chromosomes during selection, which is the defining characteristic of the elitist model [25]. The rationale for this selection is described next.

In Case 1, consider a chromosome in the discarded half that has unacceptable error. In this chromosome, the word length of at least one operand must be increased in order to reduce the error. According to the monotonic property of the cost equation, this cannot improve the fitness of that chromosome. Now consider a chromosome that has acceptable error. The fitness of this chromosome may be improved by reducing some of its word lengths, such that the corresponding error is acceptable. However, it is very likely, that a similar solution may also be generated from the selected chromosomes, perhaps with a smaller reduction in word lengths. In Case 2, from the monotonic property of the error equation it is observed that the word length of at least one operand must be increased, in order to meet the accuracy requirement. Then, the new fitness of that chromosome is lesser than or the same as the previous fitness. It is possible that the fitness of this chromosome can be significantly improved by reducing some the word lengths of some operations that

do not require high precision, such that the accuracy requirement is satisfied. However, the selected chromosomes can also produce a similar solution through subsequent genetic operations.

The deterministic nature of this selection causes the selective pressure to be high. However, by adjusting the rates and probabilities of word length change, the possibility of convergence to a local minimum is reduced. This is illustrated in SELECTION procedure shown in Figure 6.4. The two cases described above are considered in procedure

```
SELECTION(Pop, g, ε)
    /* Pop is the current population of chromosomes */
    /* g is the generation index */
    /* ε is the maximum tolerable error */
g ← g + 1
fitness_rank ← SORT(Pop, fitness)
    /* Elements of fitness_rank are the indices of chromosomes arranged in
     decreasing fitness order */
do p := 0 to ((max_pop_size/2) − 1)
        /* Consider that half of the total population in which all chromosomes
           have a better fitness than any chromosome in the other half */
   k ← fitness_rank[p]
        /* k is the index of a chromosome in Pop that has the p–th best fitness */
   Chromosome ← Pop[k]
   if Chromosome · error > ε
     then SELECTION–CASE–2(Pop, g, ε)
                /* At least one chromosome in the selected half of the population
                   violates the error constraint. Case 2 described above applies */
          return
   endif
end
    /* All chromosomes in the first half of the population sorted by fitness
     satisfy the accuracy constraint */
SELECTION–CASE–1(Pop, g)
return
```

Figure 6.4: A procedure to select chromosomes

SELECTION–CASE–1 shown in Figure 6.5 and procedure SELECTION–CASE–2 shown in Figure 6.6, respectively.

SELECTION–CASE–1(Pop, g)

$wordlen\_dec\_rate \leftarrow wordlen\_dec\_rate * 1.5$

$flag\_wordlen\_dec\_rate \leftarrow$ **true**

/* Word lengths can be decreased at a faster rate as described in Case 1 */

$\mathcal{P}^{+}(g) \leftarrow 0$

/* Word lengths need not be increased as all the selected chromosomes satisfy the error constraint */

$Pop \leftarrow Pop[0 : ((max\_pop\_size/2) - 1)]$

$pop\_size \leftarrow (max\_pop\_size/2)$

/* Update population for the next evolution cycle */

return

Figure 6.5: A procedure to select chromosomes – Case 1

SELECTION–CASE–2(Pop, g, ϵ)

$error\_rank \leftarrow$ SORT$(Pop, error)$

/* Elements of *error_rank* are the indices of chromosomes arranged in increasing error order */

do $p := 0$ **to** $(max\_pop\_size - 1)$

$k \leftarrow error\_rank[p]$

/* k is the index of a chromosome in *Pop* having the p–th least error */

$Chromosome \leftarrow Pop[k]$

if $Chromosome \cdot error > \epsilon$

then $\rho \leftarrow p$

/* The $(\rho + 1)$–th chromosome in *Pop* has the least error that violates the error constraint. Note that the index of the first chromosome is 0 */

break

endif

/* Error in at least one chromosome must be greater than ϵ. Otherwise Case 1 is true */

end

if $\rho < (max\_pop\_size/2)$

then /* At least 1/2 solutions have unacceptable error */

do $k := 0$ **to** $(max\_pop\_size - \rho)$

$ErrPop[k] \leftarrow Pop[error\_rank[\rho + k]]$

end

```
        Pop ← Pop ⋂ ErrPop
            /* ErrPop contains all the chromosomes violating the accuracy
               requirement and Pop contains all the chromosomes satisfying
               the accuracy requirement */
        ErrPop ← SORT(ErrPop, fitness)
        ErrPop ← ErrPop[0 : ((max_pop_size − ρ)/2)]
            /* Select the half of chromosomes in ErrPop with better fitness */
        Pop ← Pop ⋃ ErrPop
        pop_size ← ρ + ((max_pop_size − ρ)/2)
            /* The new population comprises all chromosomes that satisfy the error
               constraint, and a half of the chromosomes that violate the error
               constraint whose fitness is better than the other half */

        if ρ < (max_pop_size/4)
          then      /* At least 3/4–th solutions have unacceptable error */
               P+(g) ← 0.3
           else
               P+(g) ← 0.2
        endif
        if flag_wordlen_dec_rate = true
          then wordlen_dec_rate ← wordlen_dec_rate/2
               flag_wordlen_dec_rate ← false
                   /* Several chromosomes in the population violate the error
                      constraint. If word lengths were being decreased at an increased
                      rate in the previous evolution phase, decrease the rate of
                      word length reduction */
        endif
        return
endif

    /* Less than 1/2 solutions have unacceptable error */
P+(g) ← 0.1
wordlen_dec_rate ← wordlen_dec_rate * 1.5
flag_wordlen_dec_rate ← true
if ρ < (max_pop_size/10)
  then      /* Less than 10% solutions have unacceptable error */
       P+(g) ← 0
endif
Chromosome · best ← FINDBEST(Pop)
    /* Find a solution having the maximum fitness amongst those
       that satisfy the accuracy constraint */
Pop ← SORT(Pop, fitness)
```

```
Pop ← Pop[0 : ((max_pop_size/2) − 1)]
if Chromosome · best ∋ Pop
  then Pop[((max_pop_size/2) − 1)] ← Chromosome · best
            /* The best chromosome is always preserved in the elitist model */
       pop_size ← (max_pop_size/2)
       return
```

Figure 6.6: A procedure to select chromosomes – Case 2

In case 1, all the chromosomes in the new population have acceptable error. Hence, further reduction in area is possible by reducing the word lengths at a faster rate. Observe from Figure 6.5 that this is achieved by increasing the parameter *wordlen_dec_rate* to twice its value from the previous generation. A flag indicating that the word length reduction rate has been increased is also set. In case 2, when at least 3/4–th of the population contains invalid solutions that do not satisfy the accuracy requirement, chances of producing valid solutions must be increased. This is achieved by setting the probability of increasing word lengths $\mathcal{P}^+(g)$, to a maximum value of 0.3, as shown in Figure 6.6. Also, if the rate of reducing word lengths were increased in a previous generation, it is decreased. If only half the population contains invalid solutions, chances of reproducing more valid solutions are improved by setting $\mathcal{P}^+(g)$ to 0.2. The approach taken when less than half of the population contains invalid solutions is similar to case 1. Here the rate of reducing word lengths is increased in order to improve fitness. However, chances of reducing the number of invalid solutions are increased by setting $\mathcal{P}^+(g)$ to 0.1. During the evolution we expect that the word lengths of operations having a significant impact on the accuracy of the final result may be increased, while those of some other operations requiring less precision will be reduced in order to improve the fitness of valid solutions. Finally, if the number of invalid solutions is less than 10% of the population size, this special case is considered to be similar to case 1 described in Figure 6.5.

Selecting the parameters of a GA has a significant impact on its performance. However, finding good parameter values is considered to be an art, and not as much of a science [40, pp. 88–89]. We have intuitively selected the values of $\mathcal{P}^+(g)$, rates of word length change, and the probability of crossover which is explained in the next section. For the initial population the rate of word length reduction, *wordlen_dec_rate* is 5%. The rate

of word length increase is a constant 2%. Since we start with the tight upper–bound word lengths, $\mathcal{P}^{+}(0) = 0$.

6.3.8 Genetic algorithm

Once the genetic operators mutation and crossover, and the selection scheme are defined, the implementation of the genetic algorithm is straightforward. A procedure GA–WORDLENSELECT is shown in Figure 6.7.

```
GA–WORDLENSELECT(ε)
    /* ε is the maximum permissible error */
g ← 0
max_pop_size ← 500
wordlen_dec_rate ← 5%
wordlen_inc_rate ← 2%
P+(g) ← 0
Pop ← INITPOP
    /* One chromosome corresponding to each combination of ψ⃗ */
pop_size ← ∏_{t=1}^{z} langle max .ψ^t⟩
do
  while pop_size < max_pop_size
      if pop_size ≤ (0.7 * max_pop_size)
        then P^cross ← 0.3
         else P^cross ← 1 − (pop_size/max_pop_size)
      endif
      if RANDOM(0, 1) < P^cross
        then CROSSOVER(Pop)
                /* Generate two chromosomes through crossover operation */
             pop_size ← pop_size + 2
         else MUTATION(Pop)
                /* Generate one chromosome through mutation operation */
             pop_size ← pop_size + 1
      endif
  end
  SELECTION(Pop, g, ε)
  while termination_condition = false
```

Figure 6.7: Genetic algorithm for word–length selection

In order to determine convergence we use a simple terminating condition where three consecutive improvements of less than 1% in the best fitness are detected. We then assume that the best solution reachable has been obtained. Observe from Figure 6.7 that the probability of crossover is at the most 0.3 and it approaches 0 as the number of chromosomes in the population approaches *max_pop_size*. The rationale for this choice is as follows: It was reported by Schaffer *et al.* [49] that mutation plays a much stronger role in the optimization process compared to crossover. Recall from Sections 6.3.4 and 6.3.5 that a change in word lengths can occur only in mutation. The purpose of crossover is to randomly combine different values of ψ^t for various function types. As stated earlier if $\langle x \cdot \psi^p \rangle$ from chromosome X is a better choice for type p, and $\langle y \cdot \psi^q \rangle$ from chromosome Y is a better choice for type q, crossover between X and Y is likely to produce a chromosome with better fitness. Also recall from Section 6.3.3 that the initial population contains chromosomes corresponding to all combinations of $\vec{\psi}$. Hence, the rate of mutation is always high. When the population size is small the generation of new solutions by swapping the values of ψ^t between potentially good chromosomes is encouraged.

6.4 Word–length selection – overall procedure

A tool WORD–LENGTH OPTIMIZER shown in Figure 6.8 was implemented to automate the techniques described in this thesis. The algorithm is specified as a control data–flow

WORD–LENGTH OPTIMIZER$(\mathcal{G}, \tau, \epsilon)$
Read *prim_range*, *prim_error*
$[\mathcal{W}^{tight\_u.b}, \langle \max . \vec{\psi} \rangle] \leftarrow$ MAXACCURACY$(\mathcal{G}$, *prim_range*, *prim_error*$)$
Read $\langle \max . \vec{\psi} \rangle$
/* If the values reported in max. error analysis are too large */
CLUSTERING$(\mathcal{G}, \mathcal{W}^{tight_u.b}, \langle \max . \vec{\psi} \rangle)$
GA–WORDLENSELECT

Figure 6.8: The overall word–length selection procedure

graph $\mathcal{G}$. The number of maximum steps allowed in a nonpipelined implementation, or the initiation interval in a pipelined implementation is τ. The permissible worst–case error in the final result is ϵ. After reading the environment parameters i.e., the user-specified

numerical range of primary inputs, and error the tight upper–bound word lengths of all variables in the algorithm are determined from error analysis by preserving all bits. During this phase, the maximum number of distinct resource word lengths for each operation type are also determined. Since there may be dozens of word lengths, the user may also restrict this number for one or more types. Then, clustering is performed as described in Chapter 5, and the set of optimized cost functions using tight upper–bound word lengths is derived. Finally, these cost functions are further minimized to obtain the optimized set of resource and operation word lengths using the genetic algorithm described in this chapter. As described in Chapter 4, the error analysis always reports the worst–case error corresponding to the specified environment, and resource word lengths. The genetic algorithm enforces the error constraint and guarantees that the final solution is valid.

