LOW POWER DIGITAL SIGNAL PROCESSOR ARCHITECTURE FOR MULTIPLE STANDARD WIRELESS COMMUNICATIONS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Tsung-En Andy Lee

March 1998

© Copyright by Tsung-En Andy Lee 1998
All Rights Reserved

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

____

Donald C. Cox
(Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

____

Bruce A. Wooley
(Associate Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

____

John M. Cioffi

Approved for the University Committee on Graduate Studies:

____

Abstract

There are many different standards being developed and deployed all over the world for cellular radio, wireless data link, and personal communications services (PCS). These standards were adopted by different parties focusing on different technologies, user applications, and geographical regions. As the technologies in wireless communications and integrated circuits advanced, the popularity of wireless communications increased dramatically. It became the fastest growing segment of the telecommunications market. The usage of different wireless communications standards started to overlap and create a need for an integrated environment across different geographical locations and technologies. Multiple standard wireless communications is now in demand.

In order to provide an integrated service across different existing and new future standards, a flexible hardware platform is highly desirable. The same hardware architecture should be able to process different wireless signals in different standards and also be able to accommodate future modifications of the standards. On the other hand, mobility is one of the primary reasons for the popular adoption of wireless communications applications in the telecommunication market. True mobility demands a long usage interval of mobile units. Frequent battery recharging reduces the mobility significantly. Therefore, mobile units need to consume as little power as possible. Low power consumption becomes another requirement in multiple standard mobile wireless communications. However, flexibility and energy efficiency have been recognized as two conflicting factors in hardware architectures. The more flexible a hardware architecture is, the more power it consumes. This imposes an severe dilemma on the system architecture for multiple standard wireless communications.

This dissertation presents new innovations that can be used to mitigate the tradeoff between programmability and power consumption in the environment of wireless communications processing. First, a new low power reconfigurable macro-operation signal processing architecture is presented. This architecture can be used to improve the performance and power consumption of computing engines inside the wireless communications mobile units. Second, a unique low power signal processing arrangement and method for predictable data is presented. This new approach deviates from the current one instruction stream architectures to a two instruction stream architecture. This provides a more efficient structure for data management specifically for multiple standard wireless communications. Third, an overall system architecture is described to make efficient use of the inventions mentioned above. A new register file architecture is also introduced to merge the functionality of microcontrollers and digital signal processors. This further reduces overhead from the communications and data transfers that would be required between two different processors.

The proposed architectures are modeled at register-transfer-level (RTL) by hardware description language, Verilog, to demonstrate their feasibility. The performance of the architectures are also studied with high level language simulations. The routines extensively used in wireless communications processing are simulated. Compared with conventional programmable processor architectures, up to 55% power reduction and up to 37% latency reduction are achieved by the proposed architectures for representative routines extensively used in wireless communications processing.

Acknowledgments

I am deeply indebted to my advisor, Professor Donald Clyde Cox, for his guidance, support, and encouragement. He has been a great teacher to me not only in my Ph.D. studies but also in many other aspects of an engineering career. During the past few years I have acquired from Professor Cox a great deal of technical knowledge, philosophical insights, and art of dealing with complicated situations. I am very honored to be his student, and sincerely thank him for the inspiring, rewarding, and enjoyable experience of working with him.

I would like to thank other members of my orals and reading committees: Professor John M. Cioffi, Nick McKeown, and Bruce A. Wooley. They scrutinized my research results and gave me many constructive comments on my dissertation.

This research is sponsored by Advanced Micro Devices, Inc., through Center for Telecommunications at Stanford. I would like to thank AMD and Center for Telecommunications for their generous support. In addition, I owe my sincerest gratitude to two greatest mentors from AMD: Dr. Safdar Asghar and Mr. James Nichols. Both of them have given me valuable suggestions over the course of this research. I also owe thanks to Mr. James Kubenic, who coordinated this Stanford-AMD Fellow-Mentor-Advisor program.

I would like to thank Miss Linda Chao at the Office of Technology Licensing of Stanford University and our patent lawyer Mr. Robert Crawford for filing patents for our research.

During my years at Stanford, I have been very fortunate to work and enjoy my life with many friends. My knowledge was enriched by the former and current members

of my research group: Bora Akyol, Sung Chun, Kerstin Johnsson, Byoung-Jo Kim, Daeyoung Kim, Persefoni Kyritsi, Derek Lam, Yumin Lee, Angel Lozano, Ravi Narasimhan, Tim Schmidl, Mark Smith, Mehdi Soltan, Jeff Stribling, Qinfang Sun, Karen Tian, Bill Wong, and Daniel Wong. Mark Hung, my best friend at Stanford, has been always beside me whenever I needed his help. Ming-Chang Liu, my old roommate, encouraged me during those confusing days at Stanford. Robert Henzel, my SCUBA diving buddy, jumped into cold ocean water with me to explore a whole new world. We went through military-like tough training to get certified as Divemasters together. We might have saved each other's life in many dangerous situations. Every Ph.D. student of the Department of Electrical Engineering cannot forget the thrilling Ph.D. qualifying examination. The friends in my study group helped me be prepared for it: Heng-Chih Lin, Hong Liu, Tien-Chun Yang, and Wu-Pen Yuen. While I served as the president of Stanford Chinese Culture Association, renamed as Stanford Taiwanese Students Association in my tenure, many friends were very nice to be my staff members and helped me through the unique experience: Chung-Hui Chao, Jason Chen, Tsung-Liang Chen, Yih-Lin Cheng, Ching-Wen Chu, Chih-Hsiu Hsu, Mark Hsu, Yen-Fen Hsu, Jan-Ray Liao, James Pan, and some other persons mentioned above. It would be very difficult, if not impossible, to mention all the friends whom I own thanks to. All my friends at Stanford made my Ph.D. experience pleasurable and memorable. I thank all of them and wish best of luck in their careers and studies.

In closing I would like to thank the most important persons in my life. I own everything I have to my parents, Chien-Chao Lee and Yu-Fang Tang. My brother, Tsung-Yu Lee, has been my best friend whom I can always rely on. My fiancée, I-Wen Winnie Tsou, and I have been together since we were sophomores in National Taiwan University. We went thorough all the happiness and sadness together from Taiwan to the United States. My parents, brother, and fiancée are always supportive and understanding. Their love makes my life meaningful. I dedicate this dissertation to them.

Contents

ABSTRACT .... **IV**

ACKNOWLEDGMENTS .... **VI**

1. INTRODUCTION **1**

1.1 DISSERTATION OUTLINE 6

1.2 CONTRIBUTIONS 7

2. PROCESSOR ARCHITECTURES AND WIRELESS COMMUNICATIONS **9**

2.1 HISTORICAL PERSPECTIVES OF PROCESSOR ARCHITECTURES 10

2.1.1 Microprocessor 10

2.1.2 Digital Signal Processor 13

2.1.3 Unified Microcontroller-DSP 14

2.2 PROCESSOR ARCHITECTURE FACTORS 15

2.2.1 Operand Transfer Scheme 15

2.2.2 Instruction Length and Code Density 16

2.2.3 Parallelism 17

2.2.4 Pipelining 17

2.2.5 Memory Hierarchy 18

2.2.6 Interrupt Handling 18

2.2.7 ASIC-orientation 19

2.3 INHERENT CHARACTERISTICS OF WIRELESS COMMUNICATIONS 19

2.3.1 Determinism 20

2.3.2 Repetition 20
2.3.3 Just-in-Time Process (vs. As Fast as Possible) 21
2.3.4 Limited, but More Complicated, Subset of Operations 21
2.3.5 Less Control Needed 22
2.3.6 Large Internal Dynamic Range, but Low Output Resolutions 22
2.3.7 Input / Output Intensive and Heavy Internal Data Flows 22
2.3.8 One Time Programming Software 23
2.4 THE NEED FOR NEW TECHNOLOGIES 23
2.5 SUMMARY 24

3. RECONFIGURABLE MACRO-OPERATION SIGNAL PROCESSING ARCHITECTURE 26

3.1 RECONFIGURABLE MACRO-OPERATION CONCEPT 27
3.1.1 The Challenges 29
3.2 PROPOSED ARCHITECTURE 30
3.2.1 Block Description 31
3.2.2 Macro-Operation Instruction Format 35
3.3 OPERATIONS 38
3.3.1 Flow Chart 39
3.3.2 Usage of Different Operation Modes 42
3.4 EXAMPLES 48
3.4.1 Symmetric FIR Filter 49
3.4.2 Adaptive FIR Filter 50
3.5 ARCHITECTURE IMPACTS 52
3.6 SUMMARY 54

4. SIGNAL PROCESSING ARRANGEMENT AND METHOD FOR PREDICTABLE DATA 55

4.1 SEPARATED DATA MANAGEMENT INSTRUCTION STREAM 56
4.2 PROPOSED ARCHITECTURE FOR THE SIGNAL PROCESSING ARRANGEMENT AND METHOD 58

4.2.1 Block Description 58
4.2.2 Register File Bank 60
4.3 Software Arrangements 65
4.4 Example 65
4.5 Architecture Impacts 68
4.6 Summary 70
5. OVERALL SYSTEM ARCHITECTURE 72
5.1 System Architecture 74
5.1.1 Block Description for Register File Bank 74
5.1.2 Universal Operand Access 77
5.2 Software Program 79
5.3 Summary 84
6. PERFORMANCE EVALUATION 85
6.1 Power Consumption of CMOS Circuits 85
6.2 Simulation 89
6.3 Reconfigurable Macro-Operation Signal Processing Architecture 94
6.4 Signal Processing Arrangement and Method for Predictable Data 96
6.5 System Performance 99
6.5.1 Low Power 99
6.5.2 Low Latency 99
6.5.3 Low Complexity 100
6.5.4 Software Programming Complexity 100
6.6 Summary 101
7. CONCLUSIONS 102
7.1 Dissertation Summary 102
7.2 Future Work 107
7.2.1 Compiler Technology 107
7.2.2 Memory Layer Structure 107

7.2.3 Variations and Scalability 108

BIBLIOGRAPHY 109

List of Tables

Table 1 - Wireless communication standards...2
Table 2 - Microprocessor evolution 11
Table 3 - Evolution of digital signal processors...13
Table 4 - The relations between operation type, active register file assigned in the instruction (logically), and physical arrangement for the active register input/output registe files. 78
Table 5 - Software program partitioning: different types of operations in the data management code and in the computational code. 78
Table 6 - Performance summary of reconfigurable macro-operation signal processing architecture. Crossover point is defined as the minimum N for which the power consumption ratio is lower than 100%. Asymptotic power reductions listed are the asymptotic values for large N. 96
Table 7 - Summary of power reduction achieved by the proposed signal processing arrangement and method for predictable data..97
Table 8 - Summary of latency reduction achieved by the proposed signal processing arrangement and method for predictable data..98

List of Figures

Figure 1 - Tradeoff between flexibility and power consumption. System implementation approaches can be categorized into "general purpose microprocessor", "digital signal processor plus microcontroller", "digital signal processor, microcontroller, and some application specific integrated circuits", and "application specific integrated circuits only". All of these approaches align approximately on a straight line in the chart of flexibility vs. power consumption. 4

Figure 2 - Desired power-flexibility region. As shown in Figure 1, all current approaches align approximately on a straight line in the chart of flexibility v.s. power consumption. The desired region is in the area of high flexibility and lower power consumption. New technologies are needed to shift the current regions of programmable processors towards the desired region. 5

Figure 3 - Illustration of a combination of operations. Given several operation sources, we can generate several outputs with given inputs. Each output is a function of some or all inputs. All these outputs create a very sophisticated set of function groups..27

Figure 4 - Reconfigurable macro-operation signal processing architecture. The building blocks include macro-operation instruction decoder, decoded macro-operation instruction register file, macro-operation management unit (MOMU), computational units, macro-operation connection multiplexer, macro-operation input registers, macro-operation

output/interrupt registers, 8-channel memory loading DMA controller, 4-channel memory storing DMA controller, DMA source address registers, and DMA destination address registers...31

Figure 5 - Illustration of macro-operation connection multiplexer. There are a total of eight inputs to the computational units. These are numbered from 0 to 8. Each input has six choices: four from the feedback bath, one from DMA, and one from an input register. The assignment of DMA and register inputs to the computational units are fixed. Operands from different DMA or input registers are not exchangeable. 33

Figure 6 - Macro-operation instruction format that is 10-bytes long. The higher 4 bytes are for basic control and are always decoded. The lower 6 bytes are the detailed information and are ignored if a stored macro-operation is called for. The bit fields are: macro-operation indication [79:72], flag 1 - new/stored instruction [71], flag2 - discrete/loop mode [70], operation depth [69:68], instruction register number [67:64], counter setting [63:48], multiplier function [47:44], ALU 1 function [43:40], ALU 2 function [39:36], shifter function [35:32], output register setting [31:28], output DMA setting [27:24], and input/feedback setting [23:0]........36

Figure 7 - Macro-operation flow chart. Flow begins at block 1 and ends at either block 14 or block 29, depending on which mode is run. 40

Figure 8 - Hardware configuration for symmetric FIR example. Input data x(n-i) and x(n-N+1+i) are fed to an ALU for the add operation. Filter coefficient h(i) is multiplied by the sum of x(n-i) and x(n-N+1+i). The resolution of the result is adjusted before being added to the accumulation of the previous results. The index i is incremented automatically in the execution of a macro-operation. The final result is stored into a macro-operation output register. 50

Figure 9 - Hardware configuration for adaptive FIR example. In order to generate y(n), we have to update $B_k(n-1)$ to $B_k(n)$ and, therefore, we have to update $C_k(n-1)$. The multiply-and-accumulate of x(n-k) and $B_k(n)$,

and the update of $B_k(n)$ are done in the macro-operation. The update of $C_k(n-1)$ is handled by a micro-operation mixed with the macro-operation in the loop. $B_k(n-1)$ and $C_k(n-1)$ are summed by an ALU to generate the $B_k(n)$, which are fed to the multiplier to be multiplied by x(n-k) and also are stored back into the coefficient array in the memory for updating. The index k is incremented automatically in the execution of a macro-operation. The final result from the multiply-and-accumulate of x(n-k) and $B_k(n)$ is stored into a macro-operation output register. 52

Figure 10 - Architecture diagram of signal processing arrangement and method for predictable data. The building blocks include two program decoders, data management unit (DMU), computational units, several layers of memory spaces, and a special register file bank. 57

Figure 11 - Example of using active input and output register files. Computational instruction "MULTI R0, R1, R2" is issued. From the instruction itself, we do not know to which register files the registers R1, R2, and R3 belong. Data management codes assign register file 0 as the active input register file and register file 1 as the active output register file. Therefore, the multiplier gets inputs from registers R1 and R2 of register file 0. The output is stored into register R0 of register file 1..........................60

Figure 12 - Example of code reuse by swapping the active register files. A set of operations are to be performed on a set of inputs a[i], b[i], c[i], and other variables to create a set of outputs, i.e. output[i]. The inputs can be distributed into several register files so that the same computational codes can process through them by swapping the active register file assignment. No computational instruction modification is needed as we swap from one register file to another. 62

Figure 13 - Data block access to register files. Two dimensions of block access are provided. One is to access several registers in a single register file. The other is to access one register of several register files. 63
Figure 14 - Illustration of the constraints on block data transfer. Four types of block access are supported: full register file access, half register file access, full row access across register files, and half row access across register files. (x, y) in the diagram means the register y of register file x. 64
Figure 15 - Register file contents for symmetric FIR example. First, filter coefficients, h's, are stored in register file 0 and 1. The input data, x's, are stored into register files 2 and 3 in the arrangement shown in the figure. Second, pairs of inputs are summed and stored in register file 0 and 1. Then, two partial accumulations are generated and stored. Finally, the partial accumulations are summed together to provide the final output. 66
Figure 16 - The diagram of the system architecture combining the reconfigurable macro-operation signal processing architecture shown in Figure 4 and the data management architecture for predictable data shown in Figure 10..73
Figure 17 - Register file bank. The building blocks include active input register file, active output register file, micro-operation register files, virtual macro-operation register files, macro-operation input registers, macro-operation input DMA channels, macro-operation configuration multiplexer, macro-operation output registers, and macro-operation output registers. 75
Figure 18 - Software programming procedure for the proposed system architecture. 82
Figure 19 - Procedure flow for performance evaluation. The flow starts at defining global specifications (block 1) and ends at complexity analysis (block 9), power analysis (block 15), and latency analysis (block 16). 90
Figure 20 - Power consumption reduction achieved by using reconfigurable macro-operation signal processing architecture. Five benchmark routines are plotted: N-tap symmetric FIR filter, N-dimension square distance, N-tap

