Very large scale integration architectures of discrete transforms for digital signal processing applications

Wang, Chao

2008

Wang, C. (2008). Very large scale integration architectures of discrete transforms for digital signal processing applications. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/41410>

<https://doi.org/10.32657/10356/41410>

Very Large Scale Integration Architectures of Discrete Transforms for Digital Signal Processing Applications

Wang Chao

School of Electrical and Electronic Engineering

A thesis submitted to the Nanyang Technological University
in fulfilment of the requirement for the degree of
Doctor of Philosophy

2008

*To my dear parents,
and my brother,
for their love and encouragement.*

Acknowledgments

Acknowledgments

I wish to express my appreciation and gratitude to my research supervisors Associate Professor Gan Woon Seng and Dr. Tong Yit Chow of the Nanyang Technological University (NTU), for their guidance, advices, encouragement, enthusiasm and support throughout this research programme.

I am grateful to the staff and other researchers at the Digital Signal Processing Lab, NTU for their technical support and for providing a pleasant, active and effective academic environment at various stages of this research. I would also like to express my special thanks to the staff and other researchers at the Integrated System Research Lab and the Centre for Integrated Circuits and Systems, NTU, especially Dr. Gu Jiangmin, Dr. Li Juhui, Chong Kwen Siong, Shao Yu, Chua Eng Kee, Dr. Xu Pengfei, Luo Jianwen, Associate Professor Jong Ching Chuen, and Associate Professor Chang Chip Hong, for their friendship, help, discussions, suggestions and encouragement.

I do also appreciate the support and help from my colleagues in the FTD Solutions Pte Ltd and in the Centre for Signal Processing, NTU. Especially, I would like to thank Qian Yue, Lu Dawei, Yang Fan, and Dr. Yu Zhuliang for their valuable comments, suggestions and criticisms.

I have enjoyed the company and friendship of all of my friends in Singapore. I would like to thank them for making the past five years an unforgettable and invaluable experience in my life.

Last, I wish to express my deepest gratitude to my dear parents, my dear brother, and my lovely four year old nephew. It is indeed their deep love and consistent encouragement that has kept me focused, determined, cheered up, and never to say give up.

Summary

Summary

Many discrete transforms, such as the fast Fourier transform (FFT), critical-band transform (CBT), and discrete wavelet packet transform (DWPT), have been successfully used in a wide range of digital signal processing (DSP) applications. However, these discrete transform algorithms are always computational and memory intensive, which may result in a performance bottleneck, increase hardware implementation cost, or lead to significant power consumption.

The purpose of this research is to develop new architectures for the FFT, CBT, and DWPT, suitable for very large scale integration (VLSI) implementations. This research concentrates on the design of efficient VLSI architectures for these discrete transforms, either to increase throughput, minimize area, reduce power consumption, or to achieve a combination of these factors.

FFT transforms the signal samples from time domain to frequency domain with a constant frequency resolution. In this research, a recursive architecture is developed to use a single butterfly unit to compute a 256-point radix-2 decimation-in-time (DIT) FFT for low-cost, low-power speech and audio applications. As the butterfly unit consumes most of the execution time and power in the FFT computation, we propose a two-multiplier and three-adder butterfly unit to achieve a good trade-off between hardware cost and throughput. The recursive FFT architecture achieves low computational cost by using the proposed butterfly unit, and obtains a significant power reduction by applying the supply voltage scaling technique and pipeline processing.

However, psychoacoustic studies show that human ear performs spectral analysis on the acoustic signal in the form of a filterbank with nonuniform critical bandwidths. Therefore, we also develop an efficient algorithm and its corresponding architecture

Summary

for the CBT to approximate the critical-band filtering of the human ear. The proposed CBT consists of a constant-bandwidth transform in the lower frequency range and a Brown CQT in the higher frequency range. The proposed VLSI architecture achieves significant power-efficiency by reducing the computational complexity, using parallel and pipeline processing, and applying the supply voltage scaling technique. In comparison with other methods, the proposed CBT method fits the human ear critical-band analysis better with significantly fewer computations, and therefore, is more energy-efficient and well suited for low-power speech applications.

As a generalization of the discrete wavelet transform, the DWPT has the characteristic of flexible time and frequency decomposition, which has advantages over FFT and CBT in many DSP applications. By exploiting the in-place nature of the DWPT algorithm, we develop three VLSI architectures for computing the multilevel lifting-based DWPT. First, we propose a recursive architecture using a single processing element (PE) to compute the DWPT recursively for low-cost applications. Second, a pipeline architecture is developed for high-speed, high-level DWPT decomposition. By employing a folded PE for wavelet filters, this architecture computes the DWPT in a fashion of high-throughput pipeline processing by interleaving multiple groups of butterfly calculations at each level. This pipeline architecture has excellent features of fewer memory accesses, hardware regularity and simplicity, low hardware complexity, high throughput and good scalability. Third, a folded architecture is derived from the proposed pipeline architecture for some low-cost applications where high speed is not required. The hardware complexity required is further reduced significantly by folding the multilevel decomposition into a single configurable PE. The processing time for a DWPT is still much shorter than the recursive architecture.

To evaluate the efficiency of the proposed architectures, performance analysis and comparison is done in terms of computational complexity, memory accesses,

Summary

hardware complexity, hardware utilization, and processing cycles, etc. Some case studies for the proposed architectures are designed and simulated by the hardware description languages Verilog-HDL or VHDL. A number of implementations targeted towards the AMS 0.35 μm CMOS technology or TSMC 0.18 μm CMOS technology, as well as the Xilinx Virtex-II series FPGA have been produced to validate these architectures' efficiency.

Table of Contents

Table of Contents

Acknowledgements	i
Summary	ii
List of Figures	ix
List of Tables	xiii
Nomenclature	xv
1. Introduction	1
1.1 Motivation	1
1.2 Objectives	5
1.3 Major Contributions of the Thesis	7
1.4 Organisation of the Thesis	10
2. Literature Review	13
2.1 Introduction	13
2.2 VLSI Design for DSP Applications	15
2.2.1 DSP VLSI Design Principles	15
2.2.2 Algorithm and Architecture Level Transformations	20
2.3 Low-Power DSP VLSI Design	23
2.3.1 Sources of Power Consumption in CMOS Circuits	23
2.3.2 Basic Principles for Low-Power Design	26
2.3.3 Low-Power Techniques for DSP VLSI Design	29

Table of Contents

2.4	Human Perception Systems	34
2.4.1	Human Auditory System	34
2.4.2	Human Visual System	38
2.5	Front-End Spectral Analysis Transforms	40
2.5.1	Discrete Fourier Transform	40
2.5.2	Constant-Q Transform	43
2.5.3	Discrete Wavelet Transform	44
2.6	Conclusions	48
3.	A Low-Cost Fast Fourier Transform Processor	49
3.1	Introduction	49
3.2	Review of Existing FFT Architectures	52
3.2.1	Recursive/Sequential FFT Architecture	52
3.2.2	Pipeline/Cascade FFT Architecture	55
3.2.3	Parallel FFT Architecture	56
3.3	The Proposed Recursive FFT Architecture	57
3.3.1	Review of Existing Butterfly Units	59
3.3.2	The Proposed Butterfly Unit	65
3.3.3	The Proposed FFT Architecture	69
3.4	Simulation and Implementation Results	75
3.5	Conclusions	82
4.	Algorithm and Architecture of Critical-Band Transform	84
4.1	Introduction	84
4.2	Review of Existing Critical-Band Analysis Methods	87

Table of Contents

4.3	The Proposed Algorithm and Architecture of Critical-Band Transform	90
4.3.1	Critical-Band Transform Algorithm	90
4.3.2	Performance Analysis	95
4.3.3	Critical-Band Transform Architecture	101
4.3.4	Computational Complexity	109
4.4	Simulation and Implementation Results	111
4.5	Conclusions	115
5.	VLSI Architectures for Discrete Wavelet Packet Transform	118
5.1	Introduction	118
5.2	Review of Existing VLSI Architectures for DWPT	122
5.2.1	Recursive DWPT Architecture	125
5.2.2	Folded DWPT Architecture	127
5.2.3	Pipeline DWPT Architecture	128
5.2.4	Parallel DWPT Architecture	130
5.3	The Proposed VLSI Architectures for Lifting-Based DWPT	131
5.3.1	The In-Place DWPT Algorithm	131
5.3.2	The Proposed Recursive DWPT Architecture	136
5.3.2.1	Lifting-Based Wavelet Filter PE	138
5.3.2.2	Performance Evaluation	140
5.3.3	The Proposed Pipeline DWPT Architecture	143
5.3.3.1	The Proposed Pipeline DDC Architecture	144
5.3.3.2	The Proposed Folded PE for Wavelet Filters	147
5.3.3.3	Case Studies	152
5.3.3.4	Performance Evaluation	160

Table of Contents

5.3.4	The Proposed Folded DWPT Architecture	164
5.3.4.1	Proposed Configurable Folded PE	165
5.3.4.2	Performance Evaluation	168
5.4	Simulation and Implementation Results	170
5.5	Conclusions	177
6.	Conclusions and Future Work	180
6.1	Conclusions	180
6.2	Recommendations for Future Work	184
	Author's Publications	192
	Reference	194
	Appendices	205
A	Selected VHDL Codes for the Proposed Recursive FFT Processor	205
B	Selected VHDL Codes for the Proposed CBT Processor	208
C	Selected VHDL Codes for the Proposed DWPT Processors	212

List of Figures

List of Figures

1.1	Design productivity gap published in 1997.	5
2.1	(a) A simple DSP algorithm that computes $y(n) = a(n) + b(n) + c(n)$; (b) A folded architecture for the algorithm in Figure 2.1 (a).	22
2.2	Dynamic power consumption of a CMOS inverter	25
2.3	Power efficiency of a pipelined processing system	30
2.4	Clock gating	31
2.5	Human auditory system	35
2.6	Threshold in quiet and masking threshold. Acoustical events in the gray areas will not be audible.	36
2.7	Human visual system	39
2.8	SFG of an 8-point radix-2 DIT FFT	42
2.9	A 3-level DWT decomposition using the pyramid algorithm	46
2.10	A 3-level DWT decomposition using the lifting scheme	48
3.1	Conventional recursive FFT architectures: (a) single-butterfly architecture with single memory; (b) single-butterfly architecture with two memories; (c) single-butterfly architecture with cache memory.	53
3.2	Conventional pipeline FFT architectures: (a) MDC scheme; (b) SDF scheme.	55
3.3	Conventional parallel FFT architecture	57
3.4	SFG of radix-2 DIT butterfly	59
3.5	Conventional radix-2 DIT pipelined butterfly unit: (a) pipelined	60

List of Figures

	butterfly unit structure; (b) pipeline diagram.	
3.6	SFG of the full-parallel butterfly unit	61
3.7	Architecture of the 1-multiplier and 2-adder butterfly unit	62
3.8	(a) SFG and (b) architecture of the 2-multiplier and 4-adder butterfly unit	64
3.9	(a) Rescheduled SFG of the radix-2 butterfly; (b) scheduling of the butterfly operations.	66
3.10	Architecture of the proposed pipelined butterfly unit	68
3.11	Block diagram of the proposed single-butterfly FFT processor	70
3.12	Block diagram of the FFT address generator	72
3.13	Block diagram of the FFT controller	74
4.1	Degree of Approximation to the Bark scale	96
4.2	(a) First 600-ms of “ka”; (b) second 600-ms of “ka”.	98
4.3	(a) CBT analysis of the first 600-ms of “ka”; (b) CBT analysis of the second 600-ms of “ka”.	99
4.4	(a) FFT analysis of the first 600-ms of “ka”; (b) FFT analysis of the second 600-ms of “ka”.	100
4.5	Data flow graph of the CBT algorithm	102
4.6	Block diagram of the proposed CBT processor	103
4.7	Proposed pipelined CBT Data Path	104
4.8	Pipeline diagram of the proposed CBT data path	107
4.9	Block diagram of the CBT address generator	108
5.1	A 3-level DWPT full decomposition	123
5.2	Various DWPT architectures: (a) Single-PE recursive architecture;	125

List of Figures

	(b) Single-PE folded architecture; (c) Pipeline architecture; (d) Parallel architecture.	
5.3	Trenas's pipelined DWPT architectures: (a) word-serial implementation; (b) word-parallel implementation.	129
5.4	Arguello's Parallel DWPT Architecture	130
5.5	A complete 3-level DWPT implemented by tree-structured filterbank	133
5.6	Proposed SFG of a 3-level full-decomposition DWPT	134
5.7	Block diagram of the proposed recursive DWPT architecture	137
5.8	Direct implementation of lifting-based D4 wavelet filter	140
5.9	Parallel architecture of convolution-based D4 wavelet filter	142
5.10	Block diagram of the DDC architecture	144
5.11	DDC architecture for 3-level DWPT and its data-flow graph	146
5.12	Proposed folded architecture of the lifting-based D4 filter with internal pipelining	149
5.13	Folded PE of the lifting-based D4 filter at 3 rd level with internal pipelining	151
5.14	Folded architecture of lifting-based 5/3 wavelet filter where $n = 2^j$ at level j	155
5.15	Folded architecture of lifting-based 9/7 wavelet filter where $n = 2^j$ at level j	158
5.16	Folded DDC architecture for 3-level DWPT	166
5.17	Configurable PE in the folded DDC architecture for 3-level D4 DWPT	166
5.18	(a) Configurable folded delay registers in the configurable PE; (b)	168

List of Figures

	Generalized folded DDC architecture for J -level DWPT.	
5.19	Direct-mapped architecture for 3-level DWPT	172
6.1	Proposed folded architecture of the lifting-based D4 filter with bypass mode	187

List of Tables

List of Tables

2.1	Critical bandwidths of human auditory system	37
3.1	Pipeline table of the proposed butterfly unit	68
3.2	Hardware requirement and timing performance of different butterfly unit architectures	69
3.3	Estimated processing time for different-sized FFT processors at 100 MHz	77
3.4	(a) Estimated area size of the proposed butterfly unit.	77
3.4	(b) Estimated area size of the two-multiplier & four-adder benchmark.	78
3.5	Power dissipation simulation results of the 16-point 16-bit FFT processor	78
3.6	Comparisons of different FFT implementations	81
4.1	Comparison of the parameters in CBWT and CQT	94
4.2	Bandwidths and center frequencies of 21-band CBT and the Bark scale	96
4.3	Pipeline table of the CBT data path: (a) First two butterfly operations for each critical band; (b) Last CBT butterfly operation when window size $N[k_{cb}]$ is odd; (c) Last CBT butterfly operation when window size $N[k_{cb}]$ is even.	105
4.4	Comparison of computational complexity with other transforms	110
4.5	Comparison of memory access between the FFT processor and CBT processor	111

List of Tables

4.6	Power distribution of different functional units in the CBT processor	113
4.7	CBT processor power dissipation simulation results	114
4.8	Comparison of the FFT and CBT processors	115
5.1	Address generation for the butterfly inputs in a 3-level 8-point DWPT	137
5.2	Computational complexity of lifting-based DWPT versus convolution-based DWPT	141
5.3	Comparisons of several recursive architectures for J -level D4 DWPT	143
5.4	Data flow for the D4 PE at the 1 st level in Figure 5.11	148
5.5	Data flow for the folded D4 PE at the 2 nd level in Figure 5.11	150
5.6	Data flow for the folded D4 PE at the 3 rd level in Figure 5.11	152
5.7	Data flow for the folded 5/3 PE at the 2 nd level in Figure 5.11	156
5.8	Comparisons of the DDC architectures for different wavelet filters	158
5.9	Comparisons of several architectures for 3-level D4 DWPT	161
5.10	Comparisons of the folded DDC architecture with several architectures	169
5.11	Timing and hardware complexity comparisons of the proposed DWPT architectures	174
5.12	Estimated timing and area size of the four DWPT architectures	176
5.13	Estimated area size of the DDC architecture for D4 DWPT	176
5.14	Comparisons of different D4 DWPT hardware implementations	177

Nomenclature

Nomenclature

Symbols

A	The first input of the radix-2 butterfly
B	The second input of the radix-2 butterfly
C	The first output of the radix-2 butterfly
$d_{j,i}(l)$	The high-pass filter's coefficients in the i th frequency subband at level j of multilevel DWPT algorithm
$d^k(l)$	The k th intermediate high-pass coefficients during the lifting process of a wavelet filter
$dk_{j,i}(l)$	The k th intermediate high-pass coefficients during the lifting process of a wavelet filter in the i th frequency subband at level j
$d[n]$	The high-pass (wavelet) coefficient of the low-pass/high-pass filter pair in the DWT and DWPT algorithms
D	The second output of the radix-2 butterfly
f_{clk}	The operating clock frequency
$f_{k_{cb}}$	The k_{cb} th center frequency in the CBT algorithm
$f_{k_{cq}}$	The k_{cq} th center frequency in the Brown CQT algorithm
$f_{k_{cw}}$	The k_{cw} th center frequency in the CBWT
f_{minc}	The minimum center frequency in the lower frequency range of the CBT algorithm
f_{minq}	The minimum center frequency in the higher frequency range of the CBT algorithm

Nomenclature

$g[k]$	The impulse response function of the high-pass filter in the DWT and DWPT algorithms
$g[z]$	The transfer function of the high-pass filter in the DWT and DWPT algorithms
$G_{i,k}$	The low-pass and high-pass filtering of the k th pair of samples in the i th subband (or the k th butterfly in the i th group) at a given level in the in-place DWPT decomposition
$h[k]$	The impulse response function of the low-pass filter in the DWT and DWPT algorithms
$h[z]$	The transfer function of the low-pass filter in the DWT and DWPT algorithms
I_{diode}	The reverse-bias current in digital CMOS circuits
$I_{leakage}$	The leakage currents in digital CMOS circuits
I_{sc}	The short-circuit current in digital CMOS circuits
$I_{subthreshold}$	The subthreshold current in digital CMOS circuits
j	The level number of a J -level DWPT decomposition
J	The number of total levels in a multilevel DWPT decomposition
$kern[k_{cb}, n]$	The coefficient of the CBT algorithm
L	The number of taps in a FIR filter
M	The number of unit-delay elements in the direct implementation of a lifting-based wavelet filter
$M[k_{cb}]$	The number of CBT butterfly operations for the $k_{cb}th$ CBT spectral component
n_{even}	The number of the critical bands which has even window sizes

Nomenclature

n_{cb}	The number of critical bands in the CBT algorithm
n_{cq}	The number of critical bands in the Brown CQT (or the higher range in the CBT algorithm)
n_{cw}	The number of constant bandwidths (or critical bands) in the CBWT (or the lower range in the CBT algorithm)
n_{odd}	The number of the critical bands which have odd window sizes
N_c	The length of the window function in the CBWT
$N[k_{cb}]$	The length of the k_{cb} th window function in the CBT algorithm
$N[k_{cq}]$	The length of the k_{cq} th window function in the Brown CQT algorithm
P	The prediction function in the lifting
$P(z)$	The polyphase matrix for the low-pass and high-pass filter pair in the DWT and DWPT algorithms
P_{avg}	The average power in digital CMOS circuits
$P_{dynamic}$	The dynamic power in digital CMOS circuits
$P_{leakage}$	The leakage power in digital CMOS circuits
$P_{short-circuit}$	The short-circuit power in digital CMOS circuits
P_{static}	The static power in digital CMOS circuits
$P_{sw.cap}$	The capacitive switching power in digital CMOS circuits
Q	The Q factor, defined as the ratio of the center frequency f to the frequency resolution Δf
Q_{cq}	The Q factor in the Brown CQT algorithm
$Q_{k_{cb}}$	The Q factor in the CBT algorithm
$Q_{k_{cw}}$	The Q factor in the CBWT
$s_i(z)$	The Laurent polynomials in factorization of the polyphase matrix for a

Nomenclature

	wavelet filter
$s_{j,i}(l)$	The low-pass filter's coefficients in the i th frequency subband at level j of multilevel DWPT algorithm
$s^k(l)$	The k th intermediate low-pass coefficients during the lifting process of a wavelet filter
$sk_{j,i}(l)$	The k th intermediate low-pass coefficients during the lifting process of a wavelet filter in the i th frequency subband at level j
$s[n]$	The low-pass (scaling) coefficient of the low-pass/high-pass filter pair in the DWT and DWPT algorithms
S	The splitting function in the lifting
SR	The sampling frequency
$t_i(z)$	The Laurent polynomials in factorization of the polyphase matrix for a wavelet filter
T_a	The time required for performing one addition
T_d	The propagation delay of a CMOS gate
T_m	The time required for performing one multiplication
U	The updating function in the lifting
V_{dd}	The digital supply voltage
V_T	The threshold voltage of a MOS transistor
$w_{j,i}(n)$	The i th subband signal (or n th wavelet coefficient in the i th group) at a given level j in the multilevel DWPT decomposition
$w[n]$	The window function in the STFT algorithm
$w[k_{cb}, n]$	The k_{cb} th window function with a length $N[k_{cb}]$ in the CBT algorithm
$w[k_{cq}, n]$	The k_{cq} th window function with a length $N[k_{cq}]$ in the Brown CQT

Nomenclature

	algorithm
W_N	The twiddle factor of an N -point FFT algorithm
$x[n]$	The input data sample
$X[k]$	The k th FFT spectrum
$X[k_{cb}]$	The k_{cb} th CBT spectral component
$X[k_{cq}]$	The k_{cq} th CQT spectral component
$X[k_{cw}]$	The k_{cw} th CBWT spectral component
Δf	The frequency resolution
Δf_c	The frequency resolution in the CBWT
$\Delta f_{k_{cq}}$	The k_{cq} th critical bandwidth in the Brown CQT algorithm
$\Delta f_{k_{cb}}$	The k_{cb} th critical bandwidth in the CBT algorithm
Δf_{minq}	The minimum bandwidth in the higher frequency range of the CBT algorithm

Abbreviations

1-D	One-dimensional
2-D	Two-dimensional
ASIC	Application-specific integrated circuit
AMS	Austria Mikro System International AG
BU	Butterfly unit
CBT	Critical-band transform
CBWT	Constant-bandwidth transform
CMOS	Complementary metal oxide semiconductor

Nomenclature

CQT	Constant-Q transform
DAB	Digital audio broadcasting
DC	Direct current
DDC	Double-path delay commutator
DFF	D-flip-flops
DFT	Discrete Fourier transform
DIF	Decimation-in-frequency
DIT	Decimation-in-time
DM	Direct-mapped
DSP	Digital signal processing
DVB	Digital video broadcasting
DWPT	Discrete wavelet packet transform
DWT	Discrete wavelet transform
D4	Daubechies-4
D6	Daubechies-6
EDA	Electronic design automation
FBI	Federal Bureau Investigation
FFT	Fast Fourier transform
FIFO	First-in first-out
FIR	Finite impulse response
FPGA	Field programmable gate array
FSM	Finite state machine
HDL	Hardware description language
HDTV	High-definition television

Nomenclature

IC	Integrated circuit
I/O	Input and output
IP	Intellectual property
ISE	Integrated Software Environment
JPEG2000	Joint Photographic Experts Group 2000
LUT	Look-up table
LSI	Large scale integration
MAC	Multiplier-accumulator
MDC	Multipath delay commutator
MIMO	Multiple-input-multiple-output
MPEG-4	Moving Picture Experts Group-4
NMOS	n-channel metal oxide semiconductor
OFDM	Orthogonal frequency division multiplexing
PE	Processing element
PLD	Programmable logic device
PMOS	p-channel metal oxide semiconductor
RAM	Random access memory
ROM	Read only memory
RPA	Recursive pyramid algorithm
RTL	Register transfer level
R/W	Read and write
R2MDC	Radix-2 MDC
R2SDF	Radix-2 SDF
SDF	Single-path delay feedback

Nomenclature

SFG	Signal flow graph
SNR	Signal-to-noise ratio
SoC	System-on-chip
STFT	Short-time Fourier transform
SW	Switch
TF	Time and frequency
TSMC	Taiwan Semiconductor Manufacturing Company Ltd
UWB	Ultrawideband
VHDL	Very high speed integrated circuit hardware description language
VLSI	Very large scale integration
WPT	Wavelet packet transform

Chapter 1

Introduction

1.1 Motivation

The field of digital signal processing (DSP) has developed rapidly over the past three decades. Because it has many advantages over analog signal processing [1], [2], DSP has almost replaced the role of conventional analog systems in a wide range of applications, such as speech processing, biomedical signal processing, control systems, graphics/imaging, video, terrestrial television, and telecommunications [3]. Due to the advances in the DSP technology, numerous new interesting research and application areas have emerged. Some important examples include speech enhancement and recognition, image and video coding, digital video broadcasting, and wireless multiple access communications. The field of DSP is not only fuelled by the advances in DSP applications but also driven by the rapid development of the very large scale integration (VLSI) technologies.

One of the most significant technological breakthroughs in the 20th century is probably the invention of transistor in 1940s [4], which has greatly changed the way we live in the past fifty years. Especially after the subsequent invention of integrated circuits (ICs) in 1950s, the advances in silicon technology has begun to fuel the third industrial revolution. The evolution of silicon technology has followed the famous Moore's Law [4], named after Gordon Moore of Intel, who predicted in 1960s that the number of transistors on a single IC chip would double every 18 months. In the past

forty years, silicon technology has evolved from the IC technology, the large scale integration (LSI) technology, to the VLSI technology, while the number of transistors integrated in a single chip has grown from a few hundreds, to over ten thousands. To date, current generation of silicon technology still follows closely to the Moore's Law. With the explosive development of the complementary metal oxide semiconductor (CMOS) technologies where the minimum feature size of the current mature process is shrinking from 0.35 μm to 65 nm, tens of millions of transistors can be easily integrated into a single IC chip. It is predicted that by the end of 2010s, it will be possible to manufacture a system-on-chip (SoC) containing 4 billion transistors by using the 50 nm silicon technology [5].

Advances in the VLSI technologies have made significant improvements in terms of increased integration of transistors, increased speed of operations, and increased area of IC wafers. Increased integration has made it possible to implement DSP algorithms of increased complexity on a single chip, while increased speed of operations has made high-throughput real-time DSP applications possible. Another benefit from the advances in the VLSI technologies is that widespread availability of sophisticated high-speed DSP systems will become a reality because dramatic reduction in cost has been made by increased area of IC wafers.

The availability of low-cost, high-density, and high-speed VLSI devices and advanced electronic design automation (EDA) tools make the design, implementation and application of single-chip DSP systems possible. However, modern DSP applications still impose many challenges on the VLSI implementation of DSP systems.

Wireless communications and multimedia signal processing systems have come into widespread use over past ten years. Most DSP algorithms in these applications feature high throughput, intensive computations, intensive memory accesses, and high algorithm complexity. In traditional DSP systems design where high-speed real-time

operation is required, the primary design goal is to meet the throughput requirement. Once the timing requirement is satisfied, silicon area can be minimized to reduce costs. However, with the explosive growth of consumer electronic market, especially the dramatic demand for battery-powered portable consumer products, from low-performance products such as pocket calculators, wrist watches, smart toys, and portable healthcare devices, etc, to high-performance products such as cellular phones, portable digital assistants, digital video cameras, and portable multimedia terminals, etc, power dissipation becomes another important consideration in order to extend battery lifetimes. Moreover, in high-throughput applications, many DSP algorithms feature intensive computations and memory accesses, leading to massive power consumption. Lowering power dissipation can not only reduce the system costs associated with special cooling and packaging, but also enhance the system reliability [6]. Therefore, low-power design has become another challenge in the VLSI implementation of DSP systems.

On the other hand, more and more consumer electronic products, especially battery-powered portable products, for example, smart phones, portable game players, and portable digital assistants, are required to handle multimedia information, such as speech, text, image, and video, and also expected to have even more powerful capabilities and new functionalities in the future. These increased demands for higher performance, better quality services, versatile functionalities, and intelligence, have further increased the system/algorithm complexity and computation capability, which in turn leads to both high hardware complexity and high power dissipation. Therefore, advanced DSP algorithms and efficient VLSI architectures are required to reduce the hardware complexity, as well as the power dissipation.

Another challenge in the VLSI implementation of DSP systems is that DSP algorithms are always used in a wide spectrum of real-time applications with different sampling rates ranging from about 20 kHz in speech applications to over 500 MHz in

radar, high-definition television (HDTV), and high-speed digital communications. Even in a video compression system for HDTV, the computation requirement can range from 10 to 100 gigaoperations per second [1]. The vast difference in sample rate and computation requirements requires different architecture considerations for implementation of DSP algorithms. A speech application may prefer a time-multiplexed architecture, for example, a recursive architecture, which uses a single processing element (PE) to compute DSP algorithm operations recursively, by mapping many algorithm operations into the same hardware. However, in a very high-throughput video application, only a direct-mapped architecture can meet the high speed requirement by one-to-one mapping of algorithm operations to processing elements. Thus, it is very important to design a family of dedicated architectures, which can accommodate a broad range of DSP applications with different requirements.

While the VLSI technologies grow rapidly, the design productivity gap, i.e. the difference in the growth rate of the feasible number of transistors integrated in an IC chip with the growth rate of the number of transistors on an IC chip designed by a staff in one month has widen. Figure 1.1 illustrates the design productivity is always lagging behind the VLSI technologies and the gap is widening [4]. The graph shows that in the industry history, there is only a 21% annual growth rate in design productivity, while the annual growth rate in technology progress is 58% [4], [7]. Studies on efficient VLSI architectures for DSP algorithms therefore are very crucial to meet these design challenges in the single chip implementation of complex DSP systems.

Figure 1.1 Design productivity gap published in 1997. [4]

1.2 Objectives

Some discrete transforms, such as the fast Fourier transform (FFT), critical-band transform (CBT) and discrete wavelet packet transform (DWPT) have played a crucial role in a wide range of DSP applications, including multimedia systems and digital communications. These three discrete transforms perform the transformation of signals from one domain to another domain, with a constant frequency resolution, a nonuniform frequency resolution, and a flexible time and frequency decomposition, respectively. The wide applications of these three discrete transforms necessitate the studies to develop efficient architectures of these transform algorithms, suitable for VLSI implementation.

The preliminary goal of the present study is to investigate and develop efficient architectures for the FFT, CBT, and DWPT, which are suitable for VLSI implementation. The in-nature properties of these discrete transform algorithms, such as regularity, recursiveness, coefficient symmetry, and parallelism/pipelining, etc, are effectively exploited at the algorithm/architecture level design. Algorithm/architecture level transformation techniques [1], such as parallelism/pipelining, strength reduction,

and folding, etc, are explored and used in the design of efficient VLSI architectures for these discrete transforms, either to increase throughput, minimize area, reduce power dissipation, or a combination of these factors, for a variety of dedicated DSP applications.

Along with speed and area size, power dissipation becomes one of important design metrics in DSP VLSI design. In this research, low-power design techniques at algorithm/architecture level are explored and used to meet the throughput requirement, and at the same time to achieve low-power operation. This research project also aims at providing an insight into design trade-offs between speed and area metrics, and studying a variety of VLSI architectures for a wide range of different DSP applications.

The objective of our research is to design new VLSI architectures for the three discrete transforms for different DSP applications with primary emphases on:

- Real-time processing to meet the particular throughput requirement
- Low-power operation
- Cost efficiency
- Modularity and regularity

Evaluation on the efficiency of our proposed architectures is done by performance analysis and comparison in terms of computational complexity, memory accesses, hardware complexity, hardware utilization, and processing cycles. Some case studies targeted towards the AMS 0.35 μm CMOS technology or TSMC 0.18 μm CMOS technology, as well as the Xilinx Virtex-II series FPGA are examined to validate these VLSI architectures' efficiency with convincing simulation and implementation results.

1.3 Major Contributions of the Thesis

In this thesis, we make three major contributions pertaining to the investigation and development of efficient VLSI architectures for the three different discrete transform with a constant frequency resolution, a nonuniform frequency resolution, and a flexible time and frequency decomposition, respectively. This research provides efficient VLSI solutions for these discrete transforms with different transform resolutions for a variety of dedicated DSP applications. We will now describe these three contributions individually.

In our first contribution, we propose a recursive FFT architecture for low-cost, low-power speech and audio applications. By exploiting the in-nature recursiveness of the decimation-in-time (DIT) FFT algorithm, the proposed FFT architecture uses a single butterfly unit to compute a 256-point radix-2 DIT FFT recursively. As a crucial part in the FFT computation, the butterfly unit consumes most of the execution time and power. Therefore, we propose a two-multiplier and three-adder pipelined butterfly unit to achieve a good trade-off between processing speed and hardware cost. Circuit simulation and implementation results show that this recursive FFT architecture achieves an effectively low cost by using the proposed butterfly unit, and obtains a significant power reduction by applying the supply voltage scaling technique.

In our second contribution, we develop an efficient algorithm and its corresponding VLSI architecture of the CBT for low-power speech applications. With a nonuniform frequency resolution, the CBT is developed to approximate the critical-band filtering of the human ear. From the observation of the critical-band scale of the human ear, we formulate a generalized CBT algorithm, which consists of a constant-bandwidth transform in the lower frequency range and a Brown constant-Q transform (CQT) in the higher frequency range. By finding a set of suitable parameters, the CBT can achieve a close fitting to a particular critical-band scale, such as the Bark scale or

the Mel scale. A 21-band Bark scale CBT with a sampling rate of 16 kHz is designed and simulated to evaluate the efficiency of the proposed algorithm. Simulation results demonstrate its suitability for performing short-time spectral analysis on speech. Comparative studies also show that the proposed 21-band CBT has a better fitting to the human critical-band analysis, significantly fewer computations, fewer memory accesses and therefore, is more energy-efficient than other methods.

By exploiting the resursiveness, parallelism, and regularity in the proposed CBT algorithm, we propose an efficient VLSI architecture with a pipelined data path to calculate the CBT recursively. The regular butterfly-structured data path is derived by exploiting the symmetry property of the CBT coefficients. The coefficient symmetry property reduces the arithmetic operations and coefficient memory area by up to 50%. The proposed architecture achieves significant power efficiency by reducing the computational complexity and memory accesses, using parallel and pipeline processing, and applying the supply voltage scaling technique. A 21-band Bark scale CBT processor based on the proposed architecture is designed by VHDL (very high speed integrated circuit hardware description language) and synthesized by Synopsys Design Compiler. With a 0.35 μm CMOS technology, the CBT processor calculates a 160-point speech in 4.99 ms at a clock rate of 234 kHz. Transistor-level simulation results by Synopsys NanoSim show that the power dissipation is 15.6 μW at 1.1 V. The CBT processor achieves 82.1% power reduction as compared to the benchmark 256-point FFT processor.

In our third contribution, we develop efficient VLSI architectures for computing multilevel lifting-based DWPT. Due to the characteristics of flexible time and frequency decomposition, the DWPT has advantages over the FFT and CBT in a wide spectrum of DSP applications. In this research, we formulate the in-place DWPT algorithm in a regular signal flow graph by using the butterfly and group concepts. By exploiting the in-place nature of the DWPT algorithm, we propose three DWPT

architectures, providing a flexibility to make design trade-offs between hardware cost and processing speed for different DSP applications.

Firstly, we propose a recursive DWPT architecture for low-cost applications. Featuring regularity and recursiveness, this architecture utilizes just one PE interacting with a memory bank to compute J -level DWPT through butterfly-by-butterfly, subband-by-subband, and level-by-level. With comparison to the direct-mapped (DM) tree-structured architecture, it achieves significantly low cost by reducing the hardware complexity from $(2^J - 1)$ PEs to only one PE.

Secondly, we develop a generalized pipeline DWPT architecture for high-speed DWPT decomposition. By exploiting the parallelism and regularity of the in-place DWPT algorithm, an efficient pipeline DWPT architecture is proposed to implement high-throughput processing without any on-chip memory to buffer the intermediate coefficients. In order to overcome the problem of high hardware complexity and low hardware utilization in the DM architecture, this pipeline DWPT architecture employs the folding technique to assign different groups of butterfly computations into one folded PE at each decomposition level. By using folded PEs, the pipeline architecture computes J -level DWPT with a high throughput by interleaving the multiple groups of butterfly calculations at each level. The proposed folded PE is applicable for all kinds of wavelet filters with different number of filter taps.

Comparative studies demonstrate that the proposed pipeline DWPT architecture outperforms most of the other reported architectures in terms of memory accesses, hardware regularity and simplicity, and throughput. Circuit simulation and implementation results also show that the proposed architecture not only achieves a significant reduction (from $(2^J - 1)$ PEs to J PEs) in hardware complexity but also maintains both the hardware utilization and high throughput in comparison with the DM architecture. The pipeline DWPT architecture with the folded wavelet filter PEs

is also scalable for J -level, N -point lifting-based DWPT using any L -tap wavelet filters.

Thirdly, we propose a folded DWPT architecture for some low-cost applications where high speed is not required. Derived from the above pipeline DWPT architecture, this folded architecture further reduces the hardware complexity significantly, by folding the multilevel decomposition into a single configurable PE. This folded DWPT architecture utilizes a single configurable PE with a configurable delay commutator, extra I/O (input and output) switches and feedback registers, to compute multilevel DWPT recursively. Compared with the proposed recursive DWPT architecture, this folded architecture does not require extra memory bank to buffer the intermediate data during the DWPT computation and therefore, avoids intensive memory operations too. Circuit simulation and implementation results show that the processing time for the proposed folded architecture to compute multilevel DWPT is much shorter than that by the recursive DWPT architecture, while the hardware cost of the two architectures are almost identical.

The preceding three contributions have led to the journal and conference paper publications listed in the author's publications at the end of this thesis.

1.4 Organisation of the Thesis

Six chapters are presented in this thesis.

In Chapter 1, we present the motivations and objectives of our research, followed by the contributions of the research.

In Chapter 2, we provide the background fundamentals and literature review of this research. We first address the VLSI design considerations for DSP applications, followed by a review of the algorithm/architecture level transformations used for DSP

VLSI design. Next, we study the mechanisms of power consumption in CMOS circuits, and discuss the basic principles of low-power CMOS circuit design. Following that, we review the low-power techniques used in CMOS VLSI design for DSP applications. We then introduce the fundamentals of the human perception models, which motivate the use of the discrete transforms in many DSP applications, especially in multimedia applications and digital communications. Finally, we introduce three popular discrete transforms, namely, the FFT, CQT, and discrete wavelet transform.

In Chapter 3, we focus on the design of the fast Fourier transform processor for speech and audio applications. We first review the existing FFT processor designs in different VLSI architectures. After different FFT architectures have been examined, we propose a recursive FFT architecture for low cost consideration. As a crucial part in the recursive FFT architecture, a butterfly unit is proposed for providing a good trade-off between cost and performance. Subsequently, the architecture and circuits design of the proposed 256-point FFT processor are presented with performance analysis. In order to achieve a low power dissipation, the supply voltage scaling technique is employed in the FFT processor design. Finally, the circuit simulation and implementation results for the proposed recursive FFT architecture are presented and analyzed.

In Chapter 4, we propose new algorithm and architecture for the critical-band transform for speech applications. The existing algorithms and implementations of critical-band analysis are explored and the main drawbacks are discussed. Then, we present a novel CBT algorithm based on the observation of the critical-band scale of the human ear. The performance of this algorithm is evaluated by comparing it with other popular CBT methods. Based on the proposed CBT algorithm, we propose an efficient VLSI architecture. The architecture and circuits design of the proposed CBT processor is described. In the design of the CBT processor, we apply low-power

design techniques, including the arithmetic operation and memory access reduction, parallel and pipeline processing, and supply voltage scaling technique. The circuit simulation and implementation results of the proposed CBT processor and the benchmark FFT processor are provided and compared at the end of Chapter 4.

In Chapter 5, we present our research on the development of VLSI architectures for the lifting-based discrete wavelet packet transform. We review the existing architectures for the DWPT. After the drawbacks of the existing DWPT architectures are studied, we introduce our three new DWPT architectures based on the in-place DWPT algorithm. The in-place DWPT algorithm is described by introducing the butterfly and group concepts. Firstly, we present a recursive DWPT architecture for low cost consideration. Secondly, we present a pipeline DWPT architecture featuring high throughput, low intensive memory accesses, low hardware complexity, high hardware utilization, hardware regularity and simplicity, and good scalability. Thirdly, in order to further reduce the hardware complexity, we present a folded DWPT architecture derived from the proposed pipeline architecture. Some case studies including 5/3 and 9/7 wavelet filters implementations are presented to illustrate the proposed DWPT architectures. Performance evaluation of the proposed architectures is also given. Eventually, the circuit simulation and implementation results are provided and discussed to validate the efficiency of the proposed DWPT architectures.

Chapter 6 consists of conclusions and recommendations for future work. The contributions made by the present study are also highlighted in this chapter. The potential relevant topics for further research are outlined.

Chapter 2

Literature Review

2.1 Introduction

As we have discussed in Chapter 1, the field of digital signal processing (DSP) has witnessed an explosive growth with fast advances in DSP applications. As the advanced VLSI technologies allow millions of faster transistors to be integrated into a single IC chip, many ever-increasing sophisticated DSP applications, which are previously handled by board-level systems, have been taken over by single-chip solutions. In practice, real-time DSP algorithms, such as filtering, matrix operations and discrete transforms, can be implemented by three main approaches: (i) programmable DSP chips, that is, the general-purpose DSP processors first developed in early 1980s [9], [10]; (ii) application-specific integrated circuit (ASIC) chips, that is, dedicated DSP processors; and (iii) programmable logic devices (PLDs), such as the field programmable gate arrays (FPGAs) invented by Xilinx in middle 1980s [11].

A programmable DSP processor is a specialized microprocessor designed for real-time DSP operations. The programmability and flexibility of general-purpose DSP processors makes realization of complex DSP functions possible. However, programmable DSP chips are not well suited to real-time applications at high throughput rates because these DSP processors are sequential and pipeline in nature. Most of them use a single arithmetic unit, such as a multiplier-accumulator (MAC) or an arithmetic logic unit, to perform one operation at a time. Although enhanced DSP

processors introduce the dual-MAC architecture to improve the processing throughput [12], [13], the general-purpose processors still do not have the capability to meet the stringent requirement by high-throughput applications, such as the high-bandwidth communications and video applications.

The main drawback of the general-purpose DSP processors is that their architectures cannot take advantage of the parallelism/pipelining property in many DSP algorithms. In addition, general-purpose processor is also not a good solution for many specific applications with computational-intensive tasks, where there are usually stringent restrictions on speed, cost, power, weight, or I/O bandwidth.

Dedicated DSP chips provide the only feasible solution to such specific applications, in which the required performances cannot be met by general-purpose DSP chips. Advanced VLSI technologies and ASIC design methodology provide the best efficiency through careful design and optimization for these specific applications. Most dedicated DSP chips are designed with specific algorithms and architectures to fulfil the design requirements, using custom implementation or semi-custom design. This implementation approach takes full advantage of the VLSI technologies to obtain high integration, high speed, and very low cost for high-volume applications. Moreover, efficient algorithms and dedicated architectures can be developed to meet the tight constraints, such as high throughput, small area, or low power, by maximally exploiting the parallelism, pipelining, regularity, or recursiveness properties in the DSP algorithms.

Recently, advanced FPGA technology provides an alternative solution for DSP implementations. It offers a compromise between the general-purpose DSP chip and the dedicated DSP processor. The FPGA approach combines the flexibility and programmability of the former, and the high-performance and cost-effectiveness of the latter. The evolution of FPGA chips by Xilinx also provides very high performance by integrating many high-performance hard intellectual property (IP)

cores [14]. The latest FPGA chips have also integrated microprocessor cores to make the SoC in a FPGA chip possible. However, the FPGA approach still faces the serious problems of how to reduce cost and power, as compared to the ASIC solution.

Since our research project aims at the development of efficient VLSI architectures of discrete transforms for ASIC implementation and dedicated DSP applications, a detailed literature review on VLSI design principles and DSP design techniques is required. In this Chapter, Section 2.2 discusses the DSP VLSI design principles, followed by a review of algorithm and architecture level transformations for DSP design. As low-power design has become a very important challenge in VLSI design, Section 2.3 specifically reviews the low-power design techniques for VLSI DSP systems. Section 2.4 then gives a brief introduction of the human auditory and visual systems, which serves as the background knowledge for the use of discrete transforms in many applications. Section 2.5 reviews three discrete transforms, that is, the discrete Fourier transform, constant-Q transform and discrete wavelet transform, which provide the theoretical knowledge for the research presented in Chapter 3, 4 and 5. Finally, Section 2.6 presents a summary at the end of this chapter.

2.2 VLSI Design for DSP Applications

2.2.1 DSP VLSI Design Principles

As discussed in the previous section, DSP systems can be realized by the dedicated DSP processors fabricated using VLSI technology. The general goal of digital VLSI design is to meet different performance requirements, such as the circuitry size, speed, or power, while keeping the cost down. However, there are two important features in DSP algorithms: one is the real-time throughput requirement and the other is the huge amounts of computation and memory. Therefore, the hardware for a DSP system

should be designed to meet the tight throughput constraint of the real-time processing. Additionally, the hardware should also be capable of handling the intensive computations and massive memory accesses, which may be a performance bottleneck to the DSP system.

Fortunately, most DSP algorithms possess some common properties, such as regularity, recursiveness, locality, parallelism/pipelining, etc, which are very useful for dedicated DSP processor design. By maximally exploiting these attributes, DSP VLSI design methodology can take advantage of the VLSI technologies to meet the challenging design requirements by different dedicated applications. In general, careful circuit design and optimization can be done at different design abstraction levels, from the system level, to the algorithm level, to the architecture level, to the register transfer level (RTL), to the gate level, to the layout level, and finally to the process level. However, the algorithm and architecture level design have a tremendous impact on the DSP VLSI design, because the maximum amount of flexibility is available at these two levels [15].

Therefore, efficient VLSI architectures should take advantage of the VLSI technology and maximally exploit the attributes of the DSP algorithms to meet the design requirements. In order to provide design guideline pertaining to the VLSI architecture design in this research, extensive literature review has been taken on the DSP VLSI design principles [1], [4], [8], [15]-[19]. Some important VLSI architecture design criteria are summarized and discussed individually as follows.

- **Regularity and modularity:** The VLSI architecture should be as regular and modular as possible. Any form of regularity derived by a careful algorithm design is useful for mapping algorithms onto architectures, making not only the architecture design easier but also the chip floorplanning and layout design more efficient. The modular design concept gives VLSI designers a tremendous amount of flexibility to cope with ever-increasing complex VLSI

systems. It also offers the architecture design a major advantage of being very scalable technologically because the modular design makes the efforts of architecture redesign minor when the process technology is scaled down to the next generation [17].

