UNIVERSITY OF CALIFORNIA

Santa Barbara

Optimization Techniques for Arithmetic Expressions

A Dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Computer Engineering

by

Anup Hosangadi

Committee in charge:

Professor Ryan Kastner, Chair

Professor Forrest Brewer

Professor Malgorzata Marek-Sadowska

Dr. Farzan Fallah

July 2006

The dissertation of Anup Hosangadi is approved.

_____
Forrest Brewer

_____
Farzan Fallah

_____
Malgorzata Marek-Sadowska

_____
Ryan Kastner Committee Chair

July 2006

Optimization Techniques for Arithmetic Expressions

by

Anup Hosangadi

ACKNOWLEDGEMENTS

I would like to thank all those people that helped me through this long journey in graduate school and made it an enjoyable experience. I would like to thank my advisors Professor Ryan Kastner and Dr. Farzan Fallah for their guidance and admirable patience in helping me with my thesis, without which this would not be possible. I would like to thank Professor Forrest Brewer and Professor Marek-Sadowska for their useful suggestions and ideas during my research. I enjoyed working with and collaborating with my lab mates Gang Wang, Shahnam Mirzaei, Wenrui Gong, Arash Arfaee and Yan Meng, who always created an atmosphere conducive to exchange of ideas.

Being a foreign student is tough and I can never thank enough the people who made me feel at home away from home. I am grateful to the folks at the International Student Association and the India Student Association. Last but not the least I am forever indebted to my family, who continually supported and encouraged me during my studies.

Sincerely,
Anup Hosangadi

VITA OF ANUP HOSANGADI

July 2006

EDUCATION

Bachelor of Engineering in Electrical and Electronics Engineering, National Institute of Technology, Trichy, India, May 2001
Master of Science in Computer Engineering, University of California, Santa Barbara, September 2003
Doctor of Philosophy in Computer Engineering, University of California, Santa Barbara, July 2006 (expected)

PROFESSIONAL EMPLOYMENT

September 2001- December 2002, Teaching Assistant, Department of Electrical and Computer Engineering, University of California, Santa Barbara
January 2003-March 2006, Research Assistant, Department of Electrical and Computer Engineering, University of California, Santa Barbara
June 2004-September 2004, Student Intern, Fujitsu Laboratories of America, Sunnyvale, California.
April 2006-June 2006, Teaching Assistant, Department of Electrical and Computer Engineering, University of California, Santa Barbara

PUBLICATIONS

A.Hosangadi, F.Fallah, R.Kastner, "*Algebraic Methods for Optimizing Constant Multiplications in Linear Systems*", to appear in Journal of VLSI Signal Processing.

A.Hosangadi, F.Fallah, R.Kastner, "*Optimizing Polynomial Expressions by Algebraic Factorization and Common Subexpression Elimination*", to appear in the IEEE Transactions on Computer Aided Design (TCAD).

A.Hosangadi, F.Fallah, R.Kastner, " Area and Delay Optimization of Carry Save Arithmetic Structures", submitted to the IEEE Transactions on Computer Aided Design (TCAD).

A.Hosangadi, F.Fallah, R.Kastner, "*Optimizing High Speed Arithmetic Circuits Using Three Term Extraction*", in proceedings of the Design Automation and Test in Europe (DATE), Munich, Germany, March 2006

S.Mirzaei, A.Hosangadi, R.Kastner, "*Implementing High Speed FIR Filters Using Add and Shift Method*", poster presentation at the International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, February 2006.

A.Hosangadi, F.Fallah, R.Kastner, "*Simultaneous Optimization of Delay and Number of Operations in Multiplierless Implementation of Linear Systems* in proceedings of the 14th International Workshop on Logic and Synthesis (IWLS), Lake Arrowhead, CA, June 2005.

A.Hosangadi, F.Fallah, R.Kastner, "*Reducing Hardware Complexity of Linear DSP Systems by Iteratively Eliminating Two Term Common Subexpressions*". proceedings of the Asia South Pacific Design Automation Conference (ASP-DAC), Shanghai, China, January 2005

A.Hosangadi, F.Fallah, R.Kastner, "*Energy Efficient Hardware Synthesis of Polynomial expressions*", proceedings of the International Conference on VLSI Design, Kolkata, India, January 2005 (**Best Student Paper Award**).

A.Hosangadi, F.Fallah, R.Kastner, "*Factoring and Eliminating Common Subexpressions in Polynomial Expressions*", proceedings of the International Conference on Computer Aided Design (ICCAD), San Jose, November 2004.

A.Hosangadi, F.Fallah, R.Kastner, "*Common Subexpression Elimination Involving Multiple Variables for Linear DSP Synthesis*", proceedings of the IEEE International Conference on Application-Specific Architectures and Processors (ASAP), Galveston TX, September 2004.

A.Hosangadi, F.Fallah, R.Kastner, "*Optimizing Polynomial Expressions by Factoring and Eliminating Common Subexpressions*", 13th International Workshop on Logic and Synthesis (IWLS), Temecula CA, June 2004.

"*Optimization Techniques for Arithmetic Expressions*", 8th SIGDA PhD Forum, Design Automation Conference (DAC), Anaheim, CA, June 2005 (poster presentation)

AWARDS

Won the N.N. Biswas Best Student Paper Award at the International Conference of VLSI Design, Kolkata, India in 2005 for the paper titled "Energy Aware Hardware Synthesis of Polynomial Expressions", co-authored by Dr. Ryan Kastner and Dr. Farzan Fallah.

Recepient of SIGDA travel grants to attend conferences in Shanghai, China (ASPDAC'2005) and Munich, Germany (DATE'2006)

Received financial support throughout graduate study at the University of California, Santa Barbara

ABSTRACT

Optimizations for Arithmetic Expressions

by

Anup Hosangadi

The increasing complexity of systems has spawned new directions in Design Automation research. System designers need the most sophisticated tools that can design applications of ever increasing complexity and meet the stringent requirements of performance and power consumption. Most of the popular embedded system applications perform some kind of continuous numeric processing which are computationally intensive. Examples of such computing can be found in multimedia applications such as 3-D computer graphics, audio and video processing and wireless communication applications. Traditionally system designers have often used hand written library routines for implementing these functions in software and custom Intellectual Property (IP) cores for hardware. Such an approach often does not give the best results since the results depend on the target architectures and the implementation constraints that are not taken into account in these libraries. Unfortunately, the currently available software compilers and high level synthesis frameworks have been designed for general purpose applications and do not do a

good optimization of arithmetic expressions. As a result, there is an urgent need to come up with good tools that can take advantage of the flexibility of arithmetic expressions and provide the best solutions for their implementations. These methods then need to be integrated into the conventional optimization frameworks for software and/or hardware.

This thesis highlights the opportunities available for the optimization of arithmetic expressions, and presents algebraic techniques for their optimization. The heart of these techniques is based on a novel canonical representation and methodology for eliminating redundant operations in arithmetic expressions. These techniques were applied for optimizing arithmetic functions in a number of popular applications, where significant performance, power and area savings were observed over the conventional techniques. This thesis also highlights the various factors that affect the delay and power consumption of these arithmetic expressions, and discusses some solutions that give the best result. Furthermore, the challenges of verifying the precision and range of these computations and their relationship with the various optimizations are studied. Finally, we present the challenges and suggest some solutions for integrating these optimizations into conventional optimizing compilers.

TABLE OF CONTENTS

LIST OF FIGURES

1 Introduction

The past few years have seen an explosive growth in the market for embedded systems, mainly in the consumer electronics segment. The increasing trend towards high performance and portable systems has forced researchers to come up with innovative system level design techniques that can achieve these objectives and meet the strict time to market requirements. Most of these systems perform some kind of continuous processing of input data, which requires the extensive evaluation of arithmetic expressions. Though researchers have come up with many innovative architectural features for evaluating these expressions, there are not many tools that optimize these expressions at the behavioral level.

This work looks at the various issues in the evaluation of arithmetic expressions and describes our novel algebraic optimization techniques. The arithmetic expressions that are discussed include polynomial expressions commonly used to model objects in 3-D computer graphics applications, linear arithmetic expressions involving constant multiplications found in DSP transforms and filters, and integer exponentiations which are performed in many cryptographic systems. The conventional optimization routines performed in optimizing compilers such as the two term Common Subexpression Elimination (CSE), Value numbering and Partial Redundancy Elimination (PRE), have been designed for general purpose applications, and perform poorly when optimizing an application with a number of arithmetic expressions. One of the main reasons for this is that these techniques do

not take into account the commutative, associative and distributive properties of arithmetic operators. The algebraic techniques that are discussed in this thesis make use of these properties and also make use of a canonical representation that helps us to perform a number of useful optimizations that cannot otherwise be performed by these techniques. An example of the optimizations is the algebraic factorization of polynomial expressions and eliminating common subexpressions. It is shown that any common subexpression or factorization opportunity can be detected by our methods.

It is hoped that our methods can be leveraged by compiler designers and system designers for producing superior results, and that future researchers can work on some of these open problems which will make these techniques more easily usable in a wide variety of applications and compilation frameworks. The thesis is organized carefully into a number of chapters.

Chapter 1 gives an introduction to the topic and the organization of the thesis.

Chapter 2 talks about the various examples of arithmetic expressions that are found in consumer applications and forms the motivation for the thesis work. The design flow for computation intensive arithmetic expressions is described to point out the various opportunities where our techniques are applicable.

Chapter 3 describes the core algebraic techniques on which our optimizations are based. It starts with the canonical representation of arithmetic expressions and describes the algorithms for kernel generation and kernel intersection. The algorithms for optimizing polynomial expressions by algebraic factorization and common subexpression elimination are presented. This is followed by an algorithm

for reducing the number of two-input adders for linear systems with constant multiplications. Finally, the chapter describes a method for finding optimal solutions to these problems using 0-1 Integer Linear Programming.

Chapter 4 describes work related to the material presented in this thesis. It first describes the different dataflow optimizations done in software compilers such as common subexpression elimination and value numbering. This is followed by related work done in the hardware synthesis of arithmetic computations.

Chapter 5 talks about some interesting problems in hardware synthesis of arithmetic expressions. It starts with an application of the algorithms described in Chapter 3 to design an area efficient, high speed Finite Impulse Response (FIR) filter. Some optimizations in the synthesis of high speed arithmetic expressions are then described. Finally some delay aware optimization techniques are presented.

Chapter 6 describes the important problem of verifying that the arithmetic expressions satisfy the constraints of precision imposed by the application and that no overflows or underflows occur. A survey of some of the approaches used in the industry is presented, and some static error analysis techniques developed in academia is presented.

2 Arithmetic expressions in Consumer Applications

This chapter points out the various consumer applications which need to constantly evaluate arithmetic expressions. This chapter serves to form the motivation for our work. The examples that are covered are polynomial expressions in Computer Graphics and Signal Processing, matrix multiplications and filters in Digital Signal Processing (DSP) and communications and integer exponentiations in cryptographic applications. These examples are a good representative of the most popular consumer applications that are driving the embedded systems market. The chapter is concluded with a discussion of the design flow for computationally intensive embedded systems. The position of our techniques in the design flow is also discussed.

2.1 Polynomial expressions in Computer Graphics applications

3-Dimensional Computer graphics are commonly used in applications such as video games, animation movies, scientific modeling, medical procedures and many more. These applications are generally computationally very expensive, but due to the increasing demands of consumers and improvements in technology, they are increasingly being integrated into current generation embedded devices. Optimizing these programs is thus becoming of utmost importance.

In 3-D graphics, polynomials are often used to model surfaces, curves and textures. These polynomials have to be computed for each pixel of the image. As an example, consider the quartic spline polynomial used in the modeling of textures and surfaces and is shown in Figure 2.1. The original polynomial expression has 23 multiplications and four additions.

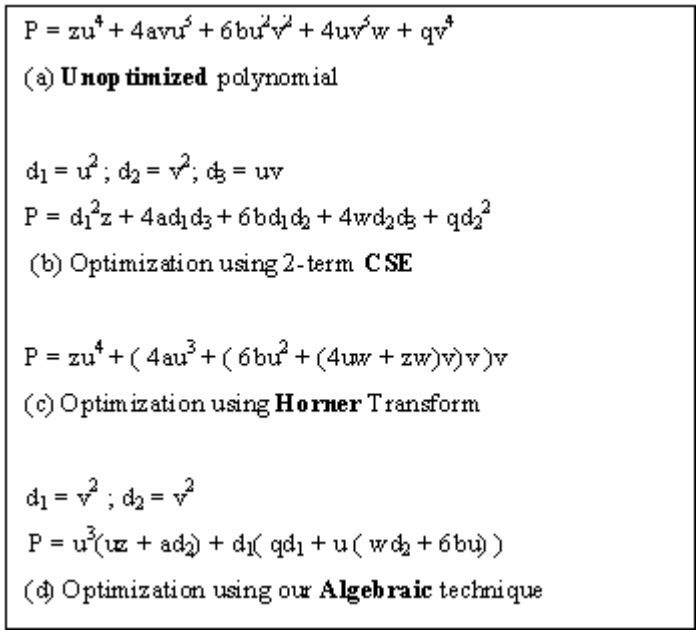


Figure 2.1 The Quartic Spline polynomial and its different optimizations

The Figure 2.1b shows the polynomial after performing the traditional 2-term Common Subexpression Elimination (CSE). This is an iterative algorithm where two term common subexpressions are extracted and eliminated, and the code is rewritten in each iteration. The result of the CSE algorithm depends on the ordering of the operations in the parse tree of the expression, and in this example the order that produces the least number of operations for the CSE algorithm is used. Figure 2.1b

shows the expression after the application of the Horner transform obtained from Maple. The Horner form converts a polynomial into a nested sequence of multiplications and additions, which is very useful for evaluation with Multiply Accumulate (MAC) operations, and is a popular form for evaluating many polynomials in signal processing libraries. The Horner form produces a good form for a univariate polynomial such as the Taylor series form for Sin(x), but it does not do as well for multivariate polynomials of the form shown in the Figure 2.1c. Figure 2.1d shows the implementation of the polynomial using the algebraic techniques discussed in this thesis. This implementation produces three fewer multiplications compared to the CSE algorithm and four fewer multiplications compared to the Horner form. These algebraic techniques can optimize more than one expression at a time, and can optimize expressions consisting of any number of variables. Other attractive features of this algorithm are the canonical form for representing the expressions and the theorem that assures the detection of any common subexpression or algebraic factorization. The details of the representations and the algorithms are presented in Chapter 3.

2.2 Polynomial expressions in Signal Processing Transforms

Signal processing applications are full of arithmetic computations. Evaluation of trigonometric functions represents one of the most common cases of computing polynomial expressions. Trigonometric functions are typically approximated by their Taylor Series expansions. For example, Sin(x) can be approximated by the equation

$$Sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!}$$

Equation 2-1

Even though these expressions can be optimized using the algebraic techniques that are presented in this thesis, computing these expressions on the fly is still very expensive. Fortunately, in most cases the arguments to these functions are known beforehand, and they can be precomputed and stored in the library. In cases, where these arguments are not known beforehand or the memory size is limited, these expressions can be optimized automatically using our methods.

The computation of DSP transforms such as the Discrete Cosine Transform (DCT) basically involves matrix multiplication between a constant matrix (M) and a vector of input samples (X). The outputs (Y) can be viewed as a set of polynomial expressions, to which our algebraic techniques can be applied. For example consider the Constant matrix for a 4 point DCT shown in Figure 2.2. The matrix multiplication with a vector of input samples is shown in Figure 2.3. In the figure A, B, C and D can be viewed as distinct constants. A trivial computation of this matrix multiplication requires 16 multiplications and 12 additions/subtractions. But by extracting common factors, these expressions can be rewritten as shown in Figure 2.4. This implementation is cheaper than the original implementation by six multiplications and four additions/subtractions. These optimizations are typically done manually in hand-coded signal processing libraries, but our method can extract these common factors and common subexpressions automatically.

$$C = \begin{pmatrix} \cos(0) & \cos(0) & \cos(0) & \cos(0) \\ \cos(\Pi/8) & \cos(3\Pi/8) & \cos(5\Pi/8) & \cos(7\Pi/8) \\ \cos(\Pi/4) & \cos(3\Pi/4) & \cos(5\Pi/4) & \cos(7\Pi/4) \\ \cos(3\Pi/8) & \cos(7\Pi/8) & \cos(\Pi/8) & \cos(5\Pi/8) \end{pmatrix}$$

Figure 2.2 Constant matrix for a 4-point DCT

$$\begin{pmatrix} y0 \\ y1 \\ y2 \\ y3 \end{pmatrix} = \begin{pmatrix} D & D & D & D \\ A & B & -B & -A \\ C & -C & -C & C \\ B & -A & A & -B \end{pmatrix} * \begin{pmatrix} x0 \\ x1 \\ x2 \\ x3 \end{pmatrix}$$

Figure 2.3 Constant matrix multiplication

$$d_1 = (x0 - x3) \qquad d_2 = (x1 + x2)$$
$$d_3 = (x1 - x2) \qquad d_4 = (x0 - x3)$$

$$y0 = D*[d_1 + d_2]$$
$$y1 = A*d_4 + B*d_3$$
$$y2 = C*[d_1 - d_2]$$
$$y3 = B*d_4 - A*d_3$$

Figure 2.4 Matrix multiplication after eliminating common subexpressions

2.3 Constant multiplications in Signal processing and Communications applications

Signal processing applications typically consist of a number of cases where multiplications with constant matrices have to be performed. Examples include the signal processing transforms such as the Discrete Cosine Transform (DCT) which is commonly used in many signal compressions algorithms such as image and speech processing. The H.264 is a digital video codec standard which is noted for achieving very high data compression. The integer transform for the video encoding is shown in Figure 2.5. The transform after eliminating common subexpressions is shown in Figure 2.6.

$$\begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{bmatrix}$$

Figure 2.5 H.264 Integer transform computation

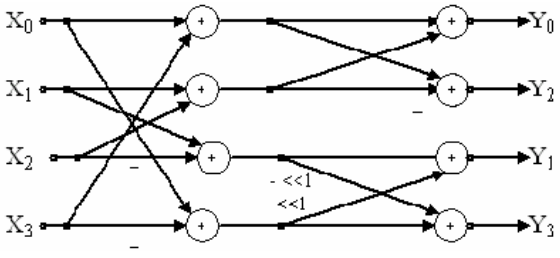


Figure 2.6 H.264 Integer transform after extracting common subexpressions

Finite impulse Response (FIR) filters are used in any application that involves transmission and reception of signals, where it is used to filter out the unwanted noise components. The digital filter computation for a filter of N taps can

be represented by the inner product computation shown in Equation 2-2. where {h(k)} represents the filter coefficients, x(n-k) represents the input signal at time n-k and y(n) represents the filter output at time n.

$$y(n) = \sum_{k=0}^{N-1} h(k) * x(n-k)$$

Equation 2-2

There are therefore a number of applications which involve performing constant multiplications. Multipliers are expensive in terms of area, delay and power consumption, and when considering hardware implementations, the full flexibility of a general purpose multiplier is not required and we can implement the constant multiplications using a sequence of additions and shift operations. The complexity of the constant multiplication is then directly related to the number of additions that are required. There are various techniques to reduce the number of additions, such as using signed digit representations for the constants, and eliminating common terms. Almost all the techniques for redundancy elimination are based on finding common digit patterns in the set of constants multiplying a single variable. Our algebraic techniques can find common subexpressions among multiple variables, and to the best of our knowledge is the only method that can do so. As an illustration, the H.264 transform as shown in Figure 2.5, can be optimized using our algebraic techniques which reduce the number of additions/subtractions by four. This is the actual hardware implementation used for the transform [1], which is obtained automatically by our method.

2.4 Modular exponentiation in Cryptographic systems

Cryptographic applications such as the RSA cryptosystem involve a lot of arithmetic computation in the message encryption and decryption. The system is based on a public key pair (n,e) and a private key pair (n.d), where n is the produce of two large prime numbers (n = p*q). When a message m has to be transmitted, it is encrypted using the public key of the recipient as $c = m^e \bmod n$. The original message is decrypted at the receiver using his private key as $m = c^d \bmod n$. These operations are very expensive as the sizes of the modulus and the exponents are of the order of 1024 bits in recent implementations [2]. Therefore it is important to come up with techniques that can optimize these computations.

The most popular method for performing the exponentiation is using the method of squaring which involves an order of N^2 multiplications for an exponent of size N bits, though there are some methods which use the concept of addition chains [3] which claim to reduce the number of multiplications on an average by 22%. The number of multiplications can also be reduced by finding common bit patterns in the exponent. For example consider the exponentiation $F = a^{21845}$. Now $2925 = (101101101101)_2$. Using the method of squaring, a^{21845} can be computed as shown in Figure 2.7a. By eliminating the common digit pattern 101 in the exponent P, it can be rewritten as $P = d_1 + d_1<<3 + d_1<<6 + d_1<<9$ (The << represents the left shift operator). By further extracting the common part $d_2 = d_1 + d_1<<3$, P can be rewritten as $P = d_2 + d_2<<6$. Using this property, the integer exponentiation is rewritten as shown in Figure 2.7b. This takes three fewer multiplications than the method of

squaring. Therefore it can be seen that further reduction in the number of operations over conventional methods is possible by extracting and eliminating redundant operations.

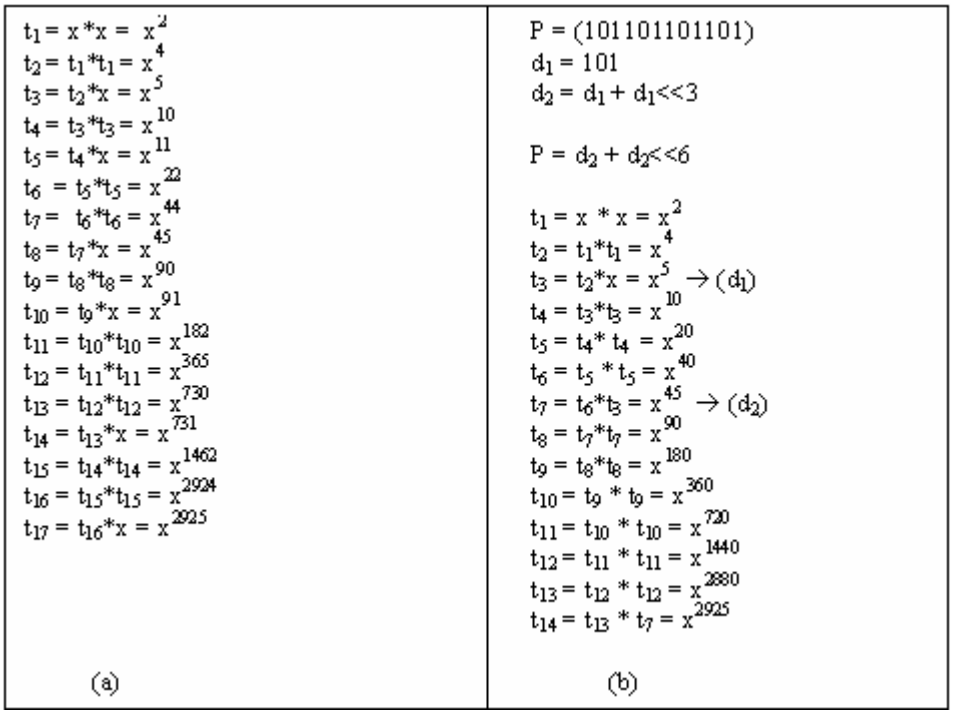


Figure 2.7 Modular exponentiation a) Using the method of squaring b) Eliminating common computations

2.5 Address calculation in data intensive applications

Data transfer intensive applications such as image/video coding, mobile telephony etc consist of a number of memory accesses which involves considerable arithmetic for the computation and selection of the different memory access pointers. Consider an indexed linear array of elements ($A[I_1][I_2]$ ….. $[I_n]$), with the maximal

number of array entries in each dimension represented by S_i (for each dimension i). Assume a row-major type organization of the array A. The address expression for an element of the array therefore becomes

AE = (…. $(I_1*S_2 + I_2)*S_3 + I_3)*S_4$ + ….. $)S_n + I_n$. With statically allocated memories, the values of the array sizes $\{S_i\}$ are known and the address expression becomes an affine form of the type AE = $C_1*I_1 + C_2*I_2 + C_3*I_3$ + …. I_n. These computations are very often on the critical paths of the loops, and are good candidates for optimization. The work in [4] addressed this issue and presented various algebraic transformations for optimizing these address calculations. The authors targeted the hardware synthesis of Custom Address Calculation units (ACUs). There were a number of transformations that were investigated including splitting and clustering the address expression, induction variable analysis and global algebraic transformations such as Common Subexpression Elimination (CSE) and constant propagation.

2.6 Arithmetic expressions optimization in embedded system design flow

Figure 2.8 shows the design flow for computationally intensive embedded system applications. The application is given in terms of a mathematical description or as source code. In addition the constraints of the application which include the available resources, desired timing, error tolerance, maximum area and power consumption are given. The computational aspect of the application is then analyzed

and an appropriate algorithm is selected. For signal processing applications, this is equivalent to choosing the appropriate transform for the application. For computer graphics applications, the polynomial models for the surfaces, curves and textures are chosen. An important step is in the conversion of floating point representation to fixed point representation. Though floating point provides greater dynamic range and precision than fixed point, it is far more expensive to compute. Most embedded system applications tolerate a certain degree of inaccuracy, and prefer the much simpler fixed point notation. As a matter of fact, 95% of the embedded system industry uses fixed point arithmetic and only 5% use floating point (source Texas Instruments). The conversion of infinite precision floating point to finite precision fixed point [5] produces some errors. These errors have to be analyzed [6] [7]to see if they are within the tolerated error limits.

An automatic hardware software partitioning tool is then used to determine which parts of the system specification should be mapped onto hardware and which parts should be mapped to software. For arithmetic intensive applications with tight timing constraints, the computation intensive kernels are often implemented in hardware. After this decision the architecture of the system and the memory hierarchy is decided. The custom hardware portions of the system are then designed by means of a behavioral description of the algorithm using a hardware description language (HDL). These hardware descriptions are synthesized into a Register Transfer Language (RTL) by means of powerful synthesis tools, with the given constraints on timing, area and power consumption. These synthesis tools mainly perform scheduling, resource allocation and binding of the various operations [8]

obtained from an intermediate representation of the behavior represented in the hardware description language. In addition the tools perform certain optimizations such as redundancy elimination (common subexpression elimination and value numbering) and critical path minimization. The constant multiplications in the arithmetic descriptions can be decomposed into shifts and additions, and the resulting complexity can be further reduced by eliminating common subexpressions [9-12]. Furthermore, there are some numeric transformations of the constant coefficients that can be applied to linear transforms to reduce the strength of the operations [13, 14]. The order and priorities of the various optimizations and transformations is largely application dependant and is a subject of current research. In most cases, this is done by evaluating a number of transformations and selecting the one that meets the constraints the best [15]. The RTL is then synthesized into a gate level netlist, which is then placed and routed using standard physical design tools.

For the software portion of the design custom instructions tuned to the particular application may be added [16-18]. For certain computation intensive kernels of the application, having platform dependant software might be important to achieve the best performance on the available architecture. This is often done manually by selecting the relevant functions from optimized software libraries. For signal processing applications, certain automatic library generators are available [15]. The software is then compiled by compilers using various transformations and optimization techniques [19]. These compiler optimizations however do limited transformations for reducing complexity of arithmetic expressions. For some

applications, the generated assembly code is optimized (mostly manually) to improve performance, though it is not practical for large and complex programs. An assembler and linker is then used to generate the executable file.

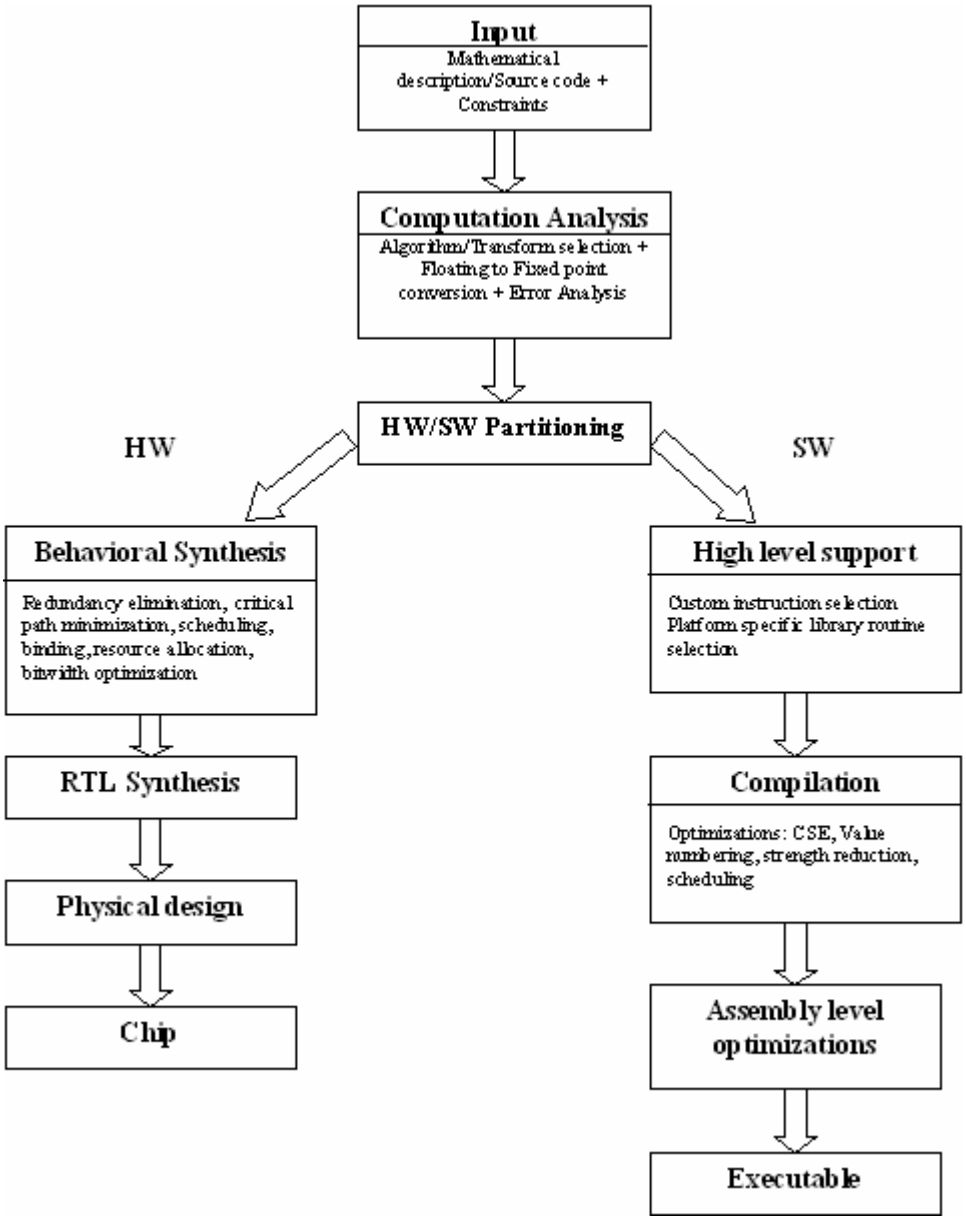


Figure 2.8 Embedded system design flow

3 Algebraic methods for redundancy elimination

This chapter presents algebraic techniques for the redundancy elimination in arithmetic expressions. These techniques form the core of our optimization algorithms. The main strength of our technique is the canonical representation of arithmetic expressions that enables the detection of all possible common subexpressions and algebraic factorizations. Similar techniques have been developed for multi-level logic synthesis for reducing Boolean expressions. Since Boolean operators (AND, OR) share properties of commutativity, associativity and distributivity with the arithmetic operators (ADD, MUL), some of these techniques can also be applied to arithmetic expressions.

