INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Design and Evaluation of a Data-Dependent Low-Power 8×8 DCT/IDCT

Cheng-Yu Pai

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillments of the Requirement

for the Degree of Master of Applied Science (Electrical) at

Concordia University

Montreal, Quebec, Canada

December 2000

ABSTRACT

Design and Evaluation of a

Data-Dependent Low-Power 8×8 DCT/IDCT

Cheng-Yu Pai[1]

Traditional fast *Discrete Cosine Transform* (DCT)/*Inverse DCT* (IDCT) algorithms have focused on reducing arithmetic complexity and have fixed run-time complexities regardless of the input. Recently, data-dependent signal processing has been applied to the DCT/IDCT. These algorithms have variable run-time complexities.

A new two-dimensional 8×8 low-power DCT/IDCT design is implemented using VHDL by applying the data-dependent signal-processing concept onto the traditional fixed-complexity fast DCT/IDCT algorithm. To reduce power, the design is based on Loeffler's fast algorithm, which uses a low number of multiplications. On top of that, zero bypassing, data segmentation, input truncation, and hardwired canonical sign-digit (CSD) multipliers are used to reduce the run-time computation, hence reduce the switching activities and the power.

When synthesized using Canadian Microelectronic Corporation 3-V 0.35 μm CMOSP technology, this FDCT/IDCT design consumes 122.7/124.9 mW with clock frequency of 40MHz and processing rate of 320M sample/sec. With technology scaling to 0.35 μm technology, the proposed design features lower switching capacitance per

---

[1] This work is supported by National Sciences and Engineering Research Council of Canada (NSERC) post-graduate scholarship. and NSERC research grants

sample, i.e. more power-efficient, than other previously reported high-performance FDCT/IDCT designs.

Keywords: Data-dependent computation, discrete cosine transform (DCT), inverse discrete cosine transform (IDCT), low power, canonical sign-digit multiplier.

Acknowledgements

I would like to express my deepest and most sincere gratitude toward my supervisors – Dr. Asim J. Al-Khalili and Dr. William E. Lynch. They have given me clear and helpful guidelines throughout my years as a master student. Above all, I wish to thank them for the great amount of time devoted to me and my work.

I wish to thank the scholarship offered by the National Sciences and Engineering Research Council of Canada (NSERC) Post-Graduate Scholarship (PGS-A), and NSERC research grants. Their financial support allows me concentrating my time and effort on my research.

I would also like to thank my fellow friends Wassim Tout, Wei Wang, and VLSI lab specialist Ted Obuchowicz for helping me throughout the technical problems with the simulation environments, and giving me their valuable opinions about the comparison strategy.

Finally, I would like to dedicate this work to my family for their love and support. I thank you all for your patience and your sacrifices. This work is as much yours as it is mine.

Table of Contents

List of Figures ..... **ix**

List of Tables ..... **x**

List of Acronyms ..... **xi**

1. Introduction **1**

1.1. Research Motivation 1

1.2. Contribution of this Thesis 3

1.3. Power Measurement Criteria 4

1.4. Thesis Organization 6

2. Background of FDCT/IDCT **7**

2.1. Definition of DCT and its Inverse 7

2.2. Choices of Algorithms 9

2.2.1. Chen's Algorithm Family 9

2.2.2. Loeffler's FDCT/IDCT Algorithm 11

2.2.3. Jeong's FDCT Algorithm 13

2.2.4. Summary and Comparison of Algorithm Complexities 14

2.3. Precision Requirements of IDCT 15

2.4. Chapter Summary 16

3. Design Choices for the FDCT/IDCT **17**

3.1. Data-Dependent Loeffler's FDCT Algorithm 17

3.1.1. Data-Dependent Bypassing Logic 17

3.1.2. Truncate Some Least-Significant Bits from Input 20

3.2. Data-Dependent Loeffler's IDCT Algorithm 24

3.3. Transpose Memory Architecture 25

3.4. Chapter Summary 28

4. Multiplier Architectures 29

4.1. Survey of Constant Multiplication Schemes 29

4.1.1. Modified Booth Multiplier 30

4.1.2. Distributed Arithmetic (DA) 30

4.1.3. Hardwired Canonical-Sign-Digit (CSD) Wallace-Tree Multiplier 31

4.1.4. Pattern-Based CSD Multiplier 34

4.2. CSD Multiplier Implementation Procedure 35

4.3. Multiplier Synthesis Result 40

4.4. Chapter Summary 42

5. Implementation 43

5.1. Hardwired CSD Multiplier Generator 43

5.2. IEEE Standard 1180-1990 IDCT Compliant 45

5.3. Pipelining Design 46

5.4. Chapter Summary 47

6. Synthesis Results 49

6.1. Synthesis Results of the Proposed Design 49

6.2. Comparison with past FDCT/IDCT VLSI implementations 50

6.3. Chapter Summary 53

7. Conclusion 54

7.1. Summary of Research 54

7.2. Conclusion 55

7.3. Possible Improvements for Future Research 56

Bibliography 59

Appendix A Truncation Test Result 65
Appendix B Sample Output of CSD Multiplier Generator 69
Appendix C Source Code of Constant Multiplier Generator 74
Appendix D IEEE Standard 1180-1990 Compliant Test Program 95

List of Figures

Figure 1: General block diagram of video compression encoder 2

Figure 2: 2-D FDCT/IDCT using row-column (separable) method 9

Figure 3: Loeffler's FDCT algorithm 11

Figure 4: Loeffler's IDCT algorithm 12

Figure 5: Jeong's fast FDCT algorithm 13

Figure 6: Setup for measuring the accuracy of a proposed 8x8 IDCT 15

Figure 7: Zero Bypassing Multiplier 18

Figure 8: Multiplication Segmentation 19

Figure 9: 2-D row-column FDCT with truncation 21

Figure 10: Test model to measure the effect of truncation 22

Figure 11: Ping-pong transpose memory 25

Figure 12: On-the-fly 8×8 Transpose Memory 26

Figure 13: States of the transpose matrix for different clock cycles 27

Figure 14: Converting binary number 0110010111 into CSD representation 33

Figure 15: Hardwired CSD multiplier for multiplying cos(3π/16) with 8-bit unsigned integer 38

Figure 16: Hardwired CSD multiplier for multiplying cos(3π/16) with 8-bit signed integer 40

Figure 17: Pipelined *kcn* block 46

List of Tables

Table 1: Transfer function of Loeffler's FDCT building blocks 11

Table 2: Transfer function of Loeffler's IDCT building blocks 12

Table 3: Complexities of different FDCT algorithms 14

Table 4: IEEE Standard 1180-1900 IDCT Precision Requirement 16

Table 5: Truncation errors against the number of truncated bits 23

Table 6: Comparison of general-purpose multiplication against ROM based multiplication 31

Table 7: Canonical sign-digit representation of $\cos(n\pi/16)$ 33

Table 8: Truth-table of b+1 36

Table 9: Truth table to simplify sign-extension 39

Table 10: Comparison of 32-bit CSD Wallace-tree multiplier with 4 different general-purpose multipliers using Xilinx 4052XL-1 FPGA technology 41

Table 11: IEEE Standard 1180-1990 Compliance for Proposed IDCT 46

Table 12: Latencies for 1-D FDCT and 1-D IDCT 47

Table 13: Latencies for 2-D FDCT and 2-D IDCT 47

Table 14: Process and Specifications of the proposed FDCT/IDCT designs 50

Table 15: Summary of specifications of several FDCT/IDCT chips 51

Table 16: Energy Efficiency (Switching Capacitances/Sample in 0.35μm technology) .. 53

Table 17: Truncation errors of test sequences: coke, salesman, and tennis 68

List of Acronyms

CCITT	International Telegraph and Telephone Consultative Committee
CMC	Canadian Microelectronic Corporation
CMG	Constant Multiplier Generator
CPA	Carry Propagate Adder
CSA	Carry Save Adder
CSD	Canonical Sign-Digit
DA	Distributed Arithmetic
dB	Decibel
DCT	Discrete Cosine Transform
DFT	Discrete Fourier Transform
FDCT	Forward Discrete Cosine Transform
FPGA	Field-Programmable Gate-Array
HDTV	High-definition TV
IDCT	Inverse Discrete Cosine Transform
IEEE	Institute of Electrical and Electronic Engineers
JPEG	Joint Photographic Experts Group
MC	Motion Compensation
ME	Motion Estimation
MHz	Mega-Hertz
MOS	Metal-Oxide Semiconductor

MPEG	Moving Picture Experts Group
MUX	Multiplexer
NMOS	N-type MOS
PMOS	P-type MOS
PSNR	Peak Signal-to-Noise Ratio
ROM	Read-Only Memory
SD	Sign-Digit
SFG	Signal Flow Graph
VLC	Variable-Length Coding
VLSI	Very Large-Scale Integration

Chapter 1

Introduction

1.1. Research Motivation

Waveform compression has been an important research topic, and it has wide industry applications. The term waveform is a generic term that can be applied to speech signal, still image, or video signals. Generally speaking, these waveforms require large storage in physical devices, and require large communication bandwidth to transmit. For example, one-hour colored 704×480 frame-size video requires 704×480 (bytes/frame) × 1.5 (for color frames) × 30 (frame/sec) × 60 (sec./min.) × 60 (min/hour) ≅ 54.7 GB to store/transmit. That is an enormous amount of data. Due to the nature of these signals, redundancies can be removed by means of waveform compression. In practice, for the video signals, one can achieve from 40:1 (for high quality) up to 80:1 (for low quality) compression ratio. In other words, one-hour of digital video requires only about 1.37 GB to store or transmit.

The discrete cosine transform (DCT) has been widely used in waveform compression because it features good energy compaction and low computational complexity. It has become an integral part of many waveform compression standards, such as JPEG, MPEG-2, MPEG-4, CCITT Recommendation H. 261 and H. 263, and HDTV. [26]

The DCT, like the Discrete Fourier Transform (DFT), is used to transform the signal to the frequency domain. Unlike DFT that uses complex exponentials as basis functions, DCT uses cosines (real numbers) as basis functions. Since the human audio-visual system is less sensitive to high frequency harmonics, waveform compression standards use DCT to transform signal to frequency domain and perform compression on the DCT coefficients.

As an example, for video compression, both temporal and spatial redundancies are eliminated as shown in Figure 1. The motion estimation/motion compensation (ME/MC) block is used to reduce temporal redundancies due to high correlation among adjacent frames. The forward DCT (FDCT) together with the quantizer is used to reduce spatial redundancies. Finally, the variable-length coder (VLC) is used to reduce coding redundancies.

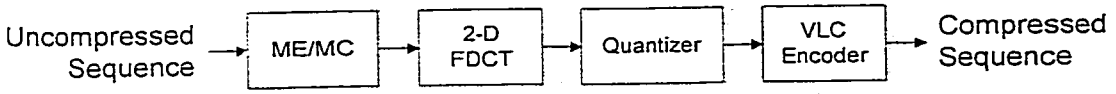


Figure 1: General block diagram of video compression encoder

With the advances in communication and VLSI technologies, it is expected that video telephony/conferencing on mobile devices will be more and more common in the future. Because mobile devices operate with battery power, in order to increase the battery life and recharging time, mobile devices always have stringent power specifications. Also, to save valuable communication bandwidth, video compression is always performed on these applications. As a result, the DCT chip is an integral part of video communication mobile devices, and the design of a low-power DCT chip is an

important problem. In this thesis, a low-power data-dependent DCT/IDCT design is presented to meet this need.

1.2. Contribution of this Thesis

Many earlier fast DCT algorithms are aimed at reducing the number of multiplications because general-purpose multipliers are assumed to be the basic hardware elements for computing the DCT. Later on, other design techniques, such as digital filtering and distributed arithmetic (DA), are also used to compute DCT [9]. In more recent works, data-dependent DCT algorithms have been introduced in [19]-[21][28]. Unlike traditional algorithms, which have fixed-computation complexity, data-dependent algorithms have variable run-time complexities that depend on the statistical properties of the input data. They may yield fewer or more computations in the run-time than the fixed complexity algorithms.

To reduce the power consumption, optimizations are performed at both the algorithmic level and the architectural level. The low-complexity Loeffler's [10] fast FDCT/IDCT algorithm is chosen to reduce the hardware requirement, which in turn reduces power.

The concept of data-dependent signal processing has also been applied to the fixed-complexity Loeffler's algorithm to reduce the switching activities. For both the FDCT and IDCT, zero-bypassing logic is inserted into the circuit to bypass redundant computations. The zero-bypassing logic takes advantages of high correlation among input data for the FDCT, and high proportion of zero inputs for the IDCT. Furthermore, the FDCT design also truncates bits from its input to reduce the amount of data to be

processed, consequently reducing power consumption. The error introduced by the truncation is also analyzed in the thesis.

Further architectural optimization is performed on multipliers. Since multiplication is a high complexity operation compared to addition, the FDCT/IDCT designs use hard-wired canonical sign-digit (CSD) Wallace-tree multipliers since it utilizes minimum amount of power over the multipliers surveyed.

To summarize, the main contributions made in this thesis are listed as following:

- Introduce new data-dependent FDCT/IDCT algorithm by merging the data-dependent processing concept with fast FDCT/IDCT algorithm.
- Empirically study the effect of truncating some least significant bits of the FDCT input to save computation.
- Derive detailed design procedure for implementing low-power constant-coefficient multipliers.
- Develop a code generator written in C++ that generate VHDL code of constant multipliers for different specifications.

1.3. Power Measurement Criteria

In VLSI design, it is always difficult to compare one design with another due to different process technology (feature size), supply voltages, operating frequency, implementation approach (full-custom, semi-custom, etc.), optimization parameters, and design algorithm/architectures. Depending on the design goal, several comparison methods have been suggested and used, such as A, P, T, PT, AT, AT^2, etc., where A stands

for area, T stands for time (delay), and P stands for power. Unfortunately, these measurement criteria give rough measures, which do not take all process technology into account.

In this thesis, the proposed design is compared with other reported designs by comparing the *switching capacitance per sample*, which has been used in [19-21][28]. In VLSI design, power can be estimated from the well-known formula:

$$P \cong \left(\frac{1}{2} p_i \cdot C_L\right) \cdot f_{Clk} \cdot V_{DD}^2 \qquad \ldots (1)$$

where P is the power, p_i is the switching probability, C_L is the load capacitance of the DCT/IDCT in this case, f_{Clk} is the clock frequency, and V_{DD} is the supply voltage. From equation (1), the switching capacitance is defined as $\frac{1}{2} p_i \cdot C_L$, and the switching capacitance per sample can be obtained by dividing the switching capacitance by the number of input/output samples per clock cycle. Since switching capacitance is directly proportional to power, this measurement method leads to comparing relative energy efficiency rather than absolute values such as in AP, PT, etc. It indicates how much power (switching capacitance) is required to obtain one output.

The main advantage of this method is that it takes out the effect of different process technology by performing technology scaling. Thus to compare one design of one technology with another design of different technology, technology scaling is first performed on the measured power, then the effects of clock frequency and voltage supply are factored out to obtain the switching capacitance per sample.

1.4. Thesis Organization

The organization of this thesis is as follows: in Chapter 2, the definition of discrete cosine transform and its inverse and the algorithm used in the proposed design are described. Chapter 3 describes the data-dependent signal processing concept and how it is incorporated into the design. Chapter 4 summarizes the pros and cons of several multiplier architectures, and provides a detailed design procedure for the selected multiplier – hardwired canonical sign-digit (CSD) Wallace-tree multiplier. Chapter 5 describes the design automation effort made to facilitate the implementation of hardwired multipliers. The IDCT accuracy test result and pipelining design are also described. In Chapter 6, synthesis results of the new FDCT/IDCT designs are reported and compared against previously reported implementations.

Chapter 2

Background of FDCT/IDCT

Since there exist many DCT definitions [38], the forward DCT (FDCT) and its inverse (IDCT) are defined in Section 2.1 for clarification.

Numerous fast algorithms for both FDCT and IDCT have been reported in the literature. Most of them attempted to minimize the number of additions and multiplications ([1], [8]-[13], [17-18], [29], etc.). These algorithms usually take advantage of the symmetry in the cosine basis functions, and the computation complexity is fixed for all input data (data independent algorithm). Since multiplication requires more hardware and computation time than adders, fewer multiplications imply low power.

In Section 2.2, several existing fast FDCT/IDCT algorithms are studied and compared. The Loeffler's [10] algorithm is chosen to be the fundamental FDCT/IDCT algorithm of the proposed design.

Since the FDCT is always followed by a quantizer, its precision requirement is not high. On the contrary, the IDCT is used to perform inverse transformation at both the encoder and the decoder, which requires high precision. It needs to conform to IEEE Standard 1180-1990, which is described in 2.3.

2.1. Definition of DCT and its Inverse

The N-point 1-D forward DCT (FDCT) is defined in equation 2:

$$X(n) = \sqrt{\frac{2}{N}} C(n) \sum_{k=0}^{N-1} x(k) \cos \frac{(2k+1)n\pi}{2N} \qquad (2)$$

The N-point 1-D inverse DCT (IDCT) is define in equation 3:

$$x(n) = \sum_{k=0}^{N-1} \sqrt{\frac{2}{N}} C(k) X(k) \cos \frac{(2n+1)k\pi}{2N} \qquad (3)$$

where $C(n) = \begin{cases} 1/\sqrt{2} & n = 0 \\ 1 & n = 1, 2, \ldots, N-1 \end{cases}$

Similarly, the $N \times N$ 2-D FDCT is defined as follows: [4]

$$X(u,v) = \frac{2}{N} C(u) C(v) \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} x(i,j) \cos\left(\frac{(2i+1)u\pi}{2N}\right) \cos\left(\frac{(2j+1)v\pi}{2N}\right) \ldots (4)$$

and the $N \times N$ 2-D IDCT is defined as:

$$x(i,j) = \frac{2}{N} \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} C(u) C(v) X(u,v) \cos\left(\frac{(2i+1)u\pi}{2N}\right) \cos\left(\frac{(2j+1)v\pi}{2N}\right) \ldots (5)$$

Notice that 2-D $N \times N$ FDCT/IDCT is a separable transformation, which means that it can be obtained by first performing 1-D N-point DCT/IDCT on the rows, then performing 1-D N-point DCT/IDCT on the columns, or the other way around. This method of computing 2-D DCT/IDCT is generally referred to as row-column method or indirect method. The general block diagram of this method is shown in Figure 2.

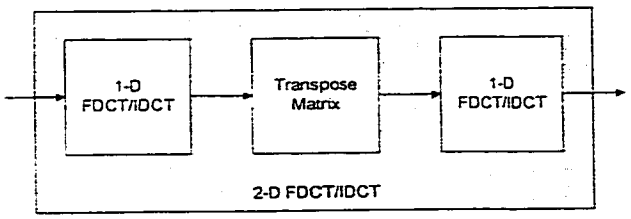


Figure 2: 2-D FDCT/IDCT using row-column (separable) method

The row-column method is the most popular method in VLSI implementations ([2]-[7], [9], [14]-[16], etc.). Also, since the 8×8 block size is used by MPEG and other standards, in this thesis, the FDCT/IDCT design presented uses 8×8 block size.

2.2. Choices of Algorithms

Many fast DCT/IDCT algorithms have been reported in the literature. In this section, several fixed-complexity algorithms are reviewed and compared based on their arithmetic complexities. The comparison suggests that Loeffler's FDCT/IDCT algorithm is the most efficient and is used as the basis of the proposed design.

2.2.1. Chen's Algorithm Family

Chen's fast algorithm [1] reported in 1977 is by far the most widely used DCT/IDCT algorithm. It has been used in [2]-[7] and many other papers. It is a fixed-complexity algorithm. The idea of Chen's algorithm is to exploit the symmetry in the DCT/IDCT transformation matrix. The 8x8 DCT can be written in matrix form:

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \\ X(4) \\ X(5) \\ X(6) \\ X(7) \end{bmatrix} = \begin{bmatrix} a & a & a & a & a & a & a & a \\ b & d & e & g & -g & -e & -d & -b \\ c & f & -f & -c & -c & -f & f & c \\ d & -g & -b & -e & e & b & g & -d \\ a & -a & -a & a & a & -a & -a & a \\ e & -b & g & d & -d & -g & b & -e \\ f & -c & c & -f & -f & c & -c & f \\ g & -e & d & -b & b & -d & e & -g \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \\ x(4) \\ x(5) \\ x(6) \\ x(7) \end{bmatrix} \tag{6}$$

where

$$[a \quad b \quad c \quad d \quad e \quad f \quad g] = \frac{1}{2}\left[\cos\frac{\pi}{4} \quad \cos\frac{\pi}{16} \quad \cos\frac{\pi}{8} \quad \cos\frac{3\pi}{16} \quad \cos\frac{5\pi}{16} \quad \cos\frac{3\pi}{8} \quad \cos\frac{7\pi}{16}\right]$$

Since the even rows of the transformation matrix are even symmetric and odd rows are odd-symmetric, by exploiting the symmetry and separating even and odd rows, equation (6) can be rewritten as follows:

$$\begin{bmatrix} X(0) \\ X(2) \\ X(4) \\ X(6) \end{bmatrix} = \begin{bmatrix} a & a & a & a \\ c & f & -f & -c \\ a & -a & -a & a \\ f & -c & c & -f \end{bmatrix} \times \begin{bmatrix} x(0)+x(7) \\ x(1)+x(6) \\ x(2)+x(5) \\ x(3)+x(4) \end{bmatrix} \tag{7}$$

$$\begin{bmatrix} X(1) \\ X(3) \\ X(5) \\ X(7) \end{bmatrix} = \begin{bmatrix} b & d & e & g \\ d & -g & -b & -e \\ e & -b & g & d \\ g & -e & d & -b \end{bmatrix} \times \begin{bmatrix} x(0)-x(7) \\ x(1)-x(6) \\ x(2)-x(5) \\ x(3)-x(4) \end{bmatrix}$$

Similarly, the 1-D IDCT can be rewritten as follows:

$$\begin{bmatrix} Y(0) \\ Y(1) \\ Y(2) \\ Y(3) \end{bmatrix} = \begin{bmatrix} a & c & a & f \\ a & f & -a & -c \\ a & -f & -a & c \\ a & -c & a & -f \end{bmatrix} \times \begin{bmatrix} X(0) \\ X(2) \\ X(4) \\ X(6) \end{bmatrix} + \begin{bmatrix} b & d & e & g \\ d & -g & -b & -e \\ e & -b & g & d \\ g & -e & d & -b \end{bmatrix} \times \begin{bmatrix} X(1) \\ X(3) \\ X(5) \\ X(7) \end{bmatrix} \tag{8}$$

$$\begin{bmatrix} Y(7) \\ Y(6) \\ Y(5) \\ Y(4) \end{bmatrix} = \begin{bmatrix} a & c & a & f \\ a & f & -a & -c \\ a & -f & -a & c \\ a & -c & a & -f \end{bmatrix} \times \begin{bmatrix} X(0) \\ X(2) \\ X(4) \\ X(6) \end{bmatrix} - \begin{bmatrix} b & d & e & g \\ d & -g & -b & -e \\ e & -b & g & d \\ g & -e & d & -b \end{bmatrix} \times \begin{bmatrix} X(1) \\ X(3) \\ X(5) \\ X(7) \end{bmatrix}$$

2.2.2. Loeffler's FDCT/IDCT Algorithm

Loeffler's 1-D 8-point FDCT algorithm uses 11 multiplications and 29 additions only. The signal flow graph (SFG) of an 8-point 1-D DCT is shown in Figure 3, and the transfer functions of the building blocks are given in Table 1.

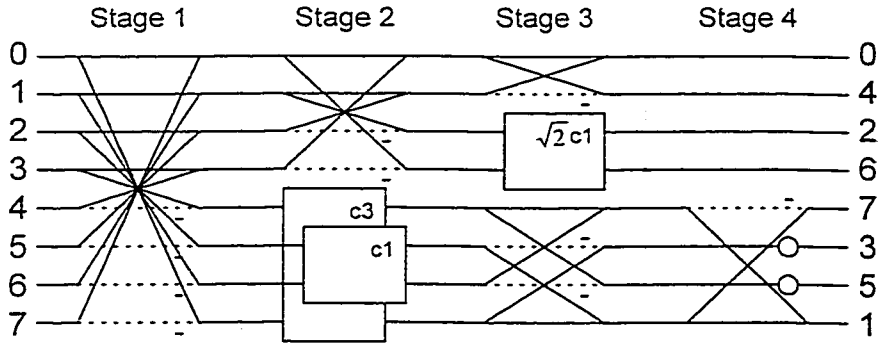


Figure 3: Loeffler's FDCT algorithm [10]

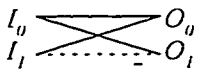
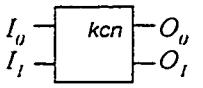
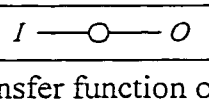
Symbol	Equation	Effort
	$O_0 = I_0 + I_1$ $O_1 = I_0 - I_1$	2 add
	$O_0 = I_0\left(k\cos\frac{n\pi}{2N}\right) + I_1\left(k\sin\frac{n\pi}{2N}\right)$ $O_1 = -I_0\left(k\sin\frac{n\pi}{2N}\right) + I_1\left(k\cos\frac{n\pi}{2N}\right)$	3 mult. + 3 add
	$O = \sqrt{2}I$	1 mult.

Table 1: Transfer function of Loeffler's FDCT building blocks [10]

Notice that the second building block (*kcn*) requires only 3 multiplications and 3 additions instead of 4 multiplications and 2 additions when equation 9 is used.

$$\begin{cases} O_0 = aI_0 + bI_1 = (b-a)I_1 + a(I_0 + I_1) \\ O_1 = -bI_0 + aI_1 = -(a+b)I_0 + a(I_0 + I_1) \end{cases}, \text{where } a = k\cos\frac{n\pi}{2N}, b = \sin\frac{n\pi}{2N} \qquad (9)$$

By reversing the transfer function of each building block shown in Table 1, and reversing the signal-flow direction, it is easy to show that the IDCT has SFG shown in Figure 4 with building block transfer function shown in Table 2. Notice that the Loeffler IDCT algorithm has the same arithmetic complexity as in the FDCT case (11 multiplications and 29 additions). Notice also that division by 2 is considered using no operation since it can be realized by ignoring the least-significant bit of the value to be divided.