LMS adaptive FIR filter. N-entries block normalization, and N-entries VSELP codebook search. When the power ratio is lower than 100%, the proposed architecture consumes less power than traditional architectures. 94

Chapter 1

Introduction

There are many different standards being developed and deployed all over the world for cellular radio, wireless data link, and personal communications services (PCS) [Cox95][Nat97]. These standards were adopted by different parties focusing on different technologies, user applications, and geographic regions [Rap96]. Table 1 lists several popular standards. As the recent advances in very large scale integrated circuits (VLSI) technology make complex systems reliable and affordable, the popularity of wireless communications increases dramatically. For example, in the United States alone, on the order of ten million new wireless communications handsets are sold to end users every year. Also, about seven million people have subscribed to the personal handy-phone (PHS) service since PHS was first introduced in July 1995 in Japan. The dream of possessing the capability to communicate with each other anytime and anywhere appears not too far away [Cox87]. However, in order truly to achieve this goal, an integrated environment across different protocols is needed [Lac95]. A single mobile unit would need to be compatible with many existing infrastructures and user applications. Furthermore, wireless communications technologies improve very fast. More and more advanced features are added to user applications constantly. In addition, many new wireless communications standards are introduced into the market every year. In the United States, at least four cellular/PCS standards, four wireless wide area data network

Table 1 - Wireless communication standards

Standard	***Type***	***Frequency (MHz)***	***Multiple Access***	***Number of Channels***	***Channel Space***	***Modu-lation***
AMPS	Analog Cellular	Rx:886-894 Tx:824-949	FDMA	832	30kHz	FM
ETACS	Analog Cellular	Rx:916-949 Tx:871-904	FDMA	1240	25kHz	FM
NTACS	Analog Cellular	Rx:860-870 Tx:915-925	FDMA	400	12.5kHz	FM
NMT - 900	Analog Cellular	Rx:935-960 Tx:890-915	TDMA /FDM	1999	12.5kHz	FM
IS-54/136	Digital Cellular	Rx:869-894 Tx:824-849	TDMA /FDM	832 3 users/ch	30kHz	π/4 DQPSK
IS-95	Digital Cellular	Rx:869-894 Tx:824-849	CDMA /FDM	20 798* users/ch	1250kHz	QPSK/ OQPSK
GSM	Digital Cellular	Rx:925-960 Tx:880-915	TDMA /FDM	124 8 users/ch	200kHz	GMSK
DCS1800	Digital Cellular	Rx:1805-1880 Tx:1710-1785	TDMA /FDM	374 8 users/ch	200kHz	GMSK
PDC	Digital Cellular	Rx:810-826 & 1429-1453 Tx:940-956 & 1477-1501	TDMA /FDM	1600 3 users/ch	25kHz	π/4 DQPSK
CT2/2+	Digital Cordless	CT2:864/868 CT2+:944/948 (TDD)	TDMA /FDM	40	100kHz	GFSK
DECT	Digital Cordless	1880-1900 (TDD)	TDMA /FDM	10 12 users/ch	1728kHz	GFSK
PHS	Digital Cordless	1895-1918 (TDD)	TDMA /FDM	300 4 users/ch	300kHz	π/4 DQPSK
CDPD	Data	Rx:869-894 Tx:824-849	FDMA	832	30kHz	GMSK
RAM - Mobitex	Data	Rx:935-941 Tx:896-902	TDMA /FDM	480	12.5kHz	GMSK
Ardis-RD-LAP	Data	Rx:851-869 Tx:806-824	TDMA /FDM	720	25kHz	FSK

* Theoretical capacity listed in the IS-95 standard.

Table 1 (cont.) - Wireless communications standards

Standard	***Type***	***Frequency (MHz)***	***Multiple Access***	***Number of Channel***	***Channel Space***	***Modu-lation***
IEEE 802.11	Data	2400-2483 (TDD)	CSMA	FHSS: 79 DSSS: 7	FHSS /DHSS: 1/11MHz	GFSK/ DBPSK DQPSK
PCS TDMA	PCS	Rx:1930-1990 Tx:1850-1910	Based on IS-136 cellular phone standard.			
PCS CDMA	PCS	Rx:1930-1990 Tx:1850-1910	Based on IS-95 cellular phone standard.			
PCS 1900	PCS	Rx:1930-1990 Tx:1850-1910	Based on GSM cellular phone standard.			
PACS	PCS	Rx:1930-1990 Tx:1850-1910	TDMA /FDM	200 8 users/ch	300kHz	π/4 DQPSK
DCT-U	PCS	Rx:1930-1990 Tx:1850-1910	Based on DECT cordless phone standard.			

(WAN) standards, and several wireless local area network (LAN) standards were deployed in the past few years. Mobile units should be upgradeable to avoid becoming obsolete in a short period. Therefore, there is a strong demand for a flexible hardware platform for wireless communications mobile units [Mit95].

On the other hand, power consumption is a major concern in all mobile applications. A long interval between battery recharging is strongly desired. Mobile units need to consume as little power as possible. However, there is an inherent tradeoff between flexibility and power consumption. Flexibility requires generalized computation and data movement structures, which can be used for different tasks. Compared with dedicated hardware that can fully exploit the properties of a given task, generalized structures consume more power. The tradeoff is illustrated in Figure 1.

Application specific integrated circuits (ASIC) are specifically designed for given tasks. Computing engines and data transfer schemes can be highly optimized. Therefore, ASIC's have the highest energy efficiency. However, the computational functions are limited to the algorithms involved in the tasks. Other algorithms cannot be executed on the system. The basic system structures are not optimized and may be unusable for other

tasks. Although the functionalities of an ASIC system can be increased by adding more dedicated blocks, this does not increase the programmability of the system [Abn96]. The functionalities of the system are still pre-fixed and cannot be modified to execute new operations. In addition, if we add many different ASIC blocks to a system in order to execute all possible operations needed, the system becomes a giant function-based general purpose processor. The extra global control signals and data transfers among all those ASIC blocks consumes a significant amount of power. This would lose the advantages of the application-specific architectures. The system would eventually consume more power than conventional micro-operation-based general purpose microprocessors and the system still could not reach the same level of programmability of

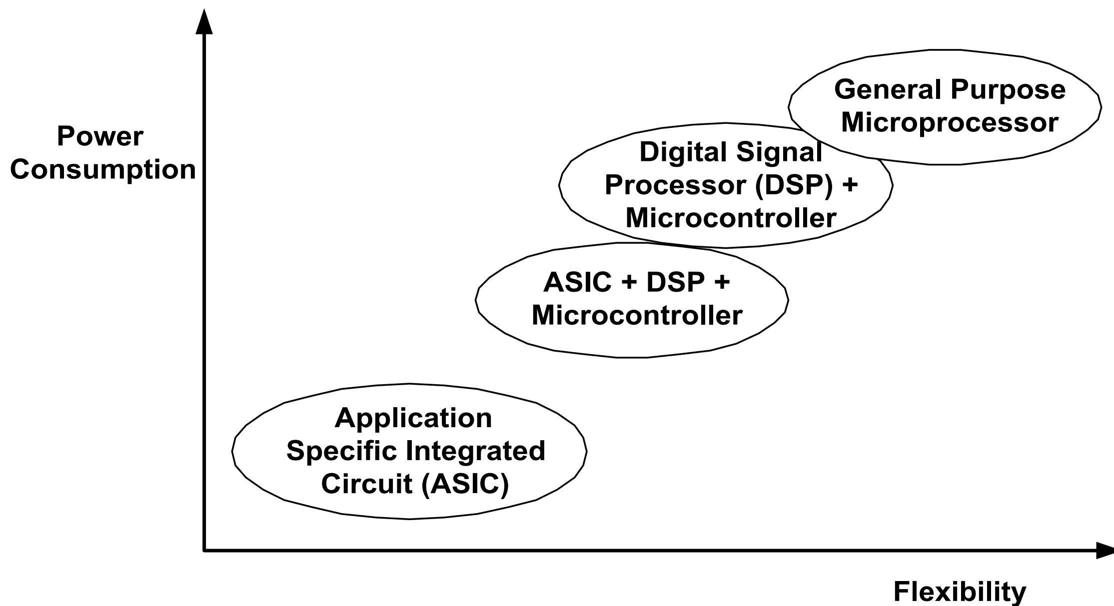


Figure 1 - Tradeoff between flexibility and power consumption. System implementation approaches can be categorized into “general purpose microprocessor”, “digital signal processor plus microcontroller”, “digital signal processor, microcontroller, and some application specific integrated circuits”, and “application specific integrated circuits only”. All of these approaches align approximately on a straight line in the chart of flexibility vs. power consumption.

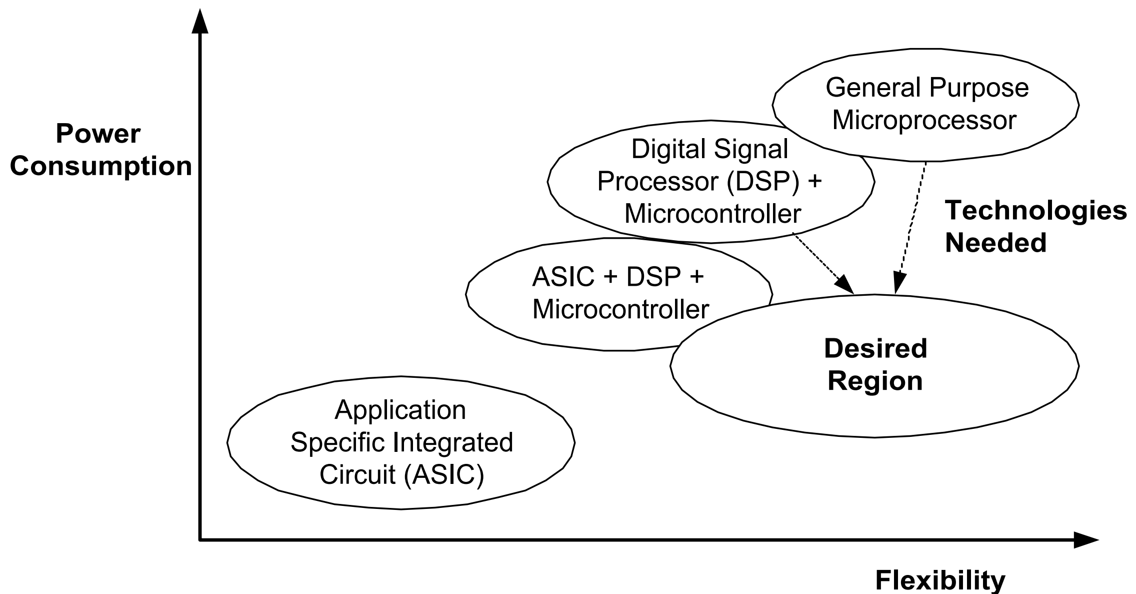


Figure 2 - Desired power-flexibility region. As shown in Figure 1, all current approaches align approximately on a straight line in the chart of flexibility v.s. power consumption. The desired region is in the area of high flexibility and lower power consumption. New technologies are needed to shift the current regions of programmable processors towards the desired region.

general purpose microprocessors. Therefore, ASIC's are considered least flexible. General purpose microprocessors, on the other hand, have the highest programmability. All controlling and computing functions are implemented in software and subject to change anytime. This provides the most straightforward solutions to multiple standard wireless communications. However, the power consumption is much higher than the level that any mobile unit can tolerate. In between general purpose microprocessors and ASIC's are some combinations of digital signal processors, microcontrollers, and/or function-dedicated ASIC's. As shown in Figure 1, all these approaches reside approximately on a line linking the two extremes - the more flexibility provided, the more power consumed. No matter what approach we take, the tradeoff between flexibility and energy efficiency cannot be avoided. This presents a dilemma in multiple standard mobile wireless communications processing, which demands both high flexibility and low power consumption. Therefore, there is a strong need for new

technologies that can make this tradeoff less severe. With new technologies, programmable processor architectures can shift to a region of lower power consumption without sacrificing the flexibility. The desired power-flexibility region is illustrated in Figure 2.

This dissertation discusses new digital signal processing architectures that have been created to push programmable processors into the desired power-flexibility region in the environment of multiple standard mobile wireless communications. First, a new low power reconfigurable macro-operation signal processing architecture is presented. This architecture can be used to improve the performance and power consumption of computing engines inside the wireless communications mobile units. Secondly, a unique low power signal processing arrangement and method for predictable data is presented. This new approach deviates from the current one instruction stream architectures to a two instruction stream architecture. This provides a more efficient structure for data management specifically for multiple standard wireless communications processing. Thirdly, an overall system architecture is described to make efficient use of the inventions mentioned above. A new register file architecture is also introduced to merge the functionality of microcontrollers and digital signal processors. This further reduces overhead from the communications and data transfers between two processors.

1.1 Dissertation Outline

Chapter 2 of this dissertation generally discusses the relations between processor architectures and wireless communications. Some factors categorizing different processor architectures are discussed. The historical perspectives of microprocessor, digital signal processor, and unified microcontroller-DSP architectures are described. The inherent characteristics of wireless communications are examined so that we can understand the advantages and disadvantages of applying different architectures to multiple standard mobile wireless communications.

In Chapter 3, a new low power reconfigurable macro-operation signal processing architecture is presented. The macro-operations can be reconfigured and optimized for current routines in real time. Extra instruction decoding, unnecessary data movement, and unneeded control signaling are eliminated. Compared with general purpose microprocessors and digital signal processors, this technology consumes much less power for a given task while retaining the flexibility needed to do the processing for different wireless standards.

In Chapter 4, a unique low power signal processing arrangement and method is presented. The software codes are separated into two parts: computational codes and data management codes. The two parts are issued to two different program decoders in parallel. Computational instructions are simplified by using register-to-register data movement only. While some data are being processed by computational codes, other data are transferred among memory hierarchies and prepared for following processes by data management codes.

Chapter 5 discusses a new overall system architecture using the technologies discussed in Chapter 3 and Chapter 4. It also presents a new register file architecture merging macro-operations and micro-operations into an universal arrangement. The functionalities of microcontroller and digital signal processor needed for processing wireless signals is provided in a single system.

Chapter 6 discusses the simulated performance of the proposed architectures. It demonstrates the feasibility of the innovations. Simulation results show that the proposed architectures can greatly improve the tradeoff between flexibility and power consumption in the environment of multiple standard wireless communications.

Chapter 7 contains the concluding remarks and some potential future work.

1.2 Contributions

Major contributions described in this dissertation include:

- Invented a low power reconfigurable macro-operation signal processing architecture.
- Created a low power signal processing arrangement and method for predictable data.
- Invented a register file architecture to merge macro-operations and micro-operations into an universal arrangement.
- Created a system architecture using the inventions listed about.
- Modeled and simulated the system in hardware description language and high level language.
- Evaluated the proposed architectures and compared the performance with traditional architectures.

Chapter 2

Processor Architectures and Wireless Communications

Microprocessors and digital signal processors have been designed for a wide range of applications found in virtually every industry sector. Different applications have different requirements for their computing engines and supporting structures. Some applications emphasize processing speed and the number of tasks processed in parallel. Some applications emphasize compactness and low power consumption. Some applications further require that computing engines perform on a real-time or near real-time basis. The selection or design can largely depend on which application is expected to be served by the computing engine. In this chapter, we first describe existing programmable processor architectures from historical perspectives. The philosophies behind micro-processors, digital signal processors, and unified microcontroller-DSP's are discussed. Secondly, we discuss the basic factors that distinguish different processor architectures. These factors help us understand how a processor architecture is constructed and how a new innovation can impact an existing processor architecture. After that, we discuss the inherent characteristics of wireless communications processing and why existing architectures are not optimized for power consumption for multiple standard wireless communications.

2.1 Historical Perspectives of Processor Architectures

The programmable processor is one of the most important innovations in modern technology evolution. It provides the computing and controlling capability that could not even be imagined before. As advanced semiconductor manufacturing technology makes integrated circuits commodities, it becomes challenging to organize millions of operations in a second. Processor architectures become more and more complicated. In this section, we discuss those architectures from a historical point of view. Learning from past innovations, we can understand what have been emphasized and in what direction the current path is heading. On the other hand, while new applications in different environments are being executed on those architectures, we can understand what has been overlooked intentionally or unintentionally in the past.