- **Extensive concurrency:** An efficient VLSI architecture should take good advantage of parallel and pipeline processing. Throughput is always the overriding design consideration in DSP systems. In order to meet the tight throughput and computation requirements, extensive concurrency by parallel and pipeline processing is always required by many applications, particularly by multimedia and wireless communications. As many DSP algorithms have the in-nature concurrency property in the data flow, the VLSI architecture should exploit the parallelism and pipelining properties in these algorithms. Because pipeline structure is more hardware cost-effective than parallel structure, pipelining techniques are more popular in VLSI systems, especially for cost-sensitive applications.
- **Simple data and control paths:** A good VLSI architecture should have a circuit complexity as low as possible to reduce both design cost and hardware cost, and at the same time, it still meets the throughput requirement. The burden of reducing the hardware complexity can be alleviated by exploiting the regularity and concurrency in DSP algorithms. Fortunately, most basic operations in the DSP algorithms are a set of data operations, such as the multiplications, additions, shifts, etc. Many DSP algorithms can also be decomposed into sub-tasks with regular data flow structure, for example, MAC operations in the filtering task and butterfly operations in the FFT algorithm [2], [3]. These sub-tasks can be processed recursively or concurrently, by exploiting the space-time relationships among these sub-tasks.

- **Interconnection and I/O constraints:** An effective VLSI realization should have localized and reduced interconnection and also take into account constraint on I/O pins. As silicon technology moves to the nanometer process technology, the speed of a silicon circuit is dominated by the interconnect delays rather than the gate delays. Therefore, minimum and local interconnections become very essential in the VLSI realization of DSP systems. Constraint on I/O pins should also be considered because of the associated costs in terms of area and time.

In order to evaluate the VLSI architecture design, we also conclude some important design metrics from the literature review. Basically, a VLSI design should perform the functionality correctly as required by a specific task. The performance of the VLSI architecture is generally measured by three basic design metrics, that is, speed, area and power. In our research, some important figure of merits for DSP VLSI design, which are derived from these three basic design metrics, are listed and described individually as follows.

- **Processing time:** It is also called computation time, defined as the time consumed by computing one specific DSP task. It is usually measured in terms of the processing cycles at a certain frequency, for example, the processing cycles used to complete an N -point FFT at a clock rate of 100 MHz. However, when comparing two systems with different clock rates, the processing time is measured by the product of processing cycles used and clock period. The clock period is determined by the critical path, which is defined as the longest path between any two storage elements in the sequential digital circuits. So the critical path decides the maximum frequency a circuit can operate.
- **Throughput:** The speed of a DSP system can also be evaluated by the throughput which is defined as the number of samples processed per second.

In fact, the throughput is another way to describe the processing speed of the VLSI system, which in essence is identical to the processing cycles.

- **Latency:** The latency is defined as the difference between the time an output is generated and the time at which its corresponding input entered the system. Latency is always used to evaluate the response speed of a system in an application where a fast response is very important.
- **Hardware complexity:** The hardware complexity or circuit complexity can be measured at architecture level, by the number of arithmetic units (multipliers and adders) and the number of storage units (registers and memories). In a typical VLSI DSP system, multipliers and memories always dominate the chip die area and the resultant cost. As discussed earlier, the hardware regularity also reflects the hardware complexity.
- **Hardware utilization:** The hardware utilization of the VLSI architecture is measured in terms of the utilization of the main processing elements (PEs) or arithmetic units in the chip, such as the MAC utilization, multiplier utilization, and adder utilization. The hardware utilization is defined as the ratio of the effective time a circuit uses to perform data operations with respect to the total processing time the circuit uses to complete a specific task.
- **Computational complexity:** The computational complexity is the measurement of the actual data operation quantity. In DSP systems, it is usually measured in terms of the number of arithmetic operations, such as the number of multiplication and addition, which are the dominating data operations of typical DSP algorithms. Lower computational complexity generally means lower hardware burden and lesser power consumption.
- **Memory accesses:** The memory accesses refer to the activities of data swapping/exchanging in the storage, such as memory read and write (R/W)

operations. As most DSP algorithms involve frequent memory accesses, intensive memory operations may also result in the bottleneck of speed performance and significant power consumption. Thus, the number of memory accesses is also another important figure of merit to evaluate the VLSI architecture.

- **Power dissipation:** The power is defined as the average energy dissipated over a period of time. In VLSI DSP systems, the power dissipation is usually measured in terms of the average energy at a certain frequency or total energy required to perform a given DSP task. To compare the power efficiency of two processors, the measurement of energy dissipation per clock cycle, is another efficient method. A lower energy number is more desirable because it requires lesser power to perform the computation at the same frequency.

In this section, we discussed the DSP VLSI design principles that can be used in our design guideline for the VLSI architecture design. We also reviewed some figure of merits, which can be used to evaluate the performance of VLSI DSP systems.

2.2.2 Algorithm and Architecture Level Transformations

Graphical representations of DSP algorithms, such as the block diagrams, signal flow graphs (SFGs), and data flow graphs are used to investigate and analyze data flow properties of the DSP algorithms [1], [8]. Graphical representations can be used to map DSP algorithms into hardware implementations. A graphic representation describes the conceptual computation algorithm of a system. It provides useful information for the hardware architecture design, such as the number of data and control operators, operation ordering dependencies, and inherent concurrency that may exist. From a graphic representation, we can derive the actual hardware architecture of a system by mapping the computation operators and direction edges to

actual hardware modules and interconnections. Different graph representations and mapping alter the speed, area and power characteristics of the hardware architecture. Graphical representations are useful for analyzing the hardware structure properties and exploring architecture alternatives using the transformation techniques.

In this research, we use the block diagram and SFG representations to investigate the discrete transform algorithms and develop different VLSI architectures. In this section, we use these two graphical representations to discuss some transformation techniques that are applied at algorithm and architecture level. Specifically, we will discuss how these transformations can be used to obtain speed improvement, minimization of silicon area, or power reduction.

- **Pipelining:** Pipeline processing is a technique used to increase the concurrency in a DSP algorithm, by decomposing the algorithm into dependent sub-tasks executable one after another in pipelined hardware units [1], [8]. By placing pipelining latches at appropriate points along the data path, the pipelining transformation reduces the critical path or increases the clock frequency (or throughput) of the system. The pipeline processing can also be combined with the supply voltage scaling technique to reduce the power consumption effectively. We will discuss this low-power design technique in the next section. The disadvantages of the pipeline processing may inevitably increase the system latency, and increase the circuit area by introducing the pipelining registers. However, these disadvantages are always minor and outweighed by its performance gain.
- **Parallelism:** Parallel processing is another technique to exploit the concurrency by decomposing an algorithm into independent sub-tasks performed simultaneously in parallel hardware units [1], [8]. Some other transformations, such as the unfolding and multirate transformations [1], [20],

can also be used to expose the parallelism property in an algorithm. The parallel processing can also be used to obtain low-power operation by using the supply voltage scaling technique. However, the parallel processing technique increases the circuit area significantly because it improves the throughput by duplicating multiple identical hardware units to operate concurrently.

- Folding:** Folding [1], [21] is a technique to reduce the hardware complexity by appropriately folding the algorithm or mapping multiple tasks onto the same hardware unit in different time plots. In some VLSI systems, multiple tasks are time-multiplexed to a single hardware unit to save hardware cost. Figure 2.1 shows an example of how two additions are time-multiplexed on a single pipelined adder [1]. Generally, the hardware complexity of an architecture using the folding technique is reduced at the expense of increasing the computation time. Recently, the folding technique has also been used for implementing multirate algorithms, such as the DWT and DWPT, to obtain lower hardware complexity and higher hardware utilization in single clock rate VLSI architectures [22].

Figure 2.1 (a) A simple DSP algorithm that computes $y(n) = a(n) + b(n) + c(n)$; (b) A folded architecture for the algorithm in Figure 2.1 (a). [1]

- Strength reduction:** Strength reduction is a technique to reduce the computational complexity by substituting a complex operator with simpler alternative [1]. A typical example is that a multiplication by a constant can be

converted into a series of shifts and additions. Some of the other transformations include sub-expression elimination and sub-expression sharing [1]. The strength reduction technique is often used to reduce the hardware complexity or power consumption. We will discuss how to use strength reduction to reduce power in the next section.

In this section, we discussed some algorithm and architecture level transformations, such as the pipelining, parallelism, folding and strength reduction techniques. These transformation techniques are used in our research to obtain high throughput, low power, high hardware utilization, and low hardware complexity. In the following section, we will discuss how to use the reviewed transformations to obtain low-power operation.

2.3 Low-Power DSP VLSI Design

As described earlier, low-power design has become a design challenge in VLSI design. In the following sections, we review the low-power design for VLSI DSP systems [6], [15], [23]-[31].

2.3.1 Sources of Power Consumption in CMOS Circuits

The low-power design techniques are targeted towards digital CMOS circuits as they are widely adopted in VLSI design. Three major sources of power consumption in digital CMOS circuits are the leakage power ($P_{leakage}$), the short-circuit power ($P_{short-circuit}$), and the capacitive switching power ($P_{sw.cap}$) [23]-[25]. The average power of the circuit is obtained by adding power consumed by these three sources:

$$P_{avg} = P_{leakage} + P_{short-circuit} + P_{sw.cap} \quad (2-1)$$

In general, the average power consumption in CMOS circuits is categorized into two main portions, that is, the static power (P_{static}) and the dynamic power ($P_{dynamic}$), as described by (2-2).

$$\begin{aligned} P_{avg} &= P_{static} + P_{dynamic} \\ &= P_{leakage} + (P_{short-circuit} + P_{sw.cap}) \end{aligned} \quad (2-2)$$

The static power represents the power caused by the static currents in the circuits, which is mainly from the leakage power. The leakage power is the product of leakage currents ($I_{leakage}$) and supply voltage (V_{dd}). The leakage currents are contributed by two major components: the reverse-bias current (I_{diode}), and the subthreshold current ($I_{subthreshold}$). Therefore, the leakage power consumption can be further decomposed into two components as shown:

$$\begin{aligned} P_{leakage} &= I_{leakage} V_{dd} \\ &= I_{diode} V_{dd} + I_{subthreshold} V_{dd} \end{aligned} \quad (2-3)$$

The leakage power is primarily determined by fabrication technology considerations. Thus, the static power in CMOS digital circuits can be reduced by the proper choice of device technology.

The dynamic power represents the power caused by the switching activities of the circuits, which consists of the short-circuit power and the capacitive switching power. To facilitate the study of the dynamic power, we use the basic CMOS inverter gate as depicted in Figure 2.2. The short-circuit power is caused by the short circuit current in CMOS circuits, as shown:

$$P_{short-circuit} = I_{sc} V_{dd}, \quad (2-4)$$

where I_{sc} is the short-circuit current due to the direct current (DC) path between the supply rails during output transitions. When a CMOS gate switches between its logic states, there is a time period during which both PMOS (p-channel metal oxide

semiconductor) and NMOS (n-channel metal oxide semiconductor) transistors are simultaneously active and creates a direct path from supply to ground, resulting in the short-circuit current. The short-circuit power consumption can be controlled to only a small portion of the total power, by appropriate circuit design at transistor and gate level. It can also be reduced by minimizing the switching activities at the gate's outputs.

Figure 2.2 Dynamic power consumption of a CMOS inverter

The most significant source of the dynamic power consumption is the capacitive switching power, which is caused by the charging and discharging of load capacitors in the CMOS circuits [24]. The capacitive switching power for a gate output node can be expressed as:

$$P_{sw.cap} = \alpha C_l V_{dd}^2 f_{clk}, \quad (2-5)$$

where α is the node switching activity factor (the average number of logic 0 to logic 1 output transitions in a clock period), C_l is the load capacitance at the output node, V_{dd} is the power supply voltage, and f_{clk} is the operating clock frequency.

As illustrated in Figure 2.2, when the input signal changes from logic 1 to logic 0, the load capacitance of the CMOS inverter is charged through the pull-up PMOS transistor and makes a full transition from 0 to V_{dd} . Half of the energy drawn from the power supply ($C_l V_{dd}^2$) is dissipated in the PMOS transistor and the other half is stored

in the load capacitor. When the input signal changes from logic 0 to logic 1, the energy stored in the load capacitance is consumed in the pull-down NMOS transistor. When these switching occurs with a certain rate (α) at a clock frequency f_{clk} , the average power is described by (2-5). It is well established from the equation that the capacitive switching power of digital CMOS circuits can be minimized by reducing the switching activity factor, the load capacitance, the power supply voltage, and the operating clock frequency.

When advance CMOS technologies are at and below 130 nm, the static power becomes more significant because the leakage power has increased significantly. However, the leakage power is primarily reduced by the low level circuit design and the fabrication process. Higher level design has little control over it. On the other hand, the capacitive power still account for a large portion of the total power dissipated in DSP systems because most DSP algorithms involve massive computations and intensive memory accesses. In our research, we focus on the architecture design for VLSI DSP systems. Therefore, the review of low-power DSP VLSI design will concentrate on how to reduce the capacitive switching power consumption.

2.3.2 Basic Principles for Low-Power Design

On the basis of the power dissipation mechanisms, the basic principles for low-power CMOS circuit design are discussed in this section.

A. Reduce Supply Voltage

It is well recognized that reducing the supply voltage (V_{dd}) is generally the most direct and effective way to reduce the power dissipated in digital CMOS circuits, especially to reduce the capacitive switching power, because the V_{dd} has a quadratic relationship

to the capacitive power as described in (2-5). Aggressive voltage reduction can yield more than an order of magnitude power savings in many cases [15]. It was shown in [26] that 80% power reduction is possible for a CMOS circuit by scaling the supply voltage from 3.3 V to 1.5 V.

However, when the supply voltage reduction technique is applied, the speed of a digital CMOS circuit is also reduced. This is because the maximum operating frequency of MOS transistors in the circuit is also decreased at lower supply voltages. To illustrate this issue, the propagation delay of a CMOS inverter gate (T_d) is given by a first-order equation [24] as follows:

$$T_d = \frac{C_l \cdot V_{dd}}{\beta(V_{dd} - V_T)^2}, \quad (2-6)$$

where C_l is the capacitance, β is the transconductance of the transistor, and V_T is the threshold voltage of the transistor. It is well recognised from (2-6) that as the supply voltage approaches the transistor threshold voltage, the gate delay increases drastically, and thus effectively limits the maximum operating frequency of the circuit.

Fortunately, when the voltage reduction is used to reduce the power in DSP systems, the resulting loss in circuit speed can be compensated by employing high-performance algorithms and architectures to maintain the throughput requirement. We will discuss how to use transformations to obtain high-performance architectures and low-power operation in the next section.

B. Reduce Capacitance

Reducing parasitic capacitance is another good way to improve performance and reduce power in digital CMOS circuit design. The parasitic capacitance corresponds directly to the circuit (hardware) complexity of a VLSI design. Basically, the capacitance of a circuit is contributed by the individual gates and the interconnection

in the circuit. Therefore, the capacitance can be minimized by reducing the circuit active area and interconnection, that is, using lesser logic, smaller devices, fewer and shorter wires.

In fact, the parasitic capacitance in CMOS circuits does not dissipate dynamic power if there are no switching activities. Thus, the power reduction can also be achieved by reducing the product of the capacitance and its switching frequency.

C. Reduce Switching Activity

Reducing switching activity offers another good technique for reducing the power consumption. The switching activity for a gate output node in a CMOS circuit depends on two components: the operating clock frequency f_{clk} , which determines the average periodicity of input data arrivals, and the switching activity factor α , which determines how many transitions each data arrival generates.

Useless or unwanted logic switching for computations in digital CMOS circuits should be eliminated when ever possible. Particularly, a significant source of the unwanted switching activity is glitches, which are the spurious and unwanted transitions before a node settles down to its intended value in the circuits. One effective way to reduce glitches of combinational logics is to shorten the logic depth by adding pipelining registers [25], because shorter logic is less glitch-prone.

In a digital CMOS circuit, the switching activity factor mainly depends on the switching activities of the input patterns and the logic function computed by the circuit. Generally, this kind of switching activities is predominantly controlled at algorithm and architecture levels.

2.3.3 Low-Power Techniques for DSP VLSI Design

In general, low-power design techniques can be explored at all levels of the design space. However, our research focuses on the architecture design for VLSI DSP systems. In addition, decisions made at the higher levels of design abstraction generally have more impact on the final quality of the implementation [24]. Low-power considerations should be initiated at high levels, which are the system level, algorithm level and architecture level. Therefore, we review some popular low-power techniques at the high levels for DSP VLSI design.

A. Speedup Transformation with Voltage Reduction

In DSP applications, the real-time processing property implies that there is no advantage in doing computation faster once the required throughput is met by the processing capability of the hardware. This enables the low-power strategy, called speedup transformation with voltage reduction, in which speedup techniques are used to maintain the required throughput rate and obtain substantial power savings at reduced supply voltage.

The high-level transformations that are traditionally used to improve the processing throughput, can be used with aggressive voltage reduction to achieve significant power reduction in a DSP system [15]. In this section, we only discuss how to use pipeline processing to maintain the system throughput and obtain low power at reduced supply voltage [25], [27]. To facilitate in explaining the low-power pipeline processing, we consider a DSP system with a throughput requirement corresponding to a clock frequency of f to be satisfied. In a uni-processing system as depicted in Figure 2.3, the power consumption is given by

$$P_{uni} = C_l V_{dd}^2 f, \quad (2-7)$$

where C_l is the average load capacitance and V_{dd} is the supply voltage of the uni-processing system.

Figure 2.3 Power efficiency of a pipelined processing system [25]

In theory, we can pipeline the uni-processing system N times to improve its power efficiency at reduced voltage. For a simple analysis, we assume that area and voltage scale linearly with the pipelining factor. As depicted in Figure 2.3, pipeline processing can increase the system throughput by inserting pipeline registers into the original uni-processing design. If we keep the supply voltage to the original V_{dd} , the propagation delay of the processing units would be reduced by $1/N$. Thus, we can operate the pipelined system with a lower voltage V_{dd}/N to maintain the identical overall throughput (the frequency f unchanged). The total load capacitance of the hardware remains almost the same since there is no hardware duplication if ignoring additional pipeline registers and control circuitry for a simple analysis. The power consumption of the pipelined system is

$$P_{pip} = C_l (V_{dd} / N)^2 f = P_{uni} / N^2. \quad (2-8)$$

The above idealized analysis ignores the overhead in voltage, area and frequency scaling, but some penalties incur during the pipeline design [25], [27]. However, the low-power pipeline processing is very suited to dedicated DSP processor design. This is because many DSP algorithms are easily pipelined by exploiting the in-nature concurrency. The overheads are always minor as compared to the substantial power savings achieved. As an additional benefit, pipelining also reduces glitch power as it decreases the logic depth as well.

Other transformations to speed up the system throughput, including (i) the parallelism to increase the concurrency at the expense of significant area increase; (ii) the retiming to reduce the critical path time; and (iii) the unfolding to unravel the hidden concurrency, can also be used with voltage reduction to minimize the power consumption [27].

B. Power Management with Switching Reduction

Switching activities are recognized as the biggest cause of power dissipation in computational-intensive DSP systems. The switching activities in a system are related, but not related to the actual computation. However, some unnecessary switching activities, such as glitches and the switching from idle circuits do not contribute to the actual computations and should be eliminated to conserve power.

Power management refers to a class of low-power techniques that carefully manage the performance of a system based on the computational needs to minimize unnecessary switching activities in the system. The most important method of power management is to selectively shut down or slow down the modules in a system, when they are not being used or can reduce their performance to suit application requirements. As depicted in Figure 2.4, clock gating is one of such low-power techniques to eliminate waste power by selectively shutting down the clock to some hardware resources that are idle at certain times. In general, useless switching activities can be effectively eliminated by deactivating idling functional units and system modules.

Figure 2.4 Clock gating

C. Computation Minimization with Switching Reduction

The useful switching activities related to actual computations can also be minimized, while the functionality of a DSP system is still preserved. This kind of switching reduction is often achieved by exploiting the redundancy in the actual computation. There are two popular methods to reduce the computational redundancy. One method is operation reduction, which is to minimize the number of the operations needed to accomplish certain computational tasks. Another method is strength reduction, which is to minimize the power required to perform a certain function.

- In operation reduction, the most common way to minimize the overall switching is by reducing the number of operations to perform certain functions in a DSP system. Polynomial computation is very common in signal processing. Algebraic transformations [28] are often used to reduce operations and obtain lower power by appropriately reordering of the operations in polynomials. For example, the result of $Y = AB + AC$ can be computed by $Y = A(B + C)$, therefore, the computation has fewer multiplications and lesser switching.
- In strength reduction, a power-hungry operator is being substituted by a less power-hungry operator to reduce the overall switching. This method is motivated by the fact that certain operations inherently are less power-consuming per computation unit than other operations. For example, a multiplication by an integer constant can be replaced by a series operations consisting of binary shift and additions/subtractions, which are much less power-hungry than the multiplication.

Additional to computation operations, memory operations is another significant source to switching activities in signal processing. The number of memory operations

required to perform certain computational tasks should also be minimized to conserve power dissipated in DSP systems.

D. Hardware Complexity Minimization with Capacitance Reduction

Generally, lowering hardware complexity can reduce the overall capacitance in a circuit. Fewer hardware resources used to implement a DSP system, such as fewer interconnects, fewer arithmetic units, or fewer storage units, lead to smaller load capacitance, therefore reducing the power dissipated in the silicon chip [23]. On the other hand, minimizing the hardware resources may result in the loss of processing throughput, which is not desirable for the aggressive voltage reduction strategy.

However, in many cases, it is possible to reduce the required amount of hardware and at the same time, maintain the system throughput by increasing the effective utilization of hardware resources in a system [28]. The hardware resource utilization can be increased by distributing the operations uniformly over the available time. The distribution or allocation is often carried out by exploiting the redundancy or idle time of the hardware resources in the system.

Interconnection capacitance can also be reduced when the amount of the computing hardware resources are minimized. This is because a larger number of arithmetic and storage units generally lead to more complex interconnection during the floor planning and routing tasks in VLSI design. In addition, the regularity in DSP algorithms should be exploited and preserved to reduce hardware complexity and therefore, minimize the capacitance and power in the VLSI DSP systems [29], [30].

In addition to the four classifications of low-power design techniques, the correlation property in the data being processed should also be exploited to minimize switching activity in the algorithm and architecture level design [15]. Generally, switching activity decreases when dedicated DSP circuits process data with high temporal correlation (in other words, data amplitude changes slowly) [31].

In this section, we reviewed the high-level low-power techniques for DSP VLSI design. By exploiting the inherent properties existing in DSP algorithms, such as real-time processing, concurrency, computational redundancy, regularity, and data correlation, these low-power techniques can scale down the supply voltage, minimize the switching activity, or reduce the capacitance to obtain substantial power reduction.

2.4 Human Perception Systems

As significant progress has been achieved in the areas of multimedia and artificial intelligence, it is well recognized that a good understanding of human perception systems and appropriate exploiting of the human perception properties is of great importance to these areas. Specifically, many multimedia areas, such as voice recognition, speaker identification, face recognition, speech and audio compression, image and video compression, speech production, and computer graphics, benefit greatly from the studies of human auditory system and human visual system [32]-[37].

2.4.1 Human Auditory System

As shown in Figure 2.5, the human auditory system can be divided into three components: (i) the outer ear, or pinna, is used to collect the sound from the outside; (ii) the middle ear, provides the channel in which the sound passes through the auditory canal and reaches the eardrum, and the movement of the eardrum is transmitted to the inner ear; (iii) the inner ear, which perceives the sound and sends information to the brain via the auditory nerve.

Figure 2.5 Human auditory system [33]

When the outer ear receives sound wave, the sound energy passes through the ear canal and reaches the eardrum. The ear canal acts like a resonant pipe, which resonates at 3-4 kHz and enhances the transmission of the sound energy substantially in the frequency range. The eardrum vibration caused by the sound waves is then transmitted by the middle ear bones to the inner ear. The middle ear bones play an important role in the hearing, which reduces the ear sensitivity progressively at lower frequencies below 1.5 kHz and effectively prevents the sound transmission at frequencies above 20 kHz.

In the inner ear, the sound energy from the middle ear reaches the cochlea. The cochlea is a spirally coiled tube, which is divided along its length by the basilar membrane within the cochlea. The basilar membrane is relatively narrow and stiff near the base of the cochlea, while it is wider and less stiff near the far end inside the tube. The structure of the cochlea is such that for any particular sound frequency, the fluid in the cochlea vibrates in a particular pattern which peaks at one point on the basilar membrane. In general, low-frequency vibrations begin at the base and reach peak amplitude as they reach the far end inside the tube, while the high-frequency

vibrations start at the base and die out as they approach the end. The low- and high-frequency vibrations arrive at their respective peak basilar membrane points nearly simultaneously, which is akin to a filterbank [32]. The auditory nerve connected along the length of the basilar membrane passes auditory information to the brain.

Figure 2.6 Threshold in quiet and masking threshold. Acoustical events in the gray areas will not be audible [39].

Psychoacoustic studies show that the human auditory system can be roughly modelled by a bank of band-pass filters with nonuniform passbands, which are called the auditory filters [38]. Fletcher deduced that the inner ear performs a specific spectral analysis where a frequency-to-place transformation occurs along the basilar membrane. Each location on the basilar membrane responds to a limited band (or range) of frequencies, and therefore, each corresponds to a filter with different center frequency. The limited frequency band is the critical band proposed by Fletcher. The critical band is often demonstrated and measured by studying the auditory masking phenomenon, where a low-level signal can be made inaudible by a simultaneously occurring stronger signal, if the two signals are close enough to each other in frequency domain, as illustrated in Figure 2.6 [39]. In an experiment on the measurement of the threshold of a sinusoidal signal with a narrow-band noise,

Fletcher found that once the noise bandwidth exceeds the auditory filter bandwidth, further increase in noise bandwidth does not increase the loudness of the noise passing through the filter. The bandwidth at which the loudness remains constant is defined as the critical bandwidth. In 1960s, Zwicker published the critical bandwidths known as the Bark scale, as given in Table 2.1 [40]. The Bark scale critical bandwidth is roughly constant at about 100 Hz for low frequencies below 500 Hz, while the critical-bandwidth increases with center frequency for high frequencies above 500 Hz.

Table 2.1 Critical bandwidths of human auditory system [34]

CB rate	Bark scale		Mel scale	
	Center frequency (Hz)	Bandwidths (Hz)	Center frequency (Hz)	Bandwidths (Hz)
1	50	100	100	100
2	150	100	200	100
3	250	100	300	100
4	350	100	400	100
5	450	110	500	100
6	570	120	600	100
7	700	140	700	100
8	840	150	800	100
9	1000	160	900	100
10	1170	190	1000	124
11	1370	210	1149	160
12	1600	240	1320	184
13	1850	280	1516	211
14	2150	320	1741	242
15	2500	380	2000	278
16	2900	450	2297	320
17	3400	550	2639	367
18	4000	700	3031	422
19	4800	900	3482	484
20	5800	1100	4000	556
21	7000	1300	4595	639
22	8500	1800	5278	734
23	10500	2500	6063	843
24	13500	3500	6964	969

Another popular critical-band scale is the Mel scale, as listed in the right column of Table 2.1. The Mel scale is based on the other auditory phenomenon that the inner ear (the basilar membrane) perceives pitch or frequency of a tone at nonlinear scale.

The Mel scale critical bandwidth is approximately 100 Hz for low frequencies below 1 kHz and also increases with center frequency for high frequencies above 1 kHz. Because the Mel scale has finer frequency resolutions than the Bark scale, the Mel scale has more critical bands than the Bark scale in a particular bandwidth. Note that only 24 critical bands for both the Bark scale and Mel scale are listed in Table 2.1.

2.4.2 Human Visual System

The human visual system is illustrated by the simplified horizontal cross section of the human eye in Figure 2.7. The eye is actually an optic globe enclosed by the three membranes: the cornea and sclera, the choroid, and the retina. The cornea is a tough, transparent tissue that covers the anterior surface of the eye, while the sclera is an opaque membrane that encloses the remainder of the globe. Lying directly below the sclera, the choroid serves as the major source of nutrition to the eye. At the anterior extreme of the choroid, the iris (the pupil) contracts or expands to control the amount of light that enters the eye. The ciliary body controls the lens to be flattened or thicker in order to focus on distant or nearby objects. Light from the objects is imaged on the retina that is innermost membrane of the eye. The light receptors over the surface of the retina sense the light intensities, encode the image, and transmit the coded visual information along the optic nerve to the brain [36].

The human vision is provided by the two classes of light receptors in the retina: cones and rods. The cones located only in the small fovea portion are highly sensitive to colour, while a larger-area distribution of rods are sensitive to low levels of illuminations. The cone vision, also called the bright-light vision, helps humans resolve fine details because each cone is connected to its own nerve end. The rod vision, also called the dim-light vision, gives a general, overall picture of the field of view as several cones share a single nerve. When looking at images, humans normally

have a coarse view of the objects first, and then examine the objects at high resolution. In the image, lower spatial frequency components generally contain more information than high frequency components which are the less useful details and noises.

Figure 2.7 Human visual system [36]

Similar to the nonlinear-scale pitch (frequency as perceived by the human ear), the subjective brightness (intensity as perceived by the human eye) is a logarithmic function of the light intensity incident on the eye. Psychovisual studies also show that the human visual system behaves as if it contains a set of critical-band filters [35]. Analogous to the frequency-dependent auditory masking, there is a similar masking dependent on the so-called local scene contents in the human visual system [35], [41].

For our research interests, we provide a brief overview of the human auditory and visual systems. In this section, we reviewed the structures of human ear and human eye, discussed the specific behaviours and properties of human hearing and human

vision, and reviewed the models of these two human perception systems from the view point of signal processing.

2.5 Front-End Spectral Analysis Transforms

As described in the previous section, the human perception systems are described by a set of band-pass filters with nonuniform critical bandwidth both in audition and vision, from a signal processing view point. Both the auditory and visual perceptions are more sensitive to low-frequency signals and less sensitive to high-frequency signals. It is also necessary to recognize that high-frequency signals in the auditory and visual information occur for a short period of time, while low-frequency signals tend to have a longer duration. Both auditory and visual signals have similar properties, such as nonstationarity, periodicity, low-pass frequency spectra in a global sense, and high-pass frequency spectra in a local sense [35].

Motivated by the time-frequency analysis in the human auditory and visual systems, filterbanks and transforms are widely used in many signal processing applications to control the frequency properties and time properties in the input signals. In the following sections, we review three popular discrete transforms: the discrete Fourier transform, constant-Q transform and discrete wavelet transform.

2.5.1 Discrete Fourier Transform

The discrete Fourier transform (DFT) is probably the most popular algorithm widely used in the fields of digital signal processing, such as filtering and spectral analysis, ranging from speech to audio, image, video, and radar applications. Since Cooley and Tukey devised a powerful and fast algorithm for the DFT [42], the fast Fourier transform (FFT) algorithm has been the most popular algorithm in DSP applications. The N -point DFT is defined as follows,

$$X[k] = \sum_{n=0}^{N-1} x[n] \times W_N^{kn}, \quad (2-9)$$

where $x[n]$ is the input signal, $X[k]$ is the frequency content of the input signal, $k = 0, 1, \dots, N-1$, and W_N is the twiddle factor, which is defined by (2-10).

$$W_N = \exp(-j2\pi k / N) = \cos(2\pi k / N) + j * (-\sin(2\pi k / N)). \quad (2-10)$$

The FFT algorithm uses divide-and-conquer approach to reduce the number of arithmetic operations of the DFT from order of N^2 (direct implementation of the DFT formula) down to an order of $M \log_r N$. Due to its regularity, flexibility, and in-place characteristics, the FFT algorithm has become the best candidate for VLSI realization of the DFT computation. Generally, a radix- r ($N = r^n$) fast Fourier formulation decomposes the DFT into a series of basic radix- r butterfly operations in $\log_r N$ stages. In the basic case where $N = 2^n$, the N -point DFT can be broken into a series of radix-2 decimation-in-time (DIT) butterflies, defined by the following equations,

$$\begin{aligned} C &= A + W_N \times B \\ D &= A - W_N \times B \end{aligned} \quad (2-11)$$

where A and B are two inputs and C and D are the two outputs of a radix-2 butterfly. Since A and B are generally complex and require two memory locations to store its value during the computation. Thus, the corresponding results (C and D) are usually written back into the same memory locations to save memory requirement. This process is referred as the in-place FFT computation. This in-place FFT computation is described by the SFG, in which the butterflies are arranged in a regular structure. For example, an 8-point FFT is used to illustrate the radix-2 DIT FFT algorithm, as depicted in Figure 2.8. Generally, an N -point FFT has $\log_2 N$ stages, of which each stage contains $N/2$ butterflies localized in groups. In each group, the butterflies share the same twiddle factors. The number of groups doubles at each new stage, while the number of butterflies per group is reduced by a factor of two. The intermediate results

from each stage computation can be stored in the same memory. Note that in this case, the input sequence must be scrambled in bit-reversed order. The classical FFT software implementation uses triple nested loops to cycle through butterflies, groups, and stages.

Figure 2.8 SFG of an 8-point radix-2 DIT FFT

The DFT is used to determine the frequency content of a signal. However, there is a trade-off between time resolution and frequency resolution in DFT analysis. The longer the time-domain signal, the finer the resolution in the frequency domain, as the frequency resolution is $\Delta f = SR/N$, where SR is the sampling frequency. However, the time resolution decreases as more samples are used in the time domain. This is an issue because many signals including auditory and visual signals are not stationary. In addition, DFT or FFT does not extract any time-domain information from the signal.

A better transform for the nonstationary signals is the short-time Fourier transform (STFT), which is actually a revised version of the FFT applying a short window $w[n]$ over the input signal, described by the following equation,

$$X[k] = \sum_{n=-\infty}^{\infty} w[n]x[n] \times W_N^{kn}, \quad (2-12)$$

where the window $w[n]$ is nonzero only in the range $[-N/2, N/2]$ for an N -point STFT and thus the transform is time-localized. When the time duration of the signal is short enough, it can be assumed to be stationary. The disadvantage of the FFT and the STFT is that they can only provide constant frequency resolution and constant time resolution, in which each has to trade off with the other due to the Uncertainty principle. These transforms lack the flexibility for the time-frequency analysis required by the auditory and visual signals as previously described.

2.5.2 Constant-Q Transform

In the literature, a number of researchers have proposed constant-Q methods to match the time-frequency property of auditory and visual signals. In the constant-Q methods, Q is defined as the ratio of the center frequency, f to the frequency resolution, Δf , described as following,

$$Q = f / \Delta f. \quad (2-13)$$

The DFT has a constant frequency resolution and therefore, an increasing Q is resulted from increasing center frequency. However, in the constant-Q methods, the Q is kept constant over center frequency and the resulting frequency resolution is increasing. Therefore, the constant-Q methods can mimic the critical-band analysis of the human perception system.

From the discrete short-time Fourier transform, Brown derived an efficient constant-Q transform (CQT) [43], as defined by the following equation,

$$X[k_{cq}] = \frac{1}{N[k_{cq}]} \sum_{n=0}^{N[k_{cq}]-1} w[n, k_{cq}] x[n] \exp\{-j2\pi Q_{cq} n / N[k_{cq}]\}, \quad k_{cq} = 1, 2, \dots, n_{cq}. \quad (2-14)$$

where $X[k_{cq}]$ is the k_{cq} th component of the CQT. For each value of k_{cq} , $w[n, k_{cq}]$ is a window function of length $N[k_{cq}]$. As described earlier, the resolution Δf of the DFT is

equal to the sampling rate divided by the window length (the number of samples analyzed in the time domain). In order to achieve a constant Q factor, Q_{cq} , the window length in the Brown CQT varies inversely with frequency, as described by

$$N[k_{cq}] = (SR / f_{k_{cq}}) Q_{cq}, \quad (2-15)$$

where $f_{k_{cq}}$ is the k_{cq} th center frequency. Thus, at low frequencies, long windows are used, while short windows are used for high frequencies. The frequency resolution decreases while the center frequency increases.

Unlike the FFT and STFT which have a constant resolution at all times and frequencies, the Brown CQT provides a good time and poor frequency resolution at high frequencies and a good frequency and poor time resolution at low frequencies, which matches to the general trend in auditory and visual signals. However, the Brown CQT is not invertible because some samples in the input data frame are never analyzed for the higher frequencies in the CQT.

2.5.3 Discrete Wavelet Transform

In the past two decades, the discrete wavelet transform (DWT) has become a successful, effective analysis tool in a wide range of applications, such as numerical analysis, signal analysis, image coding, pattern recognition, statistics and biomedicine [44], [45]. By properly choosing scaling functions and wavelet functions, DWT can decompose a signal into one or more different octave (frequency) bands or resolution levels.

In contrast to other traditional transforms, the DWT performs multiresolution analysis of signals with localization in both time and frequency domains. In addition, the DWT has other useful properties, such as good time-frequency decomposition,

inherent scalability, efficiently computational complexity, and perfect reconstruction of the analysis and synthesis wavelets. There are mainly two classes of DWT algorithms [44], [46], [47], namely, the convolution-based DWT and the lifting-based DWT.

A. Classical DWT

The DWT can be viewed as a kind of multiresolution decomposition of an input sequence. In [44], Mallat proposed a multiresolution decomposition algorithm for the DWT. Based on the Mallat's pyramid algorithm or tree algorithm, the convolution-based DWT is computed through convolution and sub-sampling with a couple of filters to produce the DWT coefficients. The DWT coefficients at any level can be calculated from the DWT coefficients at the previous level.

According to the Mallat algorithm, the convolution-based DWT can be realized by the filterbank tree structure, which is consisting of a high-pass/low-pass filter pair and a downsampler at each level. In the forward transform, the original data $x[n]$ is firstly decomposed into two subsequences at the first level. One is the low-pass sequence (or approximated information) of the original data and the other is the high-pass sequence (or detailed information). These two sequences are the results from the low-pass filter $h[k]$ and high-pass filter $g[k]$. Both of them are derived from the scaling function and the corresponding wavelet functions. After the process of filtering, downsampling operation is used to decimate half of the filtered results. The decomposition (analysis) process of DWT is expressed as follows:

$$\begin{aligned} s[n] &= \sum_{k=0}^{P-1} h[k]x[2n - k] \\ d[n] &= \sum_{k=0}^{P-1} g[k]x[2n - k] \end{aligned}, \quad (2-16)$$

where $s[n]$ and $d[n]$ are the low-pass (scaling) coefficient and high-pass (wavelet) coefficient, respectively. Multilevel DWT can be implemented based on the Mallat's pyramid algorithm. The DWT coefficients at each resolution level are calculated by using low-pass coefficients from the previous level and decimated by two. The low-pass coefficients are further processed in the similar fashion at the subsequent level, while the high-pass coefficients are kept as the final outputs, and this process repeats until the desired final decomposition level is reached. A typical 3-level DWT decomposition is depicted in Figure 2.9.

Figure 2.9 A 3-level DWT decomposition using the pyramid algorithm

B. Lifting DWT

The basic principle of the lifting-based DWT is to factorize the polyphase matrices for the wavelets filters into a sequence of alternating upper and lower triangular matrices and a constant diagonal matrix, and therefore, convert the filter implementation into banded matrix multiplication [46], [47]. Based on lifting scheme, the low-pass and high-pass filters of the DWT can be decomposed into a finite sequence of simple filtering steps, called lifting. This decomposition corresponds to one factorization for the polyphase matrix for the target wavelet filter. The process of factoring wavelet filters into lifting steps is illustrated by (2-17) and (2-18).

First, the low-pass and high-pass wavelet filters are separated into even and odd parts. The polyphase decomposition is described as follows:

$$\begin{aligned} h(z) &= h_e(z^2) + z^{-1}h_o(z^2) \\ g(z) &= g_e(z^2) + z^{-1}g_o(z^2) \end{aligned} \quad (2-17)$$

where $h(z)$ and $g(z)$ are the low-pass and high-pass filters, respectively. They can also be expressed as a polyphase matrix $P(z)$ in the first line of (2-18). The matrix can be factorized into lifting steps as described by:

$$\begin{aligned} P(z) &= \begin{bmatrix} h_e(z) & g_e(z) \\ h_o(z) & g_o(z) \end{bmatrix} \\ &= \prod_{i=1}^m \begin{bmatrix} 1 & s_i(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ t_i(z) & 1 \end{bmatrix} \begin{bmatrix} K & 0 \\ 0 & 1/K \end{bmatrix}, \end{aligned} \quad (2-18)$$

where K is a constant, and $s_i(z)$ and $t_i(z)$ are Laurent polynomials.

Daubechies and Sweldens [46], [47] proved that any finite impulse response (FIR) wavelet filters can be factorized into a finite number of alternating predicting and updating lifting steps starting from the Lazy wavelet. From (2-17) and (2-18), the factorization is actually composed of three basic lifting steps: splitting (Lazy wavelet), predicting, and updating.

For simplicity, we consider a 1-level DWT only consists of three basic lifting steps. In the splitting stage, the input sequence is split into an even-indexed sequence and an odd-indexed sequence. In the second stage, the even samples are multiplied by the prediction operator to predict the odd samples. The difference between the odd samples and the prediction values are calculated. They are the detail (or high-pass) coefficients. In the third stage, the even samples are updated with high-pass coefficients to compute the approximate (or low-pass) coefficients. Figure 2.10 illustrates a 3-level lifting-based DWT using the three basic lifting steps. The DWT coefficients at each level are calculated by the splitting function S , the prediction function P , and the updating function U .

Figure 2.10 A 3-level DWT decomposition using the lifting scheme

The lifting-based DWT outperforms the convolution-based method in many aspects, such as lower computational complexity, full in-place computation, symmetric forward and inverse transform, easy management of boundary extension and integer-to-integer wavelet transform [47]. The lifting-based DWT reduces the computational complexity by up to 50% as compared to the convolution-based method and requires fewer memories.

2.6 Conclusions

In this chapter, we have presented a literature review pertaining to the design of efficient VLSI architectures of discrete transforms for DSP applications. First, we reviewed the principles of DSP VLSI design and the high-level transformation techniques for DSP design. Next, we presented the review of the low-power design methodologies for VLSI DSP systems. We subsequently reviewed the fundamentals of the human hearing and vision systems and discussed the two human perceptions from a signal processing perspective. Finally, we reviewed three discrete transforms, that is, the fast Fourier transform, the constant-Q transform, and the discrete wavelet transform widely used as the time-and-frequency analysis tools in signal processing applications.

Chapter 3

A Low-Cost Fast Fourier Transform Processor

3.1 Introduction

As described in Chapter 2, the fast Fourier transform (FFT) is the most popular algorithm widely used in the fields of digital signal processing. In the past decade, FFT has played a very important role in the applications of multimedia and digital communications, such as speech and audio signal processing, and OFDM-based (orthogonal frequency division multiplexing based) digital communications [34], [39], [48], [49]. As previously discussed, the market demand for portable electronic products, especially the demand for multimedia and communication portable products, has increased explosively. The increased demand for these portable consumer products has shifted design goals to low cost and low power consumption in the dedicated circuit design for these products. On the other hand, the hardware implementation of the FFT algorithm is a very crucial issue, because the FFT computation involves massive arithmetic computations and extensive data swapping, which may result in performance bottleneck, increases hardware implementation cost, or leads to significant power consumption.

The rapid progress of advanced CMOS technologies has enabled the design and deployment of dedicated FFT processors in many application-specific applications. In literature, many dedicated FFT architectures have been proposed. Basically, there are three main approaches to design a dedicated FFT processor: (i) the

recursive/sequential FFT architecture (or single-butterfly architecture) [50], [51], (ii) the pipeline architecture [52]-[56], (iii) the parallel architecture [57], [58].

Each approach differs in the level of parallelization, trading off processing time and hardware complexity. Thus, when a designer chooses one of the architectures to implement FFT, the main concern is the trade-off between speed requirement and hardware cost. The pipeline and parallel architectures are widely used to implement the high-speed long-sized FFT for high-performance applications, such as image and video signal processing, radar signal processing, and high-speed digital communications. However, these pipeline and parallel architectures consume more hardware resources (silicon area) than the single-butterfly architectures, because they use multiple butterfly units to increase the throughput. Therefore, the single-butterfly architectures are more suitable for area-constrained applications, where high throughput is not the design goal.

In this chapter, we present a 256-point FFT processor design for portable speech and audio applications (with sampling rates ranging from 8 kHz to 16 kHz), in which processing throughput is moderate and is suitable for speech recognizers [34], speech and audio coders [39], hearing aids [48], as well as other general speech and audio products. As a key part in these speech and audio applications, FFT performs the transform of speech or audio signal samples from time domain to frequency domain, through intensive computations and frequent memory accesses. As size, weight and power are major considerations for portable product design, the design goal is to use the lowest hardware cost (generally means minimum chip die area) to obtain the least power consumption while reaching the desired throughput.

Therefore, for low cost requirement, we choose the recursive/sequential FFT architecture using a single butterfly unit interacting with a single memory system to compute the FFT, which consumes the minimum hardware resources and therefore is a low-cost solution. By exploiting the recursiveness property in the FFT data flow, we

employ a two-multiplier and three-adder pipelined butterfly unit to calculate the butterflies at different levels, recursively/sequentially. By using the algebraic transformation technique and a rescheduling of the butterfly arithmetic operations, we derive this two-multiplier and three-adder structure to obtain a good trade-off between hardware cost and processing speed, as compared with other existing butterfly units. For low-power operation, the aggressive supply voltage scaling technique reported in Chapter 2 is applied in the FFT processor design to obtain substantial power saving. We design a 256-point FFT processor based on the proposed single-butterfly architecture and implement the design by advanced CMOS technology. We verify the low-cost realization and low-power operation of the proposed FFT architecture by circuit simulation and implementation results. We also compare the proposed recursive FFT architecture with other FFT architectures to verify its hardware cost efficiency. Circuit simulation and implementation results indicate that the proposed 256-point FFT processor meets the tight requirement of low-cost implementation and low-power operation for portable speech and audio applications.

The organisation of this chapter is as follows. Section 3.2 gives an intensive review of various dedicated FFT architectures. Different implementations of the butterfly unit are also reviewed in Section 3.3. The derivation and the performance evaluation of the proposed pipelined butterfly unit is presented in Section 3.3, followed by the design of the proposed recursive FFT architecture. In Section 3.4, circuit simulation and implementation results are presented to evaluate the performance of the proposed 256-point FFT processor. Finally, conclusions are presented in Sections 3.5.

3.2 Review of Existing FFT Architectures

As described in Chapter 2, an N -point FFT is generally calculated in $O(\log_r N)$ stages, where r is the radix of the FFT decomposition and N is a power of the radix r . Each stage consists of N/r radix- r butterfly computation units. Each stage requires the reading and writing of all N data words. The FFT processor is the hardware implementation of FFT computations, motivated by meeting different applications. In the literature, many dedicated FFT architectures have been proposed with different advantages and disadvantages. In general, all these FFT architectures can be classified into the three main categories, as described in the following sections.