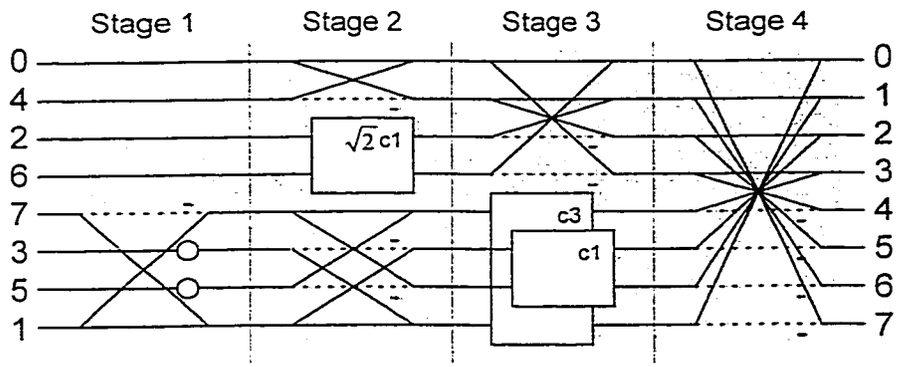


Figure 4: Loeffler's IDCT algorithm

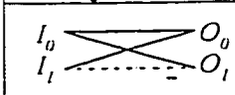
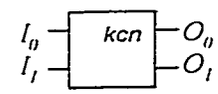
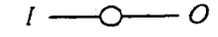
Symbol	Equation	Effort
	$O_0 = (I_0 + I_1)/2$ $O_1 = (I_0 - I_1)/2$	2 add
	$O_0 = I_0\left(\frac{1}{k}\cos\frac{n\pi}{2N}\right) - I_1\left(\frac{1}{k}\sin\frac{n\pi}{2N}\right)$ $O_1 = I_0\left(\frac{1}{k}\sin\frac{n\pi}{2N}\right) + I_1\left(\frac{1}{k}\cos\frac{n\pi}{2N}\right)$	3 mult. + 3 add
	$O = I/\sqrt{2}$	1 mult.

Table 2: Transfer function of Loeffler's IDCT building blocks

2.2.3. Jeong's FDCT Algorithm

Jeong's [13] 8-point FDCT algorithm reported in 1998 uses 28 additions, 12 multiplications. This algorithm is special because it performs most multiplications at the final stage and requires fewer multiplication stages than other algorithms, so propagation errors occurring in the fixed-point computation can be reduced.

By separating even and odd points in the DCT, this algorithm uses trigonometric identities to reduce the number of multiplication needed to calculate DCT.

- Even points:

$$X(2l) = \sqrt{\frac{2}{N}}\alpha(n)\sum_{k=0}^{N/2-1}[x(k)+x(N-1-k)]\cos\frac{(2k+1)2l\pi}{2N}, \text{ where } l \in [0,3]$$

- Odd points:

$$X(2l+1) = \left(2\cos\frac{(2l+1)\pi}{2N}\right)^{-1}\sqrt{\frac{2}{N}}\alpha(n)\times$$
$$\sum_{m=0}^{N/4-1}\left\{[y(2m-1)+y(2m)]\cos\frac{(2l+1)2m\pi}{N}+[y(2m)+y(2m+1)]\cos\frac{(2l+1)(2m+1)\pi}{N}\right\}$$
where $y(k) = x(k) - x(N-1-k)$ and $y(-1) = 0$

The signal flow graph is shown in Figure 5.

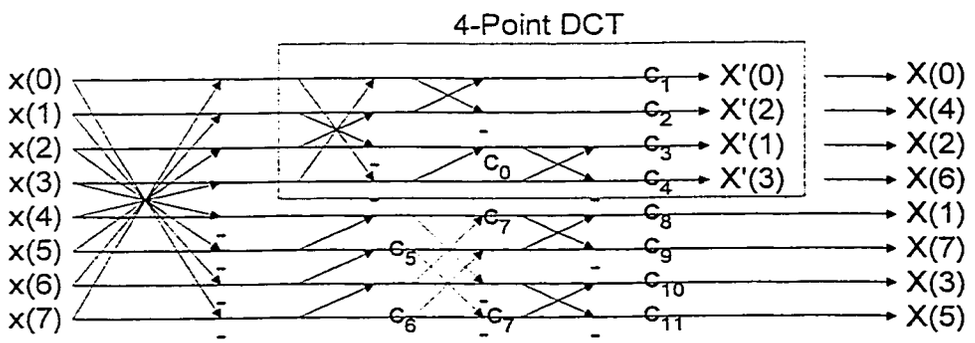


Figure 5: Jeong's fast FDCT algorithm [13]

where $c_0 = 1/C_4^1, c_1 = \sqrt{2}/4, c_2 = C_4^1/2, c_3 = C_4^1/C_8^1, c_4 = C_4^1/4C_8^3,$
$c_5 = c_3, c_6 = 1/C_8^1, c_7 = C_8^3/C_8^1, c_8 = C_8^1/4C_{16}^1, c_9 = C_8^1/4C_{16}^7, c_{10} = C_8^1/4C_{16}^3,$
$c_{11} = C_8^1/4C_{16}^5$, and $C_m^n \equiv \cos \pi n/m$

2.2.4. Summary and Comparison of Algorithm Complexities

Since in VLSI implementation, each computation, i.e. addition and multiplication, requires hardware and consumes power, algorithms with fewer addition/multiplication lead to lower power. Also, since multiplication requires more power than addition, one algorithm is better than another if it requires fewer multiplications (for integer operations).

Table 3 summarizes the complexity of several fixed-complexity FDCT algorithms. In [34], Duhamel demonstrated that the theoretical lower bound of an 8-point DCT is 11 multiplications. Since the number of multiplication in Loeffler's [10] algorithm reaches the theoretical lower bound and the number of addition is not worse than other algorithms (except Jeong's), the Loeffler's algorithm is chosen.

Algorithm	Chen [1]	Wang [31]	Lee [11]	Vetterli [32]	Suehiro [33]	Hu [12]	Loeffler [10]	Jeong [13]
Multiplication	16	13	12	12	12	12	11	12
Add	26	29	29	29	29	29	29	28

Table 3: Complexities of different FDCT algorithms (adapted column 1-7 from Table 1 in [10])

2.3. Precision Requirements of IDCT

In video compression, the precision requirement of FDCT is not high because it is always followed by heavy quantization. On the contrary, since the IDCT is used for sequence reconstruction, it is important for IDCT to be computed with high precision.

The IEEE Standard 1180-1990 [27] defines the specification for the implementations of IDCT. The step for measuring the accuracy of an 8x8 IDCT block is shown in Figure 6.

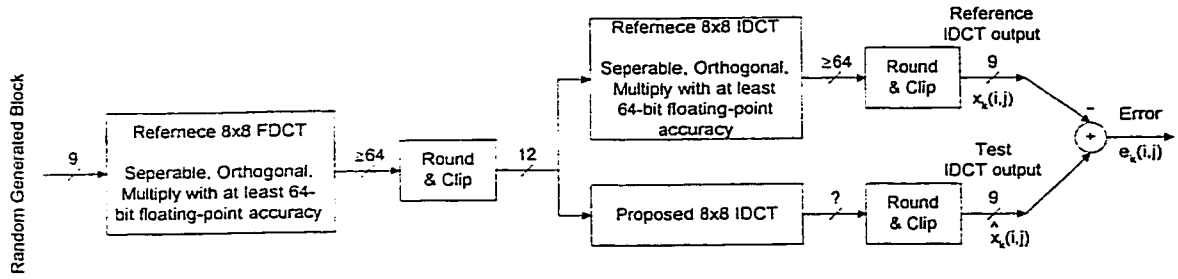


Figure 6: Setup for measuring the accuracy of a proposed 8x8 IDCT (figure 2 in [27])

The standard defines a random number generator that can generate numbers within lower and upper bounds (-L and H) inclusive. Based on these random numbers, 10000 8x8 blocks for (L=256, H=255), (L=H=5) and (L=H=300) are used as input for reference FDCT (see Figure 6), and passed through the diagram shown in Figure 6. The error, $e_k(i,j)$, is defined to be the difference between the "test" IDCT output and the "reference" IDCT output, i.e.:

$$e_k(i,j) = \hat{x}_k(i,j) - x_k(i,j)$$

The standard defines the following terms to measure the error (see Table 4).

Term	Definition	Maximum Magnitude
Peak error (*ppe*)	Max($\lvert e_k(i,j) \rvert$)	1
Mean square error for any pixel (*pmse*)	$pmse(i,j) = \frac{\sum_{k=1}^{10000} e_k^2(i,j)}{10000}$	0.06
Overall mean square error (*omse*)	$omse(i,j) = \frac{\sum_{i=0}^{7}\sum_{j=0}^{7}\sum_{k=1}^{10000} e_k^2(i,j)}{64\times 10000}$	0.02
Mean error for any pixel (*pme*)	$pme(i,j) = \frac{\sum_{k=1}^{10000} e_k(i,j)}{10000}$	0.015
Overall mean error (*ome*)	$ome(i,j) = \frac{\sum_{i=0}^{7}\sum_{j=0}^{7}\sum_{k=1}^{10000} e_k(i,j)}{64\times 10000}$	0.0015
For all-zero input, the proposed IDCT shall generate all-zero output.		

Table 4: IEEE Standard 1180-1900 IDCT Precision Requirement

2.4. Chapter Summary

In this chapter, the FDCT and IDCT are defined. Several fast fixed-complexity FDCT/IDCT algorithms are reviewed and their computational complexities are summarized in Table 3. Since low arithmetic complexity usually implies low power, the Loeffler's algorithm is used as the basis of the proposed design.

The IEEE 1180-1990 standard is also described in this chapter. The standard defines the precision requirements of IDCT, which the new IDCT design will conform to.

In the next chapter, detailed discussion/description is presented to show how the data-dependent concept is integrated into Loeffler's FDCT/IDCT algorithm to make it a data-dependent algorithm.

Chapter 3

Design Choices for the FDCT/IDCT

In this chapter, the data-dependent processing concept is applied to Loeffler's FDCT/IDCT algorithm. In Section 3.1 and 3.2, data-dependent bypassing logic is inserted into Loeffler's FDCT/IDCT algorithms to achieve more power reduction. To further reduce the computation complexity, the least significant bits of the FDCT inputs are truncated. The effect of truncation is studied in detail.

Since the row-column method is used to compute the 2-D FDCT/IDCT by using two 1-D FDCT/IDCT with a transpose memory in between (see Figure 2), Section 3.3 studies two transpose memory architectures. The on-the-fly transpose memory architecture is used in this work.

3.1. Data-Dependent Loeffler's FDCT Algorithm

To have a power-efficient design, data-dependent algorithm and truncation techniques are adopted into Loeffler's FDCT algorithm.

3.1.1. Data-Dependent Bypassing Logic

Loeffler's FDCT algorithm performs several butterfly operations on the inputs (see Figure 3). In general, the inputs are well correlated for the FDCT. Thus, the subtractions used in the butterfly are very likely to produce zeros or small numbers. Since

most multiplications are performed in the *kcn* blocks, and the inputs of the *kcn* blocks are the results of subtractions, adding zero bypassing logic in front of each multiplication in the *kcn* blocks will reduce the number of multiplications. As shown in Figure 7, the zero bypassing logic only adds the non-zero-detection logic (AND gate), a register, and a multiplexer (MUX) to the circuit. The overhead, both the area and speed, introduced is small comparing to the multiplier itself.

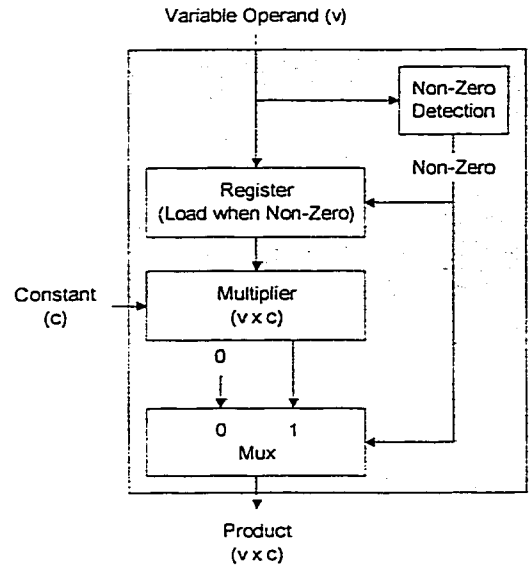


Figure 7: Zero Bypassing Multiplier

By segmenting the inputs of multipliers into several smaller chunks (data segmentation), further computational reduction can be achieved by taking advantage of the fact that the inputs of the *kcn* block are very likely to be small numbers because the inputs are obtained from butterflying highly correlated data. Thus, instead of multiplying x by c directly, the multiplication is done by breaking x into m segments, performing multiplication on each segment, and then adding the products together with proper offset if necessary (see Figure 8). The sum of the products is still $x \times c$. By inserting bypassing logic in front of each smaller multiplier, part of the small number inputs can be bypassed,

consequently reducing the switching activities and the power. For example, if x=00000111$_b$ (7$_d$), with two segments, $x \times c$ is performed as $(0000 \times c) << 4 + 0110 \times c$. With zero-bypassing logic inserted, $0000 \times c$ is bypassed and uses no operation.

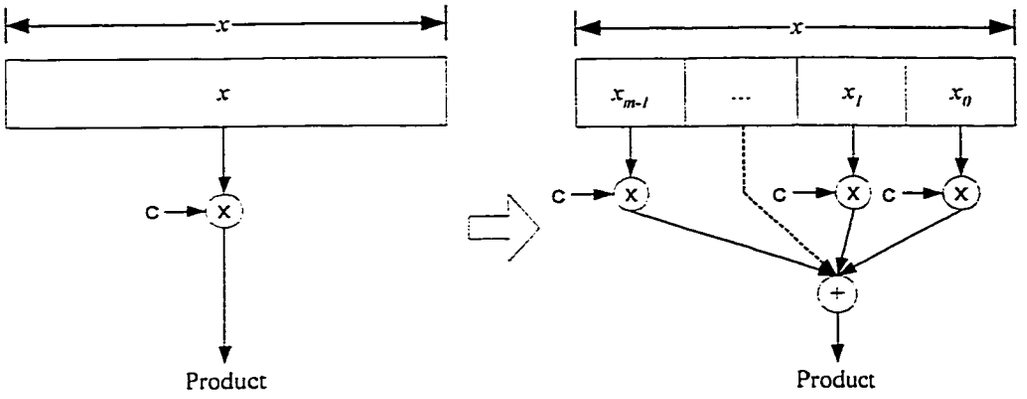


Figure 8: Multiplication Segmentation

The choice of segment size affects the probability of zero bypassing. One extreme is that there is only one segment, which is direct multiplication of $x \times c$. The other extreme is that each segment is one bit only, which is essentially performing shift-and-add operation. Theoretically, if we use segment of one bit only, one can achieve highest bypassing probability and uses lowest amount of multiplication. However, it requires the largest number of addition to add partial products to produce the final product. For n-segment, one would require to add n partial products together. Having more segments implies more complicated control logic and delay to produce the final result. Thus, having the trade-off between the probability of bypassing and the segmentation overhead in mind, we decide to use two segments for FDCT multiplications. It allows bypassing of small numbers while keeping the segmentation overhead small since there are only two partial products to be added.

3.1.2. Truncate Some Least-Significant Bits from Input

Since the IEEE standard [27] defines only the precision requirements for the IDCT, and since the FDCT is usually followed by quantization, in this thesis, some least-significant bits (LSBs) of the FDCT are truncated. Truncating input bits results in less computation, consequently, reduces power consumption and increases the speed. On the other hand, truncation introduces error at the output. Although some error introduced by the truncation will be compensated by the heavy quantization that follows the FDCT module, the error still exist. Thus, truncation allows trade-off between power and error. The goal is to find the *best* strategy to truncate input bits so that the error is in *acceptable* range depending on the application.

In 2-D 8x8 FDCT, there are eight 8-point 1-D FDCT in the first dimension (rows), and eight 8-point 1-D FDCT in the second dimension (columns). Let *Trunc(d,n)* denote the number of bits to be truncated from the n-th 1-D FDCT of dimension d, where $d = 1$ (row), 2 (column) and $n = 0...7$. The truncation for all eight inputs of any 1-D 8-point FDCT is the same. Figure 9 illustrates the detailed view of 2-D row-column FDCT with truncation.

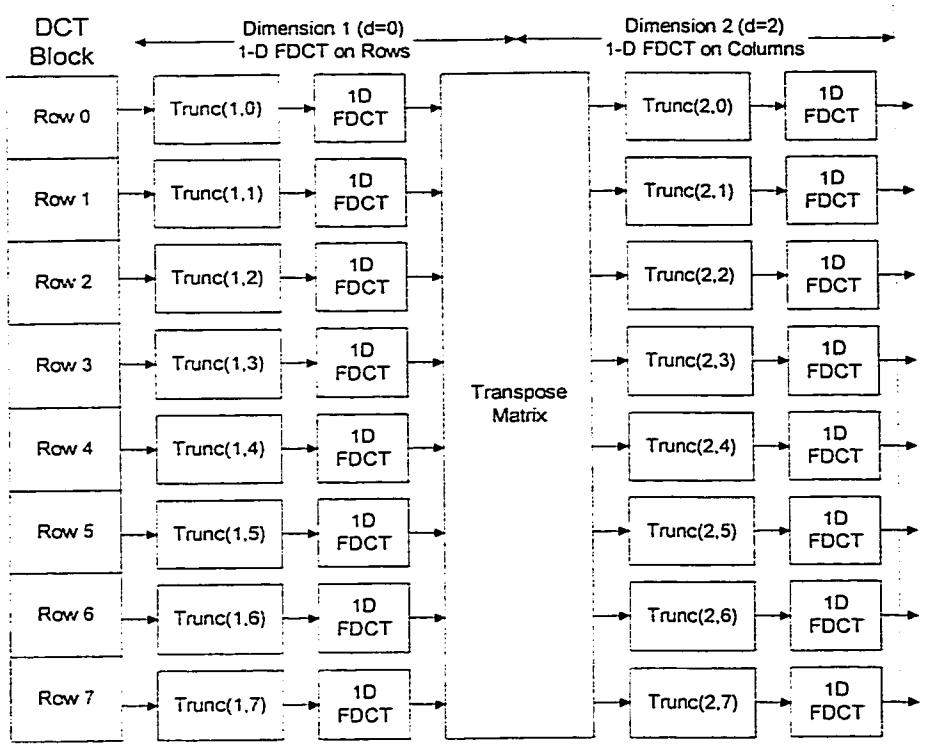


Figure 9: 2-D row-column FDCT with truncation

If we allow truncating at most m bits from each 1-D FDCT, since there are 16 1-D FDCT blocks, there are a total of $(m+1)^{16}$ possible combinations (including no truncation for m=0). Even when m is small, say m=1, there are still 65536 possibilities to be examined. Fortunately, not all combinations are valid from the distortion point of view.

In practice, since human eyes/ears are less sensitive to high frequency signal components, higher frequency FDCT coefficients (larger n) are quantized more heavily than the lower frequency coefficients. This fact suggests that the effect of truncation in higher frequency FDCT coefficients is less than the lower frequency coefficients. This argument leads to the following equation.

$$Trunc(d,n_1) \le Trunc(d,n_2) \text{ if } n_1 < n_2 \quad (10)$$

Further test cases reduction can be achieved due to the fact that the transpose matrix distributes all coefficients computed in each of the first-dimension FDCT modules

to all second-dimension FDCT modules. Thus, all first-dimension (d=1) FDCT modules are equally important, i.e.:

$$\forall n : Trunc(1,n) = k, \text{ where } k \text{ is a constant} \qquad (11)$$

Since the truncation error introduced in the first stage affects entire second stage, to have a more accurate result, k=0 (no truncation at the first dimension FDCT blocks) is used in the design of FDCT.

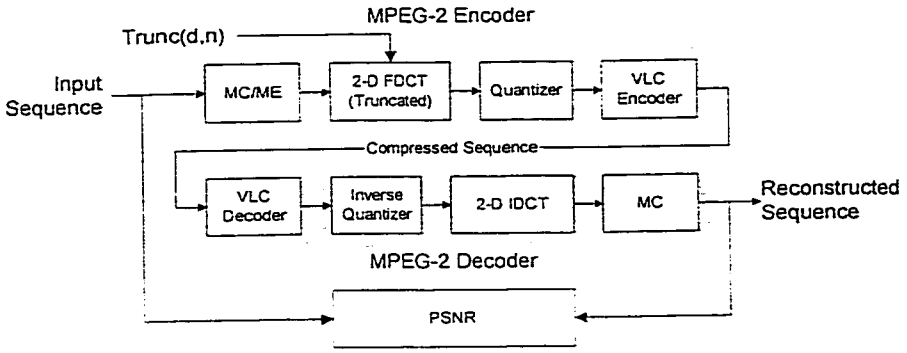


Figure 10: Test model to measure the effect of truncation

To have a quantitative measure of the truncation effect, standard MPEG-2 encoder is modified as the test model (see Figure 10). By changing the *Trunc*(2,*n*), different PSNR values are measured. The PSNR values are then compared against the reference: PSNR of no truncation (*Trunc(2,n)*=0 for all n). Smaller PSNR difference indicates smaller distortion introduced due to truncation. The truncation error is defined as:

Truncation Error = Average PSNR(reference) – Average PSNR(truncation) (12)

Since the goal is to save power, one combination is better than another if it truncates more bits, but has higher PSNR (smaller truncation error), i.e.

$$\sum_{n=0}^{7} Trunc_{Case1}(2,n) > \sum_{n=0}^{7} Trunc_{Case2}(2,n), \text{ and } PSNR_{Case1} > PSNR_{Case2} \qquad (7)$$

Three test video sequences (coke, salesman, and tennis) are used to measure the truncation errors. Each sequence has 180 frames and is encoded using pure I-frames at 8 Mb/s. The FDCT is computed with fixed-point calculation with 11-bit precision after binary points.

To show the effect of truncation, all 165 possible combinations are using m=3 (truncate at most 3 bits) and *Trunc*(1,n)=0 (no truncation for first-dimension FDCT). The testing results (truncation errors) are shown in Appendix A.

Table 5 illustrates the best truncation patterns and its average truncation error compared to all other truncation patterns with the same total truncated bit. In this thesis, truncation pattern *Trunc*(1,n)=0 and *Trunc*(2,n)={1,1,1,1,1,1,1,1} is used in the implementation of the FDCT because its truncation error is moderate (around 0.5 dB).

Total Truncated Bits	Trunc(2,n)	Truncation Error (dB)	Total Truncated Bits	Trunc(2,n)	Truncation Error (dB)
0	00000000	0.0000	13	11122222	1.5023
1	00000001	0.0621	14	11222222	1.6806
2	00000011	0.1237	15	12222222	1.8576
3	00000111	0.1831	16	22222222	2.0400
4	00001111	0.2398	17	22222223	2.6143
5	00011111	0.3064	18	22222233	3.1380
6	00111111	0.3721	19	22222333	3.5924
7	01111111	0.4403	20	22223333	3.9806
8	11111111	0.5136	21	22233333	4.3642
9	11111112	0.7327	22	22333333	4.7059
10	11111122	0.9497	23	23333333	5.0382
11	11111222	1.1406	24	33333333	5.3550
12	11112222	1.3161			

Table 5: Truncation errors against the number of truncated bits

3.2. Data-Dependent Loeffler's IDCT Algorithm

Like the FDCT, row-column method is used to compute the 2-D IDCT. Due to the heavy quantization of the encoder (for high compression), a high proportion of the coefficients are expected to be zero at the input of the first-dimension IDCT.

One problem with the Xanthopoulos's data-dependent IDCT designs in [19]-[21] is that they may result in more computation than the fixed-complexity fast algorithms. In the worst case, such as the input does not satisfy the assumed statistical property, the data-dependent design in [19]-[21] may yield as high as 1024 multiplications for 2D IDCT, i.e. degenerates to its base algorithm (direct IDCT computation).

In this work, like the FDCT, zero-bypassing logics are inserted into the IDCT circuit to reduce the number of computation. Since zero-bypassing logic does not increase the number of computation, even at the worst situation, the data-dependent design yields the same complexity as the fundamental Loeffler's algorithm. In other words, in the worst scenario (none of the bypassing logic active), data-dependent Loeffler's 2D IDCT algorithm uses 176 multiplications (2 dimensions x 8 rows (columns)/dimension x 11 multiplication/row (column)).

In real life, some zero-bypassing logics will be active, and the number of multiplications starts to depend on the distribution of input data. For instance, if there is one non-zero coefficients in the input of the 1-D IDCT, data-dependent Loeffler's IDCT algorithm requires 0, 2, 5 or 6 multiplications depending on the position of non-zero input. If the probability of the non-zero input position is the same for all 8 inputs, the algorithm requires only 3.25 multiplications in average. Thus, by applying zero-bypassing logic onto Loeffler's IDCT algorithm, the fixed-complexity algorithm is

transformed into a data-dependent algorithm. The new 2-D IDCT multiplication lower bound is the same as Xanthopoulos' (0), while the upper bound is significantly reduced from 1024 down to 176.

3.3. Transpose Memory Architecture

There are various ways to transpose 8× 8 matrix in hardware. The trivial way is to have two matrices (as shown in Figure 11). They are used for read and write alternatively (ping-pong buffering). Two matrices are required since the data arrives row-by-row.

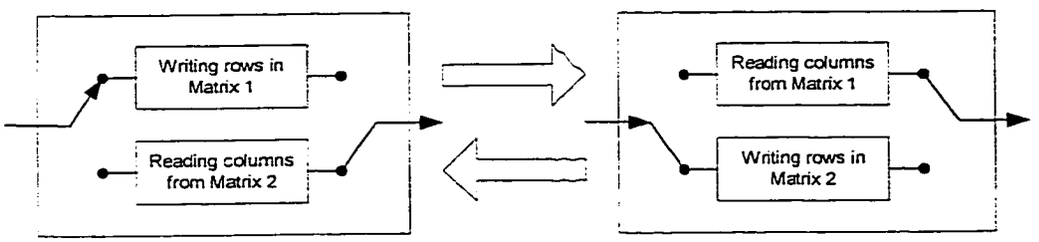


Figure 11: Ping-pong transpose memory

Another way to transpose a matrix is reported in [28]. As shown in Figure 12, only one matrix is required. Data is transposed on the fly by changing the shifting direction (top-to-bottom or left-to-right).