Using WORD–LENGTH OPTIMIZER we observed a significant reduction in the design area through multiple word length use. The results of our experimental verification are presented in the next chapter.

Chapter 7

Experimental results

In our experiments we considered three arithmetic–intensive algorithms namely, Discrete Cosine Transform (DCT), Gauss–3 elimination for systems of 3 unknowns, and 5x5 matrix determinant. These algorithms are commonly used in communication, signal processing, and control systems applications. Different accuracy and performance requirements for non–pipelined and pipelined implementations of these algorithms were considered, and word–length selection was performed using WORD–LENGTH OPTIMIZER introduced in the previous chapter. In these experiments we assumed the area of a one–bit adder/subtracter/multiplier to be the *unit area*. Area of a N–bit adder or subtracter was derived using a linear area–word length relation, and that of a $(M \times N)$–bit multiplier was obtained using a quadratic relation. The purpose of these experiments was to observe the relationship between functional resource area, numerical range of variables, desired accuracy and performance, the number of distinct resource word lengths used, and the word lengths. We observed that up to 40% reduction in the area is possible using multiple word lengths as opposed to using a single word length, for the given accuracy and performance requirements. In the following sections results of our experiments are given for each algorithm considered.

7.1 Discrete cosine transform

We considered a 1 dimensional 8–point Discrete Cosine Transform (DCT), implemented using Chen's algorithm [7]. This is a relatively small algorithm consisting of 26 additions, 8 subtractions, and 16 multiplications. Assuming the single–cycle architecture,

the fastest implementation of the DCT algorithm requires 8 steps. We considered two non-pipelined implementations requiring 8 and 12 major cycles respectively, and one pipelined implementation with an initiation interval of 2 major cycles. In this experiment, we will show the effect of the user–specified numerical range of the primary inputs and primary–input error on resource word lengths, for the same accuracy requirement in the final result. Specifically, the permissible error in the final result was allowed to be at the most 0.005% of the maximum value of the result.

7.1.1 Primary input range and error

Case 1: Small numerical range, small error

First we considered the case where the numerical range of the primary inputs as well as the primary input errors are small. Specifically, the primary inputs were allowed to vary in the range $[-5000, 1000]$ and the maximum under– and over–approximation input errors were 0.0003 and -0.0003, respectively. WORD–LENGTH OPTIMIZER performed the maximum accuracy analysis, where the tight upper–bound word lengths and the maximum number of distinct resource word lengths for each type were determined. In this case, the tight upper–bound word lengths for addition ranged from 10 bits through 31 bits. The maximum number of distinct word lengths, $\langle \max . \psi^{add} \rangle$ was 7. The tight upper–bound word lengths for subtraction ranged from 10 bits through 22 bits, and for multiplication from 10 bits through 30 bits. The value of both $\langle \max . \psi^{sub} \rangle$ and $\langle \max . \psi^{mult} \rangle$ was 5.

Case 2: Small numerical range, large error

Next we considered a situation where the numerical range of the primary inputs is small (same as the previous case) but the error is large. Specifically, we now allowed the maximum under and over approximation input errors to be as large as 0.001 and -0.001, respectively. This is approximately an order of magnitude increase in the input error compared to the previous case, yet the accuracy requirement of the computation was not changed. In this case, WORD–LENGTH OPTIMIZER found $\langle \max . \psi^{add} \rangle$, $\langle \max . \psi^{sub} \rangle$, and $\langle \max . \psi^{mult} \rangle$ to be 7, 5, and 5, respectively. The word lengths for addition and subtraction ranged from 15 bits through 32 bits, and for multiplication from 16 bits through 40 bits. The increase in

the tight upper–bound word lengths is due to more fractional bits required in the algorithm variables to satisfy the accuracy requirement.

Case 3: Large numerical range, large error

Finally we considered the case when both primary input range and errors are large. Here, the primary inputs were allowed to vary in the range $[-5.0e6, 1.0e7]$, and the maximum under and over approximation input errors were 0.001 and -0.001, respectively (as in Case 2). As the numerical range of the primary inputs is increased, additional integer bits become necessary. Consequently, the tight upper–bound word lengths increase further. Their values for addition range from 13 bits through 53 bits, for subtraction from 13 bits through 40 bits, and for multiplication from 14 bits through 53 bits. The values of $\langle \max . \psi^{add} \rangle$, $\langle \max . \psi^{sub} \rangle$, and $\langle \max . \psi^{mult} \rangle$ in this case were 7, 4, and 5, respectively.

7.1.2 Word–length optimization

Clustering of operations of all three types, for each $\psi^t \in [1, \langle \max . \psi^t \rangle]$, was performed as the first optimization step by WORD–LENGTH OPTIMIZER, for the three different cases described above. Then, the resource requirement of each cluster was determined, and optimized cost function sets were generated for the three different performance requirements stated above namely: execution delay (τ) of 5 and 8 major cycles in nonpipelined implementations, and initiation interval (λ) of 2 major cycles in the pipelined implementation. These cost functions were further minimized by selecting resource word lengths by the genetic algorithm in WORD–LENGTH OPTIMIZER. First, we considered unconstrained word–length selection where no restrictions were imposed on the size of the word length set. Second, only one resource word length was allowed for each function type. In Figure 7.1 we show a comparison between the area of nonpipelined DCT implementations with the execution delay of 8 major cycles, using a single optimized word length and multiple optimized word lengths. Table 7.1 shows the selected word lengths and the corresponding functional resource area for the three different primary input environments described above.

In Figure 7.2 and Table 7.2 we show similar results for nonpipelined DCT implementations with $\tau = 12$ major cycles, and the results for the pipelined implementations with $\lambda = 2$ are shown in Figure 7.3 and Table 7.3.

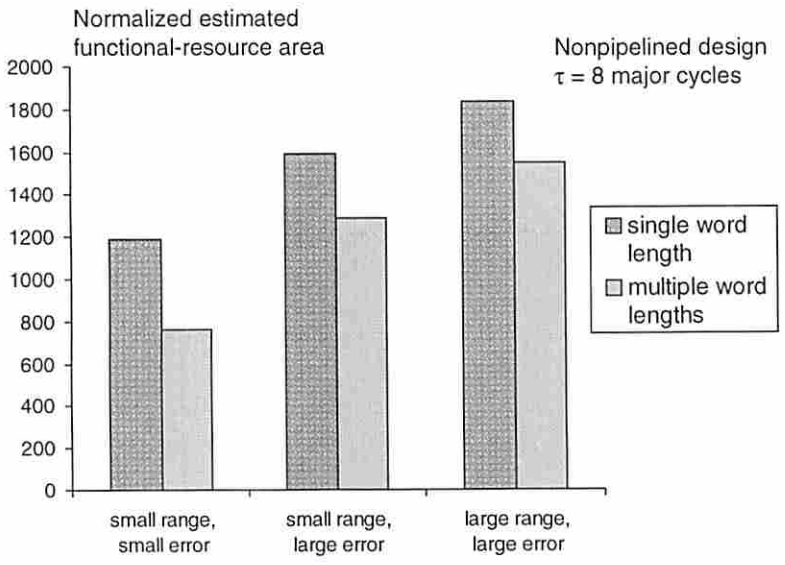


Figure 7.1: Comparison of DCT area using single and multiple word lengths ($\tau = 8$)

From Figures 7.1, 7.2 and 7.3, we observe that the estimated design area using multiple word lengths is smaller compared to a single word length use, in high speed as well as slow speed implementations. Note that the single word lengths considered above are also optimized using WORD–LENGTH OPTIMIZER. The average difference in the area is 17%. Consider the case of small input error and small input range, and a nonpipelined design requiring 8 major cycles. If a common pre–determined word length of 32 bits were used (which is the worst–case scenario that meets the desired accuracy requirement), the required estimated functional resource area would be 4320 sq. units. In comparison the design using multiple word lengths in the same error environment, and with the same performance constraint would require an estimated area of only 764 sq. units as seen from Table 7.1. The corresponding reduction in the estimated area is 82%.

It can also be computed from the data in Tables 7.1, 7.2, and 7.3 that for identical numerical range of the primary inputs, when the primary input error is low, the average difference between the functional resource area using single and multiple word lengths is as large as 27%. In the case of large input error, this difference reduces to 15%. Recall that in either case, the desired accuracy is 0.005% of the maximum value of the final result. In the case of small primary input error, only a few operations must be performed in high precision using resources with large word lengths, so as to meet the desired accuracy. In the case of large primary input error however, many operations must be performed in high precision. Resources with the largest required word length must be used in either case, when

Table 7.1: 1–D DCT optimized word lengths ($\tau = 8$)

Inputs	*add*		*subtract*		*multiply*		total area
	# res	word len	# res	word len	# res	word len	
$\vec{\psi}$		2		2		3	
small range	3	12, 12, 13	1	12, 12, 13	2	11, 10, 19	764
and	3	30, 29, 31	2	19, 22, 22	1	13, 10, 21	
small error					1	13, 19, 30	
$\vec{\psi}$		1		1		1	
	5	30, 29, 31	2	19, 22, 22	4	13, 19, 30	1187
$\vec{\psi}$		3		2		3	
small range	3	16, 17, 17	2	15, 15, 16	2	15, 14, 27	1286
and	3	29, 30, 30	1	23, 26, 26	1	17, 15, 30	
large error	1	37, 36, 38			1	17, 22, 36	
$\vec{\psi}$		1		1		1	
	5	28, 27, 29	2	17, 19, 19	4	16, 22, 36	1591
$\vec{\psi}$		2		3		3	
large range	3	20, 20, 21	1	12, 12, 13	1	11, 16, 26	1548
and	3	48, 51, 48	1	18, 19, 19	2	17, 16, 32	
large error			1	21, 29, 29	1	19, 29, 47	
$\vec{\psi}$		1		1		1	
	5	37, 36, 37	2	17, 24, 24	4	16, 25, 38	1833

the use of a single word length is allowed. Using distinct word lengths however, some operations are executed by resources with smaller word lengths. Note that the number of operations that can be performed in relatively low precision is high when the primary input error is low. Therefore, the difference in the functional resource area in this case is large compared to the case when the primary input error is large. This demonstrates that the design area is sensitive not only to the numerical range of variables in the algorithm, but also to the desired accuracy in a given error environment.