3.2.1 Recursive/Sequential FFT Architecture

The recursive/sequential FFT architecture, also called the single-butterfly architecture, applies just one time-multiplexed butterfly unit as data path to compute butterfly operations recursively and sequentially [50], [51]. By multiplexing the single butterfly unit, this single-butterfly architecture computes FFT through butterfly-by-butterfly, group-by-group, and stage-by-stage, while making frequent memory accesses at the same time. In other words, multiple butterfly operations at different levels are mapped into one butterfly unit, and the intermediate coefficients are buffered in memories during the computation.

As shown in Figure 3.1, there are mainly three recursive FFT architectures with different memory management strategies [51]: (a) the single-butterfly architecture (Figure 3.1 (a)) with a single-memory system, which uses a single butterfly unit (BU) interacting with a single memory system to calculate the FFT, recursively; (b) the single-butterfly architecture (Figure 3.1 (b)) with a dual memory system, of which one of the two memories is used for the butterfly inputs and the other buffers the butterfly

output data at any time during the computation; (c) the single-butterfly architecture (Figure 3.1 (c)) with a cache memory, which employs a small cache between the butterfly unit and the main data memory to improve the speed and energy efficiency.

Figure 3.1 Conventional recursive FFT architectures

Basically, there are three memory management strategies for the memory accesses in the single-butterfly FFT architectures, namely, the in-place mode (single-memory), the ping-pong mode (dual-memory) and the cache mode (cache-memory). We describe these three memory managements as follows:

- **In-place mode:** The simplest memory-system architecture is the single-memory architecture, as show in Figure 3.1 (a), in which a memory of at least N words is connected to a butterfly unit by a bidirectional data bus. In general, during the FFT computation, intermediate data are read from and written back to the memory once for each of the $\log_r N$ stages of an N -point FFT. This architecture achieves the minimum memory requirement by exploiting the in-place property of FFT computation.
- **Ping-pong mode:** The dual-memory architecture places two memories of size N on separate buses connected to a butterfly unit, as shown in Figure 3.1 (b). Data begin in one memory and “ping-pong” from memory to memory $\log_r N$

times until the N -point FFT computation has completed. As compared to the single-memory architecture, the dual-memory architecture improves the processing throughput at the cost of memory requirement with a size increased from N to $2N$.

- **Cache mode:** The cached-memory architecture [51] uses a cached memory. This architecture is similar to the single-memory processor except that a small cache memory resides between the butterfly unit and the main data memory. By exploiting the locality of FFT computation, this cached-memory architecture obtain two key advantages: (i) increased speed performance because a smaller (cache) memory located nearer to the data path is faster than larger one; (ii) increased power efficiency since a smaller and nearer memory requires much lower energy. However, this method also brings two primary disadvantages of additional new functional units (caches) and increased control complexity.

The disadvantage of the single-butterfly FFT architecture is the bottleneck of speed performance because this architecture uses a single butterfly unit to perform butterfly computations. In order to improve the processing throughput, a high-radix FFT algorithm can be used at the expense of increased arithmetic complexity. In addition, the intensive memory read/write accesses also create a speed bottleneck. Although memory bank partitioning can be employed to offer simultaneous data accesses, the trade-off is a more complex addressing scheme and higher silicon area. However, this single-butterfly architecture has to run at a very fast clock rate when it is used in high-performance applications.

3.2.2 Pipeline/Cascade FFT Architecture

The pipeline/cascade FFT processor is a special class of processors for FFT computation with high throughput advantage. By breaking the inherent recursiveness in the FFT algorithm and obtaining a unidirectional data processing, the pipeline architecture uses a chain of butterfly units cascaded with a series of buffers to compute the FFT in a fashion of rapid pipeline processing.

In the past three decades, vast pipeline FFT architectures have been proposed and deployed in a variety of high-throughput DSP applications [52]-[54]. As depicted in Figure 3.2, the pipeline FFT architectures can be classified into two categories: multipath delay commutator (MDC) scheme [55] and single-path delay feedback (SDF) scheme [56]. Generally, the MDC scheme can obtain a higher processing throughput, while the SDF scheme requires lesser memory and computation resources.

Figure 3.2 Conventional pipeline FFT architectures

In general, there are $\log_r N$ radix- r butterfly units interleaved with a series of buffer memories in the pipeline FFT architecture. At each stage, the pipeline FFT architecture maps N/r butterfly operations into one radix- r butterfly unit. By employing the buffer memories to reorder the data streams at each stage, the pipeline FFT architecture uses cascaded butterfly units to process the data to obtain high-speed operation. In addition, the pipeline FFT architecture has excellent advantages of

regular structure and relative simple control, which also makes it the best choice for high-speed applications. Obviously, the disadvantage is that the pipeline approach duplicates the butterfly units $\log_2 N$ times compared to a single-butterfly approach.

3.2.3 Parallel FFT Architecture

The parallel FFT architecture decomposes the FFT algorithm computation into independent tasks, and increases the numbers of independent butterfly units working in parallel to improve the overall system throughput. In parallel FFT architecture, FFT algorithm is computed by an array of a number of independent PEs (butterfly units) with local buffers, interconnected through some type of network, as depicted in Figure 3.3. The Cobra FFT processor designed in [57] uses an array of multiple chips, in which each contains one PE and one local buffer. Another parallel FFT architecture presented in [58] uses a column of radix-2 butterfly units and a number of memory banks to compute FFT algorithm. This method provides designers much flexibility by trading off hardware cost and timing performance, as compared with other methods. However, the parallel FFT architecture methods increase the overall processing throughput at the expense of higher hardware complexity. As compared to the pipeline FFT approach, the parallel FFT approach generally results in more silicon overhead.

Figure 3.3 Conventional parallel FFT architecture

3.3 The Proposed Recursive FFT Architecture

As we discussed in Section 3.2, each of the three main approaches differs in the level of parallelization, trading off processing speed and hardware cost. In array architecture, many operations required by the radix- r FFT algorithm are carried out at the same time, which always consumes a lot of butterfly units, associated memory buffers, and complicated interconnections. All these necessary hardware resources resulted in significant silicon area. The pipeline FFT architecture is the best architecture choice with acceptable hardware cost for many high-throughput applications, of which each stage requires a butterfly unit and features regularity and simple control. The pipeline architectures are widely used to implement the high-speed FFT for many high-performance applications. However, they consume more hardware resources (silicon area) than the single-butterfly architectures, because multiple butterfly units are used to increase the throughput. Compared with the parallel and pipeline architectures, the single-butterfly architectures are more suitable for area-constrained applications, where high throughput is not the design goal.

For portable speech and audio applications with sampling rates ranging from 8 kHz to 16 kHz, parallel and pipeline architectures are not necessary and too expensive

as the performance requirement is not high. At this audio sampling frequency, small area, and low cost consideration, we choose the recursive FFT architecture which consists of a single butterfly that consumes minimum hardware resources. To further reduce the hardware requirement, the single-memory system is employed in our ASIC design of FFT processor because the dual-memory system occupies more silicon area and the cache-memory system is more suitable for full-custom IC design. As described in Chapter 2, the Cooley and Tukey's FFT algorithm can decompose an N -point DFT ($N = r^n$) into a series of basic radix- r butterfly operations. Higher radix butterflies generally require more multipliers and more complicated addressing scheme, and thus, consume more silicon area. It is also commonly known that the decimation-in-time (DIT) FFT provides better SNR (signal-to-noise ratio) than the decimation-in-frequency (DIF) algorithm when finite word length is used in the FFT implementation [59]. Therefore, in this study, we choose the radix-2 DIT FFT algorithm for a low-cost single-butterfly FFT processor design.

As a low-cost implementation of radix-2 DIT FFT algorithm, the single-butterfly FFT processor multiplexes an optimized butterfly unit interacting with a single memory to compute the radix-2 butterflies at different stages, recursively. During the computation, the intermediate results at each stage are buffered in the same memory. In principle, the performance of a single-butterfly FFT processor is determined by the hardware implementation of the butterfly unit, because the butterfly operations consume most of the FFT computation time and also lead to significant power dissipation. Hence, the realization of the butterfly unit is very crucial in the design of an efficient single-butterfly FFT processor.

Therefore, the previous bit-parallel implementations of radix-2 DIT butterfly unit are reviewed. Considering the trade-off between processing throughput and hardware cost in the implementation of the butterfly unit, we propose an efficient butterfly unit structure to compute radix-2 DIT butterflies. As the aggressive voltage reduction

technique can achieve more than one order of magnitude power reduction, pipeline processing with voltage scaling is applied on the butterfly unit in order to obtain low-power operation.

3.3.1 Review of Existing Butterfly Units

The radix-2 DIT butterfly operation described in (2-11) in Chapter 2 can also be represented by the SFG, as shown in Figure 3.4. In general, all variables in the butterfly operation are complex. The radix-2 DIT butterfly computation involves two complex data read from memory, the arithmetic computation consisting of one complex multiplication and two complex additions/subtractions, and two complex data write to memory. The key issue in various hardware implementations of the radix-2 butterfly unit is how to trade off between speed performance and hardware cost. Different hardware approaches also require different memory I/O bandwidths and different control complexity. In general, high-speed processing results in higher hardware complexity in the data path, and requires a corresponding higher memory I/O bandwidth, but with reduced control complexity.

Figure 3.4 SFG of radix-2 DIT butterfly

A. Conventional 3-Stage Pipelined Butterfly Unit

In the direct implementation of the radix-2 DIT butterfly unit without employing internal pipelining, the critical path of performing a butterfly computation is the

summation of the memory read operation (R), the arithmetic computation (C), and the memory write operation (W). To increase the overall operating frequency, the simplest way is to divide the operations of the butterfly computation into three different basic steps, each corresponding to a mentioned operation as shown in Figure 3.5 [60], [61]. A dual-port memory can be employed to avoid hardware conflict when the butterfly unit intends to access more than two data in the same single-port memory bank [60], [61]. With the dual-port memory bank, each butterfly operation takes one cycle when the pipelining is stable. The latency is one cycle.

Figure 3.5 Conventional radix-2 DIT pipelined butterfly unit [60]

The throughput is limited because the pipeline clock frequency is fairly low. This pipeline clock period is determined by the computation time of one complex multiplication and one complex addition/subtraction required by the radix-2 butterfly operation.

B. 4-Multiplier and 6-Adder Pipelined Butterfly Unit

To achieve the highest processing throughput, a popular way is to divide the butterfly operation described by (2-11) into real arithmetic operations, such as real multiplication and real addition. Thus, the radix-2 DIT butterfly operation can be expanded and expressed as follows:

$$\begin{aligned}
C_R &= A_R + (B_R \times W_R - B_I \times W_I) \\
D_R &= A_R - (B_R \times W_R - B_I \times W_I) \\
C_I &= A_I + (B_I \times W_R + B_R \times W_I) \\
D_I &= A_I - (B_I \times W_R + B_R \times W_I)
\end{aligned} \quad (3-1)$$

where the subscripts “ R ” and “ I ” are used to denote the real part and the imaginary part of a complex variable, respectively. This group of equations indicates the radix-2 butterfly can be computed through four parallel data paths.

As shown in Figure 3.6, the SFG of a fully parallel butterfly structure is derived from (3-1). The four-multiplier and six-adder butterfly unit can be derived by direct-mapping the operators in this SFG into actual arithmetic hardware. By exploring the fully parallelism in the butterfly operation and using internal pipelining, this parallel structure obtains the highest throughput. This butterfly unit computes one butterfly operation every cycle. It has a latency of four cycles. The pipeline clock period is approximately determined by the gate delay of a multiplier or an adder. The clock frequency can be further increased effectively by dividing the multiplier into several sub-operations in a full-custom IC design [51].

Figure 3.6 SFG of the full-parallel butterfly unit

This implementation approach has the highest throughput at the price of maximum hardware requirement of four multipliers and six adders. However, the control is

relative simple because this approach is the direct-mapped implementation of the butterfly arithmetic operations. In order to take advantage of its full parallel processing, this butterfly structure requires multi-port or multi-bank memory system to provide multiple data R/W accesses. This four-multiplier and six-adder butterfly unit trades hardware resources for high speed performance.

C. 1-Multiplier and 2-Adder Pipelined Butterfly Unit

In order to achieve the minimum hardware requirement and a higher throughput, a pipelined butterfly unit using one multiplier and two adders is proposed in [50]. As illustrated in Figure 3.7, this butterfly unit multiplexes one multiplier and two adders to perform four real multiplications and six real additions/subtractions required by a radix-2 butterfly operation. Thus, it reduces the hardware requirement to the minimum and trades off processing speed. This structure takes four cycles to complete one butterfly operation and has a latency of five cycles.

Figure 3.7 Architecture of the 1-multiplier and 2-adder butterfly unit [50]

This implementation method uses the least hardware resources with a low processing speed. As shown in [50], the scheduling and control is fairly complex.

Thus, this architecture introduces more hardware overhead, such as many pipeline registers/latches and multiplexers. This one-multiplier and two-adder butterfly unit obtains low hardware requirement at the price of low speed performance and complicated control.

D. 2-Multiplier and 4-Adder Pipelined Butterfly Unit

It is stated in the literature [62] that a two-multiplier implementation produces the best trade-off as compared to the previously described approaches. Compared with a two-multiplier implementation, the four-multiplier and six-adder scheme doubles the processing speed but results in a significant increase in cost. The four-multiplier method nearly doubles the die area because the numbers of multipliers and adders double. In addition, the doubled I/O bandwidth requires multi-port or multi-bank memory which may lead to more storage and routing resources. On the other hand, the single-multiplier and two-adder scheme reduces the performance significantly but only results in marginal die cost reduction.

By exploiting the similarity among data path structures described in (3-1), we can rearrange the butterfly equations in another form as follows:

$$\begin{aligned} C_R(D_R) &= A_R \pm (B_R \times W_R - W_I \times B_I) \\ C_I(D_I) &= A_I \pm (W_R \times B_I + W_I \times B_R) \end{aligned} \quad (3-2)$$

The corresponding SFG is shown in Figure 3.8 (a). Based on (3-2), the four parallel data paths in Figure 3.6 are folded into two independent paths, which compute the real parts and imaginary parts of C and D , respectively. In each independent path, only one multiply operator is multiplexed to perform the required two multiplications and the following add operators are multiplexed to perform the required subsequent subtractions/additions. The corresponding butterfly unit architecture is depicted in Figure 3.8 (b). Given an appropriate scheduling and control, this pipelined butterfly unit takes two cycles to compute a butterfly operation with a latency of five cycles.

Figure 3.8 (a) SFG and (b) architecture of the 2-multiplier and 4-adder butterfly unit

This pipeline architecture is very regular and the control is simple. This two-multiplier implementation provides a best trade-off between speed performance and hardware cost, and therefore, is a good solution for the single-butterfly FFT architecture.

3.3.2 The Proposed Butterfly Unit

A. 2-Multiplier and 3-Adder Pipelined Butterfly Unit

As discussed previously, a two-multiplier implementation of the radix-2 butterfly unit provides the best trade-off between speed and cost. In this section, we propose a novel two-multiplier butterfly unit, which employs hardware multiplexing and pipeline processing to achieve small area and low power. Compared with the two-multiplier and four-adder approach, the proposed butterfly structure has one adder fewer but still maintains the throughput of two cycles per butterfly operation.

The two-multiplier and three-adder radix-2 DIT butterfly unit is derived from the rewritten butterfly equations as follows:

$$\begin{aligned}
 X &= B_R \times W_R - B_I \times W_I \\
 C_R &= A_R + X \\
 D_R &= A_R - X \\
 Y &= B_I \times W_R + B_R \times W_I \\
 C_I &= A_I + Y \\
 D_I &= A_I - Y
 \end{aligned} \tag{3-3}$$

As described in (3-3), X , C_R , and D_R are computed first, followed by the subsequent computations of Y , C_I , and D_I . By rescheduling the arithmetic operations in the butterfly computation, we can further reduce the number of adders required by the two-multiplier scheme.

From (3-3), we derive the corresponding butterfly SFG as depicted in Figure 3.9 (a). Similar to the two-multiplier and four-adder scheme, this novel two-multiplier approach also produces two complex results C and D in two data paths. However, the difference is that in the rescheduled SFG of butterfly operations depicted in Figure 3.9 (a), C_R and C_I are computed in one path, while D_R and D_I are computed in the other path. By multiplexing the add operator and properly swapping the input data, the

subtraction and addition to compute X and Y , respectively, can be performed by one arithmetic operator. Thus, the two separate data paths are combined to save an adder.

Figure 3.9 (a) Rescheduled SFG of the radix-2 butterfly; (b) scheduling of the butterfly operations

The corresponding scheduling for the butterfly operations is shown in Figure 3.9 (b). As described in Figure 3.9 (b), the structure of data flow for computing C_R and D_R is similar to that for computing C_I and D_I . Each data flow requires two multiply

operators and three add operators. By scheduling the operation sequence to compute C_I and D_I one cycle later than the operation sequence to compute C_R and D_R , the two operation sequences can share two multiply operators and three adder operators. In the similar way, the operation sequences to compute corresponding C_R (C_I) and D_R (D_I) in consecutive butterflies can be interleaved on a single two-multiplier and three-adder data path in a fashion of fast pipelining, by proper data switching and control.

Based on the rescheduled SFG of the butterfly operations, a pipelined two-multiplier and three-adder butterfly unit architecture is derived as shown in Figure 3.10. This butterfly architecture is actually an improved version of the two-multiplier and four-adder approach. For the speech and audio applications in our study, the data are represented in the format of two's complement fixed-point format. The I/O data of the butterfly unit is 16 bits. In the pipelined butterfly unit, the two multipliers in the first stage are fed with 16-bit input data and coefficients from the RAM (random access memory) and coefficients ROM (read only memory), respectively. They perform 16-bit $\times$ 16-bit two's complement multiplications and produce 32-bit intermediate results. The 32-bit wide adder in the second stage performs addition or subtraction. The corresponding results are truncated and fed to the two 16-bit adders in the last stage. The final results are written back to the RAM through the last pair of register. By using two multiplexers, the proposed butterfly unit multiplexes the two multipliers and three adders to perform appropriate butterfly computations. The overflow problem is resolved by dividing both the summation in the second stage adder and the input A by a factor of two, which is simply realized by one-bit of right shift operation. Note that the VHDL codes of structure design for the proposed butterfly unit is provided in Appendix A.

Figure 3.10 Architecture of the proposed pipelined butterfly unit

B. Performance Analysis and Evaluation

The pipeline processing of the butterfly operations is illustrated in Table 3.1. The proposed butterfly unit computes a butterfly operation in two cycles, with a latency of four cycles. For an N -point radix-2 FFT, the total computation time for $N/2 \cdot \log_2 N$ butterflies is $(4 + N \log_2 N)$ cycles. By using internal pipelining, the clock period is approximately the time required to perform one multiplication.

Table 3.1 Pipeline table of the proposed butterfly unit

Read RAM	Mults	Mults Sub Read RAM	Add Add/Sub Read RAM	Add/Sub Write RAM	Write RAM	-	-
B, W	$B_R * W_R$ $B_I * W_I$	$B_I * W_R$ $B_R * W_I$ $B_R * W_R - B_I * W_I$ Read A_R	$B_I * W_R + B_R * W_I$ $A_R + X$ $A_R - X$ Read A_I	$A_I + Y$ $A_I - Y$ Write D_R C_R	Write D_I C_I	-	-
-	-	Read RAM	Mults	Mults Sub Read RAM	Add Add/Sub Read RAM	Add/Sub Write RAM	Write RAM
-	-	B, W	$B_R * W_R$ $B_I * W_I$	$B_I * W_R$ $B_R * W_I$ $B_R * W_R - B_I * W_I$ Read A_R	$B_I * W_R + B_R * W_I$ $A_R + X$ $A_R - X$ Read A_I	$A_I + Y$ $A_I - Y$ Write D_R C_R	Write D_I C_I

The hardware requirement and timing performance comparison of different butterfly unit architectures are given in Table 3.2. Note the clock cycle is approximately determined by the gate delay of a multiplier or an adder. The proposed two-multiplier butterfly unit has a moderate hardware requirement. It achieves a good

pipeline performance that is a throughput of two cycles per butterfly operation and a latency of four cycles. Compared with the two-multiplier and four-adder scheme, our method can still maintain a throughput of two cycles, while it requires fewer hardware resources by reducing the number of adders from four to three, and minimizing the numbers of registers, switches and multiplexes. This hardware reduction contributes to a significant saving in silicon area and also results in a simpler control. With the internal pipeline processing, the proposed butterfly unit can obtain low-power operation with the supply voltage scaling technique, while still maintaining real-time processing.

Table 3.2 Hardware requirement and timing performance of different butterfly unit architectures

-	Multiplier	Adder	Registers	MUX	Switch	Throughput (cycles)	Latency (cycles)
1-mult & 2-add	1	2	9	2	0	4	5
4-mult & 6-add	4	6	16	0	0	1	4
2-mult & 4-add	2	4	14	2	2	2	5
Proposed one	2	3	11	1	1	2	4

In summary, the proposed two-multiplier and three-adder butterfly architecture is a hardware cost effective approach, which still maintains a good processing throughput. This architecture provides a good implementation of radix-2 DIT butterfly unit, and good trade-off between speed performance and hardware cost. A 256-point single-butterfly FFT processor employing this radix-2 DIT pipelined butterfly unit is designed and implemented for low-cost portable speech and audio applications.

3.3.3 The Proposed FFT Architecture

The proposed recursive FFT architecture is composed of a number of functional units as shown in Figure 3.11. In this architecture, the butterfly data path employs the

proposed two-multiplier and three-adder pipelined butterfly unit to compute the FFT butterfly operations. The coefficient ROM is used to store the twiddle factors W_k . The RAM is used to store the input data, buffer the intermediate results, and store the final results. The twiddle factors, I/O data, and intermediate data are all represented in the 16-bit two's complement fixed-point format. The address generator is used to generate the correct addresses for the coefficient ROM and RAM. The controller supervises all other units to operate properly in three operation modes, that is, the input mode, the FFT computation mode, and the output mode. In the input mode, a frame of N -point input data is written into the RAM. During the computation mode, an FFT of the N -point data is computed and the intermediate data is buffered in the RAM. The final FFT coefficients are read from the RAM in the output mode.

Figure 3.11 Block diagram of the proposed single-butterfly FFT processor

As illustrated by the SFG in Figure 2.8 in Chapter 2, an N -point DFT computation is decomposed into a series of radix-2 DIT butterfly operations, which are arranged by groups and stages in a regular form. Exploiting the inherent recursiveness and regularity in the data flow, the proposed recursive FFT architecture employs one time-multiplexed butterfly unit to compute the butterflies sequentially to reduce the required arithmetic hardware resources. According to the in-place property, the intermediate results of the each stage can be stored in the same memory to reduce the memory requirement. However, the addresses for the input data and twiddle factors of

the butterflies are not sequentially ordered due to the repeatedly decomposition and have to be recomputed for each new butterfly. In addition, the input data sequence of the FFT computation must be scrambled in bit-reversed mode in order to get the output sequentially ordered.

For the proposed pipelined butterfly unit architecture, the correct addressing for the I/O data and twiddle factors of the butterfly operations is an important task. To maximally speed up the FFT computation, the addressing for the I/O data and the twiddle factors must be performed independently in a dedicated FFT address generator. The task of the address generator is to cycle through all the butterflies in increasing order and to generate the appropriate addresses for the butterfly inputs, twiddle factors, and outputs. The address generator is also responsible for generating the correct addresses for the I/O data during the I/O modes. The circuit design of the address generator will be addressed, and followed by the discussion of the circuit design and operation for the memory and controller.

A. FFT Address Generator

The FFT address generator provides the appropriate addresses for the RAMs or coefficient ROM in three working modes, under the supervision of the controller. The block diagram of the address generator is shown in Figure 3.12. The mode signals (*ip_mode*, *fft_mode*, and *op_mode*) are generated and provided by the controller. During the I/O modes, the generator uses the signal, *io_done* to inform the controller whether the input process or output process has completed. In the FFT mode, the FFT address generator keeps track of which butterfly and which stage is being computed. When all stages of butterflies have completed, the generator informs the controller via the signal, *fft_done*.

Figure 3.12 Block diagram of the FFT address generator

As depicted in Figure 3.12, the FFT address generator is composed of five main modules: (i) a butterfly generator (*bfly_generator*), (ii) a stage generator (*stage_generator*), (iii) an address generation unit (*agu_fft*) for the RAM in the FFT mode, (iv) an address generation unit (*agu_rom*) for the coefficient ROM, and (v) an address generation unit (*agu_io*) for the RAM in the I/O modes. In the I/O modes, the butterfly generator keeps track of the I/O processes and will inform the controller by a signal, *io_done*, when the N -point data W/R operations have completed. During the FFT mode, the butterfly generator works as a $\log_2(N/2)$ -bit up counter, keeping track of which butterfly is processed in a particular stage. The butterfly counter is incremented by the clock signal, *next_bfly* from the controller. When a stage of butterfly operations has completed, the butterfly generator increases the stage counter

by using the signal, *stage_done*. The stage generator (also an up counter) keeps track of which stage of butterfly operations is being computed. When the stage count is $\log_2 N$, it generates a signal, *fft_done* to tell the controller that an FFT computation is completed.

The I/O address generation unit (*agu_io*) is actually a $\log_2 N$ -bit up counter to compute correct I/O addresses (*addr_ip* and *addr_op*) in the I/O processes. During the input mode, it generates scrambled addresses in bit-reversed mode. When in the output mode, the I/O address generation unit provides addresses in a natural sequence. The FFT address generation unit (*agu_fft*) only operates in the FFT mode and calculates the addresses for the I/O data (*addr1* and *addr2*) of the butterfly computations. The ROM address generation unit (*agu_rom*) also operates in the FFT mode and calculates the addresses of the proper twiddle factors (*addr_rom*). Provided with the mode signals, butterfly index (*index*) and number per group (*bpg*) signals, these three address generation units can operate appropriately to provide correct addresses.

B. FFT Memory

There are two memory blocks in the proposed recursive FFT architecture. The coefficient ROM is a memory block of $N/2$ -word $\times$ $2M$ -bits, where M is the word-length of the coefficients. The twiddle factors are computed and pre-stored into the ROM. The most significant M -bit data is the real part of the twiddle factors and while the least significant M -bit data is the imaginary part. This ROM is implemented in a LUT (look-up table). When fed with correct addresses from the FFT address generator, the coefficients ROM provides the appropriate twiddle factors for the butterfly unit.

The RAM is a memory block of N -word $\times$ $2M$ -bits, where M is the word-length of the I/O data and intermediate data. It consists of two identical memory banks of N -word $\times$ M -bits to store the real part or imaginary part of the data, respectively. In

order to increase the data access bandwidth to maximize the pipelining of the butterfly computation, each memory bank has two read data ports and two write data ports to support simultaneous R/W operations. The RAM is used to store the I/O data in the I/O modes and buffer the intermediate data during the FFT mode. For an N -point radix-2 FFT computation, $2N\log_2 N$ R/W memory accesses are required during the FFT computation.

C. FFT Controller

The FFT controller supervises other functional units in the FFT processor for proper operations. Depending on the feedback signals from other units, the controller sends out control signals to determine the sequence of events. The controller consists of two finite state machines (FSMs) called Mode FSM (*fsm_mode*) and FFT FSM (*fsm_fft*), and a FFT counter (*counter_fft*). The block diagram of the controller is shown in Figure 3.13.

Figure 3.13 Block diagram of the FFT controller

The Mode FSM is in charge of the FFT processor operation modes. This Mode FSM begins to work for the input process when the input signal, *ip_ready* indicates that a frame of input data is ready. Firstly, the Mode FSM enters into the input mode (*ip_mode*), in which the Mode FSM generates write control signal (*wr_ip*) and the FFT address generator provides correct memory addresses. When the signal, *io_done*

from the address generator informs that the input process is completed, the Mode FSM switches into the FFT mode. Then FFT FSM starts to operate for the FFT computation process (*fft_mode*).

Working together with the butterfly generator (*bfly_generator*) and stage generator (*stage_generator*) in the FFT address generator, the FFT FSM controls the pipelined butterfly data path to compute butterflies in groups and iterate over all the stages. By deducing and analyzing control signals (*C0*, *C1*, ..., and *C5*) from the scheduling and pipelining described in Figure 3.9 and Table 3.1, respectively, we design the FFT FSM to generate the corresponding correct control signals (*ctrl*(4 downto 0)) to supervise the pipelined butterfly unit to operate properly. The FFT FSM also provides memory write control signals (e.g., *wr_fft*), while the FFT address generator provides correct addresses for appropriate data accesses. When the signal, *fft_done* from the address generator, signals that a FFT computation is completed, the FFT FSM shuts down. Subsequently, the Mode FSM begins to operate for the output process (*op_mode*) if the input signal, *op_ready* indicates that the output process is ready.

During the output mode, the controller supervises the FFT address generator to generate correct addresses to read out the results from the RAM. After the output process has completed, the Mode FSM waits for another new input process (*ip_mode*).

3.4 Simulation and Implementation Results

The proposed recursive FFT architecture can be used to design a radix-2 DIT FFT processor with different FFT lengths. In this study, two 16-bit FFT processors based on the proposed FFT architecture using the two-multiplier and three pipelined butterfly unit are implemented in VHDL. One is a 16-point FFT processor, the other

is a 256-point FFT processor. The word length of I/O data and butterfly data path width is 16-bit. To simulate speech and audio processing, these two FFT processors are simulated in the Mentor Graphics ModelSim [63] to do real-time processing at 50% overlap on 16 or 256-point 16-bit two's complement fixed-point data at a sampling rate of 16 kHz. To estimate the timing and area size, these two 16-bit FFT processors are synthesized into the gate level with the AMS (Austria Mikro System International AG) 0.35 μm CMOS digital library in the Synopsys Design Compiler [64]. To estimate the power consumption, these two 16-bit processors are fed with a set of random patterns and simulated at transistor level in the Synopsys NanoSim [65].

In order to perform a fair timing and hardware comparison with other FFT architectures, a 256-point 8-bit FFT processor core is designed and synthesized into the gate level with the AMS 0.35 μm CMOS digital library in the Synopsys Design Compiler.

A. ASIC Design of a 16-point FFT Processor

In this section, a 16-point radix-2 DIT FFT processor based on the proposed FFT architecture is designed at RTL (register transfer level) as a prototype to verify the functionality, hardware cost effectiveness, and power-efficiency of the pipelined two-multiplier and three-adder butterfly data path. For a 16-point FFT, the recursive FFT processor has an 8-word $\times$ 32-bit coefficient ROM and a 16-word $\times$ 32-bit RAM. Both input and output processes require 16 clock cycles. The processor takes 68 cycles for the FFT process. Therefore, the total time for computing a 16-point FFT is 108 cycles, including the I/O processes and some extra overhead operations.

In order to estimate the timing and area size, the 16-point 16-bit FFT design is mapped into the gate level with the AMS 0.35 μm CMOS technology. The estimated maximum clock frequency is 100 MHz and the total time to compute a 16-point FFT is 1,080 ns. As the processing time required for computing an N -point FFT by the

proposed butterfly unit architecture is $(4 + M\log_2 N)$ cycles, the estimated time to complete different-sized FFT (excluding the data I/O processes) by the proposed recursive architecture based on the 0.35 μm CMOS technology are estimated in Table 3.3.

Table 3.3 Estimated processing time for different-sized FFT processors at 100 MHz

Size	16-point FFT	256-point FFT	512-point FFT	1024-point FFT
Clock cycle	68	2,052	4,612	10,244
Time (μs)	0.68	20.52	46.12	102.44

The estimated area size of the 16-point FFT processor and its butterfly unit is 1.18 mm^2 and 0.50 mm^2 , respectively. The two-multiplier and four-adder butterfly unit is also designed and synthesized as a benchmark, and the estimated area size is 0.58 mm^2 . The area contribution from different modules in the two butterfly units are given in Table 3.4 (a) and (b), respectively. In each butterfly unit, two multipliers dominate the data path area, while the remaining adders and control resources consume the rest. As an improved version of the two-multiplier and four-adder benchmark, our proposed butterfly unit saves hardware resources, which include one 32-bit adder, two 32-bit registers, one 16-bit registers, one 16-bit switch, and one 16-bit multiplexer. As a result, the proposed two-multiplier and three-adder implementation has a 13.6% saving in silicon area over the benchmark.

Table 3.4 (a) Estimated area size of the proposed butterfly unit

Module	Adder/Sub (32-bit)	Adder/Sub (16-bit)	Multiplier (16-bit)	DFF REG (16-bit/32-bit)	MUX/SW (16-bit)	Total
Number	1	2	2	9/2	2	1
Total area (mm^2)	0.019	0.022	0.40	0.05/0.021	0.013	0.50
Percentage	3.8%	4.4%	79.4%	14.2%	2.6%	100%

Table 3.4 (b) Estimated area size of the two-multiplier & four-adder benchmark

Module	Adder/Sub (32-bit)	Adder/Sub (16-bit)	Multiplier (16-bit)	DFF REG (16-bit/32-bit)	MUX/SW (16-bit)	Total
Number	2	2	2	10/4	4	1
Total area (mm ²)	0.038	0.022	0.40	0.051/0.042	0.0026	0.579
Percentage (%)	6.6%	3.8%	69.1%	16.1%	4.5%	100%

In order to evaluate the processor in low-power operation, the 16-point FFT processor is mapped into the transistor level and simulated by the NanoSim at 3.3V and 1.1 V, respectively. Table 3.5 shows the power simulation results under two different simulation configurations. The result is given as the power dissipation per 16-point FFT computation, including I/O processes. The 16-point FFT processor implements real-time processing at 50% overlap on 16-point random pattern data at a sampling rate of 16 kHz. The operating clock frequency is 216 kHz. Table 3.5 shows that about 93% of power reduction is obtained by scaling down the supply voltage from 3.3 V to 1.1 V.

Table 3.5 Power dissipation simulation results of the 16-point 16-bit FFT processor

-	Testing 1	Testing 2
Supply Voltage (V)	3.3	1.1
Clock Frequency (kHz)	216	216
Average Power (μ W)	142.9	10

B. ASIC Design of 256-point FFT Processors

In this section, two 256-point FFT processor designs based on the proposed recursive architecture are presented. First, a 256-point radix-2 16-bit FFT processor is designed and implemented for low-cost speech and audio portable applications. This 256-point FFT processor consumes a 128-word $\times$ 32-bit coefficient ROM and a 256-word $\times$ 32-

bit RAM. This processor computes a 256-point FFT in 2,572 cycles, including the I/O processes and some overhead operations.

In order to estimate the timing and area size, the 256-point 16-bit FFT design is mapped into the gate level with the AMS 0.35 μm CMOS technology. The estimated maximum clock frequency is 100 MHz, and the total time to compute a 256-point FFT is 25.720 μs . The estimated area size required by the FFT processor is 9.02 mm^2 . The butterfly unit, RAM, ROM, and other control logic occupy 42.6%, 41.5%, 0.3%, and 3.4% of the total area, respectively. In order to evaluate the low-power operation, the 256-point 16-bit FFT processor is also mapped into the transistor level and simulated by the NanoSim. The 256-point FFT processor performs real-time processing at 50% overlap on 256-point random pattern data at a sampling rate of 16 kHz. This FFT processor completes a 256-point FFT computation within 7.7 ms including the I/O processes, at a clock frequency of 322 kHz and 1.1 V. The estimated power dissipation by this 16-bit FFT processor is 87.1 μW per 256-point FFT computation.

Second, in order to verify the hardware cost efficiency of our proposed recursive FFT architecture, we also design a 256-point 8-bit FFT processor core to compare the proposed FFT architecture with other FFT architectures, including two pipeline architectures and two recursive architectures in [60], [61]. As the comparisons of different FFT implementations in [60], [61] use 8-bit complex multipliers and 16-bit complex adders/subtractors to construct butterfly units, we redesign the proposed radix-2 DIT pipelined butterfly unit by using two 8-bit multipliers and three 16-bit adders to provide reasonable and convincing comparisons.

The corresponding 256-point 8-bit FFT core without data RAM is designed and mapped into the gate level with the AMS 0.35 μm CMOS digital library. The estimated maximum clock frequency is 125 MHz. The estimated area size is 0.27 mm^2 . In terms of 2-input NAND gate, the gate count of the FFT core is 1,532. Table 3.6

shows the comparisons between the 256-point 8-bit FFT processor with other existing processors. As pointed out earlier, all FFT processors in Table 3.6 use 8-bit multipliers (mults) and 16-bit adders/subtractors (adds) in order to provide a fair comparison.

The comparisons of timing performance and hardware complexity among five different FFT architectures are given in Table 3.6. Note that the processing cycles required by the three recursive FFT architectures listed in the table exclude the data I/O processes and overhead operations. As shown in this table, the Rabiner's radix-2 MDC pipeline architecture [55] is the fastest solution, which only takes 128 cycles for computing a 256-point FFT. Bidet proposed an optimized pipeline FFT architecture to increase the hardware utilization [52]. As a result, this pipeline FFT architecture requires about 308 cycles for a radix-2 256-point FFT. These two pipeline architectures compute FFT algorithms much faster than the three recursive FFT architectures. However, these two pipeline FFT processors also have much higher hardware complexity in terms of gate counts. This is because the pipeline architectures employ a chain of butterfly units to do fast pipeline processing, while the recursive architectures only use a single butterfly unit. Therefore, the pipeline FFT architecture with high throughput is suitable for the high-speed OFDM-based communications, such as digital TV broadcasting systems [52].

As shown in Table 3.6, our proposed recursive FFT architecture has the least hardware requirements in terms of the number of arithmetic units and the memory size, among the three recursive FFT architectures. The Delaruelle's processor [68] is a recursive FFT architecture which consists of a radix-4 butterfly unit and a ping-pong memory system. Compared with the radix-2 butterfly unit, this radix-4 butterfly unit increases the butterfly processing speed by using more arithmetic units and memory resources, as shown in Table 3.6. This radix-4 FFT processor also has another disadvantage that it can only compute FFTs with a length of 4 to the power of r . The

Wu's processor [60], [61] uses in-place FFT algorithm and a single radix-2 butterfly unit to reduce the hardware requirements. Although it used the four-multiplier and six-adder butterfly unit, the Wu's processor requires the same number of cycles with our FFT processor because a single-port memory is used in the Wu's processor design. In addition, the critical path of the Wu's processor is approximately the time required for performing one complex multiplication and one complex addition [60]. Our FFT processor also uses in-place FFT algorithm and employs the proposed two-multiplier and three-adder pipelined radix-2 butterfly unit. Hence, the proposed recursive FFT processor has the minimum hardware requirements of memory of size N and only 1,532 gates. Our recursive FFT processor has a faster processing speed than the Wu's processor because the critical path of our processor is only the time required to perform a real multiplication.

Table 3.6 Comparisons of different FFT implementations

-	Rabiner [55]	Bidet [52]	Delaruelle [68]	Wu [60], [61]	Proposed
Architecture	pipeline	pipeline	recursive	recursive	recursive
In-place	Yes	Yes	No	Yes	Yes
No. of BUs	$\log_2 N$ radix-2	$\log_r N$ radix- r	1 radix-4	1 radix-2	1 radix-2
Arithmetic units	$8 \times (\log_4 N - 1)$ mults $(12 \times \log_4 N - 4)$ adds	$12 \times (\log_r N - 1)$ mults $(14 \times \log_r N - 6)$ adds	12 mults 22 adds	4 mults 6 adds	2 mults 3 adds
Gate counts of arithmetic units	-	$8160 \times (\log_r N - 1) + 896 \times \log_r N$	9156	2954	1532
Memory size (complex numbers)	$1.5N - 2$	N	$2N$	N	N
Processing cycles $N = 256$	$0.5N$ 128	$1.2N$ 308	$N \log_4 N$ 1024	$N \log_2 N$ 2048	$N \log_2 N$ 2048

In summary, the above comparisons and discussions demonstrate that our recursive FFT architecture based on the proposed two-multiplier and three-adder

butterfly unit is a good low-cost and low-power FFT realization. Moreover, the proposed recursive FFT architecture is also scalable to compute FFTs with different sizes by modifying the controller and address generator. The sizes of both coefficient ROM and RAM should also change in order to store different point of coefficients and data samples.

3.5 Conclusions

In this chapter, a low-cost FFT architecture has been proposed for low-cost speech and audio portable applications. The objective is to design a 256-point FFT processor for portable speech and audio applications with the constraints of low-cost and low-power operation. The FFT processor is required to perform a 50%-overlap FFT transformation on 16 kHz sampled audio signals. Firstly, we investigated various dedicated FFT architectures for different application requirements. For low-cost portable speech and audio application, the recursive FFT architecture using a single butterfly unit is chosen. To further reduce the hardware requirement, the single-memory system is used in the recursive FFT architecture to buffer the intermediate data during the FFT computation. To have a better SNR and low-cost hardware implementation, we chose the radix-2 DIT Cooley and Tukey algorithm for VLSI implementation of the FFT processor. As the butterfly unit is the crucial computing engine in the recursive FFT architecture, different implementation methods of the radix-2 DIT butterfly unit have been reviewed and compared. As an investigation result, the two-multiplier scheme provides the best trade-off between speed performance and hardware cost. Having rescheduled the radix-2 DIT butterfly arithmetic operations, we proposed a pipelined two-multiplier and three-adder butterfly structure. Compared with the two-multiplier and four-adder benchmark, the proposed method provides a low-cost solution, while maintaining the pipeline

processing throughput. The recursive FFT architecture based on the proposed butterfly unit has been presented. For low-power operation, the supply voltage scaling technique is employed in the FFT processor design to obtain substantial power reduction. The proposed recursive FFT architecture features low hardware complexity, low-power operation and scalability for different-sized FFTs.

Circuit simulation and implementation results showed that the recursive FFT architecture based on the proposed pipelined butterfly unit is a good solution for low-cost and low-power applications. The ASIC implementation verified the hardware cost efficiency of the proposed pipelined two-multiplier and three-adder butterfly structure with comparison to the two-multiplier and four-adder benchmark. The transistor-level power simulation results demonstrated that recursive FFT processor can achieve a significant power reduction by the supply voltage scaling and pipeline processing techniques. At the same time, the processor meets the real-time requirement of the 50% overlap FFT transformation on audio signals at a sampling frequency of 16 kHz. The comparisons with other existing FFT architectures also verified the proposed recursive FFT architecture is a good low-cost solution.

In conclusion, a recursive FFT architecture with a single pipelined butterfly unit and a single memory has been proposed for low-cost realization and low-power operation. The circuit simulation and implementation results indicate that this single-butterfly FFT architecture meet the tight requirements of low-cost implementation and low-power operation for portable speech and audio applications.

Chapter 4

Algorithm and Architecture of Critical-Band Transform

4.1 Introduction

In Chapter 1, we discussed the development of efficient architectures for the fast Fourier transform (FFT), critical-band transform (CBT) and discrete wavelet packet transform (DWPT), which are suitable for VLSI implementation. In Chapter 3, we presented our research on VLSI architecture of FFT for portable speech and audio applications. In this chapter, we will describe the design and implementation of an efficient CBT processor for low-power speech applications.

The spectral analysis is one of the most fundamental operations in the field of acoustic and speech signal processing. It transforms the time-domain acoustic signal into its frequency-domain spectrum. Some traditional methods, such as FFT, short-time Fourier transform (STFT), and filterbank (a group of bandpass filters), have been widely used in academia and industry. These methods usually have a constant frequency resolution. However, as we described in Chapter 2, psychoacoustical studies show that the human ear performs spectral analysis on the acoustic signal in the form of a filterbank with nonuniform critical bandwidths [38]. As shown in Table 2.1, for a wide-band speech signal with a bandwidth of 8 kHz, there are 21 critical bands for the Bark scale described by Zwicker [40] and 24 bands for the Mel scale

[34]. An interesting finding is that the bandwidths of the critical bands with center frequencies below a certain frequency are approximately constant. The bandwidths are around 100 Hz for center frequencies below 500 Hz and 1 kHz in the Bark scale and the Mel scale, respectively. Above 500 Hz in the Bark scale (or 1 kHz in the Mel scale), the bandwidths increase as the center frequencies increase, while the Q factors of these bandpass filters are approximately constant. Motivated by the human auditory perception model, many methods have been developed to approximate the critical-band analysis. These methods provide advantages over other traditional ways including the FFT and STFT methods, in speech applications, especially in the fields of speech recognition, speech coding, and speech enhancement.

In the past two decades, various schemes to implement critical-band analysis have been proposed for speech applications. These methods can be classified into four main approaches: (i) direct digital implementation of the critical-band filterbank [69], [70], (ii) FFT method [39], (iii) constant-Q transform (CQT) method [71], [72] and (iv) wavelet packet transform (WPT) method [73], [74]. However, these CBT methods have disadvantages of high computational complexity, complex hardware architecture, or inaccurate approximation to the critical-band scale. We will review these existing methods in the next section. Recently, low-power VLSI speech systems, such as speech recognizers, speech codecs, and healthcare devices, have many promising applications in large volume battery-powered portable products, for example, personal digital assistants, communicators, smart toys, and hearing aids. The front-end spectral analysis in speech applications, such as the FFT, filterbank and critical-band analysis methods, is both computational-intensive and memory-intensive, which leads to significant power consumption. The disadvantages of these existing CBT methods make them not suitable for low-power VLSI realization. Therefore, design and implementation of efficient front-end spectral analyzers for low-power speech systems is required.

In this chapter, an efficient algorithm and its corresponding VLSI architecture for the CBT are developed to approximate the critical-band filtering of human ear. From the observation of the critical-band scale, we formulate a generalized CBT algorithm, which consists of a constant-bandwidth transform in the lower frequency range and a Brown CQT in the higher frequency range. By finding a set of suitable parameters, the proposed CBT can obtain a close fitting to a particular critical-band scale, such as the Bark scale or the Mel scale. The novel CBT scheme has a regular data structure and a smaller memory requirement compared with the other CBT methods [34], [69]–[76]. Besides, it also requires fewer computations and memory accesses. All these advantages make the proposed CBT algorithm suitable for VLSI implementation.

To meet the tight requirements of low power, low voltage, and small silicon area, we apply some of the low-power techniques reported in Chapter 2 by exploiting inherent properties in the proposed CBT algorithm. These properties include the coefficient symmetry, parallelism, regularity, recursiveness of the CBT algorithm. The proposed CBT architecture reduces the computational complexity in terms of arithmetic operations and memory read/write operations by exploiting the symmetry property of the CBT coefficients. The proposed CBT architecture explores the inherent parallelism in the CBT data flow and uses the parallel and pipeline processing to increase the throughput. Therefore, a lower supply voltage and a slower clock frequency can be used to achieve significant power reduction. Similar to the proposed recursive FFT architecture presented in Chapter 3, this CBT architecture also employs a single time-multiplexed data path to compute different critical-band spectra recursively in order to reduce the hardware requirement.