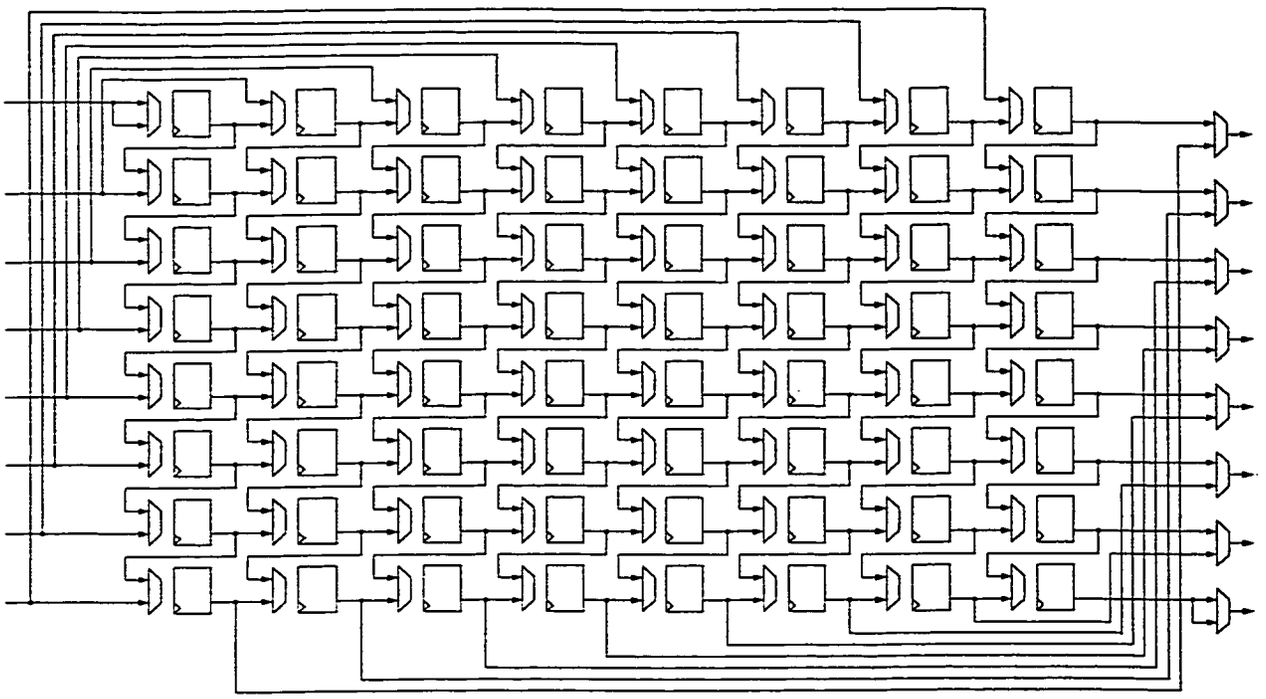


Figure 12: On-the-fly 8×8 Transpose Memory [28]

The state of the transposition matrix for clock cycles is illustrated in Figure 13. To fill up the matrix, from clock cycle 1 to 8, shifting direction is top-to-bottom. From clock cycle 9 to 16, the shifting direction is left-to-right. From clock cycle 17 to 24, the shifting direction is top-to-bottom. Clock cycle 25 is identical to clock cycle 9, and so on.

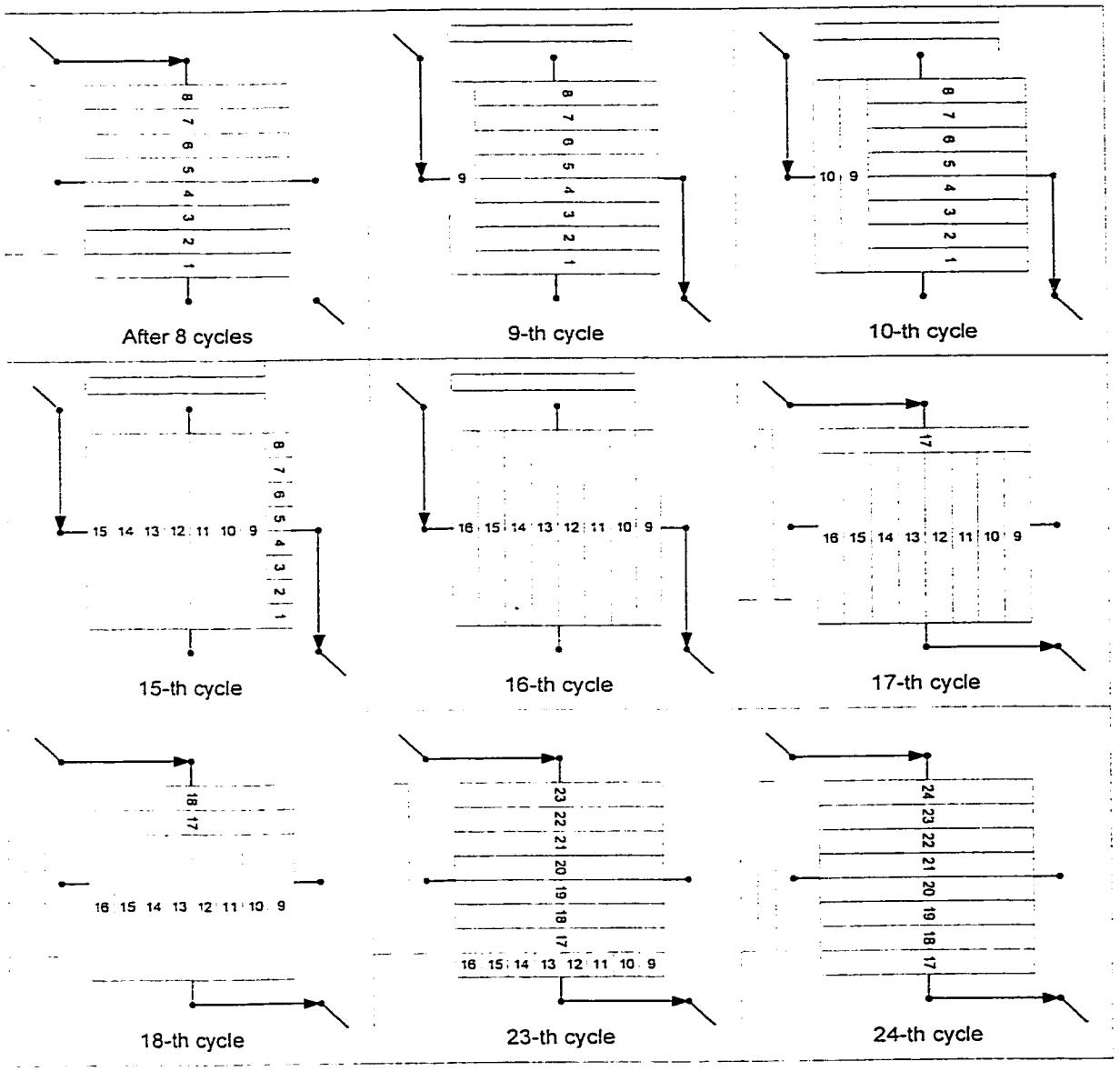


Figure 13: States of the transpose matrix for different clock cycles

Since n-bit element 8×8 matrix is built with $64n$ flip-flops, if n is large, the area consumption will also be large. In the proposed FDCT/IDCT design, the on-the-fly transposition architecture is used since it requires only $64n$ flip-flops instead of $128n$ flip-flops in the ping-pong case.

3.4. Chapter Summary

In this chapter, data-dependent Loeffler FDCT/IDCT algorithms are described. The zero-bypassing logic is inserted into fixed-complexity Loeffler's algorithm to convert it into a data-dependent algorithm, which the new design is based on. For FDCT, input truncation technique was also analyzed and applied to further reduce the amount of data to be processed, hence reduce the power consumption. Based on the simulation result, we decided to truncate one bit from the input of the second dimension FDCT.

The transpose memory architecture has also been studied. The on-the-fly transpose memory reported in [28] is chosen because it requires only half the amount of area comparing to the ping-pong architecture.

Since multiplier is the fundamental building block of FDCT/IDCT, in the next chapter, different multiplier architectures are analyzed based on low-power criteria.

Chapter 4

Multiplier Architectures

In VLSI implementation, floating-point multipliers are much larger, slower, and consume more power than fixed-point multipliers due to normalization of mantissa. For this reason, all FDCT/IDCT designs reviewed in this thesis used fixed-point multiplication instead of floating-point multiplication.

Since fixed-point or integer multipliers are larger, slower, and consume more power than adders, the choice of multiplier greatly affects the overall FDCT/IDCT performance and power consumption.

One special note about the multiplications performed in FDCT/IDCT is that they are all constant multiplications, i.e. one of the multiplicand is a constant. In Section 4.1, several constant multiplication schemes are studied, and the hardwired CSD multiplier is chosen for low-power design. Section 4.2 describes the design procedure of the hardwired CSD multipliers. In Section 4.3, synthesis is performed, and the result indicates that the CSD multipliers indeed consume less power than general-purpose multipliers.

4.1. Survey of Constant Multiplication Schemes

Following is a brief description of the characteristics of different constant multipliers. More detailed description can be found in the references.

4.1.1. Modified Booth Multiplier

Modified Booth multiplier [35] is a popular general-purpose multiplier. Both of its multiplicands are variables that can be changed at run-time. However, in DCT/IDCT multiplications, only one of the multiplicand is variable, the other one is a constant ($\cos(n\pi/16)$). Having both operands of multiplier variable implies more hardware, consequently more power. Thus, general-purpose modified Booth multiplier is not a good choice for low-power DCT/IDCT design.

4.1.2. Distributed Arithmetic (DA)

Distributed arithmetic (DA) is a bit-serial operation that performs shift-and-add operation to multiply two numbers (one of which is a constant). It replaces the multiplication with additions and a look up ROM table [14]. The input is used as index in the ROM, and the ROM contains the partial product of multiplying the address with the constant multiplicand, and the partial products are then added by using shift-and-add operations.

The main disadvantage of DA is that it is slow due to its bit-serial nature and parallel-serial/serial-parallel conversion. This implies that it needs higher internal clock frequency than parallel processing to do the same work. Moreover, shifting consumes much power because of the high switching activities. In [14] and [15], the authors evaluated the trade-off between the performance and the power for three multiplication schemes: general-purpose multiplier, pure ROM based, and mixed ROM based (DA).

Voltage (V)	Multiplier		Pure ROM		Mixed ROM	
	Delay (ns)	Power (mW)	Delay (ns)	Power (mW)	Delay (ns)	Power (mW)
5	101.56	13.49	86.5	30.4	79.5	30.1
4	132.27	7.64	111.4	18.6	103.7	18.1
3.3	162.53	4.99	137.1	10.5	129.1	11.0

Table 6: Comparison of general-purpose multiplication against ROM based multiplication [14]

As shown in Table 6, the multiplier-based implementation is slower than the DA-based implementations. However, the power is about 30-50% less than the DA-implementations because about 85% of the entire DA chip runs at higher frequency due to its bit-serial nature. As the result, DA is not a good choice for low-power design.

4.1.3. Hardwired Canonical-Sign-Digit (CSD) Wallace-Tree Multiplier

Hardwired multipliers hard code the constant multiplicand by using only shift-and-add operations. Unlike DA, which performs shift-and-add operation at run-time, these shifts are hard-wired at design time and consume no power. In other words, hardwired multipliers are simply Wallace-tree carry-save adders. This results in a smaller and more power-efficient multiplier than general-purpose multiplier.

Further power reduction can be achieved on the fixed multiplicand by not using 2's complement representation, but using radix-2 canonical sign-digit (CSD) representation. By definition, the *canonical* sign-digit representation is a redundant number system that represents number with *no adjacent non-zero digits*. Every number has a unique CSD representation [30]. It represents numbers with fewer or equal non-zero digits [4] as the algebraic sum/subtraction of several power-of-two, i.e.:

$$c = \sum s_k 2^{-k}, \text{ where } s_k \in \{-1,0,1\}$$

A procedure to transform a conventional binary number to CSD representation is described in [30]. We have also derived a more intuitive transformation algorithm:

Given a (n+1)-digit binary number **B** = $B_nB_{n-1}...B_1B_0$ with B_n=0 and $B_i \in \{0,1\}$ for $i \in [0,n-1]$. The following procedure converts **B** into the (n+1)-digit radix-2 canonical SD vector **D** = $D_nD_{n-1}...D_1D_0$ with $D_n \in \{0,1\}$ and $D_i \in \{0,1,-1\}$ for $i \in [0,n-1]$ such that both vector **D** and **B** represent the same value:

$$\alpha = \sum_{i=0}^{n} B_i 2^i = \sum_{i=0}^{n} D_i 2^i$$

1. If there are consecutive 1's in **B**, continue to step 2. Otherwise, the resulting number **B** is in CSD representation (**D**). End the process.
2. Replace the rightmost (starting from the lowest order 2^i end) occurrence of bit pattern $0\ \underbrace{1...1}_{(m-1)\text{ 1's}}\ 1$ with $1\ \underbrace{0...0}_{(m-1)\text{ 0's}}\ \bar{1}$. This replacement is possible because

$$\sum_{i=m}^{n} 2^i = 2^{n+1} - 2^m, \text{ where } n > m$$

3. Go back to step 1.

Figure 14 shows a step-by-step example that converts a binary number 0110010111_b (407_d in decimal) into CSD representation. The consecutive 1's to be replaced are shaded in the figure. The resulting CSD representation of 407_d is $10\bar{1}010\bar{1}00\bar{1}$, where $\bar{1}$ denotes –1. As expected, $407 = 2^9-2^7+2^5-2^3-2^0$. In this example, the CSD representation reduces the number of 1's from 6 down to 5.

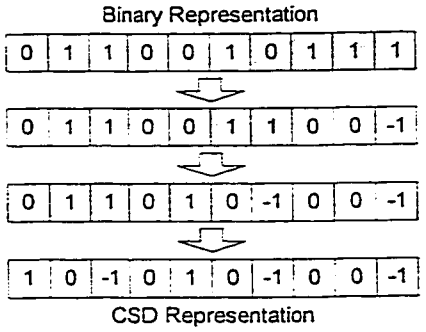


Figure 14: Converting binary number 0110010111 into CSD representation

As another example, Table 7 shows the CSD representations of the constant operands ($\cos(n\pi/16)$) used in FDCT/IDCT with 15-bit precision after binary point (total of 16 bits).

n	$\cos(n\pi/16)$				
	Traditional Binary Representation		**Canonical Sign-digit Representation**		
	Bit Pattern ($2^0,2^{-1},2^{-2},\ldots,2^{-15}$)	# Non-Zero Bits	Bit Pattern ($2^0,2^{-1},2^{-2},\ldots,2^{-15}$)	# Non-Zero Bits	% Bit Saving
1	0 1 1 1 1 1 0 1 1 0 0 0 1 0 1 0	9	1 0 0 0 0 0 -1 0 -1 0 0 0 1 0 1 0	5	44%
2	0 1 1 1 0 1 1 0 0 1 0 0 0 0 1 0	7	1 0 0 0 -1 0 -1 0 0 1 0 0 0 0 1 0	5	29%
3	0 1 1 0 1 0 1 0 0 1 1 0 1 1 1 0	9	1 0 -1 0 1 0 1 0 1 0 0 -1 0 0 -1 0	7	22%
4	0 1 0 1 1 0 1 0 1 0 0 0 0 0 1 0	6	1 0 -1 0 -1 0 1 0 1 0 0 0 0 0 1 0	6	0%
5	0 1 0 0 0 1 1 1 0 0 0 1 1 1 0 1	8	0 1 0 0 1 0 0 -1 0 0 1 0 0 -1 0 1	6	25%
6	0 0 1 1 0 0 0 0 1 1 1 1 1 1 0 0	8	0 1 0 -1 0 0 0 1 0 0 0 0 0 -1 0 0	4	50%
7	0 0 0 1 1 0 0 0 1 1 1 1 1 0 0 1	8	0 0 1 0 -1 0 0 1 0 0 0 0 -1 0 0 1	5	38%

Table 7: Canonical sign-digit representation of $\cos(n\pi/16)$

As shown in Table 7, the CSD representation can reduce the number of non-zero bits up to 50% over traditional representation. In hardwired-multiplier, each non-zero digit (except the first 3 non-zero digits) in the constant multiplicand requires one extra carry-save adder stage.

Because canonical means no adjacent non-zero digits, any n-bit number can be represented with at most $\lceil n/2 \rceil$ number of non-zero digits, which in turn reduces at least half of the carry-save adder stages comparing to general purpose array multiplier. It can also be shown that CSD generates an average of $n/3$ additions [40]. Since fewer non-zero

bits imply less computation, less switching activity, and less power consumption, the hardwired CSD multiplier is a good choice for low-power design.

4.1.4. Pattern-Based CSD Multiplier

The CSD representation uses minimum shift-and-add (S&A) operations when multiplying constant k with variable x directly. However, direct multiplication of $x \times k$ does not necessarily use minimum S&A operations to perform $x \times k$. In some situations, it is possible to find patterns inside the CSD representation, which can be reused to avoid repeated computation. Thus, instead multiplying x with k directly, x is multiplied with sub-expressions of k, then partial products are used to construct the final product. As an example, let $k = 11100111 = 100\bar{1}0100\bar{1}$ (231_d). Using CSD representation without pattern searching, $231x$ requires 4 additions. However, with pattern-based algorithm, $231x$ can be represented by $(7x<<5)+7x$, which requires 3 additions only. The Bernstein's algorithm [41], Lefèvre's algorithms [39-40], and Potkonjack algorithm [42] are pattern-based algorithms.

The pattern-based algorithms are very useful for multiplication with very large constants where the patterns can be reused frequently. For example, in encryption/decryption, the constant may have several hundreds or thousands of bits. In such situation, pattern-based algorithm can reduces the computation significantly. However, for the purpose of FDCT/IDCT and most DSP applications, the constants word lengths are usually small, and patterns (if any) are reused less frequently.

For pattern reuse, one must obtain the entire partial product, which requires using carry-save-adder (CSA) followed by carry-propagate adder (CPA). In general, in VLSI

implementation, CPA is slower, and consumes more power than CSA due to carry propagation. The slower pattern-based algorithm speed can be compensated by adding pipeline registers after each CSA used for partial product (pattern) computation. The extra power consumption due to the carry propagation in CPA can be reduced by using other types of adders such as carry-bypass adders or carry-select adders. However, given the patterns are not reused frequently, the overall power consumption of pattern-based multiplier is still larger than the one without using pattern. Since the design criterion of this thesis is power, only the CSD multiplication without using pattern is considered, and all multipliers used in FDC/IDCT are hardwired CSD multipliers.

Notice that the application of hardwired CSD Wallace-tree multiplier is not restricted to FDCT/IDCT only. It can be used in many other digital signal processing (DSP) applications, such as digital filters, where fixed-coefficient multiplication is required.

4.2. CSD Multiplier Implementation Procedure

To design a hardwired CSD multiplier for multiplying unsigned variable integer operand (v) with a constant operand, we derived the following steps:

1. Obtain the CSD representation of the constant operand by using the algorithm described in Section 4.1.3.
2. For each non-zero bit position p in constant operand:
   - For each 1 in the constant operand, place the unsigned variable operand, i.e. performing $v \times 2^p$.

- For each –1 in the constant operand, negate the unsigned variable operand with a 1 placed at the least-significant bit (2's complement), and extend 1's to the left of the most-significant bit (sign extension) of the variable operand, i.e. performing $(-v)\times 2^{p}$

3. Simplify the diagram by adding the constant 1's together to avoid redundant computation at run time. By studying the truth-table of addition, we found that further optimization can be achieved by using identity I.1:

   Identity I.1: Variable bit *b* plus constant 1 results in sum *~b* and carry *b*, where *~b* denotes *NOT* operation

b	b+1	
	Sum	Carry
0	1	0
1	0	1
Sum=b, Carry=~b		

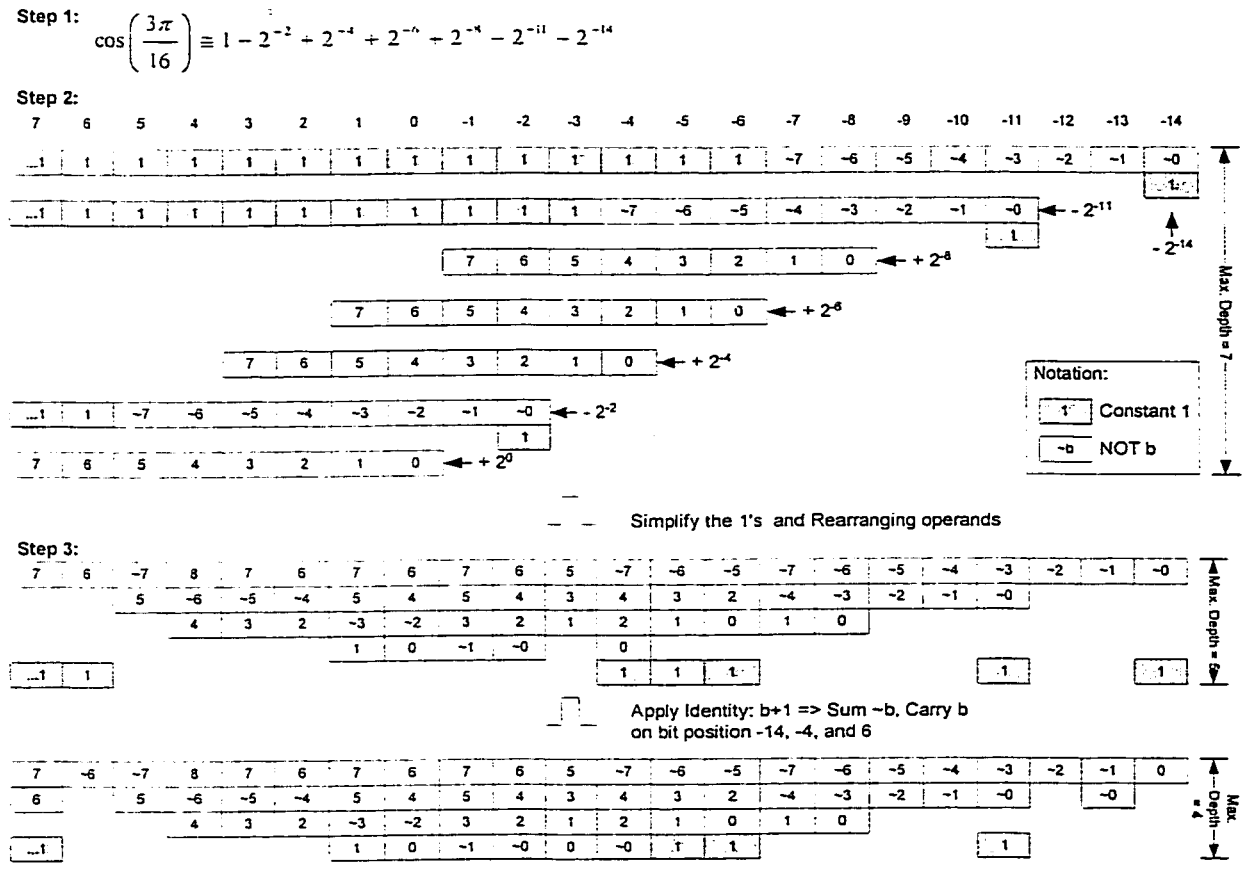
Table 8: Truth-table of b+1

   This identity allows reduction of one operand to be added for position p by increasing the number of operands to be added for position p+1 by 1. Intelligent use of this identity can reduce the number of carry-save adder (CSA) stages (critical path delay) without introducing any extra hardware.

4. Combine the operands placed in step 2 and the simplified constant 1's (in step 3) with carry-save adders in Wallace-tree form. The result of the carry-propagate adder is the result of multiplying variable input operand with the constant operand.

To illustrate the above algorithm, Figure 15 shows the procedure of constructing a CSD hardwired-multiplier of constant cos(3π/16) multiplying with an 8-bit unsigned

integer. Constant cos(3π/16) is chosen because it contains the most non-zero bits in the Table 7. As shown in Figure 15, in step 3, the application of identity I.1 reduces the depth of CSA tree from 7 down to 4. As the result, the multiplication of cos(3π/16) with an 8-bit unsigned number has critical path of only 2 full-adder stages with a 19-bit CPA adder. Notice that despite the fact that the multiplier uses CSD representation for the constant operand, both the variable operand and the product are still in 2's complement representation.



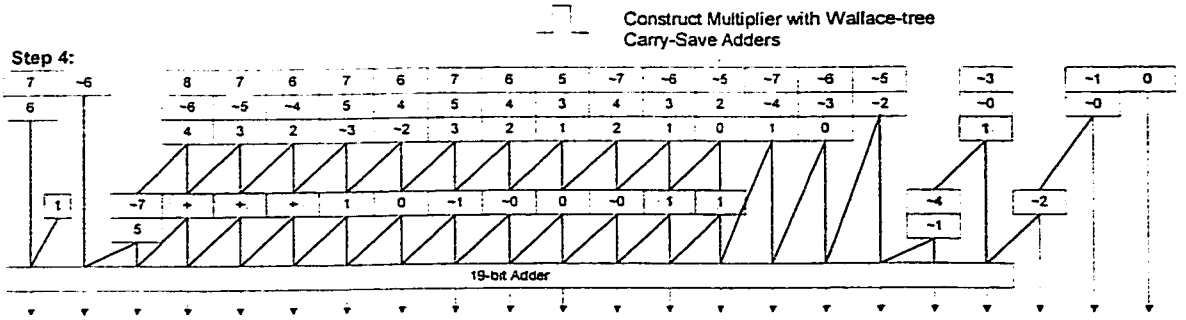


Figure 15: Hardwired CSD multiplier for multiplying cos(3π/16) with 8-bit unsigned integer

Similarly, to multiply a signed 2's complement variable operand (v) having a sign-bit (s) with a constant operand, the following procedure is derived:

1. Obtain the CSD representation of the constant operand.
2. For each non-zero bit position p in constant operand:
   - For each 1 in the constant operand, place the signed variable operand, i.e. performing $v \times 2^p$. Sign-extend towards left.
   - For each –1 in the constant operand, negate the signed variable operand v with a 1 placed at the least-significant bit (2's complement), and extend $\sim s$ (negated sign-bit s) to the left of the most-significant bit (sign extension) of the variable operand, i.e. performing $(-v) \times 2^p$
3. Simplify the sign extension bits and constant 1's in the diagram:
   - Let s=0, replace all s with 0, and $\sim s$ with 1, add all constant 1's together, and obtain a constant value SE_0.
   - Le s=1, replace all s with 1, and $\sim s$ with 0, add all constant 1's together, and obtain a constant value SE_1.