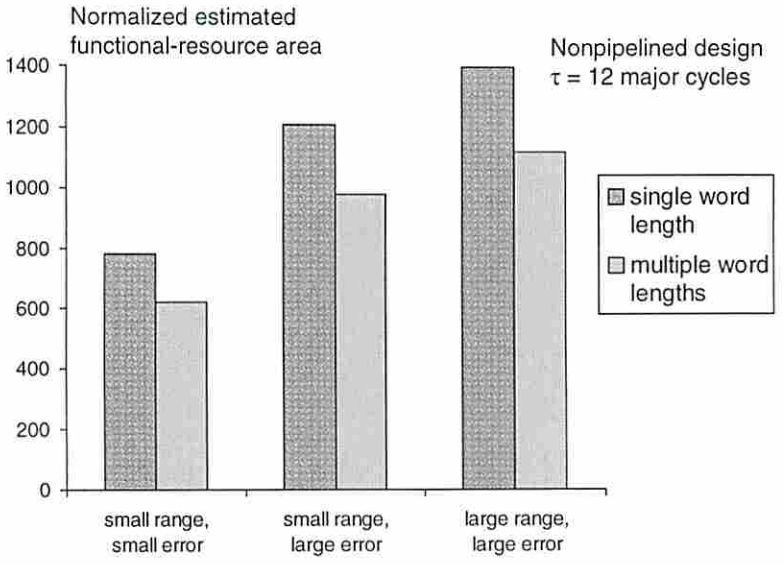


Figure 7.2: Comparison of DCT area using single and multiple word lengths ($\tau = 12$)

Table 7.2: 1–D DCT optimized word lengths ($\tau = 12$)

Inputs	*add*		*subtract*		*multiply*		total area
	# res	word len	# res	word len	# res	word len	
$\vec{\psi}$		2		2		2	
small range	2	12, 12, 13	1	10, 10, 11	2	13, 10, 21	622
and small error	2	29, 28, 29	1	17, 20, 20	1	13, 19, 30	
$\vec{\psi}$		1		1		1	
	4	21, 20, 21	2	14, 17, 17	3	13, 17, 28	781
$\vec{\psi}$		2		2		2	
small range	2	16, 16, 17	1	15, 15, 16	2	17, 15, 30	976
and large error	2	32, 32, 33	1	24, 27, 27	1	17, 19, 31	
$\vec{\psi}$		1		1		1	
	4	25, 24, 26	2	22, 25, 25	3	16, 22, 36	1210
$\vec{\psi}$		2		2		3	
large range	2	18, 19, 19	1	13, 13, 14	1	14, 16, 29	1118
and large error	2	43, 49, 43	1	22, 30, 30	1	17, 17, 18	
					1	22, 30, 30	
$\vec{\psi}$		1		1		1	
	4	37, 36, 37	2	17, 24, 24	3	16, 25, 40	1396

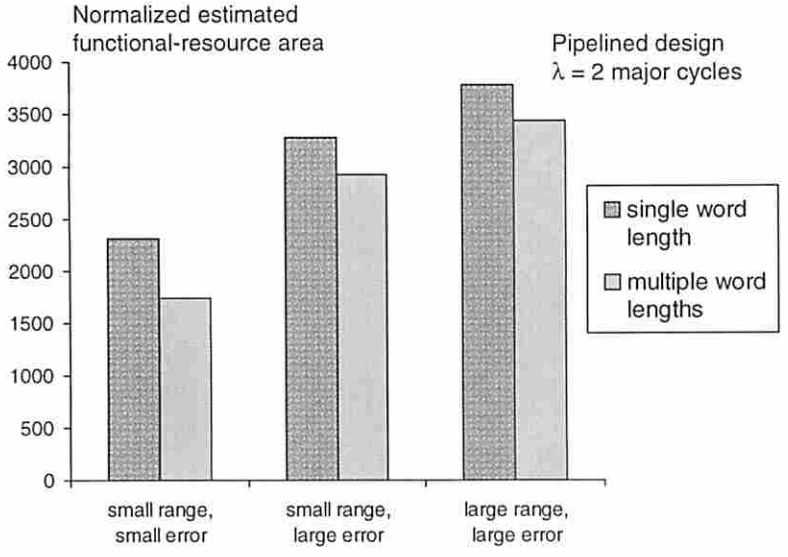


Figure 7.3: Comparison of DCT area using single and multiple word lengths ($\lambda = 2$)

Table 7.3: 1–D DCT optimized word lengths ($\lambda = 2$)

Inputs	*add*		*subtract*		*multiply*		total area
	# res	word len	# res	word len	# res	word len	
$\vec{\psi}$		4		4		3	
small range	5	11, 12, 12	1	10, 10, 10	4	11, 10, 19	1750
and	2	12, 12, 13	1	10, 10, 11	2	13, 10, 21	
small error	6	20, 21, 21	2	12, 12, 13	3	13, 19, 30	
	1	29, 28, 29	1	18, 21, 21			
$\vec{\psi}$		1		1		1	
	13	20, 20, 21	4	14, 17, 17	8	13, 19, 30	2317
$\vec{\psi}$		3		4		3	
small range	7	17, 17, 18	1	15, 15, 15	4	15, 14, 28	2927
and	6	28, 29, 29	1	15, 15, 16	2	16, 15, 28	
large error	1	37, 36, 38	2	16, 16, 17	3	17, 23, 38	
			1	29, 31, 31			
$\vec{\psi}$		1		1		1	
	13	27, 27, 28	4	23, 26, 26	8	16, 22, 36	3284
$\vec{\psi}$		2		3		3	
large range	7	18, 19, 19	1	11, 11, 12	2	11, 17, 27	3439
and	7	43, 42, 43	3	16, 16, 17	4	17, 17, 33	
large error			1	27, 35, 35	2	17, 27, 43	
$\vec{\psi}$		1		1		1	
	13	37, 36, 37	4	17, 24, 24	8	16, 25, 39	3777

7.2 Gauss–3 elimination algorithm

The Gauss–3 elimination algorithm consists of 18 multiplications, 12 subtractions, and 3 reciprocal computations. In this experiment we assumed that the reciprocal was generated externally, and its word lengths were not optimized. We also assumed primary inputs in the range [-50, 50], and the worst–case primary input error is ± 0.0001. The Gauss–3 elimination algorithm has several sequential computations and as a result, the range of intermediate variables and the tight upper–bound word lengths grow rapidly. We observed that even for a small primary input range used in this experiment, the tight upper–bound word lengths ranged from 9 bits through 230 bits. We considered two different accuracy requirements for the specified environment. In case (a), the maximum tolerable error was 0.1%, and in case (b) it was 0.15% of the maximum value of the final result. In each case, we considered five pipelined implementations of this algorithm, corresponding to initiation intervals 1, 3, 5, 7, and 9. Once again clustering was followed by word–length selection using the genetic algorithm. We considered unconstrained optimizations allowing resources of multiple word lengths, and constrained optimizations where a common word length was selected for resources of a particular type. The final solutions derived by WORD–LENGTH OPTIMIZER contain resources of distinct word lengths. These results are given in Table 7.4. A comparison of the cost of various implementations is shown in Figure 7.4.

Table 7.4: Gauss–3 elimination pipelined implementations

Initiation interval # major cycles	$\epsilon \leq 0.1\%$ $\vec{\psi}$ $[*, -]$	Normalized estimated cost	$\epsilon \leq 0.15\%$ $\vec{\psi}$ $[*, -]$	Normalized estimated cost	$\vec{\psi} = [1, 1]$, $\epsilon \leq 0.15\%$ Normalized estimated cost
1	[5, 4]	8331	[6, 2]	7154	14952
3	[3, 3]	3577	[3, 2]	2659	4984
5	[3, 3]	2850	[2, 4]	1888	3339
7	[3, 2]	2067	[3, 4]	1915	2492
9	[1, 2]	1687	[1, 2]	1687	1694

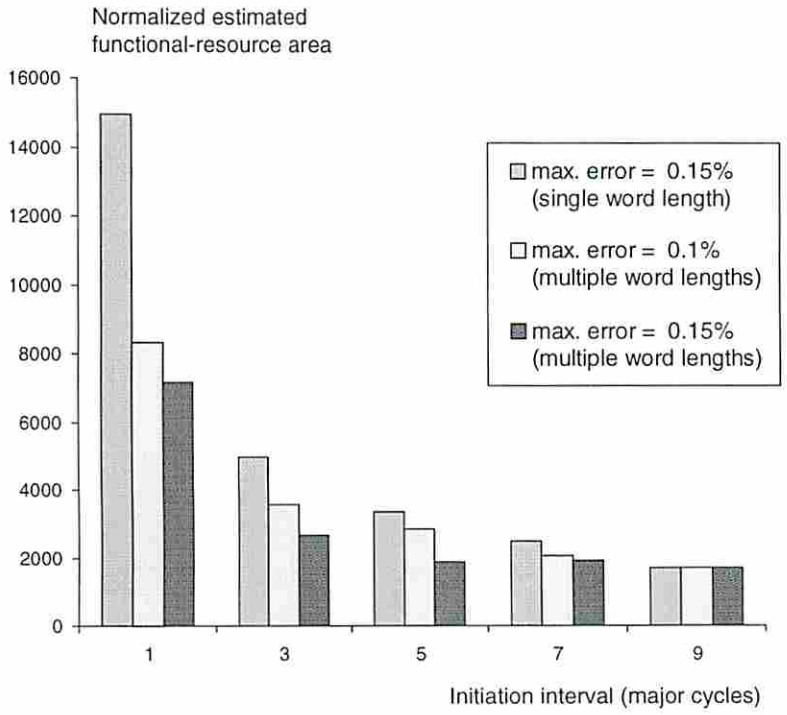


Figure 7.4: Gauss–3 elimination area requirements for different accuracy constraints

From these results it is observed that in general, designs using multiple word lengths are smaller than those using a common word length. The difference in the area is significant when the required performance is high. Particularly in the design with initiation interval of 1 major cycle, the resource requirement is extremely high. Since all resources of a type must be assigned the largest required word length when a single word length is used, the difference in the estimated area compared to a design (with the same performance and computation accuracy) using resources of multiple word lengths is as much as 52%. The average difference in the area of these designs using single and multiple word lengths is 32%. Note that in this comparison, the optimized word lengths for designs using a single word length were selected by WORD–LENGTH OPTIMIZER. In the worst case, where these word lengths are pre–determined, the difference in the estimated area could be much larger.

From the results in Table 7.4 and Figure 7.4, we also observe that for a given primary input range and error, the estimated design area changes monotonically with the desired accuracy of the computation. The improvement and saturation of the fitness of the best solution and the corresponding changes in the worst case computation error for the first

case of the implementation with initiation interval 5, are shown in Figure 7.5 and Figure 7.6 respectively.

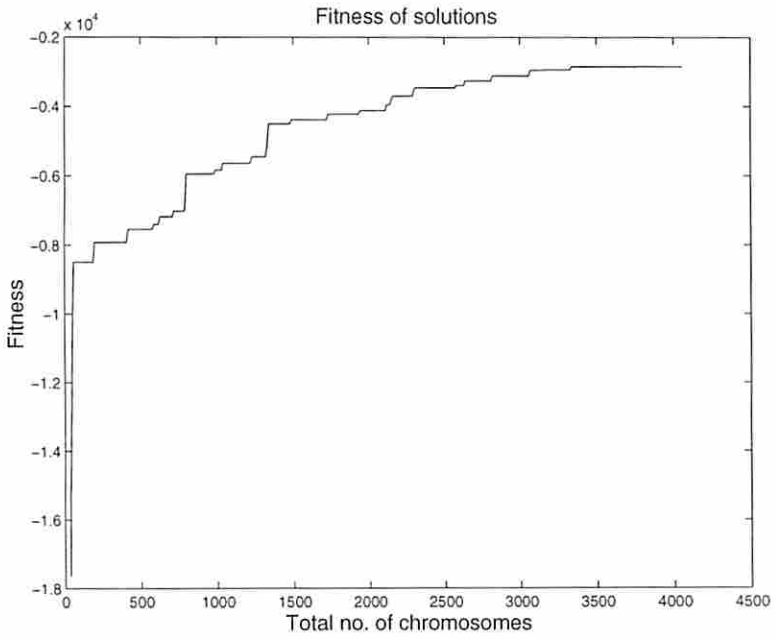


Figure 7.5: Gauss–3 elimination fitness improvement: pipelined implementation, latency = 5 major cycles, $\epsilon = 0.1\%$

7.3 Determinant of a 5x5 matrix

Computing the matrix determinant is a well known computationally intensive problem. In fact, methods such as LU–decomposition, or QR–factorization are often used to avoid the computation of matrix determinant. Using resources of multiple word lengths it may be possible to obtain a cost effective hardware implementation of matrix determinant. The 5x5 matrix determinant computation consists of 105 multiplications, 29 additions, and 40 subtractions, yet the fastest implementation requires only 14 steps. In our experiment we assume that the matrix elements (i.e. the primary inputs) are in the range [1, 150], and the error in these inputs is moderate. Then, the tight upper–bound word lengths range from 10 bits through 60 bits. We allowed the maximum number of distinct multiplier and subtracter word lengths to be 10, although the actual numbers determined during the maximum accuracy analysis were higher. The maximum number of distinct adder word lengths was 8. We considered 5 nonpipelined implementations of this algorithm starting from the