In this chapter, theoretical analysis, circuit simulation and implementation results are presented to verify the efficiency of the proposed CBT algorithm and architecture. A 21-band Bark scale CBT processor with a sampling rate of 16 kHz is designed and simulated. Simulation results verify its suitability for performing short-time spectral

analysis on speech. Comparative studies also show that the 21-band CBT has a better fitting to the human critical-band analysis, significantly fewer computations, fewer memory operations, and therefore, is more energy-efficient than other methods. With a $0.35\ \mu\text{m}$ CMOS technology, the CBT processor computes 160-point samples of speech signal in 4.99 ms at a clock frequency of 234 kHz. The power dissipation required to perform a 160-point CBT is $15.6\ \mu\text{W}$ at 1.1 V. The proposed CBT processor achieves 82.1% power reduction as compared to the benchmark 256-point FFT processor presented in Chapter 3.

The organisation of this chapter is as follows. In Section 4.2, we explore different methods to implement the critical-band analysis. We then present the proposed CBT algorithm and its corresponding VLSI architecture and circuit design in Section 4.3. In Section 4.3, we also present the short-time spectral analysis of two typical speech phonemes and the hardware requirements for a 21-band Bark scale CBT. We evaluate the efficiency of the proposed CBT architecture by designing and simulating the 21-band CBT processor, and comparing it against a benchmark 256-point FFT processor. In Section 4.4, circuit simulation and implementation results are reported and discussed. Finally, conclusions are presented in Section 4.5.

4.2 Review of Existing Critical-Band Analysis Methods

As described in Chapter 2, many multimedia and artificial intelligence applications benefit greatly from the studies of human auditory system and human visual system. Specifically, many speech and audio applications including speech recognition, speech/audio coding, and speech recognition, take advantages of the psychoacoustical studies. These applications use various critical-band analysis methods to approximate the critical-band filtering of the human ear. These CBT methods are classified into four main approaches and described as follows.

A. Direct Digital Implementation of the Critical-Band Filterbank

The most common method to implement the human ear critical-band analysis is the direct digital implementation of a nonuniform filterbank along a critical-band scale. The so-called critical-band filterbank is a bank of bandpass filters, which has uniform bandwidths (around 100 Hz) until about 500 Hz (or 1 kHz) and has logarithmic bandwidths above 500 Hz (or 1 kHz). The design of these bandpass filters should follow the specification of the Bark scale or Mel scale as depicted in Table 2.1. As it can achieve the best approximation to the human ear filtering, the critical-band filterbank method provides good results in the application of speech recognition [34], [69], [70], [75]. However, this method has high hardware complexity and should also follow the filterbank design methodologies [34], [69] strictly.

B. FFT Method

The FFT and STFT can only give a constant frequency resolution, which does not match the human auditory system. However, the CBT spectral magnitude of each critical band can be obtained by calculating the weighted sum of the FFT magnitude coefficients within the critical band in question. For example, the FFT-based method is used to calculate the masking threshold in perceptual audio coding [76]. With the spectrum $\text{Re}(\omega)$ and $\text{Im}(\omega)$ of a signal from the FFT, we can calculate the power spectrum of the signal as

$$P(\omega) = \text{Re}^2(\omega) + \text{Im}^2(\omega). \quad (4-1)$$

The power spectrum is then partitioned into critical bands as defined by the Bark scale or Mel scale. Thus, we sum up the energies in each critical band as

$$B_i = \sum_{\omega=bl_i}^{bh_i} P(\omega), \quad (4-2)$$

where bl_i and bh_i represent the lower boundary and upper boundary of critical band i , respectively, and B_i is the energy in critical band i . This discrete critical-band analysis provides a close approximation to the human ear critical-band filtering.

The FFT-based critical-band analysis has been successfully used in the applications of speech/audio coding [39], [76] and speech recognition [70]. However, this FFT-based method requires extra postprocessing in the FFT spectrum. As the FFT length is constant across all center frequencies of critical bands, the FFT-based method provides a significant lower temporal resolution for higher frequencies when compared to other methods. Thus, the approximation to the human ear critical-band filtering is not accurate.

C. Constant-Q Transform Method

The constant-Q transform provides nonuniform frequency bandwidths, which increase as center frequencies increase. As described in Section 4.1, the critical-band scale has uniform bandwidths in the lower frequency range (below 500 Hz or 1 kHz) and has nonuniform bandwidths in the higher frequency range. The CQT-based methods take advantage of this observation to mimic the critical-band filtering.

One of the CQT-based methods [71] uses constant-Q filters to approximate the critical-band filtering in the high frequency range. In the lower frequency range, the constant-bandwidth coefficients are obtained by summing the constant-Q filters coefficients within each constant-bandwidth band. Another CQT-based method [72] employs the chirp z-transform to approximate the critical-band filtering in the higher frequency range. It uses the FFT to compute the constant-bandwidth coefficients in the lower frequency range. These two methods give a close approximation to the critical-band scale. But they are computationally expensive and involve complex hardware architectures.

D. Wavelet Packet Transform Method

As the generalization of the discrete wavelet transform (DWT), the wavelet packet transform not only inherits the useful properties of the DWT reported in Chapter 2, but also offers a flexible time and frequency (TF) decomposition. Due to the characteristic of flexible TF decomposition, WPT can be used to approximate the human ear critical-band filtering.

Because of its good inherent characteristics, the WPT-based method has been widely used in speech and audio applications [73], [74], [77]. In [73], an efficient approach based on the fast orthogonal WPT is proposed to use a tree structure to decompose the input speech signal into the approximated critical bands. However, the WPT-based method provides an inaccurate approximation to the critical-band scale as WPT divides frequency bands only by a factor of two at each decomposition level. In addition, this method also involves high hardware complexity because the WPT algorithm requires 2^j wavelet filter pairs at each decomposition level j .

4.3 The Proposed Algorithm and Architecture of Critical-Band Transform

4.3.1 Critical-Band Transform Algorithm

Based on the observation of the critical-band scale depicted in Section 4.1, a novel critical-band transform algorithm is proposed to approximate the critical-band filtering of the human ear. This proposed CBT consists of two transforms: a constant-Q transform (CQT) in the higher frequency range and a constant-bandwidth transform (CBWT) in the lower frequency range. In this study, the Bark scale is approximated to demonstrate the efficiency of the proposed CBT algorithm.

In the proposed CBT, the Brown CQT algorithm [43] is employed to approximate the critical-band filtering in the higher frequency range, which is above 500 Hz for the Bark scale. The results in this study show that the Brown CQT with low Q values is a suitable algorithm for speech signal processing. The Brown CQT is also more efficient than the other constant- Q analysis methods. As described in Chapter 2, Brown derived an efficient constant- Q transform from the discrete short-time Fourier transform, which has a constant ratio (Q) of center frequency to frequency resolution. As the resolution Δf of the discrete Fourier transform is equal to the sampling rate (SR) divided by the window size (the number of samples analyzed in the time domain), the window size in the Brown CQT varies inversely with frequency in order to achieve a constant Q . Thus, in the Brown CQT, the frequency resolution decreases while the center frequency increases, which is similar to the critical bandwidth in the higher frequency range. By choosing a suitable Q value, the Brown CQT can achieve a close fitting to the critical bandwidths in the higher frequency range.

For the lower frequency range below 500 Hz, a CBWT is employed to approximate the critical bands with constant bandwidths. The CBWT is also implemented by using the Brown CQT. However, the window size in this Brown CQT is constant. This is because, given a constant window size across the lower frequency range, the Brown CQT provides constant bandwidths. Therefore, the CBWT is formally expressed as

$$X[k_{cw}] = \frac{1}{N_c} \sum_{n=0}^{N_c-1} w[n]x[n] \exp\{-j2\pi Q_{k_{cw}} n / N_c\}, \quad k_{cw} = 1, 2, \dots, n_{cw}. \quad (4-3)$$

Note that the window size N_c in the CBWT is constant, while the window size in the original Brown CQT varies for different bands. In contrast to the original Brown CQT, the Q value, $Q_{k_{cw}}$, in the Brown implementation of the CBWT is not constant. $Q_{k_{cw}}$ is different for n_{cw} constant bandwidths of the CBWT.

In CBWT, the window size is equal to the sampling rate divided by the frequency resolution of 100 Hz (Δf_c),

$$N_c = SR / \Delta f_c = (SR / f_{k_{cw}}) Q_{k_{cw}} = \text{constant} . \quad (4-4)$$

In accordance with the Brown CQT, the CBWT is normalized by scaling with a factor of N_c . The center frequency $f_{k_{cw}}$ of the k_{cw} th spectral component varies linearly with k_{cw} , and is given as

$$f_{k_{cw}} = f_{minc} + \Delta f_c (k_{cw} - 1) , \quad (4-5)$$

where f_{minc} is the minimum center frequency in the lower frequency range of the proposed CBT. In contrast, the center frequency in the Brown CQT is exponential in k_{cq} .

As both the CQT and CBWT in the CBT can be expressed in the Brown CQT form, the proposed CBT algorithm is expressed as follows:

$$X[k_{cb}] = \begin{cases} X[k_{cw}], & k_{cb} = k_{cw} = 1, 2, 3, \dots, n_{cw}; \\ X[k_{cq}], & k_{cb} = k_{cq} = n_{cw} + 1, \dots, n_{cw} + n_{cq}. \end{cases} , \quad (4-6)$$

where n_{cw} , n_{cq} are the number of critical bands in the lower and higher ranges, respectively. The CBT covering the whole frequency range can be rearranged into one equation as

$$X[k_{cb}] = \frac{1}{N[k_{cb}]} \sum_{n=0}^{N[k_{cb}]-1} w[k_{cb}, n] x[n] \exp\{-j2\pi Q_{k_{cb}} n / N[k_{cb}]\}, k_{cb} = 1, 2, \dots, n_{cw} + n_{cq}. , \quad (4-7)$$

where $X[k_{cb}]$ is the k_{cb} th spectral component of the CBT. Here, $x[n]$ is the discrete-time input speech signal with time index n and $w[k_{cb}, n]$ is a window function for each critical band. The length of each window is $N[k_{cb}]$.

In the proposed CBT, the fixed bandwidth in the low frequency range and constant-Q bandwidths in the higher frequency range are defined as

$$\Delta f_{k_{cb}} = \begin{cases} \Delta f_c = 100, & k_{cb} = 1, 2, \dots, n_{cw}; \\ (2^{1/s})^{[k_{cb} - (n_{cw} + 1)]} \times \Delta f_{minq}, & k_{cb} = n_{cw} + 1, \dots, n_{cw} + n_{cq}. \end{cases}, \quad (4-8)$$

where s is the number of constant-Q bands per octave. The k_{cb} th center frequency in the CBT is expressed as

$$f_{k_{cb}} = \begin{cases} f_{minc} + \Delta f_c (k_{cb} - 1) = 50 + 100 \times (k_{cb} - 1), & k_{cb} = 1, 2, \dots, n_{cw}; \\ (2^{1/s})^{[k_{cb} - (n_{cw} + 1)]} \times f_{minq}, & k_{cb} = n_{cw} + 1, \dots, n_{cw} + n_{cq}. \end{cases}. \quad (4-9)$$

Note that 50 Hz is chosen to be the center frequency of the lowest critical band f_{minc} . f_{minq} and Δf_{minq} are the minimum center frequency and bandwidth in the higher frequency range, respectively. The Q factor of the CBT, $Q_{k_{cb}}$, is therefore described by

$$Q_{k_{cb}} = f_{k_{cb}} / \Delta f_{k_{cb}} = \begin{cases} f_{k_{cb}} / 100, & k_{cb} = 1, 2, \dots, n_{cw}; \\ 1 / (2^{1/s} - 1), & k_{cb} = n_{cw} + 1, \dots, n_{cw} + n_{cq}. \end{cases}. \quad (4-10)$$

In order to reduce spectral leakage, a Hamming window is chosen as the window function $w[k_{cb}, n]$. The length of each window for each critical band of the CBT is determined by

$$N[k_{cb}] = SR / \Delta f_{k_{cb}} = \begin{cases} SR / 100, & k_{cb} = 1, 2, \dots, n_{cw}; \\ (SR / f_{k_{cb}}) Q_{k_{cb}}, & k_{cb} = n_{cw} + 1, \dots, n_{cw} + n_{cq}. \end{cases}. \quad (4-11)$$

A comparison between the various parameters used in the CBWT and CQT of the proposed CBT is given in Table 4.1.

By combining the Hamming window $w[k_{cb}, n]$ and the exponential part into $kern[k_{cb}, n]$, we can compute the critical-band spectrum using only multiplications and accumulations directly from the input speech data and the pre-computed coefficients in (4-12).

$$\begin{aligned}
X[k_{cb}] &= \sum_{n=0}^{N[k_{cb}]-1} x[n] \left\{ \frac{w[k_{cb}, n]}{N[k_{cb}]} \exp\{-j2\pi Q_{k_{cb}} n / N[k_{cb}]\} \right\} \\
&= \sum_{n=0}^{N[k_{cb}]-1} x[n] \text{ker}[k_{cb}, n], \quad k_{cb} = 1, 2, \dots, n_{cb}.
\end{aligned} \tag{4-12}$$

Table 4.1 Comparison of the parameters in CBWT and CQT

CBT	CBWT	CQT
Range	Low frequency range of CBT	High frequency range of CBT
Frequency	$f_{\min c} + (k_{cw} - 1) \Delta f_c$ Linear in k_{cw}	$(2^{1/5})^{[k_{cq} - (n_{cw} + 1)]} f_{\min q}$ Exponential in k_{cq}
Window size	N_c (constant)	$N[k_{cq}] = SR \times Q_{cq} / f_{k_{cq}}$ (variable)
Bandwidth	SR / N_c (constant)	$f_{k_{cq}} / Q_{cq}$ (variable)
Ratio of frequency to bandwidth	$Q_{k_{cw}}$ (variable)	Q_{cq} (constant)

The evaluation process of the CBT approximating a particular critical-band scale can be generalized and described step-by-step as follows:

- (1) Choose a critical-band scale, such as the Bark scale or Mel scale, and then determine the numbers of critical bands ($n_{cw} + n_{cq}$) in the Nyquist bandwidth dependent on the sampling rate.
- (2) According to the chosen critical-band scale, assign a transition frequency (500 Hz for the Bark scale or 1 kHz for the Mel scale) to divide the whole bandwidth into two frequency ranges for the CBWT and Brown CQT, respectively. The number of the critical bands n_{cw} and n_{cq} in the two frequency ranges can then be determined.
- (3) Find a suitable Q_{cq} value for the CQT to achieve a close fitting to the critical bands. In the CBWT, the Q factor, $Q_{k_{cw}}$, is different for each n_{cw} constant bandwidths.

- (4) Calculate the center frequency (4-9) by using the minimum center frequency f_{minc} and f_{minq} in the critical-band scale for the two ranges, respectively.
- (5) Determine the length of each window $N[k_{cb}]$ using (4-11).
- (6) By combining the Hamming window $w[k_{cb}, n]$ and exponential part to $kern[k_{cb}, n]$, we can compute the critical-band spectral components using only multiplications and accumulations directly from the input speech data and the pre-computed coefficients in (4-12).

In this study, a 21-band Bark scale CBT with 5 constant-bandwidth bands (100 Hz), and 16 constant-Q bands ($Q = 5.6$) is constructed at a sampling rate of 16 kHz. The parameter values are chosen so that the 21-band CBT closely approximates the Bark scale. For the Mel scale, there are 10 constant-bandwidth bands, and 14 constant-Q bands with $Q = 6.9$.

4.3.2 Performance Analysis

In this section, the performance of the proposed 21-band Bark scale critical-band transform is evaluated and compared with the WPT method, followed by the simulation of short-time spectral analysis on speech signal in MATLAB [78]. Table 4.2 presents the center frequencies and bandwidths of the proposed CBT and the Bark scale [40]. It can be seen from the table that the approximation by the proposed CBT is very close to the benchmark Bark scale. Figure 4.1 shows the degree of approximation to the Bark scale critical bands both for the proposed CBT and for the WPT method [73]. It shows that the proposed CBT provides a closer approximation to the Bark scale, especially in terms of the bandwidths. This is because the WPT method uses a 6-level WPT to approximate the Bark scale. At each level, the WPT can only divide the frequency bandwidths by a factor of two.

Table 4.2 Bandwidths and center frequencies of 21-band CBT and the Bark scale

CB rate (Bark)	Munich Bark scale [40]		21-band CBT with $Q_{eq}=5.6$		
	Center frequency (Hz)	Bandwidths (Hz)	Center frequency (Hz)	Bandwidths (Hz)	Windows (samples)
1	50	80	50	100	160
2	150	100	150	100	160
3	250	100	250	100	160
4	350	100	350	100	160
5	450	110	450	100	160
6	570	120	570	101.8	157
7	700	140	671.8	120	133
8	840	150	791.7	141.4	113
9	1000	160	933.1	166.7	95
10	1170	190	1099.8	196.4	81
11	1370	210	1296.1	231.5	69
12	1600	240	1527.6	272.9	58
13	1850	280	1800.4	321.6	49
14	2150	320	2121.9	379.0	42
15	2500	380	2500.8	446.7	35
16	2900	450	2947.4	526.5	30
17	3400	550	3473.7	620.6	25
18	4000	700	4094	731.4	21
19	4800	900	4825	862	18
20	5800	1100	5686.7	1016	15
21	7000	1300	6702.1	1197.4	13

Figure 4.1 Degree of Approximation to the Bark scale

The 21-band CBT algorithm has been programmed and simulated in MATLAB. A typical utterance “ka” [72] is used in our testing. The syllable “ka” consists of two 600-ms waveforms for “k” and “a”, respectively. This 1,200-ms speech spoken by a male speaker was recorded in a small room and processed by Cool Edit Pro 2.0 [79] at a sampling rate of 16 kHz. Figures 4.2 (a) and (b) show the first and second 600-ms waveforms of the utterance “ka”, respectively. The 21-band CBT uses 50%-overlap processing on the 160-point segments of the speech. The CBT spectra of the two speech waveforms are shown in Figures 4.3 (a) and (b), respectively. The corresponding FFT spectra are given in Figures 4.4 (a) and (b), respectively. These plots show the short-time spectral magnitude on the z-axis against the frequency in log scale on the x-axis. The labels on the y-axis correspond to the speech duration in second.

In the first 600 ms in Figure 4.3 (a), the initial burst of energy of the plosive “k” has a concentration of energy in the region near 2 kHz. The energy peak at the very low frequency range is also observed in the FFT spectra as shown in Figure 4.4 (a). It is commonly observed in spectral analysis of the speech signal. In the second 600 ms in Figure 4.3 (b), a clear formant structure for the vowel “a” can be observed with the first and second formant frequencies around 650 Hz and 1,100 Hz, respectively. The third formant is around 2,500 Hz. These formant frequencies are the typical features of the vowel “a” [80]. The short-time spectra as shown in Figure 4.3 for the CBT follow closely those obtained by a 256-point FFT as shown in Figure 4.4. Similar to the Brown CQT, the proposed CBT is not invertible. However, this proposed CBT is adequate to show the typical spectral features of the phonemes, as demonstrated in the short-time spectral analysis of the speech. As the pitch is ignorable and the higher frequency information is less significant in some speech applications as stated in [81], the critical-band analysis based on the Bark scale or Mel scale may work effectively in these applications, for example, speech recognition [34], [69]. This is mainly

because the critical-band analysis can still capture the phonetic part of the speech signal, as shown in Figure 4.3.

(a)

(b)

Figure 4.2 (a) First 600-ms of “ka”; (b) second 600-ms of “ka”.

In summary, the 21-band CBT can provide a close fitting the Bark scale, but also identify important temporal and spectral features of the speech signal. As it performs spectral analysis of speech satisfactorily, the proposed CBT can be used as an auditory spectral analyzer in speech applications.

(a)

(b)

Figure 4.3 (a) CBT analysis of the first 600-ms of “ka”; (b) CBT analysis of the second 600-ms of “ka”.

(a)

(b)

Figure 4.4 (a) FFT analysis of the first 600-ms of “ka”; (b) FFT analysis of the second 600-ms of “ka”.

4.3.3 Critical-Band Transform Architecture

In this section, an efficient VLSI architecture is presented for the proposed critical-band transform. By exploiting the symmetry property of the CBT coefficients, the number of multiplications of the CBT can be reduced by about 50%. In the proposed architecture, the derived CBT data path is simple and regular, which make it easy to be pipelined and parallelized. This parallel and pipeline processing allows the CBT processor to apply the supply voltage scaling technique and use a low frequency clock rate to achieve substantial power reduction, while still maintaining real-time processing of speech signal. Thus, the proposed CBT architecture is suitable for ASIC implementation.

It is observed from (4-12) that there is a symmetry property of the CBT coefficient *kern*. The coefficient consists of a real part (the cosine function) and an imaginary part (the sine function). Applying the symmetry property of the cosine function and the antisymmetry property of the sine function, the CBT equation can be rearranged as

$$\begin{aligned}
 X[k_{cb}] &= \sum_{n=0}^{N[k_{cb}]-1} x[n](\cos[k_{cb},n] + j * \sin[k_{cb},n]) \\
 &= \begin{cases} (x[0] \pm 0)kern[0] + \sum_{n=1}^{M[k_{cb}]} \{(x[n] + x[N - n])\cos[k_{cb},n] + j * (x[n] - x[N - n])\sin[k_{cb},n]\}, & \text{when } N[k_{cb}] \text{ is odd;} \\ (x[0] \pm 0)kern[0] + \sum_{n=1}^{M[k_{cb}]-1} \{(x[n] + x[N - n])\cos[k_{cb},n] + j * (x[n] - x[N - n])\sin[k_{cb},n]\} \\ + (x[M] \pm 0)kern[M], & \text{when } N[k_{cb}] \text{ is even.} \end{cases}
 \end{aligned} \tag{4-13}$$

where

$$M[k_{cb}] = \begin{cases} (N[k_{cb}] - 1)/2, & N[k_{cb}] \text{ is odd;} \\ N[k_{cb}]/2, & N[k_{cb}] \text{ is even.} \end{cases} \tag{4-14}$$

As shown in the above equations, there are two operation modes for calculating the CBT spectrum of each critical band, when the window lengths are odd and even, respectively. By inserting zeroes into the equation, we can derive regular expressions as described by (4-13). For simplicity, this equation can be simplified and expressed as follows

$$X[k_{cb}] = \sum_{n=0}^{M[k_{cb}]} \{(x[n] + x[m])\cos[k_{cb},n] + j * (x[n] - x[m])\sin[k_{cb},n]\}. \quad (4-15)$$

Therefore, the number of multiplications and memory usage are reduced by about 50%, as described in (4-13) and (4-14). These savings contribute not only to the reduction of the processing time but also to the power consumption saving. The data flow graph of the modified CBT algorithm is derived from (4-13) and (4-15), as shown in Figure 4.5. The CBT spectral component for each critical band, which consists of the real part denoted by *cbr* and imaginary part denoted by *cbi*, is obtained after all additions (or subtractions) and multiplier-accumulator (MAC) operations over a corresponding window of input speech samples have completed. We denote the addition and MAC process of a pair of data elements as one butterfly operation. In other words, one butterfly operation of the CBT involves two additions/subtractions and two subsequent MAC operations of two data elements as described in Figure 4.5. Therefore, given a frame of speech samples, calculation of the k_{cb} th CBT spectrum requires $M[k_{cb}]$ butterfly operations.

Figure 4.5 Data flow graph of the CBT algorithm

As shown in (4-13), the computation of different critical-band spectral components is decomposed into a series of basic CBT butterfly operations, which has simple and regular data flow structure described in Figure 4.5. Considering the speech sampling rate of 16 kHz, and the small area and low cost requirement, we propose a recursive architecture as shown in Figure 4.6, which employs a single data path to compute the CBT butterflies so as to minimize the hardware requirements. Similar to the recursive FFT architecture in Chapter 3, this proposed CBT architecture uses a single time-multiplexed data path to compute different critical bands by exploiting the recursiveness property in the modified CBT algorithm.

Figure 4.6 Block diagram of the proposed CBT processor

The proposed VLSI architecture for the CBT consists of a data path, a controller, a coefficient ROM, a data input RAM, a data output RAM, and an address generator, as depicted in Figure 4.6. The operation of the CBT processor is partitioned into data I/O process (I/O mode) and CBT computation process (CBT mode). During the computation process, the data path is time-multiplexed to compute different critical bands, recursively. Supervised by the controller, the data path sequentially computes butterfly operations by processing speech samples buffered in the input RAM with coefficients in the ROM. After one critical-band spectral component has been computed, the results are stored into the output RAM. When the n_{cb} -band CBT has completed, a new frame of speech samples is buffered in the input RAM in the subsequent I/O process, while the final CBT results are read out from the output RAM.

In the following sections, we will present architecture and circuit design of the proposed recursive CBT processor. In this study, the I/O data and coefficients are expressed in the 16-bit two's complement fixed-point format.

A. Pipelined CBT Data Path

From the CBT data flow depicted in Figure 4.5, we derive a two-multiplier and four-adder pipelined CBT data path as shown in Figure 4.7. The data are processed in two parallel paths. The efficient pipeline and parallel processing makes it possible to utilize the supply voltage scaling approach to achieve significant power reduction. The CBT data path has three pipeline stages to improve the processing throughput. In the first stage, the first pair of 16-bit wide adders processes two data elements from the input RAM. Subsequently, the two multipliers in the second stage compute 16-bit $\times$ 16-bit multiplications and produce 32-bit results for each multiplier. In the last stage, the second pair of 32-bit wide adders performs the accumulations. The final results are truncated into 16-bits and written to the output RAM. As shown in Figure 4.7, the control signals, $c1$, $c2$, $c3$ and cw are used to control the corresponding pipeline registers, while the reset signals, $rst1$ and $rst3$ are used to reset the associated registers properly.

Figure 4.7 Proposed pipelined CBT Data Path

The pipeline processing of butterfly operations in the proposed CBT data path is described in Table 4.3. In particular, the first two CBT butterfly operations for each critical band are described in Table 4.3 (a). In the first butterfly operation for each critical band, only one data element is read from the input RAM and fed into one of the first pair of pipeline registers. At the same time, the other register is reset to zero, which follows the corresponding expression $(x[0] \pm 0)kern[0]$ in (4-13). The last CBT butterfly operations for each critical band are described in Table 4.3 (b) and (c), for the odd and even window sizes, respectively. As shown in these two tables, the CBT data path has two working modes, that is, even mode and odd mode. This is because the last butterfly operation might be different for individual critical bands. For the odd window length, a pair of data elements is read from the input RAM. But when the window size is even, only one data element is read and stored into one of the first pair of registers, while the other register is reset to zero, which follows the expression of $\{ (x[M] \pm 0)kern[M] \}$ in (4-13). Both in the even and odd modes, the other butterfly operations between the first and the last butterflies are the same with the second butterfly operation as described in Table 4.3.

Table 4.3 Pipeline table of the CBT data path

(a) First two butterfly operations for each critical band

RAM Read	First 2 Adds	Two Multiplications	Second 2 Adds	-
$x[n]$	$x[n] + 0$ $x[n] - 0$ read $kern$	$x[n] \times cos$ $x[n] \times sin$	2 accumulations	-
-	RAM Read	First 2 Adds	Two Multiplications	Second 2 Adds
-	$x[n], x[m]$	$x[n] - x[m]$ $x[n] + x[m]$ read $kern$	$(x[n] - x[m]) \times kern$ $(x[n] + x[m]) \times kern$	2 accumulations

Chapter 4

Algorithm and Architecture of Critical-Band Transform

(b) Last CBT butterfly operation when window size $N[k_{cb}]$ is odd

RAM Read	First 2 Adds	Two Multiplications	Second 2 Adds	RAM Write
$x[n], x[m]$	$x[n] + x[m]$ $x[n] - x[m]$ read $kern$	$(x[n] - x[m]) \times kern$ $(x[n] + x[m]) \times kern$	2 accumulations	cbr, cbi

(c) Last CBT butterfly operation when window size $N[k_{cb}]$ is even

RAM Read	First 2 Adds	Two Multiplications	Second 2 Adds	RAM Write
$x[n]$	$x[n] + 0$ $x[n] - 0$ read $kern$	$x[n] \times cos$ $x[n] \times sin$	2 accumulations	cbr, cbi

As described in (4-13) and (4-14), for a particular CBT spectrum, there are $(N[k_{cb}] - 1)/2 + 1$ butterfly operations when $N[k_{cb}]$ is odd, or $(N[k_{cb}]/2 + 1)$ butterflies when $N[k_{cb}]$ is even. Thus, it takes the pipelined data path $(N[k_{cb}] - 1)/2 + 4$ cycles to compute a CBT spectrum when $N[k_{cb}]$ is odd, and $(N[k_{cb}]/2 + 4)$ cycles when $N[k_{cb}]$ is even, including read/write accesses of the I/O memories. By summing up all the CBT spectrum computation time of each critical band, the total computation time for one n_{cb} -band CBT is expressed as

$$T_{CBT} = \sum_{n_{odd}} T_{odd} + \sum_{n_{even}} T_{even} , \quad (4-16)$$

where $\begin{cases} T_{odd} = (N[k_{cb}] - 1)/2 + 4, \\ T_{even} = N[k_{cb}]/2 + 4. \end{cases}$, and n_{odd} and n_{even} are numbers of the critical bands

which have odd and even window sizes, respectively.

The proper pipeline processing with the two working modes is controlled by a controller. By multiplexing the CBT data path, CBT spectra are computed one by one from band 1 to band n_{cb} . Timing for the pipelined CBT operation is described in Figure 4.8. Supervised by the CBT controller, the coefficient ROM provides the correct coefficients to the data path, while the I/O RAMs provide the data path input speech samples and buffer the final CBT results. The VHDL structure design of the proposed CBT data path is provided in Appendix B.

Figure 4.8 Pipeline diagram of the proposed CBT data path

B. CBT Address Generator

The address generator in the CBT processor provides correct addresses for the I/O RAMs and the coefficient ROM during in the I/O process and CBT process. Mode signals, *io_mode* and *cbt_mode* are provided by the CBT controller. The block diagram of the CBT address generator with main signals is shown in Figure 4.9. The CBT address generator consists of the critical-band generator (*cb_generator*) and the address generation unit (*xcb_agu*).

During the CBT computation process, the critical-band generator keeps track of which CBT spectrum is being computed. Please refer to the VHDL design of the critical-band generator for the 21-band CBT processor, as provided in Appendix B. This generator also provides the CBT controller and the address generation unit (*xcb_agu*) with the information of each critical band, including the number of the butterfly operations (*No_bfly*), parity of the window size (*odd_even*), and the offset values (*offsets*) for calculating correct addresses in the CBT mode. This critical-band information is pre-stored in a ROM in the critical-band generator when a particular n_{cb} -band CBT is determined. There is a critical-band counter in the critical-band generator. When the critical-band count is n_{cb} , this counter generates a signal named “*cbt_done*” to inform the controller that the CBT process has completed.

Figure 4.9 Block diagram of the CBT address generator

The address generation unit (*xcb_agu*) is used to calculate correct addresses for the coefficients ROM in CBT mode and for the I/O RAMs in both CBT and I/O modes. This address generation unit consists of two main generators: CBT address generator and I/O address generator. During the CBT process, there are four counters operating in the CBT address generator for addressing the I/O RAMs and coefficients ROM. During the I/O process, the I/O address generator utilizes only one counter for addressing the I/O RAMs. When this count reaches the input frame length, it generates a signal named “*io_done*” to inform the controller that the I/O process has completed.

C. CBT Controller

This CBT controller supervises other functional units in the processor for proper operation. Depending on the feedback signals from other units, the controller sends out control signals to determine the sequence of events.

The controller consists of two finite state machines (FSMs) called I/O FSM and CBT FSM, and one butterfly generator. The I/O FSM is responsible for the I/O process. When the *io_done* from the CBT address generator informs the I/O process is completed, the CBT FSM begins to operate for the CBT process. The butterfly generator keeps track of which butterfly in a particular critical band is being processed.

When all butterflies of a critical band have been computed, this butterfly generator sends the signal, ϕ to increase the critical-band counter of the critical-band generator in the CBT address generator. Thus, the CBT processor can iterate over all the n_{cb} critical bands. When the signal, cbt_done from the CBT address generator indicates that a CBT is completed, the CBT FSM shuts down and the I/O FSM begins to operate for the subsequent I/O processing.

The other functional units are the coefficients ROM storing the coefficients $kern$, and the I/O RAMs buffering the input speech data and output CBT spectra. The capacity of the memories is determined when a particular critical band scale is chosen to be approximated by the proposed CBT algorithm.

The proposed recursive CBT architecture can be used to compute a CBT based on different critical-band scales and sampling rates. In this study, a 21-band Bark scale CBT processor is designed to evaluate the efficiency of the proposed CBT architecture. For comparison, we use the 256-point recursive FFT processor presented in Chapter 3 as a benchmark against the 21-band recursive CBT processor.

4.3.4 Computational Complexity

Since most of the operations in DSP algorithms involve multiplications and accumulation, the multiplication and addition operations are commonly used to measure the computational load of DSP algorithms. As describe in Chapter 2, these two arithmetic operations is the major source of the power consumed in DSP systems. Therefore, in this section, the numbers of multiplications and additions are also used to evaluate the power-efficiency of the proposed CBT algorithm and architecture.

In the proposed CBT, the number of the complex multiplications is half of the window lengths due to the coefficient symmetry property. The input speech data is

always represented as real samples, while the coefficients are complex values. Thus, the 21-band CBT involves 1,766 real multiplications and 3,466 real additions. In contrast, the number of real multiplications and real additions in the 256-point FFT are 4,096 each. The WPT method using 10-order Daubechies filters, requires 9,216 real multiplications and 3,800 real additions in a frame of 64 samples [73]. The comparison of the arithmetic operation numbers among the above three methods is given in Table 4.4. The number of multiplications in the CBT is 56.9% lesser compared to the FFT, while the saving in the real additions is 15.4%. In addition, the reduction as compared to the WPT method is more significant, as shown in Table 4.4.

Table 4.4 Comparison of computational complexity with other transforms

Auditory spectral analyzer	Real multiplications	Real additions
WPT	9216	3800
256-point FFT	4096	4096
Proposed 21-band CBT	1766	3466
Saving compared with FFT	56.9%	15.4%

In most typical DSP algorithms, frequent memory access is another important contribution to the total power dissipation. Because there is no report of hardware implementation about the WPT method in [73], the memory access of the proposed recursive CBT processor is only compared with the 256-point recursive FFT processor presented in Chapter 3. For the proposed 21-band CBT processor, the on-chip memory consists of a 1,766-word $\times$ 16-bit ROM, a 160-word $\times$ 16-bit input RAM, and a 42-word $\times$ 16-bit output RAM. The 256-point FFT processor requires a 256-word $\times$ 16-bit coefficient ROM and a 512-word $\times$ 16-bit RAM. This comparison indicates that the memory requirement by the CBT processor is significantly lesser than the FFT processor. The comparison on RAM access is given in Table 4.5. The

CBT requires a total of 2,010 read/write RAM accesses. This is in contrast to the 8,960 accesses required by a 256-point in-place FFT. The 21-band CBT results in a reduction of 77.6% in memory accesses as compared to the FFT.

Table 4.5 Comparison of memory access between the FFT processor and CBT processor

Auditory spectral analyzer	RAM access			Total memory accesses
	Input write	R/W during computation	Output read	
256-point in-place FFT processor	256	8192 ($512 \times 2 \times 8$)	512	8960
21-band CBT processor	160	1808 ($1766 + 42$)	42	2010

From the above comparisons, the proposed 21-band CBT processor requires fewer arithmetic operations and lesser memory access as compared to the benchmark 256-point recursive FFT processor, and therefore, is likely to be more cost-effective and energy-efficient.

4.4 Simulation and Implementation Results

The proposed recursive CBT architecture is used to design a 21-band Bark scale critical-band transform processor. In this study, the 256-point FFT processors based on the recursive FFT architecture using the two-multiplier and three pipelined butterfly unit presented in Chapter 3 is used as a benchmark for the 21-band Bark scale CBT processor.

In order to perform a reasonable and convincing comparison, both the 21-band CBT processor and the 256-point FFT processor is constructed by a same set of basic functional blocks implemented in VHDL, such as two's complement multipliers,

adders, subtractors, DFF (D-flip-flops) registers, multiplexers, counters and ROM. The RAM is also described by the same HDL (hardware description language) coding style and implemented by arrays of DFF registers. 16-bit two's complement data format is chosen for the I/O data, and coefficients in these two processors. Both the CBT processor and the FFT processor are designed and simulated at RTL by the Mentor Graphics ModelSim [63]. To estimate the timing and area size, the two processors are synthesized into the gate-level with the AMS 0.35 μm CMOS digital library in the Synopsys Design Compiler [64]. To estimate the power consumption, the two processors are simulated at transistor level in the Synopsys NanoSim [65].

To simulate speech and audio processing, the 21-band Bark scale CBT processor is designed and simulated in the Mentor Graphics ModelSim to perform real-time processing at 50% overlap on 160-point 16-bit two's complement fixed-point data at a sampling rate of 16 kHz. The CBT processor takes 1,167 cycles to compute a 21-band CBT, including I/O processing, while the FFT processor computes a 256-point FFT in 2,572 cycles, including I/O processing.

To estimate the timing and area size, the 21-band Bark scale CBT processor is synthesized into the gate-level with the AMS 0.35 μm CMOS standard cell digital library by the Synopsys Design Compiler. The estimated maximum clock frequency of the 21-band CBT processor is 83.3 MHz, while the maximum clock frequency of the 256-point FFT processor is 100 MHz. Thus, the total time to compute a 160-point CBT by the CBT processor is 15,171 ns, while the total time to compute a 256-point FFT by the FFT processor is 25,720 ns. The estimated areas of the CBT and FFT processors are 2.69 mm^2 and 9.02 mm^2 , respectively. The RAM, CBT data path, ROM, and other control logic occupy 73.5%, 13.3%, 9.5%, and 3.7% of the CBT processor area, respectively.

In order to estimate the power consumption, the 21-band Bark scale CBT processor is simulated at transistor level in the Synopsys NanoSim. Simulation at

transistor level shows that the CBT processor can still work at a maximum clock frequency of 13 MHz, when the supply voltage is scaled down to 1.1 V. The CBT processor is simulate at transistor level to operate at 50% overlap on 160-point data segments at a sampling rate of 16 kHz. This processor can achieve real-time processing at 234 kHz.

Table 4.6 lists the power consumption of different functional units in the CBT processor at a clock frequency of 234 kHz and 1.1 V. The simulation results in Table 4.6 indicate that the power consumed by the arithmetic operations in the data path and the memory operations in the I/O RAM dominate the total power consumption, which is 71.3% and 17.6%, respectively. In particular, multiplications in the data path contribute 52.1% of the total power consumption, which is the largest portion in the total power consumed by the CBT processor. This is because multiplication operation and memory operation are usually the most power-hungry operations in typical DSP algorithms.

Table 4.6 Power distribution of different functional units in the CBT processor

Functional units	Address generator	controller	I/O RAM	ROM	Data path (Multiplications)
Percentage of the total power dissipation	4.6%	2.8%	17.6%	2.9%	71.3% (52.1%)

Table 4.7 shows the power consumed by the 21-band CBT processor operating at 234 kHz, when the supply voltage is 3.3 V and 1.1 V, respectively. It is clear that the CBT processor achieves about 95.3% power saving at 234 kHz by scaling the supply voltage from 3.3 V to 1.1 V. As presented in Chapter 3, the 256-point 16-bit FFT processor performs real-time processing within 7.7 ms at a clock frequency of 322 kHz and 1.1 V. The 256-point FFT processor consumes 87.1 μ W per FFT computation when operating at 322 kHz and 1.1 V, while the 21-band Bark scale CBT

processor consumes only $15.6 \mu\text{W}$ per CBT computation when operating at 234 kHz and 1.1V.

Table 4.7 CBT processor power dissipation simulation results

Supply voltage (V)	3.3	1.1
Clock frequency (MHz)	0.234	0.234
Average power ($\mu\text{W}/\text{MHz}$)	1413.6	66.7

Table 4.8 shows the performance comparisons between the 21-band Bark scale CBT processor with the 256-point 16-bit FFT processor presented in Chapter 3. In order to reduce the hardware requirements, both processors employ a single time-multiplexed data path to do front-end spectral computation for low-power speech applications. The 256-point FFT processor uses pipelined butterfly unit consisting of two 16-bit multiplier, one 32-bit adder, and two 16-bit adders. The 21-band CBT processor uses a pipelined CBT data path consisting of two 16-bit multiplier, two 32-bit adders, and two 16-bit adders. The memory requirement of the 21-band CBT processor is 160-word $\times$ 16-bit RAM, which is lesser than the 256-point FFT processor. The CBT processor computes a transformation in 1,167 cycles including the I/O process, while the 256-point FFT processor completes in about 2,572 cycles. With the AMS 0.35 μm CMOS technology, the CBT processor consumes lesser silicon area as compared to the FFT processor. This is mainly because the FFT processor requires more memory size than the CBT processor. The estimated maximum clock frequencies for the two processors are 83.3 MHz and 100 MHz, respectively. However, the CBT processor computes faster because it takes fewer clock cycles to compute a transformation than the FFT processor. Because the 256-point FFT computation requires significantly more arithmetic operations and memory

operations than a 21-band CBT computation as described earlier, the CBT processor consumes significantly lesser power than the FFT processor.

Table 4.8 Comparison of the FFT and CBT processors

Auditory spectral analyzer	256-point FFT processor	21-band CBT processor
Architecture	Recursive	Recursive
Arithmetic units	2 multipliers and 3 adders/subtractors	2 multipliers and 4 adders/subtractors
Memory size	512-word $\times$ 16-bit	160-word $\times$ 16-bit
Processing cycles	2572	1167
CMOS technology (μm)	0.35	0.35
Area (mm^2)	9.02	2.69
Clock frequency (MHz)	100	83.3
Processing time (μs)	25.72	15.18
Power @ 1.1 V	87.1 μW @ 322 kHz	15.6 μW @ 234 kHz

From the above comparisons, the proposed recursive CBT processor has fewer hardware requirements and consumes lower power than the benchmark 256-point FFT processor. The circuit simulation and implementation results demonstrate that the proposed CBT architecture is a good front-end spectral analysis solution for low-power and low-cost speech applications.

4.5 Conclusions

In this chapter, an efficient algorithm and its corresponding VLSI architecture for the CBT have been presented for low-power speech applications. The objective is to design a CBT processor with the requirements of low cost, low power, and close approximation to the critical-band filtering of the human ear. Firstly, we explored various CBT schemes to approximate the human ear critical-band scale in the

literature review. From the observation of the critical-band scale, we formulated a generalized CBT algorithm, which consists of a constant-bandwidth transform in the lower frequency range and a Brown CQT in the higher frequency range. We showed that by finding a set of suitable parameters, the CBT can achieve a close fitting to a particular critical-band scale. Comparative studies were conducted to show that a proposed 21-band Bark scale CBT is better than the WPT-based and FFT-based CBT methods in terms of the closeness in approximation to the critical-band filtering. Simulation results verified its suitability for performing short-time spectral analysis on speech. By exploiting the recursiveness, parallelism, and regularity properties in the proposed CBT algorithm, we proposed an efficient CBT architecture with a pipelined data path to calculate the CBT recursively. The regular CBT data path is derived by exploiting the symmetry property of the CBT coefficients. The coefficient symmetry property also reduces the computations and coefficient memory area by up to 50%. Therefore, the proposed CBT architecture can obtain significant power efficiency by reducing the arithmetic operations, using parallel and pipeline processing, and applying the supply voltage scaling technique. The comparison with the WPT-based and FFT-based CBT methods also showed that the proposed CBT architecture requires fewer arithmetic operations, memory operations and memory requirements, therefore, is an area- and energy-efficient CBT solution.

Circuit design and simulation of a 21-band Bark scale CBT processor and the comparisons against the benchmark 256-point FFT processor demonstrated the efficiency of the proposed recursive CBT architecture. The ASIC implementation verified the hardware cost efficiency of the proposed CBT architecture because it has much fewer hardware requirements compared to the 256-point FFT processor. The transistor-level power simulation results showed that 21-band CBT processor can achieve a substantial power saving by the supply voltage scaling and pipeline processing techniques. With a 0.35 μm CMOS technology, the power consumed by

the CBT processor is $15.6 \mu\text{W}$ at 1.1 V. The 21-band CBT processor achieves 82.1% power reduction compared with the 256-point FFT processor. The comparisons in timing, area size and power consumption with the 256-point FFT processor showed that the proposed recursive CBT processor is a low-cost and power-efficient solution for speech applications.

In conclusion, an efficient CBT algorithm and corresponding recursive CBT architecture have been proposed to approximate the critical-band filtering of human ear. The theoretical analysis, circuit simulation and implementation, indicate that this proposed CBT algorithm provides a close approximation to the critical-band scale, and the proposed CBT architecture is a good low-cost and low-power implementation for speech applications.

Chapter 5

VLSI Architectures for Discrete Wavelet Packet Transform

5.1 Introduction

In Chapter 3 and 4, we presented our research work on the development of efficient VLSI architectures for the fast Fourier transform (FFT) and critical-band transform (CBT), respectively. In this chapter, we look into the development of different architectures for the discrete wavelet packet transform (DWPT), which are suitable for VLSI implementation. In Chapter 2, we described discrete transforms that are widely used to take advantages of the human perception properties in many signal processing applications. Such examples include the FFT and STFT with constant time and frequency resolutions, and the constant-Q transform and CBT with nonuniform time and frequency resolutions. In contrast to these traditional transforms, the discrete wavelet transform (DWT) has many useful properties as reported in Chapter 2. The DWT is an effective multiresolution analysis tool successfully and has been applied in many fields [44], [45]. DWT has been adopted as the transform coder for both JPEG2000 (Joint Photographic Experts Group 2000) still image coding [82], [83] and MPEG-4 (Moving Picture Experts Group-4) still texture coding [84]. DWT has also been used as the transform coder in the fingerprint image compression by Federal Bureau Investigation (FBI) in the United States [85].