- For each bit at position p, merge SE_0 and SE_1 together to obtain another value SE using the following truth table:

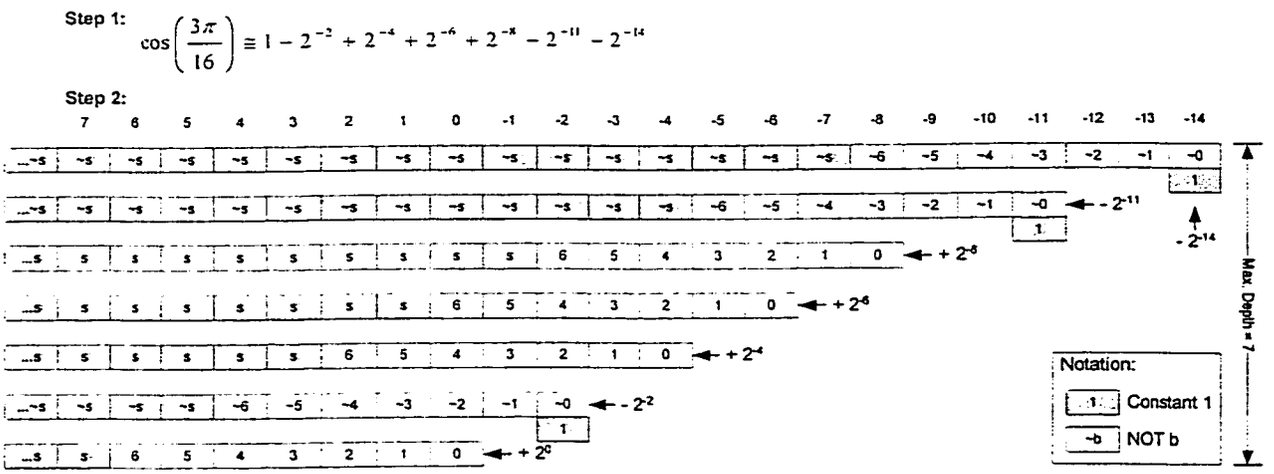
SE_0 (s=0)	SE_1 (s=1)	SE
0	0	0
0	1	s
1	0	$\sim s$
1	1	1

Table 9: Truth table to simplify sign-extension

- Remove all sign extension bit (s or $\sim s$), insert SE into the diagram.
- Like the unsigned case, apply identity I.1 where suitable.

4. Combine the operands placed in step 2 and the simplified sign-extension bits and constant 1's (in step 3) with carry-save adders in Wallace tree form. The result of the carry-propagate adder is the result of multiplying variable signed 2's complement input operand with the constant operand.

Like the unsigned case, step-by-step illustration of construction of a CSD hardwired-multiplier of constant operand cos(3π/16) multiplying with an 8-bit signed 2's complement integer is shown in Figure 16.



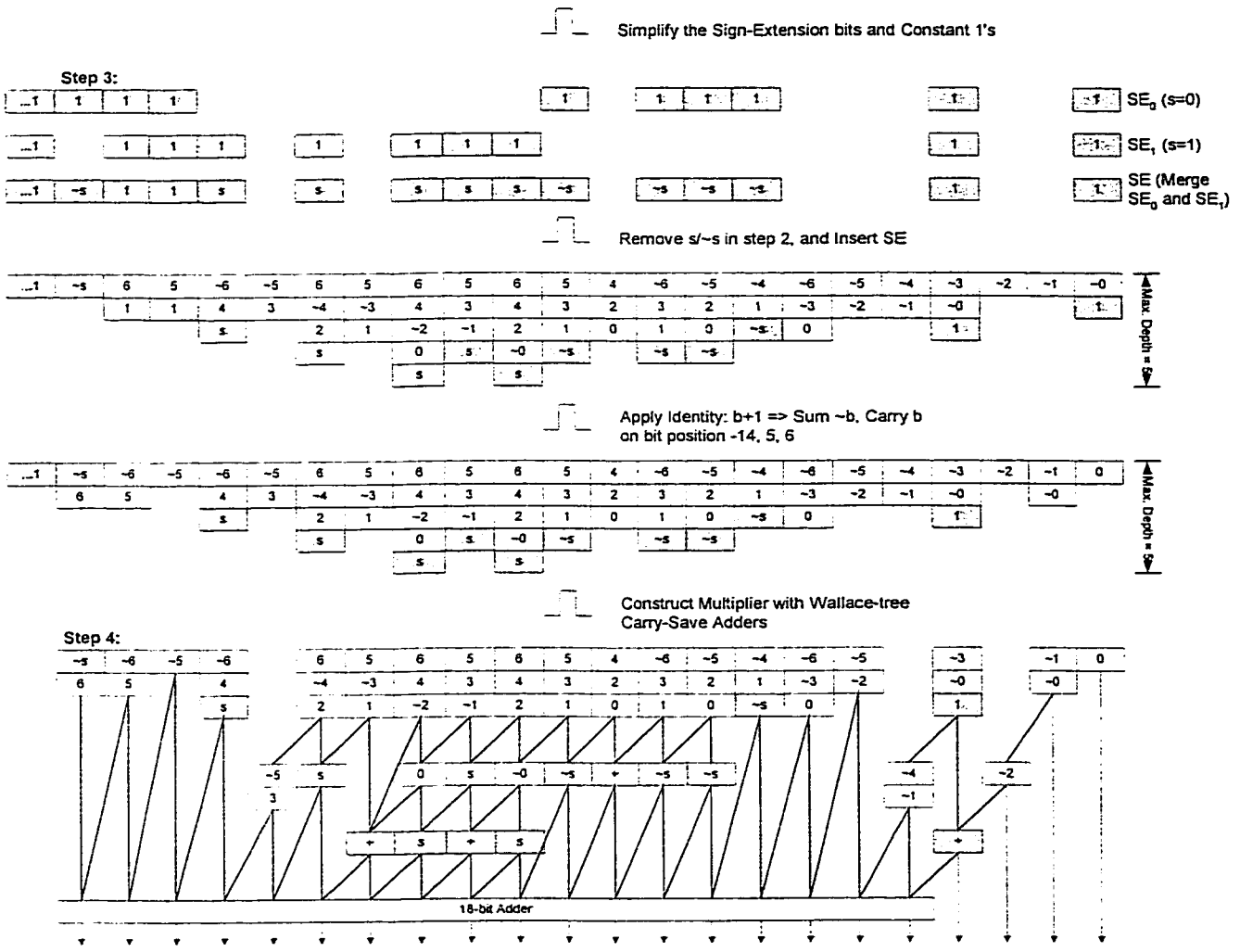


Figure 16: Hardwired CSD multiplier for multiplying cos(3π/16) with 8-bit signed integer

4.3. Multiplier Synthesis Result

To demonstrate that the hardwired CSD Wallace-tree constant multiplier consumes less power and area while offering comparable speed performance, its delay,

area, and power consumption figures are compared with the with other 32-bit popular general-purpose multipliers.

Since the hardwired CSD multiplier has one operand constant, several CSD multipliers are implemented with different constant operand used in FDCT/IDCT ($\cos(n\pi/16)$, and $2^{\pm 1/2}$). All constants have 1-bit integer part and 31-bit fraction part to form a 32-bit fixed-point number. The constants are then multiplied with a 32-bit signed integer (variable operand). All multipliers are synthesized using Xilinx 4052XL-1 FPGA technology.

32-bit Multipliers	Array Multiplier	Modified Booth Multiplier	Wallace Tree Multiplier	Modified Booth-Wallace Tree Multiplier	Proposed Scheme
Area (CLB)	1165	1292	1659	1239	493
Delay (ns)	187.87	139.41	101.14	101.43	106.21
Power (mW)	16.651	23.14	30.95	30.86	7.67

Table 10: Comparison of 32-bit CSD Wallace-tree multiplier with 4 different general-purpose multipliers using Xilinx 4052XL-1 FPGA technology (Columns 1-5 adopted from Table 1 in [36])

As shown in Table 10, the CSD multiplier uses least amount of area and power (less than half of the power than the array multiplier) while offering comparable speed performance with the other multipliers (around 100 ns). This result agrees with the analysis – hardwired CSD is more power efficient then other general-purpose multipliers. Therefore, hardwired CSD Wallace-tree multipliers are used in the FDCT/IDCT designs presented in this thesis.

4.4. Chapter Summary

In this chapter, by analyzing different constant multiplication schemes, a new constant-coefficient multiplier design is presented. The multiplier is based on canonical sign-digit representation with Wallace-tree formation. As shown in the analysis and simulation, the CSD multiplier is both more power and area efficient than general-purpose multiplier while offering similar speed performance. Consequently, it is used in the FDCT/IDCT design presented in this work. Detailed design procedures for both unsigned and signed integer are also described.

In the next chapter, more implementation details, such as design automation and pipeline design, are presented.

Chapter 5

Implementation

Since the main efforts are concentrated on the arithmetic level (data-dependent algorithm) and implementation level (hardwired CSD multipliers), we decide to use VHDL to implement the FDCT/IDCT designs. No optimization on the circuit level or technology level is made.

To ensure error-free coding, some design automation effort is made. In Section 5.1, a C++ program that generates VHDL code of hardwired CSD Wallace-tree multiplier is developed. Similarly, to make the IDCT design compliant to IEEE Std. 1180-1990, in Section 5.2, a Java program is developed that calculates the error figures defined in IEEE standard [27] for different internal bandwidths. The pipeline designs for both the FDCT and IDCT are also described in this Chapter (Section 5.3).

5.1. Hardwired CSD Multiplier Generator

Since the FDCT/IDCT design uses hardwired CSD multiplier, for each constant operand and bandwidth of variable operand, different multipliers are required. To save the design time and avoid bugs in the coding, it is ideal to generate constant multipliers through a code generator.

Several constant multipliers generators [40][43-44] have been reported in the literature. All of them are optimized for Xilinx FPGA 4000 and Virtex technologies. To

have a technology-independent constant multiplier generator, a C++ program that generates VHDL code for hardwired CSD multiplier is developed. The program is called *constant multiplier generator* (CMG). The C++ source code of the generator is listed in Appendix C and in attached CD.

The CMG is capable of generating VHDL code that multiplies signed/unsigned variable operand with any positive integer constant multiplicand. The constant operand can have the size of `long` type in C++ language. The CMG takes the following information from the user:

- VHDL entity name.
- Integer value of the constant operand: For real number constant operand, use the integer value of the corresponding fixed-point representation. For Intel Pentium® processors running Microsoft Windows® 32, the limitation of the constant operand is from 0 to 2147483647.
- Variable operand: Number of bit of the signed/unsigned variable operand.
- Product Least-Significant-Bit Truncation: This feature is useful for real number (fixed-point) multiplications. In many situations, not all bits in the real part are required. Truncating some least-significant bits from the product results in a smaller, faster, and more power-efficient multiplier. The truncation error has been analyzed in [45].

The generator uses the algorithm described in Section 4.2 to generate VHDL code. At the end of the code generation, it also reports critical statistical information: number of carry-save adder stages, number of inverters, half adders, and full adders. This information is useful for power, area, speed, and pipelining analysis.

As an example, for constant operand $\cos(3\pi/16)$ with 15-bit precision multiplied with 12-bit variable operand and no truncation, the CMG generates the VHDL code shown in Appendix B.

5.2. IEEE Standard 1180-1990 IDCT Compliant

To ensure the proposed IDCT chip conforms to IEEE Standard 1180-1990, a Java program is developed. The program reads in data path bandwidths, multiplier precisions, and truncation patterns used in Loeffler's IDCT in each pipeline stage from a file, and calculates the error figures (*ppe*, *pmse*, *omse*, *pme* and *ome*) defined in the standard (see Section 2.3.). Again, the source code is listed in Appendix D and in the attached CD. Notice that Java is chosen as the programming language because the `long` type in Java is a 64-bit integer, which is more suitable for simulating fixed-point arithmetic. In C++, the size of data type is machine dependent; while in Java, the size of data type is machine independent and fixed.

After testing different combinations of internal bandwidths, the first dimension IDCT produces 13-bit integer/5-bit precision fixed-point output. The second dimension IDCT produces 14-bit signed integer output (after rounding of 10-bit precision result). The 2-D IDCT presented in this thesis conforms to the IEEE 1180-1990 Standard. The compliance test results are shown in Table 11.

Random Data Range	pme <0.015	pmse <0.06	ome <0.0015	omse <0.02
[-300,300]	0.0121	0.0161	0.00082	0.0108
[-256,255]	0.0129	0.0144	0.00073	0.0103
[-5,5]	0	0	0	0
-[-300,300]	0.0121	0.0155	0.00081	0.0109
-[-256,255]	0.0143	0.0163	0.00085	0.0104
-[-5,5]	0	0	0	0
Zero-in zero-out test passed. |ppe|≤1				

Table 11: IEEE Standard 1180-1990 Compliance for Proposed IDCT

5.3. Pipelining Design

Since the hardwired CSD multiplier is essentially a carry-save adder, and the speed of the carry-save adder is mostly limited to the carry-propagate adder, the speed of FDCT/IDCT is directly related to the carry-propagate adders. Thus, it is logical to insert pipelining registers after each adder (including the adders in the multipliers). As shown in Figure 17, for *kcn* blocks in IDCT, there are 3 pipeline stages (add inputs, multiply, and add product). For *kcn* blocks in FDCT, there are 4 stages. The extra stage is required to add the partial products of the segmented multiplications. Therefore, there are 10 pipeline delays (latency) for 1-D FDCT, and 8 pipeline delays for 1-D IDCT (see Table 12).

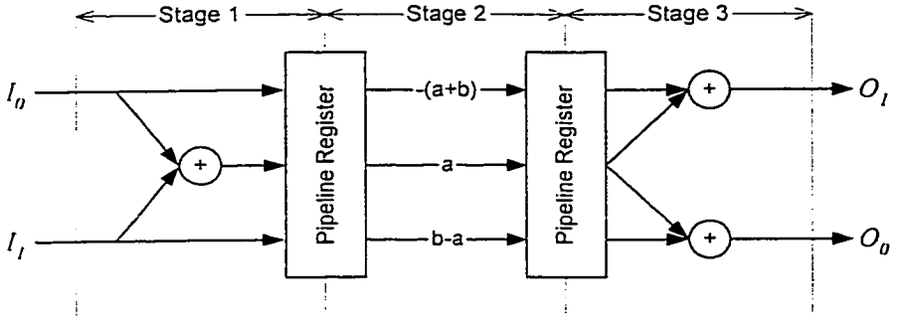


Figure 17: Pipelined *kcn* block

Latency	Stage 1	Stage 2	Stage 3	Stage 4	Total
FDCT	1	4	4	1	10
IDCT	1	3	3	1	8

Table 12: Latencies for 1-D FDCT and 1-D IDCT

For the transpose memory, the on-the-fly transpose memory architecture is used. From Figure 12, it is clear that the latency is 8 clock cycles because the transposed output can be obtained starting from the 9^{th} clock cycle.

To summarize, the proposed 2-D FDCT has latency of 28 clock cycles, and the 2-D IDCT has latency of 24 clock cycles (see Table 13).

Latency	First Dimension	Transpose Memory	Second Dimension	Total
FDCT	10	8	10	28
IDCT	8	8	8	24

Table 13: Latencies for 2-D FDCT and 2-D IDCT

5.4. Chapter Summary

In this chapter, a new constant CSD multiplier generator is introduced. Written in C++, the program generates VHDL code that multiplies constant integer operand with signed/unsigned variable operand. Truncation can also be made on the product to reduce hardware, power, and delay.

A Java program is developed to select the internal bandwidth such that the 2-D 8x8 IDCT conforms to the IEEE Standard 1180-1990.

Both FDCT and IDCT designs have also been pipelined to achieve throughput of 1 output/clock cycle. The latency is 28 clock cycles for FDCT, and 24 clock cycles for IDCT.

In the next chapter, the VHDL code of the proposed FDCT/IDCT chip is synthesized using Synopsis with Canadian Microelectronic Corporation (CMC) 3-volt 0.35-μm technology. Synthesis results (power/area/delay) are compared with previous works.

Chapter 6

Synthesis Results

In this chapter, synthesis results of proposed FDCT/IDCT are presented in Section 6.1. The proposed design is compared with previous reported designs in Section 6.2. using the switching-capacitance per sample criteria described in 1.3.

6.1. Synthesis Results of the Proposed Design

The VHDL code of the proposed FDCT/IDCT core is synthesized using Synopsis with Canadian Microelectronic Corporation (CMC) 3-volt 0.35-μm technology. Since the design goal is low power, the compiler constraint is set to minimize the dynamic power consumption (ideally zero). The synthesis result indicates that the proposed FDCT core consumes 122.7mW at 40MHz, and IDCT core consumes 124.9mW at 40MHz. The detailed specifications of the new FDCT/IDCT design are shown in Table 14.

Only the dynamic power reported by the Synopsis is compared with other designs in the next section. In real life, there may be other power consumptions, such as leakage power and short-circuit power. Since the leakage power is related to the fabrication, which is not the concern of this paper, it is ignored in the comparison. As for the short-circuit power, it is assumed to be small and negligible, which is usually the case in practice. Its effect can be minimized with proper timing design.

The power measurements are performed under the worst-case condition where the assumed statistical properties do not hold, i.e. under white noise input. In this situation, most of the bypassing logics are not active, and the power consumption is higher. This simulation condition is chosen because in real life, for MPEG-2 video compression, the assumed statistical properties apply only for I-frames, but less so for B-frames and P-frames. For those frames, the redundancies at the input are already been reduced, and the input behaves like white noise.

	FDCT	IDCT
Process Technology	CMC 0.35μm CMOSP technology	
Supply Voltage (V)	3 Volts	
Operating Frequency (MHz)	40 M	
Processing Rate (samples/sec)	320 M	
Dynamic Power (mW)	122.6666	124.8587
Leakage Power (nW)	16.8610	18.6860
Area (reported by Synopsis)	3.2548425	3.2969125
Maximum Pipeline Stage Delay (ns)	24.51	24.43
Latency (clock cycles)	28	24
Input/Output Numeric System	2's complement signed integer	
Input Specification	8 input/clock cycle	
Input Bandwidth (for each input)	9-bit	12-bit
Throughput	8 output/clock cycle	
Output Bandwidth (for each output)	17-bit	14-bit

Table 14: Process and Specifications of the proposed FDCT/IDCT designs

6.2. Comparison with past FDCT/IDCT VLSI implementations

Many FDCT/IDCT VLSI implementations have been reported in the literature. The specifications of several recent high-performance FDCT/IDCT chips are summarized in Table 15. Due to different process technologies (supply voltage, operating frequency, etc.), implementation approach (full-custom, semi-custom, etc.), optimization parameters

(RTL, transistor level, layout level, etc.), and design algorithm/architectures, comparing different implantations is always a tough job in VLSI design. Also, in some situations, not all measurement figures are reported. As the result, it is very difficult to compare one design with another accurately.

Implementation	Process	Area (mm^2) / Transistors	Supply Voltage (V)	Power (mW)	Clock Rate
Toshiba 1994 FDCT/IDCT [23][24]	0.6 μm, 2ML	13.33mm^2/ 120K	0.35W at 3.3V, 200MHz 0.15W at 2V, 100MHz		
Toshiba 1996 FDCT/IDCT [22]	0.3 μm 2ML, Triple well	4mm^2/ 120K	0.9V, VT=0.15 ±0.1V	10 mW	150 MHz
AT&T IDCT [37]	0.5 μm	7mm^2/ 69K	3V	250mW	58 MHz
Xanthopoulos's IDCT [28]	0.5 μm, 3ML	20.7mm^2/ 160K	1.1-1.9V VTN/VTP=0.66/-0.92V TOX=9.6nm	4.65mW at 1.32V, 14MHz	5-43 MHz
Xanthopoulos's FDCT [28]	0.6 μm, 3ML	20.7mm^2/ 160K	1.1-3V VTN/VTP=0.75/-0.82 V TOX=14.8nm	4.38mW at 1.56V, 14MHz	2-43 MHz
Sarmiento's FDCT [6]	0.6 μm, E/D-MESFET GaAs	32.2mm^2/ 51K	2V	7W	600 MHz

Table 15: Summary of specifications of several FDCT/IDCT chips

In order to compare the proposed design with other works fairly, like [19]-[21], the switching capacitance per sample (hence power per sample) is calculated and compared. It can be used as an indication of energy efficiency since it is directly proportional to power consumption required to process each input sample.

As described in Section 1.3, the switching capacitance of each design is obtained by dividing the power with the frequency and squared voltage. Notice that the switch capacitance per sample is obtained by dividing the switching capacitance by the number of samples per clock cycle.

Technology scaling is also performed for all designs to normalize all designs to 0.35μm technology. The scaling factor from 0.35μm (CMC 0.35μm CMOSP) technology to 0.5μm (0.6μm drawn) (CMC CMOSIS5) technology is obtained by performing HSPICE simulations on two inverters, one as the load of another. For both technologies, the power supply is 3 volts with 40-MHz 3-volt square pulse input. The PMOSs have size L=W_{min} with W=4W_{min}, and the NMOSs have size L= W_{min} with W=2W_{min}, where W_{min} is the minimum feature size of the corresponding technology. The simulation result indicates that the power consumption is 0.634 mW for 0.35μm technology, and 1.19 mW for 0.5μm technology. Since both circuits are operating on the same voltage and frequency, the ratio between the powers is the ratio between the switching capacitances. For simplicity, 0.5μm and 0.6μm technologies are treated equally, similarly for 0.3μm and 0.35μm technologies. Thus, the switching capacitance in 0.5μm (and 0.6μm) technology will be multiplied with 0.532 to scale to 0.35μm technology. The effect of circuit level optimization, such as variable threshold voltage used in [22], is ignored since it cannot be quantified correctly.

The switching capacitance per sample is shown in Table 16 after technology normalization. As an example, the switching capacitance per sample of the proposed FDCT design is calculated as $\frac{122.6666 \cdot 10^{-3}}{320 \cdot 10^{6} \cdot 3^{2}} = 42.6\,\text{pF}$. For the Xanthopoulos's FDCT, which is a 0.5μm design, technology scaling is performed, and the switching capacitance per sample is calculated as $\frac{4.65 \cdot 10^{-3} \cdot 0.532}{14 \cdot 10^{6} \cdot 1.32^{2}} = 101.6\,\text{pF}$, where 0.532 is the technology scaling factor to scale the power of a 0.5μm technology down to 0.35μm technology.

As shown in Table 16, the proposed data-dependent FDCT/IDCT designs have the least switching capacitance per sample, i.e. consume least amount of power to process each input data sample. Thus, the proposed FDCT/IDCT design is the most power efficient one among the designs reviewed in this thesis.

Implementation	Switching Capacitance / Sample (pF)
Toshiba 1994 FDCT/IDCT [23][24]	85.6 (3.3V design) 199.8 (2V design)
Toshiba 1996 FDCT/IDCT [22]	82.3
AT&T IDCT [37]	478.9
Xanthopoulos's FDCT [28]	68.5
Xanthopoulos's IDCT [28]	101.6
Sarmiento's FDCT [6]	1553.9
Proposed FDCT Design	42.6
Proposed IDCT Design	43.4

Table 16: Energy Efficiency (Switching Capacitance/Sample in 0.35μm technology)

6.3. Chapter Summary

In this chapter, the proposed FDCT/IDCT design is synthesized using Synopsis with CMC 3-volt 0.35-μm technology. To compare the proposed design with previous works, the switching capacitance per sample is used. This comparison method permits technology-independent comparison of different DCT/IDCT architectures. From Table 16, it has been show that the new FDCT/IDCT designs have the smallest switching capacitance per sample, and are the most power-efficient designs.

Chapter 7

Conclusion

7.1. Summary of Research

In this work, a data-dependent low-power FDCT/IDCT design is presented. Low power is achieved by performing optimizations on both algorithm and architectural levels.

Both the FDCT and IDCT designs are built based on low-complexity Loeffler's fast algorithm combined with data-dependent zero-bypassing logic. In FDCT, to have high zero-bypassing probability, segmented multiplication is used. Also, to reduce the internal bandwidth, hence the amount of data to be processed, least-significant-bits truncation technique has also been employed. The error introduced by truncation is empirically studied.

The multiplier architecture is optimized by developing low-power CSD multipliers. To reduce the possibility of bugs in coding, a C++ program that generates the technology-independent VHDL code for the multiplier is developed. This generator can be used in many other DSP applications where constant multiplication is required.

The FDCT/IDCT designs are coded using VHDL, and synthesized using Synopsis 1998 with CMC 0.35μm CMOSP technology. No transistor-level circuit optimization is done. Operating at 3V and 40MHz, the FDCT design consumes 122.7mW, while the IDCT design consumes 124.9mW. By comparing with other recent works, the proposed

FDCT/IDCT designs are the most power-efficient ones since they have the least switching capacitance per sample. Low-power operation is achieved through the selection of low-complexity Loeffler's algorithm, data-dependent zero-bypassing logics, and least-significant-bits truncation.

7.2. Conclusion

From the analysis and simulation results, the following conclusion can be made about this thesis:

- Data-dependent algorithm can reduce the number of operation when bypassing logics are properly inserted. Improper use of the data-dependent algorithm may lead to increasing the computation rather than decreasing the computation.
- Hardwired CSD Wallace-tree multiplier is a good choice for low-power design where constant multiplication is required. Its application is not only limited to DCT/IDCT. In many non-adaptive signal processing/filter applications, constant multiplications are required. The use of hardwired CSD multiplier can lead to a more power-efficient design.
- Low-power design can be achieved by having optimization at both design time and run time. The design time optimization is done by carefully choosing a good algorithm that reduces the number of operations. The run-time optimization is achieved by using data-dependent bypassing logics to reduce the switching activity, which is directly proportional to the power consumption.

- The data-dependent low-power design approach is not only limited to DCT/IDCT. It can be used in other applications as well where the statistical property of the input is well understood.

7.3. Possible Improvements for Future Research

Following are some recommendations and possible improvements for future research endeavors.

- *Study the effect of integrating data-dependent algorithm with other fast algorithms*: This thesis is based on Loeffler's fast algorithm. It is chosen because it has the least amount of multiplication over the surveyed papers. It is interesting to know the effect of applying bypassing logic onto other fast algorithms to determine the potential of data-dependent algorithm.
- *Study the effect of segmented multiplication*: As discussed in 3.1.1, smaller segmentation size leads to higher bypassing probability with the expense of more complicated control logic and more delay. In this work, the multiplications in FDCT are spilt into two segments. This choice may not be optimum. Having different segmentation strategy may lead to a more power-efficient design.
- *Study the truncation effect for P-frames and B-frames*: The truncation simulation is performed for I-frame only. It is a good idea to measure the truncation effect on P- and B-frames as well.
- *Explore the possibility of using truncation as a mean of quantization*: Truncation behaves like quantization since both operations reduce numerical precision. Thus,

instead of having 2-D FDCT and quantizer as two separate blocks, it could be possible to merge them together. In such a situation, a sophisticated control algorithm is necessary for adapting the FDCT for different quantization levels (Q-factors).