Firstly, the technique for optimizing polynomial expressions consisting of ADD, SUB and MUL operators is presented. A comparison with the algebraic techniques for Boolean expressions is made to point out some of the key differences between optimizing Boolean expressions and optimizing polynomial expressions. A novel transformation of constant multiplications is then presented, which enables the optimization of expressions consisting of SHIFT operations as well. A method for reducing the number of modular multiplications in integer exponentiation is presented. Finally, a method for finding optimal solutions for these problems using an Integer Linear Programming (ILP) approach is presented.

The optimization techniques presented in this chapter are aimed at reducing the number of operations in arithmetic expressions. This reduction in the number of

operations leads to improvements in performance, area and energy consumption, as shown in our experimental results.

3.1 Optimizing polynomial expressions

Polynomial expressions can be optimized using algebraic techniques. First, a subset of all algebraic divisors is generated (kernelling), which helps to determine all multiple term factorizations and common subexpressions. Factoring and multiple term common subexpression elimination are performed using these divisors. The resultant expressions are further optimized by eliminating single term common subexpressions. The algorithms for optimization are explained in the following paragraphs. The following polynomial, which is the Taylor series expansion for sin(x) (approximated to the fourth term) is used as an aid in the explanations. The subscripts in parenthesis represent the term numbers in the set of expressions.

$$\sin(x) = x_{(1)} - S_3 x^3{}_{(2)} + S_5 x^5{}_{(3)} - S_7 x^7{}_{(4)}$$

$$S_3 = 1/3!\ ,\ S_5 = 1/5!,\ S_7 = 1/7!$$

Figure 3.1 Example polynomial expression

Preliminaries

Polynomial expressions are represented in a matrix form, where there is one row for each distinct product term and one column for each variable/constant in the set of expressions. There is an additional field for each row for the sign of the

product term. For example, the polynomial for shown in Figure 3.1 is represented as shown in Figure 3.2.

+/-	x	S_3	S_5	S_7
+	1	0	0	0
-	3	1	0	0
+	5	0	1	0
-	7	0	0	1

Figure 3.2 Matrix representation of polynomial expressions

The following terminology is used to explain the technique. A **literal** is a variable or a constant (e.g. a, b, 2, 3.14…). A **cube** is a product of the variables each raised to a non-negative integer power. In addition, each cube has a positive or a negative sign associated with it. Examples of cubes are $+3a^2b$, $-2a^3b^2c$. An **SOP** representation of a polynomial is the sum of the cubes ($+3a^2b + (-2a^3b^2c) + \ldots$). An SOP expression is said to be **cube-free** if there is no cube that divides all the cubes of the SOP expression. For a polynomial P and a cube c, the expression P/c is a **kernel** if it is cube-free and has at least two terms (cubes). For example in the expression $P = 3a^2b - 2a^3b^2c$, the expression $P/ (a^2b) = (3 - 2abc)$ is a kernel. The cube that is used to obtain a kernel is called a **co-kernel**. In the above example the cube a^2b is a co-kernel. The literals, cubes, kernels and co-kernels are represented using the matrix representation.

Kernelling in polynomial expressions

The importance of kernels of a set of polynomial expressions is illustrated by the following theorem:

Theorem[1]: Two expressions f and g have a common multiple cube divisor if and only if there are two kernels K_f and K_g belonging to the set of kernels generated for f and g respectively having a multiple cube intersection.

Therefore by finding kernel intersections, all multiple term common subexpressions can be detected. Furthermore, the product of a kernel and its corresponding co-kernel provide a possible factorization opportunity. Since the set of kernels is much smaller than the set of all divisors, finding intersections amongst kernels leads to an efficient algorithm.

The algorithm for extracting all the kernels and co-kernels of a set of polynomial expressions is shown in Figure 3.3. This algorithm is analogous to the kernel generation algorithms in multi-level logic synthesis [20] and has been generalized to handle polynomial expressions of any order. The main difference is the procedure for division, where the values of the elements in the divisor are subtracted from the corresponding elements from each suitable[2] row in the dividend. The other steps have been modified to handle polynomial expressions of any order. The recursive algorithm **Kernels** is called for each expression with the literal index 0 and the cube d which is the co-kernel extracted up to this point, which is initialized to φ. The recursive nature of the algorithm extracts kernels and co-kernels within the kernel extracted and returns when there are no kernels present.

---

[1] This theorem is valid only for algebraic common factors. For example this theorem does not consider the factor (a-b) which can be obtained from $a^2 - b^2 = (a+b)(a-b)$

[2] When performing division of a polynomial by a cube, the suitable set of rows in the dividend are those rows that contain the cube. For example, when dividing $P = \{a^2bc + ac + a^3c^2\}$ by the cube $C = a^2c$, the suitable set of rows correspond to the rows $\{a^2bc, a^3c^2\}$

```
FindKernels({P_i},{L_i})
{
    {P_i} = Set of polynomial expressions;
    {L_i} = Set of Literals;
    {D_i} = Set of Kernels and Co-Kernels = ϕ;
     ∀ Expressions P_i in {P_i}
    {D_i} = {D_i} ∪ {Kernels(0, P_i, ϕ)} ∪ {P_i, 1};
    return {D_i};
}

Kernels(i,P,d)
{
   i = Literal Number ; P =expression in SOP;
   d = Cube;

  D = Set of Divisors = ϕ ;
  for( j = i; j < |L|; j++)
  {
       If( L_j appears in more than 1 row)

      {
          F_i = Divide (P , L_j);
          C = Largest Cube dividing each cube of F_i;

          if( ( L_k ∉ C) ∀ (k < j) )
         {
             F_1 = Divide(F_i, C); // kernel
             D_1 = Merge(d,C, L_j);// co-kernel

             D = D ∪ D_1 ∪ F_1;
             D = D ∪ Kernels(j ,F_1 , D_1);
         }
       }
    }
 return D;

}
```

```
Divide(P,d)
{
   P = Expression; d = cube ;
   Q = set of rows of P that contain cube d;

    ∀ Rows R_i of Q
    {
        ∀ Columns j in the Row R_i
        R_i[j] = R_i[j] – d[j];
    }

   return Q;
}

Merge(C_1, C_2, C_3)
{
  C_1, C_2, C_3 = cubes ;
   Cube M;

  for(i = 0; i < Number of literals; i++)
    M[i] = C_1[i] + C_2[i] + C_3[i];

        return M;

}
```

Figure 3.3 Kernelling for polynomial expressions

The algorithm **Divide** is used to divide an SOP expression by a cube d. It first collects those rows that contain cube d (those rows R such that all elements R[i] $\geq$ d[i]). The cube d is then subtracted from these rows and these rows form the quotient.

The biggest cube dividing all the cubes of an SOP expression is the cube C with the greatest literal count (literal count of C is $\sum_i C[i]$) that is contained in each cube of the expression.

The algorithm **Merge** is used to find the product of the cubes d, C and L_j and is done by adding up the corresponding elements of the cubes. In addition to the kernels generated from the recursive algorithm, the original expression is also added as a kernel with co-kernel '1'. Passing the index i to the Kernels algorithm, checking for $L_k \notin C$ and iterating from literal index i to |L| (total number of literals) in the for loop are used to prevent the same kernels from being generated again.

As an example, consider the expression F = sin(x) in Figure 3.1. The literal set is {L} = {x,S_3,S_5,S_7}. Dividing first by x gives $F_t = (1 - S_3x^2 + S_5x^4 - S_7x^6)$. There is no cube that divides it completely, so the first kernel $F_1 = F_t$ with co-kernel x is recorded. In the next call to Kernels algorithm dividing F by x gives F_t = $-S_3x + S_5x^3 - S_7x^5$. The biggest cube dividing this completely is C = x. Dividing F_t by C gives the next kernel $F_1 = (-S_3 + S_5x^2 - S_7x^4)$ and the co-kernel is x*x*x = x^3. In the next iteration the kernel $S_5 - S_7x^2$ with co-kernel x^5 is obtained. Finally the original expression for sin(x) is also recorded with co-kernel '1'. The set of all co-kernels and kernels generated are

$[x](1 - S_3x^2 + S_5x^4 - S_7x^6)$; $[x^3](-S_3 + S_5x^2 - S_7x^4)$;

$[x^5](S_5 - S_7x^2)$; $[1](x - S_3x^3 + S_5x^5 - S_7x^7)$;

Constructing Kernel Cube Matrix (KCM)

All kernel intersections and multiple term factors can be identified by arranging the kernels and co-kernels in a matrix form. There is a row in the matrix for each kernel generated, and a column for each distinct cube of a kernel generated. The KCM for the sin(x) expression is shown in Figure 3.4. The kernel corresponding to the original expression (with co-kernel '1') is not shown for ease of presentation. An element (i,j) of the KCM is '1', if the product of the co-kernel in row i and the kernel-cube in column j gives a product term in the original set of expressions. The number in parenthesis in each element represents the term number that it represents. A **rectangle** is a set of rows and set of columns of the KCM such that all the elements are '1'. The **value** of a rectangle is the number of operations saved by selecting the common subexpression or factor corresponding to that rectangle.

		1	2	3	4	5	6	7	8	9
		1	$-S_3x^2$	S_5x^4	$-S_7x^6$	$-S_3$	S_5x^2	$-S_7x^4$	S_5	$-S_7x^2$
1	x	$1_{(1)}$	$1_{(2)}$	$1_{(3)}$	$1_{(4)}$					
2	x^3					$1_{(2)}$	$1_{(3)}$	$1_{(4)}$		
3	x^5								$1_{(3)}$	$1_{(4)}$

Figure 3.4 Kernel extraction on example polynomial

Selecting the optimal set of common subexpressions and factors is equivalent to finding a maximum valued covering of the KCM, and is analogous to the minimum weighted rectangular covering problem described in [21], which is NP-hard. A greedy iterative algorithm described in 3.1 is used where the best prime rectangle is

picked in each iteration. A **prime rectangle** is a rectangle that is not covered by any other rectangle, and thus has more value that any rectangle it covers.

Given a rectangle with the following parameters, its value can be calculated as follows:

R = number of rows ; C = number of columns

$M(R_i)$ = number of multiplications in row (co-kernel) i.

$M(C_i)$ = number of multiplications in column (kernel-cube i).

Each element (i,j) in the rectangle represents a product term equal to the product of co-kernel i and kernel-cube j, which has a total number of $M(R_i) + M(C_i) + 1$ multiplications. The total number of multiplications represented by the whole rectangle is equal to $R*\sum_{C} M(C_i) + C*\sum_{R} M(R_i) + R*C$. Each row in the rectangle has C – 1 additions for a total of $R*(C-1)$ additions.

By selecting the rectangle, a common factor with $\sum_{C} M(C_i)$ multiplications and C -1 additions is selected. This common factor is multiplied by each row which leads to a further $(\sum_{R} M(R_i) + R)$ multiplications. Therefore the value of a rectangle, which represents the savings in the number of operators by selecting the rectangle can be written as shown in Equation 3-1.

$$\text{Value}_1 = \{(C-1)*(R + \sum_{R} M(R_i)) + (R-1)*(\sum_{C} M(C_i)\} + (R-1)*(C-1)$$

Equation 3-1

		1	2	3	4
		1	$\mathbf{x^2d_1}$	$\mathbf{S_5}$	$\mathbf{-S_7x^2}$
1	**x**	$1_{(1)}$	$1_{(2)}$		
2	$\mathbf{x^2}$			$1_{(4)}$	$1_{(5)}$

Figure 3.5 Kernel cube matrix (2nd iteration)

$$\sin(x) = (x\,d_3)_{(1)}$$

$$d_3 = (1)_{(2)} + (x^2 d_1)_{(3)}$$

$$d_2 = (S_5)_{(4)} - (x^2 S_7)_{(5)}$$

$$d_1 = (x^2 d_2)_{(6)} - (S_3)_{(7)}$$

Figure 3.6 Expressions after kernel intersection

Constructing Cube Literal Incidence Matrix (CIM)

The KCM enables the extraction of all multiple cube common subexpressions and factors. All possible cube intersections can be identified by arranging the cubes in a matrix where there is one row for each cube and one column for each literal in the set of expressions. This matrix is called the Cube Literal Incidence Matrix (CIM). The CIM is generally formed after finding all the multiple term common subexpressions and factors, though it can also be formed before that. The CIM is the same as the matrix representation of the expressions.

Each cube intersection appears in the CIM as a rectangle. A **rectangle** in the CIM is a set of rows and columns such that all the elements are non-zero. The value

of a rectangle is the number of multiplications saved by selecting the single term common subexpression corresponding to that rectangle. The best set of common cube intersections is obtained by a maximum valued covering of the CIM. A greedy iterative algorithm described in Figure 3.8 is used, where the best prime rectangle is extracted in each iteration. The common cube C corresponding to the prime rectangle is obtained by finding the minimum value in each column of the rectangle. The value of a rectangle with R rows can be calculated as follows:

Let ∑C[i] be the sum of the integer powers in the extracted cube C. This cube saves ∑C[i] – 1 multiplications in each row of the rectangle. The cube itself needs ∑C[i] – 1 multiplications to compute. Therefore the value of the rectangle is given by Equation 3-2.

$$Value_2 = (R-1) \times (\sum C[i] - 1)$$

Equation 3-2

Extracting Kernel Intersections

The algorithm for finding kernel intersections is shown in Figure 3.7 and is analogous to the **Distill** procedure in multi-level logic synthesis [22]. It is a greedy iterative algorithm in which the best prime rectangle is extracted in each iteration. In the outer loop, kernels and co-kernels are extracted for the set of expressions $\{P_i\}$ and the KCM is constructed from them. The outer loop exits if there is no favorable rectangle in the KCM. Each iteration in the inner loop selects the most valuable rectangle if present, based on the value function in Equation 3-2. This rectangle is

added to the set of expressions and a new literal is introduced to represent this rectangle. The kernel cube matrix is then updated by removing those 1's in the matrix that correspond to the terms covered by the selected rectangle.

```
FindKernelIntersections( {Pi}, {Li})
{
    while(1)
    {
      D = FindKernels({Pi}, {Li});
      KCM = Form Kernel Cube Matrix (D);
      {R} =Set of new kernel intersections = φ ;
      {V} =Set of new variables = φ;
      if (no favorable rectangle) return;
      while(favorable rectangles exist)
      {
        {R} = {R} ∪ Best Prime Rectangle;
        {V} = {V} ∪ New Literal;
        Update KCM;
      }
      Rewrite {Pi} using{R};
      {Pi} = {Pi} ∪ {R};
      {Li} = {Li} ∪ {V};
    }
}
```

Figure 3.7 Algorithm for finding kernel intersections

For example, in the KCM for the sin(x) polynomial shown in Figure 3.4, consider the rectangle that is formed by the row {2} and the columns {5,6,7}. This is a prime rectangle having R = 1 row and C = 3 columns. The total number of multiplications in rows $\sum_{R} M(R_i) = 2$ and total number of multiplications in the columns $\sum_{C} M(C_i) = 6$. This is the most valuable rectangle and is selected. Since this rectangle covers the terms 2,3 and 4, the elements in the matrix corresponding to these terms are deleted. No more favorable (valuable) rectangles are extracted in

this KCM. There are two expressions now, after rewriting the original expression for sin(x).

Extracting kernels and co-kernels as before, the KCM shown in Figure 3.5 is obtained. Again the original expressions for sin(x) and d_1 are not included in the matrix to simplify representation. From this matrix two favorable rectangles can be extracted corresponding to $d_2 = (S_5 - S_7x^2)$ and $d_3 = (1 + x^2d_1)$. No more rectangles are extracted in the subsequent iterations and the set of expressions can now be written as shown in Figure 3.6.

Extracting Cube Intersections

The algorithm for extracting cube intersections is analogous to the Condense algorithm [22] in multi-level logic synthesis, and is shown in Figure 3.8 . It is similar to the algorithm to find kernel intersections and is performed on the CIM at the end of the procedure for finding kernel intersections. After each iteration of the inner loop, the CIM is updated by subtracting the extracted cube from all rows in the CIM in which it occurs. As an example, consider the CIM in Figure 3.9 for the set of expressions in Figure 3.6. The prime rectangle consisting of the rows {3,5,6} and the column {1} is extracted, which corresponds to the common subexpression $C = x^2$. Using Equation 3-2, it can be seen that this rectangle saves two multiplications. This is the most favorable rectangle and is chosen. . No more cube intersections are detected in the subsequent iterations. A literal $d_4 = x^2$ is added to the expressions which can now be written as in Figure 3.10.

```
FindCubeIntersections( (Pi), (Li))
{
    while(1)
    {
       M = Cube Literal incidence Matrix
       (V) = Set of new variables = φ;
       (C) = Set of new cube intersections = φ;
       if( no favorable rectangle present) return;

        while(Favorable rectangles exist)
        {
            Find B = Best Prime Rectangle;
            (C) = (C) ∪ Cube corresponding to B
            (V) = (V) ∪ New Literal
            Collapse(M,B);
        }
        M = M ∪ (C);
       (Li) = (Li) ∪ (V);
    }
}

Collapse(M,B)
{
    ∀ rows i of M that contain cube B
       ∀ columns j of B
       M[i,j] = M[i,j] – B[,j];
}
```

Figure 3.8 Algorithm for extracting cube intersections

Term	+/-	1	2	3	4	5	6	7	8
		x	d_1	d_2	d_3	1	S_3	S_5	S_7
1	+	1	0	0	1	0	0	0	0
2	+	0	0	0	0	1	0	0	0
3	+	2	1	0	0	0	0	0	0
4	+	0	0	0	0	0	0	1	0
5	-	2	0	0	0	0	0	0	1
6	+	2	0	1	0	0	0	0	0
7	-	0	0	0	0	0	1	0	0

Figure 3.9 CIM for example polynomial

$$d_4 = x*x$$
$$d_2 = S_5 - S_7 * d_4$$
$$d_1 = d_2 * d_4 - S_3$$
$$d_3 = d_1 * d_4 + 1$$
$$\sin(x) = x * d_3$$

Figure 3.10 Final expressions after optimization

3.1.1 Complexity and quality of presented algorithms

The complexity of the kernel generation algorithm Figure 3.7 can be defined as the maximum number of kernels that can be generated from an expression. The total number of kernels generated from an expression depends on the number of variables, the number of terms and the order of the expressions and also the distribution of the variables in the original terms. Since kernels are generated by division by cubes (co-kernels), the set of co-kernels generated have to be distinct. Given the number of variables m, and the maximum integer exponent of any variable in the expression as n, the worst case number of kernels generated for an expressions can be expressed as the number of different co-kernels possible. The number of different co-kernels possible is $(n + 1)^m$. For example, if we have m = 2 variables, a and b, and the maximum integer power n = 2, then we have $(2 + 1)^2 = 9$ possible co-kernels which are 1, b, b^2, a, ab, ab^2, a^2, a^2b, a^2b^2.

	1	1	1	1
1		1	1	1
1	1		1	1
1	1	1		1
1	1	1	1	

Figure 3.11 Worst case complexity for finding kernel intersections

In such a scenario, the number of rectangles generated is exponential in the number of rows/columns in the matrix. The number of kernels that are generated is equal to the number of rows in the matrix and the number of distinct kernel cubes is equal to the number of columns. The algorithm for finding single cube intersections also has a worst case exponential complexity, when the Cube Intersection Matrix (CIM) is of the form shown in Figure 3.11. The number of rows of the CIM is equal to the number of terms and the number of columns is equal to the number of literals in the set of expressions.

The algorithms that we presented are greedy heuristic algorithms. To the best of our knowledge, there has been no previous work done for finding an optimal solution for the general common subexpression elimination problem, though recently there has been an approach for solving a restricted version of the problem using Integer Linear Programming (ILP) [23]. We discuss the ILP based approaches at the end of this chapter.

3.1.2 Differences with algebraic techniques in Logic Synthesis

The techniques for polynomial expressions that were presented in the previous Section are based on the algebraic methods originally developed for Boolean expressions in multi-level logic synthesis [20-22]. These methods have been generalized to handle polynomial expressions of any order and consisting of any number of variables. The changes that were made were mainly because of the

differences between Boolean and arithmetic operations. These differences are detailed below

1. Restriction on Single Cube Containment (SCC): In [22], the Boolean expressions are restricted to be a set of non-redundant cubes, such that no cube properly contains another. An example of redundant cubes is in the Boolean expression f = ab + b. Here the cube b can be written as b = ab + a'b. Therefore cube 'b' contains the other cube 'ab'. For polynomial expressions, this notion of cube containment does not exist, and there can be a polynomial P = ab + b. Dividing this polynomial by b, P/b = (ab + b)/b = a + 1, which is treated as a valid expression. To handle this, '1' is treated as a distinct literal in our representation for polynomial expressions.

For simplifying our algorithm, a different restriction is placed on the polynomial expressions. Each polynomial expression is required to be a summation of *unequal* terms (cubes). Therefore the expression $P = a^2 + ab + ab + b^2$ is not permitted, since there are two instances of the cube 'ab'. If repeated terms are allowed in the expressions, then the kernels also would have repeated terms, and there is no way to represent them in the Kernel Cube Matrix. For this expression P, there are two co-kernel and kernel pairs (a)(a + b + b) and (b)(a + a + b). Though this can be handled by modifying our kernel generation algorithm to generate all possible cube-free expressions, this complicates the kernel generation procedure. For handling such expressions, the equal terms are added up before executing the algorithm. Therefore P is rewritten as $P = a^2 + 2ab + b^2$. This prevents the factorization of the expression as P = (a+b)*(a+b), but it makes the algorithm much simpler.

2. Algebraic division and the kernel generation algorithm: For Boolean expressions, there is a restriction on the division procedure for obtaining kernels. During division, the quotient and the divisor are required to have orthogonal variable supports. For example, if there is a Boolean expression f, and we need to divide f by another expression g (f/g), then the *quotient* expression h is defined [22] as "*the largest set of cubes in f, such that* $h \perp g$ *(h is orthogonal to g, that is the variable support of h and the variable support of g are disjoint), and that* $hg \in f$ *."* For example, if f = a(b + c) + d, and g = a. Then f/g = h = (b + c). It can be seen that $h \perp g$ and hg = a(b + c) $\in$ f.

For polynomial expressions this restriction on orthogonality can be removed, so that polynomial expressions of any order can be processed. This is based on one key difference between the multiplication and the Boolean AND operation. For example, the multiplication a*a = a^2, but the AND operation a.a = a. Therefore it is possible to have a polynomial F = a(ab + c) + d and a divisor G = a. The division F/G = H = (ab + c). It can be seen that H and G are not orthogonal since they share the variable 'a'. Since the kernel generation algorithm generates kernels by recursive division, our algorithm has been modified to take into account this difference. This algorithm is detailed in Figure 3.7. In [22, 24], a central theorem is presented on which the decomposition and factorization of Boolean expressions is based on. This theorem shows how all multiple-cube intersections and algebraic factorizations can be obtained from the set of kernels. For our modified problem, this theorem still

holds and multiple term common subexpressions and algebraic factorizations can be detected by the set of kernels for the polynomial expressions.

3. Finding single cube intersections: In the decomposition and factorization methodology of Boolean expressions presented in [22], multiple term common factors are extracted first and then single cube common factors are extracted. Our methodology for polynomial expressions follows the same order, though our algorithm has been modified to handle higher order terms. All single cube common subexpressions can be detected by our method, and is described in Figure 3.8.

3.1.3 Experimental results

The goal of the experiments was to investigate the usefulness of our technique in reducing the number of operations in polynomial expressions occurring in some real applications. Polynomial expressions are prevalent in signal processing [25] and in 3-D computer graphics [26]. The polynomials were optimized using three different methods: CSE, Horner form and our algebraic technique. The Jouletrack simulator [27] was used to estimate the reduction in latency and energy consumption for computing the polynomials on the StrongARM™ SA1100 microprocessor. The same set of randomly generated inputs were used for simulating the different programs. Table 3-1 shows the result of our experiments, where a comparison is made between the number of operations produced by the three different methods. The first two examples are obtained from source codes from signal processing applications where the polynomials are obtained by approximating Trigonometric

functions by their Taylor series expansions. The next four examples are multivariate polynomials obtained from 3-D computer graphics [28]. The results show that our optimizations reduce the number of multiplications, on an average by 34% over CSE and by 34.9% over Horner. The number of additions is the same in all examples. This is because all the savings are produced by efficient factorization and elimination of single term common subexpressions, which only reduce the number of multiplications. The average run time for our technique for these examples was only 0.45s.

Table 3-1

	Application	**Function**	**Unoptimized**		**Using CSE**		**Using Horner**		**Using our Algebraic Technique**	
			A	M	A	M	A	M	A	M
1	Fast Convolution	FFT	7	56	7	20	7	30	7	10
2	Gaussian noise filter	FIR	6	34	6	23	6	20	6	13
3	Graphics	quartic-spline	4	23	4	16	4	17	4	13
4	Graphics	quintic-spline	5	34	5	22	5	23	5	16
5	Graphics	chebyshev	8	32	8	18	8	18	8	11
6	Graphics	cosine-wavelet	17	43	17	23	17	20	17	17
Average			**7.8**	**37**	**7.8**	**20.3**	**7.8**	**21.3**	**7.8**	**13.3**

Table 3-2 show the results of simulation on the StrongARM^{TM} SA1100 processor, where an average latency reduction of 26.7% over CSE and 26.1% over Horner scheme is observed. Furthermore an energy reduction of 26.4% over CSE and 26.3% over Horner scheme is obtained. The energy consumption measure by the simulator [27] is only for the processor core.

The improvement in the performance on the simple five stage RISC processor is roughly proportional to the savings in the number of operations in the polynomial functions. On the ARM 7TDMI processor, each addition takes one clock cycle and each multiplication takes five clock cycles, though the multiplier is pipelined. The savings in the energy consumption is directly proportional to the reduction in the number of clock cycles. It should be noted that the energy reduction is only for the CPU core, and the model does not take into account the energy consumed in the cache and physical memories.

Table 3-2

	Application	Function	Reduction in Latency and Energy consumption for execution on StrongARM SA1100			
			CSE		Horner	
			Latency (%)	Energy (%)	Latency (%)	Energy (%)
1	Fast Convolution	FFT	26.8	26.1	44.0	42.3
2	Gaussian noise filter	FIR	30.5	26.4	16.5	16.1
3	Graphics	quartic-spline	7.0	10.7	35.5	35.1
4	Graphics	quintic-spline	23.4	24.8	23.3	26.2
5	Graphics	chebyshev	33.4	31.3	27.5	25.4
6	Graphics	cosine-wavelet	39.1	39.0	9.7	12.8
	Average		**26.7**	**26.4**	**26.1**	**26.3**

The polynomials implemented using the three methods (CSE, Horner and algebraic) were also synthesized in hardware to observe the impact of our methods on performance, area and power consumption. The polynomials were synthesized using Synopsys Behavioral Compiler ^{TM} and Synopsys Design Compiler ^{TM} using the 1.0 μ power2_sample.db technology library, with a clock period of 40 ns, operating at 5V. The Synopsys DesignWare ^{TM} library was used for the functional units. The adder and multiplier in this library took one clock cycle and two clock

cycles respectively at this clock period. There were two different hardware constraints that were used for the hardware synthesis: minimum hardware constraints and medium hardware constraints. In the **minimum hardware** constraints, the maximum number of multipliers was set at one. In the **medium hardware** constraints, the maximum number of multipliers was set at four. The Synopsys Power Compiler ™ was then interfaced with the Verilog RTL simulator VCS ™ to capture the total power consumption including switching, short circuit and leakage power. All the polynomials were simulated with the same set of randomly generated inputs.