2.1.1 Microprocessor

The general purpose microprocessor has been the main stream of processor developments since integrated circuits were invented. As shown in Table 2 [Hen95], many new technologies were invented along the way of development. Most, if not all, of these technologies share the same goal: make the processing faster. Execution speed cannot be over-emphasized in these architectures.

General purpose registers were first introduced in 1956. This is a break-through idea to make the operations more efficient. Frequently used operands are stored in a small internal storage space inside the processor. This makes the executions much faster. Surprisingly, pipelining technology was invented in the early stage of microprocessor evolution. Pipelining is an implementation that has been responsible for dramatic increase in microprocessor performance for many decades. It exploits instruction level parallelism by overlapping the executions of instructions. Although the latency of a single instruction

Table 2 - Microprocessor evolution

Year	*Technology*	*Company/Processor*
1956	General-purpose registers	Ferranti Pegasus
1959	Pipelining	IBM Stretch 7030
1963	Stack machine	Burroughs B5000
1964	Load/store architecture	CDC 6600
1964	Dynamic scheduling	CDC 6600
1967	Dynamic scheduling	IBM 360/91
1967	Speculative execution	IBM 360/91
1967	Branch prediction	IBM 360/91
1968	Cache	IBM 360/85
1972	Virtual memory	IBM S/370
1972	SIMD/vector	CDC STAR-100
1972	SIMD/vector	TI ASC
1975	RISC	IBM801
1980	RISC	Berkeley RISC-I
1981	RISC	Stanford MIPS
1981	VLIW	AP-120B
1984	MIMD with cache coherence	Synapse N+1
1987	Superscalar	IBM America
1997	EPIC	Intel Merced

SIMD: single instruction multiple data

RISC: reduced instruction set computer

MIMD: multiple instruction multiple data

VLIW: very long instruction word

EPIC: explicitly parallel instruction computing

execution remains unchanged, the throughput is scaled up by the number of pipeline stages. The tradeoffs are in system complexity and power consumption.

The stack machine departs from the general purpose register architecture. The instructions are shorter and easy to utilize. However, there is no random access to data in the stack. Software code becomes less efficient for general tasks. Therefore, the stack has been considered a bottleneck in many situations. Not until recently has the stack machine been re-considered in processor architectures. Some new applications and program languages find the old technology of the stack machine favorable.

Data loading and storing have been emphasized more since 1964 as new technologies made the huge data flows more manageable. Around the same period, processor architectures began to utilize advanced instruction scheduling, which rearranged instruction order to increase the throughput. Furthermore, by using branch prediction, instructions are executed even before it is absolutely certain that those instructions need to be executed. This approach often jumps ahead and speeds up executions. Obviously, it sacrifices energy efficiency significantly when instructions are executed when not needed. However, in the 1970's, mobile applications were not popular and power consumption was not a major concern.

Since 1972, more and more computational resources have been added to increase the data throughput. SIMD (single-instruction-multiple-data), MIMD (multiple-instruction-multiple-data), VLIW (very-long-instruction-word), and superscalar architectures utilize multiple computational units in parallel. The major purpose is, again, to increase data throughput. While these complex architectures were being developed, another innovation appeared. That is the RISC (reduced-instruction-set-computer) architecture. RISC architecture sacrifices memory efficiency to make the instructions simpler and execute faster. Since its appearance, RISC has dominated most other processor architectures in terms of performance and become the primary architecture for high performance processors [Hen95].

The evolution of microprocessor architectures has decreased significantly since 1987. Most new advances have been at the circuit level instead of the architecture level. The old architectures were modified and combined in newer microprocessors. The only

new innovation is EPIC (explicitly parallel instruction computing) architecture. However, EPIC is not really a hardware innovation. It is a new philosophy in software compilers [Cra97]. Instead of using dynamic arrangements handled by hardware in real time, it arranges the instruction parallelism explicitly at the software level while the software programs are being complied. Even though the innovation is in software, the concern is still the data throughput, or the execution speed.

2.1.2 Digital Signal Processor

Digital signal processors were created as computing engines that were not responsible for the flow control of applications [Nor95]. Originally, they served as co-processors to microprocessors to execute extensive arithmetic operations. The most distinguishing characteristic is the single-stage multiply-and-accumulate operation, which is used extensively in dot- product types of algorithms. Except for this operation, there have not

Table 3 - Evolution of digital signal processors

Year	*Generation*	*Feature*	*Examples*
1982	1st	Basic Harvard 1 data bus, 1 program bus 1 Multiply-ALU unit	TMS32010 NEC 7720
1986	2nd	“Modified” Harvard 1 data and program bus 1 data bus	TMS320C25 AT&T DSP16A
1990	3rd	Extra addressing modes Extra functions	TMS320C5x AT&T DSP 161x
1994	4th	2 data buses 1 program bus Separate MAC and ALU	TMS320C540
1996	5th	2 Multipliers 4 ALU’s 2 Shifters	TMS320C6x

been many exciting advances in digital signal processor architectures. As shown in Table 3, the evolution of digital signal processors has not been as rapid as the advances in

microprocessors. The advances in digital signal processors have been basically adding more resources. More buses, computational units , and operand addressing modes were added to newer generations [Bai95]. The boundary between two generations was not very clear until recently. In general, digital signal processor technology is about ten years behind microprocessor technology. Many ideas are borrowed from microprocessors to enhance the performance of digital signal processors.

2.1.3 Unified Microcontroller-DSP

The unified microcontroller-DSP is a new trend in processor architectures [Fle97]. A microcontroller and a digital signal processor are combined into a single system. This single system is responsible for procedure controlling and arithmetic computing. It makes the system similar to microprocessors in functionality. However, because of different applications, the instruction sets are different. Unified microcontroller-DSP's are mainly used in those applications which need strong computational capability and have fewer control requirements. There are many advantages for using a unified microcontroller-DSP in those applications. Some of the major advantages are listed in the following:

(1) The processor has the capacities of both micro-controller and digital signal processor. An unified instruction set is integrated in a single core. It can achieve better optimization.
(2) It has low control overhead and reduced software complexity. In real-time processing, compared with using two separate processors, it results in faster system response, but does not increase power consumption.
(3) It reduces traffic on data buses. Heavy traffic on global buses is a significant reason for high power consumption. Eliminating unnecessary traffic on data buses reduces power consumption significantly.
(4) Combining two processors into a single system reduces hardware, such as deletion of signal input and output drivers. It needs less silicon area. The cost is less.

The unified microcontroller-DSP is still in the conceptual stage. Although there are several trial architectures, all of them combine microcontroller and digital signal processor functionality only at the instruction set level. After the instructions are issued, microcontroller-oriented and DSP-oriented instructions are dispatched to different portions of the processor and executed as in two separate processors. The performance of these trial architectures has improved, but they still have not fully exploited the advantages listed above.

2.2 Processor Architecture Factors

There are several factors that distinguish different processor architectures. They are operand transfer scheme, instruction length and code density, parallelism, pipelining, memory hierarchy, interrupt handling, and ASIC-orientation. In different applications, these factors are weighted differently. In mobile wireless communications, the factors that have direct impact on power consumption are more important and should be considered first.

2.2.1 Operand Transfer Scheme

One of the most important characteristics of a processor architecture is how operands are transferred into and out of computational units. This classifies the fundamental structure of an instruction set and impacts the data bus traffic. An operand transfer scheme can be characterized by two things: how operands are temporarily stored, and how operands are utilized in instructions. The methods of temporary operand storage can be categorized into stack, accumulator, and register set. On the other hand, the utilization of operands in instructions depends on how many operands can be named, and how many of them are in memory and in temporary storage. This implies that an arithmetic instruction is always associated with data movement among different memory hierarchies. For example, if an

architecture uses only a memory-to-memory operand transfer scheme, all arithmetic operations cause traffic on the data buses bridging the processor and memory devices. Heavy bus traffic results in significant power consumption. In mobile wireless commutations processing, most routines contain heavy data flows. Therefore, the operand transfer scheme has a significant impact on the overall performance.

2.2.2 Instruction Length and Code Density

Instruction length and code density are best discussed by looking at reduced-instruction-set-computer (RISC) and complete-instruction-set-computer (CISC) architectures. In RISC architectures, instructions are short and simple. They have simpler addressing modes, fewer references to main memory, and shorter execution latencies. In CISC architectures, instructions are more complex and of different lengths. More actions are taken in a single CISC instruction. The fundamental difference between these two types of architectures is the code density, that is the ratio between the number of effectively executed actions and the instruction length. Although CISC architectures have longer word lengths, the code density tends to be higher because more actions are compressed into a single instruction. The higher the code density is, the more efficient the instruction is. However, this does not imply higher performance in execution speed. It has be recognized that higher execution speed can be achieved by sacrificing memory efficiency in the encoding of an instruction set. Simple fixed-length instructions are easier to implement and executed much faster. Also, instructions of higher code density are less flexible to use. Most of the complex instructions in CISC architectures are rarely used. In general, only about 10 percent of the CISC instructions account for over 90 percent of the total software code. Therefore, there is no optimum instruction word length and code density. It depends on what is more important for the application: memory efficiency or speed of execution.

2.2.3 Parallelism

Parallelism is defined by the number of tasks that can be processed by a processor at the same time. It depends on how many resources a processor has, and how many of the resources it can utilize in parallel. In many cases, parallelism is not limited by the number of computational units. Instead, it is often limited by the instruction set and operand transfer abilities. The inherent possibility of operand conflict also keeps parallelism low in general tasks. We cannot process the operands before they are ready. Also, in non-deterministic routines, we have no knowledge of future data other than perhaps a few cycles of prediction. This prevents the processing of operands many cycles ahead, even though they might not generate any conflict. Therefore, the performance does not increase proportionally when we provide more computational resources than needed for minimal arithmetic requirements. The real performance gained by increasing hardware parallelism strongly depends on the inherent parallelism of the software programs to be executed.

2.2.4 Pipelining

Pipelining is the technology for increasing the overall data throughput. It has had a dramatic impact on processor evolution. Pipelining has been used virtually in all advanced microprocessors focusing on speed. Different processors use different numbers of pipeline stages and assign different tasks to each stage. However, all pipelined architectures have relatively higher power consumption because of the extra control and data buffering among stages. Even in those microprocessors mainly focusing on processing performance, the pipelining tradeoff is becoming a challenging problem as the clock rate keeps increasing. It is not necessarily beneficial to tradeoff significant hardware complexity and power consumption to gain only a few percent in speed, unless the speed requirement outweighs all other considerations.

2.2.5 Memory Hierarchy

Memory hierarchy is more emphasized in general purpose microprocessor architectures than in digital signal processing architectures. Different memory hierarchies are used to resolve the inherent dilemma between data transfer speed and overall data storage space. For example, the data mapping schemes can be categorized into direct mapping, set-associative mapping, fully associative mapping, and others. The cache locations related to the processor and the main memory can be categorized into back-side, front-side, sit-aside, and others. There are also many different strategies for cache write hit and miss, such as write-through, write-back, write-allocate, no-write-allocate, fetch-on-write-miss, no-fetch-on-write-miss, and etc. More detailed discussion on memory hierarchy can be found in [Hen95]. In general, the data transfer latency is proportional to the size of the memory, no matter what kind of circuit technology is used. Memory hierarchies exploit the spatial and temporal localities of software programs to mitigate the dilemma. Many control schemes between different memory levels have been created to reduce the average data fetch latency. These control schemes in conjunction with the associated memory structures make one architecture different from another.

2.2.6 Interrupt Handling

Interrupt handling ability is another consideration in processor architectures. It strongly depends on other architecture factors discussed in this section and on the operating systems (OS) running on the processors. In a multi-tasking environment, interrupt handling becomes essential to the overall system performance. A long latency in switching tasks causes a significant penalty on the performance. An architecture supporting interrupts efficiently is desirable in programmable processors handling multiple general task. On the other hand, in real time signal processing, there is usually a primary task running all the time and those interrupting tasks tend to be small and less

time critical. Special consideration is needed in interrupt handling to avoid extra power consumption.

2.2.7 ASIC-orientation

ASIC-orientation is a relatively new factor being considered in processor architectures. As the tradeoff between performance and power consumption becomes more severe, many processors are beginning to include some ASIC portions in the architectures. These ASIC portions are utilized when some very special routines are processed. Obviously, with special purpose functions, the processors become much less general purpose. The applications for these processors have to be known in advance. In traditional mobile wireless communications, this approach is often favored. Some heavy loading routines, for example, Viterbi decoding, are taken out of the main embedded software for the processors in mobile units. They are executed in ASIC portions and called by the main software as functions. In future multiple standard wireless communications processing mentioned before, the possibility of utilizing a special fixed routine is reduced. Lacking the flexibility for changes, the advantage of including ASIC portions in processors become less significant. However, the philosophy of ASIC can be broadened. According to the category of applications, certain portions of an architecture can be emphasized. Some types of resources can be added or removed. For example, an extra ALU can be added for better optimization of given algorithms, or data loading and storing blocks can be emphasized in applications requiring heavy memory traffic.

2.3 Inherent Characteristics of Wireless Communications

Although each wireless communications standard has its own particular specifications, they all share the same inherent characteristics. These characteristics distinguish wireless

communications processing applications from other real time applications run on general purpose microprocessors and digital signal processors. These important characteristics are usually not considered when determining traditional processor architectures which are not specifically designed for wireless communications. By understanding and taking advantage of these characteristics, we can provide processor technologies that improve processor performances in the environment of wireless communications. Several primary characteristics are listed in the following.

2.3.1 Determinism

Determinism is the most important characteristic in wireless communications. All tasks in wireless communications protocols have well defined functionality and timing sequences. They are highly deterministic. The operations in a certain time slot can be well predicted. This gives the software programs the privilege of prearranging data structures and even modifying hardware configurations in advance. We can also distribute the task loading to avoid a high peak demand on processing power. Therefore, determinism can relax hardware requirements. Many tradeoffs can be made for run time, according to the executed software routines. They do not have to be compromised and fixed in advance.

2.3.2 Repetition

Wireless communications utilizes countless repeated operations. For example, in a finite impulse response (FIR) filter used in many wireless transceivers, the same set of coefficients are multiplied by input data and summed together to generate output data. In general, there are several ten's to several thousand's of input data in an incoming data stream. The same multiplication and accumulation operations are executed again and again to generate corresponding several ten's to several thousand's of outputs. In a broader sense, most wireless standards incorporate time-division-multiple-access

(TDMA) in conjunction with other multiple access and multiplexing methods.[*] All received and transmitted data are divided into many small time frames. Each grouping of data goes through the same processes. The operations performed on the data repeat every time frame. This repetition characteristic allows us to modify the hardware configuration more freely. If we utilize the same operations many times, the overhead from hardware reconfiguring becomes insignificant and is outweighed by the gain achieved by better hardware configurations.

2.3.3 Just-in-Time Process (vs. As Fast as Possible)

Unlike in other applications, speed is not the most important factor in wireless communications signal processing. The radio frequency bandwidth allocated for wireless communications is limited. The modern signal processing technologies in very large scale integrated circuits provide more than enough computing power even for the most advanced algorithms currently being used in wireless communications. Instead of as fast as possible, the tasks should be finished just in time within the assigned time slot. This reduces the required clock speed and, therefore, reduces the required power supply voltage. Hence, the power consumption can be reduced to minimum.

2.3.4 Limited, but More Complicated, Subset of Operations

Compared with general tasks, wireless communications signal processing utilizes a smaller set of types of operations, but, in general, the operation elements in this subset are more complicated. Therefore, CISC-oriented instructions can be utilized more frequently in wireless communications applications than in applications in other fields.

---

[*] Even the IS-95 CDMA standard blocks its data in TDMA-like frames.

2.3.5 Less Control Needed

Compared with general processing tasks, less control is needed while wireless communications protocols are being processed. Most of processing power is consumed in well-structured digital signal processing. Those tasks are pre-scheduled and executed in well defined sequences. A few unexpected interrupts are generated by the inputs from user interfaces which are relatively infrequent. Multi-tasking arrangements are straightforward and less critical.

2.3.6 Large Internal Dynamic Range, but Low Output Resolutions

In order to achieve better performance in signal quality for extended link distances, modern algorithms in wireless communications are complex and the signal data span an incredibly large dynamic range. However, as mentioned before, the frequency bandwidth allocated for wireless communications is limited. In order to transmit more information through the limited frequency bandwidth, wireless communications signals are always compressed intensively. The output data are truncated to very few bits and have very low accuracy. These two conflicting properties require special consideration in the computational units. Some routines need extended data resolution. This implies longer word lengths. A few bits might be suitable for some other routines where data resolution is not critical. Multiple data resolutions are highly desired.