- *More power simulations under different conditions*: The power measurements presented in this work are performed under the worst-case condition where the assumed statistical properties do not hold, i.e. under white noise input. In order to get more accurate power estimation, it is recommended to pass many different real sequences (with I-, B-, and P-frames) as the input of the system, and measure the power consumptions.
- *Improve the Constant Multiplier Generator (CMG)*: Several possible improvements can be made on the CMG:
  1. *Negative constant support*: Currently, the CMG supports only multiplication with non-negative integers. In this work, the negative constant coefficients of DCT/IDCT are taken care by using subtractions instead of additions when the products are used. However, for other applications, if negative constant multiplication is required, the CMG can easily be modified to support multiplying negative integer constants.
  2. *Carry-save-adder optimization*: In some situations, there are common operands to be added in the carry-save adder array for different bit positions. It is possible to share the partial sum of the full/half adders. Unlike the pattern-based algorithm that requires full summation, sharing carry-save-adder reduces the hardware and power without increasing the delay. The only

drawback of doing so is that the overall design becomes highly irregular due to complex routing cause by sharing wires.

3. *Better full-addition support*: Currently, at the end of the CSA, CPA is used. It is possible to reduce the power consumption even further by using carry-bypass adder or carry-select adder.
4. *Support for pattern-based CSD algorithms*: As mentioned before, the CMG is designed for DSP applications where the constants are assumed to be small. However, if the constants are large, pattern-based algorithms should reduce the computation significantly, thus reducing the power.

Bibliography

[1] W. H. Chen, C. H. Smith, and S. C. Fralick, "A Fast Computational Algorithm for the Discrete Cosine Transform", IEEE Trans. on Communications, vol. Com-25, no. 9, pp. 1004-1009, September 1977

[2] S. I. Uramoto, Y. Inoue, A. Takabatake, J. Takeda, Y. Yamashita, H. Terane, and M. Yoshimoto, "A 100-MHz 2-D discrete cosine transform core processor", IEEE J. of solid-state circuits, vol. 27, no. 4, pp. 492-499, April 1992.

[3] Y. F. Jang, J. N. Kao, J. S. Yang, and P. C. Huang, "A 0.8μ 100-MHz 2-D DCT core processor", IEEE trans. on consumer electronics, vol. 40, no. 3, pp. 703-709, August 1994.

[4] A. Madisetti and A. N. Willson, "A 100 MHz 2-D 8x8 DCT/IDCT Processor for HDTV Applications", *IEEE. Tran. on Circuits and Systems for Video Tech.*, vol. 5, No. 2, pp. 158-164, April 1995.

[5] T. Masaki, Y. Morimoto, T. Onoye, and I. Shirakawa, "VLSI Implementation of Inverse Discrete Cosine Transform and Motion Compensator for MPEG2 HDTV Video Decoding", *IEEE Tran. on Circuits and Systems for Video Tech.*, vol. 5, No. 5, pp. 387-395, October 1995.

[6] R. Sarmiento, C. Pulido, F. Tobajas, V. Armas, R. E. Chaín, J. López, J. M. Nelson, and A. Núñez, "A 600 MHz 2-D DCT processor for MPEG application", Conference Record of the 31st Asilomar Conference on Signals, Systems & Computers 1997, vol. 2, pp. 1527 –1531, 1998

[7] M. T. Sun, T. C. Chen, and A. M. Gottlieb, "VLSI Implementation of a 16x16 discrete cosine transform", IEEE transaction on circuits and systems, vol. 36, no. 4, pp. 610-617, April 1989

[8] W. Li, "A new algorithm to compute the DCT and its inverse", IEEE trans. On signal processing, vol. 39, no. 6, pp. 1305-1313, June 1991

[9] D. Slawecki and W. Lee, "DCT/IDCT Processor Design for High Data Rate Image Coding", *IEEE Tran. on Circuits and Systems for Video Tech.*, vol. 2, No. 2, pp. 135-146, June 1992.

[10] C. Loeffler, A. Lightenberg, and G. S. Moschytz, "Practical fast 1-D DCT algorithms with 11-multiplications", ICASSP-89, vol. 2, pp. 988 –991, 1989

[11] B. G. Lee, "A new algorithm to compute the discrete cosine transform", IEEE trans. on acoustics, speech, and signal processing, vol. ASSP-32, no. 6, pp. 1243-1245, December 1984

[12] H. S. Hou, "A fast recursive algorithm for computing the discrete cosine transform", IEEE trans. on acoustics, speech, and signal processing, vol. ASSP-35, no. 10, pp. 1455-1461, October 1987

[13] Y. Jeong, I. Lee, H. S. Kim, and K. T. Park, "Fast DCT algorithm with fewer multiplication stages", *Electronic Letters*, vol. 34, No. 8, pp. 723-724, April 1998.

[14] E. N. Farag and M. I. Elmasry, "Low-power implementation of discrete cosine transform", Sixth Great Lakes Symposium on Proceedings VLSI, pp. 174 –177, 1996

[15] M. Kuhlmann and K. Parhi, "Power comparison of flow-graph and distributed arithmetic based DCT architectures", Conference Record of the 32nd Asilomar Conference on Signals, Systems & Computers, 1998, vol.2 , pp. 1214 –1219, 1998

[16] C. V. Schimpfle, P. Reider, and J. A. Nossek, "A power efficient implementation of the discrete cosine transform", Conference Record of the 31st Asilomar Conference on Signals, Systems & Computers, 1997, vol. 1, pp. 729 –733, 1998

[17] S. Masupe and T. Arslan, "Low power DCT implementation approach for VLSI DSP processors", ISCAS '99, vol. 1, pp. 149 -152, 1999

[18] S. Masupe and T. Arslan, "Low power DCT implementation approach for CMOS-based DSP processors", Electronics Letters, vol. 34 25, pp. 2392 –2394, Dec. 1998

[19] T. Xanthopoulos, and A. Chandrakasan, "A low-power DCT core using adaptive bitwidth and arithmetic activity exploiting signal correlations and quantization", Digest of Technical Papers. 1999 Symposium on VLSI Circuits, pp. 11 –12, 1999

[20] T. Xanthopoulos, and A. Chandrakasan, "A low-power IDCT macrocell for MPEG2 MP@ML exploiting data distribution properties for minimal activity", Digest of Technical Papers. 1998 Symposium on VLSI Circuits, pp. 38 –39, 1998

[21] T. Xanthopoulos, and A. Chandrakasan, "A low-power IDCT macrocell for MPEG2 MP@ML exploiting data distribution properties for minimal activity", IEEE J. of solid-state circuits, vol. 34, no. 5, pp. 693-703, May 1999

[22] T. Kuroda, T. Fujita, S. Mita, T. Nagamatsu, S. Yoshioka, K. Suzuki, F. Sano, M. Norishima, M. Murota, M. Kako, M. Kinugawa, M. Kakumu, and T. Sakurai, "A 0.9V 150MHz, 10mW 4mm^2, 2-D discrete cosine transform core processor with

variable threshold-voltage (VT) scheme", IEEE J. of solid-state circuits, vol. 31, no. 11, pp. 1770-1779, November 1996

[23] M. Matsui, H. Hara, Y. Uetani, L. S. Kim, T. Nagamatsu, Y. Watanabe, A. Chiba, K. Matsuda, and T. Sakurai, "A 200 MHz 13 mm^2 2-D DCT macrocell using sense-amplifying pipeline flip-flop scheme", IEEE J. of solid-state circuits, vol. 29, no. 12, pp. 1482-1490, December 1994

[24] M. Matsui, H. Hara, K. Seta, Y. Uetani, L. S. Kim, T. Nagamatsu, T. Shimazawa, S. Mita, G. Otomo, T. Oto, Y. Watanabe, F. Sano, A. Chiba, K. Matsuda, T. Sakurai, "200MHz video compression macrocells using low-swing differential logic", ISSC'94, pp. 76-77, 1994

[25] M. Hamada, T. Terazawa, T. Higashi, S. Kitabayashi, S. Mita, Y. Watanabe, M. Ashino, H. Hara, and T. Kuroda, "Flip-flop selection technique for power-delay trade-off", ISSC'99, pp. 270-271, 1999

[26] T. H. Chen, "A cost-effective 8x8 2-D IDCT core processor with folded architecture", IEEE trans. on consumer electronics, vol. 45, no. 2, pp.333-339, May 1999

[27] "IEEE Standard Specifications for the Implementation of 8x8 Inverse Discrete Cosine Transform", *IEEE Std. 1180-1990*, March, 1991.

[28] Xanthopoulos, "Low power data-dependent transform video and still image coding", Ph. D. Thesis, M. I. T., February 1999.

[29] E. Feing and S. Winograd, "Fast algorithms for the discrete cosine transform", IEEE trans. on signal processing, 40(9), pp. 2174-2193, September 1992.

[30] K. Hwang, *Computer Arithmetic – Principles, Architecture, and Design*, John Wiley & Songs, 1979, pp. 149-151.

[31] Z. Wang, "Fast Algorithms for Discrete W-Transform and for the Discrete Fourier Transform", IEEE trans. on acoustics, speech and signal processing, vol. ASSP-32, no. 4, pp. 803-816, August 1984.

[32] M. Vetterli, H. Nussbaumer, "Simple FFT and DCT Algorithms with Reduced Number of Operations", Signal Processing (North Holland), vol. 6. no. 4, pp. 264-278, August 1984

[33] N. Suehiro, M. Hatori, "Fast algorithms for the DFT and other Sinusoidal Transforms", IEEE Trans. on acoustics, speech, and signal processing, vol. ASSP-34, no. 3, pp. 642-664, June 1986

[34] P. Duhamel and H. H'Mida, "New 2^n DCT algorithms suitable for VLSI implementation", Proceedings IEEE international conference on acoustics, speech and signal processing, ICASSP-85, Dallas, pp. 1805-1808, April 1987

[35] K. Hwang, pp. 152-155

[36] S. Shah, A. J. Al-Khalili, and D. Al-Khalili, "Comparison of 32-bit multipliers of various performance measures", Proceedings of the 12th International Conference on Microelectronics, ICM'2000, pp. 75-80, October 31- November 2, 2000

[37] A. Bhattacharya and S. Haider, "A VLSI implementation of the inverse cosine transform, International J. of Pattern Recognition and AI, 9(2), pp. 303-314, 1995

[38] K. R. Rao and P. Yip, *Discrete Cosine Transform – Algorithms, Advantages, Applications*, Academic Press, 1990, pp. 10-15

[39] V. Lefèvre, "Multiplication by an integer constant", LIP research report RR1999-06, Laboratoire d'Informatique du Parallélisme, Lyon, France, 1999

[40] F. de Dinechine and V. Lefèvre, "Constant Multipliers for FPGAs", LIP research report RR2000-18, Laboratoire d'Informatique du Parallélisme, Lyon, France, 2000

[41] R. Bernstein, Multiplication by integer constants, Software – Practice and Experience, 16(7), July 1986, pp. 641-652

[42] M. Potkonjak, M. Srivastava, and A. Chandrakasan, "Multiple Constant Multiplications: Efficient and Versatile Frameworks for Exploring Common Subexpression Elimination", IEEE Trans. on CAD of IC and Systems, vol. 15, no. 2, pp. 151-165, February 1996

[43] Xilinx Cooperation, "Constant (k) Coefficient Multiplier Generator for Virtex", Application Note, Version 1.1, March 12, 1999

[44] Xilinx Cooperation, "Constant Coefficient Multipliers for XC4000E", Application Note XAPP 054, Version 1.1, December 11, 1996

[45] R. Hartley, "Optimization of Canonical Sign Digit Multipliers for Filter Design", IEEE International Sympoisum on Circuits and Systems, 1991, vol. 4, 1992-1995, 1991

Appendix A

Truncation Test Result

Table 17 shows the truncation error of 3 test video sequences: coke, salesman, and tennis. The truncation error is defined as:

Truncation Error = Average PSNR(reference) – Average PSNR(truncation)

Each sequence is encoded with pure I-frames, 8 Mb/s and 180 frames. The FDCT is computed with fixed-point calculation with 11-bit precision after binary points.

Truncation Error = Average PSNR(reference) – Average PSNR(truncation)					
Trunc(2,n)	Number of Truncated Bit	Tennis (dB)	Coke (dB)	Salesman (dB)	Average of 3 Sequences (dB)
00000000	0	0.0000	0.0000	0.0000	0.0000
00000001	1	0.0697	0.0867	0.0300	0.0621
00000011	2	0.1403	0.1709	0.0598	0.1237
00000002	2	0.3384	0.4251	0.1601	0.3079
00000111	3	0.2052	0.2524	0.0917	0.1831
00000012	3	0.4048	0.5034	0.1892	0.3658
00000003	3	1.2628	1.6514	0.5909	1.1684
00001111	4	0.2663	0.3314	0.1217	0.2398
00000112	4	0.4651	0.5786	0.2202	0.4213
00000022	4	0.6639	0.8224	0.3241	0.6035
00000013	4	1.3158	1.7101	0.6172	1.2144
00011111	5	0.3339	0.4208	0.1646	0.3064
00001112	5	0.5228	0.6519	0.2493	0.4747
00000122	5	0.7204	0.8923	0.3544	0.6557
00000113	5	1.3644	1.7675	0.6454	1.2591
00000023	5	1.5234	1.9530	0.7395	1.4053
00111111	6	0.4034	0.5111	0.2019	0.3721
00011112	6	0.5863	0.7348	0.2911	0.5374
00001122	6	0.7743	0.9604	0.3825	0.7057
00000222	6	0.9331	1.1671	0.4836	0.8612
00001113	6	1.4101	1.8231	0.6714	1.3015
00000123	6	1.5694	2.0070	0.7670	1.4478
00000033	6	2.2327	2.8048	1.2396	2.0924
01111111	7	0.4727	0.6043	0.2439	0.4403
00111112	7	0.6514	0.8190	0.3273	0.5992

00011122	7	0.8337	1.0377	0.4231	0.7648
00001222	7	0.9843	1.2310	0.5108	0.9087
00011113	7	1.4610	1.8865	0.7094	1.3523
00001123	7	1.6132	2.0596	0.7924	1.4884
00000223	7	1.7432	2.2202	0.8847	1.6160
00000133	7	2.2716	2.8492	1.2641	2.1283
11111111	8	0.5427	0.7062	0.2920	0.5136
01111112	8	0.7168	0.9059	0.3682	0.6637
00111122	8	0.8954	1.1163	0.4582	0.8233
00011222	8	1.0405	1.3038	0.5503	0.9648
00002222	8	1.1792	1.4767	0.6274	1.0945
00111113	8	1.5139	1.9509	0.7424	1.4024
00011123	8	1.6616	2.1201	0.8295	1.5371
00001223	8	1.7852	2.2705	0.9093	1.6550
00001133	8	2.3090	2.8926	1.2867	2.1628
00000233	8	2.4189	3.0258	1.3694	2.2714
11111112	9	0.7825	1.0008	0.4148	0.7327
01111122	9	0.9570	1.1973	0.4978	0.8840
00111222	9	1.0991	1.3774	0.5844	1.0203
00012222	9	1.2332	1.5450	0.6659	1.1480
01111113	9	1.5667	2.0178	0.7794	1.4546
00111123	9	1.7121	2.1810	0.8615	1.5849
00011223	9	1.8317	2.3281	0.9455	1.7018
00002223	9	1.9465	2.4648	1.0162	1.8092
00011133	9	2.3501	2.9424	1.3197	2.2041
00001233	9	2.4545	3.0673	1.3913	2.3044
00000333	9	2.9933	3.6931	1.7838	2.8234
11111122	10	1.0194	1.2866	0.5431	0.9497
01111222	10	1.1577	1.4536	0.6227	1.0780
00112222	10	1.2892	1.6149	0.6992	1.2011
00022222	10	1.4327	1.7947	0.7989	1.3421
11111113	10	1.6207	2.0917	0.8220	1.5115
01111123	10	1.7629	2.2445	0.8975	1.6350
00111223	10	1.8802	2.3863	0.9768	1.7478
00012223	10	1.9912	2.5197	1.0513	1.8541
00111133	10	2.3929	2.9931	1.3483	2.2448
00011233	10	2.4945	3.1150	1.4237	2.3444
00002233	10	2.5925	3.2292	1.4867	2.4361
00001333	10	3.0243	3.7289	1.8040	2.8524
11111222	11	1.2172	1.5378	0.6666	1.1406
01112222	11	1.3454	1.6869	0.7365	1.2563
00122222	11	1.4861	1.8605	0.8311	1.3925
11111123	11	1.8141	2.3143	0.9389	1.6891
01111223	11	1.9289	2.4470	1.0118	1.7959
00112223	11	2.0381	2.5754	1.0817	1.8984
00022223	11	2.1583	2.7199	1.1730	2.0171
01111133	11	2.4360	3.0457	1.3807	2.2875
00111233	11	2.5362	3.1639	1.4513	2.3838

00012233	11	2.6311	3.2754	1.5183	2.4749
00011333	11	3.0594	3.7700	1.8334	2.8876
00002333	11	3.1450	3.8684	1.8907	2.9680
11112222	12	1.4019	1.7668	0.7795	1.3161
01122222	12	1.5396	1.9289	0.8672	1.4453
00222222	12	1.6744	2.0973	0.9550	1.5756
11111223	12	1.9782	2.5134	1.0520	1.8479
01112223	12	2.0847	2.6334	1.1160	1.9447
00122223	12	2.2032	2.7735	1.2027	2.0598
11111133	12	2.4801	3.1040	1.4175	2.3338
01111233	12	2.5781	3.2147	1.4827	2.4252
00112233	12	2.6713	3.3224	1.5456	2.5131
00022233	12	2.7749	3.4441	1.6275	2.6155
00111333	12	3.0959	3.8120	1.8587	2.9222
00012333	12	3.1789	3.9079	1.9193	3.0021
00003333	12	3.6067	4.3936	2.2439	3.4147
11122222	13	1.5938	2.0042	0.9089	1.5023
01222222	13	1.7255	2.1621	0.9900	1.6259
11112223	13	2.1321	2.6971	1.1552	1.9948
01122223	13	2.2483	2.8287	1.2358	2.1043
00222223	13	2.3623	2.9656	1.3163	2.2147
11111233	13	2.6206	3.2706	1.5188	2.4700
01112233	13	2.7119	3.3714	1.5763	2.5532
00122233	13	2.8138	3.4893	1.6541	2.6524
01111333	13	3.1327	3.8556	1.8871	2.9584
00112333	13	3.2144	3.9485	1.9445	3.0358
00022333	13	3.3057	4.0540	2.0192	3.1263
00013333	13	3.6370	4.4286	2.2701	3.4452
11222222	14	1.7777	2.2335	1.0306	1.6806
02222222	14	1.9089	2.3903	1.1169	1.8054
11122223	14	2.2941	2.8898	1.2741	2.1527
01222223	14	2.4059	3.0187	1.3486	2.2577
11112233	14	2.7530	3.4253	1.6118	2.5967
01122233	14	2.8529	3.5364	1.6840	2.6911
00222233	14	2.9523	3.6535	1.7567	2.7875
11111333	14	3.1702	3.9038	1.9202	2.9981
01112333	14	3.2504	3.9908	1.9724	3.0712
00122333	14	3.3402	4.0933	2.0436	3.1590
00113333	14	3.6690	4.4648	2.2933	3.4757
00023333	14	3.7511	4.5585	2.3622	3.5572
12222222	15	1.9588	2.4581	1.1561	1.8576
11222223	15	2.4500	3.0772	1.3858	2.3043
02222223	15	2.5624	3.2070	1.4649	2.4114
11122233	15	2.8930	3.5886	1.7186	2.7334
01222233	15	2.9903	3.6986	1.7858	2.8249
11112333	15	3.2868	4.0377	2.0047	3.1097
01122333	15	3.3753	4.1346	2.0710	3.1936
00222333	15	3.4631	4.2364	2.1375	3.2790

01113333	15	3.7012	4.5024	2.3192	3.5076
00123333	15	3.7822	4.5937	2.3849	3.5869
00033333	15	4.1980	5.0523	2.6887	3.9796
22222222	16	2.1413	2.6887	1.2902	2.0400
12222223	16	2.6049	3.2632	1.5012	2.4564
11222233	16	3.0291	3.7487	1.8194	2.8657
02222233	16	3.1273	3.8598	1.8909	2.9593
11122333	16	3.4107	4.1799	2.1024	3.2310
01222333	16	3.4970	4.2761	2.1642	3.3125
11113333	16	3.7342	4.5440	2.3490	3.5424
01123333	16	3.8138	4.6300	2.4103	3.6180
00223333	16	3.8932	4.7212	2.4717	3.6954
00133333	16	4.2260	5.0834	2.7094	4.0063
22222223	17	2.7623	3.4554	1.6251	2.6143
12222233	17	3.1646	3.9082	1.9239	2.9989
11222333	17	3.5315	4.3201	2.1950	3.3489
02222333	17	3.6186	4.4174	2.2607	3.4322
11123333	17	3.8458	4.6706	2.4394	3.6519
01223333	17	3.9239	4.7565	2.4965	3.7257
01133333	17	4.2542	5.1161	2.7330	4.0344
00233333	17	4.3258	5.1975	2.7903	4.1046
22222233	18	3.3034	4.0742	2.0363	3.1380
12222333	18	3.6522	4.4599	2.2910	3.4677
11223333	18	3.9551	4.7959	2.5252	3.7588
02223333	18	4.0345	4.8833	2.5861	3.8346
11133333	18	4.2835	5.1523	2.7600	4.0653
01233333	18	4.3539	5.2291	2.8134	4.1321
00333333	18	4.6939	5.6152	3.0910	4.4667
22222333	19	3.7763	4.6066	2.3944	3.5924
12223333	19	4.0649	4.9214	2.6141	3.8668
11233333	19	4.3820	5.2644	2.8398	4.1621
02233333	19	4.4540	5.3432	2.8966	4.2313
01333333	19	4.7193	5.6440	3.1125	4.4919
22223333	20	4.1779	5.0538	2.7101	3.9806
12233333	20	4.4815	5.3777	2.9226	4.2606
11333333	20	4.7454	5.6761	3.1373	4.5196
02333333	20	4.8117	5.7477	3.1902	4.5832
22233333	21	4.5839	5.4967	3.0119	4.3642
12333333	21	4.8371	5.7789	3.2145	4.6102
03333333	21	5.1587	6.1360	3.4795	4.9247
22333333	22	4.9318	5.8876	3.2983	4.7059
13333333	22	5.1820	6.1646	3.5023	4.9497
23333333	23	5.2696	6.2642	3.5808	5.0382
33333333	24	5.5876	6.6137	3.8638	5.3550

Table 17: Truncation errors of test sequences: coke, salesman, and tennis

Appendix B

Sample Output of CSD Multiplier Generator

Sample VHDL code generated by the constant-coefficient multiplier generator (CMG) with the following parameters:

- Constant operand: cos(3π/16) with 15-bit precision multiplied (integer value 13623)
- Variable operand: 12-bit variable unsigned operand
- No truncation