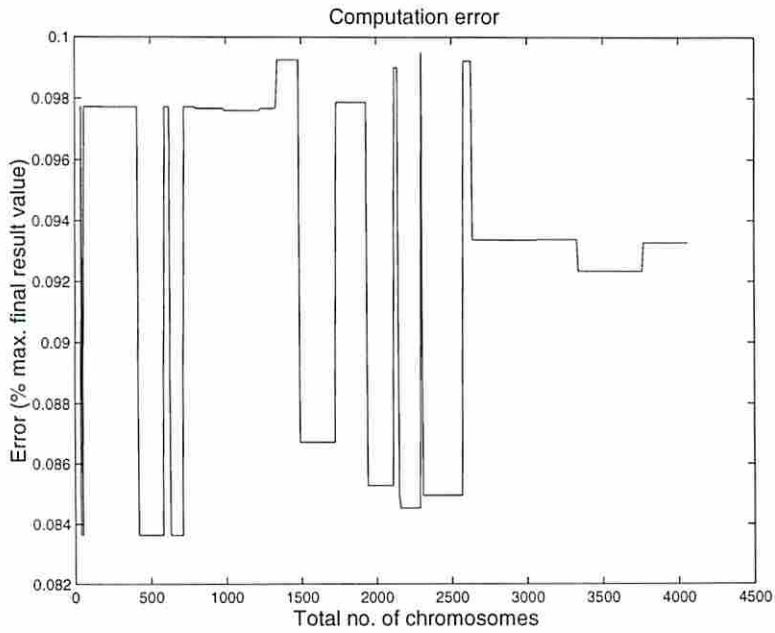


Figure 7.6: Gauss–3 elimination variation in the computation error corresponding to the best solution in the population

fastest one i.e. the execution delay of 14 major cycles through the one allowing 20 major cycles for execution. Once again we considered two different accuracy constraints: one where the maximum permissible error was 1% of the maximum value of the final result, and the other where the permissible error was at the most 1.5% of the maximum value of the final result. The $\vec{\psi}$ selected by WORD–LENGTH OPTIMIZER, and the corresponding estimated resource area in the case of unconstrained optimizations for different accuracy requirements are shown in Table 7.5. The estimated resource area in each design using a single word length selected by WORD–LENGTH OPTIMIZER from constrained optimization is also given. A comparison of the estimated area using multiple word lengths and a single word length is shown in Figure 7.7.

From Table 7.5 we observe that the estimated area using multiple word lengths does not change monotonically with the required performance. This is due to a variation in inter–cluster resource sharing. For a given number of clusters of distinct word lengths, and a given number of total steps for the execution of the algorithm, it was shown in Section 5.5.3 that resources assigned to a cluster of large word length may implement operations in a cluster of smaller word length, if and only if the execution intervals of the two clusters do not overlap. If a resource is shared between two clusters, the resource requirement of

Table 7.5: 5x5 matrix multiplication

Required # steps	$\vec{\psi}$ $[*, -, +]$	Normalized estimated cost $\epsilon = 0.1\%$	$\vec{\psi}$ $[*, -, +]$	Normalized estimated cost $\epsilon = 0.15\%$	Normalized estimated cost using 1 word length
14	[3, 2, 4]	3619	[3, 4, 6]	3508	6502
15	[2, 4, 3]	4059	[4, 2, 3]	3960	5400
16	[3, 5, 3]	3885	[3, 3, 3]	3788	5400
17	[4, 5, 3]	3297	[4, 5, 5]	3105	4995
18	[6, 3, 6]	3035	[4, 3, 5]	2624	4995

the smaller word length cluster is reduced further. The cluster execution interval jointly depends on the number of clusters and the total number of time steps available. Hence, it does not change monotonically with the performance requirement. This affects the sharing of resources between clusters. Moreover, inter–cluster resource sharing is performed using an estimate of the cluster execution interval and not an actual schedule, since our word–length optimization is performed at the algorithmic level, before architecture synthesis. Recall from Section 5.5.3 that an estimate of cluster execution interval is computed using the predicted number of resources required in each cluster. In order to guarantee that the performance requirement is met, the resource predictions are pessimistic. In Section 5.5 we showed that these predictions depend not only on the total number of time steps available for the entire algorithm, but also on the number of clusters. Therefore, the overall predicted resource requirement embedded in our cost model may not change monotonically with the performance requirement. As a consequence, the estimated area of a design requiring more execution steps may be greater compared to a design requiring fewer time steps. This is seen in solutions requiring 15 and 16 steps, whose estimated area is more than that of a design requiring only 14 steps. The objective of our technique is only to determine the optimized word–length sets for operations and resources. During architecture synthesis using these word lengths, the optimized number of resources must be determined to ensure that a design with a relatively low performance requirement also requires relatively less area.

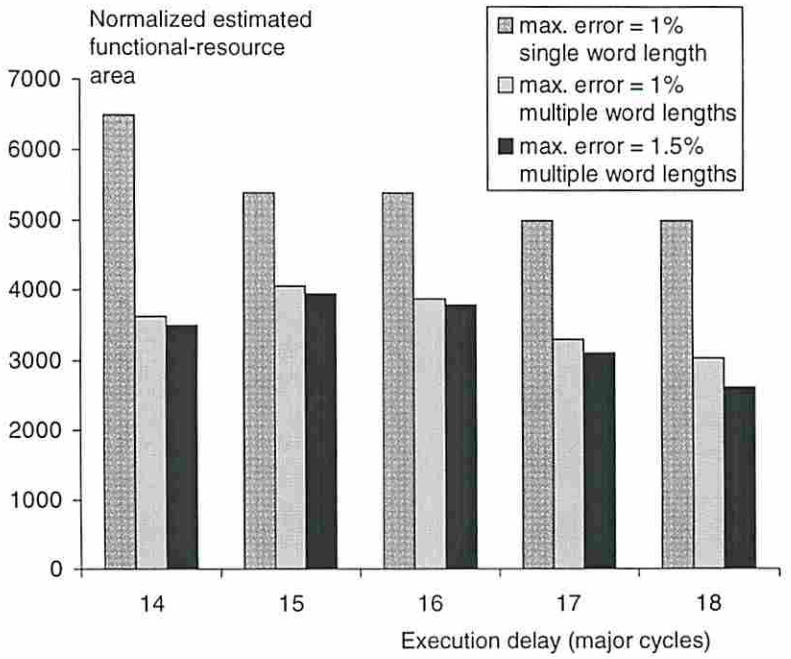


Figure 7.7: 5x5 matrix determinant normalized estimated functional–resource area using multiple and single word lengths

The monotonic area–performance relationship is observed in the case of designs using a single optimized word length. We also observe, that the overall estimated area requirement is always less using multiple word lengths compared to designs implemented using only one word length. In this example, the average reduction in area is 35%. A reduction in the estimated area when the required computation accuracy is low is also observed from Table 7.5 and Figure 7.7. In Figure 7.8 we show how the fitness improves and saturates as better chromosomes are produced during the evolution process.

From these experiments we observe that a significant reduction in the design area is possible by using resources of multiple word lengths. In the previous example, the average area reduction is 35%. In the Gauss–3 elimination example it is 32%, and in the DCT example it is 18%. From these examples we observe that for a given performance requirement, the area of implementation changes according to the accuracy requirement. In other words, the circuit area is sensitive to the desired computation accuracy as well as performance, and can be manipulated using resources of different word lengths. In a given algorithm, it may not be necessary to perform all computations in high precision to obtain the final result with the desired accuracy. If resources of smaller word lengths are

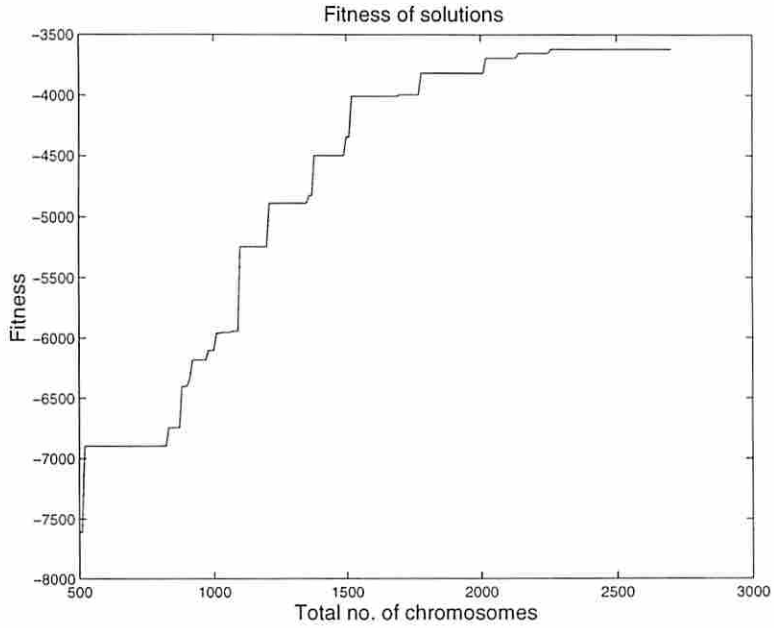


Figure 7.8: 5x5 matrix determinant fitness improvement: nonpipelined implementation, execution delay = 14 major cycles, $\epsilon = 1\%$

assigned to these operations, a significant reduction in circuit area is possible, as seen from these results.

Chapter 8

Conclusion and future work

8.1 Conclusion

In this thesis we considered the problem of word–length selection at the algorithmic level to minimize the hardware implementation cost while satisfying the performance and computation accuracy requirements. In hardware implementation of arithmetic–intensive algorithms, word lengths of operations and resources have a direct bearing on the cost, speed, and the accuracy of the computation. Resources with small word lengths cannot directly implement operations requiring large word lengths. Therefore, the choice of word lengths affects resource sharing. Accuracy of the computation is proportional to the number of bits used. Small word lengths imply smaller resources and possible reduction in area. Thus, the optimal word length set to be used in the implementation depends on the specified accuracy and performance requirements. We presented theory and techniques to automate this important step usually performed manually.

The limitations of word–length selection after architecture synthesis were illustrated in Chapter 1. In order to select word lengths at the algorithm level, we first analyze the effect of a specific choice of word lengths on the worst–case computation error in the final result of the algorithm. An analytical technique to describe the latter in terms of word lengths was developed in Chapter 4. Our technique is independent of the implementation style and is based on the numerical analysis of the algorithm and its environment i.e., the numerical range of the primary inputs and error. Using this technique, the tight upper–bound word lengths of all the algorithm variables can be determined for the specified environment. Assigning a word length larger than the tight upper–bound word length does not improve the accuracy of the final result.

In Chapter 3 we presented a novel cost function incorporating functional as well as other resources of multiple word lengths. In this cost function, the individual resource cost, that was assumed to be a constant in the traditional high–level synthesis, was modeled as a function of its word length. The exact minimization of the nonlinear cost function yields the optimal word length set for the algorithm, which is likely to be a very difficult problem. A two–step optimization technique was developed.

In the first step, named clustering, described in Chapter 5, a set of optimized cost functions was obtained. Each function in the set corresponds to a unique combination of number of distinct resource word lengths of all the operation types in the algorithm. These functions are optimized under the assumption of maximum accuracy requirement, which is achieved by using the tight upper–bound word lengths. By taking advantage of the word–length compatibility of operations, clustering minimized the requirement of resources of large word lengths. Then, the utilization of large word length resources was further improved by taking advantage of the scheduling compatibility of operations, and the fact that a resource of larger word length can directly implement an operation of a smaller word length.