2.3.7 Input / Output Intensive and Heavy Internal Data Flows

Wireless communications processing concentrates on data streams processed from source coding to channel coding, or from channel decoding to source decoding within certain time slots. Large data flows are processed and transferred from memory to memory constantly. This generates significant traffic on internal and global data buses. That demands careful arrangements of data flows. Operand forwarding and reuse become very

important. It is also beneficial to provide several data input and output channels and operate them in parallel at lower speed.

2.3.8 One Time Programming Software

Application software for wireless communications mobile units is different from software for more general applications. The software routines in mobile units are frequently hand coded and highly optimized to reduce power consumption. These codes are written once and stored into millions of mobile units. Even though more routines may be added to update the protocols, existing codes are generally reused again and again. Wireless mobile unit codes are seldom modified by end users. Therefore, it is worth the effort to tradeoff software programming complexity to gain better energy efficiency.

2.4 The Need for New Technologies

As discussed in previous sections, most technologies used in programmable processor architectures have been focused on fast arithmetic operations. The instruction sets for those architectures have centered on the efficient utilization of computational units. On the other hand, mobile wireless communications applications are just-in-time processes, where power consumption instead of speed is the primary concern. The applications also center on a data stream being processed through different stages of the wireless communications protocols. Obviously, the existing processor architectures, whether microprocessors or digital signal processors, are not optimized for mobile wireless communications.

Furthermore, existing processor architectures do not take advantage of the inherent characteristics of wireless communications processing such as determinism and repetition. The execution of operations is currently scheduled dynamically. This consumes more power and increases hardware complexity. In wireless communications,

the same level of performance can be achieved by static instruction scheduling that is pre-arranged and well optimized by software. It is also not necessary to fix hardware configurations before knowing the details of applications to be executed. Configurations can be changed frequently in real time according to the needs of the application routines.

In order to reduce the power consumption of programmable processors in mobile wireless communications, new technologies are needed to take advantage of the inherent characteristics of wireless communications. The paradigm needs to be shifted to energy efficiency from performance in data throughput. This will permit programmable processors to be used to retain the great advantage of flexibility to handle multiple wireless standards without significant sacrificing of energy efficiency.

2.5 Summary

Programmability is highly desirable in multiple standard mobile wireless communications processing. Programmability can make a single mobile unit compatible with many wireless communications protocols and insure that it is upgradeable for future modifications. Programmable processors, such as microprocessors and digital signal processors, provide great programmability. Using these processors makes multiple standard communications easier by shifting the complexity from hardware to software. However, relatively high power consumption prevents us from using those processors in many mobile applications.

In this chapter, we discuss the evolution of programmable processors and major factors that distinguish processor architectures. We also describe the inherent characteristics of wireless communications processing. By comparing the inherent characteristics with the criteria of past innovations, it becomes clear that processor architectures need to be changed to optimize them for the environment of mobile wireless communications.

In the following chapters, we present some technologies that can be used to lower the power consumption of programmable processors. These innovations mitigate the tradeoff between flexibility and power consumption for wireless applications and make it possible to use programmable processors in multiple standard mobile wireless communications transceivers.

Chapter 3

Reconfigurable Macro-Operation Signal Processing Architecture

Most microprocessors and digital signal processors have three types of computational units: multiplier, ALU, and shifter. These units provide basic micro-operations such as addition and multiplication. When optimizing power consumption, we would like to combine those micro-operations in a particular way to create a single macro-operation. This can reduce significantly the amount of effort required in instruction decoding, register reading and writing, and operation controlling. However, different software routines need different kinds of macro-operations. Some current microprocessors and digital signal processors have a few fixed macro-operations optimized specifically for only a few very limited applications. In this chapter, we present a new low power reconfigurable macro-operation architecture that can be reconfigured by software to provide many different optimized macro-operations. This novel architecture is illustrated in this chapter using one multiplier, two ALU's, and one shifter. When advantageous, an output from one unit can serve as an input to other unit(s). This option can be assigned by software programs. Therefore, the example multiplier, ALU's, and shifter can operate either dependently or independently. All of their outputs also can be final outputs, regardless of whether they are fed back or not. Using this scheme, we generate many different combinations having different operational parallelism and different pipeline depth. The hardware configuration is transparent to software programs and can be

modified in real time to meet the evolving requirements of specific applications. This architecture can be used in multiple standard wireless communications processing to improve energy efficiency significantly.

3.1 Reconfigurable Macro-Operation Concept

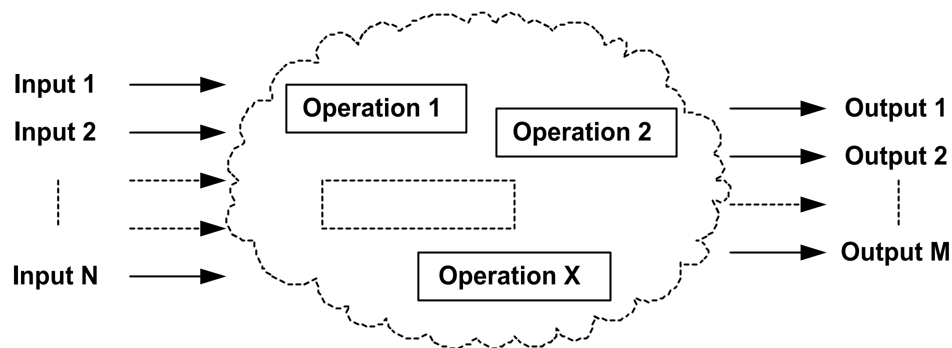


Figure 3 - Illustration of a combination of operations. Given several operation sources, we can generate several outputs with given inputs. Each output is a function of some or all inputs. All these outputs create a very sophisticated set of function groups.

As illustrated in Figure 3, we can have several inputs, which are not necessarily synchronous in time, and several operation resources. Within a particular time period, the inputs are processed by the operation resources to generate several outputs, which are not necessarily synchronous in time either. Each output is determined by a function of some or all inputs. All these outputs create a set of function groups. Even with only a few operation resources, the set of function groups can be very complicated. If the function group can be determined and changed by software in real time, many macro-operations can be provided. The software can choose a macro-operation which is most optimized for the algorithm currently running. When the routine changes, other macro-operations can be configured to meet the new requirements.

For example, assume we have four operations units named as *op1*, *op2*, *op3*, *op4*. With these units, we can execute four micro-operations independently, as illustrated by the following.

op1 (a , b) = W

op2 (c , d) = X

op3 (e , f) = Y

op4 (g , h) = Z

As described previously, we would like to combine several micro-operations into a single macro-operation with one or more outputs. In order to combine micro-operations, each micro-operation output (W, X, Y, Z) needs to be able to be fed back to another micro-operation(s) as input. In other words, every input (a, b, c, d, e, f, g, h) can be either new data or one of the (W, X, Y, Z). At the same time, any or all of the (W, X, Y, Z) can be final outputs, regardless of whether they are fed back or not. Using this scheme, we can generate many different combinations, with different numbers of final outputs and different pipeline depths. The hardware configuration is transparent to software programs. Therefore, software programs can optimize the hardware for specific applications. Several examples of macro-operations, that can be supported easily by these four micro-operations, are illustrated in the following:

Macro-Operation 1: Output 1 = *op4* (*op1* (*op2* (c , d), *op3* (e , f)) , h)

Macro-Operation 2: Output 1 = *op1* (a , *op4* (g , h))
Output 2 = *op3* (*op2* (c , d) , f)

Macro-Operation 3: Output 1 = *op1* (*op2* (c , d) , b)
Output 2 = *op4* (*op2* (c , d) , h)
Output 3 = *op3* (e , f)

Macro-Operation 4: Output 1 = *op1* (a , b)

Output 2 = *op2* (*op1* (a , b) , c)

Output 3 = *op4* (g , *op3* (e , f))

As illustrated above, instead of issuing several micro-operations and moving operands around, we can issue a single macro-operation, for example, Macro-Operation 1, and provide the necessary inputs. When the routine changes, we can issue another macro-operation, for example, Macro-Operation 2, if it will improve the performance. The repetition property of wireless communications processing gives us the opportunity to use the same macro-operation many times repetitively in a same routine. Therefore, the overhead from configuring a new macro-operation has very much less impact on the overall performance in wireless communications applications than it would in other real-time applications that do not include much repetition.

3.1.1 The Challenges

It is not difficult to just combine several micro-operations into a single macro-operation. This can be like a bundle of micro-operations issued in serial. However, in that case, the code density is not increased. The parallelism and data transfer scheme would not be improved and the power consumption would not be reduced. There are several challenges that must be met to make the concept of reconfigurable macro-operations practical. These are:

- How to make the datapath reconfigurable in real time?
- How to transfer the operands efficiently?
- How to control the whole operation efficiently?
- How to minimize power consumption with all the above?

As mentioned before, power consumption is the primary factor being optimized in mobile wireless communications while maintaining sufficient flexibility to process

signals for different standards. On the other hand, low system complexity is also highly desirable for the compactness of mobile units. In this research, we invented a novel architecture that overcomes the challenges listed above, but still keeps the complexity low. The architecture is discussed in the following sections.

3.2 Proposed Architecture

Figure 4 shows a block diagram of the proposed example of the architecture with the four operations noted earlier. It includes operation controlling blocks, operand transferring blocks, computing blocks, and other supporting structures. The reconfiguration feature is achieved by a special arrangement of the data feedback path and control mechanism included in the macro-operation instruction format. As is described later in Section 3.2.2, a single compact macro-operation provides all information needed for hardware configurations, data movements, operation timing, loop control, etc. The code density is increased significantly. Operands are also transferred more efficiently. An instruction register file is also included in the architecture to store decoded macro-operations. If a macro-operation is needed that is the same type as is stored in this file, the corresponding information can be fetched from the instruction register in which it is stored. This avoids fully decoding an issued macro-operation which is used many times within a routine.

The functionality of each block and the macro-operation instruction format are explained in the following subsections of section 3.2. The operating procedure and different operation modes are explained in Section 3.3.

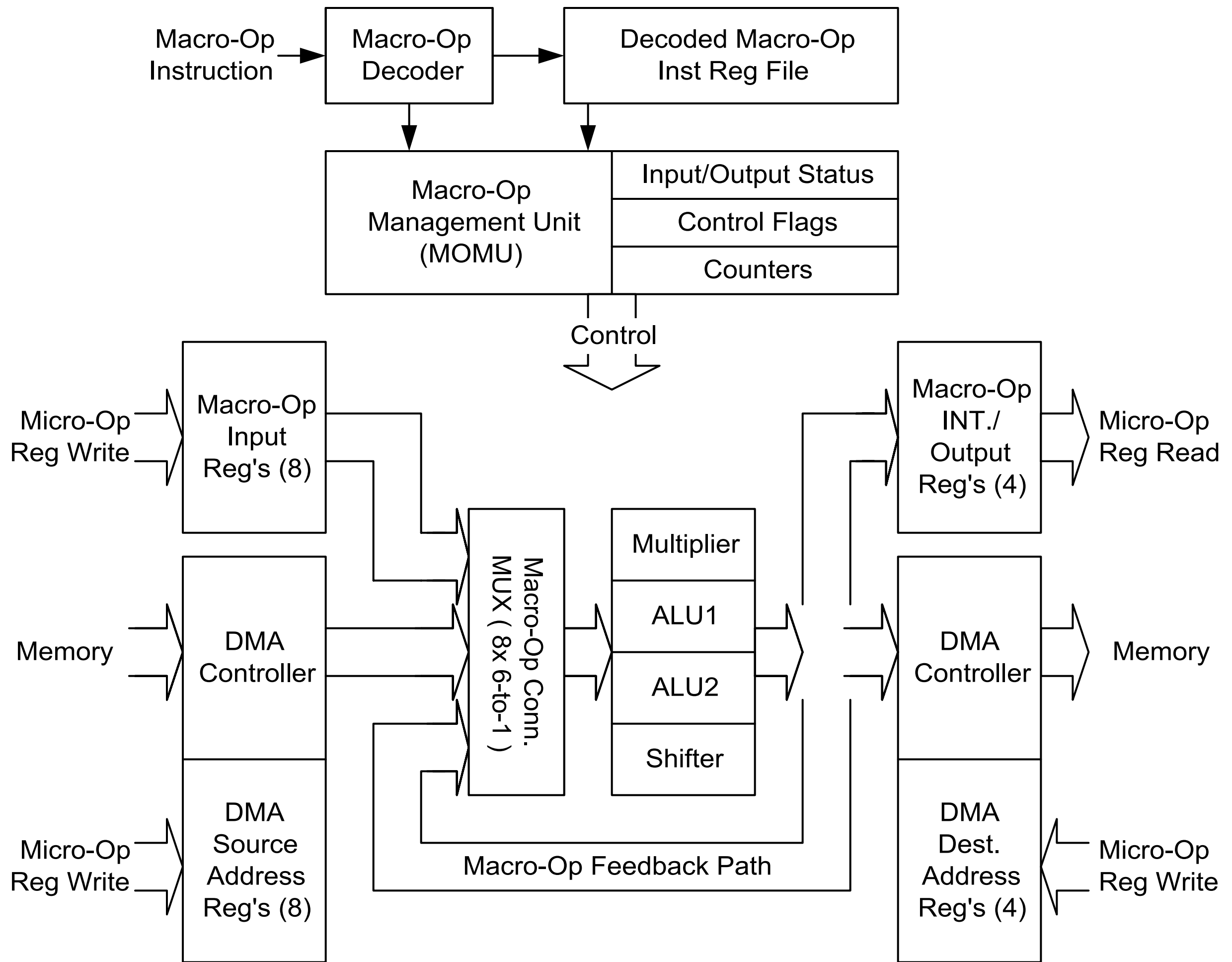


Figure 4 - Reconfigurable macro-operation signal processing architecture. The building blocks include macro-operation instruction decoder, decoded macro-operation instruction register file, macro-operation management unit (MOMU), computational units, macro-operation connection multiplexer, macro-operation input registers, macro-operation output/interrupt registers, 8-channel memory loading DMA controller, 4-channel memory storing DMA controller, DMA source address registers, and DMA destination address registers.

3.2.1 Block Description

This section describes each function block in the reconfigurable macro-operation architecture shown in Figure 4.

- *Macro-Operation Instruction Decoder*: decodes issued macro-operations.

- *Decoded Macro-Operation Instruction Register File*: stores decoded controlling and configuring information of issued macro-operations. If the same type of macro-operation is issued again, the controlling and configuring signals can be fetched from the instruction register in which it is stored. This eliminates the effort of decoding a full macro-operation instruction and, therefore, reduces power consumption. In wireless communications processing, we use only a few different kinds of macro-operations repeatedly in each software routine. Therefore, using this register file, we take advantage of the repetition property of wireless communications processing. A hardware configuration can be stored and reused without redundant decoding.

- *Macro-Operation Management Unit (MOMU)*: controls the whole operating sequence of a macro-operation. It contains input status registers, output status registers, control flags, and counters. These registers, flags, and counters record the status of macro-operation executions. The MOMU supervises the operation timing and coordinates all units. The details of the flow of control are discussed later in this chapter.

- *Computational Units*: the example includes one multiplier, one shifter, and two ALU's. The second ALU unit is added to support better optimization of macro-operations, with minimum hardware complexity tradeoff for the environment of multiple standard wireless communications processing. With these four computational units, we can execute four micro-operations dependently or independently. In order to combine several micro-operations into a single macro-operation with one or more outputs, each micro-operation output can be fed back to another micro-operation(s) as input. Therefore, there are three separate sources of inputs to these computational units: macro-operation input registers, direct memory access (DMA) channels, and fed-back intermediate results. These are selected by the

macro-operation connection multiplexer, which is discussed in the following paragraph. The computational units, if not used, can be turned off to save power.