Sample Output of Constant CSD Multiplier Generator

```
------------------------------------------------------------------
-- Hardwired Multiplier:
--   Constant Operand:
--     Integer value: 13623
--   Variable Operand:
--     Precision    : 12 bits
--     Signed       : False
--   Output
--     Truncated LSB: 0
--     Product      : True
--     Bypass Zero  : False
------------------------------------------------------------------

------------------------------------------------------------------
-- VHDL Code Generated by HWMult 1.0
------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
------------------------------------------------------------------

entity COS_3_16 is
  port
  (
    VarIn  : in  Std_Logic_Vector(11 downto 0);
    Result : out Std_Logic_Vector(25 downto 0)
  );
end;

------------------------------------------------------------------
```

```
architecture Structural of COS_3_16 is
  component HalfAdder
    port(A, B: in  Std_Logic; Sum, Cout: out Std_Logic);
  end component;

  component FullAdder
    port(A, B, Cin: in  Std_Logic; Sum, Cout: out Std_Logic);
  end component;

  signal
    S0, C0,
    S1, C1,
    S2, C2,
    S3, C3: Std_Logic_Vector(25 downto 0);

  signal N    : Std_Logic_Vector(11 downto 0);

  signal P    : Std_Logic_Vector(11 downto 0);
  signal num1 : Std_Logic_Vector(25 downto 0);
  signal num2 : Std_Logic_Vector(25 downto 0);

  signal num  : Std_Logic_Vector(25 downto 0);
  signal ZERO: Std_Logic;  -- Constant signal '0'
  signal ONE : Std_Logic;  -- Constant signal '1'
-------------------------------------------------------------------

begin

  ZERO <= '0';
  ONE  <= '1';

  P <= VarIn;

-- Inverted input signals:
  N <= not P;


-- Bit 0 Stage 0:
-- Bit 0 Stage 1:
-- Bit 0 Stage 2:
-- Bit 0 Stage 3:

-- Bit 1 Stage 0:
  HA_0_1: HalfAdder port map(N( 1),N( 0),S0( 1),C0( 1));
-- Bit 1 Stage 1:
-- Bit 1 Stage 2:
-- Bit 1 Stage 3:

-- Bit 2 Stage 0:
-- Bit 2 Stage 1:
  HA_1_2: HalfAdder port map(N( 2),C0( 1),S1( 2),C1( 2));
-- Bit 2 Stage 2:
-- Bit 2 Stage 3:

-- Bit 3 Stage 0:
  FA_0_3: FullAdder port map(N( 3),N( 0), ONE  ,S0( 3),C0( 3));
-- Bit 3 Stage 1:
-- Bit 3 Stage 2:
  HA_2_3: HalfAdder port map(S0( 3),C1( 2),S2( 3),C2( 3));
-- Bit 3 Stage 3:

-- Bit 4 Stage 0:
-- Bit 4 Stage 1:
```

```
  FA_1_4: FullAdder port map(N( 4),N( 1),C0( 3),S1( 4),C1( 4));
-- Bit 4 Stage 2:
-- Bit 4 Stage 3:
  HA_3_4: HalfAdder port map(S1( 4),C2( 3),S3( 4),C3( 4));

-- Bit 5 Stage 0:
-- Bit 5 Stage 1:
-- Bit 5 Stage 2:
  FA_2_5: FullAdder port map(N( 5),N( 2),C1( 4),S2( 5),C2( 5));
-- Bit 5 Stage 3:

-- Bit 6 Stage 0:
  FA_0_6: FullAdder port map(N( 6),N( 3),P( 0),S0( 6),C0( 6));
-- Bit 6 Stage 1:
-- Bit 6 Stage 2:
-- Bit 6 Stage 3:

-- Bit 7 Stage 0:
  FA_0_7: FullAdder port map(N( 7),N( 4),P( 1),S0( 7),C0( 7));
-- Bit 7 Stage 1:
-- Bit 7 Stage 2:
-- Bit 7 Stage 3:

-- Bit 8 Stage 0:
  FA_0_8: FullAdder port map(N( 8),N( 5),P( 2),S0( 8),C0( 8));
-- Bit 8 Stage 1:
  HA_1_8: HalfAdder port map(P( 0),S0( 8),S1( 8),C1( 8));
-- Bit 8 Stage 2:
-- Bit 8 Stage 3:

-- Bit 9 Stage 0:
  FA_0_9: FullAdder port map(N( 9),N( 6),P( 3),S0( 9),C0( 9));
-- Bit 9 Stage 1:
  FA_1_9: FullAdder port map(P( 1),S0( 9),C0( 8),S1( 9),C1( 9));
-- Bit 9 Stage 2:
-- Bit 9 Stage 3:

-- Bit 10 Stage 0:
  FA_0_10: FullAdder port map(N(10),N( 7),P( 4),S0(10),C0(10));
-- Bit 10 Stage 1:
  FA_1_10: FullAdder port map(P( 2),S0(10),C0( 9),S1(10),C1(10));
-- Bit 10 Stage 2:
  HA_2_10: HalfAdder port map(P( 0),S1(10),S2(10),C2(10));
-- Bit 10 Stage 3:

-- Bit 11 Stage 0:
  FA_0_11: FullAdder port map(N(11),N( 8),P( 5),S0(11),C0(11));
-- Bit 11 Stage 1:
  FA_1_11: FullAdder port map(P( 3),S0(11),C0(10),S1(11),C1(11));
-- Bit 11 Stage 2:
  FA_2_11: FullAdder port map(P( 1),S1(11),C1(10),S2(11),C2(11));
-- Bit 11 Stage 3:

-- Bit 12 Stage 0:
  FA_0_12: FullAdder port map(N( 9),P( 6),P( 4),S0(12),C0(12));
-- Bit 12 Stage 1:
  FA_1_12: FullAdder port map(P( 2),S0(12),C0(11),S1(12),C1(12));
-- Bit 12 Stage 2:
  FA_2_12: FullAdder port map(N( 0),S1(12),C1(11),S2(12),C2(12));
-- Bit 12 Stage 3:

-- Bit 13 Stage 0:
  FA_0_13: FullAdder port map(N(10),P( 7),P( 5),S0(13),C0(13));
```

```
-- Bit 13 Stage 1:
  FA_1_13: FullAdder port map(P( 3),S0(13),C0(12),S1(13),C1(13));
-- Bit 13 Stage 2:
  FA_2_13: FullAdder port map(N( 1),S1(13),C1(12),S2(13),C2(13));
-- Bit 13 Stage 3:

-- Bit 14 Stage 0:
  FA_0_14: FullAdder port map(N(11),P( 8),P( 6),S0(14),C0(14));
-- Bit 14 Stage 1:
  FA_1_14: FullAdder port map(P( 4),S0(14),C0(13),S1(14),C1(14));
-- Bit 14 Stage 2:
  FA_2_14: FullAdder port map(N( 2),S1(14),C1(13),S2(14),C2(14));
-- Bit 14 Stage 3:
  HA_3_14: HalfAdder port map(P( 0),S2(14),S3(14),C3(14));

-- Bit 15 Stage 0:
  FA_0_15: FullAdder port map(P( 9),P( 7),P( 5),S0(15),C0(15));
-- Bit 15 Stage 1:
  FA_1_15: FullAdder port map(N( 3), ONE  ,S0(15),S1(15),C1(15));
-- Bit 15 Stage 2:
  FA_2_15: FullAdder port map(P( 1),C0(14),S1(15),S2(15),C2(15));
-- Bit 15 Stage 3:
  FA_3_15: FullAdder port map(C1(14),S2(15),C2(14),S3(15),C3(15));

-- Bit 16 Stage 0:
  FA_0_16: FullAdder port map(P(10),P( 8),P( 6),S0(16),C0(16));
-- Bit 16 Stage 1:
  FA_1_16: FullAdder port map(N( 4), ONE  ,S0(16),S1(16),C1(16));
-- Bit 16 Stage 2:
  FA_2_16: FullAdder port map(P( 2),C0(15),S1(16),S2(16),C2(16));
-- Bit 16 Stage 3:
  FA_3_16: FullAdder port map(C1(15),S2(16),C2(15),S3(16),C3(16));

-- Bit 17 Stage 0:
  FA_0_17: FullAdder port map(P(11),P( 9),P( 7),S0(17),C0(17));
-- Bit 17 Stage 1:
  FA_1_17: FullAdder port map(N( 5), ONE  ,S0(17),S1(17),C1(17));
-- Bit 17 Stage 2:
  FA_2_17: FullAdder port map(P( 3),C0(16),S1(17),S2(17),C2(17));
-- Bit 17 Stage 3:
  FA_3_17: FullAdder port map(C1(16),S2(17),C2(16),S3(17),C3(17));

-- Bit 18 Stage 0:
  FA_0_18: FullAdder port map(P(10),P( 8),N( 6),S0(18),C0(18));
-- Bit 18 Stage 1:
  FA_1_18: FullAdder port map(P( 4), ONE  ,S0(18),S1(18),C1(18));
-- Bit 18 Stage 2:
  FA_2_18: FullAdder port map(C0(17),S1(18),C1(17),S2(18),C2(18));
-- Bit 18 Stage 3:
  HA_3_18: HalfAdder port map(S2(18),C2(17),S3(18),C3(18));

-- Bit 19 Stage 0:
  FA_0_19: FullAdder port map(P(11),P( 9),N( 7),S0(19),C0(19));
-- Bit 19 Stage 1:
  FA_1_19: FullAdder port map(P( 5), ONE  ,S0(19),S1(19),C1(19));
-- Bit 19 Stage 2:
  FA_2_19: FullAdder port map(C0(18),S1(19),C1(18),S2(19),C2(19));
-- Bit 19 Stage 3:
  HA_3_19: HalfAdder port map(S2(19),C2(18),S3(19),C3(19));

-- Bit 20 Stage 0:
  FA_0_20: FullAdder port map(P(10),N( 8),P( 6),S0(20),C0(20));
-- Bit 20 Stage 1:
```

```
  FA_1_20: FullAdder port map( ONE  ,S0(20),C0(19),S1(20),C1(20));
-- Bit 20 Stage 2:
-- Bit 20 Stage 3:
  FA_3_20: FullAdder port map(S1(20),C1(19),C2(19),S3(20),C3(20));

-- Bit 21 Stage 0:
  FA_0_21: FullAdder port map(P(11),N( 9),P( 7),S0(21),C0(21));
-- Bit 21 Stage 1:
  FA_1_21: FullAdder port map( ONE  ,S0(21),C0(20),S1(21),C1(21));
-- Bit 21 Stage 2:
  HA_2_21: HalfAdder port map(S1(21),C1(20),S2(21),C2(21));
-- Bit 21 Stage 3:

-- Bit 22 Stage 0:
  FA_0_22: FullAdder port map(N(10),P( 8), ONE  ,S0(22),C0(22));
-- Bit 22 Stage 1:
-- Bit 22 Stage 2:
  FA_2_22: FullAdder port map(S0(22),C0(21),C1(21),S2(22),C2(22));
-- Bit 22 Stage 3:

-- Bit 23 Stage 0:
  FA_0_23: FullAdder port map(N(11),P( 9), ONE  ,S0(23),C0(23));
-- Bit 23 Stage 1:
  HA_1_23: HalfAdder port map(S0(23),C0(22),S1(23),C1(23));
-- Bit 23 Stage 2:
-- Bit 23 Stage 3:

-- Bit 24 Stage 0:
-- Bit 24 Stage 1:
  HA_1_24: HalfAdder port map(P(10),C0(23),S1(24),C1(24));
-- Bit 24 Stage 2:
-- Bit 24 Stage 3:

-- Bit 25 Stage 0:
  HA_0_25: HalfAdder port map(P(11), ONE  ,S0(25),C0(25));
-- Bit 25 Stage 1:
-- Bit 25 Stage 2:
-- Bit 25 Stage 3:

  num1 <=   S0(25)& S1(24)& S1(23)& S2(22)& S2(21)& S3(20)& S3(19)& S3(18)
          & S3(17)& S3(16)& S3(15)& S3(14)& S2(13)& S2(12)& S2(11)& S2(10)
          & S1( 9)& S1( 8)& S0( 7)& S0( 6)& S2( 5)& S3( 4)& S2( 3)& S1( 2)
          & S0( 1)& P( 0);
  num2 <=   C1(24)& C1(23)& C2(22)& C2(21)& C3(20)& C3(19)& C3(18)& C3(17)
          & C3(16)& C3(15)& C3(14)& C2(13)& C2(12)& C2(11)& C2(10)& C1( 9)
          & C1( 8)& C0( 7)& C0( 6)& C2( 5)& C3( 4)&  ZERO &  ZERO &  ZERO
          &  ZERO &  ZERO ;

  num <= Unsigned(num1) + Unsigned(num2);
  Result <= num;

end;

--------------------------------------------------------------------
-- Statistical Information:
--   # Stage     : 4
--   # Inverter  : 12
--   # Half adder: 13
--   # Full adder: 48
--------------------------------------------------------------------
```

Appendix C

Source Code of Constant Multiplier Generator

The following is the C++ source code listing for constant multiplier generator. The codes are listed in alphabetic order of the source file name. The header file (.h) is always in front of the implementation file (.cpp). The main program is located inside file *IntMult.cpp*. Notice that all codes are also included in the attached CD.

CSA.h

```
#ifndef __CSA_H_
#define __CSA_H

#include <iostream>
#include "VHDL_Signal.h"
#include "NumberSystem.h"

using namespace std;

unsigned nReadyAtStage(SignalVector& imt, int curStage);

void getAdderOperand(unsigned nOp, unsigned maxConstInput, SignalVector&imt,
SignalVector& opToAdd);

void createHA(vector<SignalVector> &imt, unsigned curBit, unsigned curStage,
unsigned maxConstOp, ostream& o);
void createFA(vector<SignalVector> &imt, unsigned curBit, unsigned curStage,
unsigned maxConstOp, ostream& o);

void generate_VHDL_CSA_Body(vector<SignalVector> &imt, unsigned
CSA_Stage,unsigned &nHalfAdder, unsigned &nFullAdder, ostream& csa);

void createCSA(vector<SignalVector>& op, vector<SignalVector>& csa);
void simplifyConstants(vector<SignalVector> &c);

#endif
```

CSA.cpp

```
#include "CSA.h"
#include <algorithm>

using namespace std;

static Signal
  SIGNAL_SUM   (VARIABLE,"S",SUM   ,false,NONE,-1,-1),
  SIGNAL_CARRY(VARIABLE,"C",CARRY,false,NONE,-1,-1),
```

```
  SIGNAL_SIGN (SIGN,"Sign",0,false,NONE,-1,-1);

//-------------------------------------------------------------------------
unsigned nReadyAtStage(SignalVector& sv, int curStage)
{
  unsigned nReady=0;
  for (unsigned i=0; i<sv.size(); i++)
    if (sv[i].stage<curStage)
      nReady++;
  return nReady;
}

//-------------------------------------------------------------------------
void getAdderOperand(unsigned nOp, unsigned maxConstInput,
                     SignalVector& sv, SignalVector& opToAdd)
{
  unsigned i=0;
  while (maxConstInput>0 && i<sv.size())
    if (sv[i].stage==-1)
    {
      maxConstInput--;  nOp--;
      opToAdd.push_back(sv[i]);
      sv.erase(sv.begin()+i);
    }
    else i++;
  if (nOp==0) return;

  sort(sv.begin(), sv.end());

  i=0;
  while (nOp>0 && i<sv.size())
    if (sv[i].ID==CARRY||sv[i].ID==SUM)
    {
      nOp--;
      opToAdd.push_back(sv[i]);
      sv.erase(sv.begin()+i);
    }
    else i++;
  if (nOp==0) return;

  while (nOp>0 && sv.size()>0)
  {
    nOp--;
    opToAdd.push_back(sv[0]);
    sv.erase(sv.begin());
  }
}

//-------------------------------------------------------------------------
void createHA(vector<SignalVector> &sv,
              unsigned curBit, unsigned curStage, unsigned maxConstOp,
              ostream& o)
{
  SignalVector opToAdd;
  getAdderOperand(2, maxConstOp, sv[curBit], opToAdd);

  SIGNAL_SUM.bitPos=SIGNAL_CARRY.bitPos=curBit;
  SIGNAL_SUM.stage =SIGNAL_CARRY.stage =curStage;

  o <<"  HA_"<<curStage<<"_"<<curBit<<": HalfAdder port map("
    << opToAdd[0] <<","    // operand 1
    << opToAdd[1] <<","    // operand 2
    << SIGNAL_SUM <<","
```

```
    << SIGNAL_CARRY<<");\n";

  sv[curBit].push_back(SIGNAL_SUM);
  if (curBit==sv.size()-1) return;
  sv[curBit+1].push_back(SIGNAL_CARRY);
  return;
}

//------------------------------------------------------------------------
void createFA(vector<SignalVector> &sv,
              unsigned curBit, unsigned curStage, unsigned maxConstOp,
              ostream& o)
{
  SignalVector opToAdd;
  getAdderOperand(3, maxConstOp, sv[curBit], opToAdd);

  SIGNAL_SUM.bitPos=SIGNAL_CARRY.bitPos=curBit;
  SIGNAL_SUM.stage =SIGNAL_CARRY.stage =curStage;

  o <<"  FA_"<<curStage<<"_"<<curBit<<": FullAdder port map("
    << opToAdd[0] <<","    // operand 1
    << opToAdd[1] <<","    // operand 2
    << opToAdd[2] <<","    // operand 3
    << SIGNAL_SUM <<","
    << SIGNAL_CARRY << ");\n";

  sv[curBit].push_back(SIGNAL_SUM);
  if (curBit==sv.size()-1) return;
  sv[curBit+1].push_back(SIGNAL_CARRY);
  return;
}

//------------------------------------------------------------------------
void generate_VHDL_CSA_Body(
  vector<SignalVector> &sv, unsigned CSA_Stage,
  unsigned &nHalfAdder, unsigned &nFullAdder, ostream& csa)
{
  unsigned  i, j;

  //-------------------------------------------------------------------------
  // Generating carry-save adder VHDL code
  nHalfAdder=nFullAdder=0;   // Complexity Stat

  bool HA_for_2op = true;
  bool isFirstAdder;
  int  nReady;
  for (i=0; i<sv.size(); i++)
  {
    csa << "\n";
    isFirstAdder = true;
    for (j=0; j<CSA_Stage; j++)
    {
      csa << "-- Bit "<<i<<" Stage "<<j<<":\n";
      if (HA_for_2op)
      {
        switch (nReadyAtStage(sv[i], j))
        {
          case 0: break;
          case 1: break;
          case 2:
          {
            if (sv[i].size()==2)
            {
```

```
              createHA(sv, i, j, (isFirstAdder?2:1), csa);
              if (j==CSA_Stage-1) HA_for_2op=false;
              isFirstAdder=false;  nHalfAdder++;
            }
            break;
          }
          default:
          {
            createFA(sv, i, j, (isFirstAdder?3:1), csa);
            if (j==CSA_Stage-1) HA_for_2op=false;
            isFirstAdder=false;    nFullAdder++;
          }
        }
      }
      else  // HA_for_2Op = false
      {
        nReady = nReadyAtStage(sv[i], j);
        if (sv[i].size()==3)
        {
          if (nReady==2||nReady==3)
          {
            createHA(sv, i, j, (isFirstAdder?2:1), csa);
            isFirstAdder = false;   nHalfAdder++;
          }
        }
        else if (nReady>=3)
        {
          createFA(sv, i, j, (isFirstAdder?3:1), csa);
          isFirstAdder = false;     nFullAdder++;
        }
      }
    }
  }
}

//------------------------------------------------------------------------
void simplifyConstants(vector<SignalVector> &c)
{
  SignalVector::iterator result;
  int nOne, carry=0;
  for (unsigned i=0; i<c.size(); i++)
  {
    nOne=0;
    //count (c[i].begin(),c[i].end(),SIGNAL_ONE,nOne);
    nOne = count (c[i].begin(),c[i].end(),SIGNAL_ONE);

    // Removing constant zeros: No operation
    result = remove(c[i].begin(),c[i].end(),SIGNAL_ZERO);
    c[i].erase(result,c[i].end());

    // Simplify constant ones: adding them together
    result = remove(c[i].begin(),c[i].end(),SIGNAL_ONE);
    c[i].erase(result,c[i].end());

    nOne+=carry;
    carry=nOne/2;    nOne%=2;
    if (nOne!=0)
      c[i].push_back(SIGNAL_ONE);
  }
}

//------------------------------------------------------------------------
void create_CSA_Vector(
```

```
  vector<SignalVector>& op, vector<SignalVector>& c sa, bool isSigned)
{
  int i, j;
  unsigned maxBit=0;
  for (i=0; i<op.size(); i++)
    if (op[i].size()>maxBit)
      maxBit=op.size();

  for (i=op.size()>>1, j=0; i!=0; i>>=1, j++);  // Get the MSB position of i:
log2(op.size())
  maxBit+=j;                                    // n m-bit operand will have
output of n+m bit
  csa.resize(maxBit);

  Signal signal(VARIABLE,"Op",0,true,NONE,-1,-1);
  for (i=0; i<op.size(); i++)
  {
    signal.ID = i;
    for (j=0; j<(isSigned?op[i].size()-1:op[i].size()); j++)
    {
      if (op[i][j].type==CONSTANT)
      {
        if (op[i][j].ID==ONE) csa[j].push_back(SIGNAL_ONE);
      }
      else
      {
        signal.bitPos = j;
        csa[j].push_back(signal);
      }
    }

    if (isSigned)
    {
      SIGNAL_SIGN.ID=i;
      for (; j<maxBit; j++) csa[j].push_back(SIGNAL_SIGN);
    }
  }
}


//-----------------------------------------------------------------------
void generate_VHDL_CSA_Header(
  char* entityName, vector<SignalVector>& op, bool  isSigned,
  int nBitOut, int& CSA_Stage, ostream& o)
{
  int i;

  // Generate Library Header
  o << "-------------------------------------------------------------\n"
    << "-- VHDL Code Generated by HWMult 1.0\n"
    << "-------------------------------------------------------------\n"
    << "library ieee;\n"
    << "use ieee.std_logic_1164.all;\n"
    << "-------------------------------------------------------------\n\n";

  // Generate VHDL Entity Header
  o << "entity " << entityName << " is\n"
    << "  port\n"
    << "  (\n";

  Signal signal(VARIABLE,"Op",0,true,NONE,-1,-1);
  for (signal.ID=0; signal.ID<op.size(); signal.ID++)
    o << "    "<<signal<<": in  Std Logic Vector("<<(op[signal.ID].size()-1)<<"
```

```
downto 0);\n";

  signal.name = "Sum";
  for (signal.ID=1; signal.ID<=2; signal.ID++)
    o << "    "<<signal<<" : out Std_Logic_Vector("<<(nBitOut-1)<<" downto
0);\n";

  o << "  );\n"
    << "end;\n\n";

  o << "--------------------------------------------------------------\n\n";

  // Generate VHDL Architecture Header
  o << "architecture Structural of " << entityName <<" is\n"
    << "  component HalfAdder\n"
    << "    port(A, B: in  Std_Logic; Sum, Cout: out Std_Logic);\n"
    << "  end component;\n\n"
    << "  component FullAdder\n"
    << "    port(A, B, Cin: in  Std_Logic; Sum, Cout: out Std_Logic);\n"
    << "  end component;\n\n";

  CSA_Stage = (op.size()>=3 ? op.size()-2 : 1);
  SIGNAL_SUM.showID=SIGNAL_CARRY.showID=true;
  o << "  signal" << endl;
  for (i=0; i<=CSA_Stage; i++)
  {
    SIGNAL_SUM.ID = SIGNAL_CARRY.ID = i;
    o << "    "<<SIGNAL_SUM<<", "<<SIGNAL_CARRY;
    if (i!=CSA_Stage)
        o << ",\n";
    else o << ": Std_Logic_Vector("<<(nBitOut-1)<<" downto 0);\n\n";
  }

  if (isSigned)
  {
    o << "  signal ";
    for (SIGNAL_SIGN.ID=0; SIGNAL_SIGN.ID<op.size(); SIGNAL_SIGN.ID++)
      o << SIGNAL_SIGN << (SIGNAL_SIGN.ID<op.size()-1 ? ", " : "");
    o << ": Std_Logic_Vector("<<(op.size()-1)<<" downto 0);\n";
  }

  o << "  signal ZERO: Std_Logic;  -- Constant signal '0'\n";
  o << "  signal ONE : Std_Logic;  -- Constant signal '1'\n\n";
  o << "--------------------------------------------------------------\n\n";

  o << "begin\n\n"
    << "  ZERO <= '0';\n"
    << "  ONE  <= '1';\n\n";

  if (isSigned)
  {
    signal.name = "Op";
    for (i=0; i<op.size(); i++)
    {
      SIGNAL_SIGN.ID = signal.ID = i;
      o << "  "<<SIGNAL_SIGN<<" <= "<<signal<<"("<<(op[i].size()-1)<<");\n";
    }
  }
}

//-------------------------------------------------------------------------
void generate_VHDL_CSA_Tail(
  vector<SignalVector> &imt,
```

```
  SignalVector &out1, SignalVector &out2,
  ostream& o)
{
  //-------------------------------------------------------------------------
  // Map internal signals to output
  ostrstream num1, num2;
  Signal signal(VARIABLE,"Sum",0,true,NONE,-1,-1);

  signal.ID=1;  num1 << "  "<<signal<<" <= ";
  signal.ID=2;  num2 << "  "<<signal<<" <= ";

  int i, j=0, msb=imt.size()-1;
  for (i=msb; i>=0; i--)
  {
    switch (imt[i].size())
    {
      case 0:
      {
        num1<<(i!=msb?"& ":"  ")<<SIGNAL_ZERO;
        num2<<(i!=msb?"& ":"  ")<<SIGNAL_ZERO;
        out1.insert(out1.begin(),SIGNAL_ZERO);
        out2.insert(out2.begin(),SIGNAL_ZERO);
        break;
      }
      case 1:
      {
        num1<<(i!=msb?"& ":"  ")<<imt[i][0];
        num2<<(i!=msb?"& ":"  ")<<SIGNAL_ZERO;
        out1.insert(out1.begin(),imt[i][0]);
        out2.insert(out2.begin(),SIGNAL_ZERO);
        break;
      }
      default:
      {
        num1<<(i!=msb?"& ":"  ")<<imt[i][1];
        num2<<(i!=msb?"& ":"  ")<<imt[i][0];
        out1.insert(out1.begin(),imt[i][1]);
        out2.insert(out2.begin(),imt[i][0]);
      }
    }

    j++;
    if ((j%8)==0)
    {
      num1 << "\n          ";  num2 << "\n          ";
    }
  }
  num1<<";\n";    num2<<";\n";
  num1.flush();  num2.flush();

  char *s1 = num1.str();  s1[num1.pcount()]='\0';
  char *s2 = num2.str();  s2[num2.pcount()]='\0';
  o << "\n" << s1 << s2 << "\n";
  o << "end;\n\n";
  o.flush();
}

//-------------------------------------------------------------------------
void generate_VHDL_CSA(
  char* entityName, vector<SignalVector>& op, bool isSigned,
  SignalVector &out1, SignalVector &out2,
  ostream& o)
{
```

```
  int CSA_Stage;
  unsigned nHalfAdder, nFullAdder;
  vector<SignalVector> csa;

  create_CSA_Vector(op, csa, isSigned);
  generate_VHDL_CSA_Header(entityName, op, isSigned, csa.size(), CSA_Stage, o);
  generate_VHDL_CSA_Body  (csa, CSA_Stage,  nHalfAdder, nFullAdder, o);
  generate_VHDL_CSA_Tail  (csa, out1, out2, o);

  o << "--------------------------------------------------------------------\n";
  o << "-- Statistical Information:\n"
    << "--    # Stage      : " << CSA_Stage<<"\n"
    << "--    # Half adder: " << nHalfAdder <<"\n"
    << "--    # Full adder: " << nFullAdder <<"\n";
  o << "--------------------------------------------------------------------\n\n";
}

//------------------------------------------------------------------------
```

HWMult.h

```
#ifndef __HWMULT_H
#define __HWMULT_H

#include <limits.h>
#include <iostream>
#include <vector>

#include "VHDL_Signal.h"

using namespace std;

void HWMult(
  unsigned nVarBit, bool signedVar, unsigned long constOp,
  vector<SignalVector> &out,
  ostream& o, ostream& component, char* entityName=0,
  unsigned truncLSB=0, bool generateProduct=true, bool byPass=false);

#endif
```