Finally, the genetic algorithm described in Chapter 6 searches through discontinuous, nonlinear solution space efficiently, taking advantage of the monotonic properties of implementation–cost and computation–error equations in terms of word lengths. The word lengths of resources are altered and its effect on the worst–case computation error is examined using the analytical error model. We consider the complete set of optimized cost functions generated by clustering. When no further reduction in the implementation cost is possible while satisfying the accuracy requirement, the function resulting in the minimum cost is selected. Thus the number of distinct word lengths of each operation type, and the word lengths are selected simultaneously to form the optimized solution. The simultaneous selection of these two parameters of the final solution is essential because, the two parameters are interdependent, and they jointly determine resource sharing, which has a direct bearing on the implementation cost.

Our results show that on an average, a 30% reduction in functional–resource area is possible by using multiple word lengths as opposed to the use of a single pre–determined word length for the entire data path. This illustrates the importance of the word–length optimization step performed during manual design or prior to data path synthesis. We also illustrate that the design area is sensitive the the desired accuracy of computations. In a

computation that demands high accuracy in the final result, the precision of all operations may not have a significant impact on the accuracy of the final result. These operations can be implemented using resources of smaller word lengths, while the others may be implemented by resources with large word lengths. By determining the appropriate resource and operation word lengths at the algorithm level, a low–cost design can be generated using multiple word lengths, that meets the accuracy and performance requirements.

8.2 Contributions

The key contributions of this research are listed below.

1a. Traditional high–level synthesis explores the trade off between the cost and performance of an implementation. We have shown that the cost (area) of a design is sensitive to the desired accuracy of the computation. Thus a third optimization parameter, accuracy of computation, is introduced that allows a cost–performance–accuracy trade off.

1b. Our research shows one way to model and solve the above mentioned three–way trade off or optimization problem.

2a. An analytical technique was developed to model the worst–case computation error in an algorithm. This technique is based on the numerical analysis of the algorithm, and is independent of its implementation.

2b. An automatic tool was developed to perform the error analysis of an algorithm represented as a control data flow graph with no unbounded data–dependent loops. This tool is also used to verify the error constraint given a set of word lengths.

3. A novel cost function was developed that incorporates resources of multiple word lengths, and models the individual resource area as a function of its word lengths.

4a. Clustering is the first important step in the optimization of the cost function mentioned above. A dynamic programming solution to clustering problem (to minimize the normalized lower–bound estimated functional–resource area) taking advantage of word–length compatibility was developed.

4b. A novel behavioral technique to predict the resource requirement incorporating multiple word length use was developed. These predictions aim to optimize the cost function by minimizing the number of large word length resources, and by maximizing their utilization.

4c. An automatic tool was developed to perform clustering and resource prediction, and to generate the set of optimized cost functions under the assumption of the requirement of maximum achievable accuracy. In other words, the resources word lengths are assumed to be the tight upper–bound word lengths.

5. A genetic algorithm and a tool, WORD–LENGTH OPTIMIZER were developed to select the optimized set of word lengths for the algorithm.

8.3 Future research

This research was based on numerical properties of algorithms and high–level synthesis issues. By introducing the computation accuracy as an optimization parameter in synthesis of circuits, several new research directions in this field, and some in algorithm analysis can be suggested. We highlight some relevant research topics.

8.3.1 Synthesis

High–level synthesis Given resources of distinct, known word lengths, and operation word lengths, traditional problems in high–level synthesis namely, allocation, scheduling, and binding become more interesting. As stated in the introduction, the problem of synthesis using multiple word length resources is loosely similar to the module selection problem. In the module selection problem any operation of a given type can be implemented by any style of module of that type. When multiple word lengths are used the problem is tricky because a smaller word length resource cannot directly implement an operation requiring larger word length.

Accuracy–specific module synthesis Accuracy specific synthesis is of special interest in multipliers. Our word–length optimization may determine the optimized word lengths of a multiplier to be say 10, 15, 16 where 10 and 15 are the input word lengths and 16 is

the word length of the product. A straightforward approach is to use a 10x15 multiplier (assuming it is available) and to ignore the 9 least significant bits. However, if a multiplier could be designed that generates only the 16 most significant bits, it is likely to utilize fewer gates. Hence it could be faster, smaller, and may consume less energy. Synthesis of such data–path components is an interesting logic optimization problem.

Adaptive computing The trade off between the word lengths and precision is particularly useful in a dynamically changing environment. If the input noise or error is high, the computation is performed in high precision. Otherwise, in a low–noise environment some least significant bits are not used. This may lead to a speed up in the computation as well as reduction in power consumption.

8.3.2 Numerical analysis

In our research we modeled the worst–case error in computations. In a realistic environment the probability of a computation generating the worst–case error may be extremely low. Then a statistical model of the computation error, expressed as mean, variance and other moments if necessary can replace the worst–case model. This can be achieved by treating the input/output operands of a computation as random variables. Then the probability density function of error in the output operand is a function of the error density functions of the inputs.

8.3.3 Algorithm analysis

Algorithm transformation In our research we assumed that the sequence of computations in an algorithm remains unchanged. As long as the algorithm expresses the same functionality the sequence of operations can be transformed in order to improve the numerical robustness of the algorithm. Consider a simple example $x = a * b * c$. Assume that a and b are extremely small numbers and c is a sufficiently large number. If a product is carried out in the specified order, the partial result $y = a * b$ may require a large word length in order to preserve the accuracy. Alternately, if $z = b * c$ is performed first z may be represented with acceptable precision using fewer bits compared to y. A technique similar to our error analysis technique can be developed to identify such transformations.

Hierarchical analysis In hardware implementations, complex operations such as division, square–root, or trigonometric functions are implemented using a sequence of elementary operations such as multiplication, addition, and subtraction, and some times a table look–up operation. Several different algorithms can be considered for the implementation of a complex operation, and the implementation of these algorithms can be simultaneously optimized while optimizing the main algorithm. Especially, a saving of only a few bits in the look–up table at the expense of using a larger multiplier or adder could result in a significant reduction in the design cost.

Appendix A

Statistical estimation of system energy and power

In this thesis, we presented theory and techniques to select word lengths of algorithm variables and resources to minimize the hardware implementation cost while meeting the performance and accuracy requirements. The cost of a circuit comprises several parameters such as circuit size, die cost, cost of designing the circuit, fabrication and packaging cost, and energy/power consumption. Many of these cost parameters are closely related to the circuit area, hence, in the cost model introduced in Chapter 3, the overall circuit area expressed in terms of resource word lengths is assumed to represent the design cost. In this appendix, we describe a technique to express the energy/power consumption of a circuit in terms of word lengths of its resources. Using this technique word–length selection at the algorithmic level can be used to obtain a low–energy or low–power design. In order to model the energy/power consumption of a circuit or system in terms of word lengths of its resources we present a novel technique FREEDOM to predict the former given the behavioral specification of a circuit/system and library components. The *early prediction* gives circuit designers the *freedom* to make numerous high–level choices (such as die size, package type, and latency of the pipeline) with confidence that the final implementation will meet power and energy as well as cost and performance constraints. Our unique statistical estimation technique associates low–level, technology dependent physical and electrical parameters with expected circuit resources and interconnect. Further correlations with switching activity yield accurate results consistent with implementations. All feasible designs are investigated using this technique and the designer may trade off between small size, high speed, low energy and low power. The results for designs of two popular signal processing applications, predicted prior to synthesis, are within 10% accuracy of power estimates performed on synthesized layouts.

A.1 Motivation

Performance, area, power, and energy consumption are some of the most important attributes of complex digital signal processing systems such as secure mobile spread spectrum systems, and JPEG and MPEG image compression/restoration systems. Early predictions of cost, performance, power, and energy using the behavioral specification of the system are useful in guiding the search through the vast, discrete system–level design space. Using prediction results, architecture choices such as the number of resources to be used, and their allocation, binding and scheduling can be made to simultaneously satisfy area, speed and power constraints. The earliest choices usually have the most influence on the design. Compared to the behavioral-level the *degrees of freedom* in terms of choices available to the designer diminish significantly at the architecture level and are reduced again at the subsequent gate and transistor levels. For example, in a known architecture the number of multipliers used in a matched filter implementation is fixed. Their sharing and usage is also fixed. At this level predictions apply to a single architecture. In order to explore the vast design space even using RT–level estimators, the architecture synthesis steps must be repeated several times, using the "generate and test" process of searching for a low–energy, low–power design. The gray segments on the design space line in Figure A.1 are the low

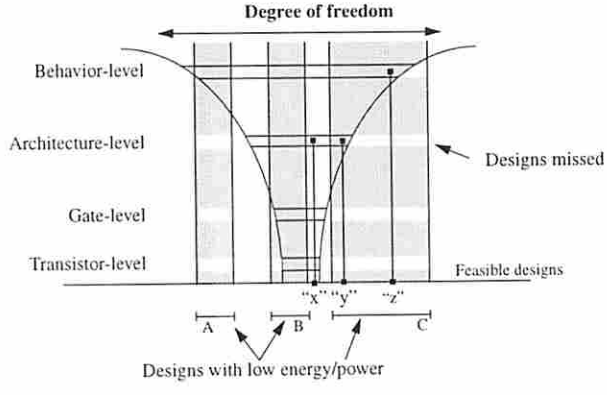


Figure A.1: Limitations at various levels of prediction

power/energy designs. Once the architecture is fixed all the low power designs in segment "A" and many in segment "C" are automatically excluded. If the selected architecture corresponds to implementation "x", failure to meet power constraints will cause the design to be excluded and exhaustive steps to obtain a new architecture will be repeated. If architecture/design "y" is then selected, it is still likely that design "z" also will meet area

and speed constraints and have lower power/energy consumption compared to design "y". However, using prediction at the behavioral level a very large spectrum of predicted feasible implementations that meet various area, speed, energy and power constraints can be identified early. Behavioral–level predictions allow the designer to significantly decrease the number of actual RTL designs he or she must examine. The designer is then able to make system and RT–level design choices that will lead to implementations with the desired characteristics.

FREEDOM is an accurate statistical technique to estimate power consumed by each task in a application specific system by means of a statistical model relating (a) the predicted lengths of nets in the implementation, and (b) the switching activities on the wires. By correlating (a) and (b) we show how power and energy consumption in a system are estimated directly from its behavioral description without taking any synthesis steps, by predicting the impact of optimizations performed during the design process on area, performance, energy consumption and power.

The appendix is organized as follows. Accuracy achievable at different levels of circuit abstraction and the factors influencing it are described in Section A.2. An overview of our approach is given in Section A.3. Section A.4 describes the theoretical basis and mathematical model of our estimation algorithm. Experimental results for real signal processing applications are presented in Section A.5, followed by conclusions in Section A.6.

A.2 Prediction accuracy at different levels of abstraction

The power consumed by CMOS circuits depends on the supply voltage, the switching capacitance and the switching activity. The well known transistor–level predictors SPICE and PowerMill [20] obtain accurate capacitances from the layout where the switching activity is simulated. At the gate level, the focus of prediction methods remains on accurate estimation of the switching activity. It was shown by Brand and Visweswariah that inaccurate modeling of the capacitance could lead to severe inaccuracies in predictions at the gate level [3]. Accurate methods at this level are reported in several publications [14,38,43]. At the architecture level the variation in capacitance is significant. This is mainly because the cost of communication, interconnect and resource sharing is significant, even dominant. Power consumption by the interconnect is significant [32,35]. It is imperative that the interconnect capacitance distribution for the entire chip (or MCM) be predicted accurately.