However, the DWT is a very computational-intensive and memory-intensive process, which is not desirable for high-speed or low-power hardware implementation. As described in Chapter 2, the lifting-based DWT outperforms the convolution-based method in many aspects, including computational complexity and boundary extension, etc. Therefore, in the past decade, many VLSI architectures for the lifting-based DWT have been proposed to meet the real time requirement in many applications, as reported in [86]. These DWT architectures were proposed to increase throughput, improve hardware utilization, reduce critical path, or minimize hardware complexity, etc.

As a generalization of DWT, DWPT provides good temporal and spectral resolutions in arbitrary regions of the time and frequency (TF) plane [45], [87]. Unlike the DWT that only processes the low-pass filter's coefficients at each resolution level, the DWPT allows further processing of both low-pass and high-pass filter coefficients at each level. Therefore, the DWPT not only inherits all advantageous properties of the DWT, but also offers a flexible TF decomposition. Due to these inherent characteristics, DWPT has been successfully used in a wide range of applications, including image and video coding, speech and audio coding, speech enhancement, speech recognition, hearing aid and digital communications [45], [87]-[89], [73], [74], [90], [91].

However, there are very few papers about the development of dedicated architectures for the DWPT. This is mainly because the DWPT is more computational-intensive and memory-intensive. As we will describe in the following sections, the DWPT requires significantly more hardware requirements than the DWT in order to provide a flexible TF decomposition. Due to the large number of arithmetic units and storage units, hardware implementation becomes a challenge for one-dimensional (1-D) DWPT, as well as two-dimensional (2-D) DWPT. Driven by the successful usage of DWPT in a wide range of applications and the challenges existing

in dedicated hardware implementation, the study of specified VLSI implementation for the DWPT is very important and inevitable.

In this chapter, we present several dedicated architectures for computing 1-D multilevel lifting-based DWPT. In order to exploit the inherent properties in the DWPT algorithm, we formulate an in-place DWPT algorithm in a regular signal flow graph (SFG) by using the butterfly and group concepts. By exploiting the useful properties including recursiveness, regularity, parallelism, and modularity, we derive three DWPT architectures, providing a flexibility to make design trade-offs between hardware cost and processing speed for different applications.

Since the direct-mapped (DM) tree-structured DWPT architecture involves very high hardware complexity, we propose a recursive DWPT architecture as our first architecture. By exploiting the regularity and recursiveness of the in-place DWPT, the proposed architecture utilizes just one PE interacting with a memory unit to compute J -level DWPT recursively. Compared with the DM architecture, this recursive architecture achieves significant cost reduction by reducing the hardware complexity from $(2^J - 1)$ PEs to one PE. Next, we develop a generalized pipeline DWPT architecture for high-speed applications. By exploiting the inherent parallelism and regularity properties, this efficient DWPT architecture is proposed to implement pipeline high-throughput processing without any on-chip memory to buffer the intermediate coefficients. Unlike the DM architecture that has very high hardware complexity (total of $(2^J - 1)$ PEs) and low efficient hardware utilization (reduced by half at each level), this pipeline architecture reduces the hardware requirements to J PEs and maintains the high throughput and the hardware utilization at 50%, by using folding technique to assign different subband computations into just one folded PE at each level. This pipeline architecture with folded PEs is scalable for J -level DWPT using any L -tap wavelet filters. Compared with the DM architecture, this proposed pipeline architecture has the advantageous features of high throughput, low intensive

memory accesses, high hardware utilization, low hardware complexity, hardware regularity and simplicity, and good scalability. Thirdly, we propose a folded DWPT architecture for area-constrained applications where high speed is not the primary goal. Derived from the proposed pipeline DWPT architecture, this folded architecture further reduces the hardware complexity significantly. By exploiting the structure similarity between the folded PEs at each level in the pipeline architecture, this folded architecture folds multilevel DWPT decomposition into a single configurable PE. Under appropriate configuration and control, this single PE can compute multilevel DWPT recursively. Compared with the recursive DWPT architecture, this folded architecture does not require extra memory to buffer the intermediate data during the computation, and therefore, avoids both the intensive memory operations and memory I/O bandwidth limitation. Meanwhile, the processing time for the folded architecture to compute multilevel DWPT is much shorter than the recursive architecture, and the hardware requirement of the two architectures are almost identical.

Following the presentation of the three proposed DWPT architectures, some case studies including $5/3$ and $9/7$ wavelet filters implementations are presented to illustrate the generalized design methodology of the scalable DWPT architectures for different wavelet filters. Performance evaluation of the proposed three architectures is also presented in terms of computational complexity, memory accesses, hardware complexity, and processing cycles, etc. Comparative studies is conducted to demonstrate the proposed pipeline DWPT architecture outperforms the other reported architectures in terms of memory accesses, throughput, hardware regularity and simplicity. Eventually, the circuit simulation and implementation results are provided and discussed to validate the efficiency of the proposed three DWPT architectures. Comparative analysis and discussion are presented to provide an insight into the design trade-offs between speed and area metrics, and conduct a study of these

different DWPT architectures for various dedicated DSP applications with different constraints.

The organisation of this chapter is as follows. In Section 5.2, we perform an extensive review of the existing DWPT architectures. In Section 5.3, we describe the in-place DWPT algorithm, and present our three DWPT architectures: (i) the proposed recursive DWPT architecture, (ii) the proposed pipeline DWPT architecture, and (iii) the proposed folded DWPT architecture. Case studies of the pipeline architecture implementations of the 5/3 and 9/7 wavelet filters is presented in Section 5.3. Performance evaluation of the proposed three DWPT architectures is also presented in Section 5.3. In Section 5.4, circuits simulation and implementation results are presented and compared to validate the efficiency of the proposed architectures. Section 5.5 concludes this chapter.

5.2 Review of Existing VLSI Architectures for DWPT

As the generalization of DWT, the DWPT involves significantly more hardware resources in order to provide a flexible time and frequency decomposition. As described in Chapter 2, the J -level DWT requires J low-pass $h(z)$ and high-pass $g(z)$ wavelet filter pairs. Only the low-pass filter's coefficients at each resolution level are further processed in the next level. However, in a J -level DWPT [45], there are 2^j ($j = 0, 1, 2, \dots, J - 1$) low-pass and high-pass wavelet filter pairs at level j to decompose the signal into approximated and detailed signals in frequency subbands at the subsequent level. As an example shown in Figure 5.1, in a 3-level DWPT full decomposition, there are 1, 2, and 4 low-pass and high-pass filter pairs at the three corresponding levels, respectively.

Figure 5.1 A 3-level DWPT full decomposition

In the direct-mapped (DM) architecture which is derived by directly mapping the data flow of the J -level DWPT into actual hardware, 2^j identical PEs are used in parallel at level j to process the intermediate coefficients (or input data) in 2^j subbands from the previous level, respectively. Each PE in the DM architecture is a low-pass and high-pass wavelet FIR filter pairs. Obviously, this DM architecture has a high processing speed with simple control. However, this DM implementation results in a very high hardware complexity of $(2^J - 1)$ wavelet filter pairs, which is impractical for high-level DWPT decomposition. Moreover, this DM architecture has low efficient hardware utilization due to the in-nature multirate property of the DWPT algorithm. Because the low-pass and high-pass coefficients are decimated by two at each level, the hardware utilization of the PEs at each level in the DM architecture is reduced by half. For example, under the assumption that only one data element per sample cycle enters the system, the hardware utilizations of PEs in the DM architecture for 3-level DWPT are 50%, 25% and 12.5%, respectively. Due to these two disadvantages, the DM architecture of multilevel DWPT is not suitable for VLSI implementation.

There are several specific DWPT architectures reported in the literature [92]-[97] to overcome these disadvantages of the DM architecture. Similar to the dedicated FFT

architectures reported in Chapter 3, there are mainly four types of architectures to compute multilevel DWPT:

- (a) the single-PE recursive architecture [92]-[94], which uses a single PE interacting with memories to calculate different subband decomposition in the DWPT, recursively, as shown in Figure 5.2 (a);
- (b) the single-PE folded architecture [95], in which one (or several) folded PE(s) with extra buffer are employed to increase the hardware utilization and avoid intensive memory operations during the computation, as shown in Figure 5.2 (b);
- (c) the pipeline architecture [96], which computes multilevel DWPT in a high-throughput pipeline processing manner, as shown in Figure 5.2 (c);
- (d) the parallel architecture [97], which increases PEs working in parallel to improve the processing throughput, as shown in Figure 5.2 (d).

Each approach differs in the level of parallelization to meet different design constraints of dedicated applications, by trading off processing time and hardware complexity. Specifically, the PE employed in each approach may be a multiplier-accumulator (MAC), a low-pass (or high-pass) filter, or a low-pass and high-pass filter pair, by making a trade-off between processing speed and hardware cost.

Figure 5.2 Various DWPT architectures

5.2.1 Recursive DWPT Architecture

Like the recursive FFT architecture, the recursive DWPT architecture also uses a single time-multiplexed PE interacting with memories to compute the DWPT decomposition recursively by exploiting the inherent recursiveness in the DWPT algorithm. Similar to the recursive FFT architecture, there are two memory management strategies for the recursive DWPT architecture, namely, the dual-memory mode and single-memory mode.

Wu *et al* [92] designed a programmable DWPT processor using two-buffer memory system and a single PE (MAC) to calculate different subband decomposition. As the DWPT is a typical filtering operation, all multiplications and accumulations required by the DWPT decomposition are multiplexed on a single pipelined MAC in

this recursive architecture. During the computation, the individual MAC outputs are stored in a buffer. The two N -word buffers work alternatively, where N is the frame size. At any time, one buffer is for data input and output, while the other is for the computation. The computation time for one frame is relatively long for multilevel DWPT decomposition because only one MAC operation every clock cycle is performed by the single time-multiplexed PE. In addition, this architecture requires $2N$ -word memory, which is probably not desirable for a complete VLSI implementation.

Similar to the Wu's architecture, Trenas' recursive architecture [93] also uses a single PE to compute multilevel in-place DWPT recursively. In order to increase the processing speed, the Trenas' architecture employs a PE consisting of two L -tap FIR filters, of which each contains L multipliers in parallel and $L - 1$ adders. During the computation, the PE is time-multiplexed to perform the low-pass and high-pass filter operations at each level. The input sequence and intermediate coefficients are buffered in a single memory of size N . This recursive DWPT architecture exploits the in-place nature of the DWPT algorithm to reduce the memory accesses by half, as compared to the a standard architecture requiring $2 \times N \times J$ memory accesses to compute a J -level WPT [93].

Unlike the above two convolution-based DWPT architectures, Aroutchelvame *et al* [94] proposed a recursive lifting-based DWPT architecture. The Aroutchelvame's recursive architecture takes advantage of the lifting technique to reduce the computational complexity as compared to the convolution-based method. This recursive architecture utilizes a single PE (a lifting implementation of the wavelet filter) to perform multilevel DWPT decomposition. During each level DWPT computation, the Aroutchelvame's architecture does not require any extra memory/FIFOs (first-in first-outs) to buffer the intermediate coefficients, while the above two DWPT architectures need to buffer the intermediate results. However, in

the Aroutchelvame's architecture, the coefficients of each level DWPT still have to be buffered in a dual-port RAM before the next level DWPT computation begins.

In principle, by exploiting the recursiveness property, the recursive DWPT architecture decomposes multilevel DWPT computation into sub-tasks, such as MAC operations, subband computation (low-pass or high-pass FIR filtering), or different level DWPT computations. By mapping multiple sub-tasks into a time-multiplexed single PE, the recursive architecture can reduce the hardware complexity significantly as compared to the DM architecture. However, this low-cost implementation is obtained by trading off processing speed. Moreover, these recursive architectures require extra memory to buffer the intermediate coefficients during the multilevel DWPT computation. The corresponding memory operations not only increase the processing time due to the bottleneck of memory I/O bandwidth, but also may lead to significant power dissipation because of the intensive memory operations. In addition, a dedicated address generator is also needed to provide proper addresses for the I/O data, filtering and intermediate wavelet coefficients.

5.2.2 Folded DWPT Architecture

The folded DWPT architecture is actually a modified recursive DWPT architecture, which employs one (or several) folded PE(s) and extra buffer (consisting of registers and switches) to increase the hardware utilization and avoid intensive memory access during the DWPT computation. As reported in Chapter 2, a folding technique called multirate folding technique [21], [22] can be used to implement a multirate DSP algorithm into a single-rate, low hardware complexity, and high hardware utilization VLSI architecture. Due to the in-nature multirate property of DWT and DWPT algorithms, this multirate folding technique is used in the VLSI implementations of both DWT and DWPT [98], [99], [95].

Paya *et al* [95] presented a RPA-based (recursive pyramid algorithm) folded architecture for computing multilevel lifting-based DWPT without intensive memory access. By using the multirate folding technique, the Paya's folded architecture maps a 3-level DWPT decomposition into two folded PEs. Each folded PE is actually a modified low-pass and high-pass wavelet filter pairs. Compared with the DM architecture, this folded architecture reduces the number of low-pass and high-pass filter pairs from 8 to 2 and improves the hardware utilizations of the two folded PEs to 87.5% and 62.5%, respectively. Unlike the above three recursive architectures computing one level DWPT decomposition at a time, the Paya's folded architecture computes multilevel DWPT simultaneously without buffering the intermediate coefficients, and therefore, increases the processing speed and avoids significant power consumed by intensive memory operations.

However, the actual circuit simulation and implementation results are not reported in [95]. The Paya's method has high scheduling and control complexity, which also introduces large numbers of switches, multiplexers and control signals. In addition, the architecture is not regular and needs to be redesigned for different levels of DWPT computation.

5.2.3 Pipeline DWPT Architecture

Trenas *et al* [96] also proposed a pipeline DWPT architecture to speed up the processing throughput. This pipeline DWPT consists of a series of J PEs communicated by J buffers. Each PE computes one level of the total J -level decomposition, while the J buffers store the input data or intermediate wavelet coefficients at each level.

Trenas presented two DWPT processors based on the proposed pipeline architecture to compute convolution-based DWPT: one is the word-serial pipeline

implementation (Figure 5.3 (a)), and the other is the word-parallel pipeline implementation (Figure 5.3 (b)). In the word-serial pipeline implementation, the PE at each level is actually a MAC, while the buffer at each level is a memory bank. Each MAC is time-multiplexed to perform both L -tap low-pass and high-pass FIR filtering. The memory banks support simultaneous read and write operations during the computation. The total memory requirements are $(2^J - 1)L$, which consume significant silicon area, especially with long-tap, high-level DWPT decomposition. In order to further improve the processing speed, each PE in the word-parallel implementation is the direct implementation of an L -tap parallel FIR filter, which consists of L multipliers and $L - 1$ adders to perform low-pass and high-pass filtering. In addition, the buffer at each level is implemented by FIFOs. However, the storage requirements are $(2^J - 1)(L - 1)$.

(a) word-serial implementation

(b) word-parallel implementation

Figure 5.3 Trenas's pipelined DWPT architectures [96]

The disadvantage of these two pipeline DWPT implementations is that both require extra memory (or FIFOs) to buffer the intermediate coefficients during the multilevel DWPT computation. In other words, the Trenas's pipeline DWPT architecture involves intensive memory/FIFOs operations during the computation. Therefore, this pipeline DWPT architecture suffers from the bottleneck of limited

memory I/O bandwidth, resulting in significant silicon area and power consumption due to the on-chip memory requirements.

5.2.4 Parallel DWPT Architecture

A parallel architecture for computing lifting-based DWPT was presented by Arguello [97]. This parallel architecture consists of a group of PEs (MACs) operating in parallel on the data pre-stored in a memory bank, as shown in Figure 5.4. The PEs are folded to compute intermediate wavelet coefficients at different levels. By a regularization of different lifting steps in DWPT, all lifting steps are changed into similar structures with similar complexity. This regularization process allows the parallel DWPT architecture to efficiently compute a wide range of lifting steps. Therefore, the parallel architecture can compute DWPTs with different types of wavelet filters over the same hardware (MAC).

Figure 5.4 Arguello's Parallel DWPT Architecture [97]

The disadvantage of the Arguello's parallel architecture is that this regularization increases the number of arithmetic operations (multiplications and additions) required

by the lifting-based DWPT. This regularization also introduces control and routing overhead due to the increase in multiplexers and extra registers. In addition, intermediate wavelet coefficients need to be buffered in the memory during the multilevel DWPT computation.

5.3 The Proposed VLSI Architectures for Lifting-Based DWPT

5.3.1 The In-Place DWPT Algorithm

In practice, both DWT and DWPT are often implemented by a tree-structured filterbank consisting of a group of low-pass and high-pass filter pairs [45]. As described in Chapter 2, the input signal is filtered separately by a low-pass and a high-pass filter in the DWT. As shown in Figure 2.9, the two output sequences of the low-pass and high-pass filters are then downsampled by a factor of two. Subsequently, only the output of the low-pass filter is further processed in a similar fashion at the next level, while the output of the high-pass filter remains as the final results. This process is repeated to produce multilevel DWT decomposition. From a multirate signal processing point of view, the multilevel DWT decomposes an input signal into two-frequency subbands that are critically sampled. The signal in the lower frequency bands is further decomposed into two subbands at the next level. And the similar decomposition repeats at the subsequent levels.

As generalization of the DWT, DWPT allows further processing of both the output of the low-pass filter and output of the high-pass filter in the subsequent levels, which can provide the flexibility to produce arbitrary decomposition of the input

signal on the TF plane. The selection of a particular decomposition tree for a dedicated application depends on the particular temporal and spectral (time and frequency) resolutions required by the specific application [93], [87].

In contrast to the 3-level DWT decomposition as shown in Figure 2.9, a complete 3-level tree for DWPT is depicted in Figure 5.5. In the 3-level full-decomposition DWPT, an input sequence $w_{j,i}(n)$ ($j = 0, 1, 2$, and $i = 0, 1, \dots, 2^j - 1$ in this case) at level j is fed into the low-pass and high-pass filters ($h(z)$ and $g(z)$), respectively. Subsequently, both the low-pass and high-pass coefficients are decimated by half. In general, at level j , signals $w_{j,i}(n)$ in 2^j subbands from level $j - 1$ are further decomposed into the subsequent 2^{j+1} subbands by 2^j low-pass and high-pass filter pairs. At each level, notation $G_{i,k}$ represents the low-pass and high-pass filtering of the k th pair of samples in the i th subband. From a multiresolution point of view, the output from the low-pass filter represents the approximated (coarse) information in the i th signal at a given level j ($w_{j,i}(n)$), while the output from the high-pass filter represents the detailed (fine) information. Thus, at each resolution level, wavelets can decompose the signal into approximated and detailed signals at the next level. The permutation denoted as dashed line in the right column of Figure 5.5, reorders the outputs into frequency order [100].

Figure 5.5 A complete 3-level DWPT implemented by tree-structured filterbank

By exploiting the in-place nature of DWPT as described in [100], [93], we derive a butterfly-based SFG to describe the lifting-based DWPT algorithm, which is equivalent to the tree-structured filterbank implementation. As illustrated in Figure 5.6, an 8-point 3-level full-decomposition DWPT can be described by the proposed SFG using the butterfly and group concepts, which is similar to the 8-point FFT depicted by Figure 2.8 in Chapter 2. In Figure 5.6, wavelet butterflies are arranged in a regular group-based structure. The wavelet butterfly represents the low-pass and high-pass filtering of two input samples, which is actually a series of basic lifting steps in the lifting-based wavelet filter, as described in Chapter 2. The first output and the other output of each butterfly are the low-pass and the high-pass coefficients, respectively. The group denotes different frequency subbands at each resolution level. Thus, the k th butterfly in the i th group at each level is denoted by $G_{i,k}$. The notation $w_{j,i}(n)$ denotes the n th (intermediate) wavelet coefficient in the i th group at the j th level.

In general, there are J decomposition levels (or butterfly stages) for an N -point DWPT, where N is 2^J . In the proposed butterfly-based SFG for a J -level DWPT, there are $N/2$ butterflies localized in groups at level j , in which each group includes $N/2^{j+1}$ ($j = 0, 1, \dots, J-1$) butterflies to process the coefficients in the same frequency subband from the previous level (or the input). At each level, 2^j groups of butterflies decompose the input coefficients in 2^j frequency bands from the previous level into the subsequent 2^{j+1} frequency subbands. Generally, the number of groups (2^j) doubles at each new level, while the number of butterflies in each group ($N/2^{j+1}$) is reduced by half. The correct wavelet coefficients at the final level can be achieved by bit-reverse addressing.

Figure 5.6 Proposed SFG of a 3-level full-decomposition DWPT

When computing J -level DWPT decomposition by the butterfly-based SFG, each wavelet butterfly (actually L -tap low-pass/high-pass filtering) produces two new coefficients from two samples (which can be stored and read from a memory),

together with the previous $L - 2$ older samples. Since $L - 2$ samples are shared by input sequences to every two consecutive butterflies (e.g., $G_{0,0}$ and $G_{0,1}$ at level 1, as shown in Figure 5.6) in the same group, the shared data can be buffered by extra storage provided by the PE to compute the butterflies [93]. Thus, the two new coefficients can be stored back into the same memory locations occupied by the two input data to save memory requirement. This process is referred as the in-place wavelet butterfly computation.

As discussed previously, the tree-structured filterbank (or DM) implementation has the disadvantages of unbalanced hardware requirements at each level (2^j PEs at level j) and uneven computational loads (wavelet filtering) distributed among the PEs (low-pass and high-pass filter pairs). However, by using the butterfly and group to represent the wavelet filtering and frequency subbands, respectively, the proposed SFG derives a regular data structure to describe the lifting-based DWPT algorithm efficiently. As shown in Figure 5.6, the butterfly-based SFG unravels the inherent parallelism in the DWPT algorithm and obtains balanced computational loads distributed over the operators (butterflies) in the data flow, by rearranging the intermediate coefficients at each level. In the proposed SFG, at each level, $N/2$ butterflies are computed in parallel by $N/2$ butterfly operators regularly localized in groups. It is observed that at level j , $N/2$ butterflies are computed with their input indices at a distance of 2^j (1, 2, 4, ..., $N/2$), as they belong to 2^j different frequency bands (groups).

By exploiting the regularity, in-place, recursiveness, and parallelism properties in the DWPT algorithm described by the butterfly-based SFG, we propose three dedicated DWPT architectures, namely, (i) the recursive DWPT architecture, (ii) the pipeline DWPT architecture, and (iii) the folded DWPT architecture, which differ in the level of parallelization by trading off processing time with hardware complexity. Without loss of generality, we consider a 3-level full-decomposition DWPT in this

study in line with other works [93], [95]-[97]. Lifting-based Daubechies-4 (D4) wavelet filter [47] is chosen to be implemented and to verify the efficacy of the proposed DWPT architectures.

5.3.2 The Proposed Recursive DWPT Architecture

Similar to the existing recursive DWPT architecture as reviewed previously, the proposed recursive DWPT architecture also decomposes multilevel DWPT computation into sub-tasks so that it can map the multiple sub-tasks into a time-multiplexed single PE to reduce the hardware requirements. However, in our recursive DWPT architecture, multiple wavelet butterfly operations in groups at different level are mapped into a single lifting-based wavelet filter PE, while the intermediate coefficients are buffered into a memory during the computation.

Figure 5.7 depicts the recursive architecture based on the proposed SFG of the in-place DWPT. By exploiting the regularity and recursiveness as shown in Figure 5.6, this recursive architecture utilizes one PE interacting with a memory to compute J -level DWPT through butterfly-by-butterfly, subband-by-subband, and level-by-level. By exploiting the in-place property of DWPT computation, this recursive DWPT architecture employs a single-memory strategy to reduce the memory requirement. A single dual-port RAM is used to store the input data, buffer the intermediate coefficients, and store the final results. Similar to the proposed recursive architectures for FFT and CBT in Chapter 3 and 4, respectively, a controller is also used in the DWPT recursive architecture to supervise other units to operate properly in three operation modes, that is, the input mode, the computation mode, and the output mode.

Figure 5.7 Block diagram of the proposed recursive DWPT architecture

Additionally, in order to supervise the wavelet filter PE to compute butterflies recursively, the controller is designed to keep track of which butterfly, which subband, and which level, are being computed. A dedicated address generator is required to provide correct memory R/W addresses not only in the I/O modes, but also in the computation mode. The mechanism of memory address generation during the DWPT computation is based on the observation that at level j , $N/2$ butterflies are computed with their input indices at a distance of 2^j , as described in the previous section. At a given level, the location of the first butterfly input in a group equals to the index of that group at that level. As an example, Table 5.1 depicts the address location of the butterfly inputs at each level in a 3-level 8-point DWPT.

Table 5.1 Address generation for the butterfly inputs in a 3-level 8-point DWPT

Butterfly index	Inputs addresses at level 0	Inputs addresses at level 1	Inputs addresses at level 2
0	0, 1 (group 0)	0, 2 (group 0)	0, 4 (group 0)
1	2, 3 (group 0)	4, 6 (group 0)	1, 5 (group 1)
2	4, 5 (group 0)	1, 3 (group 1)	2, 6 (group 2)
3	6, 7 (group 0)	5, 7 (group 1)	3, 7 (group 3)

In this recursive DWPT architecture, the PE is the direct implementation of a lifting-based wavelet filter, of which the task is actually equivalent to the low-pass

and high-pass filter pair with its associated downsamplers, as described in Figure 5.5. However, the lifting-based implementation of the wavelet filter consumes significantly lesser computation and requires fewer hardware resources than the traditional method.

5.3.2.1 Lifting-Based Wavelet Filter PE

As described in Chapter 2, lifting scheme is an efficient way to compute DWT and DWPT. By factoring the polyphase matrix for the wavelet filter into a finite product of upper and lower triangular matrices and a diagonal normalization matrix, the computational complexity required by the DWT and DWPT can be reduced by up to 50% [47].

The low-pass and high-pass filter pair corresponding to the D4 wavelet is described by the transfer functions as:

$$\begin{aligned} h(z) &= h_0 + h_1 z^{-1} + h_2 z^{-2} + h_3 z^{-3} \\ g(z) &= -h_3 z^2 + h_2 z^1 - h_1 + h_0 z^{-1} \end{aligned} \quad (5-1)$$

where $h_0 = \frac{1+\sqrt{3}}{4\sqrt{2}}$, $h_1 = \frac{3+\sqrt{3}}{4\sqrt{2}}$, $h_2 = \frac{3-\sqrt{3}}{4\sqrt{2}}$, and $h_3 = \frac{1-\sqrt{3}}{4\sqrt{2}}$. The factorization of the polyphase matrix for the D4 wavelet filter is given by [47]:

$$P(z) = \begin{bmatrix} 1 & -\sqrt{3} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \frac{\sqrt{3}}{4} + \frac{\sqrt{3}-2}{4} z^{-1} & 1 \end{bmatrix} \begin{bmatrix} 1 & z \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{\sqrt{3}+1}{\sqrt{2}} & 0 \\ 0 & (\frac{\sqrt{3}+1}{\sqrt{2}})^{-1} \end{bmatrix} \quad (5-2)$$

Based on the factorization, one of the implementations for the D4 wavelet filter in [47] is expressed as follows:

$$\begin{aligned}
s^1(l) &= x(2l) + \alpha x(2l+1), \\
d^1(l) &= x(2l+1) + \beta s^1(l) + \gamma s^1(l-1), \\
s^2(l) &= s^1(l) - d^1(l+1), \\
s(l) &= \zeta s^2(l), \\
d(l) &= 1/\zeta d^1(l),
\end{aligned} \tag{5-3}$$

where $\alpha = \sqrt{3}$, $\beta = -\sqrt{3}/4$, $\gamma = -(\sqrt{3}-2)/4$, and $\zeta = (\sqrt{3}-1)/\sqrt{2}$. Note that we start with sequences $x(2l)$ and $x(2l+1)$ to represent the even and odd indexed input samples to the butterfly operator, respectively. The k th intermediate values during the lifting are denoted as $s^k(l)$ and $d^k(l)$ ($k = 1, 2, \dots$), where l is the time index. The sequences $s(l)$ and $d(l)$ represent low-pass and high-pass filter coefficients, respectively.

From (5-3), we derive the direct implementation of the D4 wavelet filter with lifting steps, as shown in Figure 5.8. This lifting-based D4 wavelet filter PE consists of five multipliers, four adders, two delay registers, and six pipeline registers. Note that “D” and “R” in Figure 5.8 represent delay registers and internal pipeline registers, respectively. By dividing the data path into a few pipeline stages, the pipelined PE computes a wavelet butterfly operation in one cycle, with a latency of four cycles. The critical path is reduced from $(3T_m + 4T_a)$ to $(T_m + T_a)$, where T_m and T_a are the time required for performing one multiplication and one addition, respectively. Because a dual-port RAM is used to buffer the butterfly results, the processing cycles required to compute J -level N -point D4 DWPT by this PE in the recursive architecture is NJ cycles plus some overhead cycles, which are the turnaround cycles between different subband computations, and between the read operations and write operations due to the limited I/O bandwidth of the dual-port RAM. Thus, the processing time for computing one 3-level 8-point D4 DWPT is over $24(T_m + T_a)$, excluding the turnaround cycles and I/O process time. The VHDL design of the direct implementation of the lifting-based D4 wavelet filter is provided in Appendix C.

Figure 5.8 Direct implementation of lifting-based D4 wavelet filter

5.3.2.2 Performance Evaluation

As described previously, both Wu's and Trenas' recursive DWPT architectures [92], [93] are proposed to compute convolution-based DWPT. Thus, our recursive DWPT architecture requires significantly lesser computation than the above two architectures because our scheme is a lifting-based DWPT architecture. For example, as indicated in (5-1) and (5-3), the traditional convolution-based D4 filter requires eight multiplications and six additions, while the lifting-based D4 wavelet filter requires only five multiplications and four additions. Table 5.2 shows the comparisons of computational complexity in terms of arithmetic operations required to implement different wavelet filters by the convolution-based and lifting-based schemes [47]. Compared with the convolution-based scheme, the lifting-based architecture reduces the arithmetic operations, including multiplications and additions/subtractions by up to 50%.

Because the Wu's recursive architecture [92] utilizes a single MAC to compute J -level D4 DWPT recursively, the processing time required to compute a frame is very long. In order to improve the processing speed, the Trenas' recursive architecture [93] employs a pair of low-pass and high-pass FIR filters implemented by the parallel filter

architecture. However, the Trenas' architecture is designed to compute 3-level D6 (Daubechies-6) DWPT.

Table 5.2 Computational complexity of lifting-based DWPT versus convolution-based DWPT

Wavelet	Convolution-based method	Lifting-based method	Speedup
Daubechies-4 filter	14	9	56%
Daubechies-6 filter	22	14	57%
5/3 filter	10	6	67%
9/7 filter	23	14	64%

In order to provide a convincing evaluation of our lifting-based architecture, we implement the convolution-based D4 wavelet by the parallel filter architecture and compare it with our proposed architecture. From (5-1), we derive a parallel architecture implementation of the D4 wavelet filter as shown in Figure 5.9, which obtains a throughput of two outputs per cycles with a latency of four cycles. The critical path is $(T_m + 3T_a)$, which can be reduced to $(T_m + T_a)$ by applying internal pipelining. With the internal pipelining, this parallel D4 wavelet filter computes J -level in NJ cycles, excluding the turnaround cycles due to different subband computations and limited I/O bandwidth of the dual-port RAM used to buffer the intermediate data. However, the hardware cost is eight multipliers, six adders, and eight delay registers, excluding pipeline registers. Therefore, our lifting-based D4 wavelet PE requires significantly fewer hardware resources in terms of multipliers, adders, registers, as compared to the convolution-based implementation of the D4 wavelet filter.

Figure 5.9 Parallel architecture of convolution-based D4 wavelet filter

Aroutchelvame's recursive architecture [94] utilizes a single PE (also a lifting implementation of the wavelet filter) to perform multilevel DWPT decomposition. During each level of DWPT computation, the Aroutchelvame's architecture does not require any extra memory/FIFO to buffer the intermediate coefficients. As a penalty, this architecture introduces many registers, multiplexers and extra complicated routing, in order to buffer and feedback the intermediate data into the correct arithmetic modules for further processing [94]. However, only an architecture of 9/7 DWPT is presented by Aroutchelvame. Actual hardware cost and timing performance are not provided in [94].

The performance comparisons between our proposed recursive D4 DWPT architecture with existing recursive D4 DWPT architectures are summarized in Table 5.3. Because there is no report on how to implement D4 DWPT using the Aroutchelvame's architecture in [94], the Aroutchelvame's recursive DWPT architecture is not included in this table. From the comparisons as shown in Table 5.3, the proposed lifting-based recursive DWPT architecture requires significantly fewer hardware resources and arithmetic operations than the Trenas' convolution-based architectures, and therefore, consumes lesser silicon area and power consumption. For

low-cost applications where high throughput is not the primary design goal, this conventional recursive lifting-based architecture is well suited to compute multilevel DWPT decomposition.

Table 5.3 Comparisons of several recursive architectures for J -level D4 DWPT

Architecture	Wu [92]	Trenas [93]	Proposed
Algorithm	convolution	convolution	lifting
Arithmetic operation	14	14	9
Processing element	a MAC	two parallel FIR filters	a lifting-based wavelet filter
Arithmetic unit	1 mults 1 adds	8 mults 6 adds	5 mults 4 adds
Register (pipeline registers)	3 (3)	14 (6)	8 (6)
Memory size	$2N$	N	N
Critical path	T_m	$T_m + T_a$	$T_m + T_a$
Processing cycles (excluding overhead cycles)	$\gg NJ$	NJ	NJ

5.3.3 The Proposed Pipeline DWPT Architecture

Because the recursive DWPT architecture uses a single PE to compute multilevel DWPT decomposition, the recursive architecture suffers from reduced speed performance. In this section, we propose an efficient pipeline architecture for high-speed DWPT computation, by exploiting the inherent parallelism and regularity in the in-place DWPT.

As illustrated in Figure 5.6 in Section 5.3.1, the proposed SFG uses the butterfly and group concepts to unravel the inherent parallelism in the lifting-based DWPT algorithm and distributes computational loads evenly over the wavelet butterfly operators at different level. In general, an N -point multilevel DWPT is calculated by

$\log_2 N$ levels. At each level, $N/2$ butterflies are computed in parallel by $N/2$ butterfly operators regularly localized in groups.

Similar with the pipeline FFT architectures as reviewed in Chapter 3, the proposed pipeline DWPT architecture obtains a unidirectional data processing by breaking the inherent recursiveness in the DWPT algorithm. By mapping the multiple wavelet butterflies localized in different groups into a single PE at each level, the proposed pipeline architecture utilizes a chain of folded PEs cascaded with a series of double-path delay commutators (DDCs) to compute multilevel DWPT in a fashion of rapid pipeline processing. The folded PE suitable for different lifting-based wavelet filters is proposed to compute the wavelet butterflies in different groups simultaneously at each level. As showed in Figure 5.10, the proposed pipeline DWPT architecture called DDC architecture, illustrates how J -folded PEs are interleaved by a series of switches (SW) and delay registers, as well as the flow of data through the pipeline structure.

Figure 5.10 Block diagram of the DDC architecture

5.3.3.1 The Proposed Pipeline DDC Architecture

For the real-time high-speed VLSI implementation of multilevel DWPT, we derive a pipeline architecture called DDC architecture as shown in Figure 5.10. The DDC architecture has J ($J = \log_2 N$) pipeline stages, of which each is corresponding to one of the J levels in an N -point DWPT. This pipeline architecture consists of J PEs (one for each level) and a series of double-path delay commutators (DDCs), which are actually switches and delay registers. The PE is the basic processing engine to compute the wavelet butterflies in different groups at each level. Each PE is the lifting

implementation of a high-pass/low-pass wavelet filter pair. The DDC reorders the data coming from the PE in one stage so as to present them in the right order for the PE in the next stage.

As described in Figure 5.10, the DDC architecture splits the input sequence into two parallel data streams, flowing forward with correct “distance” between the data elements entering the PEs at each stage. The key idea of the DDC pipeline architecture is derived from the observation of the proposed SFG for multilevel DWPT, as shown in Figure 5.6. This feature consists of the input samples or the intermediate coefficients rearrange in a regular manner at each level. In general, in a J -level N -point DWPT, at level j , $N/2$ butterflies are computed with their input indices at a distance of 2^j . Therefore, by employing dedicated buffers (DDCs) to reorder the data appropriately between any two consecutive stages, the DDC pipeline architecture can utilize just one butterfly operator at each stage to compute multiple wavelet butterflies correctly by processing the data in the right order presented from the previous stage. This property of the architecture leads itself to rapid pipelining by appropriate data switching and delay at each pipeline stage.

The operation of the architecture in Figure 5.10 is described as follows. At the first stage, the even-indexed samples are switched into one unit-delay element, while the odd-indexed samples are fed into the other input line of the system. With the PE computation time equals to the data transfer rate interval, the first level DWPT can be performed in the subsequent four-sample intervals following the switch. The second level DWPT is computed similarly to that of the first one except that the distance of the butterfly input indices is 2. Further processing repeats this pattern with the distance of the input data increases by a factor of two at each new level. Thus, the number of unit-delay elements is doubled at each corresponding stage. The number of unit-delay elements required is $(1 + 2 + \dots + 2^{J-1}) = N - 1$. The switching frequency of the switch at a given stage is always half of that at the previous stage. By the means

of appropriate switching and delay between different stages, this architecture can process the butterflies continuously in the exact way specified by the butterfly-based SFG of a J -level DWPT. As an example, the DDC architecture for a 3-level DWPT and the pipeline processing with appropriate switches and delays are illustrated in Figure 5.11. The indices in Figure 5.11 correspond with the indices in Figure 5.6.

Figure 5.11 DDC architecture for 3-level DWPT and its data-flow graph

As described in Figure 5.11, the input signal samples $x(n)$ arrive at node A in the 3-level DDC DWPT architecture sequentially, $x(0)$, $x(1)$, etc. The even-indexed sample is switched into the unit-delay register, while the next odd-indexed sample is switched into the other input of the system. This is because the index difference between butterfly inputs at the first level is one. In this way, the reordered input sequences $x_1(n)\{n = 0, 2, 4, 6\}$ and $x_1(n)\{n = 1, 3, 5, 7\}$ arrive at the inputs of the first level PE at node B and C, respectively. Subsequently, the PE at the first stage performs the four wavelet butterflies following the first level DWPT decomposition, as described in Figure 5.6. As the index difference of the butterfly inputs at the second level is two, we have to reorder the first level results appeared in parallel pairs by the switches and delay registers at the second stage. This reordering process is illustrated

by the waveform of switch SW2 and the sequences of data elements represented by the indices at the corresponding nodes D, E, F, G, H, and I are shown in Figure 5.11. After the rearrangement of the first level results, the second level PE process these reordered two data sequences denoted by $x_2(n)\{n = 0, 1, 4, 5\}$ and $x_2(n)\{n = 2, 3, 6, 7\}$ at node H and I, respectively. Similar processing pattern repeats at the subsequent stages. Thus, by means of appropriate switching and delay between different stages, this 3-level DDC DWPT architecture can process the butterflies continuously in the exact way specified by the SFG of a 3-level DWPT in Figure 5.6. At the last stage, the 3-level DDC architecture computes the final 8 coefficients in 4 cycles.

5.3.3.2 The Proposed Folded PE for Wavelet Filters

As the wavelet butterfly operation is actually L -tap low-pass/high-pass filtering, each wavelet butterfly operation produces two new coefficients from two input samples, together with the previous $L - 2$ old samples. As described in Figure 5.6, $L - 2$ samples are shared by the input sequences to every two consecutive butterflies in the same group at each level. Thus, in a DWPT architecture, the PE at each level should compute the butterflies from a same group consecutively. However, in the proposed DDC architecture with normal PEs, the input coefficients from a same frequency band do not consecutively enter the PEs only except at the PE at the first level. In other words, the DDC architecture cannot compute the butterflies from a same group consecutively at each level. For example, as shown in Figure 5.6, two consecutive butterflies denoted by $G_{0,0}$ and $G_{0,1}$ in a group at the second level are separated by the butterfly denoted by $G_{1,0}$ from another group. Similarly, as shown in Figure 5.11, the corresponding two consecutive data pairs denoted by $[x_2(0), x_2(2)]$, and $[x_2(4), x_2(6)]$ from a subband, are separated by the data pair $[x_2(1), x_2(3)]$ belonging to another subband, when the reordered data sequences enter the PE at the second level.

To overcome the above problem, a folded architecture (or PE) for the lifting-based D4 wavelet filter is proposed with the DDC DWPT architecture. At each level in the DDC architecture, multiple groups of butterfly operations are mapped on one folded PE of the D4 filter, as depicted in Figure 5.12. At level j , there are $n = 2^j$ unit-delay registers at the corresponding unit-delay register locations. As stated in (5-3), in each butterfly operation, the current outputs ($s(l)$ and $d(l)$) are not only calculated from two current inputs, but also rely on the $s1(l - 1)$ in the previous butterfly and the $d1(l + 1)$ in the next butterfly. Therefore, two unit-delay registers are used in the PE at level 0, which is the direct implementation of the lifting-based D4 wavelet filter, as shown in Figure 5.8. The sequences of the butterfly computation at level 0 is butterfly operation $G_{0,0}$, $G_{0,1}$, $G_{0,2}$, and $G_{0,3}$ as specified in Figure 5.6. The corresponding data flow for the D4 PE at the first level is illustrated in Table 5.4. The intermediate signals $s1(l)$, $s2(l)$ and $d1(l)$ defined in (5-3) are represented in Table 5.4 by $s1_{j,i}(l)$, $s2_{j,i}(l)$ and $d1_{j,i}(l)$, respectively. Here, j and i denote the decomposition level and the group, respectively, while D1 and D2 represent the signals at the outputs of the unit-delay registers, as specified in Figure 5.8. In this example, we ignore the pipeline registers in the direct implementation of the lifting-based D4 filter, as shown in Figure 5.8.

Table 5.4 Data flow for the D4 PE at the 1st level in Figure 5.11

Cycle	Group	Input A B	C	D1	E F	D2	Output G H
1	0	$w_{0,0}(0) = x1(0)$ $w_{0,0}(1) = x1(1)$	$s1_{0,0}(0)$		$d1_{0,0}(0)$		
2	0	$w_{0,0}(2) = x1(2)$ $w_{0,0}(3) = x1(3)$	$s1_{0,0}(1)$	$s1_{0,0}(0)$	$s2_{0,0}(0)$ $d1_{0,0}(1)$	$d1_{0,0}(0)$	$s_{0,0}(0) \dots w_{1,0}(0)$ $d_{0,0}(0) \dots w_{1,1}(0)$
3	0	$w_{0,0}(4) = x1(4)$ $w_{0,0}(5) = x1(5)$	$s1_{0,0}(2)$	$s1_{0,0}(1)$	$s2_{0,0}(1)$ $d1_{0,0}(2)$	$d1_{0,0}(1)$	$s_{0,0}(1) \dots w_{1,0}(1)$ $d_{0,0}(1) \dots w_{1,1}(1)$
4	0	$w_{0,0}(6) = x1(6)$ $w_{0,0}(7) = x1(7)$	$s1_{0,0}(3)$	$s1_{0,0}(2)$	$s2_{0,0}(2)$ $d1_{0,0}(3)$	$d1_{0,0}(2)$	$s_{0,0}(2) \dots w_{1,0}(2)$ $d_{0,0}(2) \dots w_{1,1}(2)$
5				$s1_{0,0}(3)$	$s2_{0,0}(3)$	$d1_{0,0}(3)$	$s_{0,0}(3) \dots w_{1,0}(3)$ $d_{0,0}(3) \dots w_{1,1}(3)$

However, at level j , any two consecutive butterflies in the same group, are separated by $2^j - 1$ ($0, 1, 3, \dots, N/2 - 1$) butterflies from the other groups, as shown in Figure 5.6. In order to interleave the butterfly operations from different groups at level j , $n = 2^j$ unit-delay registers can be placed at each unit-delay element position in the folded PE, as shown in Figure 5.12. In this way, signal $s1(l)$ is delayed by 2^j cycles for the next butterfly computation belonging to the same group at level j . For synchronization with $s2(l)$, $d1(l)$ is also delayed by 2^j cycles before the new $d1(l)$ sample to calculate the $s2(l)$. In the meantime, the folded PE can still process the next pair of inputs to calculate the butterfly from the next group without pausing.

Figure 5.12 Proposed folded architecture of the lifting-based D4 filter with internal pipelining

For example, in the 3-level DDC architecture for D4 DWPT, 2, 4 and 8 unit-delay registers are used in the folded PEs at level 0, 1 and 2, respectively. As an example, the data flow shown in Table 5.5 illustrates how two groups of butterfly calculations are interleaved on a single folded PE at the second level. Here, D1, D2, D3, and D4 represent the signals at the outputs of the unit-delay registers, as specified in Figure 5.12 when $n = 2$ for the folded PE at the second level. We also ignore the pipeline registers in the folded PE of the D4 wavelet filter, as shown in Figure 5.12 in this example. The sequences of the butterfly computation at level 1 consists of butterfly operations $G_{0,0}$, $G_{1,0}$, $G_{0,1}$, and $G_{1,1}$ as specified in Figure 5.6. In the data flow, when a butterfly $G_{0,0}$ (or $G_{0,1}$) is being computed, signal $s1(l)$ is delayed in registers D1 and D2 by $2^j = 2$ cycles for the subsequent butterfly $G_{1,0}$ (or $G_{1,1}$) in the same group 0 (or 1)

at level $j = 1$. Meanwhile, in order to calculate $s2(l)$, $d1(l)$ is also delayed in registers D3 and D4 by $2^j = 2$ cycles for waiting new $d1(l)$ sample in the next butterfly operation of the same group. Each butterfly outputs at level j are delayed by $2^j = 2$ cycles. In this way, from cycle #1 to #3, three butterflies $G_{0,0}$, $G_{1,0}$, and $G_{0,1}$ from two different groups (group 0 and 1) are being computed simultaneously by a single folded PE. After the butterfly operation $G_{0,0}$ completes at cycle #3, the subsequent three butterfly computations of $G_{1,0}$, $G_{0,1}$, and $G_{1,1}$ from the two groups are interleaved on the same folded PE in the following cycles. Because the folded PE is kept busy all the time to compute different groups of butterflies simultaneously, the DDC architecture can obtain the high processing and maintains the hardware utilization of the PE at each level. Note that the VHDL design of the folded PE for the lifting-based D4 wavelet filter at the 2nd level in the DDC architecture is provided in Appendix C.