Table 3-3 Synthesis results with minimum hardware constraints

	Area (%) (I)		Energy @5V (%) (II)		Energy Delay @5V (%) (III)		Energy (scaled voltage) (%) (IV)	
	C	**H**	**C**	**H**	**C**	**H**	**C**	**H**
1	18.6	6.5	33.5	69.9	58.4	90.8	88.9	99.0
2	7.5	0.1	13.6	25.6	20.4	39.4	24.6	49.5
3	0.3	-4.2	21.6	29.3	39.0	48.8	52.2	64.6
4	-7.5	-24.2	29.4	10.4	47.6	25.9	62.2	36.9
5	5.6	2.5	37.0	28.7	57.1	46.1	74.3	59.8
6	3.7	2.0	44.8	36.8	62.8	54.8	78.3	69.7
Average	**4.7**	**-2.8**	**30.0**	**33.4**	**47.5**	**50.9**	**63.4**	**63.2**

Table 3-3 shows the synthesis results when the polynomials were scheduled with minimum hardware constraints. Typically the hardware allocated consisted of a single adder, a single multiplier, registers and control units. The reduction in the area, energy and the energy-delay product achieved by our method over CSE (C) and Horner (H) were measured. The results show that there is not much difference in

area for the different implementations, but there is a significant reduction in total energy consumption (average 30% over CSE and 33.4% over Horner), and energy delay product (average 47.5% over CSE and 50.9% over Horner). The similarity in the area numbers is because of the minimal hardware constraint. Since our method produced the least number of operations, it led to fewer number of clock cycles and thereby lesser energy consumption.

The voltage scaling was done by comparing the latencies of the polynomials using our technique with that produced from CSE and Horner separately. The new voltage was then obtained by using the library parameters. The energy consumption was recomputed using the quadratic relationship between the energy consumption and the supply voltage. The results (Column IV) show an energy reduction of 63.4% over CSE(C) and 63.2% reduction over Horner (H) with voltage scaling.

Table 3-4 shows the synthesis results for medium hardware constraints. Medium hardware implementations trade off area for energy efficiency. Though they have larger area, they have a shorter schedule and lesser energy consumption compared to those produced from minimum hardware constraints. The Synopsys Behavioral Compiler ^{TM} was constrained to allocate a maximum of four multipliers for each example. Table 3-5 shows the number of adders (A) and multipliers (M) allocated for the different implementations. The scheduler will allocate fewer multipliers than four, if the same latency can be achieved by the fewer number of resources.

Table 3-4 Synthesis results with medium hardware constraints

	Area (%)		Energy @5V (%)		Energy Delay @5V (%)	
	C	H	C	H	C	H
1	44.0	48.0	9.8	63.9	-12.7	81.0
2	30.5	3.9	16.1	39.2	9.7	44.1
3	14.8	1.0	9.7	29.6	20.3	58.7
4	8.3	3.7	42.5	29.1	44.9	37.0
5	8.9	9.0	28.2	29.5	39.5	40.6
6	8.0	6.6	41.4	40.8	58.4	59.7
Avg	**19.0**	**12.0**	**24.6**	**38.7**	**26.7**	**53.5**

Table 3-5 Multipliers (M) and Adders(A) allotted with a maximum hardware constraints of 4 multipliers

	CSE		Horner		Our Techniuqe	
	M	A	M	A	M	A
1	4	2	4	1	2	1
2	3	1	2	1	2	1
3	4	1	4	1	4	1
4	3	1	3	2	3	2
5	4	1	4	2	4	2
6	3	1	3	1	3	2

The results show a significant reduction in total energy consumption (average 24.6% over CSE(C) and 38.7% over Horner (H) scheme), as well as energy delay product (average 26.7% over CSE and 53.5 % over Horner). Reduction in energy delay product is obtained for every example except for the first one, where the energy delay product for the expression synthesized using CSE is 12.7% lesser than that produced by our method. This is because, the latency of the CSE optimized expression, when four multipliers are used is much lesser compared to the one optimized using our algebraic method (which uses two multipliers). The delay for Horner scheme is much worse than both CSE and our technique, because the nested additions and multiplications of Horner scheme typically results in a longer critical path.

3.2 Optimizing linear computations with constant multiplications

Linear arithmetic expressions have a number of constant multiplications which are often decomposed into shifts and additions when implemented in hardware. The number of shifts and additions can be significantly reduced by finding common subexpressions among these shifts and additions. One of the popular methods to find common subexpressions is to find common digit patterns in the set of constant multiplying a single variable. Common subexpressions involving any number of variables can be extracted by means of a polynomial transformation of these linear arithmetic expressions. This section presents this polynomial transformation. Two methods are presented for extracting common subexpressions. The first one is similar to the rectangular covering algorithm that was explained for polynomial expressions in Section 3.1. The second algorithm iteratively extracts and eliminates two-term common subexpressions.

Linear arithmetic expressions are often found in signal processing applications. Many of these computations can be expressed as a multiplication of a constant matrix with a vector of input samples X as shown in Equation 3-3, where $c_{i,j}$ represents the $(i,j)^{th}$ element in an NxN constant matrix.

$$Y[i] = \sum_{j=0}^{N-1} c_{i,j} * X[j]$$

Equation 3-3

Transforms of this type include Discrete Cosine Transform (DCT), Discrete Fourier Transform (DFT) etc, which are widely used in image, video and audio processing.

Polynomial transformation of constant multiplications

Using a given representation of the integer constant C, the multiplication with the variable X can be represented as shown in Equation 3-4

$$C \times X = \sum_i \pm (XL^i)$$

Equation 3-4

Here L represents the left shift from the least significant digit and the i's represent the digit positions of the non-zero digits of the constant, 0 being the digit position of the least significant digit. Each term in the polynomial can be positive or negative depending on the sign of the non-zero digit. Thus this representation can handle signed digit representations such as the Canonical Signed Digit (CSD).

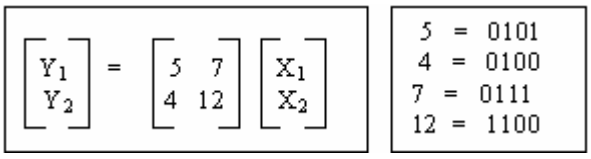


Figure 3.12 Example Linear system

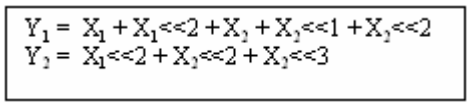


Figure 3.13 Decomposing linear system into shifts and additions

For example the constant multiplication $(12)_{decimal}$ *X = $(1100)_{binary}$*X = $XL^2 + XL^3$ in our polynomial representation. In case of systems where the constants are real numbers represented in fixed point, the constant can be converted into an integer,

and the final result can be corrected by shifting right. For example in the constant multiplication $(0.101)_{binary} * X = (101)_{binary} * X * 2^{-3} = (X + XL^2) * 2^{-3}$ in our polynomial representation. Consider the linear system shown in Figure 3.12. By decomposing the constant multiplications into additions and shifts, the linear system can be rewritten as the expressions Y_1 and Y_2 shown in Figure 3.13. For convenience the constant multiplications can be assumed to be in binary form. Using the polynomial transformation described in Equation 3-4 the expressions can be rewritten as shown in Figure 3.14. The subscripts in parenthesis represent the term numbers which will be used in our optimization algorithm.

$$Y_1 = {}_{(1)}X_1 + {}_{(2)}X_1L^2 + {}_{(3)}X_2 + {}_{(4)}X_2L + {}_{(5)}X_2L^2$$

$$Y_2 = {}_{(6)}X_1L^2 + {}_{(7)}X_2L^2 + {}_{(8)}X_2L^3$$

Figure 3.14 Polynomial transformation of example linear system

3.2.1 Optimizations using the Rectangle Covering techniques

This section presents optimization techniques for linear systems based on the rectangle covering methods. The goal of the optimization is to reduce the number of additions as much as possible by eliminating redundant additions. Similar to the technique presented for polynomial expressions, the reduction is achieved by performing kernel intersections. The kernels for linear systems have slight differences to the kernels in polynomial expressions. Furthermore, the kernel intersection algorithm for linear systems is different from the corresponding algorithm in polynomials. Therefore, the kernelling and rectangle covering

algorithms for linear systems are presented here in detail. The linear system is first transformed using the polynomial transformation as shown in Figure 3.14. Kernelling and kernel intersections are performed on these expressions.

Generating kernels of the polynomial transformation

A **kernel** of an expression is a subexpression that is derived from the original expression by dividing by an exponent of L, and contains at least one term with a zero exponent of L, and all the terms of the original expression that had higher exponents of L. A **divisor** of an expression is a subexpression of the expression having at least one term with a zero exponent of L. The set of kernels of an expression is a subset of the set of divisors of the expression. The algorithm for generating kernels of a set of polynomial expressions is shown in Figure 3.15. The algorithm recursively finds kernels within kernels generated by dividing by the smallest exponent of L. The exponent of L that is used to obtain the kernel is the corresponding **co-kernel** of the kernel expression.

```
Kernels(Pi)
{
   {D} = set of kernels and co-kernels = φ;
    L = 1;

   if(Pi is a kernel)
     {D} = {D} ∪ (Pi, 1);

    while (at least 2 terms with non-zero exponents of L exist in Pi)
   {
      MinL = Minimum (non-zero) exponent of L;
       Pi    = Divide(Pi , MinL);
       L     = L * MinL;
      {D}   = {D} ∪ (Pi, L);
    }
}

Divide(Pi, MinL)
{
  {Ti} = Terms of Pi having non-zero L;
  Divide the exponent of L in each term
   in {Ti} by MinL;
   return {Ti};
 }
```

Figure 3.15 Algorithm for generating kernels for linear systems

For example consider the expression Y_1 in Figure 3.14. Since Y_1 satisfies the definition of a kernel, we record Y_1 as a kernel with co-kernel '1'. The minimum non-zero exponent of L in Y_1 is L. Dividing by L obtains the kernel expression $X_1L + X_2 + X_2L$, which is recorded as a kernel with co-kernel L. This expression has L as the minimum non-zero exponent of L. Dividing by L obtains the kernel $X_1 + X_2$ with co-kernel L^2. No more kernels are generated and the algorithm terminates. For expression Y_2 the kernel $X_1 + X_2 + X_2L$ is obtained with co-kernel L^2. The set of kernels and co-kernels for the expressions in Figure 3.14 are shown in Figure 3.16.

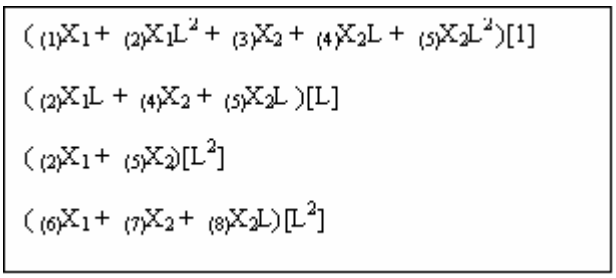


Figure 3.16 Kernels and co-kernels generated for example linear system

The importance of kernels is illustrated by the following theorem.

Theorem: There exists a k-term common subexpression if and only if there is a k-term non-overlapping intersection between at least 2 kernels.

This theorem basically states that each common subexpression in the set of expressions appears as a non-overlapping intersection among the set of kernels of the polynomial expressions. Here a non-overlapping intersection implies that the set of terms involved in the intersection are all distinct. As an example of overlapping terms, consider the binary constant "1001001" represented in Figure 3.17. Converting the multiplication of this constant with the variable X into the polynomial representation, the expression $_{(1)}X + _{(2)}XL^3 + _{(3)}XL^6$ is obtained. There are two kernels generated: $X + XL^3 + XL^6$ covering the terms 1,2 and 3 and $X + XL^3$, covering the terms 2 and 3. An intersection between these two kernels will detect two instances of the subexpression $X + XL^3$, but they overlap as they both cover the term 2. This overlap can be seen in the overlap between the two instances of the bit-pattern "1001" as seen in Figure 3.17.

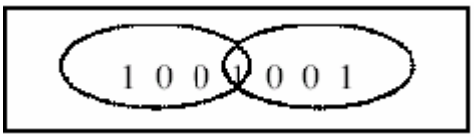


Figure 3.17 Overlap between kernel expressions

Implementing the polynomial $X + XL^3 + XL^6$ with two instances of $X + XL^3$ will result in the second term XL^3 being covered twice. Though it can still be implemented this way, and subtracting XL^3 at the end, it will not be beneficial as it would result in more additions than the original polynomial representation.

Proof:

If: If there is a k-term non-overlapping intersection among the set of kernels, then it means there is a multiple occurrence of the k-term subexpression in the set of polynomial expressions, and the terms involved in the intersection are all distinct. Therefore a k-term non-overlapping intersection in the set of kernels implies a k-term common subexpression.

Only If: Assume there are two instances of the k-term common subexpression. First assume the common subexpression satisfies the definition of a divisor, which means there is at least one term with a non-zero exponent of L. Now each instance of the subexpression will be a part of some kernel expression. This is because from the kernel generation algorithm (Figure 3.15), each kernel is generated recursively dividing through the lowest (non-zero) exponent of L. Each time a division by L is performed all terms that already had a zero exponent of L will be discarded and other terms will be retained. The exponents of L in all the remaining terms will then be

reduced by the lowest exponent of L. Each kernel expression contains all possible terms with zero exponent of L at that step, as well as all terms with higher exponents of L. Therefore if an instance of the common subexpression belongs to a polynomial, it will be a part of one of the kernel expressions of the polynomial. Both instances of the common subexpression will be a part of some kernel expressions and an intersection in the set of kernels will detect the common subexpression.

If the common subexpression does not satisfy the definition of a divisor, which means it does not have any term having a zero exponent of L, the subexpression obtained through division by the smallest exponent of L will also be a common subexpression, and will satisfy the definition of a divisor. Reasoning as above, this common subexpression will be detected by an intersection among the set of kernels.

Matrix transformation to find kernel intersections

The set of kernels generated is transformed into a matrix form called the Kernel Intersection Matrix (KIM), to find kernel intersections. There is one row for each kernel generated and one column for each distinct term in the set of kernel expressions. Each term is distinguished by its sign (+/-), variable and the exponent of L.

Figure 3.18 shows the KIM for our example linear system from its set of kernels and co-kernels. The rows are marked with the co-kernels of the kernels which they represent. Each ‘1’ element (i,j) in the matrix represents a term in the original set of expressions which can be obtained by multiplying the co-kernel in

row i with the kernel term in column j. The number in parenthesis represents the term number that the element represents.

		1	2	3	4	5	6
		$+X_1$	$+X_1L^2$	$+X_2$	$+X_2L$	$+X_2L^2$	$+X_1L$
1	1	$1_{(1)}$	$1_{(2)}$	$1_{(3)}$	$1_{(4)}$	$1_{(5)}$	
2	L			$1_{(4)}$	$1_{(5)}$		$1_{(2)}$
3	L^2	$1_{(2)}$		$1_{(5)}$			
4	L^2	$1_{(6)}$		$1_{(7)}$	$1_{(8)}$		

Figure 3.18 Kernel Intersection Matrix (KIM) for example Linear System

Each kernel intersection appears in the matrix as a rectangle. A **rectangle** is defined as a set of rows and columns such that all the elements are '1'. For example in the matrix in Figure 3.18, the row set {1,4} and the column set {1,3,4} together make a rectangle. A **prime rectangle** is defined as a rectangle that is not contained in any other rectangle.

The **value** of a rectangle is defined as the number of additions saved by selecting that rectangle (kernel intersection) as a common subexpression and is given by

$$Value(R,C) = (R-1)\times(C-1)$$

Equation 3-5

where R is the number of rows and C is the number of columns of the rectangle. The value of the rectangle is calculated after removing the appropriate rows and columns to remove overlapping terms. The algorithm for finding a Maximal Irredundant Rectangle (MIR) is explained in the next section.

The goal of the algorithm is to find the best set of common subexpressions (rectangles) such that the number of additions/subtractions is a minimum. A similar problem is encountered in multi-level logic synthesis [20]where the least number of literals in a set of Boolean expressions is obtained by a minimum weighted rectangular covering of the matrix, which is NP hard [24] . This problem is different since it has to deal with overlapping rectangles in the matrix. A greedy algorithm is used, where in each iteration, the best non-overlapping prime rectangle is picked. Picking only non-overlapping prime rectangles need not always give the best results, since sometimes it may be beneficial to select a rectangle with overlapping terms and then subtract the terms that are covered more than once.

Worst case analysis of KIM: In the worst case, the columns of the KIM are formed from every possible term. The rows of the KIM are formed from every possible exponent of L. Consider an m x m constant matrix with N bits of precision. Since there are m variables and N possible exponents of L, and 2 possible signs (+/-), the worst case number of columns is 2*m*N. For the number of rows, consider the co-kernels generated for each expression. The exponents of L can range from 0 to N-1 and hence there are N possible kernels for each expression. Since there are m expressions, the worst case number of rows is m*N. The worst case number of prime rectangles happens in a square matrix where all the elements in the matrix are 1 except the ones on one diagonal [24], and the number of prime rectangles is exponential in the number of rows/columns of the square matrix. Therefore in our

KIM transformation, the worst case number of prime rectangle and the subsequent time complexity to find the best prime rectangle is of $O(2^{mN})$.

Iterative algorithm to find kernel intersections

The algorithm for extracting kernel intersections is shown in Figure 3.19. It is a greedy iterative algorithm where the best prime rectangle is extracted in each iteration using a ping pong algorithm. In the outer loop, kernels and co-kernels are extracted from the set of expressions $\{P_i\}$ and the KIM is formed from that. The outer loop exits if there is no favorable kernel intersection in the KIM. Each iteration in the inner loop selects the most valuable rectangle if present based on the Value function. The KIM is then updated by removing those 1's in the matrix that correspond to the terms covered by the selected rectangle.

Since the extracted prime rectangle (kernel intersection) may contain overlapping terms, certain rows and/or columns from this rectangle may need to be removed to remove the overlapping terms. The algorithm for the extraction of MIR (ExtractMIR) is also shown in Figure 3.19. The algorithm iteratively removes a row or a column from the rectangle that produces the maximum reduction in the number of overlapping terms. As an illustration consider the rectangle in the KIM in Figure 3.18 comprising of the rows {1,2,4} and the columns {3,4}. This rectangle has overlapping terms since the term 4 is repeated in the rectangle.

It can be observed that removal of either row 1 or row 2 will remove the overlap in the rectangle. Consider the KIM for the example linear system shown in

Figure 3.18. The procedure for ping_pong_row returns the prime rectangle R_1 consisting of the rows {1,4} and the columns {1,3,4}, which has a value 2 (Equation 3-5) . The procedure for ping_pong_col produces the rectangle comprising the rows {1,3,4} and the columns {1,3}, which also has a value 2. Rectangle R_1 which corresponds to the subexpression $D_1 = X_1 + X_2 + X_2L$ is chosen arbitrarily. This rectangle covers the terms {1,3,4,6,7,8}.

```
Find_Kernel_Intersections( {Pi}, {Xi})
{
  {Pi} = Set of polynomial expressions
  {Xi} = Original Variables

  while(1)
  {
      For (all expressions in {Pi} )
        {D} = {D} ∪ Kernels(Pi);
      KIM = Form Kernel Intersection Matrix({D});
      if( no favorable rectangle exists) return;

      {R} = Set of new kernel intersections = φ;
      {V} = Set of new variables = φ;

    while(favorable rectangle exists)
     {
      R1 = Best_Rectangle_Ping_Pong(KIM);
      MIR_R1 = ExtractMIR(R1);
      {R} = {R} ∪ MIR_R1;
      {V} = {V} ∪ New Variable;
       Update KIM;
     }

      Rewrite {Pi} using {R};
      {Pi} = {Pi} ∪ {R};
      {Xi} = {Xi} ∪ {V};
   }
}
```

```
ExtractMIR(R)
{
   R = Rectangle;
  while(R has at least 2 rows and 2 columns)
  {
    if(R does not have overlapping terms)
     return R;

    Remove row or column that reduces the most
     number of overlapping terms from R;
  }

  return φ; // could not find a useful rectangle
}



 Best_Rectangle_Ping_Pong(KIM)
{
    seed_row = seed row in KIM;
    Rectangle R1 = ping_pong_row(seed_row);

  seed_col = seed column in KIM;
  Rectangle R2 = ping_pong_col(seed_col);

   if( value(R2) > value(R1) ) return R2
   else return R1
}
```

Figure 3.19 Algorithm to find kernel intersections for linear systems

The KIM is updated by removing these terms covered by the selected rectangle. No more kernel intersections can be found in the KIM and the inner loop exits.

$$Y_1 = {}_{(1)}D_1 + {}_{(2)}X_1L^2 + {}_{(3)}X_2L^2$$
$$Y_2 = {}_{(4)}D_1L^2$$
$$D_1 = {}_{(5)}X_1 + {}_{(6)}X_2 + {}_{(7)}X_2L$$

Figure 3.20 Expressions after 1st iteration

		1	2	3	4	5	6
		$+D_1$	$+X_1L^2$	$+X_2L^2$	$+X_1$	$+X_2$	$+X_2L$
1	1	$1_{(1)}$	$1_{(2)}$	$1_{(3)}$			
2	L^2				$1_{(2)}$	$1_{(3)}$	
3	1				$1_{(5)}$	$1_{(6)}$	$1_{(7)}$

Figure 3.21 Extracting kernel intersections (2nd iteration)

$$D_2 = X_1 + X_2$$
$$D_1 = D_2 + X_2<<1$$
$$Y_1 = D_1 + D_2<<2$$
$$Y_2 = D_1<<2$$

Figure 3.22 Set of expressions after the end of kernel intersection

The expression for D_1 is added to the set of expressions and a new variable D_1 is added to the set of variables. The expressions are rewritten as shown in Figure 3.20. The KIM is constructed for these expressions, and is shown in Figure 3.21. There is only one valuable rectangle in this matrix that corresponds to the rows {2,3} and the columns {4,5}, and has a value 1. This rectangle is selected. No more rectangles are selected in the further iterations and the algorithm terminates. The final set of expressions is shown in Figure 3.22, which has only three additions and three shifts.

Finding the best prime rectangle using a ping pong algorithm

The procedure for finding the best prime rectangle is similar to the ping pong algorithm used in multi-level logic synthesis [24]. The algorithm extracts two rectangles from the procedures ping_pong_row and ping_pong_col and selects the best one between them. The procedure ping_pong_row tries to build a rectangle, starting from a seed row, each time adding a row by intersecting the column set, till the number of columns becomes less than 2. The criterion for selecting the best row in each step is based on the value of the maximum irredundant rectangle (MIR) that can be extracted from the rectangle created by intersection with that row. The rectangle with the best value (the one that saves the most number of additions/subtractions) is recorded during this construction and is returned at the end of the procedure. This algorithm is quadratic in the number of rows in the matrix. The algorithm for ping_pong_col follows the same steps as ping_pong_row, except that the rectangle is built by intersecting with a column in each step, and the algorithm is quadratic in the number of columns in the KIM.

3.2.2 Common Subexpression Elimination by iteratively eliminating two term common subexpressions

The previous method (Section 3.2.1) eliminated common subexpressions using rectangular covering. This section presents a much simpler and faster method

for doing the same. The method is based on iteratively finding and eliminating two term common subexpressions. This method overcomes a major limitation of the rectangular covering method, that is in detecting common subexpressions that have their signs reversed. The technique is explained using an example of the integer transform used in the latest H.264 video coding standard [1, 29], shown in Figure 2.5.

Generating two-term divisors

Figure 3.24 shows the algorithm ***Divisors*** that is used to generate two-term divisors. A **two-term divisor** of a polynomial expression is a set of two terms obtained after dividing any two terms of the expression by their least exponent of L. This is equivalent to factoring by the common shift between the two terms. Therefore, the divisor is guaranteed to have at least one term with a zero exponent of L. A **co-divisor** of a divisor is the exponent of L that is used to divide the terms to obtain the divisor. A co-divisor is useful in dividing the original expression if the divisor corresponding to it is selected as a common subexpression.

$$
\begin{aligned}
Y_0 &= X_0 + X_1 + X_2 + X_3 \\
Y_1 &= X_0L + X_1 - X_2 - X_3L \\
Y_2 &= X_0 - X_1 - X_2 + X_3 \\
Y_3 &= X_0 - X_1L + X_2L - X_3
\end{aligned}
$$

Figure 3.23 Polynomial transform of the H.264 Integer transform

As an illustration of the divisor generating procedure consider the expression Y_1 in Figure 3.23. Consider the terms X_0L and $-X_3L$. The minimum exponent of L in

both these terms is L. Therefore, after dividing by L, we obtain the divisor $(X_0 – X_3)$ with co-divisor L. The other divisors generated for Y_1 are $(X_0L + X_1)$, $(X_0L – X_2)$, $(X_1 – X_2)$, $(X_1 – X_3L)$ and $(-X_2 – X_3L)$. All these divisors have co-divisors 1.

```
Divisors({Pi})
{
   {Pi} = Set of expressions in polynomial form;
   {D} = Set of divisors and co-divisors = {Φ};

   for (every expression Pi in {Pi})
    {
      for (every pair of terms (ti, tj) in Pi)
      {
        MinL = Minimum power of L in (ti, tj); // co-divisor
        ti1 =  ti/MinL;
        tj1 =  tj/MinL;
        d  = (ti1 + tj1); // divisor;
       {D} = {D} ∪ (d, MinL);
      }
    }

  return {D};

}
```

Figure 3.24 Algorithm to generate two-term divisors

The importance of these two-term divisors is illustrated by the following theorem.

Theorem: There exists a multiple term common subexpression in a set of expressions *if and only if* there exists a non-overlapping intersection among the set of divisors of the expressions.

This theorem basically states that there is a common subexpression in the set of polynomial expressions representing the linear system, if and only if there are at least two non-overlapping divisors that intersect. Two divisors are said to be **intersecting** if their absolute values are equal. For example, $(X_1 - X_2L)$ intersects both $(-X_2L + X_1)$ and $(X_2L - X_1)$. Two divisors are considered to be **overlapping** if one of the terms from which they are obtained is common. For example consider the following constant multiplication $(10101)_{binary}$*X, which is transformed to ${}_{(1)}X + {}_{(2)}XL^2 + {}_{(3)}XL^4$ in the polynomial representation. The numbers in parenthesis represent the term numbers in this expression. Now according to the divisor generating algorithm, there are two instances of the divisor $(X + XL^2)$ involving the terms (1, 2) and (2, 3) respectively. Now these divisors are said to overlap since they contain the term 2 in common.

Proof:

(If) If there is an M-way non-overlapping intersection among the set of divisors of the expressions, by definition it implies that there are M non-overlapping instances of a two-term subexpression corresponding to the intersection.

(Only if) Suppose there is a multiple term common subexpression C, appearing N times in the set of expressions, where C has the terms $\{t_1, t_2, \ldots t_m\}$. Take any $e = \{t_i, t_j\} \in$ C. Consider two cases. In the first case, if *e* satisfies the definition of a divisor, then there will be at least N instances of *e* in the set of divisors, since there are N instances of C and our divisor extraction procedure extracts all 2-term divisors. In the second case where *e* does not satisfy the definition of a divisor (there are no terms in *e* with zero exponent of L), there exists $e^1 = \{t_i^1, t_j^1\}$ obtained (by dividing

by the minimum exponent of L) which satisfies the definition of a divisor, for each instance of *e*. Since there are N instances of C, there are N instances of *e*, and hence there will be N instances of e^1 in the set of divisors. Therefore in both cases, an intersection among the set of divisors will detect the common subexpression

Algorithm to eliminate common subexpressions

Figure 3.25 shows our iterative algorithm for detecting and eliminating two-term common subexpressions. In the first step, frequency statistics of all distinct divisors are computed and stored. This is done by generating divisors $\{D_{new}\}$ for each expression and looking for intersections with the existing set {D}. For every intersection, the frequency statistic of the matching divisor d_1 in {D} is updated and the matching divisor d_2 in $\{D_{new}\}$ is added to the list of intersecting instances of d_1. The unmatched divisors in $\{D_{new}\}$ are then added to {D} as distinct divisors.

In the second step of the algorithm, the best two-term divisor is selected and eliminated in each iteration. The best divisor is the one that has the most number of non-overlapping divisor intersections. The set of non-overlapping intersections is obtained from the set of all intersections by using an iterative algorithm in which the divisor instance that has the most number of overlaps with other instances in the set is removed in each iteration till there are no more overlaps. After finding the best divisor in {D}, the set of terms in all instances of the divisor intersections is obtained. From this set of terms, the set all divisors that are formed using these terms is obtained. These divisors are then deleted from {D}. As a result the frequency statistics of some divisors in {D} will be affected, and the new statistics for these

divisors is computed and recorded. New divisors are formed using the new terms formed during division of the expressions. The frequency statistics of the new divisors are computed separately and added to the dynamic set of divisors {D}.

```
Optimize ({Pi})
{
    {Pi} = Set of expressions in polynomial form;
   {D} = Set of divisors = φ ;
  // Step 1. Creating divisors and their frequency statistics
    for each expression Pi in {Pi}
    {
        {Dnew} = Divisors(Pi);
        Update frequency statistics of divisors in {D};
        {D} = {D} ∪ { Dnew};
   }

  //Step 2. Iterative selection and elimination of best divisor
  while (1)
  {
        Find d = divisor in {D} with most number
                 of non-overlapping intersections;
        if (d == NULL) break;
        Divide affected expressions in {Pi} by d;
       {d^j} = set of intersecting instances of d;

      for each instance d^j in {d^j}
          Remove from {D} all instances of divisors formed using the terms in d^j;

       Update frequency statistics of affected divisors;
      {Dnew} = Set of new divisors from new terms added by division;
      {D} = {D} ∪ {Dnew};
   }

}
```

Figure 3.25 Algorithm for eliminating two-term common subexpressions

Applying this technique to the set of expressions in Figure 3.23 results in four common subexpressions (D_0-D_3) being detected. The final set of expressions is shown in Figure 3.26a, which can be implemented as shown in Figure 2.6. This is

the same implementation that is reported in [1] where the result is obtained manually. From Figure 3.26a, it can be seen that the common subexpressions $D_1 = (X_1 + X_2)$ and $D_2 = (X_1 - X_2)$ have instances that have their signs reversed (from Figure 3.26a, D_1 is positive in Y_0 and negative in Y_2 and D_2 is positive in Y_1 and negative and shifted in Y_3). Since the rectangle covering method cannot detect common subexpressions with signs reversed, it cannot detect D_1 and D_2. Hence, it results in an implementation with 10 additions/subtractions (two more than this method). The result of the rectangle covering algorithm is shown in Figure 3.26b.