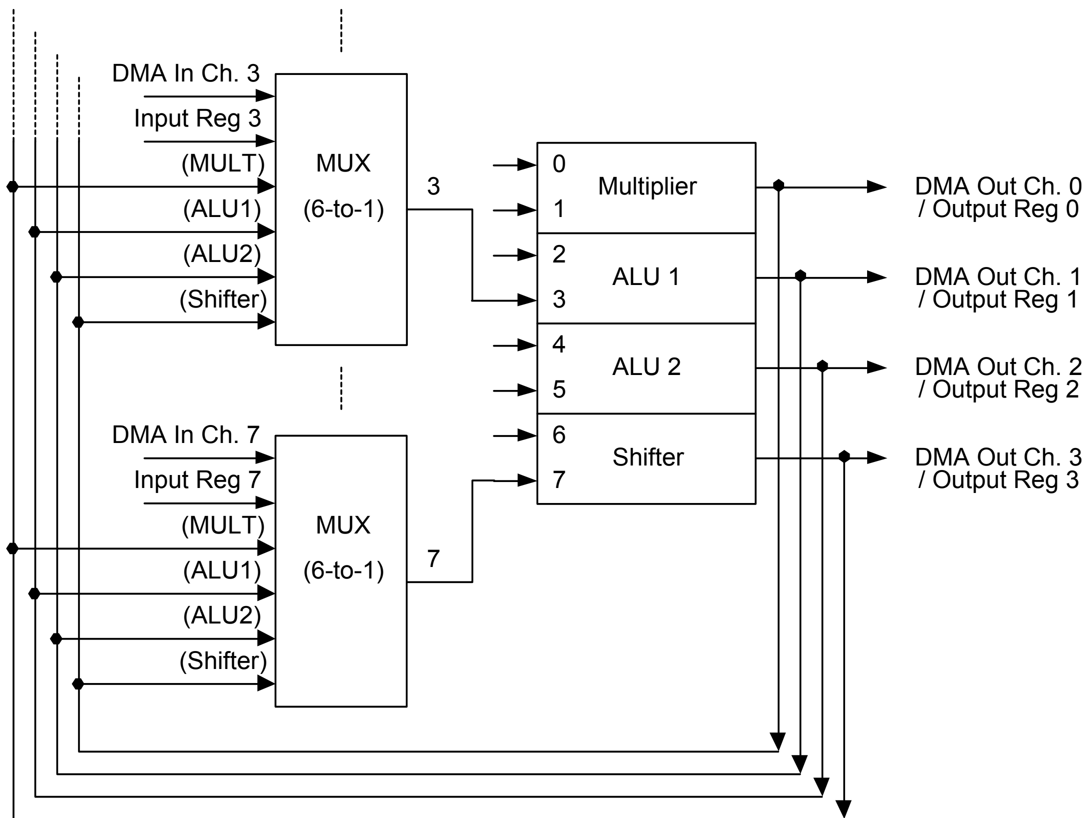


Figure 5 - Illustration of macro-operation connection multiplexer. There are a total of eight inputs to the computational units. These are numbered from 0 to 8. Each input has six choices: four from the feedback bath, one from DMA, and one from an input register. The assignment of DMA and register inputs to the computational units are fixed. Operands from different DMA or input registers are not exchangeable.

- *Macro-Operation Connection Multiplexer*: determines the configuration of the macro-operation datapath. As illustrated in Figure 5, there are a total of eight inputs to the computational units. There are six choices for each input. Four are from the feedback path. The other two are from the corresponding DMA input channel and

macro-operation input register. Therefore, DMA channels are not interchangeable. The date from the first DMA input channel, if selected, always goes to the multiplier. It is the same for macro-operation input registers. This scheme reduces the complexity of the macro-operation connection multiplexer, the DMA controllers, and those macro-operation registers. The property of determinism in the wireless communications processing gives us the opportunity to pre-arrange parameters and stored data for the right macro-operation registers or DMA input channels. Therefore, this constraint does not reduce the performance or the optimization of macro-operations needed by software routines.

- *Macro-Operation Input Registers*: provide eight inputs to the macro-operation datapath. They can be written into by a micro-op instruction “register-write” as can be done for any general register. However, these registers cannot be read by software. They provide temporary data storage for constant parameters for a macro-operation. They are not changed while macro-operations are executed in loop mode to be described in Section 3.3.

- *Macro-Operation Output/Interrupt Registers*: are similar to the macro-operation input registers above. They are simplified general registers which support register read only. These registers can be assigned to store the output results of corresponding computational units. As shown in Figure 5 they are not interchangeable. While an interrupt is processed, these registers also store intermediate results to avoid any data loss. Those intermediate results are fed into the datapath again after the interrupt processing is completed.

- *8-Channel Memory Loading DMA Controller*: controls the eight DMA input channels providing needed input data from memory. The controller gets control message from the MOMU and turns on the necessary input channels at the right time. The memory locations to be fetched are determined by the values in the DMA source address registers. Again, as shown in Figure 5, the DMA channels are not interchangeable.

The DMA controller and channels are simplified versions of typical DMA blocks. Only memory loading is needed. In practice, not all DMA channels are used at the same time. By using virtual channels, we can reduce real DMA bandwidth significantly without sacrificing overall performance.

- *4-Channel Memory Storing DMA Controller*: controls the four DMA output channels providing needed data output to memory. They are identical to the DMA input channels above, except they operate in an opposite direction to store data into memory. The memory locations to be stored into are determined by the values in the DMA destination address registers.

- *DMA Source Address Registers*: store memory buffer addresses for the memory loading DMA controller discussed below. Like macro-operation input registers, they are write-only. Each address register provides a source address for the corresponding DMA input channel. Depending on the macro-operation instruction, the source address is either increased or decreased by one per DMA access.

- *DMA Destination Address Registers*: are similar to DMA source address registers. Each address register provides a destination address for the corresponding DMA output channel. The destination address is increased by one per DMA access.

3.2.2 Macro-Operation Instruction Format

Figure 6 shows an example of the macro-operation instruction format for the architecture described in the previous sections. It is 10-bytes long. The first byte [79:72] identifies a macro-operation. This byte is not passed to the macro-operation instruction decoder. All information needed for a macro-operation is provided in the remaining nine bytes [71:0]. The lower six bytes are ignored if a stored macro-operation is called for. The bit fields and their functionality are explained in the following.

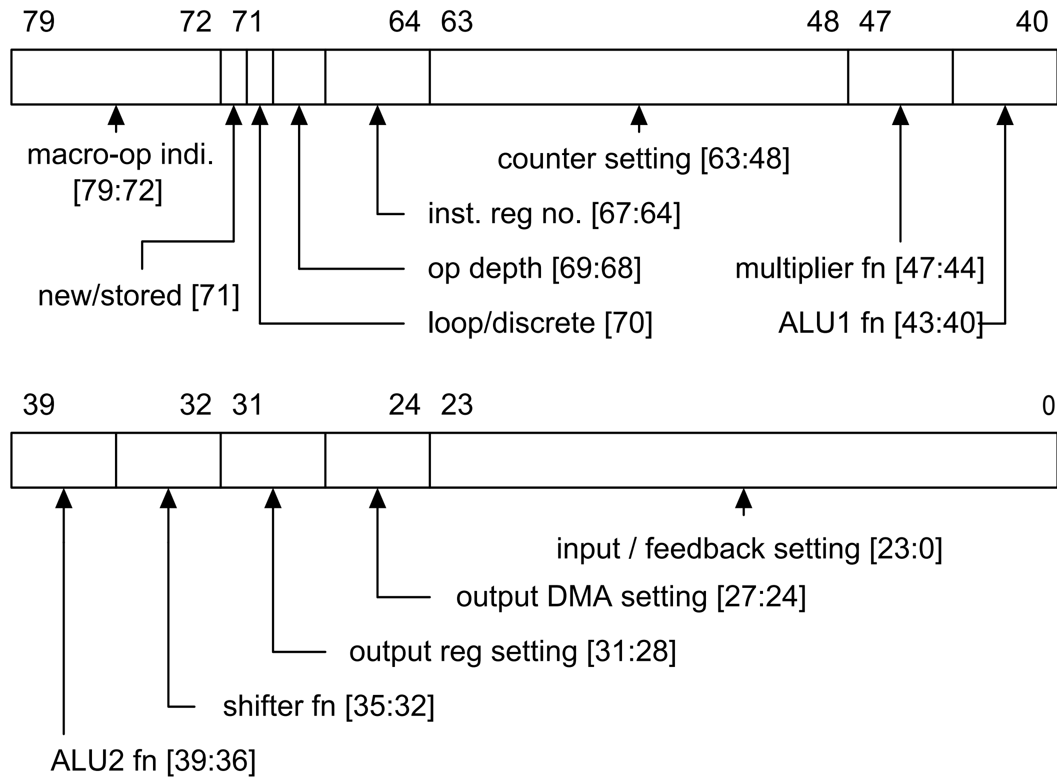


Figure 6 - Macro-operation instruction format that is 10-bytes long. The higher 4 bytes are for basic control and are always decoded. The lower 6 bytes are the detailed information and are ignored if a stored macro-operation is called for. The bit fields are: macro-operation indication [79:72], flag 1 - new/stored instruction [71], flag2 - discrete/loop mode [70], operation depth [69:68], instruction register number [67:64], counter setting [63:48], multiplier function [47:44], ALU 1 function [43:40], ALU 2 function [39:36], shifter function [35:32], output register setting [31:28], output DMA setting [27:24], and input/feedback setting [23:0].

- *Macro-Operation Indication [79:72]*: is an indication to the program decoder, telling that the issued instruction is a macro-operation. It is not passed to the macro-operation instruction decoder.

- *Flag 1 - New/Stored Instruction [71]*: tells the MOMU if it is a new type of macro-operation or the same type of macro-operation stored in one of the macro-operation instruction registers.

- *Flag2 - Discrete/Loop Mode [70]*: tells MOMU if it is in discrete mode or in loop mode. Discrete and loop modes are described in Section 3.3.2.

- *Operation Depth [69:68]*: indicates how many stages the whole macro-operation takes. Depending on the configuration, each macro-operation can take from one to four micro-op execution cycles.

- *Instruction Register Number [67:64]*: identifies which stored instruction to use if Flag 1 indicates using a stored type of macro-operation. If Flag 1 indicates a new type of macro-operation, this field tells in which instruction register the new macro-operation should be stored.

- *Counter Setting [63:48]*: sets the counter to the execution count needed.

- *Multiplier Function [47:44]*: sets the multiplier function; ignored if a stored macro-operation is used.

- *ALU 1 Function [43:40]*: sets the first ALU function; ignored if a stored macro-operation is used.

- *ALU 2 Function [39:36]*: sets the second ALU function; ignored if a stored macro-operation is used.

- *Shifter Function [35:32]*: sets the shifter function; ignored if a stored macro-operation is used.

- *Output Register Setting [31:28]*: in discrete mode, sets which macro-operation output registers should be enabled at the end of execution; in loop mode, sets which macro-operation output registers should be enabled at the end of the whole loop; ignored if a stored macro-operation is used.

- *Output DMA Setting [27:24]*: sets which DMA output channels should be enabled at the end of each execution; ignored if a stored macro-operation is used.

- *Input/Feedback Setting [23:0]*: sets the configuration of macro-operation connection multiplexer. These bytes determine which inputs should be used and how intermediate results are fed back. There are eight independent settings. Each of them sets the inputs for each computational unit. If the DMA input channel is used, these bytes also tell the DMA to increase or decrease the source memory address by one per DMA access. If not used, a computational unit can also be disabled to eliminate its power consumption during that cycle. These bytes are ignored if a stored macro-operation is used.

3.3 Operations

The proposed reconfigurable macro-operation signal processing architecture not only packs several micro-operations into a compact macro-operation, but also provides an intelligent operation flow control. In addition to input and output configurations, it provides several operation modes to software programs. A software program can dispatch a new macro-operation or an old macro-operation stored in a macro-operation instruction register. Macro-operations can also work alone or in conjunction with micro-operations to execute arithmetic functions. No matter what mode is utilized, a software program can always use the counter setting inside a macro-operation to control the number of executions. This section discusses the operation procedure and different operation modes of the proposed architecture.

3.3.1 Flow Chart

Figure 7 shows the procedure flow chart for the reconfigurable macro-operation architecture. The procedure flow is mainly controlled by the macro-operation management unit (MOMU). Flow begins at block 1 where we determine if the issued instruction is a macro-operation or micro-operation. If it is a micro-operation, it follows the normal procedure for a micro-operation, which is not described in this dissertation. If it is a macro-operation, flow proceeds from block 1 to block 2 where we separate macro-operations into two modes: loop mode and discrete mode. The usage of different modes is discussed later.

If it is loop mode, we start from block 3 and determine if the instruction is new. If it is new, the full instruction is decoded (block 4) and stored in the decoded macro-operation instruction register file (block 5). The MOMU then reconfigures the data path according to the information decoded (block 9). However, if the instruction has already been stored, flow proceeds from block 3 to block 6, where the MOMU retrieves the status of the previously executed macro-operation (block 6) and determines whether it is the same as, or different from, the current one (block 7). If it is different, flow proceeds to block 8, where the stored macro-operation information is retrieved from a macro-operation instruction register, and then to block 9 for reconfiguration of the datapath.

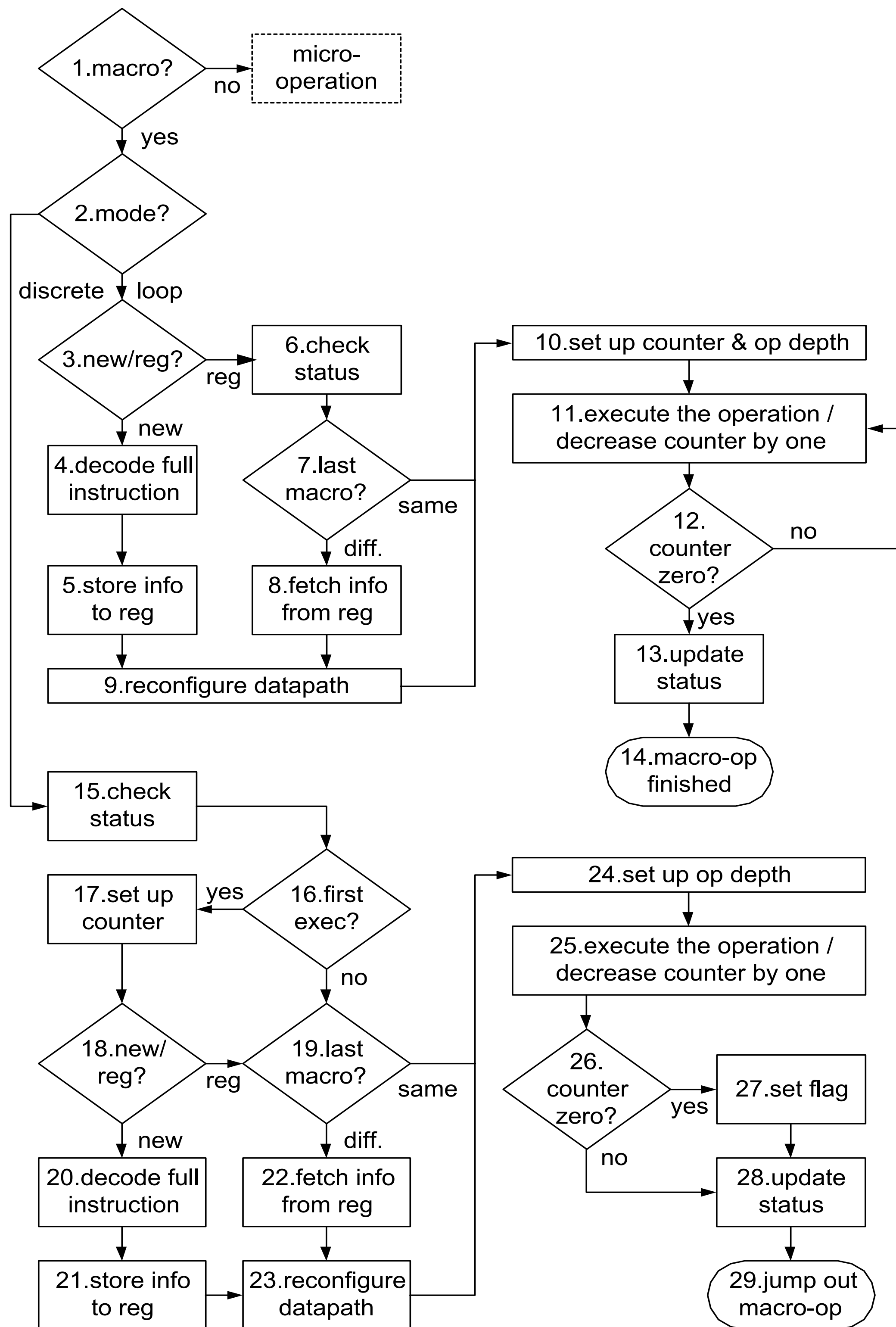


Figure 7 - Macro-operation flow chart. Flow begins at block 1 and ends at either block 14 or block 29, depending on which mode is run.