A comprehensive technique for capacitance estimation based on layout estimation after architectural synthesis was reported [28]. At the behavioral level, Potkonjak *at al.* considered several DCT algorithms [47] such as Lee's, Wong's and Vetterli's. The objective of their research was to determine a low power algorithm for the given cost–performance requirements. They synthesized each algorithm to obtain an architecture and compared the results of predictions. Our research is significantly different than this technique because we replace expensive behavioral synthesis steps by architecture estimation. Specifically, we estimate the number of functional resources such as adders and multipliers, control resources such as muxes and registers, and also the interconnect. We generate these estimates for *all predicted feasible designs*, and predict power for each one. Thus, we explore the complete behavior of an algorithm for the entire feasible design space, and not just a given cost–performance requirement. Methods to estimate the architecture before predicting power were also investigated by Mehra and Rabaey [39] however, their functional area estimates were off by as much as 96.1%, and interconnect capacitance estimates by 52.1%. Consequently, the power predictions were also significantly off. The architecture estimation techniques we use [26] were shown be accurate [17]. *In our approach we predict the impact of architectural optimization and technology mapping, hence avoiding the estimation error suggested by Brand.*

A.3 Behavioral system–level predictions

At the behavioral level, critical design choices have not been made and the architecture is not fixed. Hence, detailed wiring information is not available. FREEDOM provides a unique solution by producing accurate wire length and capacitance distribution estimation throughout a chip or MCM. The wire–length distribution in chips was shown to be geometric by Kurdahi [31]. This distribution is also used by others [3, 35]. Additionally, layout and technology–dependent parameters such as wire capacitance per unit length, and average gate input, output and internal capacitances are used to derive the capacitance distribution.

We first estimate the area and performance characteristics of possible system designs using the Behavioral ESTimator BEST [26]. Consider the Task Flow Graph (TFG) representation of a homomorphic system for processing speech shown in Figure A.2. Each task

is associated with a Control Data Flow Graph (CDFG)[1]. The implementation style for each

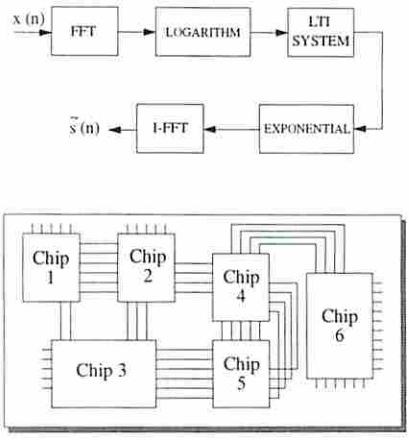


Figure A.2: System Task–Flow Graph and implementation

task could be different, such as semi–custom ASIC, FPGA, or off–the–shelf component. A system composed of these chips or dies is assembled on a PC board or an MCM or a combination of the two. For off–the–shelf components such as dynamic RAMS, cost, delay and power consumption are known. Our predictor estimates these attributes for undesigned tasks as described below.

Given the behavioral description of a task (CDFG) and a library of physical modules to be used for synthesis, many designs are feasible depending upon the area and timing constraints. Under tight area constraints, the resulting design tends to be *serial* as it has a small number of functional units. On the other hand, under strict timing constraints, the resulting designs are *parallel* or have a large number of functional operators. A formal method to predict each of these design possibilities was introduced in BEST. BEST investigates all feasible execution times (expressed in number of steps of clock cycle) for a given task, based on critical path analysis. Then, corresponding to each possible time, the required number of functional operators of each type, multiplexers, registers and 2–point nets are predicted. We use the predicted designs generated by BEST as a starting point in our analysis. Figure A.3 shows the overall prediction method.

After BEST, functional operator energy estimation is performed. Functional operators used in chips typically consist of a few hundred gates. In semi–custom ASIC design, these are pre–synthesized as macro cells. Thus, we have an *a priori* knowledge of the internal

[1] This can be derived from a behavioral VHDL description.

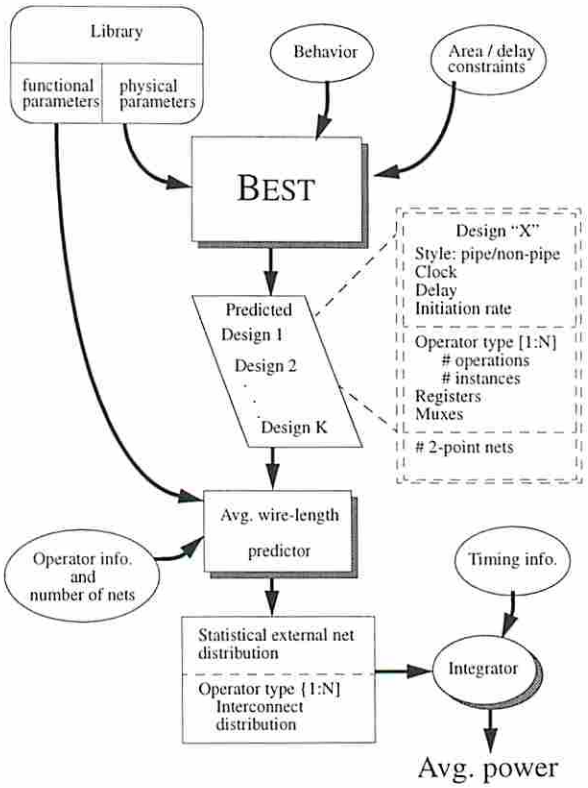


Figure A.3: System–level power prediction

structure of these cells. Specifically, we know the number of standard cells or gate array cells used, the number of internal nets and physical dimensions. For each macro cell, we predict energy consumption by predicting the internal wire–length distribution using Kurdahi's model [31] and the associated switching capacitance. Here *internal* implies the nets connecting the constituent transistors, cells or gates of the macro cell. In this novel technique, we model a functional unit as a collection of nets with varying switching activity and capacitances. Correlation of the latter two is illustrated in Section A.4.1.1. Accuracy of the results is improved by considering how frequently operators of each type are used in executing the algorithm.

Finally, energy consumed by the interconnect is estimated. As mentioned earlier, the number of external 2–point nets for a design is estimated by BEST. Here, *external* implies inter–functional–unit nets. In addition, from BEST we also know the number of operators of each type and their physical dimensions. Using this information and Kurdahi's model for wire–length distribution, we predict energy consumed by the external nets. The addition of all these components yields the total energy consumed by the chip for the predicted

design being considered. We also know the execution delay for this design thus, we compute the average power consumption.

Estimation of the average wire length inside a functional component as well as in the target chip is critical in our method because that is the only parameter that characterizes the geometric interconnect distribution. Given a predicted design, we determine the expected number of operators of each type and predict their interconnectivity. We use interconnect length analysis based on Rent's rule.

A.4 Statistical power estimation

In our analysis, the only distinction we make is between nets and modules. Therefore, in the mathematical model, functional units also include registers and multiplexers. A mathematical formulation for power consumption corresponding to a predicted design is given below.

A.4.1 Estimating power consumption of a functional unit

A.4.1.1 Switching nets model

Switching capacitances. We model any functional unit as a collection of wires and standard cells (or gates in a gate array design). The numbers of wires and cells in a given unit are fixed and known. Dimensions of the unit are also known. Assuming standard cell placement, the average length of a wire in the unit is given by Feuer's formula as implemented by Kurdahi [31]. The average load on a wire depends on the average input/output capacitance of the standard cells. Thus, the total switching capacitance associated with each wire consists of

1. average load capacitance,
2. average internal capacitance of the standard cells, and
3. capacitance of the wire.

Items 1 and 2 are obtained from the standard cell library while item 3 depends on the length of the wire. In the case of small functional units, comprised of a few standard cells (gates

in a gate array design), the interconnects between the cells are short and the switching capacitance is dominated by the transistor gate and source/drain capacitances. However, in large functional units, routing capacitance between cells is comparable to gate input/output capacitances and hence, cannot be ignored.

Switching activity. Our estimator starts with a behavioral description of the task and predicts power for all feasible designs. The procedure to estimate power for one functional unit must be fast because several designs are to be explored and each may consist of many functional units. In our approach, we make the following assumptions regarding the switching activity:

1. A good placement and routing algorithm is expected to place cells that share ports together. Thus, most short wires are likely to have high switching activity.
2. Many of the long wires carrying information such as clocks or multiplexer control are also likely to have high switching activity.
3. Medium length wires are likely to have relatively low switching activity.

A typical wire–length switching activity relation is shown in Figure A.4.

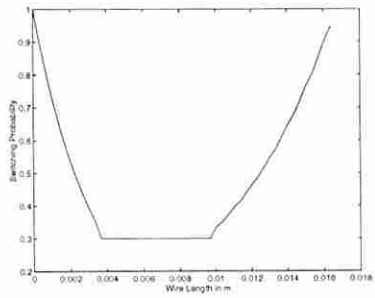


Figure A.4: Wire–length switching activity relation

Mathematical model. We have created an empirical model for switching activity, heavily influenced by the work of Vaishnav and Pedram [60], Su *et al.* [56] and Kojima *et al.* [29] at the RT (architecture) level. As described earlier, the switching activity – wire length relation consists of three regions. Region *A* is characterized by $0 \leq l \leq l_1$. Switching activity in this region is given by

$$\omega_l = \Omega_1 \ \alpha^{-\frac{l}{l}} \tag{A.1}$$

where Ω_1 = switching activity on short wires
(in transitions/ clock period),
$\bar{l}$ = average wire length,
$l_1 = \frac{1}{2}\,\bar{l}$, and
α is a constant > 0.

In region B, characterized by $l_1 < l \leq l_2$, the switching activity is constant, denoted Ω_m; where $l_2 = \frac{3}{2}\,\bar{l}$. Region C is characterized by $l_2 < l \leq l_{max}$. In this region, the switching activity is given by

$$\omega_l = \Omega_2\ \beta^{\frac{l-l_{max}}{l}} \tag{A.2}$$

where Ω_2 = Switching activity on very long wires,
$l_{max} = \frac{5}{2}\,\bar{l}$, and
β is a constant > 0.

From Eq. (A.1) and Eq. (A.2), α and β are computed as

$$\ln \alpha = 2 \ln \frac{\Omega_1}{\Omega_m} \tag{A.3}$$

$$\ln \beta = \ln \frac{\Omega_2}{\Omega_m} \tag{A.4}$$

At this point we do not know the impact of this coarse model of switching activity on power prediction accuracy. However, a more elaborate model could be used and the overall method for power prediction would still be valid. Exact methods to estimate switching activity at the chip or system level are not available. However, the parameters Ω_1, Ω_2, and Ω_m can be determined according to the overall resource usage. Analytical methods to determine different switching rates throughout the chip have been published [35]. Mehra and Rabaey used the white noise model for switching activity [39].

A.4.1.2 Mathematical model to predict functional unit power

A task implemented by a chip consists of one or more types of functions such as addition and multiplication. Let functional unit $\mathcal{F}_j$ used to implement function j consist of

$\mathcal{S}_j$ standard cells and $\mathcal{N}_j$ nets. The average length of a wire in this unit, given by Feuer's formula [12] is

$$\bar{l}_j = \sqrt{2}\frac{2r_m(3+2r_m)}{(1+2r_m)(2+2r_m)} \ \frac{\mathcal{S}_j^{r_m-\frac{1}{2}}}{(1+\mathcal{S}_j^{r_m-1})} \ \times \ \bar{l}_\sigma \tag{A.5}$$

r_m is the Rent's exponent for macro cells or functional units. Kurdahi and Liu *et al.* reported [31, 35] that r_m approximately equals 0.7. $\bar{l}_\sigma$ is the average standard cell length. Consider an event where the length of a randomly selected wire/net in this unit equals l. Kurdahi showed that the probability density function for this event is geometric and that the parameter of the function is the reciprocal of the average length. Thus, the total number of nets in $\mathcal{F}_j$ of length l is given by

$$N_j^l = \left(\frac{1}{\bar{l}_j}\right) \ \left(1-\frac{1}{\bar{l}_j}\right)^{l-1} \times \mathcal{N}_j \tag{A.6}$$

The average switching capacitance associated with a net of length l in $\mathcal{F}_j$ is given by

$$\mathcal{C}_j^l = \mu \cdot l \ + \ \mathcal{C}_\sigma \tag{A.7}$$

where

μ = Capacitance of the metal wires per unit length (including contacts), and

$\mathcal{C}_\sigma$ = Average switching capacitance associated with a standard cell.