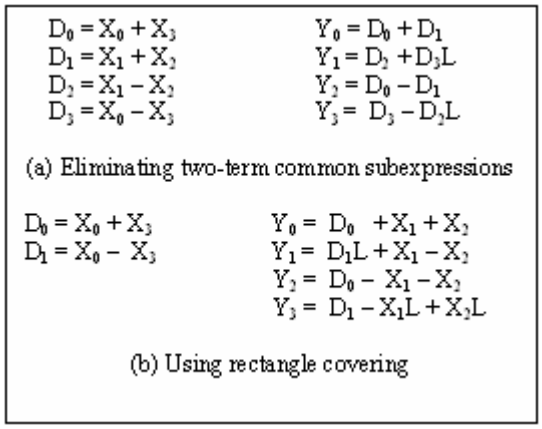


Figure 3.26 Optimization of the H.264 Integer transform

Differences with the Fast-Extract (FX) method in multi-level logic synthesis

The two-term extraction method discussed in this section has been derived from the Fast-Extract (FX) method in logic synthesis [30]. The differences mainly lie in the way the divisors are generated. For Boolean expressions, both single term and double term divisors are generated. For example “ab” is a single term divisor. In [30]

the single cube divisors can only have two literals. Therefore “abc” cannot be a divisor, but instead “ab”, “ac” and “bc” will be the divisors extracted from “abc”. Since Boolean expressions can have complements, the divisors generation procedure also generates the complements of the divisors. Compared to the logic synthesis techniques, the two-term extraction algorithm for arithmetic expressions only generates two-term divisors. This is because these expressions have a single type of operator (addition/subtraction) compared to Boolean expressions which have both AND and OR gates. The number of divisors that have to be generated for arithmetic expressions is significantly lower because of this. For arithmetic expressions, there can be two divisors that are the same but have their signs reversed. For example (a – b) and (b – a) are two divisors having the same absolute value. These two divisors are considered the same and the signs are adjusted during the expression rewriting.

The greedy algorithm for the extraction is very similar for Boolean and Arithmetic expressions. For Boolean expressions, the divisor with the greatest savings in literals in selected, and for arithmetic expressions, the divisor that obtains the greatest savings in the number of additions is selected.

3.2.3 Experimental results

This section compares the reduction in the number of additions/subtractions obtained by different optimization techniques and observes its effect on the area, latency and power consumption of the synthesized hardware. The following transforms were considered: Discrete Cosine Transform (DCT), Inverse Discrete

Cosine Transform (IDCT), Discrete Sine Transform (DST) and Discrete Hartley Transform (DHT). For DFT, the matrix was split into a real part (RealDFT) and an imaginary part (ImagDFT) and the optimizations were carried out separately for the two matrices. The experimental results were performed for the 8x8 constant matrices (8 point), where the constants are represented in fixed point using 16 digits of precision.

Four main optimization techniques are compared in the results. They are the two methods discussed in this thesis (Section 3.2.1 and Section 3.2.2) besides the methods described in [9, 14]. The comparison with [14] was done because it is a recent work on the matrix form of linear systems. The with [9] is done since it is the most widely referenced work on the multiple constant multiplication problem. In [9] a post processing step is suggested for matrix forms of linear systems where each expression after the first stage of optimization is viewed as a bit pattern and the number of additions are further reduced by finding common bit patterns. The number of additions/subtractions obtained by various methods is compared in Table 3-6.

From Table 3-6, it can be seen that the iterative two-term CSE method gives the least number of additions/subtractions compared to all known techniques. This method produces a reduction of 10% over the rectangular covering method. The average CPU time for this method is only 0.24s, which is more than three times faster than the rectangular covering technique which takes 0.84 s.

Table 3-6 Comparing number of additions produced by different methods

Example	Number of Additions/Subtractions				
	Original (I)	Potkonjak (II)	RESANDS (III)	Rectangular Covering	Two-term

				(IV)	CSE (V)
DCT	274	227	202	174	153
IDCT	242	222	183	162	143
RealDFT	253	208	193	165	144
ImagDFT	207	198	178	134	124
DST	320	252	238	200	187
DHT	284	211	209	175	158
Average	**263.3**	**219.7**	**200.5**	**168.3**	**151.5**

The designs were synthesized using Synopsys Design Compiler™ and Synopsys Behavioral Compiler™ using the 1.0 μm CMOS power_sample.db technology library (since it was the only available library characterized for RTL power estimation) using a clock period of 50 ns. The Synopsys DesignWare™ library was used for the functional units. The designs were scheduled with minimum latency constraints. With these constraints, the tool schedules the design with the minimum possible latency, and then minimizes the area with the achieved latency. The Synopsys Power Compiler™ was then interfaced with the Verilog RTL simulator VCS™ to capture the total power consumption including switching, short circuit and leakage power. All the designs were simulated with the same set of randomly generated inputs.

The synthesis results (Area and Latency) and RTL power estimation of the designs were compared for the three methods RESANDS, Rectangular covering and two-term CSE. These methods were chosen because they gave the least number of additions/subtractions (Table 3-6). The results are shown in Table 3-7 and Table 3-8. In these tables the columns marked (III), (IV) and (V) represent the results for the RESANDS method, the rectangle covering method and the two-term extraction method respectively.

Table 3-7 Synthesis results (Area and Latency)

Example	Area (Library Units)			Latency (Clock Cycles)		
	(III)	(IV)	(V)	(III)	(IV)	(V)
DCT	90667	73311	66759	10	11	10
IDCT	81868	66864	62883	10	11	10
RealDFT	90946	69827	64026	10	11	10
ImagDFT	75140	55940	54606	10	10	10
DST	108101	84715	81214	11	11	11
DHT	93939	71272	67775	11	11	10
Average	**90110**	**70322**	**66211**	**10.3**	**10.8**	**10.2**

Table 3-8 Power consumption results

Example	Power Consumption (µWatts)		
	(III)	(IV)	(V)
DCT	729	504	531
IDCT	662	547	569
RealDFT	707	544	554
ImagDFT	644	575	490
DST	607	718	595
DHT	598	545	527
Average	**657.8**	**572.2**	**544.3**

The numbers for the area and power consumption are proportional to the number of additions for all the examples. All the designs are constrained to give the minimum possible delay. Therefore Synopsys tries to allocate as many adders as needed to achieve the minimum delay. As a result, the fewer the number of additions, the fewer is the number of adders that are required, which explains the area reductions. Dynamic power consumption is directly proportional to the amount of switched capacitance, which is directly proportional to the number of adders that are switching every clock cycle.

3.2.4 Limitations of these methods

The methods for redundancy elimination for linear arithmetic expressions (described in Sections 3.2.1 and 3.2.2) do not guarantee the least possible number of additions. One of the reasons for this is the use of the greedy heuristic instead of an optimal algorithm. Another reason is the exponential number of ways a constant multiplication can be implemented. It may be better to use different representations for different constants, which may lead to the detection of more common subexpressions. Another technique that can reduce the complexity of constant multiplications is the factoring of the constants. For example, for performing the multiplication 105*X, the constant 105 can be decomposed as 105 = 15*7 = (16-1)*(8 -1). Therefore the constant multiplication can be performed as 105*X = ((X<<4 – 1) <<3 – 1). This is the optimal solution to this problem, and cannot be achieved by any of the number representation schemes. But factoring such numbers is very hard for large constants, and is generally not considered.

3.3 Optimizing constant multiplications in general polynomial expressions

The polynomial transformation for constant multiplication (L variable) helps us to perform optimizations on arithmetic expressions consisting of shift and add/subtract operations, as well as multiply operations. Most polynomial expressions found in practice have constant coefficients. Sometimes it is important to eliminate

as many multiplications as possible, especially in hardware implementations. In such cases, the constant multiplications can be decomposed into shifts and additions. The number of additions can be reduced by finding common subexpressions among the operations resulting from the constant multiplications. Section 3.1 presented optimization techniques that could handle only add/subtract and multiply operations. Section 3.2 presented techniques for expressions that consisted of add/subtract and shift operations. This section explains how these techniques can be extended to handle expressions consisting of all these operations (Add, Subtract, Multiply and Shift). A modification of the two-term extraction algorithm (Section 3.2.2) is used to perform this optimization.

Problem formulation

Given a set of polynomial expressions where the constant multiplications have been decomposed into shifts and additions, reduce the number of operations. The relative costs of the different operators have to be taken into account while performing the reduction.

Changes to the two-term CSE algorithm explained in Section 3.2.2

1) Divisor extraction procedure:

The original algorithm for divisor generation divided any two terms in an expression by the common cube (polynomial expressions) and common exponent of L (linear expressions). For this problem, we need to check if the two terms have the

same **variable cube**[3]**.** If the two cubes have the same variable cube, then it implies that these terms have been obtained by the decomposition of a constant multiplication into shifts and additions. In such a case, these terms will have to be divided by their minimum exponent of L, to get a useful divisor (just like the divisor generation for the Linear arithmetic expressions).

For example consider the two cubes $\{+a^2bL, + ab^2L^2\}$. These two cubes have different variable cubes. So the divisor generated would be obtained by dividing by the common cube abL, and is {a + bL}. For the set of cubes $\{+abL, +abL^2\}$, which have the same variable cube, the divisor is generated by dividing by L, which is the minimum exponent of L to get the divisor {ab + abL}.

2) Calculating value of the divisors

The value of the divisor is a function of the number of multiplications, additions/subtractions and shifts saved by selecting the divisor. A divisor obtained by dividing by variables will save multiplications, since we are factoring the two terms involved in the divisor. For example, the divisor {a + bL} obtained from the terms $\{a^2bL, ab^2L^2\}$ by dividing by the common cube (abL) saves two multiplications. A divisor can also save additions if there are non-overlapping instances of the same divisor in the set of all divisors. The value of the divisor can be calculated as a weighted sum of the savings in the number of operations.

---

[3] A variable cube of a cube is the cube obtained dividing by the exponent of L. So the variable cube of the

3) Single cube divisors

In the original two-term extraction algorithm (Section 3.2.2) there was no concept of single cube divisors. A single cube divisor is a divisor that has only one term. For example a*b can be a single cube divisor. But for polynomials there will be a number of single cube common subexpressions. There are two ways to approach this problem. The simpler method would be to perform extraction using two term divisors as explained in points 1) and 2) above. At the end of the extraction algorithm, single term common subexpressions can be detected by constructing a Cube Variable Incidence Matrix (Section 3.1) and perform rectangle covering on this matrix using the algorithm shown in Figure 3.8.

Another way to approach this problem is to perform both two-term extraction and single term extraction at the same time. To achieve this, single term divisors have to be generated in addition to two term divisors. Single term divisors can be obtained from each term of the set of polynomial expressions by taking every pair of literals from the term. For example, consider the term "abcd". Single term (two-literal) divisors which are every combination of two literals for this term are {a*b, a*c, a*d, b*c, b*d, c*d}. The extraction algorithm can then be performed by considering both the single term divisors as well as the two term divisors.

---

cube a^2bL^2 is a^2b.

Examples

Consider the following expression $P = 6ab^2 + 5a^2bc$.

Decomposing the constant multiplications using our polynomial formulation, we have the expression

$$P = {}_{(1)}ab^2L + {}_{(2)}ab^2L^2 + {}_{(3)}a^2bc + {}_{(4)}a^2bcL^2$$

The subscript for each term denotes the term number. Using the new divisor extraction algorithm, the divisors and co-divisors generated are shown in Figure 1.

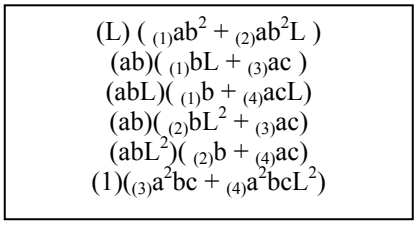


Figure 3.27 Set of divisors for polynomial expression

After performing the two-term extraction procedure followed by the rectangle covering on the Cube Literal Incidence Matrix (CIM), the following optimized polynomial is obtained.

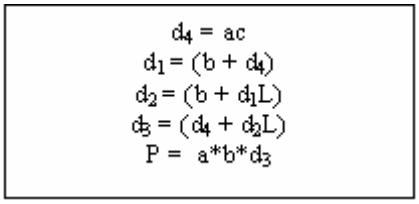


Figure 3.28 Polynomial obtained after two-term and single term extraction

Consider another example where there are two polynomials P_1 and P_2 shown in Figure 3.29. These polynomials initially have eight multiplications and two addition operations. After the optimization, they have three additions but the number of

multiplications is reduced from eight to just two. There are also three shift operations now.

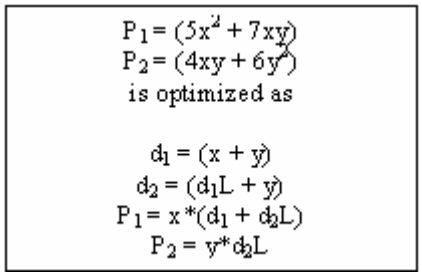


Figure 3.29 Example for optimizing polynomials with constant multiplications

3.4 Optimal solutions for redundancy elimination based on 0-1 Integer Linear Programming (ILP)

In general, finding an optimal solution to the common subexpression elimination problem is NP-Hard. All the works on reducing the number of operations for general purpose applications [19], for polynomial expressions [31-33], for constant multiplications [34, 35], [9, 36], and for integer exponentiation [3, 37] are based on heuristic approaches. Only recently, an optimal solution for a restricted version of the problem using 0-1 Integer Linear Programming [23] was presented. The paper focuses on the sharing of partial terms for the optimization of the multiplier block of the transposed form of the FIR filter. In this configuration, the set of constants can be seen as multiplying a single variable. As a result, the task of finding common subexpressions becomes equivalent to finding common digit patterns in the signed digit (Binary, CSD, MSD) representations.

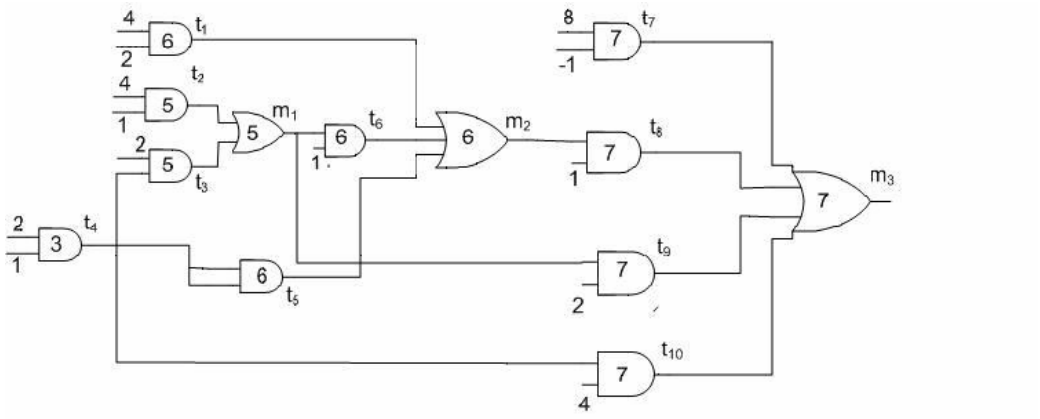


Figure 3.30 AND-OR circuit showing all the partial sums used to obtain constant 7

The method transforms all possible solutions into an AND-OR circuit, where each AND gate represents an addition/subtraction, and each OR gate collects all the different paths for computing a particular value (partial product). For each gate, the Conjunctive normal form (CNF) of the gates's functionality is written out, from which the 0-1 ILP constraint is obtained. The ILP optimization function is then formulated as a minimization of the linear combination of the AND gate outputs. For example, consider the constant multiplication 7*X. The circuit in Figure 3.30 illustrates all the partial sums that add up to the constant 7. Each AND gate has a distinct name t_i and each OR gate has a distinct name m_i. The partial sum computed by each gate is also shown in the figure.

The constant 7 can be obtained in four different ways as (8-1), (6 + 1), (5 + 2) and (4 + 3), and comprise the four inputs to the OR gate m_3 in the figure. Similarly, there

are three ways to compute 6 as (4 + 2), (5 + 1) and (3+3), and is collected by the OR gate m_2 in the figure. The optimization function of the ILP solver is set as:

Minimize $t_1 + t_2 + t_3 + t_4 + t_5 + t_6 + t_7 + t_8 + t_9 + t_{10}$

The final OR gate output m_3 is set to 1 to indicate that one of the 4 choices has to be selected.

From CNF form to 0-1 ILP clauses

The AND gates have an extra input which is either set 1 or 0 by the ILP solver depending on whether the gate is required for the solution. As an example, consider the 3 input AND gate shown in Figure 3.25. Besides the regular inputs a and b, there is a third input p_1. The CNF form for the AND gate output is given by

$$f = (y' + a)(y' + b)(y' + p_1)(y + a' + b' + p_1').$$

The function f has to be set to 1, if the function of the AND gate has to be satisfied, which implies that each clause has to be equal to 1. The ILP clauses for this gate can be written by considering complements y' = 1 – y. For this gate, we have the following ILP clauses shown in Figure 3.31

$$a - y \geq 0$$
$$b - y \geq 0$$
$$p_1 - y \geq 0$$
$$y - a - b - p_1 \geq -2$$
$$0 \leq \{a,b,p_1,y\} \leq 1$$

Figure 3.31 ILP clauses for the AND gate

For the example in Figure 3.30, it is obvious that the minimum cost is obtained by just setting t_7 to 1, which requires just one AND gate, or in other words just one addition (subtraction) to evaluate 7*X, which is done by 8*X – X = X<<3 – X.

3.4.1 Generalizing ILP model for solving a larger class of redundancy elimination problems

The ILP model introduced in the paper [23] can be applied for a more general class of redundancy elimination problem. We can formulate optimal solutions to the problems of polynomial expressions, multi-variable constant multiplications and the integer exponentiation problem. In this section, we show how the ILP model can be extended to the other cases.

We consider the case of polynomial expressions, and derive an algorithm to generate the AND-OR circuit from which the ILP optimization function and constraints can be directly derived. The use of kernels and co-kernels of polynomial expressions helps to reduce the number of solutions that need to be looked up and therefore reduces the number of optimization variables in the ILP formulation. The algorithm for generating the AND-OR circuit (GenCkt) is described using pseudo code in Figure 3.32. The algorithm is a recursive algorithm, and starts with the original polynomial, and recursively finds solutions for different subexpressions and combines them. When there is more then one polynomial, the same procedure is repeated for all the polynomials.

```
Node GenCkt( {Ti} )
{
        // {Ti} = Set of terms representing the polynomial expression
        //  This function returns a node representing the gate output that computes the
      polynomial

        Term CK = CoKernel( {Ti} ) // returns the co-kernel of this polynomial
        {Ti} = {Ti}/(CK) ;  // dividing the polynomial by the kernel

        result = HASH({Ti} );
        if(result != φ) // checking to see if this polynomial has been generated already
        {       CKNode = GenCkt(CK); // generating a node for the Co-Kernel
                result  = CombineAND(result, CKNode, x)  //combine with co-kernel
                return result;
        }
        ORGate = new Node;

        // Generate all possible partitions of the set of terms {Ti}
        { {Ti}^A, {Ti}^B }  = GenAllCombinations( {Ti} );

        ∀ Combinations {Ti}_j^A and {Ti}_j^B

        {       Node N1 = GenCkt( {Ti}_j^A ); // Creating the circuits for each combination

                Node N2 =  GenCkt( {Ti}_j^B );

                Node ANDi;

                if( |Ti| == 1) // if the polynomial had only 1 term
                        ANDi = CombineAND(N1, N2, x);
                else
                        ANDi  =  CombineAND(N1, N2, +);
                ORGate->Insert(ANDi); // Insert ANDi as one of the inputs of ORGate
        }

        if( CK != '1')
        {
                CKNode = GenCkt(CK);
                result = CombineAND(ORGate, CKNode, x);
        }
        else
                result = ORGate;

        HashTable->Insert(result); // Inserting the result in the Hash Table
        return result;
}
```

Figure 3.32 Algorithm for generating AND-OR circuit for a polynomial

The algorithm has a set of terms representing the polynomials as input. The co-kernel CK of the polynomial is equal to a term that can divide all the terms of the polynomial. The polynomial is then divided by this co-kernel to get the kernel. By Theorem[1] (Section 3.1) we know that in order to detect multiple term common subexpressions, it is sufficient to only form intersections among kernels. Also we only consider kernel and co-kernel factorizations as opposed to all possible algebraic factorizations, since the former gives the least number of operations for computing the polynomial. The polynomial is then Hashed to see if it has already been computed, and if it has been computed, it is combined with the node representing the co-kernel and the resultant node is then returned. The function CombineAND returns a node that represents an AND gate combining the two nodes. The AND gate may represent either a multiplication or an addition, and is passed as an argument to the CombineAND function. An OR gate (ORGate in the algorithm) is created to collect all the different ways of computing the polynomial.

The function GenAllCombinations takes as input a set of terms and generates all possible combination pairs of $\{T_i\}$ terms. The number of combination pairs for N terms is given by the formula C(N) as

$$C(N) = \binom{N}{1} + \binom{N}{2} + \cdots\cdots \binom{N}{\lfloor N/2 \rfloor} = o(2^{N-1})$$

Equation 3-6

If the number of terms of the polynomial is only one ($|\{T_i\}| = 1$), then the function returns all possible combination of the literals in the term. If the term has N literals,

then the number of combinations is given by the same equation C(N) as in Equation 3-6.

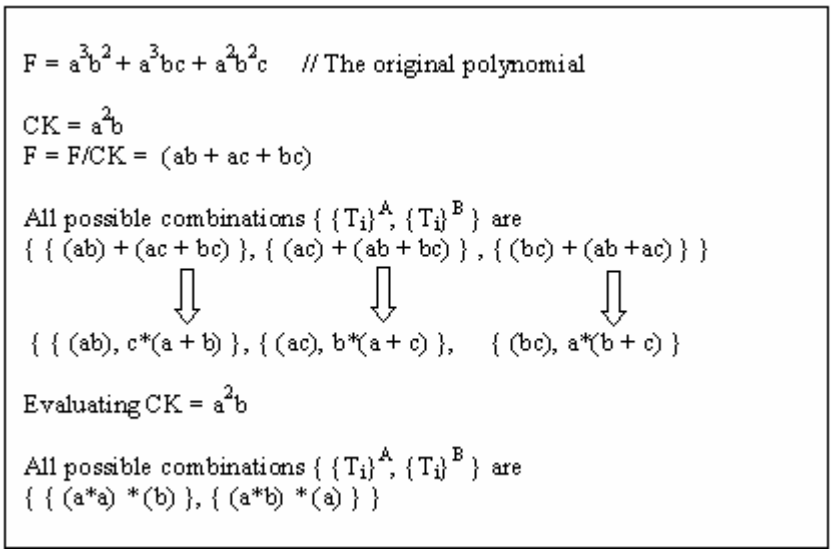


Figure 3.33 Example showing the steps in generating the AND-OR circuit

The function GenCkt is recursively called for each polynomial of each combination pair and the two polynomials are then combined using the CombineAND function. The resultant node ANDi is then inserted into the OR gate as one of the ways of computing the polynomial. After combining all the combination pairs, the Co-kernel node (CKNode) is then combined with the OR gate to obtain the resultant polynomial node. This node is then inserted into the global hash table.

Figure 3.33 llustrates the working of the algorithm for an example polynomial expression. For this polynomial, the co-kernel a^2b is first extracted. After dividing the original polynomial by this co-kernel, we obtain the polynomial (ab + ac + bc).

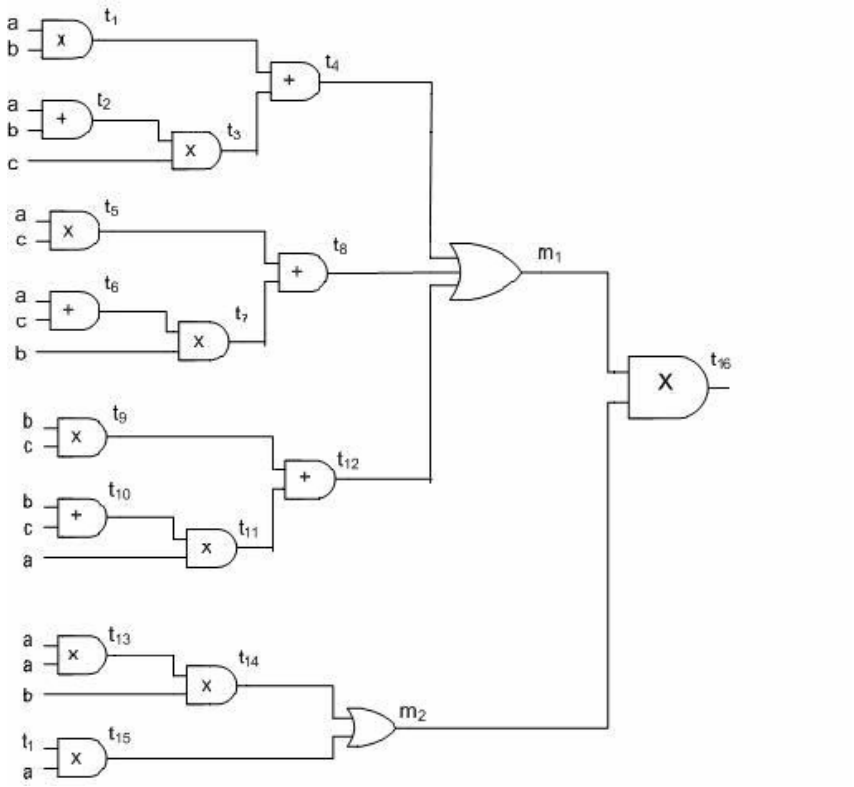


Figure 3.34 AND-OR circuit for the example polynomial

There are 3 possible combinations for this polynomial, and the recursive evaluations of these expressions are shown in the figure. Finally, the AND-OR circuit generated is shown in Figure 3.34.The optimization function for this circuit can be written as

Minimize $\sum_{i=1}^{i=16} t_i$, given that the final output $t_{16} = 1$.

The other constraints can be generated as described previously, and are not shown here for convenience. Intuitively, we can see that the solution for this problem can be obtained by setting $\{t_1, t_2, t_3, t_4, t_{15}, t_{16}\} = 1$, which corresponds to the solution

$$t_1 = (a * b)$$

$$F = a*t_1 * (c * (a + b) + t_1)$$

This solution has four multiplications and two additions, and is the implementation with the least cost.

Complexity of the algorithm

Besides the complexity of solving the ILP formulation, the complexity of generating the AND-OR circuit is exponential in the number of terms in a polynomial. This is because each polynomial has a complexity of $o(2^{N-1})$ in generating all the combinations. This involves finding combinations among N-1 terms which is $o(2^{N-2})$ and so on. The final complexity is $o(2^{N-1} + 2^{N-2} + 2^{N-3} + \ldots\ldots 1) = o(2^N)$.

This method can also be extended to the problem of finding common subexpressions in linear arithmetic expressions (Section 3.2). The algorithm for this case is very similar to the algorithm shown in Figure 3.32, but the operations now consist only of additions.

4 Related Work

This chapter presents some related work in the synthesis and optimization of arithmetic computations. The related work is divided into two parts. The first part presents the optimization techniques in modern software compilers that are applied for general purpose programs (and not just arithmetic expressions). Various techniques for redundancy elimination used in the compilers are presented. The second part presents the works in the hardware synthesis of arithmetic computations.

4.1 Dataflow optimizations in modern software compilers

This section introduces some of the common dataflow optimizations that are performed in almost all modern optimizing software compilers. The optimization process begins with dataflow analysis, which provides global information about how a procedure or a larger segment of a program manipulates the data. The information provided by dataflow analysis enables the application of the optimizations such as local and global common subexpression elimination, constant propagation, strength reduction, loop invariant code motion etc. which are described in brief in this section. For example, the constant propagation analysis procedure will seek to determine if all assignments to a particular variable evaluate to the same constant at all times. The dataflow analysis information is stored in data structures such as the Define Use (DU) chains to hold information about the definitions and uses of all

variables in the procedure. This section presents some of the main dataflow optimizations. A detailed description of these data structures and the procedure for performing dataflow analysis can be found in compiler texts such as [19]. The transformations are typically performed on an intermediate form of the code such the flowgraph shown in Figure 4.1. The flowgraph is in the form of a Control Dataflow Graph (CDFG), where each node represents a basic block[4]

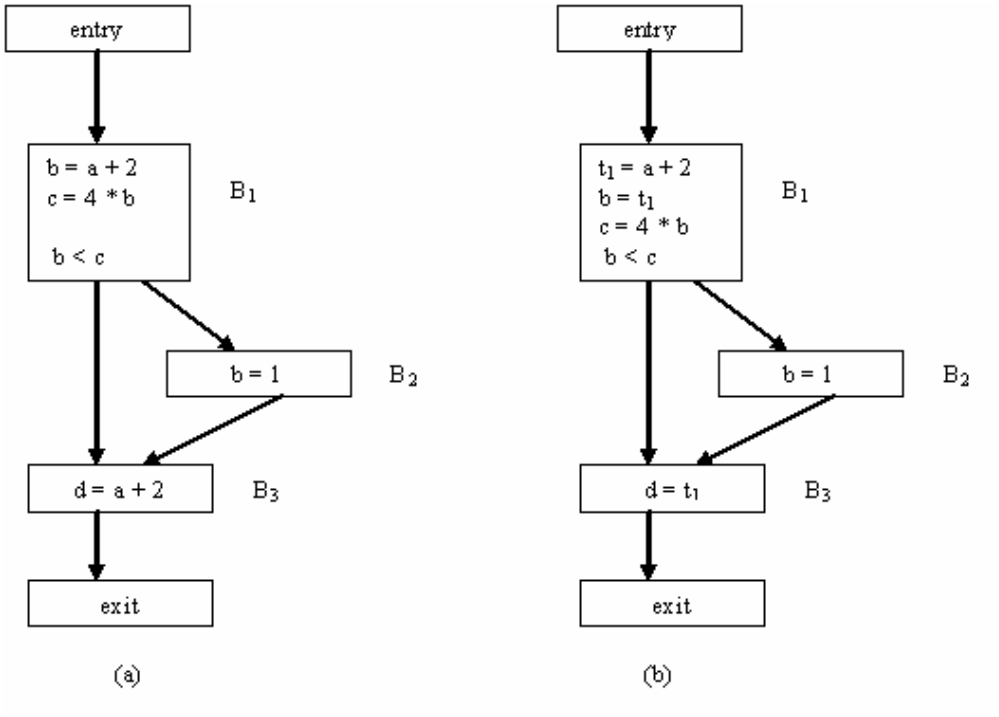


Figure 4.1 Example of doing common subexpression elimination (a) Original flowgraph (b) flowgraph after eliminating common subexpression (a + 2)

[4] A basic block is defined as a set of instructions with a single entry point and a single exit point such that all the instructions can be executed sequentially without any control between them.