From either block 9 or block 7 (for the same macro-operation previously executed), flow proceeds to block 10 through block 14 for completion of the operation. At block 10 and 11, the MOMU configures the counter and the operation depth, and then executes the operation with a decrement of the counter upon completion. At block 12, the counter is compared to zero. If the counter has not reached zero, flow returns to block 11. If the counter has reached zero, flow proceeds from block 12 to block 13, where the MOMU updates the status of the operation. At block 14, the macro-operation is complete.

If it is determined to be discrete mode at block 2, flow proceeds to block 15 to check the status recorded in the MOMU. Unlike in loop mode, the discrete mode sets up the counter (block 17) immediately after determining that this is the first execution of the macro-operation at block 16. Then flow proceeds from block 17 to block 18. If this is not the first execution, flow proceeds from block 16 to block 19. The procedures at block 18, 19, 20, 21, 22, and 23 are similar to those at block 3, 4, 5, 7, 8, and 9 in loop mode. At these blocks, we check if the instruction is new or if it is the same as the macro-operation previously executed. Depending on the results we either decode the full instruction and store the decoded information in a register, or fetch stored information. Then we reconfigure the datapath.

From block 23 or block 19 (for the same macro-operation previously executed), flow proceeds to block 24, where the operation depth is configured. From block 24, flow proceeds to block 25 for execution and counter decrement, and then to block 26 where the counter is compared to zero. If the counter has not reached zero, flow proceeds to block 28 to update the status of the operation. If the counter has reached zero, flow proceeds from block 26 to block 27, where a zero flag is set to record this condition, and then to block 28. At block 29, the execution jumps out of the issued macro-operation to the instruction following the executed macro-operation instruction.

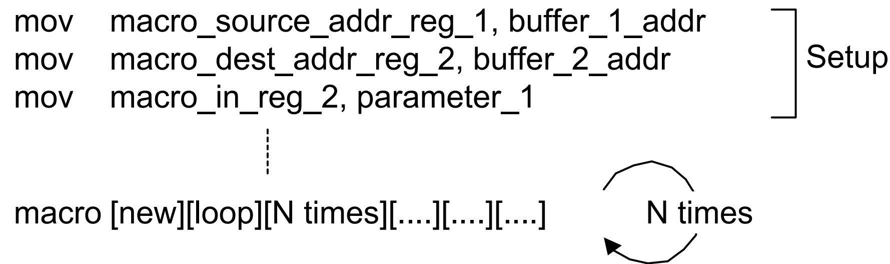
3.3.2 Usage of Different Operation Modes

As discussed previously, macro-operations can be categorized into four different operation modes: new-instruction loop, stored-instruction loop, new-instruction discrete, and stored-instruction discrete modes. These modes are indicated by the two instruction flag bits. The difference between new and stored instruction modes is whether or not to use the decoded control information stored in a macro-operation instruction register. If it is a new instruction, the decoded information will be automatically stored into a macro-operation instruction register indicated by instruction bits [67:64]. If it is not a new instruction, the lower six bytes of the instruction are ignored and the stored information is used instead. The difference between loop and discrete modes is whether or not to execute a macro-operation many times before the program counter (PC) proceeds to the instruction that follows the issued macro-operation. In loop modes, the macro-operation is executed repeatedly until the macro-operation counter inside the MOMU is decreased to zero. The outputs of each execution are stored into memory by the DMA units while the macro-operation is being executed repetitively. The outputs of the last execution are stored into macro-operation output registers or memory. In discrete modes, the macro-operation is executed only once and the program counter (PC) proceeds to the next instruction. When a discrete-mode macro-operation is mixed with several micro-operations in a loop, the macro-operation counter can be used to control the program flow. The macro-operation counter is set at the first execution, which is indicated by the MOMU control flags. The counter is decreased by one per execution of the same macro-operation. When the counter drops to zero, it sets the counter-zero-flag inside MOMU to one. A micro-operation is used to check the counter-zero-flag and set the direction of the program flow. This eliminates the need to use an additional variable to control the program flow of a loop. The following illustrates how different modes can be used to implement the algorithms required.

3.3.1.1 New-Instruction Loop Mode

In new-instruction loop mode, the macro-operation is fully decoded to configure the datapath and loop count. This single macro-operation is executed repeatedly for the number of times determined by the loop count. This instruction is not mixed with micro-operations while being executed. This mode can be used in a small isolated task, which requires only a few operations in its continuous processing. A few set-up operations are needed to arrange input and output memory blocks. Some fixed parameters are also stored into corresponding macro-operation input registers in the set-up phase. The macro-operation fetches input data directly from main memory. It stores output results to either main memory after each execution or to macro-operation output registers after the whole loop is finished. The storage location depends on the setting indicated by instruction bits [31:24]. An illustration of possible code structure is shown in the following.

Illustration of a possible structure for new-instruction loop mode:



As illustrated above, we set up the necessary DMA source and destination addresses before we execute a macro-operation. We also move a parameter to the macro-operation input registers. These setup instructions provide necessary information to the macro-operation. When a new-instruction-loop-mode macro-operation is issued, the macro-operation is fully decoded and executed N times continuously as indicated by the counter setting inside the macro-operation.

3.3.1.2 Stored-Instruction Loop Mode

Stored-instruction loop mode is similar to new-instruction loop mode. The macro-operation is executed many times without being mixed with micro-operations. The

purpose of having this mode is to process the same sequence of operations on many different sets of data in sequence. Once a macro-operation is fully decoded and stored in the instruction register file, the same type of macro-operation can be executed on different data with the same or different loop count. The addresses for input and output and the loop count can be changed for each data set. Compared with the stored-instruction discrete mode to be discussed below, stored-instruction loop mode will be applied to many sets of data which are not stored in continuous memory space. An illustration of possible code structure is shown in the following.

Illustration of a possible structure for stored-instruction loop mode:

```
macro [new][loop][N times][inst reg 1][....][....]  <---- Previous Execution
        :
mov   macro_source_addr_reg_1, buffer_3_addr   ]
mov   macro_dest_addr_reg_2, buffer_4_addr     ]  Setup
mov   macro_in_reg_2, parameter_4              ]
        :
macro [stored][loop][M times][inst reg 1]         M times
```

As illustrated above, two macro-operations are executed. The first macro-operation is in new-instruction loop mode. It is fully decoded and the decoded information is stored into the macro-instruction register number one. After a few cycles, we would like to execute the same type of macro-operation. We can issue a stored-instruction-loop-mode macro-operation without redundant decoding of the full instruction. However, we would like execute the macro-operation on different set of data with different values in the macro-operation input registers. Therefore, before we execute the second macro-operation, we need to set up the DMA source and destination addresses again. We also move a different parameter into the macro-operation input register

number 2. After that, the second macro-operation is partially decoded and executed M times continuously as indicated by the counter setting inside the macro-operation.

3.3.1.3 New-Instruction and Stored-Instruction Discrete Modes

As the word "discrete" suggests, a discrete mode macro-operation is executed only once while the instruction is being pointed to by the program counter. Similar to loop modes, we can issue a new macro-operation or use the information stored in a macro-operation instruction register in discrete mode. An illustration of possible code structure for new-instruction discrete mode is shown in the following. The code structure for stored-instruction discrete mode is similar except the same type of macro-operation was executed previously and stored into a macro-operation instruction. This concept has been illustrated in Section 3.3.1.2.

Illustration of a possible special structure for discrete mode:

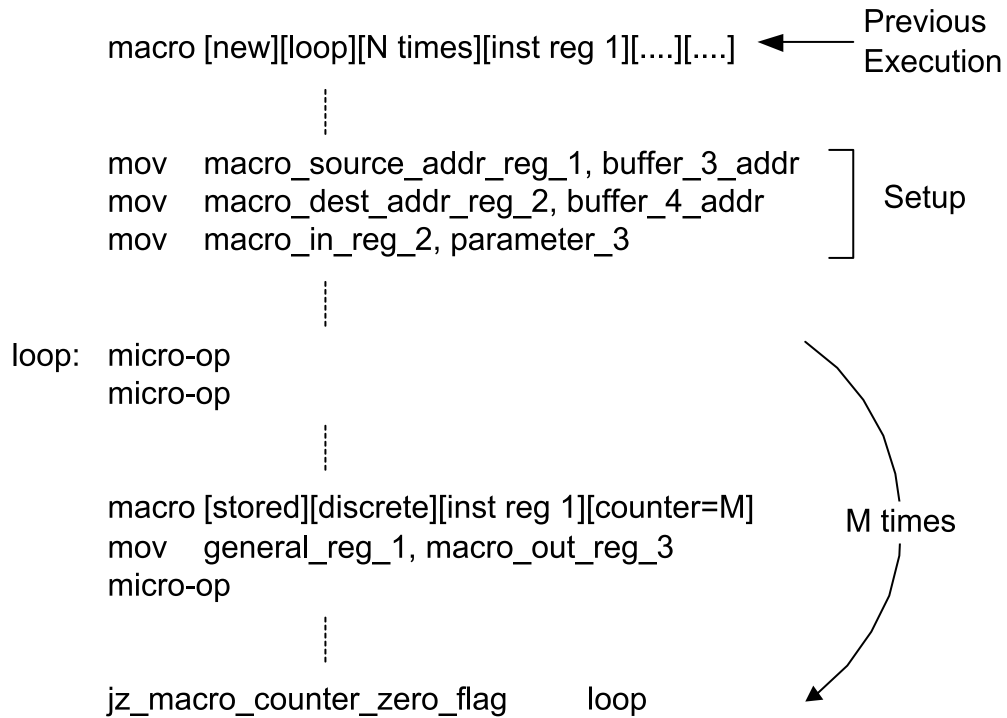
```
micro-op
micro-op
mov    macro_source_addr_reg_1, buffer_1_addr   ┐
mov    macro_dest_addr_reg_2, buffer_2_addr     │ Setup
mov    macro_in_reg_2, parameter_1              ┘
micro-op
   ⋮
micro-op
macro [new][discrete][counter ignored][....][....][....]  ◄── Execution
mov    general_reg_1, macro_out_reg_3
micro-op
micro-op
```

As illustrated above, we first set up the information needed for the macro-operation. We move the memory addresses of buffer 1 and buffer 2 into the DMA source and destination address registers separately. We also move parameter 1 into the macro-operation input register number 2. Then, the macro-operation is decoded and executed

only once no matter what value the counter is set to. Then, we use a micro-operation to move the result in the macro-operation output register number 3 into a general register.

Although a macro-operation can be executed only once in the whole routine as discussed above, the main reason for having discrete modes is to be able to mix macro-operations and micro-operations in a loop. This provides more operation choices and enhances the optimization. While being mixed with micro-operations in a loop, a macro-operation is, in fact, executed many times. It is executed once every time around the loop. The loop counter is "embedded" in the macro-operation and decreased by one per macro-operation execution as discussed in the procedure flow chart. A micro-operation conditional jump is used to check the counter status and controls the program flow direction. In this case, there is no difference between new-instruction discrete mode and stored-instruction discrete mode except in the first execution of the macro-operation. In the first execution, the macro-operation is either fully or partially decoded according to the new/stored instruction flag of the macro-operation. However, in the following execution(s) of the same macro-operation, the MOMU overrules the new/stored flag and follows the procedure defined in Figure 7. As shown in the flow chart, in discrete mode macro-operation, the MOMU always checks to see if it is the first time that the macro-operation is being executed to determine the next step. The new/stored instruction flag is ignored after the first execution. The purpose is to improve the overall performance when macro-operations and micro-operations are mixed in a same loop. The following illustrates a possible code structure of mixing a macro-operations and several micro-operations in a loop. Although the macro-operation in the loop is listed as stored-instruction discrete mode, it could be new-instruction discrete mode. If it is the new-instruction discrete mode, the macro-operation is fully decoded in the first pass through the loop. In the following passes through the loop, the MOMU overrules the flag and the macro-operation executes like stored-instruction discrete mode.

Illustration of a typical structure for discrete mode:



As illustrated above, the same type of macro-operation that is needed in the loop was executed previously. The configuration information was stored into a macro-operation instruction register. As before, we need to set up the information that is different from the previous macro-operation, such as the memory locations of input data. When the loop starts, several micro-operations are executed first. Then, we issue a stored-instruction-discrete-mode macro-operation. The information needed is fetched from the macro-operation instruction register. The macro-operation is executed once and then the program proceeds to the next instruction, a micro-operation move. The macro-operation counter associated with this macro-operation is decreased by one from the initial value M. The micro-operation following the macro-operation stores the value of macro-operation output register number 3 into a general register for micro-operation usage. Several micro-operations are then executed. At the end of the loop, we use a micro-operation conditional jump to jump to the beginning of loop since the macro-operation counter has not been reduced to zero. The loop is executed the second time. The macro-operation counter is decreased by one. The conditional jump at the end of the

loop directs the program flow to the beginning of the loop again. The same process repeats until the macro-operation counter is reduced to zero. When the counter reaches zero, the counter-zero-flag inside the MOMU is set to one. The loop is finished. The conditional jump instruction detects this change and directs the program flow forward to the next instruction following the whole loop.

One special issue should be noted before we conclude this section. While the MOMU checks if a discrete mode macro-operation is be being executed the first time, the decision is really made by checking if the counter associated with the instruction register indicated by instruction bits [67:64] is zero while the macro-operation is being issued. If the counter is zero, the macro-operation previously issued has been finished. A zero counter means it must be the start of another sequence of operations. If the counter is not zero, it means there must be a macro-operation that is still using this counter. A non-zero counter indicates it is not the first execution. If more than one macro-operation is utilized in a same loop, we must keep their register numbers different. Otherwise, the same counter would be used by different macro-operations. Unless this is arranged intentionally, it would cause incorrect executions. However, this controlling scheme is made transparent to software on purpose. In some cases, it is beneficial for several macro-operations in a loop to share the same counter. Special caution is needed in such situations.

3.4 Examples

Two examples are discussed in the following to illustrate loop mode and discrete mode operations. The first example, symmetric finite impulse response filter, demonstrates loop mode operations. The second example, adaptive finite impulse response filter, demonstrates discrete mode operations.

3.4.1 Symmetric FIR Filter

Equation 3.1: $$y(n) = \sum_{k=0}^{\frac{N}{2}-1} h(k)[x(n-k) + x(n-N+1+k)]$$

A symmetric finite impulse response (FIR) filter is widely used in wireless communications because of its linear phase response [Lee97]. Equation 3.1 represents a symmetric FIR filter. Implementation of the equation is straightforward using a single loop-mode macro-operation. The hardware configuration is shown in Figure 8. The pseudo-code is listed in the following, where the numbering is referred to Figure 5.

Pseudo-Code:

```
setup:  mov     macro_source_addr_reg_2, addr_x(n)
        mov     macro_source_addr_reg_3, addr_x(n-N+1)
        mov     macro_source_addr_reg_1, addr_h(0)
        mov     macro_in_reg_7, resolution_adjust
        macro:  macro  [new][loop][counter=(N/2)][setting as in Figure 8]
        result: mov    result_y(n), macro_out_reg_2
```

As shown above, the macro needs to set up the corresponding DMA addresses for the input data stream. The routine jumps into a single macro-operation after the set up. The macro-operation is executed N/2 times. The macro-operation increases the DMA source address automatically to fetch following inputs. When it finishes, the output is stored in a macro-operation output register and moved back to main memory by a micro-operation move shown in the last line of the pseudo-codes.

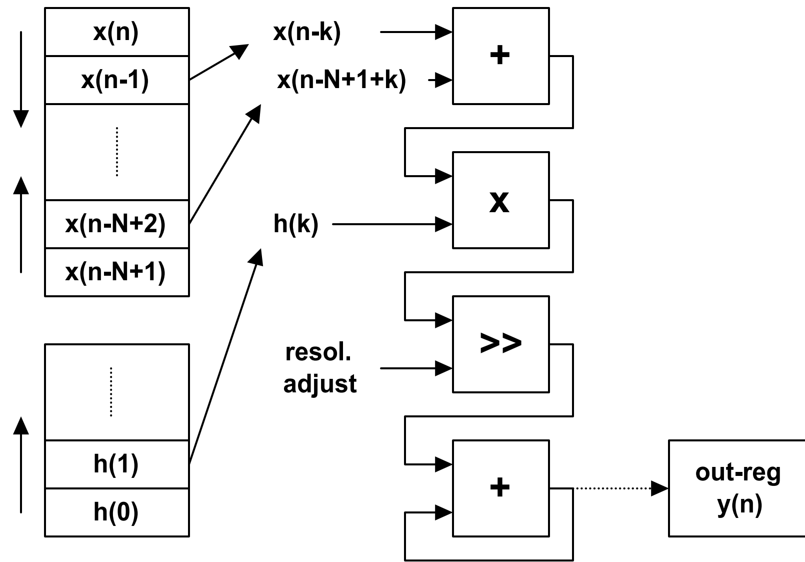


Figure 8 - Hardware configuration for symmetric FIR example. Input data x(n-i) and x(n-N+1+i) are fed to an ALU for the add operation. Filter coefficient h(i) is multiplied by the sum of x(n-i) and x(n-N+1+i). The resolution of the result is adjusted before being added to the accumulation of the previous results. The index i is incremented automatically in the execution of a macro-operation. The final result is stored into a macro-operation output register.