Table 5.5 Data flow for the folded D4 PE at the 2nd level in Figure 5.11

Cycle	Group	Input A B	C	D1, D2	E F	D3, D4	Output G H
1	0	$w_{1,0}(0) = x2(0)$ $w_{1,0}(1) = x2(2)$	$s1_{1,0}(0)$		$d1_{1,0}(0)$		
2	1	$w_{1,1}(0) = x2(1)$ $w_{1,1}(1) = x2(3)$	$s1_{1,1}(0)$	$s1_{1,0}(0)$	$d1_{1,1}(0)$	$d1_{1,0}(0)$	
3	0	$w_{1,0}(2) = x2(4)$ $w_{1,0}(3) = x2(6)$	$s1_{1,0}(1)$	$s1_{1,1}(0), s1_{1,0}(0)$	$s2_{1,0}(0)$ $d1_{1,0}(1)$	$d1_{1,1}(0), d1_{1,0}(0)$	$s1_{1,0}(0) \dots w_{2,0}(0)$ $d1_{1,0}(0) \dots w_{2,1}(0)$
4	1	$w_{1,1}(2) = x2(5)$ $w_{1,1}(3) = x2(7)$	$s1_{1,1}(1)$	$s1_{1,0}(1), s1_{1,1}(0)$	$s2_{1,1}(0)$ $d1_{1,1}(1)$	$d1_{1,0}(1), d1_{1,1}(0)$	$s1_{1,1}(0) \dots w_{2,2}(0)$ $d1_{1,1}(0) \dots w_{2,3}(0)$
5				$s1_{1,1}(1), s1_{1,0}(1)$	$s2_{1,0}(1)$	$d1_{1,1}(1), d1_{1,0}(1)$	$s1_{1,0}(1) \dots w_{2,0}(1)$ $d1_{1,0}(1) \dots w_{2,1}(1)$
6				$s1_{1,1}(1)$	$s2_{1,1}(1)$	$d1_{1,1}(1)$	$s1_{1,1}(1) \dots w_{2,2}(1)$ $d1_{1,1}(1) \dots w_{2,3}(1)$

The folded architecture for the lifting-based D4 wavelet filter at the third level in the DDC architecture is depicted in Figure 5.13. The two unit-delay locations are

assigned by two 4 delay registers, respectively. D1, D2, D3, D4 represent the signals at the each output of the first 4 delay registers, while D5, D6, D7, D8 represent each output of the other 4 delay registers. In Table 5.6, the data flow is shown to illustrate how different groups of butterfly calculations are interleaved on the folded PE without internal pipeline registers. The VHDL structure design of a 3-level D4 DWPT processor core using the proposed DDC architecture and folded PEs is also provided in Appendix C.

Figure 5.13 Folded PE of the lifting-based D4 filter at 3rd level with internal pipelining

In general, at level j , only one D4 filter is folded to allow computation of the 2^j groups of butterflies by placing $n = 2^j$ unit-delay registers into the corresponding unit-delay element locations in the direct implementation of lifting-based D4 wavelet filter. In this way, the DDC architecture can utilize the folded PEs to compute different butterflies exactly specified by the butterfly-based SFG of J -level N -point DWPT decomposition. Therefore, the DDC architecture maintains not only the real-time high-throughput processing with a processing time of $N/2$ cycles for an N -point DWPT, but also obtains 50% hardware utilization with just one clock domain at each level. Compared with the DM architecture of the tree-structured DWPT, the DDC pipeline architecture also reduces the hardware complexity significantly from $(2^J - 1)$ PEs to only J folded PEs with a few extra delay registers.

Table 5.6 Data flow for the folded D4 PE at the 3rd level in Figure 5.11

Cycle	Group	Input A B	C	D1–D4	E F	D5–D8	Output G H
1	0	$w_{2,0}(0) = x_3(0)$ $w_{2,0}(1) = x_3(4)$	$s_{12,0}(0)$		$d_{12,0}(0)$		
2	1	$w_{2,2}(0) = x_3(1)$ $w_{2,2}(1) = x_3(5)$	$s_{12,2}(0)$	$s_{12,0}(0)$	$d_{12,2}(0)$	$d_{12,0}(0)$	
3	2	$w_{2,1}(0) = x_3(2)$ $w_{2,1}(1) = x_3(6)$	$s_{12,1}(0)$	$s_{12,2}(0), s_{12,0}(0)$	$d_{12,1}(0)$	$d_{12,2}(0), d_{12,0}(0)$	
4	3	$w_{2,3}(0) = x_3(3)$ $w_{2,3}(1) = x_3(7)$	$s_{12,3}(0)$	$s_{12,1}(0), s_{12,2}(0), s_{12,0}(0)$	$d_{12,3}(0)$	$d_{12,1}(0), d_{12,2}(0), d_{12,0}(0)$	
5				$s_{12,3}(0), s_{12,1}(0), s_{12,2}(0), s_{12,0}(0)$	$s_{22,0}(0)$	$d_{12,3}(0), d_{12,1}(0), d_{12,2}(0), d_{12,0}(0)$	$s_{2,0}(0) \dots w_{3,0}(0)$ $d_{2,0}(0) \dots w_{3,1}(0)$
6				$s_{12,3}(0), s_{12,1}(0), s_{12,2}(0)$	$s_{22,2}(0)$	$d_{12,3}(0), d_{12,1}(0), d_{12,2}(0)$	$s_{2,2}(0) \dots w_{3,4}(0)$ $d_{2,2}(0) \dots w_{3,5}(0)$
7				$s_{12,3}(0), s_{12,1}(0)$	$s_{22,1}(0)$	$d_{12,3}(0), d_{12,1}(0)$	$s_{2,1}(0) \dots w_{3,2}(0)$ $d_{2,1}(0) \dots w_{3,3}(0)$
8				$s_{12,3}(0)$	$s_{22,3}(0)$	$d_{12,3}(0)$	$s_{2,3}(0) \dots w_{3,6}(0)$ $d_{2,3}(0) \dots w_{3,7}(0)$

5.3.3.3 Case Studies

The proposed DDC architecture with folded wavelet filter PEs can be generalized for J -level, N -point lifting-based DWPT using any L -tap wavelet filters. As described in Figure 5.10, there are generally J ($J = \log_2 N$) PEs interleaved with a series of DDCs consisting of switches and delay registers in the pipeline architecture. The PE at the first level ($j = 0$) is the direct lifting-based implementation of a given wavelet filter. At level j ($j = 1, 2, \dots, J - 1$), the PE is folded by placing 2^j unit-delay registers into the corresponding unit-delay positions in the PE of the direct implementation of the wavelet filter. The number of the unit-delay registers in J folded PEs is therefore $M(1 + 2 + \dots + 2^{J-1}) = M(N - 1)$, where M is the number of unit-delay element in the direct

implementation of lifting-based wavelet filter. As discussed previously, the number of unit-delay registers in the DDCs at J levels is $N - 1$. Thus, the storage requirement by J -level DWPT are $(M + 1)(N - 1)$ unit-delay registers (excluding the internal pipeline registers).

As illustrated in Figure 5.11, by employing the DDCs to reorder the data streams at each stage, the DDC pipeline architecture utilizes the J cascaded folded PEs to compute multiple groups of wavelet butterflies simultaneously to obtain high-speed throughput. The processing time for computing a J -level DWPT decomposition is $N/2$ cycles, where $N = 2^J$. Additionally, this pipeline DWPT architecture has advantages of regular structure and relative simple control, which also makes it the best choice for high-speed applications.

In order to prove the generalized design methodology of the proposed DDC pipeline architecture for different wavelet filters, two case studies are given in the following sections, including the DDC implementations of the 5/3 and 9/7 wavelet filters that have been used in many applications, such as the JPEG2000 standard. Without loss of generality, we consider 3-level DDC architecture implementation of these two wavelet filters. The generalized design method of the DDC pipeline DWPT architecture is also summarized at the end of this section.

A. 5/3 Wavelet Filter

The low-pass and high-pass filter pair corresponding to the 5/3 filter is described by the transfer functions as:

$$\begin{aligned} h(z) &= h_0 z^{-2} + h_1 z^{-1} + h_2 z^0 + h_3 z + h_4 z^2 \\ g(z) &= g_0 z^{-2} + g_1 z^{-1} + g_2 z^0 \end{aligned} \quad (5-4)$$

where $h_0 = -1/8$, $h_1 = 1/4$, $h_2 = 3/4$, $h_3 = 1/4$, $h_4 = -1/8$, $g_0 = -1/2$, $g_1 = 1$, and $g_2 = -1/2$ [86].

The factorization of the polyphase matrix for the 5/3 wavelet filter is given by [86]:

$$P(z) = \begin{bmatrix} 1 & -0.5(1+z^{-1}) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0.25(1+z) & 1 \end{bmatrix}. \quad (5-5)$$

Based on the factorization, the implementation for the 5/3 wavelet filter to compute the i th butterflies at the j th level in the J -level 5/3 DWPT is expressed as the following butterfly equations:

$$\begin{aligned} s0_{j,i}(l) &= x(2l), \\ d0_{j,i}(l) &= x(2l+1), \\ d_{j,i}(l) &= d0_{j,i}(l) + (-0.5)[s0_{j,i}(l) + s0_{j,i}(l+1)], \\ s_{j,i}(l) &= s0_{j,i}(l) + 0.25[d_{j,i}(l) + d_{j,i}(l-1)]. \end{aligned} \quad (5-6)$$

Note that we start with sequences $x(2l)$ and $x(2l+1)$ to represent the even and odd indexed input samples to the butterfly operator, respectively. The k th intermediate values during the lifting process are denoted as $sk_{j,i}(l)$ and $dk_{j,i}(l)$ ($k = 0, 1, 2, \dots$), where l is the time index. The sequences $s_{j,i}(l)$ and $d_{j,i}(l)$ represent the low-pass and high-pass filter coefficients in the i th frequency subband at level j , respectively.

We derive the direct implementation of the lifting-based 5/3 wavelet filter from (5-6) as depicted in Figure 5.14 ($n = 1$ in this case). This lifting-based 5/3 wavelet filter PE consists of two multipliers, four adders, and three delay registers. Note that “D” in Figure 5.14 represents delay registers. By placing $n=2^j$ unit-delay registers into the corresponding three unit-delay element locations, the folded PE for the 5/3 wavelet filter at level j is derived for the multilevel DDC architecture. By dividing the PEs into a few pipeline stages based on the basic lifting steps, the critical path can be reduced to a multiply-accumulation time of $(T_m + T_a)$. Thus, the processing time for computing one 3-level 8-point 5/3 DWPT is $4(T_m + T_a)$. Then, we can build the DDC pipeline architecture with the folded PEs for the 5/3 wavelet filter to compute 3-level 5/3 DWPT decomposition.

Figure 5.14 Folded architecture of lifting-based 5/3 wavelet filter where $n = 2^j$ at level j

Based on the butterfly computation equation as described in (5-6) and the proposed folded PE in Figure 5.14, data-flow tables can be constructed to express how multiple groups of butterflies at each level are computed by the folded PEs. In this case study, only the data flow for the folded 5/3 PE at level 1 is shown as an example in Table 5.7. The folded PE for the 5/3 filter at the 2nd level is depicted by Figure 5.14 with $n = 2$. Notations from D1 to D6 are used to represent the outputs of the three groups of 2 delay registers in the folded PE. From the Table 5.7, we can see that the two groups of butterfly computation are interleaved on the single folded PE in a high throughput of two butterfly outputs in one cycle.

Based on the proposed DDC architecture as shown in Figure 5.10, we can build the pipeline DWPT architecture for computing 3-level 5/3 DWPT, in which three PEs for the 5/3 filter are interleaved by a series of DDCs in a similar manner as illustrated by Figure 5.11. To supervise the DDC architecture to operate properly, a controller can be design based on the data switching and operation scheduling as described by the waveform in Figure 5.11 and the data flow expressed in Table 5.7.

Table 5.7 Data flow for the folded 5/3 PE at the 2nd level in Figure 5.11

Cycle	Group	Input A B	D1, D2 D3, D4	C	D5, D6	Output E F
1	0	$w_{1,0}(0) = s_{0,0}(0)$ $w_{1,0}(1) = d_{0,0}(0)$				
2	1	$w_{1,1}(0) = s_{0,1}(0)$ $w_{1,1}(1) = d_{0,1}(0)$	$s_{0,0}(0)$ $d_{0,0}(0)$			
3	0	$w_{1,0}(2) = s_{0,0}(1)$ $w_{1,0}(3) = d_{0,0}(1)$	$s_{0,1}(0), s_{0,0}(0)$ $d_{0,1}(0), d_{0,0}(0)$	$d_{1,0}(0)$		$s_{1,0}(0) \dots w_{2,0}(0)$ $d_{1,0}(0) \dots w_{2,1}(0)$
4	1	$w_{1,1}(2) = s_{0,1}(1)$ $w_{1,1}(3) = d_{0,1}(1)$	$s_{0,0}(1), s_{0,1}(0)$ $d_{0,0}(1), d_{0,1}(0)$	$d_{1,1}(0)$	$d_{1,0}(0)$	$s_{1,1}(0) \dots w_{2,2}(0)$ $d_{1,1}(0) \dots w_{2,3}(0)$
5			$s_{0,1}(1), s_{0,0}(1)$ $d_{0,1}(1), d_{0,0}(1)$	$d_{1,0}(1)$	$d_{1,1}(0), d_{1,0}(0)$	$s_{1,0}(1) \dots w_{2,0}(1)$ $d_{1,0}(1) \dots w_{2,1}(1)$
6			$s_{0,1}(1)$ $d_{0,1}(1)$	$d_{1,1}(1)$	$d_{1,0}(1), d_{1,1}(0)$	$s_{1,1}(1) \dots w_{2,2}(1)$ $d_{1,1}(1) \dots w_{2,3}(1)$

B. 9/7 Wavelet Filter

As described in [47], the factorization of the polyphase matrix for the 9/7 wavelet filter is given by

$$P(z) = \begin{bmatrix} 1 & \alpha(1+z^{-1}) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \beta(1+z) & 1 \end{bmatrix} \begin{bmatrix} 1 & \gamma(1+z^{-1}) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \delta(1+z) & 1 \end{bmatrix} \begin{bmatrix} \zeta & 0 \\ 0 & \zeta^{-1} \end{bmatrix}, \quad (5-7)$$

where $\alpha = -1.586134342$, $\beta = -0.05298011854$, $\gamma = 0.8829110762$, $\delta = 0.4435068522$, and $\zeta = 1.149604398$. This factorization leads to the implementation for the 9/7 wavelet filter as described by the following butterfly equations:

$$\begin{aligned}
s0_{j,i}(l) &= x(2l), \\
d0_{j,i}(l) &= x(2l+1), \\
d1_{j,i}(l) &= d0_{j,i}(l) + \alpha[s0_{j,i}(l) + s0_{j,i}(l+1)], \\
s1_{j,i}(l) &= s0_{j,i}(l) + \beta[d1_{j,i}(l) + d1_{j,i}(l-1)], \\
d2_{j,i}(l) &= d1_{j,i}(l) + \gamma[s1_{j,i}(l) + s1_{j,i}(l+1)], \\
s2_{j,i}(l) &= s1_{j,i}(l) + \delta[d2_{j,i}(l) + d2_{j,i}(l-1)], \\
s_{j,i}(l) &= \zeta s2_{j,i}(l), \\
d_{j,i}(l) &= 1/\zeta d2_{j,i}(l).
\end{aligned} \tag{5-8}$$

Similar with the 5/3 filter, we start with sequences $x(2l)$ and $x(2l+1)$ to represent the even and odd indexed input samples to the 9/7 filter, respectively. The k th intermediate values during the lifting are also denoted by $sk_{j,i}(l)$ and $dk_{j,i}(l)$ ($k = 0, 1, 2, \dots$), where l is the time index. The sequences $s_{j,i}(l)$ and $d_{j,i}(l)$ represent the low-pass and high-pass filter coefficients in the i th frequency subband at level j , respectively.

Similar to the case study of 5/3 filter, we derive the folded PE of the lifting-based 9/7 filter, as shown in Figure 5.15, in which $n = 2^j$ at level j in the 3-level DDC DWPT architecture. This folded PE consists of six multipliers, eight adders, and $6n$ delay registers. With internal pipelining, the critical path of the PEs can also be reduced to a multiply-accumulation time of $(T_m + T_a)$. Therefore, the processing time required by the DDC architecture to compute a 3-level 9/7 DWPT is $4(T_m + T_a)$. The data flow for the folded 9/7 PEs at different levels is not provided here because the folded 9/7 PEs compute multiple wavelet butterflies in different groups at each level, in the same processing pattern with the folded PEs for the D4 and 5/3 wavelet filters as described in the previous sections.

Figure 5.15 Folded architecture of lifting-based 9/7 wavelet filter where $n = 2^j$ at level j

Finally, the hardware requirement and timing performance of the 3-level DWPT architectures for the D4, 5/3, and 9/7 wavelet filters are summarized in Table 5.8. From this table, we can see that in the generalized DDC architecture, not only the numbers of DDC delay registers and multiplexers, but the timing performance in terms of the processing cycles, are only related to the decomposition level J (or the frame length $N = 2^J$). Only the numbers of hardware resources related to the PEs, such as the multipliers, adders, and folding delay registers, are determined by the wavelet filter to be implemented. Therefore, the DDC architecture is easy to design for different wavelet filters, as we have shown in the previous two case studies.

Table 5.8 Comparisons of the DDC architectures for different wavelet filters

J -level N -point DWPT DDC architecture	Generalized results	D4 filter	5/3 filter	9/7 filter
Processing elements	J	3	3	3
Multipliers	Filter-dependent	15	6	18
Adders	Filter-dependent	12	12	24
Folding delay registers	$M(N-1)$	14	21	42
DDC delay registers	$N-1$	7	7	7
DDC multiplexers	J	3	3	3
Processing cycles	$N/2$	4	4	4

Based on the DDC architecture design of the above three wavelet filters, the general design methodology of the DDC pipeline architecture for different wavelet filters is described by the following steps:

- (1) Decompose a given wavelet filter into lifting steps by factorization of the filter polyphase matrix of the wavelet filter [47];
- (2) Derive the corresponding butterfly equations to describe the computation of the i th wavelet butterfly at the j th level in the butterfly-based DWPT algorithm;
- (3) Construct the direct-mapped (DM) architecture of the lifting-based wavelet filter described by the butterfly equations;
- (4) Derive the folded PEs for the DDC architecture by replacing each unit-delay register in the DM architecture of the filter with $n = 2^j$ delay registers;
- (5) Construct the data-flow tables to illustrate how each of the J folded PEs process the reordered data samples to compute different groups of butterflies simultaneously at each level;
- (6) Draw the data-flow graph with the waveforms of control signals to express how the input and intermediate data samples are reordered appropriately by the DDCs consisting of switches and delay registers at each level in the DDC architecture;
- (7) Build the DDC pipeline DWPT architecture with J folded PEs interleaved by a series of DDCs in a similar manner as illustrated by Figure 5.10 and 5.11;
- (8) Design a controller to generate the control signals deduced from the above mentioned data-flow graph and data-flow tables.

5.3.3.4 Performance Evaluation

In the previous sections, we proposed a generalized DDC pipeline architecture with the folded PE for computing multilevel DWPT with different wavelet filters. In order to evaluate the DDC architecture, we implemented a folded architecture of the wavelet D4 filter suitable for the DDC pipeline architecture. In this section, we provide the performance evaluation of the proposed DDC architecture by the comparisons of the D4 DDC DWPT architecture with other DWPT architectures.

As shown in Figure 5.12, the critical path of the folded PE for the D4 filter is $(3T_m + 4T_a)$, where T_m and T_a are the time required for performing one multiplication and one addition, respectively. In order to reduce the critical path, internal pipelining can be applied by using pipeline registers in the folded PE. By dividing the PE into a few pipeline stages (normally based on the MACs or basic lifting steps), the critical path can be reduced to a multiply-accumulation time of $(T_m + T_a)$. Thus, the processing time for computing one 3-level D4 DWPT is $4(T_m + T_a)$. In general, the processing cycle for computing a J -level DWPT is $N/2$ cycles. Storage required by the D4 DDC architecture are $3(N - 1)$ delay registers excluding internal pipeline registers.

The comparisons of the computation, hardware and timing requirements of the different J -level N -point D4 DWPT architectures are presented in Table 5.9. The computational complexity is compared with respect to the number of arithmetic operations required to perform D4 wavelet filtering, including multiplications and additions/subtractions. The hardware cost is mainly compared with respect to the data path (PEs), which is evaluated in terms of both arithmetic units and storage units. The timing performance is compared with respect to the number of processing cycles to perform a J -level D4 DWPT decomposition.

In terms of computational complexity, our DDC architecture is one of the architectures with the least arithmetic operations (number of multiplication & addition,

including scaling operations) because these DWPT architectures employ the lifting scheme to reduce the computational complexity substantially. However, there is a significant increase in arithmetic operations from the Arguello's lifting-based parallel architecture [97]. This is because the Arguello's architecture uses a regularization of lifting steps to derive the parallel architecture, which increases the computational complexity. In general, compared with the convolution-based DWPT architectures, the lifting-based DWPT architectures can reduce the computational complexity by up to 50%, and therefore, consume lesser power consumption when computing multilevel DWPT decomposition.

Table 5.9 Comparisons of several architectures for 3-level D4 DWPT

Author	Architecture	Algorithm	Arithmetic operation	PE	Arithmetic units	Storage (size)	Processing cycles
Wu [92]	Recursive	Convolution	14	a MAC	1 mults 1 adds	Memory ($2N$)	-
Trenas [93]	Recursive	Convolution	14	two parallel FIR filters	8 mults 6 adds	Memory (N)	$JN/2$
Trenas [96]	Pipeline	Convolution	14	a MAC	3 mults 3 adds	Memory ($4(N-1)$)	$LN=4N$
				two parallel FIR filters	24 mults 18 adds	FIFOs ($3(N-1)$)	
Arguello [97]	Parallel	Lifting	12	MACs	-	Memory (N)	$1.5N$ (3 procs)
Arout. [94]	Recursive	Lifting	9	a lifting-based wavelet filter	-	Memory (N)	-
Paya [95]	Folded	Lifting	9	a lifting-based wavelet filters	-	Registers (-)	-
Proposed	Pipeline	Lifting	9	a lifting-based wavelet filter	15 mults 12 adds	Registers $3(N-1)$	$N/2$

In terms of arithmetic units, the recursive and folded architectures have fewer hardware requirements as they use a single PE to do computation sequentially. The

Wu's single-MAC recursive architecture in [92] has the lowest hardware complexity with a penalty of the slowest processing speed. The Arguello's parallel architecture [97], the Trenas' pipeline architecture using parallel filters [96] and our DDC architecture, are more complex than the other architectures, because they increase the numbers of PEs working in parallel or in a pipeline to improve the system concurrency. As shown in Table 5.9, the three main approaches of recursive, pipeline, and parallel architectures differ in the level of parallelization in terms of the number of PE to meet different design goals by trading off hardware complexity and timing performance. Additionally, the implementation of PEs employed in these architectures may also have a varying degree of hardware complexity. For example, the PE can be a MAC in the serial implementation of the filters in the Trenas's parallel architecture [96], or two FIR filters in parallel implementation in the Trenas's architecture [96].

In terms of storage units, the Trenas's pipeline architecture has the largest memory requirements. Both Trenas' and Aroutchelvame's recursive architectures have a memory capacity of size N . In contrast to other DWPT architectures, the Paya's folded architecture [95] and our DDC architecture use switches and registers to feedback or buffer the intermediate data during the DWPT computation. Compared with the memory-based architectures, these two architectures may be more area-efficient because switches and shift registers generally consume lesser silicon area than memory. In addition, these two architectures are likely to be more energy-efficient not only because of fewer arithmetic computations required, but also by avoiding intensive on-chip memory operations and data fetching on long wires [99]. More importantly, the limited memory I/O bandwidth is always a bottleneck to the timing performance of memory-based architectures, which is discussed in the following paragraphs. Therefore, our DDC architecture is not prone to the above performance issues caused by the on-chip memory requirements.

In terms of timing performance, our proposed DDC architecture has the shortest processing time of $N/2$ cycles for computing N -point final coefficients while the Trenas' pipeline architecture and Arguello's parallel architecture requires $4N$ and $1.5N$ cycles, respectively. This is because the DDC pipeline architecture computes multilevel DWPT with a high throughput by interleaving multiple groups of butterfly computations at each level. The Wu's recursive architecture has to use a high clock frequency to obtain real-time processing within one frame period because it only utilizes a single MAC to do the computation. Although the Trenas' and Aroutchelvame's recursive architectures [93], [94] use a low-pass and high-pass filter pair (or the lifting-based wavelet filter) to implement the PE, both architectures have a longer processing time than the DDC architecture. The reason is not only due to the in-nature sequential architecture, but also because they involve intensive memory operations during the computation.

As pointed out earlier, the DWPT architectures in [92]-[94], [96], [97] all requires extra memory/FIFOs to buffer the intermediate coefficients when they perform the multilevel DWPT computation. The timing performance of these memory-based DWPT architectures may suffer the bottleneck of the memory I/O bandwidth. Particularly, because of the limited I/O ports of a dedicated memory, multiple memory operations may not be supported. In order to avoid the memory operation collision, overhead (turnaround) cycles have to be inserted between read and write operations, or multiple memory operations are reallocated in different time slots. Therefore, the processing cycle for computing a multilevel DWPT by the memory-based architectures is generally longer than those DWPT architectures, such as the Paya's folded architecture and our DDC architecture, which only use switches and registers to buffer the intermediate data.

Finally, our DDC architecture is regular and scalable, which makes it easy to generalize to different level of DWPT decomposition with different wavelet filters, as

described in the previous section. Most of the other DWPT architectures need to be redesigned with more design effort to accommodate different wavelet filters and different level decomposition. For example, the Paya's folded architecture is not regular and has complicated scheduling and control. Therefore, this folded architecture is not easy to be redesigned for different number of level of DWPT computation with different wavelet filters. Specifically, the processing cycle of the DDC architecture ($N/2$) is only related to the decomposition level J (or $N = 2^J$), whatever a wavelet filter with different number of filter tap is chosen. However, most existing DWPT architecture is dependent on the wavelet filter to be implemented. For instance, the processing time of the Trenas's pipeline architecture is NL , which increases with both the frame length N and filter tap number L .

5.3.4 The Proposed Folded DWPT Architecture

In the previous sections, we propose the DDC pipeline architecture for high-speed DWPT computation by exploiting the inherent properties of regularity and parallelism in multilevel DWPT decomposition. However, the hardware complexity of the DDC scheme is still high for some area-constrained applications where high speed is not required. This is because the DDC architecture utilizes J folded PEs, in which each PE is a lifting-based wavelet filter with extra folding registers.

In this section, we propose a single-PE folded DWPT architecture, in which a configurable PE is used to compute J -level DWPT recursively. The folded DWPT architecture is derived from the proposed DDC architecture. In this folded DDC architecture, the hardware complexity is reduced from J PEs to one PE, by folding the multilevel DWPT decomposition into a configurable folded PE.

Similar to other folded architectures, the proposed folded architecture does not require extra memory/FIFOs to buffer the intermediate data when it performs multilevel DWPT computation. In order to avoid intensive memory operations, this folded DDC architecture employs a configurable delay commutator consisting of a series of switches and registers to buffer and feedback the intermediate data to the inputs of the configurable folded PE for further processing. In actual factor, the configurable delay commutator is derived from the DDCs in the proposed DDC architecture. As a result, dedicated address generation required by memory operations are also avoided in this folded DDC architecture.

5.3.4.1 Proposed Configurable Folded PE

The key idea to derive the folded DDC architecture lies in the structure similarity between the folded PEs in the DDC architecture. It is observed from Figure 5.12 that the folded PEs at different levels in the DDC architecture are almost the same with each other, except for different numbers of ($n = 2^j$) unit-delay registers that are placed at the unit-delay positions in these PEs. We also notice that the DDCs between the folded PEs are also very similar, as shown in Figure 5.11. These DDCs exclude the input DDC, and only differ in the length of the delay registers used to reorder the data elements between two consecutive levels.

Therefore, based on the above observations, we can derive a folded architecture from the DDC architecture by folding the multilevel DWPT decomposition into a single configurable PE with a configurable DDC. As an example, the folded DDC architecture for computing 3-level DWPT is illustrated in Figure 5.16. This folded DDC architecture utilizes a single configurable PE, a configurable DDC, some extra I/O switches and feedback registers to compute the multilevel DWPT recursively. The configurable DDC is a modified version of the DDC at level $J - 1$ in the DDC

architecture, in which the correctly delayed data samples are selected and feed to the switch SWJ and the output ports to the DDC by switches SWA and SWB, respectively. In this 3-level case, by using the two 3-to-1 switches SWA and SWB, the data samples from the configurable PE outputs are delayed by one cycle, two cycles, zero cycle, for the 1st, 2nd, and 3rd level DWPT decomposition outputs, respectively. In this manner, the configurable DDC can be configured as the DDC at level j ($j = 0, 1, 2$) in the DDC architecture, as shown in Figure 5.11.

Figure 5.16 Folded DDC architecture for 3-level DWPT

Similarly, the configurable PE is actually configurable version of the folded PE at level $J - 1$ of the DDC architecture. The corresponding configurable PE for the 3-level D4 DWPT is depicted by Figure 5.17. Generally, the n -unit delay registers in the folded PE are configured to output the data samples delayed by correct time cycle of one, two, and four cycles for level 0, level 1, and level 2, respectively. By properly setting the two 3-to-1 switches SWD1 and SWD2, the configurable PE can be configured as the PE at level j in the DDC architecture, as shown in Figure 5.11.

Figure 5.17 Configurable PE in the folded DDC architecture for 3-level D4 DWPT

In general, when computing the j th level DWPT, the PE in the folded DDC architecture is configured as the folded PE at the j th level in the DDC architecture, by appropriately choosing the number of unit-delay registers. The DDC is also configured as the appropriate DDC to reorder the intermediate data correctly. For example, the appropriate configuration of the DDC is set by the switches SWA and SWB in the folded DDC architecture for the 3-level D4 DWPT, as shown in Figure 5.16. The corresponding configuration of the PE is set by the switches SWD1 and SWD2, as shown in Figure 5.17. These switches choose the data from the inputs j ($j = 0, 1, 2$) for the level j . The level- j intermediate coefficients will be sent back into the PE by I/O switches SWO1, SWO2, SWI1, and SWI2, as shown in Figure 5.16. After completed the j th level decomposition, the PE will be configured to compute the subsequent $(j + 1)$ th level DWPT by processing the feedback level- j intermediate data. When the J -level DWPT has completed, the final coefficients are switched to the outputs by switch SWO1 and SWO2, and this folded architecture is ready to compute another new J -level DWPT computation. The new frame of samples to be processed can be stored into a memory during the J -level DWPT computation. By this means, the multilevel DWPT decomposition by the DDC architecture as described in Figure 5.11, can be folded into a single PE with switches, delay registers and feedback registers under correct configurations. Note that the VHDL design of the configurable PE for 3-level D4 DWPT is provided in Appendix C.

As the derivation of the DDC pipeline architecture, this folded DDC architecture is generalized and scalable for different levels DWPT by using correct numbers of unit-delay registers and feedback registers, and J -to-1 multiplexes. The generalized J -level folded DDC architecture is depicted by Figure 5.18 (b). To configure the DDC for the j th DWPT decomposition, the J -to-1 switches SWA and SWB select the data samples from the input port j of the switches, which are delayed by 2^{j+1} cycles ($j = 0, 1, \dots, J - 2$) for reordering, or not delayed at all ($j = J - 1$) for output, as illustrated by

the 3-level D4 DWPT case in Figure 5.16. Similarly, the configurable folded delay registers in the configurable PE is shown in Figure 5.18 (a). In order to appropriately configure the folded PE, the configurable folded delay registers can function as the n ($n = 2^j$) delay registers for the j th level DWPT computation, by using a J -to-1 switch to select the correct delayed data samples, as illustrated by the 3-level D4 DWPT case in Figure 5.17. Note the notations $N/2$ and $N/4$ represent the numbers of delay registers used in the switches.

Figure 5.18 (a) Configurable folded delay registers in the configurable PE

Figure 5.18 (b) Generalized folded DDC architecture for J -level DWPT

5.3.4.2 Performance Evaluation

As described previously, the Paya's folded architecture [95] maps a 3-level 5/3 DWPT decomposition into two folded PEs. Each PE is a modified lifting-based wavelet filter with many extra switches and registers. By using the multirate folding technique, the Paya's folded architecture increases the hardware utilization and avoids intensive memory operations.

Because Paya only presents the folded architecture for the 5/3 filter in [95], a comparison of the timing performances and hardware requirements of the different 5/3 filter implementations, including the proposed three DWPT architectures and the Paya's architecture, are provided in Table 5.10. Note the number of registers for the folded DDC architecture is counted from a folded DDC architecture for 5/3 DWPT, which is omitted in the thesis.

Table 5.10 Comparisons of the folded DDC architecture with several architectures

<i>J</i> -level 5/3 DWPT	Paya's folded [95]	DDC	Recursive	Folded DDC
Processing cycle	-	$N/2$	$>NJ$	$\approx NJ/2$
Processing elements	2 PE	J PEs	1 PE	1 PE
Multipliers	10	6	2	2
Adders	8	12	4	4
Storage (size)	registers (35)	registers (28)	memory (8)	registers (19)
Switches	29	3	0	11

The timing comparison in terms of the processing time among four different architectures for multilevel 5/3 DWPT is given in Table 5.10. The conventional recursive architecture has the slowest processing speed as the timing performance of this single-PE architecture is limited by the I/O bandwidth of a dual-port RAM used in the architecture. Because the DDC architecture uses more (J) PEs in a pipeline to perform J -level DWPT decomposition, the DDC architecture has the shortest processing time among the four architectures. Although the timing report in terms of processing cycle is not reported for the Paya's architecture, the Paya's 5/3 folded architecture is likely to be faster than our folded DDC architecture. This is because different level DWPT computations are interleaved on the two folded PEs in the

Paya's folded architecture. However, our folded DDC architecture only uses one configurable folded PEs to compute the J -level DWPT consecutively.

In terms of the hardware complexity, the DDC architecture has the maximum hardware requirements as this pipeline architecture aims at high-speed DWPT computation. The folded DDC architecture decreases the number of PEs from J to 1 with comparison to the DDC architecture. However, this hardware cost reduction results in the increased processing time from $N/2$ to around $NJ/2$ cycles because the folded DDC architecture utilizes just one PE to compute the J -level DWPT recursively. The folded DDC architecture has an almost identical hardware requirement in the data path with the conventional recursive architecture because the hardware complexity of the configurable folded PE only increases a bit due to the extra registers and switches.

The Paya's folded architecture uses approximately twice as much hardware resources than our folded DDC architecture. The large numbers of the required registers and switches indicate that the Paya's architecture has complex scheduling and control. In addition, as reported in [95], the Paya's architecture is not regular. The Paya's folded architecture needs to be redesigned with a great effort for different number of level of DWPT computation and different wavelet filters. In contrast to the Paya's architecture, our folded DDC architecture inherits the advantages of regularity and scalability from the proposed DDC architecture, as describe earlier.

5.4 Simulation and Implementation Results

In the previous sections, we have proposed three different dedicated DWPT architectures for different application requirements. Performance evaluation of these three architectures has been carried out by comparing them with other existing DWPT

architectures found in the literature in terms of hardware requirement and timing performance at architecture level. In this section, circuit simulation and implementation results are provide to evaluate the three dedicated DWPT architectures.

The direct circuit comparisons between the proposed DWPT architectures with the existing DWPT architectures are possible but tedious. The DWPT processors based on the dedicated architectures reported in the literature differ in the type of the implemented wavelet filters, number of decomposition levels, semiconductor process, wordlengths of both I/O and internal data in the data path, circuit style of the arithmetic and storage units, etc. Moreover, the information of the hardware implementation of different DWPT architectures is often limited or even not available in some literature. Therefore, in order to have a fair comparison with the proposed DWPT architectures, a 3-level D4 DWPT processor based on the direct-mapped (DM) tree architecture is designed as a benchmark by the same group of sub-modules in Verilog-HDL, and implemented into the same CMOS process with the proposed three D4 DWPT processors.

In the following sections, the DM architecture of multilevel D4 DWPT is described, followed by the discussion and comparisons of the four different 3-level D4 DWPT implementations.

A. Benchmark of a DM D4 DWPT Processor

As described previously, the DM architecture for computing multilevel DWPT applies 2^j identical PEs in parallel at level j to process the input or intermediate data in 2^j frequency bands from the previous level, respectively. The total $(N - 1)$ PEs are organized in a tree structure as shown in Figure 5.1, in which each PE is the direct implementation of the given lifting-based wavelet filter.

As shown in Figure 5.19, the DM architecture of the 3-level D4 DWPT applies one PE for the first level decomposition, two PEs at the second level to process the intermediate coefficients in two frequency bands from the first level, and four PEs in the subsequent level to process the data in four bands from the previous level, respectively. Each PE in this architecture is the direct implementation of a lifting-based D4 wavelet filters, as depicted in Figure 5.8. When the DM architecture computes 3-level DWPT decomposition, each PE is assigned to perform one frequency subband computation or a group of butterfly operations independently, as shown in Figure 5.5 and 5.6, respectively. In other words, one PE compute a group of butterflies (one subband) denoted by $G_{0,0}$, $G_{0,1}$, $G_{0,2}$, and $G_{0,3}$ at level 1. At level 2, two PEs are assigned to compute two groups of butterflies (one group is $G_{0,0}$ and $G_{0,1}$, the other group is $G_{1,0}$ and $G_{1,1}$), respectively. Similarly, four groups of butterflies are computed by four PEs at the subsequent level, respectively.

Figure 5.19 Direct-mapped architecture for 3-level DWPT

As described previously, this DM architecture computes a J -level DWPT decomposition in a high throughput, which produces the N -point final coefficients in $N/2$ cycles. However, the disadvantages are the high hardware complexity of $(N - 1)$

PEs and low efficient hardware utilization reduced by half at each level. In addition, the DM tree-structured implementation, as described in Figure 5.19, employs multiple clock domains with frequency rate reduced by half at each level.

B. Design of four D4 DWPT Processor Cores

In this study, four D4 DWPT processors cores employing the DM, recursive, DDC, and folded DDC architecture have been captured by VHDL/Verilog-HDL and the functionality is simulated and verified by RTL simulation in the Mentor Graphics ModelSim [63]. In these processor designs, we apply the zero-padding method as stated in [99] to deal with the boundary extension issue. Data in these designs is represented in fixed-point two's complement format. We use the method proposed in [101] to handle the overflow and precision problems. For the D4 filter, it is sufficient for the internal wordlength in each PE to be 2 bits more than the input signals. Thus, 16-bit wordlength is chosen for the I/O data and 18-bit precision is used for both the internal data and the multiplier coefficients in our designs.

The timing and hardware requirement comparison among the four different DWPT architectures is provided in Table 5.11. It takes both the DM architecture and DDC architecture only $N/2$ clock cycles to compute J -level DWPT. This is because they apply parallel or pipeline structure to improve the system throughput. As a penalty, the DM method has the highest hardware cost of $(2^J - 1)$ PEs. Our proposed DDC structure reduces the hardware complexity from $(2^J - 1)$ PEs to J PEs. The folded DDC architecture further decreases the number of PEs from J to 1. However, the processing time increases to around $NJ/2$ cycles because this folded architecture utilizes one PE to compute the DWPT recursively.

Table 5.11 Timing and hardware complexity comparisons of the proposed DWPT architectures

<i>J</i> -level <i>N</i> -point DWPT	Direct-Mapped	DDC	Recursive	Folded DDC
Processing Time (Clock Cycle)	$N/2$	$N/2$	$>NJ$	$\approx NJ/2$
Hardware Complexity	$(2^J - 1)$ PEs	J PEs	1 PE	1 PE

The recursive architecture has the longest processing time due to the limited I/O bandwidth of the memory. In this study, dual-port RAM is used by the recursive architecture to buffer the intermediate data and also used by the folded architecture to buffer the input data frame waiting to be processed. In the recursive architecture, the total time for the memory R/W operations during the computation is $(N/2) \times 2 \times J = NJ$ cycles as the memory has only two R/W ports. However, the actual processing time for a *J*-level DWPT is much longer than this value. Note that in Table 5.11, the PEs used by the DM, DDC, recursive, and folded DDC architectures are the direct implementation, the folded implementation, the direct implementation, or the configurable folded implementation of the lifting-based wavelet filter, respectively, as described in the previous sections. As we have provided the detailed comparisons of the hardware resources required by these architectures excluding the DM method to implement the 5/3 filter in Table 5.10, the detailed comparisons on hardware requirements by the D4 filter are not repeated in this section.

To estimate the timing and area size information, the four DWPT processor cores are synthesized by the Synopsys Design Compiler [64] into the gate level with the latest CMOS technology available, that is, the TSMC (Taiwan Semiconductor Manufacturing Company Ltd) 0.18 μm CMOS digital library. The estimated timing and area results are given in Table 5.12. The estimated main clock period of the four DWPT cores is 10 ns. The DM and DDC architectures have the shortest processing time of 40 ns because they employ parallel or pipeline processing to improve the

throughput. The folded DDC architecture computes the 3-level D4 DWPT much faster than the recursive method, as the folded architecture replaces the memory with switches and registers.

Compared with the DM architecture, the DDC architecture not only achieves a 53.3% saving in silicon area size, but also has a regular ladder structure with a single clock, because the DDC architecture uses the folding technique on the PEs in a pipeline. It can be seen from Table 5.13 that both the folded PE1 and PE2 have a little amount of increase in area size, that is, 1.3% and 2.2%, respectively, as compared to the DDC core area. Thus, the folding register overhead of the folded PEs in the D4 DDC architecture is insignificant when the total area is compared. In contrast to the DDC method, the DM architecture has a tree structure with multiple clock domains, which may cause design difficulties during the Place and Route phase of ASIC design. Compared with the DDC architecture, the folded DDC architecture results in a further 61.9% reduction in area size. It can be seen from Table 5.12 that the circuit areas consumed by the folded DDC and recursive architectures are almost identical. This indicates that the overhead effect of the configurable DDC and folding registers in the folded DDC architecture is minor. We can observe from Table 5.12 that among these architectures, when the hardware complexity in terms of gate count and area size decreases (or increases) significantly, the time performance in terms of processing time improves (or degraded) dramatically. This phenomenon demonstrates that when a designer chooses a dedicated architecture for a given algorithm, the timing performance and hardware complexity must be traded off between one another.

Table 5.12 Estimated timing and area size of the four DWPT architectures

3-level D4 DWPT	Direct-Mapped	DDC	Recursive	Folded DDC
Processing time (ns)	40	40	>240	120
Gate count	36453	17015	6538	6352
Area (μm^2)	1183852.5	548699.71	224166	209057.6
Processing time increased	-	0	5x	2x
Saving in area	-	53.3%	82.3%	82.6%

Table 5.13 Estimated area size of the DDC architecture for D4 DWPT

Modules	PE0	PE1	PE2	REG&SW	DDC DWPT core
Area (mm^2)	0.172	0.179	0.184	0.014	0.549
Percentage (%)	31.3%	32.6%	33.5%	2.6%	100%

We also implement the four designs in a Xilinx Virtex-II FPGA device (XC2V2000-4BG575) [66] using the Xilinx ISE (Integrated Software Environment) Software [67]. The estimated clock frequency for these designs is 98.406 MHz. Similar to the ASIC implementation results, the DM architecture has the largest device utilization of 1,821 slices [66] and 35 embedded two's complement 18×18 multipliers. The DDC architecture consumes 744 slices and 15 embedded 18×18 multipliers. Both the recursive and folded DDC architectures use only 5 embedded 18×18 multipliers. The slices consumed by these two architectures are 341 and 417, respectively. However, the recursive architecture uses one single-port block RAM (RAMB16S) more than the folded DDC architecture. Finally, the comparisons of the four D4 DWPT dedicated architecture are summarized in Table 5.14.

Table 5.14 Comparisons of different D4 DWPT hardware implementations

<i>J</i> -level <i>N</i> -point D4 DWPT processor	Direct-Mapped	DDC	Recursive	Folded DDC
Architecture	parallel	pipeline	recursive	folded
No. of PEs (3-level)	$2^J - 1$ (7)	J (3)	1	1
Arithmetic units	35 multipliers 28 adders	15 multiplier 12 adders	5 multipliers 4 adders	5 multipliers 4 adders
Gate count of processor core	36453	17015	6538	6352
Storage requirement Size ($N = 8$ and $M = 2$)	registers 28	registers $(M + 1)(N - 1)$ (28)	memory N (8)	registers $> 0.5(M + 1)N$ (> 12)
Processing cycles ($N = 8$)	$0.5N$ (4)	$0.5N$ (4)	$> NJ$ (> 24)	$\approx NJ/2$ (≈ 12)
Area size (μm^2) (0.18μ)	1183852.5	548699.71	224166	209057.6
Processing time (100 MHz @ 0.18μ)	40 ns	40 ns	> 240 ns	≈ 120 ns

5.5 Conclusions

In this chapter, three efficient dedicated DWPT architectures for computing multilevel lifting-based DWPT have been presented. The objective is to develop different DWPT architectures to provide a design flexibility to make trade-offs between hardware complexity and processing speed for different dedicated applications. Firstly, we have investigated various dedicated DWPT architectures in the literature. Subsequently, in order to unravel the inherent properties in the DWPT, we presented a regular SFG to describe the in-place DWPT algorithm by using the butterfly and group concepts. By

exploiting some useful properties, we derive three efficient architectures for computing J -level DWPT.

For low-cost applications, a recursive lifting-based DWPT architecture was proposed by exploiting the recursiveness property. Compared with the DM architecture, this recursive architecture obtains substantial hardware saving by reducing the number of PEs from $(2^J - 1)$ to one. Secondly, we presented a generalized pipeline DDC architecture with J folded PEs for high-speed DWPT computation. By exploiting the inherent parallelism and regularity, this DDC architecture obtains the features of high throughput, low memory operations, high hardware utilization, low hardware complexity, hardware regularity and simplicity. Case studies demonstrated this DDC architecture is easily scalable for J -level DWPT with different wavelets. Thirdly, we derived a folded DWPT architecture from the DDC architecture for area-constrained applications. By exploiting the structure similarity in both the folded PEs and DDCs of the DDC architecture, this folded architecture folds multilevel DWPT decomposition into a configurable PE with a configurable DDC. Compared to the recursive architecture, this folded architecture does not suffer from the issues related to on-chip memory requirement by avoiding intensive memory operations. Meanwhile, this folded architecture has a better timing performance and an identical hardware requirement as compared to the recursive architecture.