4.1.1 Common Subexpression Elimination (CSE)

An occurrence of an expression in a program is a common subexpression if there is another occurrence of the expression whose evaluation always precedes this one in execution order and if the operands of the expression remain unchanged between the two evaluations. CSE can be performed both locally within the basic blocks and globally across the whole CDFG, though the global CSE procedure can detect all the commons subexpressions that the local CSE can. The algorithm for local CSE works within single basic blocks. The method essentially keeps track of available expressions block (called AEB) i.e. those expressions that have been computed so far in the block and have not had an operand changed since then. The algorithm then iterates through the basic block, adding entries to and removing them from the AEB as appropriate, inserting instructions to save the expressions' values in temporaries, and modifying the instructions to use the temporaries instead. The iteration stops when there can be no more common subexpressions detected.

The global CSE procedure operates on the whole flowgraph, and solves the dataflow problem of available expressions. An expression *exp* is said to be available at the entry to a basic block if along every control path from the entry to this block there is an evaluation of exp that is not subsequently killed by having one or more of its operands assigned a new value. In determining what expressions are available, EVAL(i) is used to denote the set of expressions evaluated in block i that are still available at its exit and KILL(i) to denote the set of expressions that are killed by block i. EVAL(i) is computed by scanning block i from beginning to the end,

accumulating the expressions evaluated in it and deleting those whose operands are later assigned new values in the block. Using the terminology AEin(i) and AEout(i) to represent the set of expression that are available on entry to and exit from block i respectively, the system of dataflow equations can be written as:

$$AEin(i) = \bigcap_{j \in \Pr ed(i)} AEout(j)$$

$$AEout(i) = EVAL(i) \bigcup (AEin(i) - KILL(i))$$

Equation 4-1

4.1.2 Value numbering

Value numbering is another method that is useful for redundancy elimination and is based on symbolic evaluation of expressions. It associates a symbolic value with each computation without interpreting the operation performed by the computation, but in such a way that any two computations with the same symbolic value always compute the same value. Though value numbering has the same effect as that of CSE for basic blocks, there are differences for global transformations as pointed out by the examples in Figure 4.2. In Figure 4.2a the variables j and l are proved equal by value numbering, but cannot be proved equal by CSE because there is no common subexpression. On the other hand, the example in Figure 4.2b shows an example where global CSE is able to determine that the assignment (2*i) to variables l and i are the same, but value numbering cannot determine that they are the same because the variables l and j are not assigned the same value at all times.

The original formulation of value numbering operated on individual basic blocks, but has been extended to global form [38] [39]. To do value numbering in basic blocks, hashing is used to partition the expression that are computed into classes. Upon encountering an expression, its hash value is computed. If it is not already in the sequence of expressions with that hash value, it is added into the sequence. The hash function and the expression matching function are defined to take commutativity of the operator into account.

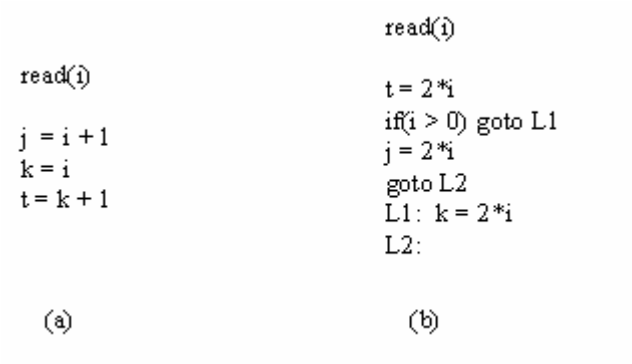


Figure 4.2 Showing the differences between the capabilities of Value numbering and CSE

4.1.3 Loop Invariant Code Motion

Loop Invariant code motion recognizes computations in loops that produce the same value on every iteration of the loop and moves them out of the loop. Most of these instances occur in addressing computations that access elements of arrays. These calculations are not exposed to optimization until they are translated to the intermediate form. Identifying loop invariant computations is fairly simple. Assuming that we have performed control flow analysis and have identified loops,

and that we know which definitions can affect given uses of a variable, then the set of loop invariant instructions in a loop can be defined inductively as follows. An instruction is loop-invariant if, for each of its operands:

(i) The operand is constant

(ii) All definitions that reach this use of the operand are located outside the loop, or

(iii) there is exactly one definition of the operand that reaches the instruction and that definition is an instruction inside the loop that is itself loop-invariant.

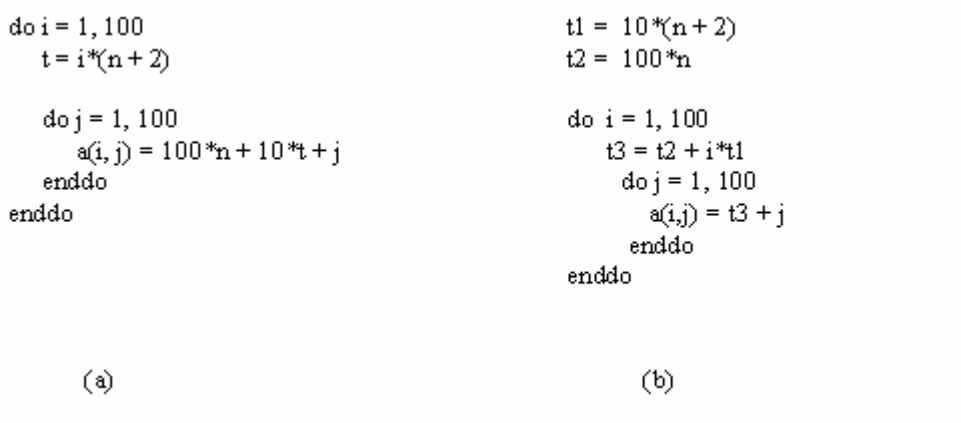


Figure 4.3 (a) Example of a loop having loop invariant computations and (b) the code after transforming it

Figure 4.3a shows a piece of Fortran code where there is some loop invariant computation. The code after moving the loop invariant code is shown in Figure 4.3b. This transformation saves about 10,000 multiplications and 5000 additions in about 5000 iterations of the loop.

4.1.4 Partial-Redundancy Elimination

Partial-redundancy elimination (PRE) is an optimization that combines global common subexpression elimination and loop-invariant code motion and also makes some additional code improvements. An expression is *partially redundant* at point p if it is redundant along some, but not all, paths that reach p. PRE converts partially-redundant expressions into redundant expressions. The basic idea is simple. First it uses data-flow analysis to discover where expressions are partially redundant. Next, it solves a data-flow problem that shows where inserting copies of a computation would convert a partial redundancy into a full redundancy. Finally, it inserts the appropriate code and deletes the redundant copy of the expression.

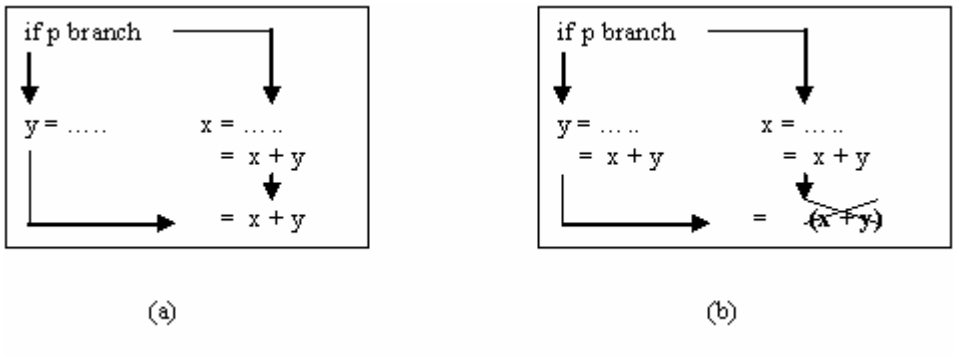


Figure 4.4 Partial Redundancy Elimination (a) Partially redundant (b) Redundant

In Figure 4.4a, the expression (x + y) is available only along the branch on the right, and is therefore only partially redundant at the join of the two branches. Inserting another copy of the expression on the other part makes the computation redundant and allows it to be eliminated, as shown in Figure 4.4b.

Loop-invariant expressions are also partially redundant, as illustrated in Figure 4.5. In Figure 4.5a, x+y is partially redundant since it is available from one predecessor (along the back edge of the loop), but not the other. Inserting an evaluation of (x+y) before the loop allows it to be eliminated from the loop body.

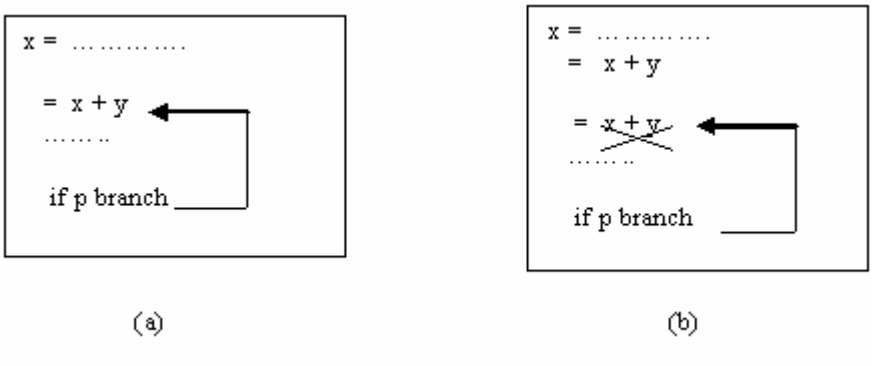


Figure 4.5 PRE can move loop invariant computations

4.1.5 Operator Strength Reduction

Operator strength reduction is a transformation that a compiler uses to replace costly (strong) instructions with cheaper (weaker) ones. A weak form of strength reduction replaces 2*x with either x + x or x<<1. The more powerful form of strength reduction replaces an iterated series of strong computations with an equivalent series of weaker computations. The classis example replaces certain multiplications inside a loop with equivalent additions. This case arises frequently in loop-based array address calculations, and many other operations can be reduced in this manner.

4.2 Hardware synthesis optimizations

This section first presents some work on the synthesis of polynomial expressions. Traditionally, polynomial expressions were prominent only in scientific applications and simulations, which were carried out by supercomputers or work stations. This explains in part why there has not been a lot of work in this area. But due to the increasing demand for multimedia applications such as 3-D computer graphics, there has been some interest recently in finding out the best way to implement polynomial expressions. 3-D graphics are popular additions to the portable electronic devices such as cellphones and PDAs, and regularly use polynomials to model surfaces, curves and textures. Furthermore, DSP transforms such as the DCT, and DFT are used frequently in audio and video compression algorithms.

4.2.1 Implementing polynomials using the Horner form

The Horner form is one of the most popular ways of evaluating polynomial approximations of trigonometric functions, and is the default method in many libraries including the GNU C library. The Horner form of evaluating a polynomial transforms the expression into a sequence of nested multiplications and addition, which is suitable for sequential machine evaluation, using Multiply Accumulate (MAC) operations. For example, a polynomial in variable x

$p(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + a_3x^{n-3} + ...+ a_n$ is evaluated as

$p(x) = (.....((a_0x + a_1)x + a_2)x + ...+ a_{n-1}) + a_n$

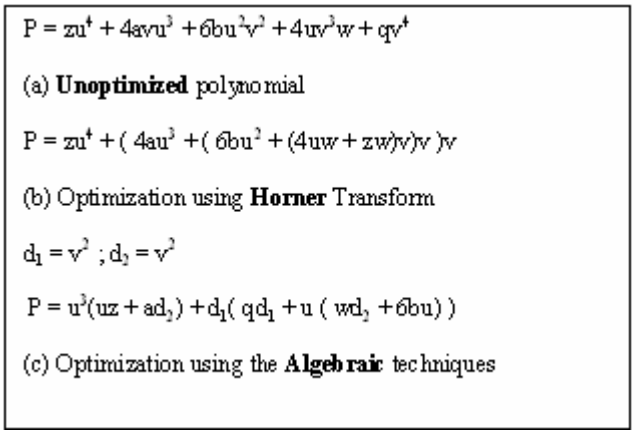


Figure 4.6 Optimizations on Quartic-spline polynomial

The disadvantage of this method is that it optimizes only a single polynomial expression at a time and does not look for common subexpressions among a set of polynomial expressions. Furthermore, it is not good at optimizing multivariate polynomial expressions, of the type found in computer graphics applications [28]. For example, consider the polynomial shown in Figure 4.6, which is the quartic polynomial used in 3-D computer graphics for modeling textures. The original polynomial consists of 23 multiplications. The Horner form for this polynomial was obtained using Maple™ [40]. It has 17 multiplications. But, using the algebraic methods that are presented in this thesis, the polynomial can be evaluated using only 13 multiplications.

4.2.2 Synthesis and optimization for constant multiplications

The problem of reducing the number of operations after decomposing a constant multiplication into shifts and additions has been studied for a long time [41-43]. This early work led to the development of a novel number representation scheme, called the Canonical Signed Digit (CSD) [44]. The CSD representation, on an average reduces the number of non-zero digits in a constant representation by 33% compared to conventional two's complement representation. This leads to a direct reduction in the number of shifts and additions. CSD multipliers have been used for low complexity FIR filter design [45]. There have been a number of works on optimizing constant multiplications in FIR filter design by efficient encoding of coefficients and sharing of common computations [10, 11, 46, 47].

The first work that addressed the different issues and suggested solutions to the multiple constant multiplication (MCM) problem was in [9]. This work used an iterative bipartite matching algorithm, where in each iteration, the best 2 matching constants were selected and the matching parts eliminated. This bipartite matching algorithm was combined with a preprocessing scaling of the constants for an increased solution space. Using an appropriate scaling factor for the constants can reduce the number of non-zero digits in the representations of the constants, leading to a fewer number of operations. Though the scaling of the constants is an effective technique in reducing the number of operations, the exponential search space for finding the right scaling factor does not make it a practical approach at solving the problem. Another drawback of the presented methods is that the bipartite matching

algorithm is not the best way to approach the problem of extracting common subexpressions. A more global approach is required to extract the best common subexpressions. Furthermore, the algorithm does not consider common subexpressions among shifted forms of the constants. For example their algorithm would not be able to detect the common subexpression "101" between the two constants "0101" and "1010".

Dempster and McLeod [36] used the concept of addition chains (described in Section 3.4), and by enumerating all possible adder-chains with 4 or less adders found that multiplications by all constants ip to 2^{12} can be computed using only four additions. Gustaffson in [48] generalized the result to five adders, and found that 5 adders are sufficient to compute multiplications by constants up to 2^{19}. Though very expensive, this is a very strong result, and can be used in the optimal construction of constant multiplications, where the maximum size of the constants is 2^{19}. It is widely used in the construction of the transposed form of the FIR filters, since that form has constants multiplying only a single variable. However, this method does not work when there are expressions involving multiple variables.

In [13, 49] a greedy optimization technique to minimize the area of linear digital systems using a combination of common subexpression elimination and modification of multiplier coefficients was presented. The method is mainly based on the fact that the amount of logic required by a coefficient multiplier is dependant on the value of the coefficient, and transforms the linear system by a splitting of coefficients such that the overall area is reduced. The two main methods for number splitting are row-based number splitting (RBNS) and column-based number splitting

(CBNS). In RBNS, two coefficients in the same row of the constant matrix c_{ip} and c_{iq} are modified. For example in the state equation $s_i(t+1) = c_{i1}x_1(t) + c_{i2}x_2(t) + \ldots + c_{ip}x_p(t) + \ldots + c_{iq}x_p(t) \ldots + c_{iZ}x_Z(t)$ is rewritten as

$s_i(t+1) = c_{i1}x_1(t) + c_{i2}x_2(t) + \ldots + c_{ip}[x_p(t) + x_q(t)] + .. + \delta x_q(t) + \ldots + c_{iZ}x_Z(t)$, where $\delta = c_{iq} - c_{ip}$. The computation tree for $s_i(t+1)$ has been affected by the conversion of two multiplications by two multiplications and two additions. If the cost of the additional addition is offset by the decrease in the multiplier complexity due to the choice of the multiplier coefficients, then overall hardware savings can be achieved.

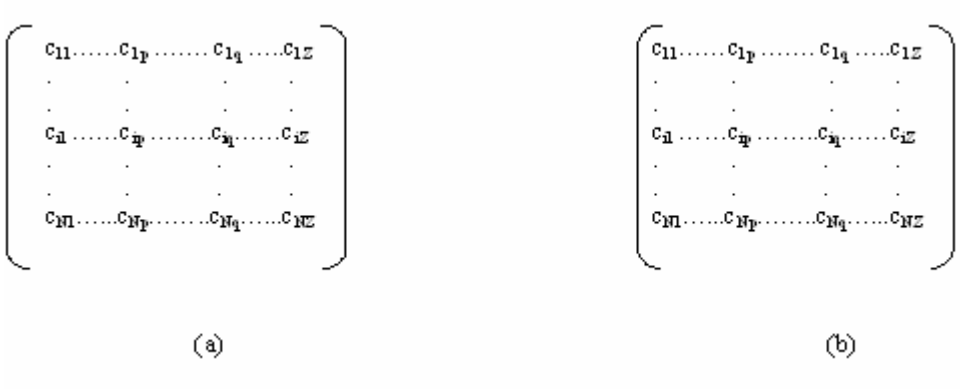


Figure 4.7 Matrix splitting (a) Original matrix (b) Matrix after constant splitting

The above figure shows the effect of the RBNS transform to the i^{th} row of the constant matrix C. It is apparent that the addition of the row R = [0 …. c_{ip} … c_{ip} … 0] to the i^{th} row of the matrix C' yields the original matrix C. Each time RBNS is performed, the corresponding row R is appended to an augmented matrix AG.

Similar to RBNS, a column-based number splitting (CBNS) is performed where elements in the same column of the constant matrix are affected. The optimization algorithm consists of a greedy steepest descent algorithm, where the

benefits of both RBNS and CBNS are recorded for each element in the matrix, and the transformation with the greatest benefit is carried out.

A new approach to optimizing linear systems by a multi-level decomposition of the system matrices was presented in [50].The hardware optimization algorithm decomposed a system constant matrix M into a product of 2z + 1 matrices $M = M_z M_{z-1} M_{z-2} \ldots M_0 \ldots M_{-z+1} M_{-z}$. If the matrix M is an n_{out} x n_{in} matrix, then M_{-z} is in general a n_{-z+1} x n_{in} matrix, M_{-z+1} is a n_{-z+1} x n_{-z} matrix and so on and M_z is a n_{out} x n_{z-1} matrix where n_i represents the number of rows in M_i for all integer $-z \leq i \leq z$. Figure 4.8 shows the decomposition of the original matrix (a) into a transformed system (b) by matrix splitting. The matrix transformation is obtained by row and column transformations.

$$\begin{pmatrix} s_0(t+1) \\ s_1(t+1) \\ s_2(t+1) \end{pmatrix} = \begin{pmatrix} 0.667 & 1 & -0.333 & 0.333 \\ -1 & -0.667 & 0.333 & -0.333 \\ 1 & 0.333 & -0.667 & 0 \end{pmatrix} X \begin{pmatrix} s_0(t) \\ s_1(t) \\ s_2(t) \end{pmatrix}$$

(a)

$$\begin{pmatrix} s_0(t+1) \\ s_1(t+1) \\ s_2(t+1) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} X \begin{pmatrix} 0 & 1 & 0.5 & 0.3 \\ -0.3 & 0 & 0 & 0 \\ 1 & 1.3 & 0 & 0 \end{pmatrix} X \begin{pmatrix} 1 & -1 & 0 & 0 \\ 0 & 1 & -0.5 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 0 & -1 & 1 \end{pmatrix} X \begin{pmatrix} s_0(t) \\ s_1(t) \\ s_2(t) \end{pmatrix}$$

(b)

Figure 4.8 Example of matrix splitting transformation (a) Original System (b) Transformed system

This matrix splitting was combined with shift-and-add decomposition of the constant multiplications and an algorithm to extract common subexpressions in these computations. The algorithm was based on finding common digit patterns in the set of constants multiplying a single variable at a time. The algorithm uses bipartite matching of the constant digit patterns, similar to the algorithm in [9] but the matching has been extended even to the shifted forms of the digit patterns. For example this algorithm can detect the common pattern "101" among the patterns "0101" and "1010". A major shortcoming of this method is in its inability to detect common subexpressions involving more than one variable.

Very recently, an optimal formulation for the constant multiplication problem was formulated [23] using Integer Linear Programming (ILP). This was the first attempt at deriving an optimal solution to the common subexpression elimination problem. The details of the formulation using Circuit SAT and formulating the ILP constraints are described in detail in Section 3.4.

There are many cases where we need to perform multi-operand addition, such as multiplication, where we need to perform the summation of partial products. The propagation of carries is the biggest contributor to the delay of addition, and Carry Save Adder (CSA) is an attractive architecture choice for such situations. Multi-operand addition can be implemented using a tree of CSAs, where the carry propagation is delayed to the final step. Each CSA takes three numbers and produces the sum output and the carry outputs separately. Hence CSA is also known as a 3:2 compressor. Figure 4.9 shows a CSA tree to add 6 inputs, where the 6 numbers are reduced to 2 numbers, which are then added using a fast adder such as a Carry

Lookahead Adder (CLA). Since the CSAs are able to reduce three numbers to two numbers, they are also known as 3:2 compressors.

The most recent work on using CSAs is in [51], where the authors have developed a technique to restructure dataflow graphs in programs to cluster as many arithmetic operators as possible and then implement them using CSA trees. The major contribution of the paper was in making most use of CSA structures for high speed, but there was no attempt in optimizing the CSA structures. [52] is one of the early works in this area, and presents a set of transformations to transform a dataflow graph to maximize the utilization of CSAs. In [53], an optimal algorithm for designing a CSA structure with shortest delay is presented, given the arrival times of all inputs.

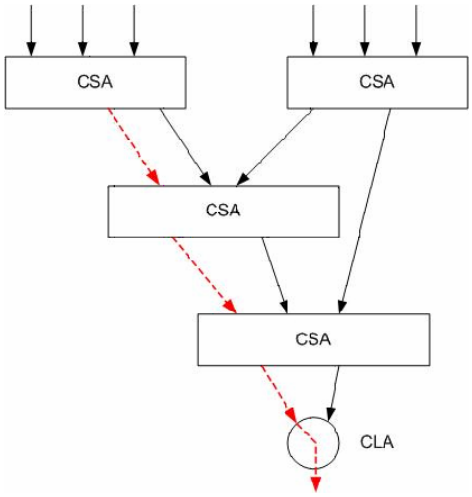


Figure 4.9 Carry Save Adder (CSA) tree for performing multi-operand addition

In [54], algorithms for timing and low power driven synthesis of CSA structures are presented, where the optimizations are done at bit-level. For low power

synthesis, the authors consider switching probabilities of the inputs, and allot the inputs to the CSAs to minimize the switching probabilities of various signals. In [55], some layout driven optimizations like converting half adders and merging them into full adders is suggested for layout improvement. The work described in [56], was one of the earliest contributions in the application of Carry Save arithmetic, where the implementation of high speed Digital Signal Processing structures is investigated. In addition, the author also discusses the use of carry save arithmetic for such procedures as division, modulo multiplication and CORDIC algorithms.

None of these works actually explore transformations such as redundancy elimination in these CSA implementations, which can potentially have a tremendous impact on the area and power of the circuit. We describe methods for reducing the area using three term extraction in Chapter 5. Very recently [57] there has been some work on performing common subexpression elimination for a general class of arithmetic circuits at the bit level. The authors use Reed Muller form for the XOR dominated Boolean circuits, and use an exhaustive common subexpression elimination and factorization algorithm based on Grobner bases. Though such an algorithm should also be able to detect the kind of common subexpressions that we detect at the word level, the exhaustive nature of their algorithm makes it impractical for large circuits. In fact the paper [57] present only small examples in their experimental results, and the biggest example is a 4 x 3-bit multiplier.

5 Hardware Synthesis of Arithmetic Expressions

This chapter presents some of the issues in the synthesis of arithmetic expressions. This chapter first builds upon some of the algorithms presented in chapter 3 and develops a technique for synthesizing high speed Finite Impulse Response (FIR) filters. Some techniques to synthesize high speed arithmetic circuits using Carry Save Addition techniques are then presented. The final section of this chapter discusses how the optimization techniques presented in Chapter 3 can be modified to take into account the architectural and performance constraints. Specifically, some ideas for performing delay aware and resource aware scheduling and optimization are discussed.

5.1 Synthesis of Finite Impulse Response (FIR) filters

Digital Signal Processing (DSP) functions such as filtering and transforms are used in a number of applications such as communications and multimedia. These functions are the major determinants of the performance and power consumption of the whole system, and therefore it is important to have good tools for optimizing these functions. The following equation represents the output of an L tap FIR filter, which is the convolution of the latest L input samples. L is the number of coefficients h(k) of the filter and x(n) represents the input time series.

$$y[n] = \sum_{k=0}^{L-1} h[k] \times x[n-k]$$

Equation 5-1

Field Programmable Gate Arrays (FPGAs) are increasingly used for a wide variety of computationally intensive applications in the DSP and communication area. Due to a rapid scaling of the technology, current generations of FPGAs contain a very high number of Configurable Logic Blocks (CLBs), and are becoming more feasible for implementing a wide range of applications. In this section we concentrate on synthesis of FIR filters on FPGAs.

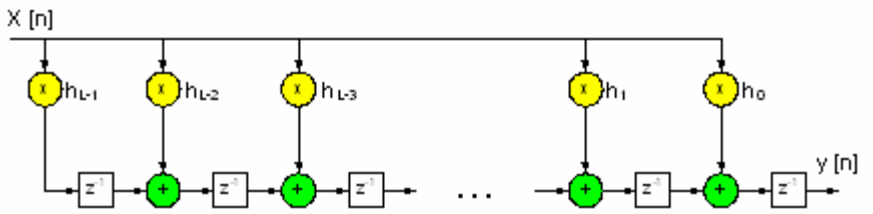


Figure 5.1 MAC FIR Filter block diagram

The conventional tapped delay line realization of this inner product is shown in Figure 5.1. This implementation translates to L multiplications and L-1 additions per sample to compute the result. This can be implemented using a single Multiply Accumulate (MAC) engine, but it would require L MAC cycles, before the next input sample can be processed. Using a parallel implementation with L MACs can speed up the performance L times. A general purpose multiplier occupies a large area on FPGAs. Since all the multiplications are with constants, the full flexibility of a general purpose multiplier is not required, and the area can be vastly reduced using

techniques developed for constant multiplication. Though most of the current generation FPGAs such as Virtex II™ have embedded multipliers to handle these multiplications, the number of these multipliers is typically limited. Furthermore, the size of these multipliers is limited to only 18 bits, which limits the precision of the computations for high speed requirements. The ideal implementation would involve a sharing of the Combinational Logic Blocks (CLBs) and these multipliers. In this paper, we present a technique that is better than conventional techniques for implementation on the CLBs.