3.4.2 Adaptive FIR Filter

Equation 3.2: $e(n-1) = d(n-1) - y(n-1)$

Equation 3.3: $C_k(n-1) = 2Be(n-1)x(n-1-k)$

Equation 3.4: $b_k(n) = b_k(n-1) + C_k(n-1)$

Equation 3.5: $y(n) = \sum_{k=0}^{N-1} b_k(n)x(n-k)$

An adaptive FIR filter requires extensive computations to update the filter coefficients in real time. Equations 3.2 to 3.5 show the required computations. To compute all equations in sequence, we need to use discrete modes to mix a macro-operation with several micro-

operations in a loop. The hardware configuration is shown in Figure 9. The pseudo-code is listed in the following, where the numbering is again referred to Figure 5.

Pseudo-Code:

```
setup:  mov    macro_source_addr_reg_3, addr_b0(n)
        mov    macro_source_addr_reg_1, addr_x(n)
        mov    macro_source_addr_reg_6, addr_x(n)
        mov    macro_in_reg_7, constant_zero
        mov    macro_dest_addr_reg_1, addr_b0(n)
        sub    e(n-1), d(n-1), y(n-1)
        mult   e(n-1), e(n-1), 2B
        mult   macro_in_reg_2, e(n-1), x(n-1)
loop:
macro:  macro  [new][discrete][counter=N][setting as in Figure 9]
        mult   macro_in_reg_2, e(n-1), macro_out_reg_3
        jn_macro_counter_zero_flag  loop
result: mov    result_y(n), macro_out_reg_2
```

As shown above, similar to the first example, some set up and pre-calculations for operations in the loop are required. In this example, we use discrete mode in order to mix a macro-operation and a few micro-operations in the loop. The macro-operation is executed once every loop. It decreases the counter by one per execution. Since the macro-operation does not finish all needed operations, we use a micro-operation to implement the rest. Then, we use a micro-operation to check if the loop count has dropped to zero. If not, we keep processing the loop. If yes, the sequence of operations is completed and we store the final result back to main memory using a micro-operation move.

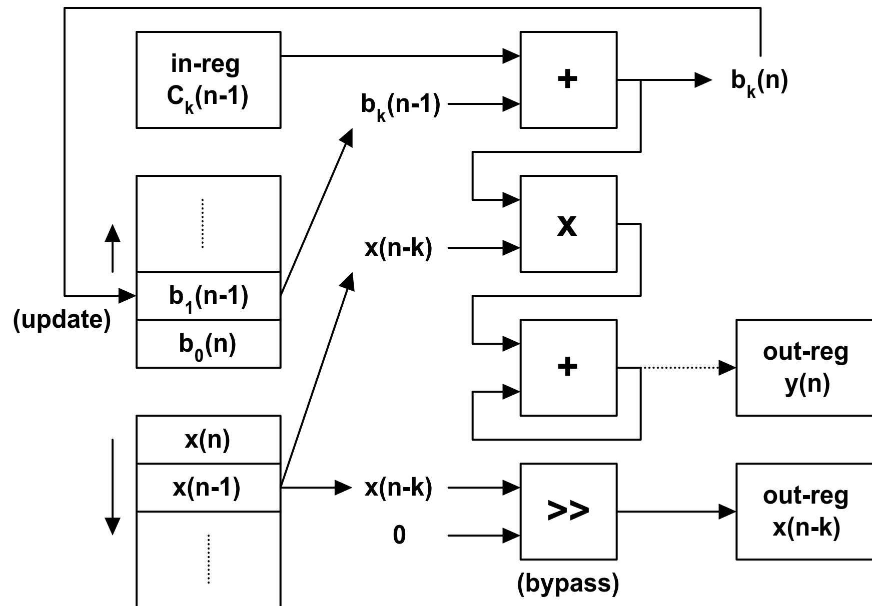


Figure 9 - Hardware configuration for adaptive FIR example. In order to generate y(n), we have to update $B_k(n-1)$ to $B_k(n)$ and, therefore, we have to update $C_k(n-1)$. The multiply-and-accumulate of x(n-k) and $B_k(n)$, and the update of $B_k(n)$ are done in the macro-operation. The update of $C_k(n-1)$ is handled by a micro-operation mixed with the macro-operation in the loop. $B_k(n-1)$ and $C_k(n-1)$ are summed by an ALU to generate the $B_k(n)$, which are fed to the multiplier to be multiplied by x(n-k) and also are stored back into the coefficient array in the memory for updating. The index k is incremented automatically in the execution of a macro-operation. The final result from the multiply-and-accumulate of x(n-k) and $B_k(n)$ is stored into a macro-operation output register.

3.5 Architecture Impacts

The proposed reconfigurable macro-operation signal processing architecture can be used to reduce power consumption in the environment of multiple standard mobile wireless communications. In response to the processor factors discussed in Chapter 2, the impacts on traditional processor architectures are discussed in the following paragraphs. The architecture modeling and performance evaluation are discussed in Chapter 6.

Based on general register methodology, the proposed architecture mixes different operand transfer schemes. Operands are not directly named in the macro-operation

instructions. Instead, they are assigned implicitly in the hardware configurations. Operands can be in registers or memory. Operand forwarding and reuse are arranged directly by software programs. Macro-operation instructions are more CISC-oriented. Instruction length and code density are increased. The instruction efficiency is enhanced without compromising the flexibility. On the other hand, the instruction length of macro-operations is fixed. This eliminates the effort required to find the end of the instructions that is encountered in conventional CISC architectures.

Using macro-operations does not really increase the instruction parallelism. The same level of parallelism can be achieved by issuing several micro-operations in parallel. However, macro-operations reduce the scheduling and decoding efforts significantly. In other words, they achieve the same parallelism with lower power consumption and less system complexity. Macro-operations also prevent possible operand conflicts.

Pipelining is addressed differently in the proposed architecture. Since speed is not the major concern here, it is not necessary to use pipeline architectures on simple operations. The concept of pipelining is used to line up the available resources to create more complicated operations.

Memory hierarchy is not changed by the proposed architecture.

Interrupts are handled by storing the intermediate values in the datapath. The datapath can switch between a primary signal processing task and other control-oriented tasks. In order to retain low system complexity, the interrupting tasks cannot utilize macro-operations. This does not reduce the benefit of introducing macro-operations into the system because the interrupting tasks are usually control-oriented and can be served as well by using micro-operations directly.

The proposed architecture also makes the system more ASIC-oriented. The datapath is customized in real time for specific algorithms. This drives the programmable processor architecture towards ASIC in terms of power consumption.

3.6 Summary

The benefits from reduced-instruction-set-computer (RISC) and complete-instruction-set-computer (CISC) architectures have been debated for a long time. The dilemma between execution speed and memory efficiency is considered seriously in all processor architectures. On the other hand, the choice of programmable processors or ASIC's produces a tradeoff between flexibility and energy efficiency.

This chapter presents an architecture that can be used in programmable processors to reduce the power consumption. A reconfigurable macro-operation signal processing architecture provides a flexible and low power datapath suitable for the highly deterministic environment of wireless communications processing. We discuss the building blocks and the instruction format. We also explain how the operations are performed. The new invention takes advantage of the high code density of CISC architectures to reduce memory traffic. It also makes the processors more AISC-oriented and drives the power consumption down. Even with these improvements, the invention still retains simplicity of system hardware in the tradeoff with software complexity. A quantitative discussion of performance is presented later in Chapter 6.

In the next chapter, we discuss another new invention focusing on data management. This has significant impact on the operand transfer scheme, on parallelism, and on memory hierarchy. The new data management architecture can be used in conjunction with the reconfigurable macro-operation signal processing architecture in the environment of multiple standard mobile wireless communications processing. With the support from each other, these two inventions further reduce the power consumption of multiple standard mobile units.

Chapter 4

Signal Processing Arrangement and Method for Predictable Data

Large data flows are processed and transferred to and from different memory hierarchies in wireless communications applications. In order to reduce power consumption, it is essential to move data among different memory spaces efficiently. On the other hand, all wireless communications protocols have well defined functionality and timing sequences which are known in advance. These properties distinguish wireless communications applications from other more general real-time applications run on general-purpose microprocessors and digital signal processors. In this chapter, we present a unique signal processing arrangement and method that takes advantage of the inherent characteristics of wireless communications processing. We separate software codes into two parts: computational codes and data management codes. These two parts are coded separately and issued to two different program decoders in parallel. The computational codes control the computational units with register-to-register data movements only. The data management codes control data movements through the data management unit to prepare and transfer data for the computational codes. Multiple register files with an active file swapping scheme are used to simplify the computational operations without sacrificing performance. While the data in the active register files are being processed, other data are being prepared and stored into inactive register files in advance and transferred back to other memory hierarchies after being processed by the computational codes. The timing synchronization between the two parts is transparent to software programs. Deterministic

memory management controlled by software replaces real-time dynamic memory management controlled by hardware. Memory management becomes controllable by software programs and can be optimized for each software routine individually. Using this scheme, data are transferred and reused much more efficiently among different memory hierarchies. This reduces power consumption and overall processing latency significantly.

4.1 Separated Data Management Instruction Stream

Traditional microprocessors and digital signal processors focus more on arithmetic operations than on the data involved in the operations. All instructions focus on the efficient utilization of the computational units. This structure does not match the needs of wireless communications processing well. Wireless communications processing focuses on data streams that are processed from voice coding to channel coding to modulation, or from demodulation to channel decoding to voice decoding, within particular time slots [Skl88]. Protocol processing becomes more data oriented. Thus, unlike more general tasks run on programmable processors, wireless communications processing is highly deterministic. In non-deterministic routines, we have no knowledge about the future data other than perhaps for a few cycles of prediction. We can transfer in the required data only a few cycles ahead of the arithmetic operations. In time critical situations, we have to increase the clock rate in order to finish the data transfer and process the data in time. However, in wireless communications processing, the determinism of fixed protocols releases us from these constraints. Data can be treated as objects and moved among memory hierarchies much more independently. Separating data management codes into a second instruction stream running in the background provides the flexibility to focus on datum objects while the primary instruction stream is handling arithmetic operations.

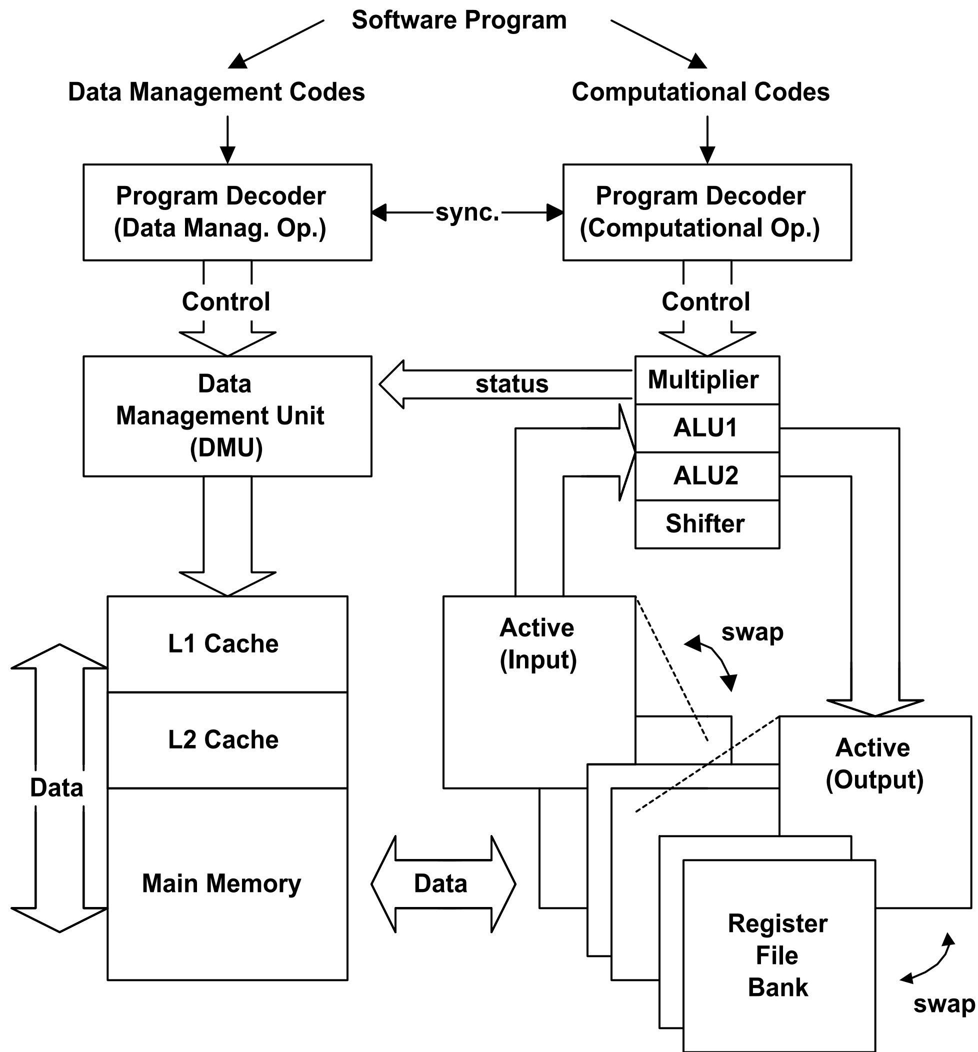


Figure 10 - Architecture diagram of signal processing arrangement and method for predictable data. The building blocks include two program decoders, data management unit (DMU), computational units, several layers of memory spaces, and a special register file bank.

This provides direct control over data and with this control another dimension for optimization of power consumption. On the other hand, two instruction streams are processed in parallel. In the just-in-time processing environment of wireless

communications applications, this can relax the clock rate requirement to approximately half. It, therefore, can relax the requirement on power supply voltage and further reduce the overall power consumption.

4.2 Proposed Architecture for the Signal Processing Arrangement and Method

Figure 10 shows the architecture for the proposed signal processing arrangement and method for predictable data. The architecture includes two data flow paths. One is for transferring data among different memory layers and for transferring data between register files and other memory locations. The other one is for transferring data from register files to computational units, then back to register files. The two data paths are controlled by two separated instruction streams and are connected inside the register file bank.

Section 4.2.1 explains the functionality of each block in Figure 10 and Section 4.2.2 describes the details of the register file bank. The procedure for arranging the software codes are explained later Section 4.3.

4.2.1 Block Description

This section describes each function block in the architecture diagram shown in Figure 10.

- *Program Decoders*: decode the instructions issued by a software program. Two program decoders are used to decode the instructions from the computational codes and the data management codes separately. They are synchronized by the program counter (PC). However, in order to reduce the system complexity, they do not have

knowledge of the instructions issued by each other. The operation scheduling and sequencing are arranged solely by the software program.

- *Data Management Unit (DMU)*: controls the data transfer operations required by the data management codes. Since the data transfers are arranged explicitly by software programs, this unit translates the instructions into control signals for each memory block. While an exception such as divided-by-zero will be generated by computational units, the DMU accepts such a status signal from the computational side and interrupts the following operations. Exceptions are handled by the operating system (OS), instead of the hardware, as is done in all other programmable processors.

- *Computational Units*: include one multiplier, one shifter, and two ALU's. The second ALU unit is added to provide better optimization of macro-operations, which are described in the previous chapter. The inputs and outputs of these units are connected to the register bank only. There is no direct connection between these units and other memory hierarchies.

- *Memory Hierarchies*: include a special register file bank, different layers of caches, and main memory. The proposed architecture focuses on the method for transferring data among these different memory locations. However, the detailed arrangements, such as the size of the caches, are beyond this discussion of the proposed architecture. The primary considerations in this innovation are the structure of the register bank and how data are transferred into and out of the register file bank.