HWMult.cpp

```
#include "HWMult.h"
#include "CSA.h"
#include "NumberSystem.h"
#include "NonZero.h"

#include <vector>
#include <iomanip>
#include <algorithm>

using namespace std;

static Signal
  SIGNAL_SIGN_P(SIGN,"Sign",0,false,POSITIVE,-1,-1),
  SIGNAL_SIGN_N(SIGN,"Sign",0,false,NEGATIVE,-1,-1);

//------------------------------------------------------------------------
void generate_VHDL_HWM_Header(
  unsigned nVarBit,  bool signedVar,        unsigned long constOp,
  char* entityName,
```

```
  unsigned nOutBit1, unsigned nOutBit2,      unsigned outBit2_offset,
  unsigned CSA_Stage,unsigned nCSABit,       bool invertedInput,
  ostream& o, ostream& c, bool generateProduct, bool byPass)
{
  unsigned i;
  if (entityName==0)
  {
    ostrstream name;
    name << "HWM_" <<constOp<<"x"<<nVarBit<<"_Bit";
    entityName = name.str();
  }

  // Generate Library Header
  o << "--------------------------------------------------------------------\n"
    << "-- VHDL Code Generated by HWMult 1.0\n"
    << "--------------------------------------------------------------------\n"
    << "library ieee;\n"
    << "use ieee.std_logic_1164.all;\n";
  if (generateProduct)
    o << "use ieee.std_logic_arith.all;\n";
  o << "--------------------------------------------------------------------\n\n";

  if (outBit2_offset!=0)
  {
    o << "-- Note: num2 is offseted by "<<outBit2_offset<<" bits.\n\n";
    o << "--------------------------------------------------------------------
\n\n";
  }

  // Generate VHDL Entity Header
  o << "entity " << entityName << " is\n"
    << "  port\n"
    << "  (\n"
    << "    VarIn  : in  Std_Logic_Vector("<<(nVarBit -1)<<" downto 0);\n";
  c << "  component " << entityName << "\n"
    << "  port\n"
    << "  (\n"
    << "    VarIn  : in  Std_Logic_Vector("<<(nVarBit -1)<<" downto 0);\n";

  if (generateProduct)
  {
    o << "    Result : out Std_Logic_Vector("<<(nOutBit1-1)<<" downto 0)\n";
    c << "    Result : out Std_Logic_Vector("<<(nOutBit1-1)<<" downto 0)\n";
  }
  else
  {
    o << "    Result1: out Std_Logic_Vector("<<(nOutBit1-1)<<" downto 0);\n"
      << "    Result2: out Std_Logic_Vector("<<(nOutBit2-1)<<" downto 0)\n";
    c << "    Result1: out Std_Logic_Vector("<<(nOutBit1-1)<<" downto 0);\n"
      << "    Result2: out Std_Logic_Vector("<<(nOutBit2-1)<<" downto 0)\n";
  }

  o << "  );\n"
    << "end;\n\n";
  c << "  );\n"
    << "  end component;\n\n";

  o << "--------------------------------------------------------------------\n\n";

  // Generate VHDL Architecture Header
  o << "architecture Structural of " << entityName <<" is\n"

    << "  component HalfAdder\n"
```

```
    << "    port(A, B: in  Std_Logic; Sum, Cout: out Std_Logic);\n"
    << "  end component;\n\n"

    << "  component FullAdder\n"
    << "    port(A, B, Cin: in  Std_Logic; Sum, Cout: out Std_Logic);\n"
    << "  end component;\n\n";

  if (byPass)
  {
    char BP[50];
    sprintf(BP,"NZ%i",nVarBit);
    NonZero(BP,nVarBit,o);
  }

  o << "  signal" << endl;
  for (i=0; i<CSA_Stage; i++)
  {
    o << "    S" << i <<", C" << i;
    if (i!=CSA_Stage-1)
        o << ",\n";
    else o << ": Std_Logic_Vector("<<(nCSABit-1)<<" downto 0);\n\n";
  }

  if (invertedInput)
  {
    o << "  signal N    : Std_Logic_Vector("<<(nVarBit-1)<<" downto 0);\n\n";
    if (signedVar)
      o << "  signal "<<SIGNAL_SIGN_P<<", "<<SIGNAL_SIGN_N<<": Std_Logic;\n";
  }

  o << "  signal P    : Std_Logic_Vector("<<(nVarBit -1)<<" downto 0);\n"
    << "  signal num1 : Std_Logic_Vector("<<(nOutBit1-1)<<" downto 0);\n"
    << "  signal num2 : Std_Logic_Vector("<<(nOutBit2-1)<<" downto 0);\n\n";

  if (generateProduct)
    o << "  signal num  : Std_Logic_Vector("<<(nOutBit1-1)<<" downto 0);\n";


  o << "  signal ZERO: Std_Logic;  -- Constant signal '0'\n"
    << "  signal ONE : Std_Logic;  -- Constant signal '1'\n";

  if (byPass)
    o << "  signal NonZeroIn: Std_Logic;\n"
      << "  signal ZERO_Out : Std_Logic_Vector("<<(nOutBit1-1)<<" downto
0);\n\n";

  o << "-------------------------------------------------------------------\n\n";

  o << "begin\n\n"
    << "  ZERO <= '0';\n"
    << "  ONE  <= '1';\n\n";

  if (byPass)
  {
    o << "  BP: NZ"<<nVarBit<<" port map(VarIn,NonZeroIn);\n"
      << "  P <= VarIn when (NonZeroIn='1') else P;\n\n";
  }
  else o << "  P <= VarIn;\n\n";

  if (invertedInput)
  {
    o << "-- Inverted input signals:\n";
    //for (i=0; i<nVarBit; i++)
```

```
    //  o << "  N("<<i<<") <= not P("<<i<<");\n";
    o << "  N <= not P;\n";
    o << "\n";

    if (signedVar)  // Signed variable operand
      o << "  " << SIGNAL_SIGN_P << " <= P("<<(nVarBit-1)<<");\n"
        << "  " << SIGNAL_SIGN_N << " <= N("<<(nVarBit-1)<<");\n\n";
  }
}

//-------------------------------------------------------------------------
void generate_VHDL_HWM_Tail(
  vector<SignalVector> &imt,
  vector<SignalVector> &out,
  ostream& o, bool generateProduct, bool byPass)
{
  //-----------------------------------------------------------------------
  // Map internal signals to output
  ostrstream num1, num2;
  num1 << "  num1 <= ";
  num2 << "  num2 <= ";

  out.resize(imt.size());
  int i, j=0, msb=imt.size()-1;
  for (i=msb; i>=0; i--)
  {
    switch (imt[i].size())
    {
      case 0:
      {
        num1<<(i!=msb?"& ":"  ")<<SIGNAL_ZERO;
        num2<<(i!=msb?"& ":"  ")<<SIGNAL_ZERO;
        break;
      }
      case 1:
      {
        num1<<(i!=msb?"& ":"  ")<<imt[i][0];
        num2<<(i!=msb?"& ":"  ")<<SIGNAL_ZERO;
        out[i].push_back(imt[i][0]);
        break;
      }
      default:
      {
        num1<<(i!=msb?"& ":"  ")<<imt[i][1];
        num2<<(i!=msb?"& ":"  ")<<imt[i][0];
        out[i].push_back(imt[i][0]);
        out[i].push_back(imt[i][1]);
      }
    }

    j++;
    if ((j%8)==0)
    {
      num1 << "\n          ";  num2 << "\n          ";
    }
  }
  num1<<";\n";    num2<<";\n";
  num1.flush();  num2.flush();

  char *s1 = num1.str();  s1[num1.pcount()]='\0';
  char *s2 = num2.str();  s2[num2.pcount()]='\0';
  o << "\n" << s1 << s2 << "\n";
```

```
  if (byPass)
  {
    o << "  ZERO_Out <= \"";
    for (i=0; i<out.size(); i++) o<<"0";
    o << "\";\n\n";
  }

  if (generateProduct)
  {
    o << "  num <= Unsigned(num1) + Unsigned(num2);\n";
    if (byPass)
        o << "  Result <= num  when (NonZeroIn='1') else ZERO_Out;\n\n";
    else o << "  Result <= num;\n\n";
  }
  else
  {
    if (byPass)
        o << "  Result1<= num1 when (NonZeroIn='1') else ZERO_Out;\n"
          << "  Result2<= num2 when (NonZeroIn='1') else ZERO_Out;\n\n";
    else o << "  Result1<= num1;\n"
           << "  Result2<= num2;\n\n";
  }

  o << "end;\n\n";
  o.flush();
}

//---------------------------------------------------------------------------
void simplify_SIGN(bool signedVar, vector<SignalVector>& sign)
{
  int i;

  if (signedVar)
  {
    vector<SignalVector> signZero(sign), signOne(sign);
    SignalVector::iterator result;

    for (i=0; i<sign.size(); i++)
    {
      // For Sign=0 => Remove all "Sign_P (=0)" & Replace Sign_N with "1"
      result = remove(signZero[i].begin(),signZero[i].end(),SIGNAL_SIGN_P);
      signZero[i].erase(result,signZero[i].end());
      replace(signZero[i].begin(),signZero[i].end(),SIGNAL_SIGN_N,SIGNAL_ONE);

      // For Sign=1 => Remove all "Sign_N (=0)" & Replace Sign_P with "1"
      result = remove(signOne[i].begin(),signOne[i].end(),SIGNAL_SIGN_N);
      signOne[i].erase(result,signOne[i].end());
      replace(signOne[i].begin(),signOne[i].end(),SIGNAL_SIGN_P,SIGNAL_ONE);
    }

    simplifyConstants(signZero);
    simplifyConstants(signOne);

    // Merge 2 possible sign vectors together
    for (i=0; i<sign.size(); i++)
    {
      sign[i].clear();
      if (signZero[i].size()==0)
      {
        if (signOne[i].size()==1)
          sign[i].push_back(SIGNAL_SIGN_P);
      }
      else // signZero[i].size()==1
```

```
      {
        if (signOne[i].size()==0)
             sign[i].push_back(SIGNAL_SIGN_N);
        else sign[i].push_back(SIGNAL_ONE);
      }
    }
  }
  else simplifyConstants(sign);
}

//---------------------------------------------------------------------------
void create_HWM_Vector (
  unsigned nVarBit, bool signedVar, unsigned long constOp,
  vector<SignalVector>& imt, bool &invertedInput, unsigned &CSA_Stage)
{
  unsigned  i, j;
  vector<SignalVector> sign;

  vector<char> constBit;
  ulongToBit(constOp, constBit);

  cout << "Constant Operand (" << constBit.size() << " bits):\n";
  showBit(constBit);
  cout << "\n\n";

  unsigned nOutBit = nVarBit + constBit.size();        // Number of output bit
  imt .resize(nOutBit);  // Intermediate signals
  sign.resize(nOutBit);  // Sign and constant 1's

  // Variable bit <= Constant bit => Perform Consant * Variable
  binaryToSignDigit(constBit);

  cout << "Constant Operand (" << constBit.size() << " bits) in SD form:\n";
  showBit(constBit);
  cout << "\n\n";

  // Insert all intermediate signals
  Signal signal;

  invertedInput = false;
  for (i=0; i<constBit.size(); i++)
  {
    if (constBit[i]==1)
    {
      for (j=0; j<(signedVar?nVarBit-1:nVarBit); j++)
      {
        signal.bitPos=j;
        signal.inverted=POSITIVE;
        imt[i+j].push_back(signal);
      }

      if (signedVar)
        for (j+=i; j<imt.size(); j++) sign[j].push_back(SIGNAL_SIGN_P);
    }
    else if (constBit[i]==-1)
    {
      invertedInput = true;
      for (j=0; j<(signedVar?nVarBit-1:nVarBit); j++)
      {
        signal.inverted=NEGATIVE;
        signal.bitPos=j;
        imt[i+j].push_back(signal);
      }
```

```
      sign[i].push_back(SIGNAL_ONE);

      if (signedVar)
        for (j+=i; j<imt.size(); j++) sign[j].push_back(SIGNAL_SIGN_N);
      else
        for (j+=i; j<imt.size(); j++) sign[j].push_back(SIGNAL_ONE);
    }
  }

  simplify_SIGN(signedVar, sign);

  // Merge sign/constants togeter and perform optimization for constant 1's
  unsigned maxDepth=0;
  for (i=0; i<imt.size(); i++)
  {
    if (sign[i].size()!=0)
    {
      if (sign[i][0]==SIGNAL_ONE && i<imt.size()-1 && imt[i].size()==1 &&
imt[i+1].size()<=2)
      {
        // bit + 1 => sum=(not bit), carry=bit
        imt[i+1].push_back (imt[i][0]);
        imt[i][0].inverted = imt[i][0].inverted==POSITIVE ? NEGATIVE: POSITIVE;
        invertedInput = true;
      }
      else imt[i].push_back(sign[i][0]);
    }

    if (imt[i].size()>maxDepth)
      maxDepth=imt[i].size();
  }

  CSA_Stage=(maxDepth>3) ? maxDepth-2 : (maxDepth>0 ? 1 : 0);
}

//------------------------------------------------------------------------
void HWMult (
  unsigned nVarBit, bool signedVar, unsigned long constOp,
  vector<SignalVector> &out,
  ostream& o, ostream& component, char* entityName,
  unsigned truncLSB, bool generateProduct, bool byPass)
{
  unsigned  i, j;

  vector<SignalVector> imt;
  bool invertedInput;
  unsigned  CSA_Stage;

  //------------------------------------------------------------------------
  // Construct Mutiplication vector to be used in CSA
  create_HWM_Vector(nVarBit, signedVar, constOp, imt, invertedInput,
CSA_Stage);

  if (truncLSB!=0)
  {
    cout << "\nBefore truncation:\n";
    printVSV(imt);

    imt.erase(imt.begin(),imt.begin()+truncLSB);
    cout << "\nAfter truncating "<<truncLSB<<" bits:\n";
  }
```

```
  printVSV(imt);

  //-------------------------------------------------------------------------
  // Generating VHDL Code
  generate_VHDL_HWM_Header(nVarBit, signedVar, constOp, entityName,
    imt.size(), imt.size(), 0, CSA_Stage, imt.size(), invertedInput,
    o, component, generateProduct, byPass);
  o.flush();

  //-------------------------------------------------------------------------
  // Generating carry-save adder VHDL code
  unsigned nHalfAdder, nFullAdder;
  generate_VHDL_CSA_Body(imt, CSA_Stage, nHalfAdder, nFullAdder, o);
  o.flush();

  //-------------------------------------------------------------------------
  // Generating VHDL tail (end architecture & statistical information
  generate_VHDL_HWM_Tail(imt, out, o, generateProduct, byPass);

  // Print Statistical Information
  o << "-------------------------------------------------------------------\n";
  o << "-- Statistical Information:\n"
    << "--    # Stage     : " << CSA_Stage<<"\n"
    << "--    # Inverter  : " << (invertedInput?nVarBit:0)<<"\n"
    << "--    # Half adder: " << nHalfAdder <<"\n"
    << "--    # Full adder: " << nFullAdder <<"\n";
  o << "-------------------------------------------------------------------\n\n";
}
```

IntMult.cpp

```
#include <math.h>
#include <iostream>
#include <fstream>
#include <vector>
#include <stdio.h>
#include "HWMult.h"

using namespace std;

const double pi = 3.14159265358979323846;

int main(int argc, char* argv[])
{
  unsigned long val;
  cout << "Constant Operand              : ";
  cin  >> val;

  int  nVarBit, truncLSB, bypass;
  int signedVar, generateProduct;
  cout << "# bit of Variable Operand     : ";
  cin  >> nVarBit;
  cout << "Signed variable operand (0/1): ";
  cin  >> signedVar;
  cout << "# bit truncated at LSB        : ";
  cin  >> truncLSB;
  cout << "Generate product         (0/1): ";
  cin  >> generateProduct;
  cout << "Bypass Zero              (0/1): ";
  cin  >> bypass;
```

```
  char entityName[256], fileName[256];
  cout << "Entity (file) name           : ";
  cin  >> entityName;
  if (entityName[0]=='\0')

sprintf(entityName,"HWM_%i_x_%i_Bit_%s",val,nVarBit,(signedVar?"S":"U"));

  sprintf(fileName,"%s.vhd",entityName);
  ofstream vhdl(fileName,ios::out);

  cout << "\n\n";
  cout << "Constant Value: "<<val<<"\n\n";

  vhdl << "--------------------------------------------------------------------\n"
       << "-- Hardwired Multiplier:\n"
       << "--   Constant Operand:\n"
       << "--     Integer value: "<<val<<"\n"
       << "--   Variable Operand:\n"
       << "--     Precision    : "<<nVarBit<<" bits\n"
            << "--     Signed       : "<<(signedVar?"True\n":"False\n")
            << "--   Output\n"
            << "--     Truncated LSB: "<<truncLSB<<"\n"
            << "--     Product      : "<<(generateProduct?"True\n":"False\n")
            << "--     Bypass Zero  : "<<(bypass?"True\n":"False\n")
       << "--------------------------------------------------------------------
\n\n";

  vector<SignalVector> out;
  ostrstream component;
  HWMult(nVarBit,signedVar!=0,val,out,vhdl,component,entityName,truncLSB,
         generateProduct!=0,bypass!=0);
  vhdl.close();

  cout << "Component Header:\n" << component.str() << "\n\n";

  cout << "Output Signals:\n";
  printVSV(out,cout);

  return 0;
}
//---------------------------------------------------------------------------
```

NonZero.h

```
#include "NonZero.h"
#include <fstream>

using namespace std;

void NonZero(char* entityName, unsigned nBit, ostream& c)
{
  char fileName[256];
  sprintf(fileName,"%s.vhd",entityName);
  fstream f(fileName,ios::out);

  f << "--------------------------------------------------------------------\n"
    << "library ieee;\n"
    << "use ieee.std_logic_1164.all;\n"
    << "--------------------------------------------------------------------
\n";

  f << "entity "<<entityName<<" is\n"
```

```
    << "  port\n"
    << "  (\n"
    << "    D : in  Std_Logic_Vector("<<(nBit-1)<<" downto 0);\n"
    << "    NZ: out Std_Logic\n"
    << "  );\n"
    << "end;\n\n";

  c << "  component "<<entityName<<"\n"
    << "  port\n"
    << "  (\n"
    << "    D : in  Std_Logic_Vector("<<(nBit-1)<<" downto 0);\n"
    << "    NZ: out Std_Logic\n"
    << "  );\n"
    << "  end component;\n\n";

  f << "architecture Structural of "<<entityName<<" is\n"
    << "begin\n"
    << "  NZ <= \n";

  for (int i=0; i<nBit; i++)
  {
    if (i%4==0) f<<"    ";
    f << "D("<<i<<")"<<(i==nBit-1?";":" or ");
    if (i%4==3||i==nBit-1) f<<"\n";
  }
  f << "end;\n\n";

  f.close();
}
```

NumberSystem.h

```
#ifndef __NUMBERSYSTEM_H
#define __NUMBERSYSTEM_H

#include <iostream>
#include <vector>

using namespace std;

// Convert unsigned long to a sequence of bit.
// The MSB of the returning bit is always 0

void     ulongToBit(unsigned long l, vector<char>& bit);

void     binaryToSignDigit(vector<char>& bit);
void     optimizeSD        (vector<char> &bit);  // Reduce -1's
void     ulongToSignDigit (unsigned long l, vector<char>& sd);

void     ulongToBooth      (unsigned long l, vector<char>& booth);
void     BoothToSignDigit (vector<char>&booth, vector<char>& sd);

ostream& printBit(ostream& o, vector<char>& bit);
void     showBit(vector<char>& bit);

#endif
```

NumberSystem.cpp

```
#include "NumberSystem.h"
#include <iomanip>

// Convert unsigned long to a sequence of bit.
// The MSB of the returning bit is always 0
void ulongToBit(unsigned long l, vector<char>& bit)
{
  bit.clear();
  for (; l!=0; l>>=1)
    bit.push_back(l&1 ? 1: 0);
}

void showBit(vector<char>& bit)
{
  printBit(cout, bit);
}

ostream& printBit(ostream& o, vector<char> &bit)
{
  int weight=0;

  if (bit.size()==0)
  {
    o << "The number is ZERO  Weight=" << weight;   return o;
  }

  for (int i=bit.size()-1; i>=0; i--)
  {
    o << setw(3) << (int)bit[i];
    if (bit[i]!=0) weight++;
  }
  o << "  Weight=" << weight;
  return o;
}

void binaryToSignDigit(vector<char>& bit)
{
  int i, nBit=bit.size();
  int start, end;    // Start and end position of consecutive ones

  bit.push_back(0);
  for (i=0; i<nBit; i++)
    if (bit[i]!=0)
    {
      start=i;
      for (end=i+1; end<nBit; end++)
        if (bit[end]==0) break;

      if (end-start>1)    // More then one 1's
      {
        bit[start]=-1;    bit[end]=1;
        for (start++; start<end; start++) bit[start]=0;
      }
      i = end-1;
    }

  if (bit[bit.size()-1]==0)
    bit.erase(bit.end()-1);
}

void optimizeSD(vector<char>& bit)
{
```

```
  int nBit = bit.size();
  for (int i=0; i<nBit-2; i++)
    if (bit[i]==-1 && bit[i+1]==0 && bit[i+2]==1)
    {
      bit[i]=1;  bit[i+1]=1;  bit[i+2]=0;
    }
}

void ulongToSignDigit(unsigned long l, vector<char>& bit)
{
  ulongToBit(l, bit);
  binaryToSignDigit(bit);
//  optimizeSD(bit);
}

void ulongToBooth(unsigned long l, vector<char>& booth)
{
  static const char toBooth[] = { 0, 1, 1, 2, -2, -1, -1, 0 };

  if (l==0) { booth.push_back(0); return; }

  booth.push_back(toBooth[(l&3)<<1]);
  l>>=1;
  while (l!=0)
  {
    booth.push_back(toBooth[l&7]);
    l>>=2;
  }
}

void BoothToSignDigit (vector<char>&booth, vector<char>& sd)
{
  for (int i=0; i<booth.size(); i++)
    switch (booth[i])
    {
      case -2: sd.push_back( 0);  sd.push_back(-1);  break;
      case -1: sd.push_back(-1);  sd.push_back( 0);  break;
      case  0: sd.push_back( 0);  sd.push_back( 0);  break;
      case  1: sd.push_back( 1);  sd.push_back( 0);  break;
      case  2: sd.push_back( 0);  sd.push_back( 1);  break;
    }
}
```

VHDL_Signal.h

```
#ifndef __VHDL_SIGNAL_H
#define __VHDL_SIGNAL_H

#include <vector>
#include <string>
#include <iostream>
#include <iomanip>
#include <strstream>

using namespace std;

typedef enum { CONSTANT, VARIABLE, SIGN } signalType;
typedef enum { NONE, POSITIVE, NEGATIVE } invertType;
typedef enum { ZERO , ONE, OPEN    } constIDType;
typedef enum { INPUT, SUM, CARRY   } varIDType;

struct Signal
```

```
{
  Signal(): type(VARIABLE), name(""), ID(0), inverted(NONE), bitPos(-1),
stage(-1), showID(false) {};
  Signal(signalType t, char *n, unsigned id, bool showid, invertType inv, int
pos, int stg)
    :
type(t),name(n),ID(id),showID(showid),inverted(inv),bitPos(pos),stage(stg) {};

  signalType type;
  char*      name;
  unsigned   ID;         // ID for the signal
  bool       showID;
  invertType inverted;
  int        bitPos;     // Bit position (Non-negative integer. -1 will not show
the bit position)
  int        stage ;     // Stage where this signal is generated (-1: input or
constant signal & will not show the stage)

  const char* toString();

  static int createNewSignal() { unsigned save=idCount++; return save; }
  static int idCount;
};

ostream& operator << (ostream& stream, const Signal& signal);
bool operator==(const Signal&s1, const Signal &s2);
bool operator!=(const Signal&s1, const Signal &s2);
bool operator <(const Signal&s1, const Signal &s2);

const Signal
  SIGNAL_ZERO  (CONSTANT," ZERO "  ,ZERO,false,NONE,-1,-1),
  SIGNAL_ONE   (CONSTANT," ONE  "  ,ONE ,false,NONE,-1,-1),
  SIGNAL_OPEN  (CONSTANT," \'X\'  ",OPEN,false,NONE,-1,-1);

typedef vector<Signal> SignalVector;
void printSV (SignalVector &imt, ostream& o=cout);
void printVSV(vector<SignalVector> &imt, ostream& o=cout);

#endif
```

VHDL_Signal.cpp

```
#include "VHDL_Signal.h"

using namespace std;

int Signal::idCount=0;

//---------------------------------------------------------------------------
const char* Signal::toString()
{
  ostrstream result;
  result << *this;
  return result.str();
}

//---------------------------------------------------------------------------
ostream& operator << (ostream& stream, const Signal& signal)
{
  stream << signal.name;
  if (signal.type==CONSTANT) return stream;
```

```
  if (signal.showID)
    stream<<signal.ID;

  if (signal.inverted!=NONE)
    stream << (signal.inverted==POSITIVE ? "P": "N");

  // Variable signal
  if (signal.stage>=0)
    stream << signal.stage;

  if (signal.bitPos>=0)
    stream<<"("<<setw(2)<< signal.bitPos <<")";

  return stream;
}

//---------------------------------------------------------------------------
bool operator==(const Signal&s1, const Signal &s2)
{
  if (s1.type!=s2.type) return false;
  if (s1.type==CONSTANT) return (s1.ID==s2.ID);
  return (s1.ID==s2.ID && s1.inverted==s2.inverted && s1.bitPos==s2.bitPos &&
s1.stage==s2.stage);
}

//---------------------------------------------------------------------------
bool operator!=(const Signal&s1, const Signal &s2)
{
  return !(s1==s2);
}

//---------------------------------------------------------------------------
bool operator <(const Signal &s1, const Signal &s2)
{
  if (s1.stage<s2.stage) return true;
  if (s1.type==CONSTANT) return true;
  if (s1.stage==s2.stage)
    if (s1.ID==SUM && s2.ID==CARRY) return true;
  return false;
}

//---------------------------------------------------------------------------
void printSV(SignalVector&sv, ostream& o)
{
  o << "[";
  for (unsigned j=0; j<sv.size(); j++)
  {
    o << sv[j] << " ";
  }
  o << "]\n";
}

//---------------------------------------------------------------------------
void printVSV(vector<SignalVector> &sv, ostream& o)
{
  for (unsigned i=0; i<sv.size(); i++)
  {
    o << setw(3) << i << ": ";
    printSV(sv[i], o);
  }
}

//---------------------------------------------------------------------------
```

Appendix D

IEEE Standard 1180-1990 Compliant Test Program

The following is the Java source code listing for IEEE Standard 1180-1990 compliance test program for IDCT. It is used to determine the internal bandwidth of the IDCT for both the first dimension IDCT and second dimension IDCT.

The codes are listed in alphabetic order based on the source file name. The main program is located inside file *IEEE_1180_1990.java*. Notice that all codes are also included in the attached CD.

To execute the program, use the following command: *java IEEE_1180_1990.* The program reads the internal bandwidth configuration from file *Setup.txt*, and perform test to check if the bandwidth yields IEEE 1180-1990 compliance.