An alternative to the above approach is Distributed Arithmetic (DA) which is a well known method to save resources. Using DA method, the filter can be implemented either in bit serial or fully parallel mode to trade bandwidth for area utilization. Assuming coefficients c[n] are known constants, Equation 5-1 can be rewritten as follows:

$$y[n] = \sum_{n=0}^{N-1} c[n] \times x[n]$$

Equation 5-2

Variable x[n] can be represented by:

$$x[n] = \sum_{b=0}^{B-1} x_b[n] \times 2^b \quad x_b \in [0,1]$$

Equation 5-3

where x_b [n] is the b^{th} bit of x[n] and B is the input width. Finally, the inner product can be rewritten as follows:

$$y = \sum_{n=0}^{N-1} c(n) \times \left(\sum_{b=0}^{B-1} x_b[k] \times 2^b \right)$$

Equation 5-4

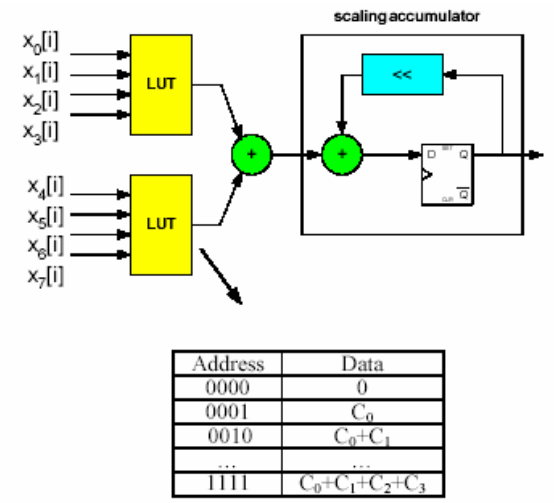
$$= c[0] (x_{B-1}[0]2^{B-1} + x_{B-2}[0]2^{B-2} + \ldots + x_0[0]2^0) + c[1] (x_{B-1}[1]2^{B-1} + x_{B-2}[1]2^{B-2} + \ldots + x_0[1]2^0) + \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots + c[N-1] (x_{B-1}[N-1]2^{B-1} + x_{B-2}[0]2^{B-2} + \ldots + x_0[N-1]2^0)$$

$$= (c[0] x_{B-1}[0] + c[1] x_{B-1}[1] + \ldots + c[N-1] x_{B-1}[N-1])2^{B-1} + (c[0] x_{B-2}[0] + c[1] x_{B-2}[1] + \ldots + c[N-1] x_{B-2}[N-1])2^{B-2} + \ldots\ldots\ldots + (c[0] x_0[0] + c[1] x_0[1] + \ldots + c[N-1] x_0[N-1])2^0$$

The coefficients in most of the DSP applications for the multiply accumulate operations are constants. The partial products are obtained by multiplying the coefficients c_i by multiplying one bit of data x_i at a time in AND operation. The AND functions and adders can be replaced by Look Up Tables (LUTs) that gives the partial product. This is shown in Figure 5.2. Input sequence is fed into the shift register at the input sample rate. The serial output is presented to the RAM based shift registers (registers are not shown in Figure for simplicity) at the bit clock rate which is n+1 times (n is number of bits in a data input sample) the sample rate. The RAM based shift register stores the data in a particular address. The outputs of registered LUTs are added and loaded to the scaling accumulator from LSB to MSB and the result which is the filter output will be accumulated over the time. For an n bit input, n+1 clock cycles are needed for a symmetrical filter to generate the output.

In conventional MAC method with a limited number of MAC engines, as the filter length is increased, the system sample rate is decreased. This is not the case with serial DA architectures since the filter sample rate is decoupled from the filter

length. As the filter length is increased, the throughput is maintained but more logic resources are consumed.



Address	Data
0000	0
0001	C_0
0010	C_0+C_1
…	…
1111	$C_0+C_1+C_2+C_3$

Figure 5.2 A serial DA FIR filter block diagram

Though the serial DA architecture is efficient by construction, its performance is limited by the fact that the next input sample can be processed only after every bit of the current input samples are processed. Each bit of the current input samples takes one clock cycle to process. Therefore, if the input bitwidth is 12, then a new input can be sampled every 12 clock cycles. The performance of the circuit can be improved by modifying the architecture to a parallel architecture which processes the data bits in groups. Figure 5.3 shows the block diagram of a 2 bit parallel DA FIR filter. The tradeoff here is performance for area since increasing the number of bits sampled has a significant effect on resource utilization on FPGA. For instance, doubling the number of bits sampled, doubles the throughput and results in the half the number of clock cycles.

This change doubles the number of LUTs as well as the size of the scaling accumulator. The number of bits being processed can be increased to its maximum size which is the input length n. This gives the maximum throughput to the filter. For a fully parallel implementation of the DA filter (PDA), the number of LUTs required would be enormous. In this work we show an alternative to the PDA method for implementing high speed FIR filters that consumes significantly lesser area and power.

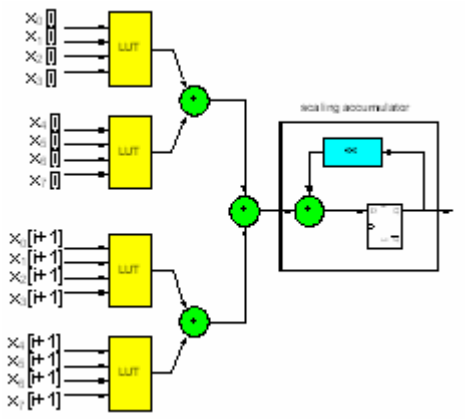


Figure 5.3 A 2 bit parallel DA FIR filter block diagram

A popular technique for implementing the transposed form of FIR filters is the use of a multiplier block, instead of using multipliers for each constant as shown in Figure 5.4. The multiplications with the set of constants $\{h_k\}$ are replaced by an optimized set of additions and shift operations, involving computation sharing. Further optimization can be done by factorizing the expression and finding common subexpressions.

The performance of this filter architecture is then limited by the latency of the biggest adder and is the same as that of the PDA. The filter architecture that we used for synthesizing high speed FIR filters is similar to the one shown in Figure 5.4. Our synthesis technique based on this architecture uses a modified common subexpression elimination algorithm that minimizes both the number of adders as well as the number of latches. Such a scheme produces a performance similar to that produced by the PDA method, with significantly reduced area. Our technique achieves 50% lesser slices and 75% lesser number of LUTs (Look Up Tables) compared to that produced by Xilinx Coregen™ for fully parallel implementations on a Virtex II device. The Xilinx Coregen™ uses PDA method for parallel multiplierless implementations of FIR filters.

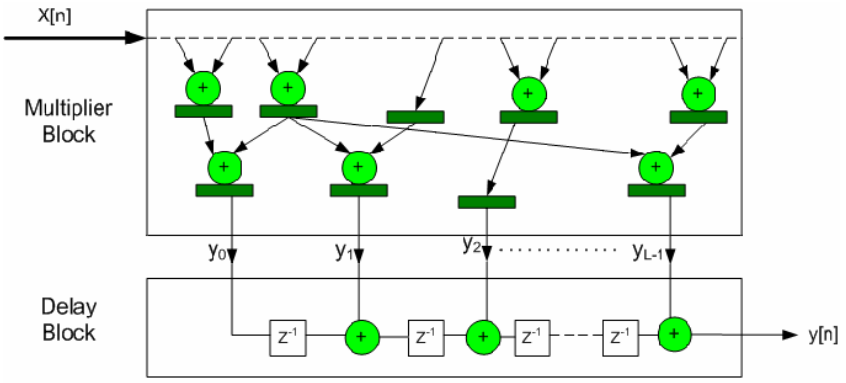


Figure 5.4 Transposed form FIR filter architecture using adders and latches

5.1.1 Filter architecture

The filter can be divided into two main parts, the multiplier block and the delay block, and is illustrated in Figure 5.4. In the multiplier block, the current input

variable x[n] is multiplied by all the coefficients of the filter to produce the y_i outputs. These y_i outputs are then delayed and added in the delay block to produce the filter output y[n].

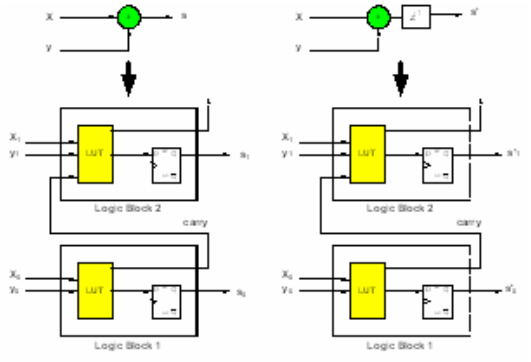


Figure 5.5 Registered adder in an FPGA at no additional cost

All the optimizations are done in the multiplier block, where the constant multiplications are decomposed into registered additions[5] and hardwired shifts. The additions are all done using two input adders which are arranged in the fastest tree structure. By using registered adders, the performance of the filter is only limited by the slowest adder. The registered adder is obtained at very little additional cost from the unregistered adder due to the presence of a D flip flop at the output of the Look Up Table (LUT) in most FPGA architectures as shown in Figure 5.5. Common subexpression elimination is used extensively, to reduce the number of adders, which leads to a reduction in the area. Registers are inserted in the dataflow wherever necessary to synchronize all the intermediate values in the computation.

---

[5] A registered adder is an adder with a register (Flip Flops) at the output. Most FPGA devices have a D Flip Flop at the output of each LUT, which makes it easy to create registered adders.

Performing subexpression elimination can sometimes increase the number of registers substantially, and the overall area could possibly increase. Consider the two expressions F_1 and F_2 which could be part of the multiplier block.

$$F_1 = A + B + C + D$$

$$F_2 = A + B + C + E$$

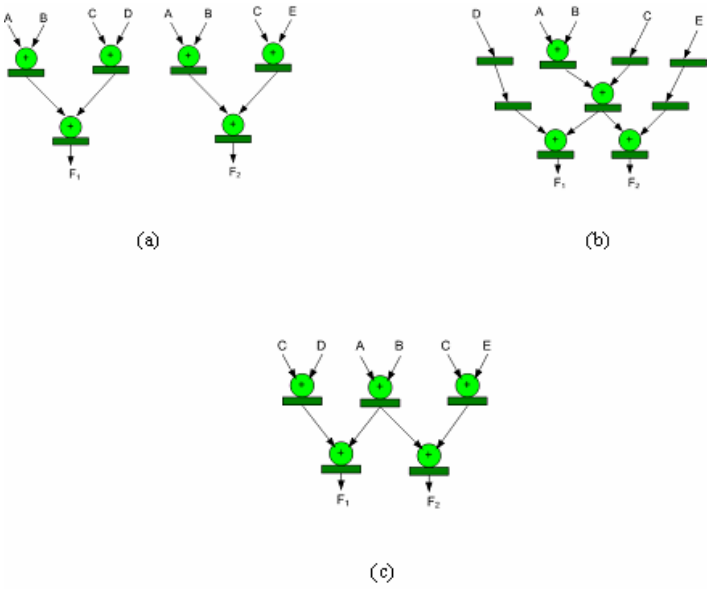


Figure 5.6 Reducing number of registered adders a) Unoptimized expression trees b) Extracting common subexpression (A+B+C) c) Extracting common subexpression (A + B)

Figure 5.6a shows the original unoptimized expression trees. Both the expressions have a minimum critical path of two addition cycles. These expressions require a total of six registered adders for the fastest implementation, and no extra registers are required. From the expressions it can be observed that the computation A + B + C is common to both the expressions. If this

subexpression is extracted, the structure shown in Figure 5.6b is obtained. Since both D and E need to wait for two addition cycles to be added to (A + B + C), we need to use two registers each for D and E, such that new values for A,B,C,D and E can be read in at each clock cycle. Assuming that the cost of an adder and a register with the same bitwidth are the same, the structure shown in Figure 5.6b occupies more area than the one shown in Figure 5.6c. A more careful subexpression elimination algorithm would only extract the common subexpression A + B (or A + C or B + C). The number of adders is decreased by one from the original, and no additional registers are added. This is illustrated in Figure 5.6c. The algorithm for performing this kind of an optimization is discussed in the next section.

5.1.2 Optimization algorithm

The goal of the optimization is to reduce the area of the multiplier block by reducing the number of adders and any additional registers required for the fastest implementation of the FIR filter. The algorithm uses the polynomial formulation of the constant multiplications that was introduced in Section 3.2. The algorithm is a modification of the main algorithm described in detail in Section 3.2. First the minimum number of registers required for the FIR filter using the filter architecture shown in Figure 5.4 is calculated. This is calculated by arranging the original expressions in the fastest possible tree structure, and then inserting registers. For example, for the six term expression F = A + B + C + D + E + F, the fastest tree

structure is obtained with three addition steps, and only one register to synchronize the intermediate values, such that new values for A,B,C,D,E,F can be read in every clock cycle. This is illustrated in Figure 5.7

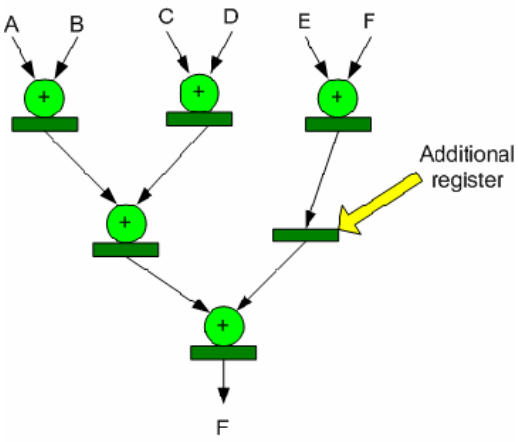


Figure 5.7 Calculating registers required for fastest evaluation

First, the set of all the divisors for the set of expressions describing the multiplier block are generated. An iterative algorithm is then used to perform the optimization. In this algorithm, the divisor that has the greatest **value** is extracted in each iteration. For calculating the value of the divisor, the cost of a registered adder and the cost of an extra (synchronizing) register is considered to be the same. The value of a divisor is then calculated as the number of additions that are saved by extracting the divisor minus the number of extra registers that have to be added. In each iteration, the expressions are rewritten after extracting the best divisor. New divisors that cover the new terms added during the expression rewriting, are then generated. These divisors are added to the dynamic list of divisors. The iterative

algorithm terminates when there is no valuable divisor remaining in the set of divisors.

Consider the expressions shown in Figure 5.6a. Six registered adders are needed for the fastest evaluation of F_1 and F_2. No additional registers are required in this case. Now consider the selection of the divisor $d_1 = (A+B)$. This divisor saves one addition and does not increase the number of registers. Divisors (A + C) and (B + C) also have the same value, but (A+B) is selected randomly. The expressions are now rewritten as:

$$d_1 = (A + B)$$

$$F_1 = d_1 + C + D$$

$$F_2 = d_1 + C + E$$

After rewriting the expressions and forming new divisors, the divisor $d_2 = (d_1 + C)$ is considered. This divisor saves one adder, but introduces five additional registers, as can be seen in Figure 5.6b. Therefore this divisor has a value of - 4. No other valuable divisors can be found and the iteration stops. The final set of expressions is shown in Figure 5.6c.

5.1.3 Experimental validation

The goal of the experiments was to compare the number of resources consumed by the "add and shift" method with that produced by the cores generated by the commercial Coregen™ tool, based on Distributed Arithmetic. The power consumption and performance of the two implementations were also measured and

compared. For the experiments, 10 FIR filters of various sizes (6, 10, 13, 20, 28, 41, 61, 71, 119 and 152 tap filters) were considered. The targeted device for the experiments was the Xilinx Virtex II device. The constants were normalized to 16 digit of precision and the input samples were assumed to be 12 bits wide. For the "add and shift" method, all the constant multiplications were decomposed into additions and shifts and the expressions were optimized using the algorithm explained in Section 3.2. The Xilinx Integrated Software Environment (ISE) was used for performing synthesis and implementation of the designs. All the designs were synthesized for maximum performance.

Table 5-1

Filter (# taps)	Slices	LUTs	FFs	Performance (Msps)
6	264	213	509	251
10	474	406	916	222
13	386	334	749	252
20	856	705	1650	250
28	1294	1145	2508	227
41	2154	1719	4161	223
61	3264	2591	6303	192
71	2232	1517	4314	223
119	6009	4821	11551	203
152	7579	6098	146111	180

Table 5-1 shows the resources utilized for the various filters and the performance in terms of Million samples per second (Msps) for the filters implemented using the add and shift method. Table 5-2 shows the same numbers for the filters implemented using Xilinx Coregen, using the Parallel Distributed Arithmetic (PDA) method. Figure 5.8 plots the reduction in the number of resources, in terms of the number of Slices, Look Up Tables (LUTs) and the number of Flip Flops (FFs). From the results, an average reduction of 58.7% in the number of LUTs,

and about 25% reduction in the number of slices and FFs can be observed. Though our algorithm does not optimize for performance, the synthesis produces better performance in most of the cases, and for the 13 and 20 tap filters, about 26% improvement in performance is observed.

Table 5-2

Filter (# taps)	Slices	LUTs	FFs	Performance (Msps)
6	524	774	1012	245
10	781	1103	1480	222
13	929	1311	1775	199
20	1191	1631	2288	199
28	1774	2544	3381	199
41	2475	3642	4748	222
61	3528	5335	6812	199
119	6484	9754	12539	205
152	8274	12525	15988	199

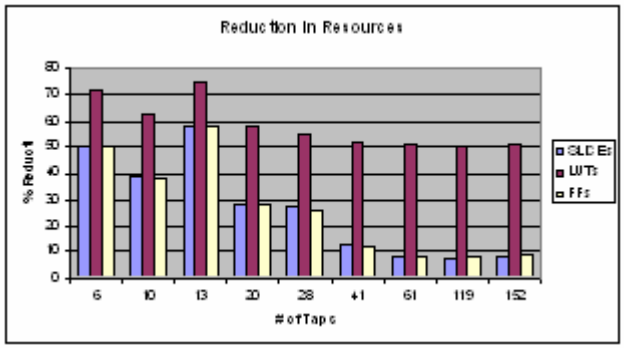


Figure 5.8 Reduction in resources for multiplierles FIR implementation

Figure 5.9 compares power consumption for our add/shift method versus Coregen™. From the results up to 50% reduction in dynamic power consumption can be observed. The quiescent power was not included into the calculation since that value is the same for both methods. The power consumption is the result of

applying the same test stimulus to both designs and measuring the power using XPower tools provided by Xilinx ISE software. The reduction in power consumption is related to the reduction in the number of resources (LUTs and FFs) that are required for the implementation of the filters. The lesser the number of resources, the lesser is the switching activity and hence the reduced power consumption.

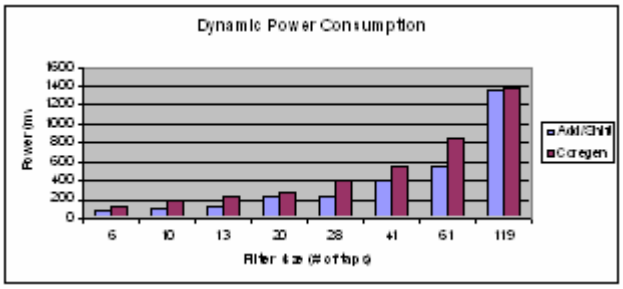


Figure 5.9 Comparing power consumption with Xilinx Coregen™

Coregen™ can produce FIR filters based on the Multiply Accumulate (MAC) method, which makes use of the embedded multipliers. The filters were implemented using the MAC method to compare the resource usage and performance with our "add and shift" method. Due to tool limitations the experiments had to be redone using a Xilinx Virtex IV device. Comparison of the "add and shift" method implementation was made with the Coregen's MAC based filter on this device. Synthesis results for the number of slices and the performance in Msps on the Virtex IV device is presented in Table 5-3.

From the table, it can be seen that the MAC filter uses fewer number of slices compared to the add-shift method, but it also uses the embedded multipliers. The number of multipliers is equal to the number of taps of the filter. For implementing

the 61-tap MAC filter, a higher capacity device had to be used, since the device we were using did not have 61 embedded multipliers. The results show that higher performance is achieved as the filter size increases. The better performance of the "add and shift" method over the MAC method can be attributed to the delay of the MAC operation in the DSP slice of the FPGA as well as the communication latency between the DSP slices and the ordinary FPGA slices. Compared to the MAC method, the "add and shift" method uses only the ordinary slices on the FPGA.

The performance of the "add and shift" method is higher than both the PDA method and the MAC method of Coregen. These impressive gains in area and performance compared to the industry standard tool are only for the fully parallel high speed FIR filters.

Table 5-3 Comparison with Coregen MAC based FIR filter on Virtex IV device

Filter (# taps)	Add Shift Method		MAC filter	
	Slices	Msps	Slices	Msps
6	264	296	219	262
10	475	296	418	253
13	387	296	462	253
20	851	271	790	251
28	1303	305	886	251
41	2178	296	1660	243
61	3284	247	1947	242
119	6025	294	3581	241
151	7623	294	7631	215

5.2 Synthesis of high speed arithmetic circuits

Multi-operand addition is one of the most common operations in datapaths, resulting from Sum of Product computations. There are many ways in which these computations can be implemented depending on the area and speed constraints. For

high speed, these computations are typically implemented using a tree of Carry Save Adders (CSAs) and a final summing Carry Propagate Adder (CPA) for each Sum of Product output. The high speed of such an arrangement is because of the fact that the carry propagation is delayed until the final step. Each CSA compresses 3 operands to two operands (representing the sum and carry bits), and therefore is called as a 3:2 compressor. The delay through a CSA is only equal to the delay through a Full Adder. Considering this delay as unit delay, the delay through the CSA tree is equal to the height of the tree and is given by the equation

$$Delay(N) = \left\lceil \log_{1.5} \left\lceil \frac{N}{2} \right\rceil \right\rceil$$

Equation 5-5

A structure for adding six numbers is shown in Figure 4.9, for which the delay is equal to three units. In a given data-flow graph, arithmetic operators can be clustered together by following certain rules to arrange them over logical operators such as AND, OR etc [51]. These clustered arithmetic operators can then be transformed into compressor trees implemented using CSAs followed by a single Carry Propagate final adder. Using such a scheme has shown to reduce the critical paths of these data-flow graphs by as much as 46% [51].

In this section, a novel algorithm is presented for the minimization of the number of CSAs required in a datapath. The reduction in the number of CSAs is achieved by performing three term extraction. Eliminating common three input terms reduces both the number of CSAs as well as the number of wires required for

implementing the datapath. The main algorithm is augmented for performing delay aware optimization. The optimizations are based on expressions that are in the Sum of Products form, where all the multiplications are with constants. The polynomial model (introduced in Section 3.2) is used for the constant multiplications. Firstly, a set of all potential 3-term common subexpressions called ***divisors*** are generated. An iterative greedy algorithm is then used for performing the reduction.

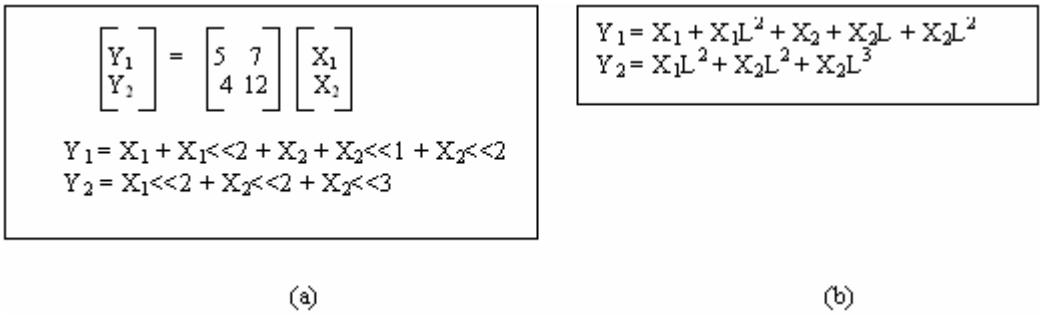


Figure 5.10 Example expressions (a) Original expressions b) Expressions using polynomial transformation

An example constant matrix multiplication shown in Figure 5.10a is used to explain the technique. The expressions for Y_1 and Y_2 are rewritten after decomposing the constant multiplications into additions and shift operations. Figure 5.10b shows the expressions written using the polynomial transformation. The CSA tree structure for the original expressions are shown in Figure 5.11a and the optimized structure after eliminating a common three term expression is shown in Figure 5.11b.

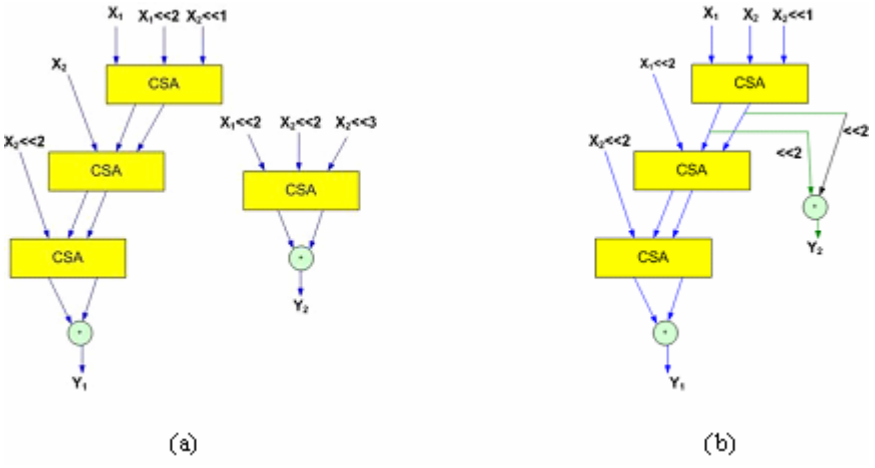


Figure 5.11 CSA trees (a) Original expressions (b) Expressions after sharing

5.2.1 Three-term divisor extraction

Divisors are extracted from every expression by considering every combination of three terms and then dividing by the minimum exponent of L. For example, consider the expression Y_2 in Figure 5.10. The minimum exponent of L in this case is L^2. Dividing by L^2 gives us the divisor $d = X_1 + X_2 + X_2L$. The number of divisors in an expression with N terms is $\binom{N}{3}$. Figure 5.12 shows the algorithm for extracting three-term divisors.

The importance of these three-term divisors is illustrated by the following theorem.

Theorem: There exists a three-term common subexpression iff there exists a non-overlapping intersection among the set of three-term divisors.

```
Divisors({Pi})
{
   {Pi} = Set of expressions in polynomial form;
   {D} = Set of divisors and co-divisors = {Φ};

  for (every expression Pi in {Pi})
  {
    for (every combination of 3 terms (ti, tj, tk,) in Pi)
    {
      MinL = Minimum exponent of L in(ti, tj, tk,); // co-divisor
        ti^1 =  ti/MinL;
        tj^1 =  tj/MinL;
             tk^1 = tk/MinL;
        d  = (ti^1 + tj^1 + tk^1); // divisor;
       {D} = {D} ∪ (d, MinL);
      }
  }

 return {D};

}
```

Figure 5.12 Algorithm for extracting three-term divisors

This theorem states that there is a three-term common subexpression if and only if there are at least two non-overlapping divisors that intersect. Two divisors are said to **intersect**, if their absolute values are equal. For example, $(X_1 + X_2 + X_2<<1)$ intersects with $(X_1 + X_2 + X_2<<1)$. Two divisors are said to be **overlapping** if at least one of the terms from which they are derived is common. For example, consider the expression $F = X_1 + X_1<<2 + X_1<<4 + X_1<<6 = X_1 + X_1L^2 + X_1L^4 + X_1L^6$. From the first three terms the divisor $d_1 = X_1 + X_1L^2 + X_1L^4$ is obtained. From the last three terms the divisor $d_2 = (X_1 + X_1L^2 + X_1L^4)$ is obtained by dividing those

three terms by L^2. Even though these divisors (d_1 and d_2) intersect, they are said to overlap since two of the terms from which they are derived are common.

The proof of this theorem is quite straightforward.

Proof:

(If case): If M of the divisors in the set of all divisors intersect and are non-overlapping, then there are M instances of the same three-term expression in the set of expressions.

(Only If case): Suppose there are M non-overlapping instances of the same three-term expression $d_1 = (t_i + t_j + t_k)$ among the set of expressions. Now consider two different cases. In the first case, assume the minimum exponent of L among the terms in d_1 is 0. Then d_1 satisfies the definition of a divisor. Since our divisor generation algorithm extracts all possible three-term divisors, there will be M non-overlapping divisors representing d_1.

In the second case, assume that d_1 does not satisfy the definition of a divisor (i.e., there are no terms in d_1 with a zero exponent of L). Then we have $d_1' = (t_i' + t_j' + t_k')$ which is obtained by dividing each term in d_1 by the minimum exponent of L. Now d_1' satisfies the definition of a divisor, and reasoning as above, there will be M non-overlapping divisors representing d_1'.

Iterative common subexpression elimination algorithm

Figure 5.13 shows the algorithm for three-term extraction. In the first step, **frequency statistics** of all distinct divisors is computed and stored. Frequency

statistic of each divisor is the the number of instances of that divisor. This is done by generating divisors $\{D_{new}\}$ for each expression and looking for intersections with the existing set $\{D\}$. For every intersection, the frequency statistic of the matching divisor d_1 in $\{D\}$ is updated and the matching divisor d_2 in $\{D_{new}\}$ is added to the list of intersecting instances of d_1. The unmatched divisors in $\{D_{new}\}$ are then added to $\{D\}$ as distinct divisors.

```
Optimize ({Pi})
{
    {Pi} = Set of expressions in polynomial form;
   {D} = Set of divisors = φ ;
  // Step 1. Creating divisors and their frequency statistics
    for each expression Pi in {Pi}
   {
       {Dnew} = Divisors(Pi);
       Update frequency statistics of divisors in {D};
       {D} = {D} ∪ { Dnew};
  }

  //Step 2. Iterative selection and elimination of best divisor
  while (1)
  {
      Find d = divisor in {D} with most number
                of non-overlapping intersections;
       if (d == NULL) break;
      Rewrite affected expressions in {Pi} using d;

      Remove divisors in {D} that have become invalid;

     Update frequency statistics of affected divisors;
     {Dnew} = Set of new divisors from new terms added
              by division;
       {D} = {D} ∪ {Dnew};
     }
}
```

Figure 5.13 Algorithm for three-term extraction

In the second step of the algorithm, the best three-term divisor is selected and eliminated at each iteration. The best divisor is the one that has the most number of non-overlapping divisor intersections. Those expressions that contain this best divisor are then rewritten. Since each CSA produces two outputs, a sum and a carry, each divisor also produces two numbers representing the two outputs. Figure 5.14 shows the rewriting of the expressions after the selection of the subexpression $D_1 = X_1 + X_2 + X_2<<1$, where D_1 is the extracted divisor, and D_1^S and D_1^C represent the sum and the carry outputs of D_1, respectively.

$$D_1 = X_1 + X_2 + X_2<<1$$

$$Y_1 = (D_1^S + D_1^C) + X_1<<2 + X_2<<2$$

$$Y_2 = (D_1^S + D_1^C)<<2$$

Figure 5.14 Expression rewriting using 3-term common subexpressions

After selecting the best divisor, those divisors that overlap with it, no longer exist and have to be removed from the dynamic list {D}. As a result the frequency statistic of some divisors in {D} will be affected, and the new statistics for these divisors is computed and recorded. New divisors are generated for the new terms formed during division of the expressions. The frequency statistics of the new divisors are computed separately and added to the dynamic set of divisors {D}.