The circuit simulation and implementation results of four DWPT processor cores employing the DM, recursive, DDC and folded DDC architectures were provided to demonstrate the efficacy of these architectures. Performance evaluation of the proposed three architectures was performed by comparing the hardware requirements and timing performance based on the simulations and implementations. The ASIC and FPGA implementation results demonstrated that the DDC architecture outperforms the benchmark DM architecture in terms of hardware complexity, hardware utilization,

hardware regularity and simplicity, while maintaining the high-speed processing. Finally, comparative studies and discussions are conducted to provide an insight into design trade-offs between hardware complexity and processing speed for various dedicated applications with different constraints.

In conclusion, three efficient DWPT architectures have been proposed for computing multilevel DWPT decomposition. Comparative studies and analyses both at architecture level and on the basis of circuit simulation and implementation, demonstrate the efficacy of the proposed DWPT architectures. Case studies on design trade-offs between speed and area metrics show that the three proposed DWPT architectures can accommodate the different requirements on hardware cost and processing speed in various dedicated applications.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we investigated and developed efficient VLSI architectures for the fast Fourier transform (FFT), critical-band transform (CBT), and discrete wavelet packet transform (DWPT) for a variety of dedicated DSP applications.

In Chapter 1, we presented the motivations and objectives of this research work, and subsequently presented three major contributions of our research. In Chapter 2, we provided the background fundamentals and literature review of this research, including the VLSI design principles for DSP applications, low-power DSP VLSI design methodology, basics of the human perception models, and three popular discrete transforms (that is, the FFT, the CQT, and the DWT) with different transform resolutions.

In Chapter 3, we proposed a recursive FFT architecture for low-cost, low-power speech and audio applications. By exploiting the recursiveness and regularity properties in the radix-2 decimation-in-time (DIT) FFT algorithm, the proposed FFT architecture uses a novel butterfly unit interacting with a single memory unit to compute 256-point FFT recursively so as to reduce the hardware requirements. By rescheduling the radix-2 DIT butterfly arithmetic operations, we proposed a pipelined two-multiplier and three-adder butterfly structure with a good trade-off between

processing speed and hardware cost. Compared with the two-multiplier and four-adder benchmark, this proposed butterfly unit provides a low-cost solution with a 13% saving in silicon area, while maintaining the pipeline processing throughput of two cycles per butterfly operation. By employing the supply voltage scaling and pipeline processing techniques, this FFT architecture based on the proposed butterfly unit can obtain a 93% power reduction by scaling down the supply voltage from 3.3 V to 1.1 V. In summary, the proposed recursive FFT architecture features a low hardware complexity, low-power operation, and scalability for different-sized FFTs. Comparative studies based on the architecture level analysis, circuit simulation and implementation results have demonstrated that the proposed recursive FFT architecture achieves an effectively low hardware cost, obtains a significantly low power operation, and therefore, is a good solution for portable speech and audio applications which are constrained to low-cost and low-power operations.

In Chapter 4, we proposed an efficient CBT algorithm and its corresponding VLSI architecture for low-power speech applications. Unlike the FFT with a constant frequency resolution, the CBT with a nonuniform frequency resolution is developed to approximate the critical-band filtering of the human audition system. From the observation of the critical bandwidth of the human ear, we have formulated a generalized CBT algorithm, which consists of a constant-bandwidth transform in the lower frequency range and a Brown CQT in the higher frequency range. By designing and simulating a 21-band Bark scale CBT, we have shown that the proposed CBT algorithm is suitable for performing short-time spectral analysis on speech. Based on the comparative studies, we have also demonstrated that the proposed 21-band CBT has a better fitting to the critical bandwidth of the human hearing, significantly fewer computations (a saving up to 56.9%), fewer memory accesses (a reduction up to 77.6%), and therefore, is more energy-efficient than other methods, including the FFT and WPT methods.

By exploiting the regularity, resursiveness, and parallelism properties in the proposed CBT algorithm, we proposed an efficient VLSI architecture with a pipelined data path to calculate the CBT recursively. The computations and coefficient memory area required by the CBT computation are reduced significantly by exploiting the symmetric nature of the CBT coefficients. The circuit simulation results has proven that the proposed CBT architecture obtains significant power efficiency by reducing the computational complexity and memory accesses, using parallel and pipeline processing, and applying the supply voltage scaling technique. Comparative studies on the basis of the architecture level analysis, circuit simulation and implementation results have also shown that the proposed 21-band CBT processor achieves both significant power saving and hardware cost reduction in comparison to the benchmark 256-point FFT processor. With a $0.35\ \mu\text{m}$ CMOS technology, the proposed CBT processor achieves about 82.1% power reduction and 70.2% silicon area saving as compared to the FFT processor. Therefore, the proposed CBT architecture is a good low-cost and low-power implementation for speech applications.

In Chapter 5, we proposed three efficient VLSI architectures for computing J -level lifting-based DWPT. Due to its property of flexible time and frequency decomposition, the DWPT has advantages over the FFT and CBT in a wide range of DSP applications. In order to unravel the useful inherent properties in the DWPT, we have presented a regular SFG to describe the in-place DWPT algorithm by using the butterfly and group concepts. By exploiting several useful properties, we developed three DWPT architectures, providing a flexibility to make design trade-offs between hardware cost and processing speed for different dedicated applications.

Firstly, by exploiting the regularity and recursiveness properties, we have proposed a recursive lifting-based DWPT architecture for low-cost applications. Compared with the DM tree-structured architecture, this recursive architecture obtains substantial hardware saving by reducing the number of PEs from $(2^J - 1)$ to one.

Secondly, by exploiting the regularity and parallelism properties, we developed a generalized pipeline DDC architecture with J folded PEs for high-speed applications. The proposed DDC architecture features the characteristics of high throughput, low memory operations, high hardware utilization, low hardware complexity, hardware regularity and simplicity. Case studies have demonstrated that this DDC architecture is easily scalable and redesigned for J -level DWPT with different wavelets. Thirdly, by exploiting the structure similarity in both the folded PEs and DDCs in the DDC architecture, we derived a folded DWPT architecture from the DDC architecture for area-constrained applications. With extra multiplexers and registers, this folded architecture folds multilevel DWPT decomposition into one configurable PE with a configurable DDC. Compared with the recursive architecture, this folded architecture does not have the bottlenecks due to the on-chip memory requirement by avoiding intensive memory operations. This folded architecture also has a better timing performance and has an identical hardware requirement with comparison to the recursive architecture.

Circuit simulation and implementation results have verified the efficacy of these proposed DWPT architectures. Specifically, the ASIC and FPGA implementation results of an 3-level D4 DWPT have demonstrated that the DDC architecture outperforms the benchmark DM architectures in terms of hardware complexity (a 53.3% reduction in silicon area), hardware utilization (50%), hardware regularity and simplicity, while maintaining the high-speed processing (a throughput of 40 ns). In a comparison of the four DWPT architectures on the basis of simulation and implementation results, we provided an insight into design trade-offs between hardware complexity and processing speed for various dedicated applications with different design constraints. When a designer chooses a dedicated architecture for a given algorithm, the timing performance and hardware cost must be traded off between one another.

In summary, three major contributions in this thesis are on the investigation and development of dedicated architectures for three discrete transforms with different transform resolutions. By exploiting the inherent properties in these transforms, we provided efficient VLSI solutions of the discrete transforms for a variety of dedicated DSP applications. Comparative studies on the basis of the architecture level analyses, circuit simulation and implementation results have proven the efficiency of these proposed architectures. Case studies into design trade-offs between speed and area metrics have also shown that specific architectures derived from different design trade-offs can accommodate various dedicated applications with different design constraints. The proposed FFT architecture is a good solution for low-cost speech and audio applications, such as speech recognizers, hearing-aids, speech and audio coders. The proposed CBT processor can be used as the front-end spectral analyzer in low-cost and low-power speech applications. The proposed three DWPT architectures accommodate a wide range of real-time DSP applications, from low sampling rate speech and audio applications, to high-throughput image and video applications, as well as high-speed digital communications. The three major contributions have led to the journal and conference paper publications listed in the author's publications at the end of this chapter.

6.2 Recommendations for Future Work

Based on the research work presented in this thesis, we will now suggest several potential relevant topics for future research.

A. Dedicated FFT Architecture for OFDM-based Communications

As described in Chapter 2, the FFT is the most popular algorithm used in a wide range of applications. In particular, the FFT has played a very crucial role in the OFDM-

based high-speed digital communications [49], such as DAB (digital audio broadcasting), DVB (digital video broadcasting), UWB (ultrawideband), MIMO (multiple-input-multiple-output) OFDM systems [52]-[54]. As the demand for wireless communication products increases explosively, we suggest further research on the dedicated FFT architectures for the OFDM-based communication systems.

As the pipeline FFT architecture is a good approach for high-speed OFDM-based communications, we suggest to review the pipeline FFT architectures that have been proposed for a variety of high-throughput DSP applications [52]-[54], [102]. As depicted in Figure 3.2, the pipeline FFT architectures can be classified into two categories: multipath delay commutator (MDC) scheme and single-path delay feedback (SDF) scheme, which are briefly described respectively as follows.

- MDC: In general, by employing dedicated buffers (called delay commutators) to reorder the data appropriately between different stages, the MDC architecture can use just one butterfly unit at each stage to process M parallel data streams simultaneously with high throughput. As in the straightforward approach for pipeline implementation of radix-2 FFT algorithm, the radix-2 MDC (R2MDC) [55] breaks the input sequence into two parallel data streams, employs delay commutators to reorder the data between any two consecutive stages, and uses one radix-2 butterfly unit at each stage to process the data in the right order presented from the previous stage. In this architecture, $\log_2 N$ radix-2 butterfly units (complex multipliers) and $3/2N - 2$ registers are required [53].
- SDF: The SDF [56] uses the buffer more efficiently by storing the butterfly output in feedback shift registers. In the SDF scheme, a single data stream goes through the butterfly unit in every stage. For simplicity, we discuss the radix-2 SDF (R2SDF) to illustrate the SDF scheme. At each stage in the

R2SDF, half of the input data is processed to perform butterfly operations, while the other half is fed back into the shift register. When all the butterfly operations have been completed, the other half of the data from the shift registers is allowed to be processed by the butterfly unit, while the first half of the next input data is being loaded into the shift register. There is no break in the data stream between one transform and the next. The R2SDF method has the same number of butterfly units as in the R2MDC approach, while it has fewer memory requirement of $N - 1$ registers. In general, the MDC scheme can obtain a higher processing throughput, while the SDF scheme requires lesser memory and hardware resources.

In the past two decades, many researchers proposed different pipeline architectures, of which most are evolved from the conventional MDC and SDF architectures [52]-[54], [102]. Because the fast evolving communications standards impose different requirements into the dedicated FFT processor designs, we suggest the research on dedicated pipeline architectures to meet the different constraints, such as the different length of the FFT, the varying number of the channels, the different neighbouring functional modules, etc, by making different design trade-offs between the speed, area, and power metrics.

B. Dedicated DWPT Architectures for Arbitrary Sub-trees

Due to the inherited properties from the DWT and the characteristic of flexible TF decomposition, the DWPT method has successfully implemented in many applications, as described in Chapter 5. The DWPT can provide optimal decomposition trees adapting to the requirement of dedicated applications and the characteristics of the input signals. For example, the entropy-based criterion can be used to select the best decomposition tree for data and signal compression applications [87]. As we described in Chapter 3, the DWPT with a predefined tree

structure can be used to decompose the input speech signal into the approximated critical bands, for different speech applications [73], [74], [77].

In order to provide DWPT decomposition with any required sub-tree, we suggest further research on the proposed DDC architecture in Chapter 5. In order to improve the DDC architecture as shown in Figure 5.10 and 5.11, we can modify the PEs or folded PEs in the architecture to support normal and bypass operations. By adding extra control signals denoted by byp_ji (for example, $j = 0, 1, 2, \dots, N-1$, $i = 1, 2, 3$ and 4, for D4 wavelet filter, as shown in Figure 6.1) to set the bypass mode to the arithmetic units and delay registers, the operations at each internal pipeline stage can be configured to operate in normal mode or bypass mode. The control signal byp_ji is used to determine the operation mode of the multipliers, adders and delay registers at the i th internal pipeline stage of the j th decomposition level. Thus, in this way, the DDC architecture with the configurable PE can be easily configured to compute DWPT with any arbitrary sub-tree by appropriate configuring the folded PEs at each level. Similarly, we can also modify the proposed recursive and folded DDC architectures to produce arbitrary TF decompositions.

Figure 6.1 Proposed folded architecture of the lifting-based D4 filter with bypass mode

C. Generalized DDC Architectures for M -Band DWPT

Unlike the traditional 2-band (dyadic) DWPT decomposes the signal from a given frequency band into only two frequency subbands, the generalized M -band DWPT decomposes the signal into M ($M = 2, 3, 4, \dots$) frequency subbands. For example, the 4-band J -level DWPT expands the signals into 4^j ($j = 0, 1, 2, \dots, J-1$) subbands at level j . Compared with the 2-band DWPT, this M -band DWPT leads to more compact representations, which makes it useful in many applications. From a hardware implementation perspective, the M -band DWPT decomposes the input signal into multiple subbands much faster than the traditional way. However, the hardware complexity increases by two and the hardware utilization reduces by half, which is similar to the tree-structured DM architecture for multilevel 2-band DWPT, as described in Chapter 5.

Fortunately, we can generalize our DDC architecture for the 2-band DWPT into a general architecture, called MDC (multiple-delay path commutator) architecture for computing multilevel M -band lifting-based DWPT decomposition, because the M -band DWPT inherits the regularity and parallelism properties of the in-place 2-band DWPT. In general, by employing dedicated buffers (called MDCs) to reorder the data appropriately between different stages, the MDC architecture can use just one folded PE at each stage to process M parallel data streams simultaneously with high throughput. For a practical implementation, provided that $N = M^r$, $M = 2^p$, where $p = 1, 2, 3, \dots$, there are generally $\log_r N$ folded PEs interleaved with a series of buffer memories in the MDC architecture. At each stage, this pipeline architecture maps multiple subband decomposition into just one PE. Similar with the DDC architecture, the MDC architecture uses cascaded folded PEs to process the data to obtain high-speed operation by employing the switches and registers to reorder the data streams at each stage. The folded PE can also be derived by using the butterfly-based SFG to

represent the multilevel M -band DWPT and employing folding techniques similar with the DDC architecture.

Obviously, the MDC architecture inherits the excellent advantages from the DDC architectures as described in Chapter 5. In addition, the MDC architecture with $M > 2$, computes the multilevel decomposition significantly faster than the DDC architecture. For instance, a 4-band folded PE at each level in the 4-band MDC architecture operates as twice as fast as the 2-band DDC architecture. This advantage is very useful in reducing the latency significantly for the multilevel DWPT with long filter taps and high decomposition levels. However, the lifting factorization of the 4-band wavelet filters might probably requires more arithmetic units and but fewer arithmetic operations than the 2-band wavelet filters.

The proposed recursive and folded DDC architecture presented in Chapter 5 can also be extended to implement the M -band DWPT.

D. Dedicated Architectures for M -Band DWT

As described previously, the M -band DWT decomposes the input signal in the similar fashion as the traditional 2-band DWT, except that the M -band DWT decomposes the input signal or the lowest frequency coefficients at each level into the subsequent M frequency subbands. Thus, the M -band DWT algorithm inherits many characteristics of the 2-band DWT, except that increased hardware complexity at each level.

Therefore, we suggest to extend the dedicated architectures for the traditional 2-band DWT reported in [86] to the M -band DWT. For example, the recursive DWT architecture in [99] computes multilevel DWT decomposition simultaneously by exploiting the idle time of the same folded datapath to improve the hardware utilization and processing speed. As the M -band multilevel DWT also has unbalanced computational load distributed over the filters at different levels (reduced by M at each new level), we could utilize the multirate folding technique in [22] to interleave

different level decomposition into one folded datapath and propose a folded M -band DWT architecture similar to the one in [99]. In a similar way, different architectures for 1-D and 2-D M -band DWT can also be proposed to meet different design goals as stated in [86], to increase throughput, improve hardware utilization, reduce critical path, or minimize hardware complexity, etc.

E. Dedicated Architectures for 2-D DWPT

As we have described in Chapter 5, there are very few papers about the dedicated DWPT architectures, mainly because the DWPT requires significantly more hardware requirements than the DWT. Similarly, the 2-D DWPT is significantly more computational-intensive and memory-intensive, which requires more arithmetic units and storage units. The dedicated hardware implementation becomes a challenge for the 2-D DWPT due to the stringent hardware requirement. This difficulty is probably the main reason why there are almost no papers in the literature reporting the dedicated architecture for the 2-D DWPT. However, the flexibility of TF decomposition has made the 2-D DWPT successfully used in many 2-D signal processing applications, including image and video applications.

The challenge of dedicated hardware implementation for the 2-D DWPT not only reveals its great design difficulty, but also indicates much research potential in this topic. We suggest new research on the topic of dedicate architecture for the 2-D DWPT. As the direct approach for 2-D implementation, a 1-D DWPT row-wise and then a 1-D DWPT column-wise are applied at each decomposition level to produce multiple subbands. Because the DWPT is essentially a filtering operation, the column-wise processing can only begin when all the rows have been filtered, which requires a lot of storage resources. As the most important design consideration in 2-D architecture is the memory configuration for both 2-D DWT and 2-D DWPT, we suggest that research work on the investigation for the different memory configuration

techniques used in the existing 2-D DWT architectures [86]. We also suggest to explore efficient memory scan techniques suitable for efficient VLSI implementations of the 2-D DWPT. Additionally, we suggest to explore the algorithm properties of the 2-D DWPT algorithm, which might be useful for developing efficient 2-D DWPT architectures.

Author's Publications

Papers in Peer Review Journals

1. Chao Wang and Woon-Seng Gan, "Efficient VLSI Architecture for Lifting-Based Discrete Wavelet Packet Transform", *IEEE Transaction on Circuits and Systems Part II*, vol. 54, no. 5, pp. 422-426, May 2007.
2. Chao Wang and Woon-Seng Gan, "Efficient Algorithm and Architecture of Critical-Band Transform for Low-Power Speech Applications," *EURASIP Journal on Advances in Signal Processing*, vol. 2007, Article ID 89264, 10 pages, 2007.
3. Chao Wang and Woon-Seng Gan, "VLSI Architectures for Lifting-Based Discrete Wavelet Packet Transform", submitted to *Journal of VLSI Signal Processing*.

Papers Presented at Conferences

1. Chao Wang and Yit-Chow Tong, "An Improved Critical-band Transform Processor for Speech Applications," in *Proc. of 2004 IEEE International Symposium on Circuits and Systems (ISCAS 2004)*, vol. 3, pp. 461-464, May 23-26, 2004, Vancouver, Canada.
2. Chao Wang, Yit-Chow Tong, and Yu Shao, "VLSI Design and Analysis of a Critical-band Transform Processor for Speech Recognition," in *Proc. of 2004*

Author's Publications

- IEEE International SOC (System-on-Chip) Conference (SOCC 2004)*, pp. 365-368, Sep. 12-15, 2004, Santa Clara, USA.
3. Yu Shao, Yit-Chow Tong, and Chao Wang, "Wavelet Statistical Model of Speech for Feature Extraction and Denoising," in *Proc. of 2004 IEEE Asia-Pacific Conference on Circuits and Systems (APCCAS 2004)*, vol. 1, pp. 189-192, Dec. 6-9, 2004, Taiwan.
 4. Chao Wang and Woon-Seng Gan, "Efficient VLSI architecture of lifting-based wavelet packet transform for audio and speech applications," in *Proc. of 2006 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2006)*, vol. 3, pp. 940-943, May 14-19, 2006, Toulouse, France.
 5. Chao Wang, Woon-Seng Gan, Ching-Chuen Jong, and Jianwen Luo, "A Low-Cost 256-Point FFT Processor for Portable Speech and Audio Applications", *2007 International Symposium on Integrated Circuits (ISIC 2007)*, pp. 81-84, Sep. 26-28, 2007, Singapore.
 6. Chao Wang, Woon-Seng Gan, Xiaomeng Shi, and Kiat-Seng Yeo, "VLSI Architectures for Lifting-Based Discrete Wavelet Packet Transform", *2007 International Symposium on Integrated Circuits (ISIC 2007)*, pp. 188-191, Sep. 26-28, 2007, Singapore.

Reference

Reference

- [1] Keshab K. Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation*. USA: Wiley-Interscience, 1999.
- [2] John G. Proakis and Dimitris G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications*, 3rd edition. New Jersey: Prentice Hall, 1995.
- [3] Sanjit K. Mitra. *Digital Signal Processing: a Computer-Based Approach*, 2nd edition. Irwin: McGraw-Hill, 2001.
- [4] Jan M. Rabaey. *Digital Integrated Circuits: a Design Perspective*, 2nd edition. New Jersey: Prentice Hall, 2003.
- [5] L. Benini and G. De. Micheli, “Networks on chips: a new SoC paradigm”, *IEEE Computer Magazine*, vol. 35, no. 1, pp. 70-78, 2002.
- [6] K. S. Yeo, S. S. Rofail, and W. L. Goh. *CMOS/BiCMOS ULSI, Low Voltage, Low Power*. New Jersey: Prentice Hall, 2002.
- [7] S. Hamilton, “Taking Moore’s law into the next century”, *IEEE Computer Magazine*, vol. 32, no. 1, pp. 43-48, 1999.
- [8] S. Y. Kung. *VLSI Array Processors*. New Jersey: Prentice Hall, 1988.
- [9] Edward A. Lee, “Programmable DSP architectures: part I”, *IEEE ASSP Magazine*, vol. 5, no. 4, pp. 4-19, 1988.
- [10] Edward A. Lee, “Programmable DSP architectures: part II”, *IEEE ASSP Magazine*, vol. 6, no. 1, pp. 4-14, 1989.

Reference

- [11] Important dates in Xilinx[®] History, available at:

<http://www.xilinx.com/company/history.htm>.
- [12] Jennifer Eyre and Jeff Bier, "DSP processors hit the mainstream", *IEEE Computer*, vol. 31, no. 8, pp. 51-59, 1998.
- [13] Ravi K. Kolagatla, Jose Fridman, Bradley, et al, "High performance dual-MAC DSP architecture", *IEEE Signal Processing Magazine*, vol. 19, no. 4, pp. 42-53, 2002.
- [14] Xilinx. *Virtex-II ProTM Platform FPGA Handbook*. UG012(v2.0) October 14, 2002.
- [15] A. P. Chandrakasan and R. W. Brodersen, "Minimizing power consumption in digital CMOS circuits," *Proc. of IEEE*, vol. 83, pp. 498-523, Apr. 1995.
- [16] L. R. Rabiner and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975.
- [17] C. Mead and L. Conway. *Introduction to VLSI systems*. Addison-Wesley, 1980.
- [18] B. Randell and P. C. Treleaven. *VLSI Architecture*. New Jersey: Prentice Hall, 1982.
- [19] L. Wanhammar. *DSP Integrated Circuits*. New York: Academic Press, 1999.
- [20] K. J. R. Liu, A. Y. Wu, A. Raghupathy, and J. Chen, "Algorithm-based low-power and high-performance multimedia signal processing", *Proc. of the IEEE*, vol.86, pp.1155-1202, June 1998.
- [21] Keshab K. Parhi, C. Wang, and Andrew P. Brown, "Synthesis of control circuits in folded pipelined DSP architectures", *IEEE Journal of Solid-State Circuits*, vol. 27, no. 1, pp. 29-43, Jan. 1992.

Reference

- [22] Tracy C. Denk and Keshab K. Parhi, "Synthesis of folded pipelined architectures for multirate DSP algorithms", *IEEE Transactions on VLSI Systems*, vol. 6, no. 4, pp. 595-607, Dec. 1998.
- [23] A. Bellaouar and M. I. Elmasry. *Low-Power Digital VLSI Design*. Norwell, MA: Kluwer Academic Publishers, 1995.
- [24] A. P. Chandrakasan and R. W. Broderon. *Low Power Digital CMOS Design*. Norwell, MA: Kluwer Academic Publishers, 1995.
- [25] G. Yeap. *Practical Low Power Digital VLSI Design*. Norwell, MA: Kluwer Academic Publishers, 1998.
- [26] A. P. Chandrakasan, S. Sheng and R. W. Broderon, "Low-power CMOS digital design," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 4, pp. 473-484, Apr. 1992.
- [27] T. Arslan, A. T. Erdogan, and D. H. Horrocks, "Low power design for DSP: methodologies and techniques", *Microelectronics Journal*, vol. 27, no. 8, pp. 731-744, Nov. 1996.
- [28] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. W. Broderon, "Optimizing power using transformations", *IEEE Transaction on Compute-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 1, pp. 12-31, Jan. 1995.
- [29] J. Rabaey, L. Guerra, and R. Mehra, "Design guidance in the power dimension", *IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, pp. 2837-2840, May. 1995.
- [30] R. Mehra and J. Rabaey, "Exploiting regularity for low-power design", *IEEE/ACM International Conference on Compute-Aided Design*, pp. 166-172, Nov. 1996.

Reference

- [31] A. P. Chandrakasan, "Ultra low power digital signal processing", *Proceedings of 9th International Conference on VLSI Design*, pp. 352-357, Jan. 1995.
- [32] B. Gold and N. Morgan. *Speech and Audio Signal Processing: Processing and Perception of Speech and Music*. New York, NY: John Wiley & Sons Inc., 1999.
- [33] E. Zwicker and H. Fastl. *Psychoacoustics: Facts and Models*, second updated edition. Springer, 2006.
- [34] J. W. Picone, "Signal modelling techniques in speech recognition", *Proc. of the IEEE*, vol. 81, no. 9, pp. 1215-1247, Sep. 1993.
- [35] N. Jayant, J. Johnston, and R. Safranek "Signal compression based on models of human perception", *Proc. of the IEEE*, vol. 81, no. 10, pp. 1385-1422, Oct. 1993.
- [36] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*, 2nd edition. New Jersey: Prentice Hall, 2002.
- [37] D. J. Granrath, "The role of human visual models in image processing", *Proc. of the IEEE*, vol. 69, no. 5, pp. 552-561, May 1981.
- [38] H. Fletcher, "Auditory patterns", *Res Modern Physics*, 12, pp. 47-65, 1940.
- [39] P. Noll, "Digital audio coding for visual communications", *Proc. of the IEEE*, vol. 83, no. 6, pp. 925-943, June 1995.
- [40] E. Zwicker, "Subdivision of the audible frequency range into critical bands (Frequenz-gruppen)," *Journal of the Acoustical Society of America*, vol. 33, pp. 248, 1961.
- [41] C. F. Stromeyer and B. Julesz, "Spatial frequency masking in vision: critical bands and spreading of masking", *Journal of the Optical Society of America*, pp. 1221-1232, Oct. 1972.

Reference

- [42] J. Cooley and J. Tukey, "An algorithm for the machine calculation of the complex Fourier series," *Mathematics of Computations*, vol. 19, pp. 297-301, 1965.
- [43] J. C. Brown, "Calculation of a constant Q spectral transform," *Journal of the Acoustical Society of America*, vol. 89, pp. 425-434, 1991.
- [44] S. G. Mallat, "A theory for multiresolution signal decomposition: the wavelet representation", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, no. 7, pp. 674-693, July 1989.
- [45] S. Mallat. *A Wavelet Tour of Signal Processing*. New York: Academic Press, 1998.
- [46] S. Sweldens, "The lifting scheme: a custom-design construction of biorthogonal wavelet", *Applied and Computational Harmonic Analysis*, vol. 3, pp. 186-200, 1996.
- [47] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps", *Applied Fourier Analysis Application*, vol. 4, pp. 247-269, 1998.
- [48] Y. Park, D. Kim, and I. Kim, "Design of high-performance digital hearing aid processor," *IEE Electronic Letters*, vol. 34, no. 17, pp. 1631-1632, Aug. 1998.
- [49] D. Sinha and A. H. Tewfik, "TDCS, OFDM, and MC-CDMA: a brief tutorial," *IEEE Communications Magazine*, vol. 43, no. 9, pp. S11-S16, Sep. 2005.
- [50] E. Cetin, R. C. S. Morling, and I. Kale, "An extensible complex fast Fourier transform processor chip for real-time spectrum analysis and measurement," *IEEE Transactions on Instrumentation and Measurement*, vol. 47, no. 1, pp. 95-99, Feb. 1998.

Reference

- [51] B. M. Bass, "A low-power, high-performance 1024-point FFT processor," *IEEE Journal of Solid-State Circuits*, vol. 34, no. 3, pp. 380-387, Mar. 1999.
- [52] E. Bidet, D. Castelain, C. Joanblanq, and P. Senn, "A fast single-chip implementation of 8912 complex point FFT," *IEEE Journal of Solid-State Circuits*, vol. 30, no. 3, pp. 300-305, Mar. 1995.
- [53] S. He and M. Torkelson, "Designing pipeline FFT processor for OFDM (de)modulation," in *Proceedings of URSI International Symposium on Signals, Systems, and Electronics*, vol. 29, pp. 257-262, 1998.
- [54] Y. Lin, H. Liu, and C. Lee, "A 1-GS/s FFT/IFFT processor for UWB applications," *IEEE Journal of Solid-State Circuits*, vol. 40, no. 8, pp. 1726-1735, Aug. 2005.
- [55] L. R. Rabiner and B. Gold. *Theory and Applications of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1975.
- [56] E. H. Wold and A. M. Despain, "Pipeline and parallel-pipeline FFT processors for VLSI Implementation," *IEEE Transaction on Computers*, vol. 33, no. 5, pp. 414-426, May 1984.
- [57] G. Sunada, J. Jin, M. Berzins, and T. Chen, "COBRA: a 1.2 million transistor expandable column FFT chip", in *Proc. of IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp.546-550, Oct. 1994.
- [58] Z. Liu, Y. Song, T. Ikenaga, and S. Goto, "A VLSI array processing oriented fast Fourier transform algorithm and hardware implementation," *IEICE Transaction Fundamentals*, vol. E88-A, no. 12, pp. 291-295, Dec. 2005.
- [59] Tran-Thong and B. Liu, "Fixed-point fast Fourier transform error analysis," *IEEE Transaction on Acoustics, Speech, Signal Processing*, vol. 24, no. 6, pp. 563-573, 1976.

Reference

- [60] H. Luo, M. Shieh, and C. Wu, "Design of an efficient FFT processor for DAB system," in *Proc. of IEEE International Symposium on Circuits and Systems*, vol. 4, IV-654-IV-657, May 2001.
- [61] C. Wu, M. Shieh, H. Luo and M. Hu, "Implementation of channel demodulator for DAB system," in *Proc. of IEEE International Symposium on Circuits and Systems*, vol. 2, II-137-II-140, May 2003.
- [62] P. A. Ruetz and M. M. Cai, "A real time FFT chip set: architecture issues," in *Proc. of 10th International Conference on Pattern Recognition*, vol.2, pp.385 – 388, 1990.
- [63] Mentor Graphics. *ModelSim User's Manuel*. Version 6.1, May 18, 2005.
- [64] Synopsys. *Design Compiler User Guide*. TLD-2001.06, Jun. 2001.
- [65] Synopsys. *NanoSim User Guide*. Version U-2003.03, Mar. 2003.
- [66] Xilinx. *Virtex-II Platform FPGA User Guide*. UG002(v2.0) March 23, 2005.
- [67] Xilinx. *Xilinx ISE 8 Software Manuals*. 2005.
- [68] A. Delaruelle, J. Huisken, J. V. Loon, and F. Welten, "A chip set for a digital audio broadcasting channel broadcasting", in *Proc. of IEEE Custom Integrated Circuits Conference*, pp. 47-50, 1994.
- [69] B. A. Daurich, L. R. Rabiner, and T. B. Martin, "On the effects of varying filter bank parameters on isolated word recognition," *IEEE Transaction on Acoustics, Speech and Signal Processing*, vol. 31, no. 4, pp. 793-807,1983.
- [70] S. B. Davis, and P. Mermelstein, "Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences," *IEEE Transaction on Acoustics, Speech and Signal Processing*, vol. 28, no. 4, pp. 357-366, 1980.

Reference

- [71] T. L. Peterson and S. F. Boll, "Critical band analysis-synthesis," *IEEE Transaction on Acoustics, Speech and Signal Processing*, vol. 31, no. 3, pp. 656-663, 1983.
- [72] J. M. Kates, "An auditory spectral analysis model using the chirp z-transform," *IEEE Transaction on Acoustics, Speech and Signal Processing*, vol. 31, no. 1, pp. 148-156, 1983.
- [73] B. Carnero and A. Drygajlo, "Perceptual speech coding and enhancement using frame-synchronized fast wavelet packet transform algorithms," *IEEE Transaction on Signal Processing*, vol. 47, no. 6, pp. 1622-1635, 1999.
- [74] O. Farooq and S. Datta, "Mel filter-like admissible wavelet packet structure for speech recognition," *IEEE Signal Processing Letters*, vol. 8, pp. 196-198, 2001.
- [75] G. Zhou, John. H. L. Hansen, and J. F. Kaiser, "Nonlinear feature based classification of speech under stress", *IEEE Transaction on Speech and Audio Processing*, vol. 9, no. 3, pp. 201-216, Mar. 2001.
- [76] James D. Johnston, "Transform coding of audio signals using perceptual noise criteria", *IEEE Journal on Selected Areas in Communications*, vol. 6, no. 2, pp. 314-323, Feb. 1988.
- [77] D. Sinha and H. Tewfik, "Low bit rate transparent audio compression using adapted wavelets", *IEEE Transaction on Signal Processing*, vol. 41, no. 12, pp. 3463-3479, Dec. 1993.
- [78] MathWorks. *Getting Started with Matlab*. Aug. 2005.
- [79] University of Arizona. *Cool Edit Pro User Guide*. Jun. 2002.
- [80] L. Rabiner and B. Juang. *Fundamentals of Speech Recognition*. Englewood Cliffs, NJ: Prentice Hall, 1993.

Reference

- [81] J. N. Holmes and W. J. Holmes. *Speech Synthesis and Recognition*, 2nd Edition, Taylor & Francis, pp. 111-112, 2001.
- [82] JPEG2000 Part II Final Committee Draft (2000, Dec.). URL: <http://www.jpeg.org>.
- [83] ISO/IEC. ISO/IEC 15444-1. Information Technology – JPEG2000. Image Coding System. 2000.
- [84] ISO/IEC JTC/SC29/WG11. N2502a. Generic Coding of Audio-Visual Objects: Visual 14496-2. Final Draft IS. Atlantic City. Dec. 1998.
- [85] T. Hopper, C. Brislawn, and J. Bradley. WSQ Gray-scale Fingerprint Image Compression Specification. Federal Bureau of Investigation, Feb. 1993. {IAFIS-IC-0110v2}.
- [86] T. Acharya and C. Chakrabarti, “A survey on lifting-based discrete wavelet transform architectures”, *Journal of VLSI Signal Processing*, vol. 42, no. 3, DOI: 10.1007/s11266-006-4191-3, Mar. 2006.
- [87] R. R. Coifmen and M. V. Vickerhauser, “Entropy-based algorithms for best basis selection,” *IEEE Transaction on Information Theory*, vol. 38, pp. 713-718, Mar. 1992.
- [88] Z. Xiong, K. Ramchandran, and M. T. Orchard, “Wavelet packet image coding using space-frequency quantization,” *IEEE Transaction on Image Processing*, vol. 7, pp. 160-174, 1998.
- [89] H. Khalil, A. Jacquin, and C. Podilchuck, “Constrained wavelet packets for tree-structured video coding algorithms,” in *Proc. of Data Compression Conference*, pp. 354-363, Mar. 1999.

Reference

- [90] L. A. Drake, J. C. Rutledge, and J. Cohen, "Wavelet analysis in recruitment of loudness compensation," *IEEE Transaction on Signal Processing*, vol. 41, no. 12, pp. 3306-3312, 1993.
- [91] A. R. Lindsey, "Generalized orthogonally multiplexed communication via wavelet packet bases", Ph. D. Thesis, Ohio University, 1995.
- [92] X. Wu, Y. Li, and H. Chen, "Programmable wavelet packet transform processor," *IEE Electronics Letters*, vol. 35, no. 6, pp. 449-450, 1999.
- [93] M. A. Trenas, J. Lopez, M. Sanchez, F. Arguello, and E. L. Zapata, "Architecture for wavelet packet transform with best tree searching," in *Proc. of IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pp. 289-298, 2000.
- [94] S. M. Aroutchelvame and K. Raahemifar, "Architecture of wavelet packet transform for 1-D signal," in *Proc. of 2005 IEEE Canadian Conference on Electronics and Computer Engineering*, pp. 1304-1307, May 2005.
- [95] G. Paya, M. M. Peiro, F. Ballester, V. Herrero, and F. Mora, "Lifting folded pipelined discrete wavelet packet transform architecture," *Proceedings of the SPIE*, vol. 5117, pp. 321-328, 2003.
- [96] M. A. Trenas, J. Lopez, and E. L. Zapata, "A configurable architecture for wavelet packet transform," *Journal of VLSI Signal Processing*, vol. 32, pp. 255-273, 2002.
- [97] F. Arguello, J. Lopez, M. A. Trenas, and E. L. Zapata, "Architecture for wavelet packet transform based on lifting steps," *Journal of parallel computing*, vol. 28, no. 7-8, pp. 1023-1037, 2002.

Reference

- [98] P. Chen, "VLSI implementation for one-dimensional multilevel lifting-based wavelet transform", *IEEE Transaction on Computers*, vol. 53, no. 4, pp. 2004, Apr. 1997.
- [99] H. Liao, M. Kr. Mandal, and B. F. Cockburn, "Efficient architecture for 1-D and 2-D lifting-based wavelet transforms," *IEEE Transaction on Signal Processing*, vol. 52, no. 5, pp. 1315-1326, May 2004.
- [100] J. Jensen and A. la Cour-Harbo. *Ripples in Mathematics: The Discrete Wavelet Transform*. Berlin and Heidelberg: Springer-Verlag, GmbH & Co. K, Jun. 2001.
- [101] C-T Huang, P-C Tseng, and L-G Chen, "Flipping structure: an efficient VLSI architecture for lifting-based discrete wavelet transform," *IEEE Transaction on Signal Processing*, vol. 52, no. 4, pp. 1080-1089, Apr. 2004.
- [102] W. Yeh and C. Jen, "High-speed and low-power split-radix FFT," *IEEE Transaction on Signal Processing*, vol. 51, vol. 3, pp. 864-874, Mar. 2003.

Appendices

Appendices

Appendix A

Selected VHDL Codes for the Proposed Recursive FFT Processor

A. Structure Design of the Two-Multiplier and Three-Adder Butterfly Unit

```

-----
-- Module: datapath_bfly23.vhd
-- Function: the data path of the proposed recursive FFT architecture
--           the proposed pipelined two-multiplier and three-adder butterfly unit
--
-- Inputs:   ctrl--control signals from the FFT controller
--           rst--asynchronous reset signal
--           din1 (AR), din2 (AI), din3 (BR), din4 (BI), din5 (WR), din6 (WI)
-- Outputs:  dout1 (CI or CR), dout2 (DI or DR)
--
-- Author:   Wang Chao
-- Institute: ISR Lab-EEE2-NTU
-- Date:     12-05-2002
-- Simulator: ModelSim (Mentor Graphics)
-- Synthesis: Design Compiler (Synopsys)
-----

library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.std_logic_arith.all;

entity datapath_bfly23 is
    generic (width: integer :=16);
    port (
        ctrl: in STD_LOGIC_VECTOR (5 downto 0);
        rst: in STD_LOGIC;
        din1: in STD_LOGIC_VECTOR (width-1 downto 0);
        din2: in STD_LOGIC_VECTOR (width-1 downto 0);
        din3: in STD_LOGIC_VECTOR (width-1 downto 0);
        din4: in STD_LOGIC_VECTOR (width-1 downto 0);
        din5: in STD_LOGIC_VECTOR (width-1 downto 0);
        din6: in STD_LOGIC_VECTOR (width-1 downto 0);
        dout1: out STD_LOGIC_VECTOR (width-1 downto 0);
        dout2: out STD_LOGIC_VECTOR (width-1 downto 0));
end datapath_bfly23;

```

Appendices

architecture structure of datapath_bfly23 is

```

component dff_up      -- D-flip-flop register triggered by clock rising edge
  generic (width: integer :=16);
  port (clk, rst :in STD_LOGIC;
        din: in STD_LOGIC_VECTOR (width-1 downto 0);
        q: out STD_LOGIC_VECTOR (width-1 downto 0));
end component;

component mux2_1      -- 2-to-1 Multiplexer
  generic (width: integer :=16);
  port (din_1,din_2: in STD_LOGIC_VECTOR (width-1 downto 0);
        sel: in STD_LOGIC;
        dout: out STD_LOGIC_VECTOR (width-1 downto 0));
end component;

component adder      -- two's complement adder
  generic (width: integer :=16);
  port (din1: in STD_LOGIC_VECTOR (width-1 downto 0);
        din2: in STD_LOGIC_VECTOR (width-1 downto 0);
        dout: out STD_LOGIC_VECTOR (width-1 downto 0) );
end component;

component adder_sub  -- two's complement adder/subtractor
  generic (width: integer :=16);
  port (mode: in STD_LOGIC;
        din1: in STD_LOGIC_VECTOR (2*width-1 downto 0);
        din2: in STD_LOGIC_VECTOR (2*width-1 downto 0);
        dout: out STD_LOGIC_VECTOR (width-1 downto 0));
end component;

component sub      -- two's complement subtractor
  generic (width: integer :=16);
  port (din1: in STD_LOGIC_VECTOR (width-1 downto 0);
        din2: in STD_LOGIC_VECTOR (width-1 downto 0);
        dout: out STD_LOGIC_VECTOR (width-1 downto 0) );
end component;

component tsc_multiplier -- two's complement multiplier
  generic (width: integer:= 16);
  port (din1: in STD_LOGIC_VECTOR (width-1 downto 0);
        din2: in STD_LOGIC_VECTOR (width-1 downto 0);
        dout: out STD_LOGIC_VECTOR (2*width-1 downto 0));
end component;

component switch2    -- 2-to-2 switch
  generic (width: integer :=16);
  port (a, b: in STD_LOGIC_VECTOR (width-1 downto 0);
        sel: in STD_LOGIC;
        c, d: out STD_LOGIC_VECTOR (width-1 downto 0));
end component;

component dff_dn      -- D-flip-flop register triggered by clock falling edge
  generic (width: integer :=16);
  port (clk, rst: in STD_LOGIC;
        din: in STD_LOGIC_VECTOR (width-1 downto 0);
        q: out STD_LOGIC_VECTOR (width-1 downto 0));
end component;

```

Appendices

```

-- butterfly input signals and twiddle factors
signal br, bi, wr, wi, ar, ai: STD_LOGIC_VECTOR (width-1 downto 0);
-- other interconnection signals
signal wi_in, bi_in, br_in, wr_in: STD_LOGIC_VECTOR (width-1 downto 0);
signal bir, bri: STD_LOGIC_VECTOR (width-1 downto 0);
signal mult_1, mult_2, uv_1, uv_2: STD_LOGIC_VECTOR (2*width-1 downto 0);
signal uv, uv_in, ar_in, ai_in, a_in: STD_LOGIC_VECTOR (width-1 downto 0);
signal cri, dri: STD_LOGIC_VECTOR (width-1 downto 0);
-- butterfly output signals
signal c, d: STD_LOGIC_VECTOR (width-1 downto 0);

begin
  -- inputs assignments and scaling to prevent overflow
  ar<=din1(width-1)&din1(width-1 downto 1);
  ai<=din2(width-1)&din2(width-1 downto 1);
  br<=din3; bi<=din4; wr<=din5; wi<=din6;

  -- first pipeline stage
  d0: dff_up port map (clk=>ctrl(0), rst=>rst, din=>wi, q=>wi_in);
  d1: dff_up port map (clk=>ctrl(0), rst=>rst, din=>bi, q=>bi_in);
  d2: dff_up port map (clk=>ctrl(0), rst=>rst, din=>br, q=>br_in);
  d3: dff_up port map (clk=>ctrl(0), rst=>rst, din=>wr, q=>wr_in);
  sw1: switch2 port map (a=>bi_in, b=>br_in, sel=>ctrl(1), c=>bir, d=>bri);
  mult1: tsc_multiplier port map (din1=>wi_in, din2=>bir, dout=>mult_1);
  mult2: tsc_multiplier port map (din1=>bri, din2=>wr_in, dout=>mult_2);

  -- 2nd pipeline stage
  d4: dff_dn generic map (32) port map (clk=>ctrl(2), rst=>rst, din=>mult_1, q=>uv_1);
  d5: dff_dn generic map (32) port map (clk=>ctrl(2), rst=>rst, din=>mult_2, q=>uv_2);
  add1: adder_sub port map (mode=>ctrl(1), din1=>uv_2, din2=>uv_1, dout=>uv);

  -- 3rd pipeline stage
  d6: dff_dn port map (clk=>ctrl(3), rst=>rst, din=>uv, q=>uv_in);
  d7: dff_up port map (clk=>ctrl(4), rst=>rst, din=>ar, q=>ar_in);
  d8: dff_up port map (clk=>ctrl(4), rst=>rst, din=>ai, q=>ai_in);
  mux1: mux2_1 port map (din_1=>ar_in, din_2=>ai_in, sel=>ctrl(5), dout=>a_in);
  add2: adder port map (din1=>a_in, din2=>uv_in, dout=>cri);
  add3: sub port map (din1=>a_in, din2=>uv_in, dout=>dri);

  -- output stage
  d9: dff_up port map (clk=>ctrl(3), rst=>rst, din=>cri, q=>c);
  d10: dff_up port map (clk=>ctrl(3), rst=>rst, din=>dri, q=>d);

  --output assignments
  dout1<=c; dout2<=d;

end structure;

```

Appendices

Appendix B

Selected VHDL Codes for the Proposed CBT Processor

A. Structure Design of the Proposed CBT Processor Data Path

```

-----
-- Module:      datapath_cbt.vhd
-- Function :    the pipelined two-multiplier and four-adder CBT data path
--
-- Inputs:       c1, c2, c3, cw--control signals from the CBT controller
--               rst, rst1, rst3--reset signals
--               xn, xm, kern--data inputs and coefficients to the data path
-- Outputs:      cqr, cqi--real and imaginary parts of the CBT transformation
--
-- Institute:    ISR Lab-EEE2-NTU
-- Author:       Wang Chao
-- Date:         18/12/2003
-- Simulator:    ModelSim (Mentor Graphics)
-- Synthesis:    Design Compiler (Synopsys)
-----

```

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

```

```

entity datapath_cbt is
    generic (width: integer :=16);
    port ( c1, c2, c3, cw: in STD_LOGIC;
          rst, rst1, rst3: in STD_LOGIC;
          xn, xm: in STD_LOGIC_VECTOR (width-1 downto 0);
          kern: in STD_LOGIC_VECTOR (2*width-1 downto 0);
          cqr, cqi: out STD_LOGIC_VECTOR (width-1 downto 0));
end datapath_cbt;

```