CSD.java

```
/*
   Convert conventional binary number to canonical sign-digit representation
   Algorithm: H. Hwang, Computer Arithmetic, Wiley, 1979, pp. 150
   Coding   : Pai, Cheng-Yu
   Note     :
     To compile, execute "javac SignDigit.java"
     To run    , execute "java  SignDigit xxxx",
                 where xxxx is the number wish to convert.
*/

public class CSD
{
  public static byte[] toCSD(long l)
  {
//     System.out.println("Integer value = "+l);
//     System.out.println("Integer bits  = "+Long.toBinaryString(l));

       // Construct bit array representation of the input
       byte[] b = ("0"+Long.toBinaryString(l)).getBytes();
       for (int i=0,j=b.length-1; i<=j; i++,j--)
```

```
        { byte temp=(byte)(b[i]-'0');  b[i]=(byte)(b[j]-'0');     b[j]=temp; }

      byte[] d = new byte[b.length];
    byte ci=0, ci_1;
    for (int i=0; i<b.length; i++, ci=ci_1)
    {
      if (i==b.length-1)
           ci_1 = (byte)((b[i]+ci>1)?1:0);
      else ci_1 = (byte)((b[i]+b[i+1]+ci>1)?1:0);
      d[d.length-i-1] = (byte)(b[i]+ci-2*ci_1);
    }

    for (int i=0, j=d.length-1; i<j; i++, j--)
      { byte temp=d[i];  d[i]=d[j];  d[j]=temp; }
/*
    System.out.println("CSD value:");
    for (int i=d.length-1; i>=0; i--)
      System.out.print((d[i]==0?" 0":(d[i]==1?" 1":"-1"))+" ");
    System.out.println();
*/
    return d;
  }
}
```

FDCT.java

```
public class FDCT
{
  static double s  [][] = new double[5][8];
  static double tmp[][] = new double[8][8];
  static final int map[]={0,4,2,6,7,3,5,1};

  static void Butterfly(int stage, int x0, int x1)
  {
    s[stage+1][x0] = s[stage][x0] + s[stage][x1];
    s[stage+1][x1] = s[stage][x0] - s[stage][x1];
  }

  static void Loeffler(double A, double BminusA, double AplusB, int stage, int
x0, int x1)
  {
    double temp = A*(s[stage][x0]+s[stage][x1]);
    s[stage+1][x0] = temp + BminusA * s[stage][x1];
    s[stage+1][x1] = temp - AplusB  * s[stage][x0];
  }

  static void Lr1(int stage, int x0, int x1)
  {
    final int    n=1;
    final double k=Math.sqrt(2);
    final double a=k*Math.cos(n*Math.PI/16),
           b=k*Math.sin(n*Math.PI/16),
           BminusA=b-a,
           AplusB=a+b;
    Loeffler(a, BminusA, AplusB, stage, x0, x1);
  }

  static void Lc1(int stage, int x0, int x1)
  {
    final int    n=1;
    final double k=1;
    final double a=k*Math.cos(n*Math.PI/16),
```

```
            b=k*Math.sin(n*Math.PI/16),
            BminusA=b-a,
            AplusB=a+b;
    Loeffler(a, BminusA, AplusB, stage, x0, x1);
  }

  static void Lc3(int stage, int x0, int x1)
  {
    final int    n=3;
    final double k=1;
    final double a=k*Math.cos(n*Math.PI/16),
            b=k*Math.sin(n*Math.PI/16),
            BminusA=b-a,
            AplusB=a+b;
    Loeffler(a, BminusA, AplusB, stage, x0, x1);
  }

  public static void fdct(short block[][])
  {
    int i, j, k;
    double root2=Math.sqrt(2);

    for (i=0; i<8; i++)
    {
      // Input mapping
      for (j=0; j<8; j++) s[0][j]=block[i][j];

      // Stage 1: Butterfly
      for (j=0; j<4; j++)
        Butterfly(0,j,7-j);

      // Stage 2
      for (j=0; j<2; j++)
        Butterfly(1,j,3-j);
      Lc3(1,4,7);
      Lc1(1,5,6);

      // Stage 3
      Butterfly(2,0,1);
      Lr1       (2,2,3);
      Butterfly(2,4,6);
      Butterfly(2,7,5);

      // Stage 4
      for (j=0; j<4; j++) s[4][j]=s[3][j];
      Butterfly(3,7,4);
      s[4][5] = root2 * s[3][5];
      s[4][6] = root2 * s[3][6];

      // Output mapping
      for (j=0; j<8; j++)
        tmp[map[j]][i] = s[4][j];
/*
      System.out.println("1D S:");
      for (j=0; j<5; j++)
      {
        for (k=0; k<8; k++)
          System.out.print(s[j][k]+", ");
        System.out.println();
      }
*/
    }
/*
```

```
    System.out.println("FDCT 1D:");
    for (i=0; i<8; i++)
    {
      for (j=0; j<8; j++)
        System.out.print(tmp[i][j]+", ");
      System.out.println();
    }
    System.out.println();
*/
    for (i=0; i<8; i++)
    {
      // Input mapping
      for (j=0; j<8; j++) s[0][j]=tmp[i][j];

      // Stage 1: Butterfly
      for (j=0; j<4; j++)
        Butterfly(0,j,7-j);

      // Stage 2
      for (j=0; j<2; j++)
        Butterfly(1,j,3-j);
      Lc3(1,4,7);
      Lc1(1,5,6);

      // Stage 3
      Butterfly(2,0,1);
      Lr1       (2,2,3);
      Butterfly(2,4,6);
      Butterfly(2,7,5);

      // Stage 4
      for (j=0; j<4; j++) s[4][j]=s[3][j];
      Butterfly(3,7,4);
      s[4][5] = root2 * s[3][5];
      s[4][6] = root2 * s[3][6];

      // Output mapping
      for (j=0; j<8; j++)
        block[i][map[j]] = (short)Math.round(s[4][j]);
/*
      System.out.println("2D S:");
      for (j=0; j<5; j++)
      {
        for (k=0; k<8; k++)
          System.out.print(s[j][k]+", ");
        System.out.println();
      }
*/
    }
  }
}
```

IDCT.java

```
import java.text.*;

public class IDCT
{
  static double s   [][] = new double[5][8];
  static double tmp[][] = new double[8][8];
  static final int map[]={0,4,2,6,7,3,5,1};
```

```
  static DecimalFormat nf = new DecimalFormat("####0.####");

  static void IButterfly(int stage, int x0, int x1)
  {
    s[stage+1][x0] = (s[stage][x0] + s[stage][x1])/2;
    s[stage+1][x1] = (s[stage][x0] - s[stage][x1])/2;
  }

  static void ILoeffler(double C, double DminusC, double DplusC, int stage, int
x0, int x1)
  {
    double tmp = C*(s[stage][x0]+s[stage][x1]);
    s[stage+1][x0] = DplusC  * s[stage][x0] - tmp;
    s[stage+1][x1] = DminusC * s[stage][x1] + tmp;
  }

  static void Ic1(int stage, int x0, int x1)
  {
    final int    n = 1;
    final double k = 1;
    final double c = Math.sin(n*Math.PI/16)/k,
                 d = Math.cos(n*Math.PI/16)/k,
                 DminusC = d-c,
                 DplusC  = d+c;
    ILoeffler(c,DminusC,DplusC,stage,x0,x1);
  }

  static void Ic3(int stage, int x0, int x1)
  {
    final int    n = 3;
    final double k = 1;
    final double c = Math.sin(n*Math.PI/16)/k,
                 d = Math.cos(n*Math.PI/16)/k,
                 DminusC = d-c,
                 DplusC  = d+c;
    ILoeffler(c,DminusC,DplusC,stage,x0,x1);
  }

  static void Ir1(int stage, int x0, int x1)
  {
    final int    n = 1;
    final double k = Math.sqrt(2);
    final double c = Math.sin(n*Math.PI/16)/k,
                 d = Math.cos(n*Math.PI/16)/k,
                 DminusC = d-c,
                 DplusC  = d+c;
    ILoeffler(c,DminusC,DplusC,stage,x0,x1);
  }

  public static void idct(short block[][])
  {
    int i, j, k;
    final double invRoot2 = 1.0/Math.sqrt(2);
/*
    System.out.println();
    System.out.println("Dimension 0:");
*/
    for (i=0; i<8; i++)
    {
      // Input mapping
      for (j=0; j<8; j++)
        s[0][j] = block[i][map[j]];
```

```
      // Stage 1
      for (j=0; j<4; j++) s[1][j]=s[0][j];
      IButterfly(0,7,4);
      s[1][5] = s[0][5] * invRoot2;
      s[1][6] = s[0][6] * invRoot2;

      // Stage 2
      IButterfly(1,0,1);
      Ir1       (1,2,3);
      IButterfly(1,4,6);
      IButterfly(1,7,5);

      // Stage 3
      for (j=0; j<2; j++)
        IButterfly(2,j,3-j);
      Ic3(2,4,7);
      Ic1(2,5,6);

      // Stage 4
      for (j=0; j<4; j++)
        IButterfly(3,j,7-j);

      for (j=0; j<8; j++)
        tmp[j][i] = s[4][j];
/*
      System.out.println("Row "+i);
      for (j=0; j<5; j++)
      {
        System.out.print("Stage "+j+": ");
        for (k=0; k<8; k++)
          System.out.print(nf.format(s[j][k])+", ");
        System.out.println();
      }
*/
    }
/*
    System.out.println("IDCT 1D:");
    for (i=0; i<8; i++)
    {
      for (j=0; j<8; j++)
        System.out.print(tmp[i][j]+", ");
      System.out.println();
    }
    System.out.println();
*/
/*
    System.out.println();
    System.out.println("Dimension 1:");
*/
    for (i=0; i<8; i++)
    {
      // Input mapping
      for (j=0; j<8; j++)
        s[0][j] = tmp[i][map[j]];

      // Stage 1
      for (j=0; j<4; j++) s[1][j]=s[0][j];
      IButterfly(0,7,4);
      s[1][5] = s[0][5] * invRoot2;
      s[1][6] = s[0][6] * invRoot2;

      // Stage 2
      IButterfly(1,0,1);
```

```
      Ir1       (1,2,3);
      IButterfly(1,4,6);
      IButterfly(1,7,5);

      // Stage 3
      for (j=0; j<2; j++)
        IButterfly(2,j,3-j);
      Ic3(2,4,7);
      Ic1(2,5,6);

      // Stage 4
      for (j=0; j<4; j++)
        IButterfly(3,j,7-j);

      for (j=0; j<8; j++)
        block[i][j] = (short)Math.round(s[4][j]);
/*
      System.out.println("Row "+i);
      for (j=0; j<5; j++)
      {
        System.out.print("Stage "+j+": ");
        for (k=0; k<8; k++)
          System.out.print(nf.format(s[j][k])+", ");
        System.out.println();
      }
*/
    }
  }
}
```

IDCT_Trunc.java

```
import java.text.*;

public class IDCT_Trunc
{
  static long s  [][] = new long[5][8];
  static long tmp[][] = new long[8][8];
  static final int map[]={0,4,2,6,7,3,5,1};

  static byte invRoot2[];
  static byte cOp[][][] = new byte[3][3][];

  public static long nMul, nAdd;

//  static DecimalFormat nf = new DecimalFormat("###############");

  static void init_IDCT_help (int idx, double c, double sub, double sum, int
prec)
  {
    int i;
    long factor = ((long)1)<<prec;
    long cL, subL, sumL;
    cL  = (long)Math.round(c  *factor);
    subL= (long)Math.round(sub*factor);
    sumL= (long)Math.round(sum*factor);

    System.out.println("cL="+cL+", subL="+subL+", sumL="+sumL);

    cOp[idx][0] = CSD.toCSD(cL);
    cOp[idx][1] = CSD.toCSD(subL);
    cOp[idx][2] = CSD.toCSD(sumL);
```

```
  }

  public static void init_IDCT_Trunc(int prec[])
  {
    final double k[]={1,1,Math.sqrt(2)};
    final int    n[]={1,3,1};
    double c, d, sub, sum;

    System.out.println("Initialize IDCT coefficients:");

    for (int i=0; i<3; i++)
    {
      System.out.print((i!=2?"1":"R")+"c"+n[i]+": ");
      c = Math.sin(n[i]*Math.PI/16)/k[i];
      d = Math.cos(n[i]*Math.PI/16)/k[i];
      sub=d-c;              sum=d+c;
      init_IDCT_help(i,c,sub,sum,prec[i]);
    }

    final double ir = 1/Math.sqrt(2);
    long factor=((long)1)<<prec[3];
    long r2L    = (long)Math.round(ir*factor);
    invRoot2 = CSD.toCSD(r2L);

    System.out.println("1/Sqrt(2)="+r2L);
  }
/*
  static long mult(byte sd[], long val, int trunc)
  {
    long result=0;
    long pp;
    for (int i=0; i<sd.length; i++)
    {
      if (sd[i]==0) continue;
      if (i<trunc) pp=val>>(trunc-i);
      else pp=val<<(i-trunc);
      if (sd[i]==1)
           result+=pp;
      else result-=pp;
    }
    return result;
  }
*/

  static long mult(byte sd[], long val, int trunc)
  {
    long result=0;
    long pp;
    if (val==0) return 0;
    nMul++;

    for (int i=0; i<sd.length; i++)
    {
      if (sd[i]==0) continue;
      if (sd[i]==1)
      {
        if (i<trunc) pp=val>>(trunc-i);
        else pp=val<<(i-trunc);
      }
      else // sd[i]==-1
      {
        if (i<trunc) pp=(~val)>>(trunc-i);
        else pp=(-val)<<(i-trunc);
```

```
      }
      result+=pp;
    }
    return result;
  }

  static void IButterfly(int stage, int x0, int x1)
  {
    s[stage+1][x0] = (s[stage][x0] + s[stage][x1])/2;
    s[stage+1][x1] = (s[stage][x0] - s[stage][x1])/2;
    nAdd+=2;
  }

  static void IButterfly2(int stage, int x0, int x1)
  {
    s[stage+1][x0] = (s[stage][x0] + s[stage][x1]);
    s[stage+1][x1] = (s[stage][x0] - s[stage][x1]);
    nAdd+=2;
  }

  static void ILoeffler(byte C[], byte DminusC[], byte DplusC[], int stage, int
x0, int x1, int trunc)
  {
    long tmp = mult(C, s[stage][x0]+s[stage][x1], trunc);
    s[stage+1][x0] = mult(DplusC , s[stage][x0] , trunc) - tmp;
    s[stage+1][x1] = mult(DminusC, s[stage][x1] , trunc) + tmp;

    if (tmp!=0)
    {
      if (s[stage][x1]!=0) nAdd+=2; else nAdd++;
    }
  }

  static void Ic1(int stage, int x0, int x1, int trunc)
  {
    ILoeffler(cOp[0][0],cOp[0][1],cOp[0][2],stage,x0,x1,trunc);
  }

  static void Ic3(int stage, int x0, int x1, int trunc)
  {
    ILoeffler(cOp[1][0],cOp[1][1],cOp[1][2],stage,x0,x1,trunc);
  }

  static void Ir1(int stage, int x0, int x1, int trunc)
  {
    ILoeffler(cOp[2][0],cOp[2][1],cOp[2][2],stage,x0,x1,trunc);
  }

  static void adjustOffset(long stage[], int offset[])
  {
    for (int i=0; i<8; i++)
    {
      if (offset[i]<0)
        stage[i]>>=(-offset[i]);
      else if (offset[i]>0)
        stage[i]<<=offset[i];
    }
  }

  static long calcR(int r)
  {
    if (r<2) return 0;  // Do nothing
    int i, j;
```

```
    long offset;
    for (i=1,offset=1; i<r-1; i++) offset=(offset<<1)|1;
//    offset = ((long)1)<<(r-2);
    return offset;
  }

  public static void idctTrunc(short block[][], int trunc[][], int
offset[][][],int round[])
  {
    int i, j, k;
    long r0, r1;

    nMul = nAdd = 0;

    r0 = calcR(round[0]);
    r1 = calcR(round[1]);

//  System.out.println("r0="+r0+", r1="+r1);

/*
    System.out.println();
    System.out.println("Dimension 0:");
*/
    for (i=0; i<8; i++)
    {
      // Input mapping
      for (j=0; j<8; j++)
        s[0][j] = block[i][map[j]];

      adjustOffset(s[0],offset[0][0]);

      // Stage 1
      IButterfly2(0,0,1);
      Ir1        (0,2,3,trunc[0][2]);
      IButterfly2(0,7,4);
      s[1][5] = mult(invRoot2,s[0][5],trunc[0][3]);
      s[1][6] = mult(invRoot2,s[0][6],trunc[0][3]);

      adjustOffset(s[1],offset[0][1]);

      // Stage 2
      IButterfly2(1,0,3);
      IButterfly2(1,1,2);
      IButterfly2(1,4,6);
      IButterfly2(1,7,5);

      adjustOffset(s[2],offset[0][2]);

      // Stage 3
      for (j=0; j<4; j++) // Rounding
        s[3][j] = s[2][j] + (r0 << (-offset[0][4][j]-round[0]));
      nAdd+=4;
      Ic3(2,4,7,trunc[0][1]);
      Ic1(2,5,6,trunc[0][0]);

      adjustOffset(s[3],offset[0][3]);

      // Stage 4
      for (j=0; j<4; j++)
        IButterfly2(3,j,7-j);

      adjustOffset(s[4],offset[0][4]);
```

```
      for (j=0; j<8; j++)
        tmp[j][i] = s[4][j];
//        tmp[j][i] = (s[4][j]+r0)>>round[0];

/*
      // Debug
      System.out.println("Row "+i);
      for (j=0; j<5; j++)
      {
        System.out.print("Stage "+j+": ");
        for (k=0; k<8; k++)
          System.out.print(s[j][k]+", ");
        System.out.println();
      }
      System.out.print("Round  : ");
      for (j=0; j<8; j++)
        System.out.print(tmp[j][i]+", ");
      System.out.println();
*/
    }
/*
    System.out.println();
    System.out.println("Dimension 1:");
*/
    for (i=0; i<8; i++)
    {
      // Input mapping
      for (j=0; j<8; j++)
        s[0][j] = tmp[i][map[j]];

      adjustOffset(s[0],offset[1][0]);

      // Stage 1
      IButterfly2(0,0,1);
      Ir1         (0,2,3,trunc[1][2]);
      IButterfly2(0,7,4);
      s[1][5] = mult(invRoot2,s[0][5],trunc[1][3]);
      s[1][6] = mult(invRoot2,s[0][6],trunc[1][3]);

      adjustOffset(s[1],offset[1][1]);

      // Stage 2
      IButterfly2(1,0,3);
      IButterfly2(1,1,2);
      IButterfly2(1,4,6);
      IButterfly2(1,7,5);

      adjustOffset(s[2],offset[1][2]);

      // Stage 3
      for (j=0; j<4; j++)
        s[3][j] = s[2][j] + (r1 << (-offset[1][4][j]-round[1]));
      nAdd+=4;
      Ic3(2,4,7,trunc[1][1]);
      Ic1(2,5,6,trunc[1][0]);

      adjustOffset(s[3],offset[1][3]);

      // Stage 4
      for (j=0; j<4; j++)
        IButterfly2(3,j,7-j);

      adjustOffset(s[4],offset[1][4]);
```

```
      for (j=0; j<8; j++)
        block[i][j] = (short)s[4][j];
//        block[i][j] = (short)((s[4][j]+r1)>>round[1]);

/*
      System.out.println("Row "+i);
      for (j=0; j<5; j++)
      {
        System.out.print("Stage "+j+": ");
        for (k=0; k<8; k++)
          System.out.print(s[j][k]+", ");
        System.out.println();
      }
      System.out.print("Round  : ");
      for (j=0; j<8; j++)
        System.out.print(block[i][j]+", ");
      System.out.println();
*/
    }
  }
}
```

IEEE_1180_1990.java

```
import java.io.*;

public class IEEE_1180_1990
{
  static long eMAX    = 1,
              pmseMAX= (long)(0.06  *10000),
              pmeMAX = (long)(0.015 *10000),
              omeMAX = (long)(0.0015*64*10000),
              omseMAX= (long)(0.02  *64*10000);

  static long e   [][] = new long[8][8],
              pmse[][] = new long[8][8],
              pme [][] = new long[8][8],
              ome,
              omse;

  static boolean checkError(short xCal[][], short xRef[][])
  {
    int  i, j, err, e2;
    long eAbs, pmeAbs, omeAbs;

    for (i=0; i<8; i++)
    {
      for (j=0; j<8; j++)
      {
        e[i][j]=err = xCal[i][j]-xRef[i][j];
        e2          = err * err;
        pmse[i][j] += e2;
        pme [i][j] += err;
        ome        += err;
        omse       += e2;

//        sumE       += err;
//        System.out.print(err+" ");

        eAbs   = (err<0 ? -err : err);
        pmeAbs = (pme[i][j]<0 ? -pme[i][j] : pme[i][j]);
```

```
        omeAbs = (ome<0 ? -ome : ome);

//      System.out.println("["+i+","+j+"]: xCal="+xCal[i][j]+",
xRef="+xRef[i][j]+", pmse="+pmse[i][j]+", pme="+pme[i][i]+", ome="+ome+",
omse="+omse);
        if (eAbs>eMAX)          { System.out.println("Error: ppe ="+e[i][j]
+", ppeMAX ="+eMAX   );  return false; }
        if (pmse[i][j]>pmseMAX) { System.out.println("Error:
pmse="+pmse[i][j]+", pmseMAX="+pmseMAX);  return false; }
        if (pmeAbs>pmeMAX)      { System.out.println("Error: pme ="+pme[i][j]
+", pmeMAX ="+pmeMAX );  return false; }
        if (omeAbs>omeMAX)      { System.out.println("Error: ome ="+ome
+", omeMAX ="+omeMAX );  return false; }
        if (omse>omseMAX)       { System.out.println("Error: omse="+omse
+", omseMAX="+omseMAX);  return false; }
      }
//      System.out.println();
    }
    //System.out.println("Sum error="+sumE);
    return true;
  }

  static boolean checkZero(short xCal[][])
  {
    for (int i=0; i<8; i++)
      for (int j=0; j<8; j++)
        if (xCal[i][j]!=0)
        { System.out.println("Error: Expect zero output.");  return false; }
    return true;
  }

  static void clipFDCT(short block[][])
  {
    for (int i=0; i<8; i++)
      for (int j=0; j<8; j++)
        if (block[i][j]<-2048)     block[i][j]=-2048;
        else if (block[i][j]>2047) block[i][j]= 2047;
  }

  static void clipIDCT(short block[][])
  {
    for (int i=0; i<8; i++)
      for (int j=0; j<8; j++)
        if (block[i][j]<-256)     block[i][j]=-256;
        else if (block[i][j]>255) block[i][j]= 255;
  }

  static void transformBlock(short b1[][], short b2[][], int trunc[][], int
offset[][][], int round[])
  {
    FDCT.fdct(b1);
    clipFDCT(b1);

    for (int i=0; i<8; i++)
      for (int j=0; j<8; j++)
        b2[i][j]=b1[i][j];

    IDCT.idct(b1);
    clipIDCT(b1);

    IDCT_Trunc.idctTrunc(b2,trunc,offset,round);
    clipIDCT(b2);
  }
```

```
  static boolean checkLH(long L, long H, boolean negatePixel, int trunc[][],
int offset[][][], int round[])
  {
    short b [][] = new short[8][8],
          b1[][] = new short[8][8],
          b2[][] = new short[8][8];
    int i, j, k;

    System.out.print("Check ["+L+","+H+","+negatePixel+"] ... ");

    // Initialize stat variables
    for (i=0; i<8; i++)
      for (j=0; j<8; j++)
        { e[i][j]=pmse[i][j]=pme[i][j]=0; }
    ome=omse=0;

    // Check 0 in 0 out
    for (i=0; i<8; i++)
      for (j=0; j<8; j++)
        b1[i][j] = 0;

    IDCT_Trunc.idctTrunc(b1,trunc,offset,round);
    if (!checkZero(b1)) return false;

    IEEE_Random.init(L,H);

    for (i=0; i<10000; i++) // Do for 10000 blocks
    {
//      System.out.println("------------------");
//      System.out.println("["+L+","+H+","+negatePixel+"] Block "+i+": ");

      for (j=0; j<8; j++)  // Generate random pixel data
        for (k=0; k<8; k++)
        {
          b1[j][k] = (short)IEEE_Random.rand();
          if (negatePixel) b1[j][k] =(short)-b1[j][k];
        }

      transformBlock(b1,b2,trunc,offset,round);

      if (!checkError(b1,b2))
      {
        System.out.println("Block="+i+", L="+L+", H="+H+",
Nagate="+negatePixel);
        return false;
      }
    }
    long max;
    int  percent;
    System.out.print("PASSED: pme(max)=");
    for (i=0,max=0; i<8; i++)
      for (j=0; j<8; j++)
        if (pme[i][j]>max) max=pme[i][j];
    percent = (int)((max*100.0)/pmeMAX);
    System.out.print((max/10000.0)+"("+percent+"%), psme(max)=");
    for (i=0,max=0; i<8; i++)
      for (j=0; j<8; j++)
        if (pmse[i][j]>max) max=pmse[i][j];
    percent = (int)((max*100.0)/pmseMAX);
    System.out.print((max/10000.0)+"("+percent+"%), ");
    percent = (int)((ome*100.0)/omeMAX);
    System.out.print("ome="+(ome/(64*10000.0))+"("+percent+"%), ");
```

```
    percent = (int)((omse*100.0)/omseMAX);
    System.out.print("omse="+(omse/(64*10000.0))+"("+percent+"%)");
    System.out.println();
    return true;
  }

  static int getInt(StreamTokenizer s) throws Exception
  {
    for (int token=s.nextToken(); token!=s.TT_NUMBER; token=s.nextToken());
    return (int)s.nval;
  }

  static void initSetup(int trunc[][], int offset[][][], int round[]) throws
Exception
  {
    int d, i, j;

    StreamTokenizer setup = new StreamTokenizer(new FileReader("Setup.txt"));

    int prec[] = new int[4];
    for (i=0; i<4; i++)
      prec[i] = getInt(setup);          //Precision

    IDCT_Trunc.init_IDCT_Trunc(prec);

    for (d=0; d<2; d++)
      for (i=0; i<4; i++)
        trunc[d][i] = getInt(setup);

    for (d=0; d<2; d++)
    {
      for (i=0; i<5; i++)
        for (j=0; j<8; j++)
          offset[d][i][j] = getInt(setup);
      round[d] = getInt(setup);
    }
  }

  public static void main(String args[]) throws Exception
  {
    int trunc[][]    = new int[2][4],
        offset[][][] = new int[2][5][8],
        round[]      = new int[2];
    int L[]={300,256,5},
        H[]={300,255,5};
    boolean negate=false;
    int i, j, k, prec;

    initSetup(trunc,offset,round);

    for (i=0; i<2; i++, negate=!negate)
      for (j=0; j<3; j++)
        if (!checkLH(L[j],H[j],negate,trunc,offset,round))
          return;
    System.out.println("All test passed!");
  }
}
```

IEEE_Random.java

```
public class IEEE_Random
{
  static long   randx, L, H;
  static double z;

  public static void init(long l, long h)
  {
    randx = l;
    z     = Double.longBitsToDouble(0x7fffffff);
    L=l;  H=h;
  }

  public static long rand()
  {
    long   i, j;
    double x;

    randx = (randx * 1103515245) + 12345;
    i = randx & 0x7ffffffe;
    x = (Double.longBitsToDouble(i))/z;
    x*= (L+H+1);
    j = (long)x;
    return j-L;
  }

  public static void main(String args[])
  {
    long l, h, n;
    n = Long.parseLong(args[0]);
    l = Long.parseLong(args[1]);
    h = Long.parseLong(args[2]);

    init(l,h);
    for (int i=1; i<=n; i++)
    {
      System.out.print(rand()+", ");
      if (i%8==0) System.out.println();
    }
    System.out.println();
  }
}
```