The algorithm terminates when there are no more useful divisors. For our example expressions, after rewriting the expressions as shown in Figure 5.14, the set of dynamic divisors {D} is updated. No more useful divisors are found after this, and the algorithm terminates. The optimized circuit is shown in Figure 5.11.

Algorithm complexity and quality: The algorithm spends most of its time in the first step where the frequency statistics of all distinct divisors are computed and stored. For an expression with N terms, the number of 3-term divisors is $\Theta(N^3)$. Therefore, the complexity of the first step, for the case of M expressions is $\Theta(MN^3)$. In the second step of the algorithm, each time a divisor is selected, the number of terms in the affected divisor is reduced by one. In the worst case, all expressions are reduced from N terms to two terms at the end of the algorithm. The number of steps to reduce from N terms to two terms is (N-2). Since there are M expressions, the complexity of this step is $\Theta(MN)$.

5.2.2 Experimental Validation

The goal of the experiments was to validate the three-term extraction technique through real synthesis results. A set of benchmarks in Sum of Product (SOP) form were considered. These included H.264 video encoding, Discrete Cosine Transform (DCT), Inverse Discrete Cosine Transform (IDCT), Discrete Fourier Transform (DFT), Discrete Hartley Transform (DHT) and Discrete Sine Transform (DST) which are all matrix multiplications. Furthermore a couple of FIR filters (20-tap and 41-tap) were considered where the SOP forms were obtained from the constant multiplications in the multiplier block. The Synopsys Design Compiler™ was used for performing the synthesis. The compile_ultra option in Design Compiler

was used in the synthesis. The compile_ultra command performs very sophisticated datapath optimizations and arguably the best commercial tool available.

For the first six examples which are matrix multiplications, the constants were set to be 16 bits wide and for the two FIR filter examples, the constants were set to be 24 bits wide. For all examples, the input signals were set to be 12 bits wide. The bitwidths of the outputs were derived from the input bitwidths and the constant bitwidths as well as the bitwidths of the intermediate results without using any bit-truncation.

For all the examples, the original expressions (without any 3-term extraction) were synthesized as well as the optimized expressions after 3-term extraction. For the optimized designs, both the binary as well as the Canonical Signed Digit (CSD) representations of the constants were considered. The mmi25.db technology library and the Synopsys Designware^{TM} library were used for the synthesis. The mmi25.db library is a 0.25 micron technology library, and the Designware library provides the datapath components. Very tight timing constraints were set for all the examples. For the H264 example, a timing constraint of 7 ns was used. For the examples DCT, IDCT, DFT DHT and DST the timing constraint was set at 8 ns, and for the two FIR filter examples, was 5 ns. All the designs were able to meet these constraints and the critical paths for all the designs of the same example were very close to each other (within 1%). Table 5-4 compares the areas produced by the different methods. We call our extraction method as Continuous Arithmetic Extraction (CAX). The extraction using Binary representation of the constants is called CAX-Bin and the extraction using CSD representation of the constants is called CAX-CSD. We

compare the areas produced by these methods with the area produced by using only compile_ultra. Figure 5.15 plots the areas produced by the different methods.

Table 5-4 Comparing areas produced by different methods on synthesis

Example	CAX-Bin	CAX-CSD	compile_ultra	Improvement of CAX-CSD over Compile_ultra (%)
H264	199370	172810	202020	14.46
DCT8	618940	526180	659190	20.18
IDCT8	558490	488780	579900	15.71
DFT8	532740	500110	608140	17.76
DST8	637830	550470	682070	19.29
DHT8	813230	636100	767230	17.09
FIR20	132540	110550	124700	11.35
FIR41	278350	238720	282490	15.5
Average	**471436**	**402965**	**488217**	**16.42**

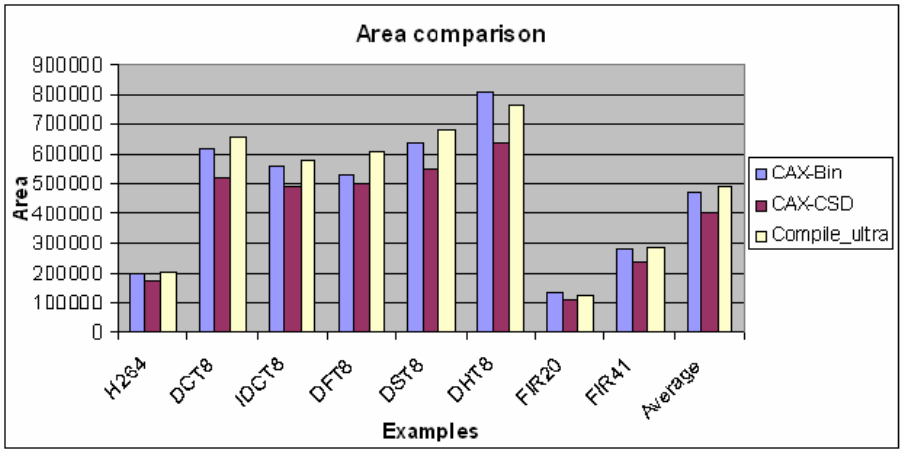


Figure 5.15 Area comparison for Sum of Products implementations using CSAs

The results show that the CAX-CSD method gives the least area in all cases, and on an average produced about **16.42 %** lesser area than the original expressions synthesized using compile_ultra command. These results are very significant because they are achieved over the best datapath synthesis tool used in the industry.

It should be noted that these results are after the gate level synthesis in Design Compiler.

The area reduction that is calculated is only due to the reduction in the number of logic gates for implementing the functions, and does not factor in the area reduction due to the reduction in the number of wires. This can be calculated only after detailed place and route stages. While there could be further area reduction due to the reduction in the number of wires, there will be congestion in the layout due the presence of a number of nets with high fanout arising from the extracted common subexpressions. This congestion may offset some of the area reduction due to reduced wiring. A study of the impact of our optimizations on the final placed and routed designs is a topic of future work.

5.3 Performing delay aware optimization

This section presents some work on the delay aware commons subexpression elimination. First it is shown that extending the redundancy elimination techniques for the linear arithmetic expressions (with constant multiplications) is straightforward. This is demonstrated for both type of circuits: the one with two-input adders (Section 3.2.2) and the one with Carry Save adders (Section 5.2). The task of performing such an optimization in the presence of a limited number of resources is very hard since the scheduling of the different operations have to be taken into account. An approach similar to the dynamic common subexpression

elimination algorithm in [58] is used to perform resource constrained redundancy elimination. Results for this approach for a set of benchmarks are presented.

5.3.1 Delay aware two-term common subexpression elimination

The original algorithm for performing redundancy elimination in arithmetic expressions is modified (Section 3.2) to take into account the critical path of the computations. Common subexpression elimination and delay optimization is now done in one single pass. To the best of our knowledge, this is the only technique that performs both optimizations at the same time. The main attractive features of our technique are that it can be used for expressions consisting of any number of variables, with different arrival times for the variables. In conventional high level synthesis, the redundancy elimination transformations are applied independently of the delay minimization transformations, which are typically done by algorithms such as Tree Height Reduction (THR) [59]. The THR algorithm is an expensive algorithm involving extensive backtracking. Instead of doing these transformations separately, our algorithm performs redundancy elimination in such a way that the maximum limit on the delay is not exceeded.

As a motivational example, consider the evaluation of the expression F = a + b + c + d + a<<2 + b<<2 + c<<2 + a<<3 + b <<3 + e. All the signals are available at time t = 0, except for a, which is available at time t = 1. Figure 5.16a shows the evaluation of this expression. This expression consists of nine additions and using the fastest tree structure for this expression, we get a critical path of four additions.

From this expression, there are two common subexpressions $d_1 = a + b$, and $d_2 = a + b + c = d_1 + c$. By eliminating these common subexpressions, the implementation shown in Figure 5.16b is obtained, which now has only six additions, but the critical path has been increased to four addition steps. By doing a delay aware common subexpression method, only the the common subexpression $d_1 = (a + b)$ is extracted, and the result has seven additions, but the critical path does not increase, as shown in Figure 5.16c.

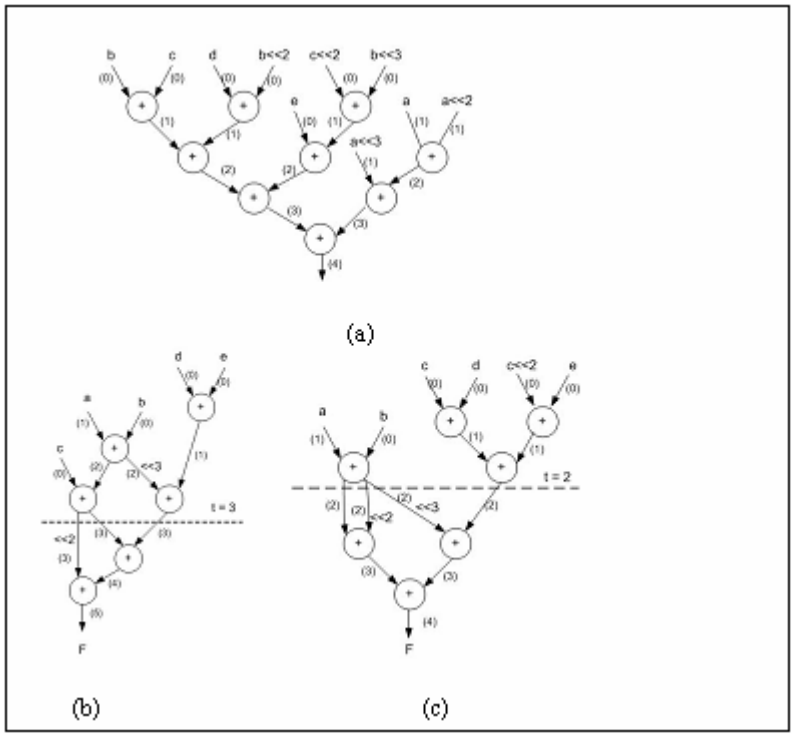


Figure 5.16 Optimization example (a) Original expression tree (b) Eliminating common subexpressions (c) Delay aware common subexpression elimination

The availability times of the various signals in the dataflow graph are shown using parenthesis on each edges of the graph. This example is used to show the

working of the delay aware algorithm. Thus it can be seen that delay can increase if common subexpressions are not extracted carefully.

Problem formulation

Some key concepts are now introduced and the problem is formally stated.

A. Minimum latency of a linear system

Only the fastest tree implementation of the set of additions and subtractions of a linear system is considered. It is assumed that there are enough adders to achieve the fastest tree structure and achieve the minimum possible latency. When all the variables of the system are available at the same time (t = 0), then the latency can be determined by the number of terms N_{max} of the longest expression in the system. The latency is then given by

$$Min - Latency = \lceil \log_2 N_{\max} \rceil$$

Equation 5-6

When the signals have different arrival times, even then the latency can be calculated by Equation 5-6, but the number of terms has to be adjusted to take into account the different availability times of the terms. The arrival times are assumed to be integer numbers and the delay of an adder/subtractor is assumed to be one unit. For each term with arrival time $t_i > 0$, the term can be viewed as being produced by the summation of 2^{t_i} dummy terms, which are available at time t = 0 (the delay of the summation being t_i). Therefore the number of terms for the expression is increased by $2^{t_i} - 1$. For example, consider the expression shown in Figure 5.16a. This

expression has 10 terms, out of which three of them have arrival times equal to 1. Therefore the number of terms is calculated as $10 + 3*(2^1 - 1) = 13$. The minimum delay of the expressions calculated from Equation 5-6 is 4 units.

B. Recursive and Non-Recursive Common Subexpressions:

A recursive common subexpression is a subexpression that contains at least one other common subexpression extracted before. For example, consider the constant multiplication (1010-101010-1)*X as shown in Figure 5.17a. The common subexpression $d_1 = X + X<<2$ (Figure 5.17b) is non-recursive, and it reduces the number of additions by one. Now, the common subexpression $d_2 = (d_1<<2 - X)$ is recursive since it contains the variable d_1 which corresponds to a previously extracted common subexpression. Extracting this leads to the elimination of one more addition.

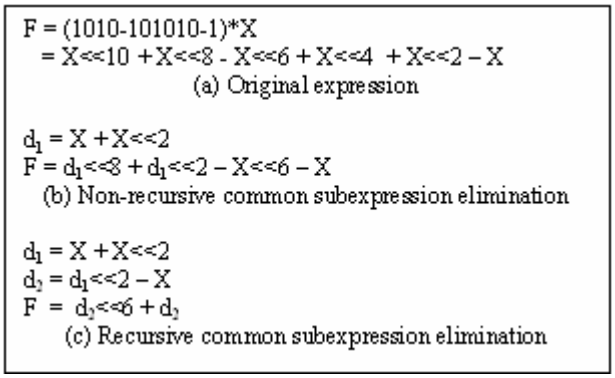


Figure 5.17 Recursive and Non-Recursive common subexpression elimination

C. Problem statement

The problem can be formulated thus. Given a multiplierless realization of a linear system, minimize the number of additions/subtractions as much as possible such that the latency does not exceed the minimum specified latency. For purposes of simplicity the explanations in this section target the minimum possible latency as described by Equation 5-6. As per the problem statement, both recursive and non-recursive common subexpressions are explored to give the maximum reduction in the number of additions, without exceeding the delay constraint.

Delay aware common subexpression elimination algorithm

The algorithm takes into account the effect of delay on selecting a particular divisor as a common subexpression. Only those instances of a divisor that do not increase the delay of the expression beyond the maximum specified delay limit are considered. First the delay calculation of an expression on selection of divisor is explained. The main algorithm is then explained. For simplicity it is assumed that the delay for a single addition/subtraction is one time unit, and that the arrival times of all the variables have been normalized to integer numbers.

A. Calculating delay

Each divisor is associated with a level which represents the time (in integer units), when the value of the divisor is available. Furthermore, each divisor is associated with the number of original terms covered by it. To handle variables with different arrival times, it is assumed that each term available at time t_i is covered by

2^i original dummy terms. This has no impact on the quality of the solution, and helps to predict the delay using a simple formula, as is explained later.

Consider the expression F as shown in Figure 5.18. The arrival times of the variables are shown as superscripts. The calculation of the level of the divisor and the original terms covered by the divisor is illustrated in the figure. The procedure for the calculation of the delay of an expression, after the selection of a divisor that is contained in the expression is illustrated in Figure 5.19. The terms $\{T_E\}$ of the expression are partitioned into the terms $\{T_1\}$ covered by the divisor and the remaining terms $\{T_2\}$.

$F = a^{(0)} + b^{(1)} + c^{(2)} + d^{(0)}$

$d_1 = (b^{(1)} + c^{(2)})$
Level $(d_1) = 3$
Original terms covered$(d_1) = 2^1 + 2^2 = 6$

$d_2 = d_1 + a$ Level$(d_2) = 4$
Original terms covered$(d_2) = 6 + 1 = 7$

Figure 5.18 Divisor information

p = # of instances of Divisor D in expression
t = Delay(adder-steps) in computing divisor D

$\{T_1\}$ = current terms covered by 'p' instances of D
$\{T_E\}$ = current terms in the expression
$\{T_2\} = \{T_E\} - \{T_1\}$ = Remaining terms

K = # of Values in $\{T_2\}$ still available for computation after time t

Total values available = p + K
Delay of expression = $(t + \lceil \log_2(p + K) \rceil)$

Figure 5.19 Procedure for calculating delay

The delay is calculated from the number of values that are available for computation after the time (t) taken to compute the divisor under investigation. Among $\{T_1\}$ terms, there will be 'p' values available corresponding to the 'p' instances of the divisor. The number of values from $\{T_2\}$ that are available after time t has to be calculated. In general, the terms in $\{T_2\}$ have to be scheduled to get this information. But scheduling for every candidate divisor using a simple algorithm like As Soon As Possible (ASAP), which is quadratic in the number of terms is expensive. For a lot of cases, this can be calculated using a simple formula.

Let T_{2o} be the number of original terms corresponding to the terms in $\{T_2\}$. If none of the terms in $\{T_2\}$ have been covered by any divisor, or they are covered by divisors covering power of two original terms implemented in the fastest tree structure (covering 2^j original terms with delay j), then K can be quickly calculated using the formula

$$K = \left\lceil \frac{T_{2o}}{2^t} \right\rceil$$

Equation 5-7

The cases in which the algorithm can be speeded up are:

1. The divisor covers power of 2 original terms with the fastest possible tree structure (2^j original terms with delay of j). In this case, the delay need not be estimated, and all non-overlapping instances can be extracted without increasing the delay.

2. The remaining terms (terms not covered by the divisor) have not been covered by any divisor.

3. Of the remaining terms (terms not covered by the divisor), some or all of the terms may be covered by divisors. If these divisors cover power of 2 original terms with the fastest possible tree structure, then the formula can be used. Using these pruning conditions helps to significantly speed up the algorithm.

If the terms in $\{T_2\}$ do not satisfy this criterion, then K has to be calculated using ASAP (As Soon As Possible) Scheduling [8].

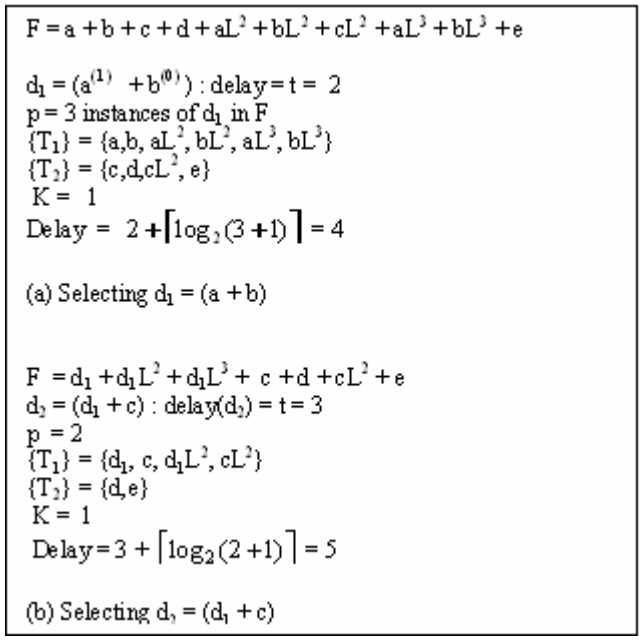


Figure 5.20 Delay calculation for example expression

The delay calculation for the example expression is illustrated in Figure 5.20. In part (a), the delay calculation for divisor $d_1 = (a + b)$ is illustrated. The delay of this divisor is two units. The expression tree corresponding to the selection of this divisor

is shown in Figure 5.16c. From this figure, it can be seen that four values are available for computation after one adder step. Three of them (p) correspond to the three uses of the divisor d_1 and K = 1 of them is from $\{T_2\}$ (the terms other than those covered by d_1). K can also estimated by the formula in Equation 5-7. The delay is calculated to be 4.

Figure 5.20b shows the delay calculation when $d_2 = (d_1 + c)$ is selected. The delay of the divisor d_2 is three units. The expression tree showing the selection of d_2 can be seen in Figure 5.16b. From the figure, the number of values available for computation after t = 3 adder steps is three. Two of them (p) correspond to the two uses of divisor d_2 and the other one (K) corresponds to the value e. K can also be calculated using Equation 5-7 . The delay is calculated to be 5, and the expression tree is as shown in Figure 5.16b.

B. Optimization algorithm

The optimization algorithm is a modification of the two-term iterative common subexpression elimination algorithm (Figure 3.25) discussed in Section 3.2. The only difference between the delay aware optimization algorithm and the original algorithm is that in the delay aware version, the ***true value*** of each divisor instead of just calculating the number of additions that are saved by it. This true value is equal to the number of instances of the divisor that can be selected without increasing the delay beyond the maximum specified delay (MaxDelay). The delay for each instance of each divisor is calculated using the method described earlier. Since the procedure for calculating true value is the only difference between the delay aware

optimization algorithm and the original algorithm in Figure 3.25, only that procedure is illustrated in Figure 5.21.

```
Find_true_value( d, MaxDelay)
{
   d = distinct divisor in set of divisors {D}
  { d_instances } = Set of instances of divisor d
  {P_i} = Set of expressions containing
          any of the instances in { d_instances }
 {Allowed_instances} = φ;

  for each expression P_i in {P_i}
  {
    {Valid_d}_i = Set of non-overlapping instances
                  that can be extracted in P_i without
                  making its delay > MaxDelay;
    {Allowed_instances}
          = {Allowed_instances} ∪ {Valid_d}_i ;
  }

  True_value = |{Allowed_instances}| ;
  return True_value;
}
```

Figure 5.21 Algorithm to find the true value of a divisor

This procedure is called for each candidate divisor d, with the given limit on the maximum delay MaxDelay. In this procedure, first the set of all expressions which contain instances of that divisor are collected in the set $\{P_i\}$. Now for each expression P_i in that set, the number of non-overlapping instances of divisor d that do not increase the critical path of P_i beyond MaxDelay is calculated. The true value of the divisor is then equal to the sum of such instances over all the expressions in $\{P_i\}$.

C .Experimental results

Results for the delay aware algorithm on a set of FIR filters and DSP transforms are presented. The delay produced by the delay ignorant algorithm (Section 3.2) with the delay aware common subexpression elimination algorithm is compared. The number of additions and subtractions produced by the two algorithms and the Non-Recursive Common Subexpression Elimination (NRCSE) algorithm [60] is compared. The delay and the number of operations produced by the delay aware extraction method is compared with that produced by the delay ignorant method (Section 3.2.2) and the Non-Recursive Common Subexpression Elimination (NRCSE) method [60]. The results are shown in Table 5.3, where A denotes the number of additions and CP denotes the delay. The results for the NRCSE algorithm, delay ignorant algorithm and the delay aware algorithm are arranged in columns (I), (II) and (III) respectively. All the coefficients in these filters are converted to integers and rounded, and are represented using 24 digits of precision. The four filters shown in the results have 401, 401, 20 and 41 taps respectively.

Table 5-5 Experiments on FIR filters (I) = NRCSE, (II) = Delay ignorant (III) = Delay aware methods

Filter #	Original		(1)		(II)		(III)	
	A	CP	A	CP	A	CP	A	CP
1	2002	4	1184	4	316	5	317	4
2	1540	3	921	3	239	4	254	3
3	106	4	59	4	23	4	23	4
4	238	4	146	4	54	4	54	4

The results show that the delay ignorant method produced the least number of additions compared to the other methods, but the critical path is exceeded in two cases. In comparison, the delay aware optimization produces the fastest

implementation in all cases, at the expense of a few more additions compared to [35]. The NRCSE method also gives the fastest implementation in all cases. Since our delay aware method also explores recursive common subexpressions, a further reduction of the **67.4%** in the number of operations is achieved over the NRCSE method. The average run time for our delay aware optimization algorithm was only **0.3s.**

Transform	Original		Delay ignorant		Delay aware	
	A	CP	A	CP	A	CP
DCT	2033	8	946	8	946	8
IDCT	1784	7	861	**8**	923	7
DFT	1761	7	853	**8**	925	7
DST	2066	8	949	8	949	8
DHT	2174	8	1013	8	1013	8
Average	**1963.6**	**7.6**	**924.4**	8	**951.2**	**7.6**

Experiments were also performed with a few common DSP transforms the Discrete Cosine Transform (DCT), Inverse Discrete Cosine Transform (IDCT), Discrete Fourier Transform (DFT), Discrete Sine Transform (DST) and Discrete Hartley Transform (DHT) [61]. The 8-point transforms were considered for these examples, where the constant matrices are 8x8 (8 rows and 8 columns). Each coefficient has been quantized to 24 digits of precision. Each transform takes eight input samples and produces eight outputs. The arrival times of the eight input variables $\{X_0, X_1 \ldots\ldots X_7\}$ were set as {0,0,1,1,2,2,3,3} respectively. The number of additions/subtractions and the delay produced by the delay aware and delay ignorant methods were compared.

Table 5-6 Experiments on DSP transforms (multi-variable)

Table 5-6 shows the results for the multiple variable case. Here A represents the number of additions and CP represents the critical path in the number of adder steps. We did not compare the results produced by the NRCSE algorithm [60], since that algorithm only optimizes FIR filters, where only a single variable is involved. From the results, it can be seen that the delay ignorant algorithm produces the least number of additions/subtractions for all examples, but also the latency is increased for each case. In comparison, the delay aware optimization method produces the fastest implementation, at the expense of just **3%** increase in the number of additions. The delay aware algorithm still manages to reduce the number of additions compared to the original unoptimized case by an average of **45.6%**. The average run time of the delay aware optimization algorithm was **13.9s** for these examples.

5.3.2 Delay aware three term extraction

The idea for performing delay aware two-term common subexpression elimination can be extended to perform delay aware three-term extraction. The minimum possible delay of the system is calculated before performing the optimization. During the three term extraction algorithm (Figure 5.13), each

candidate divisor is evaluated for its impact on delay, and only select those divisors that do not increase the delay are selected. At the end of the modified algorithm an implementation with reduced number of adders is obtained, but now with the minimum possible delay. In this section, the delay model is first presented. This delay model is used to calculate the delays of the expressions involved. This is followed by the working of the algorithm and the experimental results.

A. Delay model

The delay model is simple and is similar to the one used in [53] . A unit delay is assumed for both the sum and the carry outputs of a CSA. The arrival times of the various signals in the circuit are normalized to integer numbers. This model can be easily generalized to handle actual values for arrival times and delays of the CSAs. The authors in [53] present an **optimal** polynomial time algorithm for finding the fastest CSA tree for every expression. This algorithm is an iterative algorithm where in each step the terms of the expression are sorted according to non-decreasing availability times. The first three terms are then allotted to a CSA. This continues till only two terms remain. This algorithm can be used to find the minimum delay of given expressions, using our delay model. The extraction is performed such that at each step, the delay of the expressions does not exceed this minimum delay.

B. Example

Consider the evaluation of the following arithmetic expressions

$$F_1 = a + b + c + d + e$$

$$F_2 = a + b + c + d + f$$

All signals are available at time t = 0, except for a, which is available at time t = 2. Using the optimal CSA allocation algorithm in [53], the minimum delay for both F_1 and F_2 is calculated as **3 + D(Add)**, where D(Add) is the delay of the final two input adder.

Figure 5.22 shows the evaluation of the two expressions after performing delay ignorant extraction. The arrival times of signals are shown along the edges of the circuit. In this example, the subexpression $D_1 = (a + b + c)$ is first extracted and then the subexpression $D_2 = D_1^S + D_1^C + d$ is extracted. This leads to an implementation with only four CSAs, but the delay of the circuit is now **5 + D(Add**), which is two units more than the optimal delay.

Figure 5.23 shows the result of delay aware extraction. Here the subexpression (a + b + c) is not extracted because by doing so the delay increases. The divisor $D_1 = (b+c+d)$ does not increase the delay so it is extracted. After rewriting the expressions, the common subexpression $(D_1^S + D_1^C + a)$ is considered, but is not selected because it increases the delay. The delay aware extraction has one more CSA than the delay ignorant one, but it has the minimum delay.

$D_1 = a + b + c$
$Delay(D_1) = 3$
$F_1 = D_1^{\#} + D_1^{C} + d + e$
$F_2 = D_1^{\#} + D_1^{C} + d + f$

$Delay(F_1, F_2) = 5 + D(Add)$

$D_2 = D_1^{\#} + D_1^{C} + d$
$Delay(D_2) = 4$
$F_1 = D_2^{\#} + D_2^{C} + e$
$F_2 = D_2^{\#} + D_2^{C} + f$

$Delay(F_1, F_2) = 5 + D(Add)$

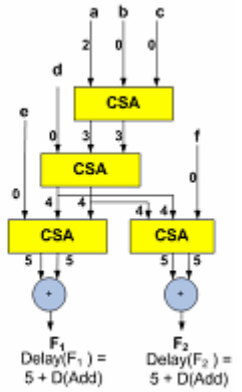


Figure 5.22 Delay ignorant three-term extraction

$D_1 = b + c + d$
$Delay(D_1) = 1$
$F_1 = D_1^{\#} + D_1^{C} + e + a$
$F_2 == D_1^{\#} + D_1^{C} + f + a$

$Delay(F_1, F_2) = 3 + D(Add)$

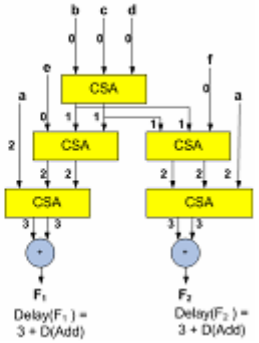


Figure 5.23 Delay aware three-term extraction

C. Algorithm

The delay aware extraction algorithm is a modification of the original algorithm that does not consider delay. In the algorithm shown in Figure 5.13, instead of finding the divisor that has the most number of non-overlapping instances, the divisor that has the most number of non-overlapping instances that do not increase the minimum delay is selected. This requires that the delay be calculated for

every candidate divisor. The complexity of calculating the delay of an expression using the algorithm in [53] is quadratic in the number of terms in the expression.

D.Results

Experiments were performed with the same set of examples that we presented in Section 5.2.2. Using the delay model described earlier, the delay and the number of CSAs produced by the delay ignorant algorithm and the delay aware algorithm are compared. The results are presented in Table 5-7 .