```

architecture structure of datapath_cbt is
    component dff_up
        generic (width: integer :=16);
        port (clk, rst :in STD_LOGIC;
              din: in STD_LOGIC_VECTOR (width-1 downto 0);
              q: out STD_LOGIC_VECTOR (width-1 downto 0));
    end component;

```

```

    component adder
        generic (width: integer :=16);
        port ( din1: in STD_LOGIC_VECTOR (width-1 downto 0);
              din2: in STD_LOGIC_VECTOR (width-1 downto 0);
              dout: out STD_LOGIC_VECTOR (width-1 downto 0) );
    end component;

```

```

    component sub
        generic (width: integer :=16);

```

Appendices

```

        port ( din1: in STD_LOGIC_VECTOR (width-1 downto 0);
              din2: in STD_LOGIC_VECTOR (width-1 downto 0);
              dout: out STD_LOGIC_VECTOR (width-1 downto 0));
    end component;

    component tsc_multiplier
    generic (width: integer:= 16);
    port ( din1: in STD_LOGIC_VECTOR (width-1 downto 0);
          din2: in STD_LOGIC_VECTOR (width-1 downto 0);
          dout: out STD_LOGIC_VECTOR (2*width-1 downto 0));
    end component;

    signal xn_in, xm_in: STD_LOGIC_VECTOR (width-1 downto 0);
    signal x, y: STD_LOGIC_VECTOR (width-1 downto 0);
    signal rst1_in: STD_LOGIC;

    signal x_in, y_in: STD_LOGIC_VECTOR (width-1 downto 0);
    signal kerncos_in, kernsin_in: STD_LOGIC_VECTOR (width-1 downto 0);
    signal mult1, mult2: STD_LOGIC_VECTOR (2*width-1 downto 0);

    signal rst3_in: STD_LOGIC;
    signal mult1_in, mult2_in: STD_LOGIC_VECTOR (2*width-1 downto 0);
    signal cqtr, cqtr_in: STD_LOGIC_VECTOR (2*width-1 downto 0);
    signal cqti, cqti_in: STD_LOGIC_VECTOR (2*width-1 downto 0);

    begin
        -- First pipeline stage
        rst1_in <= rst OR rst1;
        d1: dff_up port map (clk=>c1, rst=>rst, din=>xn, q=>xn_in);
        d2: dff_up port map (clk=>c1, rst=>rst1_in, din=>xm, q=>xm_in);
        add_1: adder port map (din1=>xn_in, din2=>xm_in, dout=>x);
        add_2: sub port map (din1=>xn_in, din2=>xm_in, dout=>y);

        -- Second pipeline stage
        d3: dff_up port map (clk=>c2, rst=>rst, din=>kern(2*width-1 downto width), q=>kerncos_in);
        d4: dff_up port map (clk=>c2, rst=>rst, din=>x, q=>x_in);
        mult_1: tsc_multiplier port map (din1=>kerncos_in, din2=>x_in, dout=>mult1);
        d5: dff_up port map (clk=>c2, rst=>rst, din=>y, q=>y_in);
        d6: dff_up port map (clk=>c2, rst=>rst, din=>kern(width-1 downto 0), q=>kernsin_in);
        mult_2: tsc_multiplier port map (din1=>y_in, din2=>kernsin_in, dout=>mult2);

        -- Third pipeline stage
        rst3_in <= rst OR rst3;
        d7: dff_up generic map (32) port map (clk=>c3, rst=>rst, din=>mult1, q=>mult1_in);
        d8: dff_up generic map (32) port map (clk=>c3, rst=>rst3_in, din=>cqtr, q=>cqtr_in);
        add_3: adder generic map (32) port map (din1=>mult1_in, din2=>cqtr_in, dout=>cqtr);
        d9: dff_up generic map (32) port map (clk=>c3, rst=>rst3_in, din=>cqti, q=>cqti_in);
        d10: dff_up generic map (32) port map (clk=>c3, rst=>rst, din=>mult2, q=>mult2_in);
        add_4: adder generic map (32) port map (din1=>mult2_in, din2=>cqti_in, dout=>cqti);

        -- Output stage
        d11: dff_up port map (clk=>cw, rst=>rst, din=>cqtr(2*width-1 downto width), q=>cqr);
        d12: dff_up port map (clk=>cw, rst=>rst, din=>cqti(2*width-1 downto width), q=>cqi);

    end structure;

```

Appendices

B. Design of the Critical-Band Generator for the 21-band CBT Processor

```

-----
-- Module: cband_generator.vhd
-- Function: a critical band counter to keep track of which critical band is being computed and provide
--           dedicate information of each band for the address calculation in the address generator
--
-- Inputs :   rst--reset signal
--            phi (cband_done)--indicate that one critical band calculation is completed
-- Outputs :  lm--numbers of CBT butterflies to be calculated for each critical band
--            offset_xm, offset_x--offset for addressing x in a particular critical band
--            offset_kern--offset for addressing coefficient kern in a particular critical band
--            oddeven--parity of the windlength for a particular critical band
--            cbt_done--indicate that a CBT transformation is completed
--
-- Institute:  ISR Lab-EEE2-NTU
-- Author:    Wang Chao
-- Date:      18/12/2003
-- Simulator: ModelSim (Mentor Graphics)
-- Synthesis: Design Compiler (Synopsys)
-----

```

```
library ieee;
```

```
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
```

```
entity cband_generator is
    generic (ncp: integer:=22;
             addrcb_siz: integer:=5;
             ncb: integer:=21); -- the number of critical bands: 21
    port ( phi, clk:   in STD_LOGIC;
           rst, reseto: in STD_LOGIC;
           addr_cb:   in STD_LOGIC_VECTOR (addrcb_siz-1 downto 0); -- address from AGU
           lm:        out STD_LOGIC_VECTOR (6 downto 0);
           offset_x:  out STD_LOGIC_VECTOR (6 downto 0);
           offset_kern: out STD_LOGIC_VECTOR (9 downto 0);
           offset_xm: out STD_LOGIC_VECTOR (7 downto 0);
           cbt_done:  out STD_LOGIC;
           oddeven:   out STD_LOGIC
    );
end cband_generator;
```

Architecture behav of cband_generator is

```
signal cband_cnt: STD_LOGIC_VECTOR (4 downto 0);
signal decoder_out: STD_LOGIC_VECTOR (32 downto 0);
signal qn: STD_LOGIC_VECTOR (4 downto 0);
signal reset, rst_cnt, cbt_done_b, cbt_done_d: STD_LOGIC;
```

```
begin
```

```

-- cband decoder used to provide dedicated information for each 21 critical bands, including
-- the parity of windowlength, the number of CBT butterflies, the offsets for addressing, etc.
cband_decoder: process ( cband_cnt )
begin
CASE cband_cnt IS
    WHEN "00001" => decoder_out <= "110100001010000100000000000000000000";

```

Appendices

```

        WHEN "00010" => decoder_out <= "110100001010000100000000001010001";
        WHEN "00011" => decoder_out <= "110100001010000100000000010100010";
        WHEN "00100" => decoder_out <= "110100001010000100000000011100111";
        WHEN "00101" => decoder_out <= "1101000010100001000000000101000100";
        WHEN "00110" => decoder_out <= "010011101001111100000010110010101";
        WHEN "00111" => decoder_out <= "010000101001001100011010111100100";
        WHEN "01000" => decoder_out <= "001110001000100100101111000100111";
        WHEN "01001" => decoder_out <= "001011111000000001000001001100000";
        WHEN "01010" => decoder_out <= "001010000111100101001111010010000";
        WHEN "01011" => decoder_out <= "001000100111001101011011010111001";
        WHEN "01100" => decoder_out <= "100111010110110101100101011011100";
        WHEN "01101" => decoder_out <= "000110000110100101101111011111010";
        WHEN "01110" => decoder_out <= "100101010110010101110101100010011";
        WHEN "01111" => decoder_out <= "000100010110001001111101100101001";
        WHEN "10000" => decoder_out <= "100011110110000010000011100111011";
        WHEN "10001" => decoder_out <= "000011000101110110000111101001011";
        WHEN "10010" => decoder_out <= "000010100101101110001011101011000";
        WHEN "10011" => decoder_out <= "100010010101100110001101101100011";
        WHEN "10100" => decoder_out <= "000001110101100010010001101101101";
        WHEN "10101" => decoder_out <= "000001100101011110010011101110101";
        WHEN OTHERS => decoder_out <= (OTHERS => 'Z');
    END CASE;
end process cband_decoder ;

-- assign the dedicated information for each critical band
oddeven <= decoder_out ( 32 );
lm <= decoder_out ( 31 downto 25 );
offset_xm <= decoder_out ( 24 downto 17 );
offset_x <= decoder_out ( 16 downto 10 );
offset_kern <= decoder_out ( 9 downto 0 );

cb_counter: process(phi, reset)    -- critical-band counter incremented by phi (cbt_done)
begin
    if reset='1' then
        qn<=(others=>'0'); -- reset counter
    elsif phi'event and phi='1' then
        qn<=qn+1;
    end if;
end process cb_counter;
cband_cnt<=qn;

-- auto reset generation
rst_cnt<='1' when qn=ncp else '0';
cbt_done_b<=phi when addr_cb=ncb else '0';
reset<= rst OR rst_cnt;

delay_cbtdone: process(clk, reseto) -- delay the cbt_done signal for one cycle
begin
    if (reseto='1') then
        cbt_done_d<='0';
    elsif (clk'event and clk='1') then
        cbt_done_d<=cbt_done_b;
    end if;
end process delay_cbtdone;
cbt_done<=cbt_done_d;

end behavior;
```


Appendices

Appendix C

Selected VHDL Codes for the Proposed DWPT Processors

A. Structure Design of the Direct Implementation of the Lifting-Based D4

Wavelet Filter

```

-----
-- Module:      D4b_PE1.vhd
-- Function :    The direct implementation of the lifting-based D4 wavelet filter
--              (Second lifting implementation of the D4 filter as presented in the Swedlens' paper)
--
-- Inputs:       S_IN, D_IN--the even- and odd-indexed input data elements
--              RSTN--reset signal
--              CLK--system clock
-- Outputs:      S_OUT, D_OUT--the low-pass and high-pass coefficients, respectively
--              I/O data 16-bit, internal data 18-bit
--
-- Institute:    DSP Lab-EEE3-NTU
-- Author:       Wang Chao
-- Date:         16/05/2005
-- Simulator:    ModelSim (Mentor Graphics)
-- Synthesis:    Design Compiler (Synopsys)
-----

```

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

```

```

entity D4b_PE1 is
    generic (width: integer :=16);
    port (RSTN: in STD_LOGIC;
          rstn1, rstn2: in STD_LOGIC; --for zero-padding test
          byp_11, byp_12, byp_13, byp_14: in STD_LOGIC; -- for bypass operation mode
          CLK: in STD_LOGIC;
          S_IN: in STD_LOGIC_VECTOR (width-1 downto 0);
          D_IN: in STD_LOGIC_VECTOR (width-1 downto 0);
          S_OUT: out STD_LOGIC_VECTOR (width-1 downto 0);
          D_OUT: out STD_LOGIC_VECTOR (width-1 downto 0));
end D4b_PE1;

```

architecture struct of D4b_PE1 is

```

    component adder -- two's complement adder with bypass operation mode
    generic (width: integer :=18);
    port (DIN_1: in STD_LOGIC_VECTOR (width-1 downto 0);
          DIN_2: in STD_LOGIC_VECTOR (width-1 downto 0);
          byp_mode: in STD_LOGIC;
          SUM: out STD_LOGIC_VECTOR (width-1 downto 0));
    end component;

```

```

    component sub -- two's complement subtractor with bypass operation mode
    generic (width: integer :=18);

```

Appendices

```

        port (D_MIN: in STD_LOGIC_VECTOR (width-1 downto 0);
              D_SUB: in STD_LOGIC_VECTOR (width-1 downto 0);
              byp_mode: in STD_LOGIC;
              DIFF: out STD_LOGIC_VECTOR (width-1 downto 0));
    end component;

    component rshift -- 1-bit right shifter with bypass operation mode
    generic (width: integer :=18);
    port (DIN: in STD_LOGIC_VECTOR (width-1 downto 0);
          byp_mode: in STD_LOGIC;
          DOUT: out STD_LOGIC_VECTOR (width-1 downto 0));
    end component;

    component SR1 -- one-cycle delay register
    generic (width: integer :=18);
    port (CLK: in STD_LOGIC;
          RSTN: in STD_LOGIC;
          DIN: in STD_LOGIC_VECTOR (width-1 downto 0);
          DOUT: out STD_LOGIC_VECTOR (width-1 downto 0));
    end component;

    component multiplier18x18 -- two's complement multiplier with bypass operation mode
    generic (width: integer:= 18);
    port (din1: in STD_LOGIC_VECTOR (width-1 downto 0);
          din2: in STD_LOGIC_VECTOR (width-1 downto 0);
          byp_mode: in STD_LOGIC;
          dout: out STD_LOGIC_VECTOR (width-1 downto 0) );
    end component;

    signal rstn_s, rstn_d: STD_LOGIC;
    -- the coefficients for the lifting-based D4 wavelet filter
    signal alpha, beta, gama, kesai, kesai_inv: STD_LOGIC_VECTOR (width+1 downto 0);

    signal sin_ext, din_ext: STD_LOGIC_VECTOR (width+1 downto 0);
    signal sin_rsh, prod_1, s1: STD_LOGIC_VECTOR (width+1 downto 0);
    signal s1_pip, din_rsh_pip: STD_LOGIC_VECTOR (width+1 downto 0);

    signal din_rsh, prod_2, d1_tmp: STD_LOGIC_VECTOR (width+1 downto 0);
    signal s1_dly, prod_3, d1_new: STD_LOGIC_VECTOR (width+1 downto 0);
    signal s1_dly_pip, d1_tmp_pip: STD_LOGIC_VECTOR (width+1 downto 0);
    signal s1_dly_pip2, d1_new_pip: STD_LOGIC_VECTOR (width+1 downto 0);

    signal s2, d1: STD_LOGIC_VECTOR (width+1 downto 0);
    signal prod_4, prod_5: STD_LOGIC_VECTOR (width+1 downto 0);

    begin
        -- Input 16-bit=>18-bit by zero-padding
        sin_ext <= S_IN(width-1 downto 0)& "00";
        din_ext <= D_IN(width-1 downto 0)& "00";

        -- 1st updating: s1
        rshift1: rshift port map (DIN=>sin_ext, byp_mode=>byp_11, DOUT=>sin_rsh);
        alpha<="011011101101100111";
        mult1: multiplier18x18 port map (din1=>alpha, din2=>din_ext, byp_mode=>byp_11,
                                         dout=>prod_1);
        add1: adder port map (DIN_1=>prod_1, DIN_2=>sin_rsh, byp_mode=>byp_11, SUM=>s1);
        ----- 1st pipeline stage
        r_1: SR1 port map (CLK=>CLK, RSTN=>RSTN, DIN=>s1, DOUT=>s1_pip);
    
```

Appendices

```

r_2: SR1 port map (CLK=>CLK, RSTN=>RSTN, DIN=>din_rsh, DOUT=>din_rsh_pip);

-- 1st predicting: d1_new=d1_tmp+prod3
rshift2: rshift port map (DIN=>din_ext, byp_mode=>byp_11, DOUT=>din_rsh);
beta<="110010001001001100";
mult2: multiplier18x18 port map (din1=>beta, din2=>s1_pip, byp_mode=>byp_12,
                                dout=>prod_2);
add2: adder port map (DIN_1=>prod_2, DIN_2=>din_rsh_pip, byp_mode=>byp_12,
                     SUM=>d1_tmp);
----- 2nd pipeline stage
r_3: SR1 port map (CLK=>CLK, RSTN=>RSTN, DIN=>s1_dly, DOUT=>s1_dly_pip);
r_4: SR1 port map (CLK=>CLK, RSTN=>RSTN, DIN=>d1_tmp, DOUT=>d1_tmp_pip);

rstn_s<=RSTN and rstn1;
-- unit delay register
sr_1: SR1 port map (CLK=>CLK, RSTN=>rstn_s, DIN=>s1_pip, DOUT=>s1_dly);
gama<="000010001001001100";
mult3: multiplier18x18 port map (din1=>gama, din2=>s1_dly_pip, byp_mode=>byp_13,
                                dout=>prod_3);
add3: adder port map (DIN_1=>prod_3, DIN_2=>d1_tmp_pip, byp_mode=>byp_13,
                     SUM=>d1_new);
----- 3rd pipeline stage
r_5: SR1 port map (CLK=>CLK, RSTN=>RSTN, DIN=>s1_dly_pip, DOUT=>s1_dly_pip2);
rstn_d<=RSTN and rstn2;
r_6: SR1 port map (CLK=>CLK, RSTN=>rstn_d, DIN=>d1_new, DOUT=>d1_new_pip);

-- 2nd updating: s2
sub1: sub port map (D_MIN=>s1_dly_pip2, D_SUB=>d1_new_pip, byp_mode=>byp_14,
                   DIFF=>s2);
-- unit delay register
sr_2: SR1 port map (CLK=>CLK, RSTN=>RSTN, DIN=>d1_new_pip, DOUT=>d1);

-- Scaling
kesai<="001000010010000011";
kesai_inv<="011110111010001101";
mult4: multiplier18x18 port map (din1=>kesai, din2=>s2, byp_mode=>byp_14,
                                dout=>prod_4);
mult5: multiplier18x18 port map (din1=>kesai_inv, din2=>d1, byp_mode=>byp_14,
                                dout=>prod_5);

-- Output 18-bit=>16-bit by truncating the last 2 LSB
S_OUT <= prod_4(width+1 downto 2);
D_OUT <= prod_5(width+1 downto 2);

end struct;

```

B. Structure Design of the Folded PE for the Lifting-Based D4 Wavelet

Filter at the 2nd Level in the Proposed DDC Architecture

```

-- Module:      D4b_PE2.vhd
-- Function :   The folded PE of the lifting-based D4 wavelet filter at the 2nd level in the DDC
--              architecture for the 3-level D4 DWPT
--              (Second lifting implementation of the D4 filter as presented in the Swedlens' paper)
--

```

Appendices

```
-- Inputs:      S_IN, D_IN--the even- and odd-indexed input data elements to the wavelet filter
--              RSTN--reset signal
--              CLK--system clock
-- Outputs:     S_OUT, D_OUT--the low-pass and high-pass coefficients, respectively
--              I/O data 16-bit, internal data 18-bit
--
-- Institute:    DSP Lab-EEE3-NTU
-- Author:      Wang Chao
-- Date:        29/05/2005
-- Simulator:   ModelSim (Mentor Graphics)
-- Synthesis:   Design Compiler (Synopsys)
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
```

```
entity D4b_PE2 is
  generic (width: integer :=16);
  port (RSTN: in STD_LOGIC;
        rstn1, rstn2: in STD_LOGIC; --for zero-padding test
        byp_21, byp_22, byp_23, byp_24: in STD_LOGIC; -- for bypass operation mode
        CLK: in STD_LOGIC;
        S_IN: in STD_LOGIC_VECTOR (width-1 downto 0);
        D_IN: in STD_LOGIC_VECTOR (width-1 downto 0);
        S_OUT: out STD_LOGIC_VECTOR (width-1 downto 0);
        D_OUT: out STD_LOGIC_VECTOR (width-1 downto 0));
end D4b_PE2;
```

architecture struct of D4b_PE2 is

```
  component adder -- two's complement adder with bypass operation mode
    generic (width: integer :=18);
    port (DIN_1: in STD_LOGIC_VECTOR (width-1 downto 0);
          DIN_2: in STD_LOGIC_VECTOR (width-1 downto 0);
          byp_mode: in STD_LOGIC;
          SUM: out STD_LOGIC_VECTOR (width-1 downto 0));
  end component;

  component sub -- two's complement subtractor with bypass operation mode
    generic (width: integer :=18);
    port (D_MIN: in STD_LOGIC_VECTOR (width-1 downto 0);
          D_SUB: in STD_LOGIC_VECTOR (width-1 downto 0);
          byp_mode: in STD_LOGIC;
          DIFF: out STD_LOGIC_VECTOR (width-1 downto 0));
  end component;

  component SR1 -- one-cycle delay register
    generic (width: integer :=18);
    port (CLK: in STD_LOGIC;
          RSTN: in STD_LOGIC;
          DIN: in STD_LOGIC_VECTOR (width-1 downto 0);
          DOUT: out STD_LOGIC_VECTOR (width-1 downto 0));
  end component;

  component SR2 -- two-cycle delay register
    generic (width: integer :=18);
    port (CLK: in STD_LOGIC;
```

Appendices

```

        RSTN: in STD_LOGIC;
        DIN: in STD_LOGIC_VECTOR (width-1 downto 0);
        DOUT: out STD_LOGIC_VECTOR (width-1 downto 0));
    end component;

    component SR2_BYP -- two-cycle delay register with bypass operation mode
        generic (width: integer :=18);
        port(CLK: in STD_LOGIC;
            RSTN: in STD_LOGIC;
            byp_mode: in STD_LOGIC;
            DIN: in STD_LOGIC_VECTOR (width-1 downto 0);
            DOUT: out STD_LOGIC_VECTOR (width-1 downto 0));
    end component;

    component multiplier18x18 -- two's complement multiplier with bypass operation mode
        generic (width: integer:= 18);
        port (din1: in STD_LOGIC_VECTOR (width-1 downto 0);
            din2: in STD_LOGIC_VECTOR (width-1 downto 0);
            byp_mode: in STD_LOGIC;
            dout: out STD_LOGIC_VECTOR (width-1 downto 0) );
    end component;

    signal rstn_s, rstn_d: STD_LOGIC;
    signal alpha, beta, gama, kesai, kesai_inv: STD_LOGIC_VECTOR (width+1 downto 0);

    signal sin_ext, din_ext: STD_LOGIC_VECTOR (width+1 downto 0);
    signal sin_rsh, prod_1, s1: STD_LOGIC_VECTOR (width+1 downto 0);
    signal s1_pip, din_rsh_pip: STD_LOGIC_VECTOR (width+1 downto 0);

    signal din_rsh, prod_2, d1_tmp: STD_LOGIC_VECTOR (width+1 downto 0);
    signal s1_dly, prod_3, d1_new: STD_LOGIC_VECTOR (width+1 downto 0);
    signal s1_dly_pip, d1_tmp_pip: STD_LOGIC_VECTOR (width+1 downto 0);
    signal s1_dly_pip2, d1_new_pip: STD_LOGIC_VECTOR (width+1 downto 0);

    signal s2, d1: STD_LOGIC_VECTOR (width+1 downto 0);
    signal prod_4, prod_5, s1_byp: STD_LOGIC_VECTOR (width+1 downto 0);

begin
    -- Input 16-bit=>18-bit by zero-padding
    sin_ext <= S_IN(width-1 downto 0)& "00";
    din_ext <= D_IN(width-1 downto 0)& "00";

    -- 1st updating: s1
    rshift1: rshift port map (DIN=>sin_ext, byp_mode=>byp_21, DOUT=>sin_rsh);
    alpha<="011011101101100111";
    mult1: multiplier18x18 port map (din1=>alpha, din2=>din_ext, byp_mode=>byp_21,
        dout=>prod_1);
    add1: adder port map (DIN_1=>prod_1, DIN_2=>sin_rsh, byp_mode=>byp_21,SUM=>s1);
    ----- 1st pipeline stage
    r_1: SR1 port map (CLK=>CLK, RSTN=>RSTN, DIN=>s1, DOUT=>s1_pip);
    r_2: SR1 port map (CLK=>CLK, RSTN=>RSTN, DIN=>din_rsh, DOUT=>din_rsh_pip);

    -- 1st predicting: d1_new=d1_tmp+prod3
    rshift2: rshift port map (DIN=>din_ext, byp_mode=>byp_21, DOUT=>din_rsh);
    beta<="110010001001001100";
    mult2: multiplier18x18 port map (din1=>beta, din2=>s1_pip, byp_mode=>byp_22,
        dout=>prod_2);
    add2: adder port map (DIN_1=>prod_2, DIN_2=>din_rsh_pip, byp_mode=>byp_22,

```

Appendices

```

SUM=>d1_tmp);
----- 2nd pipeline stage
r_3: SR1 port map (CLK=>CLK, RSTN=>RSTN, DIN=>s1_dly, DOUT=>s1_dly_pip);
r_4: SR1 port map (CLK=>CLK, RSTN=>RSTN, DIN=>d1_tmp, DOUT=>d1_tmp_pip);

rstn_s<=RSTN and rstn1;
-- the folded two-unit delay registers
sr_1: SR2_BYP port map (CLK=>CLK, RSTN=>rstn_s, DIN=>s1_pip, byp_mode=>byp_22,
DOUT=>s1_dly);
gama<="000010001001001100";
mult3: multiplier18x18 port map (din1=>gama, din2=>s1_dly_pip, byp_mode=>byp_23,
dout=>prod_3);
add3: adder port map (DIN_1=>prod_3, DIN_2=>d1_tmp_pip, byp_mode=>byp_23,
SUM=>d1_new);
----- 3rd pipeline stage
r_5: SR1 port map (CLK=>CLK, RSTN=>RSTN, DIN=>s1_dly_pip, DOUT=>s1_dly_pip2);
rstn_d<=RSTN and rstn2;
r_6: SR1 port map (CLK=>CLK, RSTN=>rstn_d, DIN=>d1_new, DOUT=>d1_new_pip);

-- 2nd updating: s2
sub1: sub port map (D_MIN=>s1_dly_pip2, D_SUB=>d1_new_pip, byp_mode=>byp_24,
DIFF=>s2);
-- the folded two-unit delay registers
sr_2: SR2 port map (CLK=>CLK, RSTN=>RSTN, DIN=>d1_new_pip, DOUT=>d1);
sr_3: SR2 port map (CLK=>CLK, RSTN=>RSTN, DIN=>s1_dly_pip2, DOUT=>s1_byp);

-- Scaling
kesai<="001000010010000011";
kesai_inv<="011110111010001101";
mult4: multiplier18x18 port map (din1=>kesai, din2=>s2, byp_mode=>byp_24,
dout=>prod_4);
mult5: multiplier18x18 port map (din1=>kesai_inv, din2=>d1, byp_mode=>byp_24,
dout=>prod_5);

-- Output 18-bit=>16-bit by truncating the last 2 LSB
S_OUT <= prod_4(width+1 downto 2) when byp_24='0' else
s1_byp(width+1 downto 2);
D_OUT <= prod_5(width+1 downto 2);

end struct;

```

C. Structure Design of the Proposed DDC Architecture for 3-level D4

DWPT

```

-- Module:      datapath_daub4.vhd
-- Function :    data path of the proposed DDC architecture for computing 3-level D4 DWPT
--              (Second lifting implementation of the D4 filter as presented in the Swedlens' paper)
--
-- Inputs:       D_IN--the input data elements to the 3-level DWPT processor
--              RSTN--reset signal
--              CLK--the data clock
--              CLK2 (SEL1)--the data path clock; SEL2, SEL3--the switch control signals
-- Outputs:      S_OUT, D_OUT--the final DWPT coefficients
--              I/O data 16-bit, internal data 18-bit
--

```

Appendices

```
-- Institute:    DSP Lab-EEE3-NTU
-- Author:      Wang Chao
-- Date:        29/05/2005
-- Simulator:    ModelSim (Mentor Graphics)
-- Synthesis:    Design Compiler (Synopsys)
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
```

```
entity datapath_daub4 is
  generic (width: integer :=16); --default value would be 16-bit
  port (RSTN: in STD_LOGIC;
        rstn1, rstn2, rstn3: in STD_LOGIC; --for zero-padding test
        rstn4, rstn5, rstn6: in STD_LOGIC; --for zero-padding test
        byp_11, byp_12, byp_13, byp_14: in STD_LOGIC;
        byp_21, byp_22, byp_23, byp_24: in STD_LOGIC;
        byp_31, byp_32, byp_33, byp_34: in STD_LOGIC;
        CLK: in STD_LOGIC; --DATA CLOCK
        CLK2: in STD_LOGIC; --DATAPATH CLOCK and SEL1
        SEL2, SEL3: in STD_LOGIC;
        DATA_IN: in STD_LOGIC_VECTOR (width-1 downto 0);
        S_OUT: out STD_LOGIC_VECTOR (width-1 downto 0);
        D_OUT: out STD_LOGIC_VECTOR (width-1 downto 0));
end datapath_daub4;
```

architecture struct of datapath_daub4 is

```
  component SR1 -- one-cycle delay register
    generic (width: integer :=18);
    port (CLK: in STD_LOGIC;
          RSTN: in STD_LOGIC;
          DIN: in STD_LOGIC_VECTOR (width-1 downto 0);
          DOUT: out STD_LOGIC_VECTOR (width-1 downto 0));
  end component;

  component SR2 -- two-cycle delay register
    generic (width: integer :=18);
    port (CLK: in STD_LOGIC;
          RSTN: in STD_LOGIC;
          DIN: in STD_LOGIC_VECTOR (width-1 downto 0);
          DOUT: out STD_LOGIC_VECTOR (width-1 downto 0));
  end component;

  component switch2 -- 2-to-2 switch
    generic (width: integer :=16);
    port (DIN_1, DIN_2: in STD_LOGIC_VECTOR (width-1 downto 0);
          SEL: in STD_LOGIC;
          DOUT_1, DOUT_2: out STD_LOGIC_VECTOR (width-1 downto 0));
  end component;

  component D4b_PE1 is -- the PE at the 1st level in the DDC architecture for 3-level DWPT
    generic (width: integer :=16);
    port (RSTN: in STD_LOGIC;
          rstn1, rstn2: in STD_LOGIC; --for zero-padding test
          byp_11, byp_12, byp_13, byp_14: in STD_LOGIC;
          CLK: in STD_LOGIC;
          S_IN: in STD_LOGIC_VECTOR (width-1 downto 0);
```

Appendices

```

        D_IN: in STD_LOGIC_VECTOR (width-1 downto 0);
        S_OUT: out STD_LOGIC_VECTOR (width-1 downto 0);
        D_OUT: out STD_LOGIC_VECTOR (width-1 downto 0));
    end component;

    component D4b_PE2 is -- the PE at the 2nd level in the DDC architecture for 3-level DWPT
    generic (width: integer :=16);
    port (RSTN: in STD_LOGIC;
          rstn1, rstn2: in STD_LOGIC; --for zero-padding test
          byp_21, byp_22, byp_23, byp_24: in STD_LOGIC;
          CLK: in STD_LOGIC;
          S_IN: in STD_LOGIC_VECTOR (width-1 downto 0);
          D_IN: in STD_LOGIC_VECTOR (width-1 downto 0);
          S_OUT: out STD_LOGIC_VECTOR (width-1 downto 0);
          D_OUT: out STD_LOGIC_VECTOR (width-1 downto 0));
    end component;

    component D4b_PE3 -- the PE at the 3rd level in the DDC architecture for 3-level DWPT
    generic (width: integer :=16);--default value would be 16-bit
    port (RSTN: in STD_LOGIC;
          rstn1, rstn2: in STD_LOGIC; --for zero-padding test
          byp_31, byp_32, byp_33, byp_34: in STD_LOGIC;
          CLK: in STD_LOGIC;
          S_IN: in STD_LOGIC_VECTOR (width-1 downto 0);
          D_IN: in STD_LOGIC_VECTOR (width-1 downto 0);
          S_OUT: out STD_LOGIC_VECTOR (width-1 downto 0);
          D_OUT: out STD_LOGIC_VECTOR (width-1 downto 0));
    end component;

    signal DATA_IN_DLY1, DATA_IN_DLY2 : STD_LOGIC_VECTOR (width-1 downto 0);
    signal S_IN, D_IN: STD_LOGIC_VECTOR (width-1 downto 0);
    signal s1, d1, d1_d, s1_F, d1_I, s1_H: STD_LOGIC_VECTOR (width-1 downto 0);
    signal s2, d2, d2_d, s2_J, d2_N, s2_L: STD_LOGIC_VECTOR (width-1 downto 0);
    signal s3, d3: STD_LOGIC_VECTOR (width-1 downto 0);

    begin

        --serial to parallel circuits
        IN_DLY1: SR1 generic map (16) port map (CLK=>CLK, RSTN=>RSTN, DIN=>DATA_IN,
                                                DOUT=>DATA_IN_DLY1);
        IN_DLY2:SR1generic map (16) port map (CLK=>CLK, RSTN=>RSTN,
                                                DIN=>DATA_IN_DLY1, DOUT=>DATA_IN_DLY2);
        S2P_1: SR1 generic map (16) port map (CLK=>CLK2, RSTN=>RSTN,
                                                DIN=>DATA_IN_DLY2, DOUT=>S_IN);
        S2P_2: SR1 generic map (16) port map (CLK=>CLK2, RSTN=>RSTN,
                                                DIN=>DATA_IN_DLY1, DOUT=>D_IN);

        -- first pipeline stage
        Daub4_1: D4b_PE1 port map (RSTN=>RSTN, CLK=>CLK2, S_IN=>S_IN, D_IN=>D_IN,
                                   S_OUT=>s1, D_OUT=>d1, rstn1=>rstn1, rstn2=>rstn2, byp_11=>byp_11,
                                   byp_12=>byp_12, byp_13=>byp_13, byp_14=>byp_14);

        -- 2nd pipeline stage
        SR_2: SR1 generic map (16) port map (CLK=>CLK2, RSTN=>RSTN, DIN=>d1,
                                                DOUT=>d1_d);
        SW_2: switch2 port map (DIN_1=>s1, DIN_2=>d1_d, SEL=>SEL2, DOUT_1=>s1_F,
                                DOUT_2=>d1_I);
        SR_3: SR1 generic map (16) port map (CLK=>CLK2, RSTN=>RSTN, DIN=>s1_F,
                                                DOUT=>s1_H);
    
```


Appendices

```

Daub4_2: D4b_PE2 port map (RSTN=>RSTN, CLK=>CLK2, S_IN=>s1_H, D_IN=>d1_I,
                           S_OUT=>s2, D_OUT=>d2, rstn1=>rstn3, rstn2=>rstn4, byp_21=>byp_21,
                           byp_22=>byp_22, byp_23=>byp_23, byp_24=>byp_24);

-- 3rd pipeline stage
SR_4: SR2 generic map (16) port map (CLK=>CLK2, RSTN=>RSTN, DIN=>d2,
                                     DOUT=>d2_d);
SW_3: switch2 port map (DIN_1=>s2, DIN_2=>d2_d, SEL=>SEL3, DOUT_1=>s2_J,
                       DOUT_2=>d2_N);
SR_5: SR2 generic map (16) port map (CLK=>CLK2, RSTN=>RSTN, DIN=>s2_J,
                                     DOUT=>s2_L);
Daub4_3: D4b_PE3 port map (RSTN=>RSTN, CLK=>CLK2, S_IN=>s2_L, D_IN=>d2_N,
                           S_OUT=>s3, D_OUT=>d3, rstn1=>rstn5, rstn2=>rstn6, byp_31=>byp_31,
                           byp_32=>byp_32, byp_33=>byp_33, byp_34=>byp_34);

-- Output Assignments
S_OUT <= s3;
D_OUT <= d3;

end struct;

```

D. Structure Design of the Configurable PE in the Proposed Folded DDC

Architecture for 3-level D4 DWPT

```

-- Module:      D4b_PE3_Folded_DDC.vhd
-- Function :    PE unit to calculate three levels of DWPT in the folded DDC architecture
--              for computing 3-level D4 DWPT
--              (Second lifting implementation of the D4 filter as presented in the Swedlens' paper)
--
-- Inputs:       S_IN, D_IN--the even- and odd-indexed input data elements to the wavelet filter
--              RSTN--reset signal; CLK--system clock
--              sel1, sel2 -- switch control signals to configure the PE for different level of DWPT
-- Outputs:      S_OUT, D_OUT--the low-pass and high-pass coefficients, respectively
--              I/O data 16-bit, internal data 18-bit
--
-- Institute:    DSP Lab-EEE3-NTU
-- Author:       Wang Chao
-- Date:         16/04/2006
-- Simulator:    ModelSim (Mentor Graphics)
-- Synthesis:    Design Compiler (Synopsys)

```

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

```

```

entity D4b_PE3_Folded_DDC is
  generic (width: integer :=16);
  port (  RSTN: in STD_LOGIC;
         rstn1, rstn2: in STD_LOGIC; --for zero-padding test
         byp_31, byp_32, byp_33, byp_34: in STD_LOGIC;
         sel1, sel2: in STD_LOGIC_VECTOR (1 downto 0);
         CLK: in STD_LOGIC;
         S_IN: in STD_LOGIC_VECTOR (width-1 downto 0);
         D_IN: in STD_LOGIC_VECTOR (width-1 downto 0);

```

Appendices

```

        S_OUT: out STD_LOGIC_VECTOR (width-1 downto 0);
        D_OUT: out STD_LOGIC_VECTOR (width-1 downto 0)
    );
end D4b_PE3;

architecture struct of D4b_PE3_Folded_DDC is

    component adder -- two's complement adder with bypass operation mode
        generic (width: integer :=18);
        port (DIN_1: in STD_LOGIC_VECTOR (width-1 downto 0);
              DIN_2: in STD_LOGIC_VECTOR (width-1 downto 0);
              byp_mode: in STD_LOGIC;
              SUM: out STD_LOGIC_VECTOR (width-1 downto 0));
    end component;

    component sub -- two's complement subtractor with bypass operation mode
        generic (width: integer :=18);
        port (D_MIN: in STD_LOGIC_VECTOR (width-1 downto 0);
              D_SUB: in STD_LOGIC_VECTOR (width-1 downto 0);
              byp_mode: in STD_LOGIC;
              DIFF: out STD_LOGIC_VECTOR (width-1 downto 0));
    end component;

    component rshift -- 1-bit right shifter with bypass operation mode
        generic (width: integer :=18);
        port (DIN: in STD_LOGIC_VECTOR (width-1 downto 0);
              byp_mode: in STD_LOGIC;
              DOUT: out STD_LOGIC_VECTOR (width-1 downto 0));
    end component;

    component SR1 -- one-cycle delay register
        generic (width: integer :=18);
        port (CLK: in STD_LOGIC;
              RSTN: in STD_LOGIC;
              DIN: in STD_LOGIC_VECTOR (width-1 downto 0);
              DOUT: out STD_LOGIC_VECTOR (width-1 downto 0));
    end component;

    component SR1_BYP -- one-cycle delay register with bypass operation mode
        generic (width: integer :=18);
        port (CLK: in STD_LOGIC;
              RSTN: in STD_LOGIC;
              byp_mode: in STD_LOGIC;
              DIN: in STD_LOGIC_VECTOR (width-1 downto 0);
              DOUT: out STD_LOGIC_VECTOR (width-1 downto 0));
    end component;

    component SR2 -- two-cycle delay register
        generic (width: integer :=18);
        port (CLK: in STD_LOGIC;
              RSTN: in STD_LOGIC;
              DIN: in STD_LOGIC_VECTOR (width-1 downto 0);
              DOUT: out STD_LOGIC_VECTOR (width-1 downto 0));
    end component;

    component SR2_BYP -- two-cycle delay register with bypass operation mode
        generic (width: integer :=18);
        port (CLK: in STD_LOGIC;
              RSTN: in STD_LOGIC;

```

Appendices

```

        byp_mode: in STD_LOGIC;
        DIN: in STD_LOGIC_VECTOR (width-1 downto 0);
        DOUT: out STD_LOGIC_VECTOR (width-1 downto 0));
    end component;

    component multiplier18x18 -- two's complement multiplier with bypass operation mode
        generic (width: integer:= 18);
        port (din1: in STD_LOGIC_VECTOR (width-1 downto 0);
              din2: in STD_LOGIC_VECTOR (width-1 downto 0);
              byp_mode: in STD_LOGIC;
              dout: out STD_LOGIC_VECTOR (width-1 downto 0) );
    end component;

    signal rstn_s, rstn_d: STD_LOGIC;
    signal alpha, beta, gama, kesai, kesai_inv: STD_LOGIC_VECTOR (width+1 downto 0);
    signal sin_ext, din_ext: STD_LOGIC_VECTOR (width+1 downto 0);
    signal sin_rsh, prod_1, s1: STD_LOGIC_VECTOR (width+1 downto 0);
    signal s1_pip, din_rsh_pip: STD_LOGIC_VECTOR (width+1 downto 0);
    signal din_rsh, prod_2, d1_tmp: STD_LOGIC_VECTOR (width+1 downto 0);

    --first group of folded shift register outputs
    signal s1_dly, s1_dly1, s1_dly2, s1_dly3: STD_LOGIC_VECTOR (width+1 downto 0);
    signal prod_3, d1_new: std_logic_vector (STD_LOGIC_VECTOR downto 0);
    signal s1_dly_pip, d1_tmp_pip: std_logic_vector (STD_LOGIC_VECTOR downto 0);
    signal s1_dly_pip2, d1_new_pip: std_logic_vector (STD_LOGIC_VECTOR downto 0);
    signal s2: std_logic_vector (STD_LOGIC_VECTOR downto 0);

    --second group of folded shift register outputs
    signal d1, d1_1, d1_2, d1_3: STD_LOGIC_VECTOR (width+1 downto 0);
    signal prod_4, prod_5: STD_LOGIC_VECTOR (width+1 downto 0);
    signal s1_byp, s1_byp2, s1_byp3: STD_LOGIC_VECTOR (width+1 downto 0);

    begin
        -- Input 16-bit=>18-bit by zero-padding
        sin_ext <= S_IN(width-1 downto 0)& "00";
        din_ext <= D_IN(width-1 downto 0)& "00";

        -- 1st Updating: s1
        rshift1: rshift port map (DIN=>sin_ext, byp_mode=>byp_31, DOUT=>sin_rsh);
        alpha<="011011101101100111";
        mult1: multiplier18x18 port map (din1=>alpha, din2=>din_ext, byp_mode=>byp_31,
                                         dout=>prod_1);
        add1: adder port map (DIN_1=>prod_1,DIN_2=>sin_rsh, byp_mode=>byp_31, SUM=>s1);
        -- 1st pipeline stage
        r_1: SR1 port map (CLK=>CLK, RSTN=>RSTN, DIN=>s1, DOUT=>s1_pip);
        r_2: SR1 port map (CLK=>CLK, RSTN=>RSTN, DIN=>din_rsh, DOUT=>din_rsh_pip);

        -- 1st Predicting: d1_new=d1_tmp+prod3
        rshift2: rshift port map (DIN=>din_ext, byp_mode=>byp_31, DOUT=>din_rsh);
        beta<="110010001001001100";
        mult2: multiplier18x18 port map (din1=>beta, din2=>s1_pip,byp_mode=>byp_32,
                                         dout=>prod_2);
        add2: adder port map (DIN_1=>prod_2, DIN_2=>din_rsh_pip, byp_mode=>byp_32,
                              SUM=>d1_tmp);

        -- 2nd pipeline stage
        r_3: SR1 port map (CLK=>CLK, RSTN=>RSTN, DIN=>s1_dly, DOUT=>s1_dly_pip);
        r_4: SR1 port map (CLK=>CLK, RSTN=>RSTN, DIN=>d1_tmp, DOUT=>d1_tmp_pip);

        ----- First group of folded shift registers -----

```

Appendices

```

rstn_s<=RSTN and rstn1;
sr_11: SR1_BYP port map (CLK=>CLK, RSTN=>rstn_s, DIN=>s1_pip, byp_mode=>byp_32,
    DOUT=>s1_dly1);
sr_12: SR1_BYP port map (CLK=>CLK, RSTN=>rstn_s, DIN=>s1_dly1, byp_mode=>byp_32,
    DOUT=>s1_dly2);
sr_13: SR2_BYP port map (CLK=>CLK, RSTN=>rstn_s, DIN=>s1_dly2, byp_mode=>byp_32,
    DOUT=>s1_dly3);

-- switches to configure the PE for different level of DWPT
s1_dly<=s1_dly1 when sel1="00" else
    s1_dly2 when sel1="01" else
    s1_dly3 when sel1="10" else
    (others=>'0');

gama<="000010001001001100";
mult3: multiplier18x18 port map (din1=>gama, din2=>s1_dly_pip, byp_mode=>byp_33,
    dout=>prod_3);
add3: adder port map (DIN_1=>prod_3, DIN_2=>d1_tmp_pip, byp_mode=>byp_33,
    SUM=>d1_new);
-- 3rd pipeline stage
r_5: SR1 port map (CLK=>CLK, RSTN=>RSTN, DIN=>s1_dly_pip, DOUT=>s1_dly_pip2);
    rstn_d<=RSTN and rstn2;
r_6: SR1 port map (CLK=>CLK, RSTN=>rstn_d, DIN=>d1_new, DOUT=>d1_new_pip);

-- 2nd Updating: s2
sub1: sub port map (D_MIN=>s1_dly_pip2, D_SUB=>d1_new_pip, byp_mode=>byp_34,
    DIFF=>s2);

----- Second group of folded shift registers -----
sr_21: SR1 port map (CLK=>CLK, RSTN=>RSTN, DIN=>d1_new_pip, DOUT=>d1_1);
sr_22: SR1 port map (CLK=>CLK, RSTN=>RSTN, DIN=>d1_1, DOUT=>d1_2);
sr_23: SR2 port map (CLK=>CLK, RSTN=>RSTN, DIN=>d1_2, DOUT=>d1_3);

-- switches to configure the PE for different level of DWPT
d1<=d1_1 when sel2="00" else
    d1_2 when sel2="01" else
    d1_3 when sel2="10" else
    (others=>'0');

sr_32: SR2 port map (CLK=>CLK, RSTN=>RSTN, DIN=>s1_dly_pip2, DOUT=>s1_byp2);
sr_33: SR2 port map (CLK=>CLK, RSTN=>RSTN, DIN=>s1_byp2, DOUT=>s1_byp3);
s1_byp<=s1_byp2 when sel2="01" else
    s1_byp3 when sel2="10" else
    (others=>'0');

-- Scaling
kesai<="001000010010000011";
kesai_inv<="011110111010001101";
mult4: multiplier18x18 port map (din1=>kesai, din2=>s2, byp_mode=>byp_34,
    dout=>prod_4);
mult5: multiplier18x18 port map (din1=>kesai_inv, din2=>d1, byp_mode=>byp_34,
    dout=>prod_5);

-- Output 18-bit=>16-bit by truncating the last 2 LSB
S_OUT <= prod_4(width+1 downto 2) when byp_34='0' else
    s1_byp(width+1 downto 2);
D_OUT <= prod_5(width+1 downto 2);

end struct;

```

Appendices