As described earlier, switching activity on the nets is assumed to be a function of the net length. Total energy per cycle consumed by a functional unit $\mathcal{F}_j$ during one isolated use $E_{\mathcal{F}_j}$ is obtained by combining Equations (A.1), (A.2), (A.6) and (A.7).

$$\begin{aligned} E_{\mathcal{F}_j} = \frac{\mathrm{Vdd}^2}{2}\Big[\sum_0^{l_1} \mathcal{C}_j^l \, N_j^l \, \Omega_1 \alpha^{-\frac{l}{\bar{l}_j}} \ + \ \sum_{l_1}^{l_2} \mathcal{C}_j^l \, N_j^l \, \Omega_m \\ + \ \sum_{l_2}^{l_{max}} \mathcal{C}_j^l \, N_j^l \, \Omega_2 \beta^{\frac{l-l_{max}}{\bar{l}_j}}\Big] \times d_j \end{aligned} \tag{A.8}$$

where $l_1 = \frac{1}{2}\bar{l}_j$, $l_2 = \frac{3}{2}\bar{l}_j$, $l_{max} = \frac{5}{2}\bar{l}_j$, and d_j = Delay of $\mathcal{F}_j$ (in ns.).

Abstractly, the model computes average power consumption in a functional unit as a function of unit length, switching activity, number of wires, and delay of the functional unit.

A.4.2 Estimating power consumption for a design

A.4.2.1 Functional units

For a function type j, the number of functional units (or operators) required by design i, denoted O_j^i; and the average utilization of these functional units in a large number of executions of the task, denoted U_j^i; is estimated by BEST [26]. From Equation (A.8) the expression for the average power consumed by *all* functional units in a design i is written as

$$P_{\mathcal{F}}^i = \frac{\sum_j O_j^i \times U_j^i \times E_{\mathcal{F}_j}}{c} \tag{A.9}$$

where c is the clock period. Resource prediction allows nonpipelined and pipelined, and single as well as multi–cycle implementations.

A.4.2.2 External nets

Let the total number of external nets (nets extending outside a functional unit) for design i given by BEST be $\mathcal{N}^i$. The average length of external wires in this design given by Feuer's formula is

$$\bar{l}^i = \sqrt{2}\frac{2r_c(3+2r_c)}{(1+2r_c)(2+2r_c)} \quad \frac{\mathcal{O}^{ir_c-\frac{1}{2}}}{(1+\mathcal{O}^{ir_c-1})} \times \bar{l}_{\mathcal{F}^i} \tag{A.10}$$

where r_c = Rent's exponent for chips
(its value approximately equals 1.0).

$\mathcal{O}^i = \sum_j O_j^i$, is the number
of operators in design i

$\bar{l}_{\mathcal{F}^i} = \frac{\sum_j O_j^i \cdot (\mathcal{H}_j + \mathcal{L}_j)}{2\mathcal{O}^i}$, is the average length

of functional units in design i

$\mathcal{H}_j$ = Height of operators $\mathcal{F}_j$, and

$\mathcal{L}_j$ = Width of operators $\mathcal{F}_j$

Once again, using the geometric distribution, the total number of nets in design i of length l is

$$N^{l^i} = \left(\frac{1}{\bar{l}^i}\right)\left(1 - \frac{1}{\bar{l}^i}\right)^{l-1} \times \mathcal{N}^i \tag{A.11}$$

and, the average switching capacitance associated with a net of length l in the chip is

$$\mathcal{C}^{l^i} = \mu \cdot l \; + \; \mathcal{C}_\sigma \tag{A.12}$$

Net length – switching activity assumptions made in Section A.4.1.1 also hold at the chip level. Functional units exchanging data frequently are more likely to be placed in close proximity. In addition, very long wires carrying global control signals and clock have higher switching probabilities. Thus, the average power consumed by the switching activity on the external nets $P_{\mathcal{N}}{}^i$ is given by

$$\begin{aligned} P_{\mathcal{N}}{}^i = \frac{\mathrm{Vdd}^2}{2 \cdot c}\Big[\sum_{0}^{l_1} \mathcal{C}^{l^i} N^{l^i} \Omega_1 \alpha^{-\frac{l}{\bar{l}^i}} \; + \; \sum_{l_1}^{l_2} \mathcal{C}^{l^i} N^{l^i} \Omega_m \\ + \sum_{l_2}^{l_{max}} \mathcal{C}^{l^i} N^{l^i} \Omega_2 \beta^{\frac{l-l_{max}}{\bar{l}^i}}\Big] \end{aligned} \tag{A.13}$$

where $l_1 = \frac{1}{2}\bar{l}^i$, $l_2 = \frac{3}{2}\bar{l}^i$, and $l_{max} = \frac{5}{2}\bar{l}^i$.

Finally, the average chip power for design i is the sum of the total average functional power and average external net power given by Equations (A.9) and (A.13):

$$P^i_{chip} = P_{\mathcal{F}}{}^i \; + \; P_{\mathcal{N}}{}^i \tag{A.14}$$

A.4.3 The energy/power prediction algorithm

PREDICTPOWER

Use BEST to generate all prediction points

For each prediction point

/ Interconnect energy */*

Compute average width/length of functional operators

/ Dimensions are obtained from* PROMAN[2] **/*

Use Feuer's formula to compute

the average external wire–length.

Use geometric distribution and wire–length,

switching activity correlation to compute

the external net energy consumption.

For each functional unit

Use the number of constituent standard cells and

the number of nets obtained from PROMAN.

Compute the average wire–length using

the average standard cell width.

Use geometric distribution and wire–length,

switching activity correlation

to compute the internal net energy consumption.

Accumulate the internal net energy consumption.

Total energy ← external net energy +

accumulated internal net energy.

Average power ← total energy / delay of

the predicted design

A.5 Experimental verification

Energy and power consumption estimates were generated for the entire feasible design space of two commonly used signal processing applications namely, AR–FILTER and 1–D DCT. Three designs of AR–FILTER and two designs for DCT from this space, each with different execution delays, were selected for comparison with architecture–level predictions. Note that architectures of these three AR–FILTER implementations are completely different in terms of resource and interconnect requirement. The same observation also holds for the two DCT implementations. These designs were then synthesized to obtain

[2]Layout analysis tool from Cascade Design Automation.

RT–level net lists using the USC suite of tools [17]. Physical designs were generated for these net lists using Cascade Design Automation's tool EPOCH. A few commonly used macro cells such as Booth's multiplier, look–ahead and ripple–carry adders, and registers were pre–synthesized to be used in the synthesis of these data paths. Finally, power estimates for the synthesized designs were obtained from PROMAN^2. PROMAN employees a probabilistic prediction algorithm at the layout level. Although our predictor is well suited for many design styles, we chose standard cell design for the purpose of our experiments. The layout process used was MOSIS 1.2 μm. double–metal single–poly. Library information on standard cells and macro cells required by the power predictor is listed in Tables A.1 and A.2. The results of energy and power prediction using FREEDOM are shown in Table A.3 and Table A.4, respectively. In Table A.4, we also present the results of power prediction for these designs using PROMAN for the purpose of comparison.

Table A.1: Standard cell data

no. of gates	53
no. of inputs	206
Average width	42 μm
Average internal cap.	34 fF
Average gate input cap.	225 fF
Average output cap.	30 fF
Average total switching cap.	100 fF
Average transistor–gate pair (N/P) cap.	58 fF
Metal and Via cap.	0.22 fF/μm

Table A.2: Macro cell data (16–bit components)

Name	# std. cells	# nets	$\mathcal{H}$x$\mathcal{L}$ (μm x μm)	Delay (ns)
adder	48	97	811x231	15
subtracter	64	113	88x874	15
multiplier	1320	1225	991x1181	85
register	80	83	676x82	2
multiplexer	32	67	650x56	2

Table A.3: Experimental results (Estimates of energy consumption)

Algorithm	Delay (major cycles)	Predicted function energy (μJ)	Predicted external net energy (μJ)	Predicted total energy (μJ)
AR–FILTER	7	21.00	30.90	51.90
AR–FILTER	9	18.20	39.80	58.00
AR–FILTER	18	17.70	75.90	93.60
DCT	8	37.00	110	147
DCT	20	44.70	279	323.70

Table A.4: Experimental results (Estimates of power consumption)

Algorithm	Delay (major cycles)	Predicted power (mW)	PROMAN power (mW)	% difference in power
AR–FILTER	7	74.10	71.51	3.62
AR–FILTER	9	64.40	59.46	8.31
AR–FILTER	18	52.30	49.02	6.69
DCT	8	229	213	7.51
DCT	20	202	202	0

The 1–D DCT example is fairly large and has over 400 operations. Notice, that the design that requires 20 major cycles consumes significantly more energy than the one requiring 8 major cycles. Average power consumption in either case is comparable because additional functional operators in the faster design are replaced by several multiplexers and nets in the slower one. Figure A.5 shows how the estimates of function, interconnect, total energy and average power vary with the execution delay over the entire design space of 72 feasible implementations of DCT.

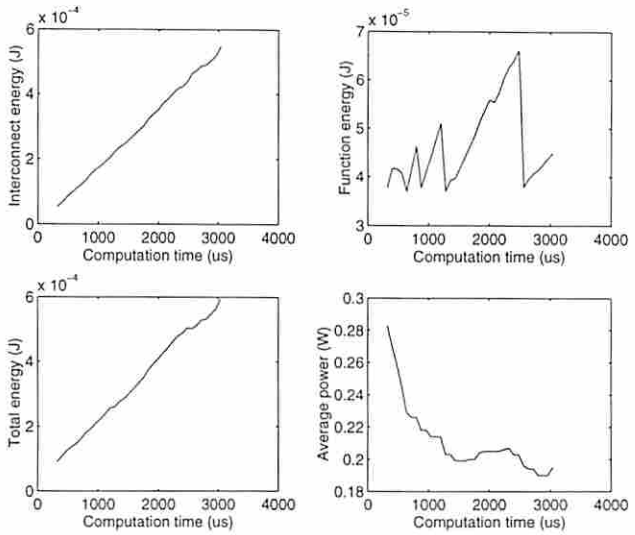


Figure A.5: Estimated DCT energy/power

A.6 Conclusions

We have presented a comprehensive system–level energy/power predictor that explores the vast design space prior to requiring the designer to make critical design choices. Results obtained are accurate yet the predictor is fast (In the 1–D DCT example, 72 distinct feasible architectures were analyzed in 600 milliseconds). This is achieved mainly due to statistical modeling of the target chip based only on the behavioral description of the task, a simplified model of switching activity, information about total size and delay of functional units, and predicted layout characteristics of the design. The FREEDOM predictor is designed to provide rough guidance to a designer, so that the design space can be quickly searched without producing unnecessary implementations which are far from the design goals and constraints. Thus, during the synthesis steps, the designer is now able to control the search space more efficiently such that circuits with desired cost, performance, battery life and thermal characteristics are synthesized. Results obtained from our predictor could also be used to partition a single task over one or more chips. This method allows us to associate transistor–level, technology and layout–dependent electrical characteristics of the circuit directly with its behavior. Therefore, not only is our predictor accurate at the behavioral–level, it can also be readily used with different technologies such as FPGA, by simply changing the technology parameters embedded in BEST and in our power predictor.

A.7 Low power design and word–length selection

Observe from Equation (A.10) that the average length of external wires in the chip is proportional to the height $\mathcal{H}$, and width $\mathcal{L}$ of functional resources. The energy consumption of external nets is proportional to their average length as seen from Equations (A.11), (A.12), and (A.13). Therefore, the power consumption of external nets is proportional to resource word lengths because height and width of resources are proportional to their word lengths. Moreover Equation (A.13) can directly model this relation if $\mathcal{H}$, and $\mathcal{L}$ are expressed as functions of word lengths. Similarly, the number of constituent cells and the number of internal nets are also proportional to word lengths of functional units. Then, from Equations (A.5), (A.6), (A.7), and (A.8) we state that energy/power consumption of resources is proportional to their word lengths. The number of constituent cells and the number of internal nets can be easily expressed in terms of in terms of word lengths. Now, Equation (A.8) can also be expressed as a function of word lengths. Finally, the expression for po wer consumption of a chip can be written as a function of resource word lengths. Then, word length selection can be used to minimize the energy/power function to obtain low power, low energy designs.