From the results, one can see that the delay ignorant algorithm produces the least number of CSAs, but the delay is increased in all examples. The delay aware algorithm produces the optimal delay according to our delay model and [53], but due to the selective extraction of common subexpressions, the number of CSAs is increased by an average of **15.5%** over the delay ignorant algorithm. The delay aware algorithm still reduces the number of CSAs by **31.1%** over the original unoptimized expressions, for the same delay. The average CPU time for the delay aware algorithm is 11.4s, compared to 2.05s for the original algorithm.

Table 5-7 Comparing delay ignorant (I) and delay aware (II) three-term extraction

Example	# CSAs		Delay		CPU time (s)	
	(I)	(II)	(I)	(II)	(I)	(II)
H.264	78	79	9	8	0.2	1.95
DCT8	222	232	14	13	8.5	44.9
IDCT8	34	201	14	13	3.3	20.9
6 tap FIR	11	15	5	4	0.01	0.03
20 tap FIR	34	45	6	5	0.04	0.16
41 tap FIR	79	91	6	5	0.26	0.7
Average	**103.2**	**110.5**	**9**	**8**	**2.05**	**11.4**

5.4 Performing resource aware common subexpression elimination

The techniques for delay aware redundancy elimination presented in the earlier sections assumed an unlimited number of resources. In practice though, there are a limited number of processors (functional units). With limited resources, the operations have to be scheduled, such that the task is completed in the shortest time. To get the best possible results, the redundancy elimination has to be carried out such that it gives the fastest evaluation on the given set of resources. In the traditional high level synthesis flow, the dataflow optimizations such as common subexpression elimination and copy propagation are performed before scheduling. But doing so, can create certain dependencies in the operations, which can hamper the schedule on the given set of processors. If those optimizations are performed after scheduling, then the scheduling can miss a lot of improvements that are possible due to the elimination of redundant operations.

Unfortunately, most of the scheduling algorithms, including the resource constrained, are NP complete problems in themselves, and seeking an optimal solution to combined CSE and scheduling is a very hard problem. Recently, there was some work on a dynamic CSE algorithm [58], wherein the authors present a method that dynamically eliminates common subexpressions based on new opportunities that are created during the scheduling of control-intensive designs. Conceptually, the technique examines the list of remaining, ready to be scheduled operations, and determines which of these has a common subexpression with the

currently scheduled operation. This common subexpression can now be eliminated due to the code motion of the currently scheduled operation. A dynamic CSE algorithm was implemented by us called dynamic Continuous Arithmetic Extraction (CAX), to see whether any improvement in the schedule lengths can be achieved by combining the arithmetic extraction with scheduling. For simplicity, only the linear arithmetic expressions were considered, where the constant multiplications are decomposed into additions and shift operations (Section 3.2). Therefore the only operations to be scheduled are additions (and subtractions).

Figure 5.24 illustrates the working of the Dynamic CAX algorithm that was implemented by us. This algorithm takes as inputs the operations to be scheduled and the number of resources. First, the set of all two-term divisors are generated as explained in Section 3.2.2. The outer loop runs for each clock cycle of the schedule length. In each iteration of the outer loop, all operations that can be scheduled in that clock cycle are scheduled. For each clock cycle, operations are selected for each functional unit (adder in this case).

```
Dynamic-CAX ()
{
    N = max_resources;
    Divisors = GenerateDivisors();

    current_cycle = 0;
    While(operations to schedule)
    {
        current_cycle++;

        for(resource = 1; resource <= N; resource++)
        {
              if (no schedulable operation in this cycle) break;
              E = Expression with greatest latency on N processors;
              E_Divisor = Pick_Best_Divisor(E);
              Schedule operation corresponding to E_Divisor;
             Rewrite_Expressions using E_Divisor;
                 Update frequency statistics of Fx-Divisors;
         }
   }

  Schedule_length = current_cycle;
}

Divisor* Pick_Best_Divisor(Expression *E)
{
     Among all Divisors {D} that belong to E,
     return the best divisor according to some cost function
}
```

Figure 5.24 Dynamic Continuous Arithmetic Extraction

For each function unit, the operation is selected from the expression that has an available operation, and has the greatest critical path. Among all the operations in that expression, the one that has the most number of instances, and thereby saves the most number of additions is selected. After the operation is selected, all the expressions that contain that operation as a common subexpression are rewritten, and the frequency statistics of the set of divisors are recomputed and stored.

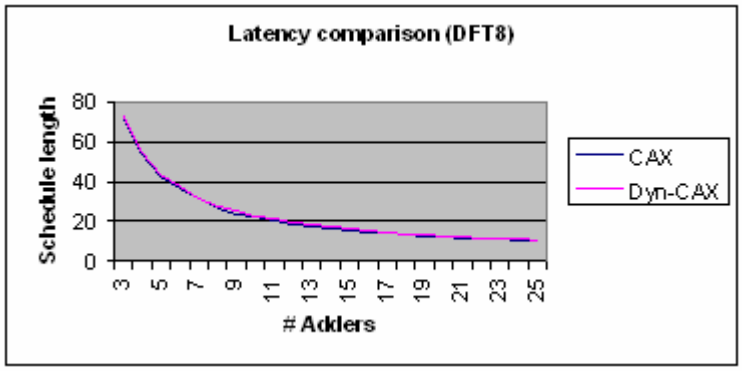


Figure 5.25 Comparing CAX and Dynamic CAX for DCT8 example

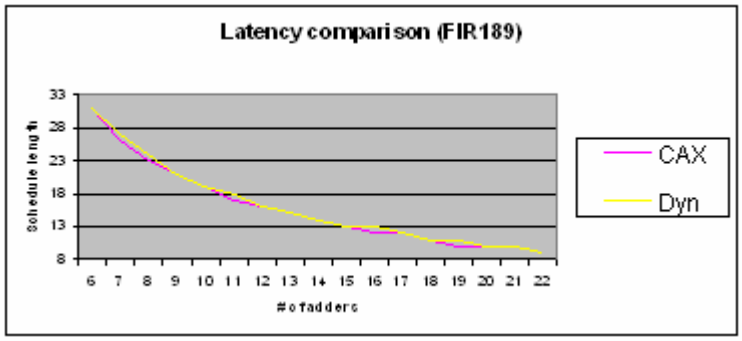


Figure 5.26 Comparing CAX and Dynamic CAX for FIR189 example

Using this algorithm experiments were run for a couple of benchmarks where we the number of resources were varied, and the schedule length for the original CAX algorithms (Section 3.2) and the dynamic CAX algorithm were compared. The schedule length for the CAX algorithm was calculated after the end of the extraction, using List Scheduling. The schedule length for the Dynamic CAX algorithm is calculated automatically. The plots for the latency with varying resources are shown for the DCT8 example and the 189-tap FIR filter in Figure 5.25 and Figure 5.26 respectively.

From the results, we can see that the CAX algorithm still gives the best schedule length in all cases. The Dynamic CAX algorithm is able to match the schedule length in most cases, but it never gets better than the CAX algorithm.

Conclusions

The motivation for performing schedule aware common subexpression elimination is very strong. But it is difficult to come up with a good heuristic for doing it.

6 Verifying Precision and Range in Arithmetic Expressions

Many numerical computations, especially those modeling physical phenomena are inherently inaccurate. They will not deliver the “true” value, but an approximate value that is close to the true value. Approximate computation has been studied for several centuries by mathematicians and scientists, but it has been receiving a lot of attention in the Electronic Design community. Most embedded system applications employ finite wordlengths (floating point or fixed point) representations of signals and constant coefficients. The limited precision of these representations lead to quantization (roundoff) noise. The limited ranges of the representations lead to overflow and underflow during computation. Though these problems affect both Floating point as well as Fixed point computations, they are more pronounced in Fixed point. Most embedded systems today cannot afford the power of Floating point computations because of constraints on performance, cost and power consumption.

After performing the various optimizations on the arithmetic expressions, such as factorization and restructuring, the restructured polynomial has to be verified for its accuracy and range on the target architecture. Most applications have constraints on the maximum allowed error (truncation or roundoff) at the outputs. This error can be specified either as an absolute error, or as a signal to noise ratio. As stated earlier, the topic of error analysis is very vast, and there are various aspects of

the problem. Investigating all these aspects and coming up with a comprehensive analysis methodology is out of the scope of this thesis. Instead, an important procedure in static analysis methods is discussed. This procedure can be used for performing an analysis of the errors in arithmetic expressions. This procedure basically propagates the ranges of the numbers and keeps track of the uncertainties in the inputs and intermediate values in the expression. This propagation technique is based on affine arithmetic, which was developed in detail in [62] after being applied to Computer Graphics applications [63, 64]. More recently it was applied to the DSP computations in fixed point as well as floating point [65-67]. This chapter points out some of the key features of the modeling of ranges and errors of a variable using affine arithmetic and the propagation of the values in an expression.

6.1 Data required for performing error analysis

The following are the requirements for doing static error analysis for an arithmetic expression.

1. Input quantization error

This is the error due to quantization of the inputs or the constant coefficients, and represents the uncertainty in the values. This may be either an absolute value or an upper bound.

2. Range of the input values

The amount of error depends on the magnitudes of the input variables and constants. The minimum and maximum values can be used to find the maximum error in the

computations. These bounds can either be found from simulation or is specified by the application.

3. Modeling error produced by different operators

Each operator (+, -, X, <<) produces an error that are due to the associated errors of the input operands as well as the finite wordlength of the result register. A model for each operator is required for calculating the error at the output.

4. Propagating ranges and errors

A methodology is required to propagate the ranges of the input operands and the errors through the intermediate computations. This is required because the error produced by an operation is a function of the data ranges of its operands and the associated errors.

6.2 Interval arithmetic

Interval arithmetic (IA) [68] also known as interval analysis, was invented in the 1960s by Moore to solve range problems. The uncertainty in a variable x is represented by the interval $\bar{x} = [\bar{x}.lo, \bar{x}.hi]$, meaning that the true value of x is known to satisfy $\bar{x}.lo \leq x \leq \bar{x}.hi$. For each operation $f : R^m \rightarrow R$, there is a corresponding range extension $\bar{f} : \bar{R}^m \rightarrow \bar{R}$. Taking addition as an example, the corresponding IA operation is obtained as

$$\bar{z} = \bar{x} + \bar{y} = \bar{x}.lo + \bar{y}.lo,\ \bar{x}.hi + \bar{y}.hi$$

Equation 6-1

The main problem of IA is that it does not consider the correlations between the variable leading to overestimation. To illustrate this problem, suppose that in the above equation, $\bar{x}$ = [-1,1] and $\bar{y}$ = [-1,1], and that x and y have the relation y = -x. Using (4.1) above, we get $\bar{z}$ = [-2,2], but actually z = x + y = 0. The effect of this kind of overestimation may accumulate along the computation chain, and may result in an exponential range explosion.

6.3 Affine arithmetic

Affine arithmetic (AA), or affine analysis, is a recent refinement in range arithmetic to alleviate the problem of over-estimation in IA [69]. It has been used in areas such as computer graphics and analog circuit sizing. In contrast to IA, AA preserves the correlations between the variables. In affine arithmetic, the uncertainty of a variable x is represented in an affine form $\hat{x}$ given by

$$\hat{x} = x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \ldots\ldots + x_n\varepsilon_n \quad , \quad \text{with } -1 \le \varepsilon_i \le 1$$

Equation 6-2

Each symbol ε_i stands for an independent component of the total uncertainty of the variable x, and the corresponding coefficient x_i gives the magnitude of that component. For affine operations $\hat{x} \pm \hat{y}, a \pm \hat{x}, a\hat{x}$, the resulting affine forms are easily obtained from equation (II). For multiplication of two variables, the resulting uncertainty of the product is no longer in an affine form, and an approximating affine form has to be constructed.

The key feature of AA is that one noise symbol may contribute to the uncertainties of two or more variables, indicating correlations among them. When these variables are combined, error terms may cancel out. Returning to the example demonstrating the drawback of the IA method, suppose that x and y have affine forms $\hat{x} = 0 + 1\varepsilon$ and $\bar{y} = 0 - 1\varepsilon$. In this case the affine form of the sum $\bar{z} = \hat{x} + \bar{y} = 0$ perfectly coincides with the actual range of the variable z.

6.3.1 AA-based fixed point error model

A real number x is represented in fixed point (x_f) representation (i,f) where i is the bitwidth of the integer part and f is the bitwidth of the fractional part as

$$x_f = x + 2^{-(f+1)} \bullet \varepsilon \qquad with\ \varepsilon \in [-1,1]$$

Equation 6-3

The value x_f is an affine interval, whose uncertainty, caused by the random variable ε, is caused by rounding. For a constant Equation 5.3 is reduced to

$$c_f = c + e,$$

$$E_c = e,\ |e| \leq 2^{-(f_c + 1)}$$

Equation 6-4

The quantization error E_c is known given the fractional bitwidth. For a variable that lies in the range $[v_0 - v_1,\ v_0 + v_1]$, its fixed point representation has one more uncertainty term $v_1\varepsilon$, shown as

$$v_f = v_0 + v_1\varepsilon + 2^{-(f_v + 1)}$$

$$E_v = 2^{-(f_v + 1)}$$

Equation 6-5

The two independent random variables ε and ε_e represent the uncertainties in the input range and the quantization error respectively. It is these random variables that capture the source of the sources of the uncertainties and keep track of the correlations among a large number of fixed point variables.

The affine form of a real number in fixed point can be generalized from Equation 6-3 as

$$x_f = x_0 + \sum_{i=1}^{i=n} x_i \varepsilon_i + E_x \ , \quad E_x = x_{e0} + \sum_{i=1}^{i=m} x_{ei} \varepsilon_{ei}$$

Equation 6-6

All the ε_i's and the ε_{ei}'s are independently distributed in [-1,1], the former capturing the range uncertainty due to the insufficient knowledge of the exact value of the variables, and the latter capturing the error uncertainty caused by fixed point quantization.

6.3.2 Modeling affine operations

For affine operations, the result is the combination of two input affine forms and a quantization error term. The existence of this term depends on the fraction bit-width. The quantization error denoted by Φ, is only introduced when the result has a smaller fraction bit-width than either operands, and is given by $2^{-(f+1)}$, where f is the fractional bit-width of the result.

Addition

$$x_f \pm y_f = (x_0 \pm y_0) + \sum_{i=1}^{i=n} (x_i \pm y_i)\varepsilon_i + E_x \pm E_y + \phi$$

$$E_z = E_x \pm E_y + \phi$$

$$\text{where } \Phi = \begin{cases} 2^{-(f_z + 1)}\varepsilon_Z, & f_Z < \max(f_x, f_y) \\ 0, \text{ otherwise} \end{cases}$$

Equation 6-7

Multiplication with a constant

$$c_f x_f = (c + E_C)(x_0 + \sum_{i=1}^{i=n} x_i \varepsilon_i) + E_x . c + \phi$$

$$E_z = E_x . c + E_c (x_0 + \sum_{i=1}^{n} x_i \varepsilon_i) + \phi$$

$$\text{where } \Phi = \begin{cases} 2^{-(f_z + 1)}\varepsilon_Z, & f_Z < \max(f_x, f_y) \\ 0, \text{ otherwise} \end{cases}$$

Equation 6-8

Modeling multiplication

When two fixed-point variables x_f and y_f in affine form are multiplied, the product is no longer in affine form due to the presence of the following quadratic terms

$$Q_1 = \sum_{i=1}^{n} x_i \varepsilon_i \sum_{i=1}^{n} y_i \varepsilon_i$$

$$Q_2 = (\sum_{i=1}^{n} x_i \varepsilon_i) E_y + (\sum_{i=1}^{n} y_i \varepsilon_i) E_x$$

Equation 6-9

A solution is to approximate Q_1 and Q_2 to linear terms, by the introduction of a new bounding operator B such that

$$\widetilde{Q}_1 = B(\sum_{i=1}^{n} x_i \varepsilon_i) B(\sum_{i=1}^{n} y_i \varepsilon_i) \varepsilon_k$$

$$\widetilde{Q}_2 = B(\sum_{i=1}^{n} x_i \varepsilon_i) E_y + B(\sum_{i=1}^{n} y_i \varepsilon_i) E_x$$

where ε_k is a new random variable in [-1,1], and the bounding operator B is defined by

$$B(\sum_{i=1}^{n} x_i \varepsilon_i) = \sum_{i=1}^{n} |x_i|$$

Equation 6-10

which computes the hard upper bound of its argument. This is a conservative approximation, in the sense that the new interval always includes the original true interval.

The multiplication of x_f and y_f can be written as

$$x_f y_f \approx x_0 y_0 + \sum_{i=1}^{n} (y_0 x_i + x_0 y_i)\varepsilon_i + r\varepsilon_k + B(\sum_{i=1}^{n} x_i)E_y + B(\sum_{i=1}^{n} y_i)E_x + \phi$$

$$E_z \approx B(\sum_{i=1}^{n} x_i)E_y + B(\sum_{i=1}^{n} y_i)E_x + \phi$$

where r equals $B(\sum_{i=1}^{n} x_i \varepsilon_i) B(\sum_{i=1}^{n} y_i \varepsilon_i)$.

Equation 6-11

6.3.3 Impact of eliminating common subexpressions

The complexity of arithmetic expressions can often be reduced by eliminating common subexpressions and by factorization. With ordinary arithmetic, or with Interval arithmetic, computing redundant operations is merely a waste of time. With Affine Arithmetic however, multiple evaluations of the same sub-expression make the result of AA less accurate. The reason is that each evaluation of a shared sub-formula represents the linearization errors of the latter by a different set of noise symbols, preventing those errors from canceling out in later steps.

7 REFERENCES

[1] H. S. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky, "Low-complexity transform and quantization in H.264/AVC," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 13, pp. 598-603, 2003.

[2] B. Schneier, *Applied Cryptography Second Edition: protocols, algorithms and source code in C*: John Wiley and Sons Inc, 1996.

[3] P.Downey, B.Leong, and R.Sethi, "Computing Sequences with Addition Chains," *SIAM Journal of Computing*, vol. 10, pp. 638-646, 1981.

[4] M. A. Miranda, F. V. M. Catthoor, M. Janssen, and H. J. De Man, "High-level address optimization and synthesis techniques for data-transfer-intensive applications," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 6, pp. 677-686, 1998.

[5] C. Shi and R. W. Brodersen, "An automated floating-point to fixed-point conversion methodology," presented at Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03). 2003 IEEE International Conference on, 2003.

[6] C. F.Fang, Rob A. Rutenbar, and T. Chen, "Fast, Accurate Static Analysis for Fixed-Point Finite-Precision Effects in DSP Designs," presented at International Conference on Computer Aided Design (ICCAD), San Jose, 2003.

[7] D. Menard and O. Sentieys, "Automatic evaluation of the accuracy of fixed-point algorithms," presented at Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings, 2002.

[8] G. D. Micheli, *Synthesis and optimization of digital circuits*: McGraw-Hill, 1994.

[9] M.Potkonjak, M.B.Srivastava, and A.P.Chandrakasan, "Multiple Constant Multiplications: Efficient and Versatile Framework and Algorithms for Exploring Common Subexpression Elimination," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 1996.

[10] R.Pasko, P.Schaumont, V.Derudder, V.Vernalde, and D.Durackova, "A new algorithm for elimination of common subexpressions," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 1999.

[11] R. Pasko, P. Schaumont, V. Derudder, and D. Durackova, "Optimization method for broadband modem FIR filter design using common subexpression elimination," presented at System Synthesis, 1997. Proceedings., Tenth International Symposium on, 1997.

[12] A.Hosangadi, F.Fallah, and R.Kastner, "Common Subexpression Elimination Involving Multiple Variables for Linear DSP Synthesis," presented at IEEE International Conference on Application-Specific Architectures and Processors (to appear), Galveston, TX, 2004.

[13] A.Chatterjee, R.K.Roy, and M.A.D'abreu, "Greedy Hardware Optimization for Linear Digital Circuits using Number Splitting and Refactorization," *IEEE Transactions on VLSI*, pp. 423-431, 1993.
[14] H.T.Nguyen and A.Chatterjee, "Number-splitting with shift-and-add decomposition for power and hardware optimization in linear DSP synthesis," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 8, pp. 419-424, 2000.
[15] M.Puschel, B.Singer, J.Xiong, J.M.F.Moura, J.Johnson, D.Padua, M.Veloso, and R.W.Johnson, "SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms," *Journal of High Performance Computing and Applications*, 2004.
[16] R. Kastner, S. Ogrenci-Memik, E. Bozorgzadeh, and M. Sarrafzadeh, "Instruction generation for hybrid reconfigurable systems," presented at Computer Aided Design, 2001. ICCAD 2001. IEEE/ACM International Conference on, 2001.
[17] A. Peymandoust, L. Pozzi, P. Ienne, and G. De Micheli, "Automatic instruction set extension and utilization for embedded processors," presented at Application-Specific Systems, Architectures, and Processors, 2003. Proceedings. IEEE International Conference on, 2003.
[18] "Tensilica Inc. http://www.tensilica.com."
[19] S.S.Muchnick, *Advanced Compiler Design and Implementation*: Morgan Kaufmann Publishers, 1997.
[20] A.S.Vincentelli, A.Wang, R.K.Brayton, and R.Rudell, "MIS: Multiple Level Logic Optimization System," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 1987.
[21] R.K.Brayton, R.Rudell, A.S.Vincentelli, and A.Wang, "Multi-level Logic Optimization and the Rectangular Covering Problem.," presented at International Conference on Compute Aided Design, 1987.
[22] R.K.Brayton and C.T.McMullen, "The Decomposition and Factorization of Boolean Expressions," presented at International Symposium on Circuits and Systems, 1982.
[23] P.F.Flores, J.C.Monteiro, and E.C.Costa, "An Exact Algorithm for the Maximal Sharing of Partial Terms in Multiple Constant Multiplications," presented at International Conference on Computer Aided Design (ICCAD), San Jose, CA, 2005.
[24] R.Rudell, "Logic Synthesis for VLSI Design," PhD Thesis, University of California, Berkeley, 1989.
[25] P.M.Embree, *C Algorithms for Real-Time DSP*: Prentice Hall, 1995.
[26] R.H.Bartels, J.C.Beatty, and B.A.Barsky, *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*: Morgan Kaufmann Publishers, Inc., 1987.
[27] A. Sinha and A. P. Chandrakasan, "JouleTrack-a Web based tool for software energy profiling," presented at Design Automation Conference, 2001. Proceedings, 2001.

[28] G.Nurnberger, J.W.Schmidt, and G.Walz, *Multivariate approximation and splines*: Springer Verlag, 1997.
[29] T. Wiegand, G. J. Sullivan, G. Bjntegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 13, pp. 560-576, 2003.
[30] J. Vasudevamurthy and J. Rajski, "A method for concurrent decomposition and factorization of Boolean expressions," presented at International Conference on Computer Aided Design (ICCAD), 1990.
[31] A.Hosangadi, F.Fallah, and R.Kastner, "Factoring and Eliminating Polynomial Expressions in Polynomial Expressions," presented at International Conference on Computer Aided Design (ICCAD), San Jose, CA, 2004.
[32] A.Hosangadi, F.Fallah, and R.Kastner, "Energy Efficient Hardware Synthesis of Polynomial Expressions," presented at International Conference on VLSI Design, Kolkata, India, 2005.
[33] M.A.Breuer, "Generation of Optimal Code for Expressions via Factorization," *Communications of the ACM*, vol. 12, pp. 333-340, 1969.
[34] A.Hosangadi, F.Fallah, and R.Kastner, "Common Subexpression Involving Multiple Variables for Linear DSP Synthesis," presented at IEEE International conference on Application Specific Architectures and Processors (ASAP), Galveston, TX, 2004.
[35] A.Hosangadi, F.Fallah, and R.Kastner, "Reducing Hardware Compleity of Linear DSP Systems by Iteratively Eliminating Two Term Common Subexpressions," presented at Asia South Pacific Design Automation Conference, Shanghai, 2005.
[36] A. G. Dempster and M. D. Macleod, "Use of minimum-adder multiplier blocks in FIR digital filters," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on]*, vol. 42, pp. 569-577, 1995.
[37] K.Koyama and T.Tsuruoka, "Speeding up elliptic cryptosystems by using a signed binary window method," presented at CRYPTO, Berlin, 1992.
[38] J.R.Reif and H.R.Lewis, "Symbolic Evaluation and the Global Value Graph," presented at Principles of Programming Languages (POPL), 1977.
[39] B.Alpern, M.N.Wegman, and F.K.Zadeck, "Detecting Equality of Variables in Programs," presented at Principles of Programming Languages (POPL), 1988.
[40] "Maple," Waterloo Maple Inc, 1998.
[41] A.W.Burks, H.Goldstine, and J. V. Neumann, "Preliminary discussion of the logical design of an electronic computing instrument," Institute for Advanced Studies, Princeton, NJ 1947.
[42] J. O. Penhollow, "Study of arithmetic recoding with applications in multiplication and division," in *University of Illinois, Urbana*: University of Illinois, Urbana, 1962.

[43] J.E.Robertson, "Theory of computer arithmetic employed in the design of the computer at the University of Illinois," Digital Computer Lab, University of Urbana June 1960.
[44] K.Hwang, *Computer Arithmetic: Principle, Architecture and Design*: Wiley, 1979.
[45] R.I.Hartley, "Subexpression sharing in filters using canonic signed digit multipliers," *IEEE Transactions on Circuits and Systems II*, vol. 43, pp. 677-688, 1996.
[46] I.-C. Park and H.-J. Kang, "Digital filter synthesis based on an algorithm to generate all minimal signed digit representations," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 21, pp. 1525-1529, 2002.
[47] M.Mehendale, S.D.Sherlekar, and G.Venkatesh, "Synthesis of multiplier-less FIR filters with minimum number of additions," presented at ICCAD, 1995.
[48] O.Gustaffson, A.G.Dempster, and L.Walhammar, "Extended results for minimum-adder constant integer multipliers," presented at IEEE International Symposium on Circuits and Systems, 2002.
[49] H. Nguyen and A. Chatterjee, "OPTIMUS: a new program for OPTIMizing linear circuits with number-splitting and shift-and-add decompositions," presented at Advanced Research in VLSI, 1995. Proceedings., Sixteenth Conference on, 1995.
[50] A.Chattejee and R.Roy, "An architectural transformation program for optimization of digital system by multi-level decomposition," presented at Design Automation Conference (DAC), 1993.
[51] A. K. Verma and P. Ienne, "Improved use of the carry-save representation for the synthesis of complex arithmetic circuits," presented at International Conference on Computer Aided Design (ICCAD), 2004.
[52] T. Kim, W. Jao, and S. Tjiang, "Arithmetic optimization using carry-save-adders," presented at Design Automation Conference (DAC), 1998. Proceedings, 1998.
[53] J. Um, T. Kim, and C. L. Liu, "Optimal allocation of carry-save-adders in arithmetic optimization," presented at (ICCAD) Computer-Aided Design, 1999. Digest of Technical Papers. 1999 IEEE/ACM International Conference on, 1999.
[54] J. Um, T. Kim, and C. L. Liu, "A fine-grained arithmetic optimization technique for high-performance low-power data path synthesis," presented at Design Automation Conference (DAC), 2000. Proceedings 2000. 37th, 2000.
[55] J. Um and T. Kim, "Layout-aware synthesis of arithmetic circuits," presented at Design Automation Conference (DAC) , 2002. Proceedings. 39th, 2002.
[56] T. G. Noll, "Carry-save arithmetic for high-speed digital signal processing," presented at Circuits and Systems, 1990., IEEE International Symposium on, 1990.
[57] A.K.Verma and P.Ienne, "Towards the Automatic Exploration of Arithmetic-Circuit Architectures," presented at Design Automation Conference, 2006.

[58] S. Gupta, M. Reshadi, N. Savoiu, N. Duff, R. Gupta, and A. Nicolau, "Dynamic common sub-expression elimination during scheduling in high-level synthesis," presented at System Synthesis, 2002. 15th International Symposium on, 2002.

[59] A. Nicolau and R. Potasman, "Incremental tree height reduction for high level synthesis," presented at Design Automation Conference, 1991. 28th ACM/IEEE, 1991.

[60] M. Martinez-Peiro, E. I. Boemo, and L. Wanhammar, "Design of high-speed multiplierless filters using a nonrecursive signed common subexpression algorithm," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on]*, vol. 49, pp. 196-203, 2002.

[61] S.K.Mitra, *Digital Signal Processing: A computer based approach*, second ed: McGraw-Hill, 2001.

[62] L. H. d. Figueiredo and J.Stolfi, "Self-validated numerical methods and applications," presented at Brazilian Mathematics Colloquium, Rio de Janeiro, Brazil, 1997.

[63] J.L.D.Comba and J.Stofli, "Affine arithmetic and its applications to computer graphhics," presented at Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI), 1993.

[64] L. H. d. Figueiredo, "Surface intersection using affine arithmetic," presented at Graphics Interface, 1996.

[65] C. Fang Fang, R. A. Rutenbar, M. Puschel, and T. Chen, "Toward efficient static analysis of finite-precision effects in DSP applications via affine arithmetic modeling," presented at Design Automation Conference, 2003. Proceedings, 2003.

[66] C. F. Fang, R. A. Rutenbar, and T. Chen, "Fast, accurate static analysis for fixed-point finite-precision effects in DSP designs," presented at Computer Aided Design, 2003. ICCAD-2003. International Conference on, 2003.

[67] D.-U. Lee, A. A. Gaffar, O.Mencer, and W.Luk, "MiniBit: bit-width optimization via affine arithmetic," presented at Design Automation Conference, Anaheim, CA, 2005.

[68] R.E.Moore, *Interval Analysis*, 1966.

[69] L. H. d. Figueiredo and J.Stolfi, "Self-validated numerical methods and applications," presented at Brazilian Mathematics Colloquium monograph (IMPA), Rio de Janeiro, 1997.