Reference list

[1] D. Anguita and B.A. Gomes. Mixing floating– and fixed–point formats for neural network learning on neuroprocessors. *Microprocessing and Microprogramming*, 41:757–769, June 1996.

[2] G. Bohlender. What do we need beyond IEEE arithmetic? In C. Ullrich, editor, *Computer arithmetic and self–validating numerical methods*, pages 1–32. Academic Press, 1990.

[3] D. Brand and C. Visweswariah. Inaccuracies in power estimation during logic synthesis. In *Proc. European Design Automation Conference (EURO-DAC)*, pages 388–394, November 1996.

[4] A. Brindle. *Genetic Algorithms for Function Optimization*. PhD thesis, University of Alberta, Edmonton, 1981.

[5] K. Chang and W.G. Bliss. Finite word length effects of pipelined recursive digital filters. *IEEE Transactions on Signal Processing*, 42:1983–1995, August 1994.

[6] S. Chaudhuri and R.A. Walker. Computing lower bounds on functional units before scheduling. In *Proc. International Workshop on High–Level Synthesis*, pages 36–41, 1994.

[7] W.H. Chen, C.H. Smith, and S.C. Fralick. A fast computational algorithm for the discrete cosine transform. *IEEE Transactions on Communications*, COM–25:1004–1009, September 1977.

[8] C.M. Chu and J.M. Rabaey. Hardware selection and clustering in the HYPER synthesis system. In *Proc. European Conference on Design Automation (EDAC)*, pages 176–180, 1992.

[9] B.Y. Chung, C. Chien, H. Samueli, and R. Jain. Performance analysis of an all–digital BPSK direct sequence spread–spectrum IF receiver architecture. *IEEE Journal on Selected Areas in Communications*, 11:1096–1107, September 1993.

[10] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms*. McGraw Hill Inc., 1991.

[11] G. De Micheli. *Synthesis and optimization of digital circuits*. McGraw Hill Inc., 1994.

[12] M. Feuer. Connectivity of random logic. *IEEE Transactions on Computers*, C-31(1):29–33, January 1982.

[13] M. Flynn. On division by functional iteration. *IEEE Transactions on Computers*, 19(8):702–706, August 1970.

[14] A. Ghosh, S. Devadas, K. Keutzer, and J. White. Estimation of average switching activity in combinational and sequential circuits. In *Proc. ACM/IEEE Design Automation Conference*, pages 253–259, 1992.

[15] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison–Wesley Pub. Co., 1989.

[16] A. Grzeszczak, M.K. Mandal, S. Panchanathan, and T. Yeap. VLSI implementation of discrete wavelet transform. *IEEE Transactions on VLSI Systems*, 4(4):421–433, December 1996.

[17] P. Gupta, C.T. Chen, J.C. DeSouza-Batista, and A.C. Parker. Experience with image compression chip design using **U**nified **S**ystem **C**onstruction tools. In *Proc. ACM/IEEE Design Automation Conference*, pages 250–256, June 1994.

[18] J.L. Holt and J-N. Hwang. Finite precision error analysis of neural network hardware implementations. *IEEE Transactions on Computers*, 42(3):281–290, March 1993.

[19] Y. Hu, A. Ghouse, and B.S. Carlson. Lower bounds on the iteration time and the number of resources for functional pipelined data flow graphs. In *Proc. International Conf. Computer Design (ICCD)*, pages 21–24, 1993.

[20] C. Huang, B. Zhang, A. Deng, and B. Swirski. The design and implementation of PowerMill. In *Proc. International Symposium on Low Power Design*, pages 105–110, 1995.

[21] R. Jain. MOSP: Module selection for pipelined designs with multi–cycle operations. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 212–215, 1990.

[22] R. Jain, A.C. Parker, and N. Park. Predicting system–level area and delay for pipelined and nonpipelined designs. *IEEE Transactions on Computer-Aided Design*, 11(8):955–965, August 1992.

[23] V.K. Jain, S.A. Wadekar, and L. Lin. Universal nonlinear component and its application to WSI. *IEEE Transactions on Components, Hybrids, and Manufacturing Technology*, 16:656–664, November 1993.

[24] Y. Jang and S.P. Kim. Block digital filter structures and their finite precision responses. *IEEE Transactions on Circuits and Systems – II: Analog and Digital Signal Processing*, 43(7):495–506, July 1996.

[25] K. A. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan, 1975.

[26] K. Küçükçakar. *System-Level Synthesis Techniques with Emphasis on Partitioning and Design Planning*. PhD thesis, University of Southern California, Los Angeles, CA, USA., 1991.

[27] V. Kantabutra. On hardware for computing exponential and trigonometric functions. *IEEE Transactions on Computers*, 45(3):328–339, March 1996.

[28] D.W. Knapp. Fasolt: A program for feedback-driven data-path optimization. *IEEE Transactions on Computer-Aided Design*, 11(6):677–695, June 1992.

[29] H. Kojima, D.J. Gorny, K. Nitta, and K. Sasaki. Power analysis of a programmable DSP for architectural/program optimization. In *Proc. IEEE Symposium on Low Power Electronics*, pages 26–27, 1995.

[30] K. Kota and J.R. Cavallaro. Numerical accuracy and hardware tradeoffs for CORDIC arithmetic for special–purpose processors. *IEEE Transactions on Computers*, 42(7):769–779, July 1993.

[31] F.J. Kurdahi and A.C. Parker. Techniques for area estimation of VLSI layouts. *IEEE Transactions on Computer-Aided Design*, 8(1):81–92, January 1989.

[32] P.E. Landman and J.M. Rabaey. Activity-sensitive architectural power analysis. *IEEE Transactions on Computer-Aided Design*, 15(6):571–598, June 1996.

[33] C.H. Lee, M. Kawamata, and T. Higuchi. State–space approach to roundoff error analysis of fractal image coding. *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences*, E80A:159–165, January 1997.

[34] F.L. Lewis. *Optimal estimation with an introduction to stochastic control theory*. John Wiley & Sons, 1986.

[35] D. Liu and C. Svensson. Power consumption estimation in CMOS VLSI chips. *IEEE Journal of Solid-State Circuits*, 29(6):663–670, June 1994.

[36] K. Liu, R.E. Skelton, and K. Grigoriadis. Optimal controllers for finite wordlength implementation. *IEEE Transactions on Automatic Control*, 37(9):1294–1304, September 1992.

[37] D. G. Luenberger. *Introduction to linear and nonlinear programming*. Addison–Wesley Pub. Co., 1974.

[38] R. Marculescu, D. Marculescu, and M. Pedram. Switching activity analysis considering spatiotemporal correlations. In *Proc. ACM/IEEE Design Automation Conference*, pages 294–299, 1994.

[39] R. Mehra and J. Rabaey. Behavioral level power estimation and exploration. In *Proc. First International Workshop on Low Power Design*, pages 197–202, April 1994.

[40] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer–Verlag, second, extended edition, 1994.

[41] D. Michelucci and J.-M. Moreau. Lazy arithmetic. *IEEE Transactions on Computers*, 46(9):961–975, September 1997.

[42] S.H. Mullins, W.W. Charlesworth, and D.C. Anderson. A new method for solving mixed sets of equality and inequality constraints. *Journal of Mechanical Design*, 117:322–328, June 1995.

[43] F.N. Najm. Transition density, a stochastic measure of activity in digital circuits. In *Proc. ACM/IEEE Design Automation Conference*, pages 644–649, 1991.

[44] S.Y. Ohm, F.J. Kurdahi, and N. Dutt. Comprehensive lower bound estimation from behavioral descriptions. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 182–186, 1994.

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

[46] P. Paulin and J. Knight. Force–directed scheduling for the behavioral synthesis of ASIC's. *IEEE Transactions on Computer-Aided Design*, CAD–8(6):661–679, July 1989.

[47] M. Potkonjak, K. Kim, and R. Karri. Methodology for behavioral synthesis–based algorithm–level design space exploration: Dct case study. In *Proc. ACM/IEEE Design Automation Conference*, pages 252–257, June 1997.

[48] R. Salcedo, M.J. Gonçalves, and S. Feyo De Azevedo. An improved random–search algorithm for non–linear optimization. *Computers and Chemical Engineering*, 14:1111–1126, October 1990.

[49] J. Schaffer, R. Caruana, L. Eshelman, and R. Das. A study of control parameters affecting online performance of genetic algorithms for function optimization. In *Proc. Third International Conf. on Genetic Algorithms*, 1989.

[50] M.J. Schulte and E.E. Swartzlander, Jr. Hardware designs for exactly rounded elementary functions. *IEEE Transactions on Computers*, 43(8):964–973, August 1994.

[51] M.J. Schulte and E.E. Swartzlander, Jr. A variable-precision, interval arithmetic processor. *Zeitschrift fur Angewandte Mathematik und Mechanik*, 76, suppl. 1:527–528, 1996.

[52] A.B. Sesay and M. Patton. QR–decomposition decision feedback equalisation and finite–precision results. *IEEE Proceedings–F*, 140(2):89–97, April 1993.

[53] N.R. Shanbhag and K.K Parhi. Finite–precision analysis of the pipelined ADPCM coder. *IEEE Transactions on Circuits and Systems – II: Analog and Digital Signal Processing*, 41(5):364–368, May 1994.

[54] D. Springer and D. Thomas. Exploiting the special structure of conflict and compatibility graphs in high–level synthesis. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 254–259, 1990.

[55] D. Springer and D. Thomas. New methods for coloring and clique partitioning in data path allocation. *Integration–The VLSI Journal*, 12:267–292, December 1991.

[56] C-L Su, C-Y Tsui, and A.M. Despain. Low power architecture design and compilation techniques for high-performance processors. Technical Report 94-01, Computer Engineering Division, EE-Systems, USC, Los Angeles, CA, 1994.

[57] W. Sung and Ki-II Kum. Simulation–based word-length optimization method for fixed–point digital signal processing systems. *IEEE Transactions on Signal Processing*, 43(12):3087–3090, December 1995.

[58] A.G. Tsirukis and G.V. Reklaitis. Application of generalized Hopfield networks to discrete nonlinear optimization problems. *Computers and Chemical Engineering*, 18:459–468, May 1994.

[59] S. Uramoto, Y. Inoue, A. Takabatake, J. Takeda, Y. Yamashita, H. Terane, and M. Yoshimoto. A 100–MHz 2–D discrete cosine transform core processor. *IEEE Journal of Solid-State Circuits*, 27(4):492–499, April 1992.

[60] H. Vaishnav and M. Pedram. PCUBE: a performance driven placement algorithm for low power design. In *Proc. European Design Automation Conference (EURO-DAC)*, pages 72–77, 1993.

[61] S. A. Wadekar and A. C. Parker. Accuracy sensitive word–length selection for algorithm optimization. In *Proc. International Conf. on Computer Design (ICCD), (To appear)*, October 1998.

[62] S. A. Wadekar, A.C. Parker, and C.P. Ravikumar. FREEDOM: Statistical behavioral estimation of system energy and power. In *Proc. Eleventh International Conference on VLSI Design*, pages 30–36, January 1998.

[63] W.F. Wong and E. Goto. Fast hardware–based algorithms for elementary function computations using rectangular multipliers. *IEEE Transactions on Computers*, 43(3):278–294, March 1994.

[64] II D. Yun and S. Uk. Lee. On the fixed–point error analysis of several fast IDCT algorithms. *IEEE Transactions on Circuits and Systems – II: Analog and Digital Signal Processing*, 42(11):685–693, November 1995.

[65] B. Zeng and Y. Neuvo. Analysis of floating point roundoff errors using dummy multiplier coefficient sensitivities. *IEEE Transactions on Circuits and Systems*, 38:590–601, June 1991.