- *Register File Bank*: contains several register files, which have relatively fewer registers inside than the typical register files of conventional programmable processors. All of these register files can be accessed by the data management codes. On the other hand, although all register files can be physically accessed by arithmetic operations, only the register numbers are named in computational codes. The

computational codes do not explicitly state which register file those registers belong to. They always access the currently active input and output register files, which are assigned by data management codes. The details of the register file bank is discussed in the following section.

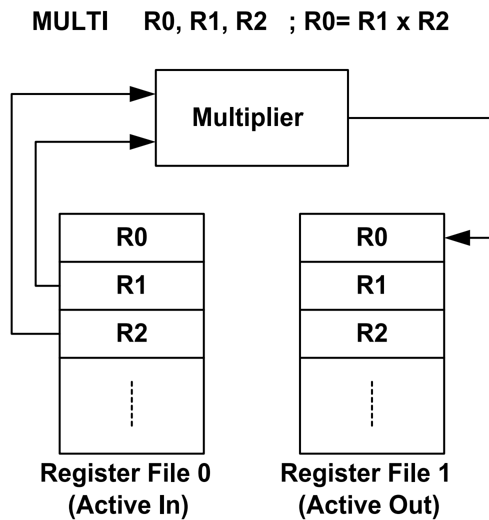


Figure 11 - Example of using active input and output register files. Computational instruction "MULTI R0, R1, R2" is issued. From the instruction itself, we do not know to which register files the registers R1, R2, and R3 belong. Data management codes assign register file 0 as the active input register file and register file 1 as the active output register file. Therefore, the multiplier gets inputs from registers R1 and R2 of register file 0. The output is stored into register R0 of register file 1.

4.2.2 Register File Bank

As mentioned above, the register file bank contains several small register files of the same size. The data management code assigns one of the register files as the active input register file. The data management code also assigns either the same or another register file as the active output register file. Logically, the computational codes can access only the active register file(s) assigned by the data management codes. The computational

codes have no knowledge of which register file(s) they are utilizing. In other words, from the perspective of the computational codes, there is logically only one register file even if input operands and output operands are physically assigned to two different register files. Figure 11 shows an example. Register file 0 is assigned as the active input register file, and register file 1 is assigned as the active output register file. While the instruction "MULTI R0, R1, R2" is issued by the computational codes, it is logically treated as operating on three operands of a same register file from the perspective of computational codes. But, in reality, operands R1 and R2 are from register file 0 and the output R0 is stored into register file 1. Separating logical and physical accessibilities of the register files, keeps the the computational instructions very simple and highly efficient. This provides extra flexibility in operation arrangements. With the support from data management codes, it also provides another dimension for optimization of power consumption.

Since the computational codes can utilize only register-to-register data movements, other kinds of data movements are handled by the data management codes. While the computational codes are being executed, the data management codes move data among different memory hierarchies in the background. The data management codes have access to inactive register files without disturbing the operations executed on the active register files. Future operands can be prepared in inactive register files and past operands can be stored back into other memory layers. In addition to moving data among different memory hierarchies, data management codes also control the active register file swapping by re-assigning which are the active register files. By changing the register files to be used by computational codes, the same set of computational codes can be reused efficiently. Figure 12 illustrates the reuse of computational codes by swapping active register file assignments. A subroutine is used to process a small group of operands and the same subroutine repeats on different set of operands. In traditional architectures, we use a loop structure and need to re-assign the operands in the beginning of every loop execution. In the proposed architecture, different sets of operands are stored in different register files in the same arrangement. In the computational codes, all operands named in the instructions are fixed. The active file swapping controlled by the

data management codes changes the operands that are being processed. The same few computational codes are used repetitively. While a register file is being used by the computational codes, the data in other register files can be updated or stored into main memory.

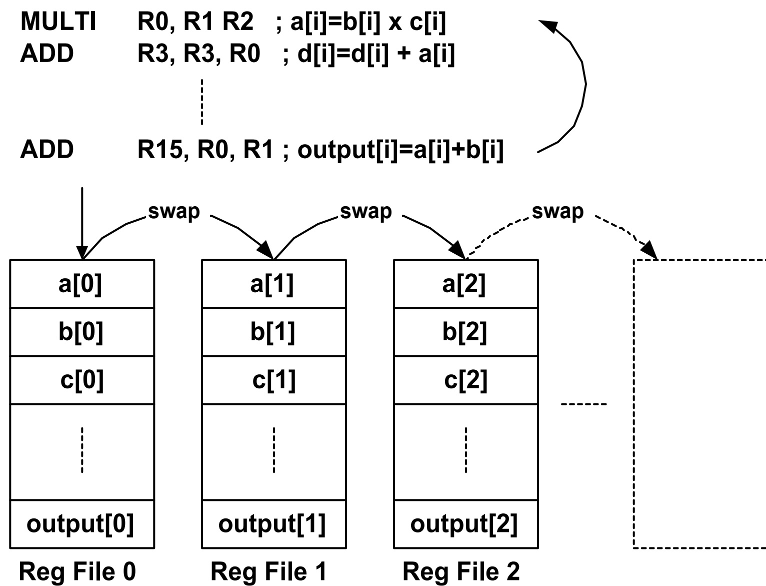


Figure 12 - Example of code reuse by swapping the active register files. A set of operations are to be performed on a set of inputs a[i], b[i], c[i], and other variables to create a set of outputs, i.e. output[i]. The inputs can be distributed into several register files so that the same computational codes can process through them by swapping the active register file assignment. No computational instruction modification is needed as we swap from one register file to another.

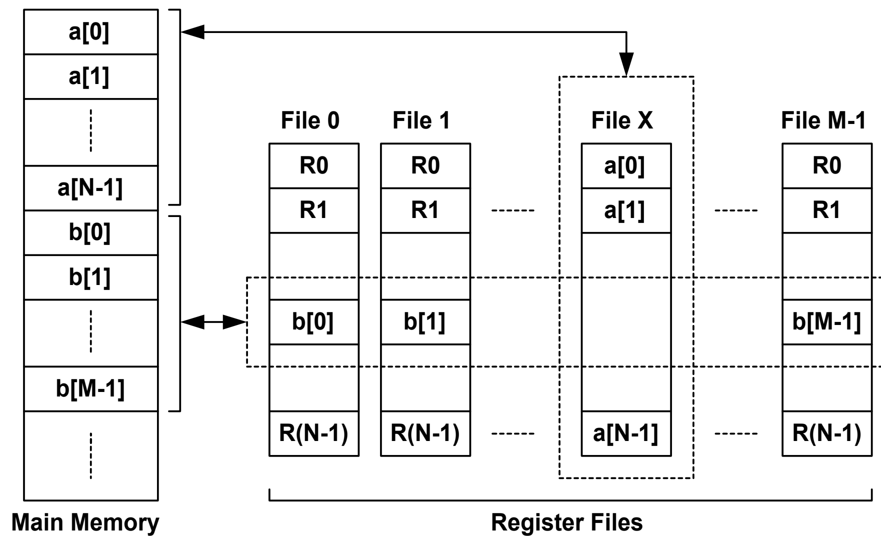


Figure 13 - Data block access to register files. Two dimensions of block access are provided. One is to access several registers in a single register file. The other is to access one register of several register files.

In addition to single datum transfer, two types of block data transfers are supported by the proposed architecture. As shown in Figure 13, block data transfers can be made for data in a same register file or for data in registers with the same register number across several register files. For example, assume data a[0] to a[N-1] and b[0] to b[M-1] are stored in main memory at continuous addresses. Data a[0] to a[N-1] can be transferred to the registers numbered from 0 to N-1 in a single register file. On the other hand, b[0] to b[M-1] can be transferred together, but each of them can be stored into a register of a different register file. All these registers have the same register number. Providing these two orthogonal accesses and the active register file swapping scheme enhances the flexibility of matrix and vector operations. However, block data transfers cannot be done randomly. In order to reduce the system complexity, block sizes and locations are limited. For a block data transfer to or from a single register file, the data block can only be the full register file, the lower half of the register file, or the upper half of the register file. For a block data transfer across several register files, the data block can only be the full row of the registers, the front half of the row of the registers, or the back half of the row of the registers. Figure 14 illustrates an examle of the constraints on

a register file bank which has eight register files with eight registers each. The impact of this constraint is limited. In a process that utilizes block transfers with more data in the transfer than the block can accommodate, most of the transfers are not affected. Only the last few data are left out and cannot be handled by the block data transfer. The remaining data need to be handled by a few single datum transfers. In order to avoid such extra effort, a few zeros can be padded at the tail of the data array. Limiting the locations and sizes of block data transfers reduces significantly the amount of overhead on hardware. This also avoids a complicated instruction format. All data management instructions remain simple and straightforward.

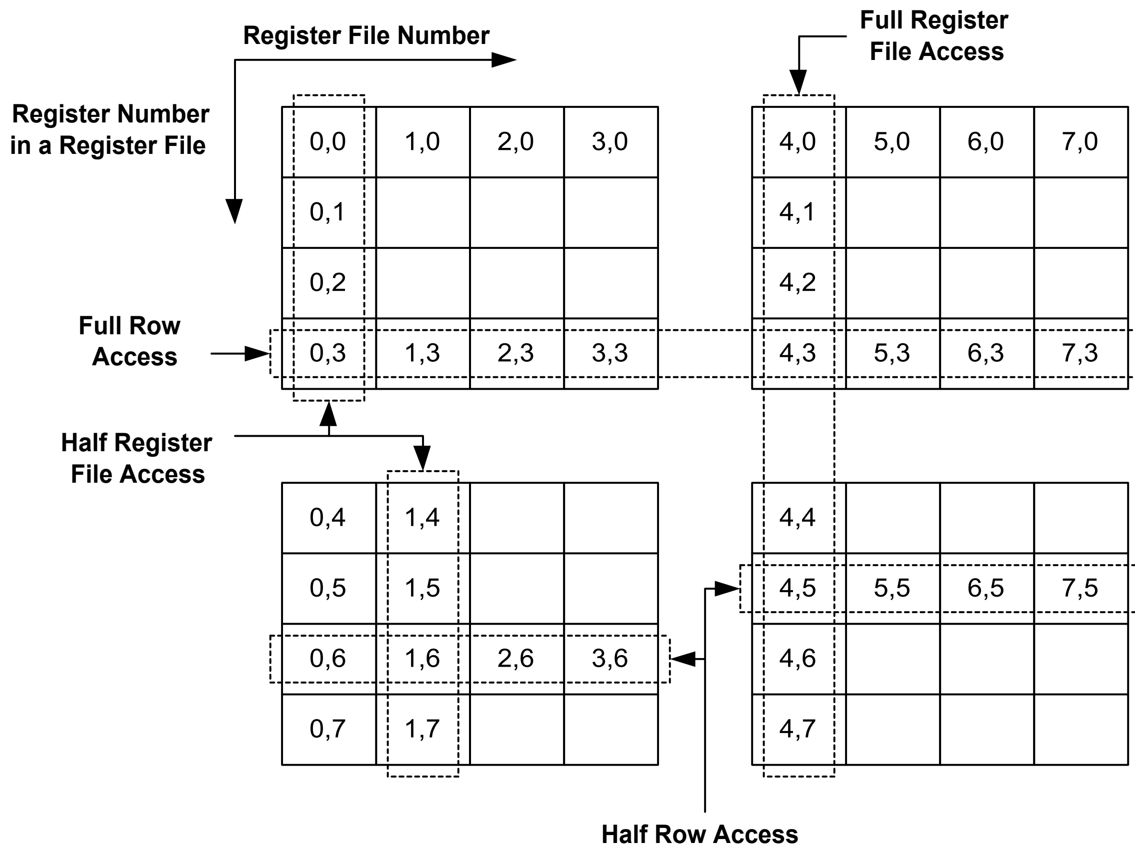


Figure 14 - Illustration of the constraints on block data transfer. Four types of block access are supported: full register file access, half register file access, full row access across register files, and half row access across register files. (x, y) in the diagram means the register y of register file x.

4.3 Software Arrangements

The procedure for arranging software codes can be divided into several stages. First, the details of the hardware architecture are hidden from the preliminary compiler. A single instruction stream is assumed in this first-level compiling. Also, only a single register file of the register file bank is utilized. In other words, there is no active file swapping. Preliminary assembly codes are generated from the signal processing programs written in high level language as is done in conventional processor architectures. Second, tables of operations and operands are created from the preliminary assembly codes. Operations and operands are analyzed and re-arranged. Operands are re-named for the environment of the multiple register files of the register file bank. The commands for swapping active register files are created. Third, the assembly codes are separated into computational codes and data management codes. The sequences of the codes are re-arranged. All of the data management codes are moved to the earliest time slots in which they can be executed. Then, the data transfers are packed into block data transfers to the extent possible. The computational codes then are streamlined to create possible code re-usage. Finally, all the codes are examined for correct timing and operations.

The details of the software programming procedure are discussed in Section 5.2. In Section 5.2, the software arrangements discussed above are combined with the software programming procedure for reconfigurable macro-operations for the overall system architecture discussed in Chapter 5.

4.4 Example

Equation 4.1:

$$y(n) = \sum_{k=0}^{7} h(k)[x(n-k) + x(n-15+k)]$$

A symmetric finite impulse response (FIR) filter is widely used in wireless communications because of its linear phase response. Therefore, we use a 16-tap symmetric FIR filter to illustrate the operations of the proposed design. In this case, we assume that eight register files with eight registers each are used. Equation 4.1 represents a 16-tap symmetric FIR filter. Figure 15 shows the corresponding data movements in register files along the operations. First, the filter coefficients are stored in the lower half portions of register file 0 and 1. These coefficients remain there and unchanged during the whole process. Second, input data x[n] to x[n-3] and x[n-12] to x[n-15] are transferred into register file 2. Input x[n-4] to x[n-11] are transferred into register file 3. Third, each

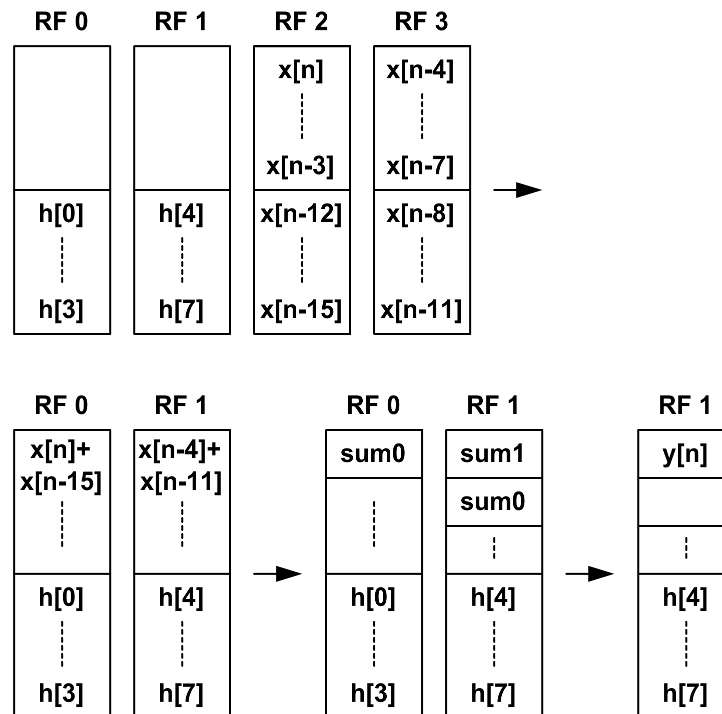


Figure 15 - Register file contents for symmetric FIR example. First, filter coefficients, h's, are stored in register file 0 and 1. The input data, x's, are stored into register files 2 and 3 in the arrangement shown in the figure. Second, pairs of inputs are summed and stored in register file 0 and 1. Then, two partial accumulations are generated and stored. Finally, the partial accumulations are summed together to provide the final output.

input data pair, such as x[n] and x[n-15], is summed into the upper portions of registers 0 and 1. Fourth, the multiply-and-accumulate operates on register file 0 and 1. The final output is generated in register file 0 and moved to memory. The same procedure repeats to provide the next output. The unfolded pseudo-operations are listed in the following.

Pseudo-operations:

	Data Management Op.	*Computational Op*	
	(move h[0]~h[3] into RF0_LowerHalf)		
	(move h[4]~h[7] into RF1_LowerHalf)		
0000	move x[n]-x[n-3] into RF2_UpperHalf		(operations for y[n-1])
0001	move x[n-12]~x[n-15] into RF2_LowerHalf	(operations for y[n-1])	
0002	assign RF2 as active input file		
	assign RF0 as active output file		
0003	move x[n-4]~x[n-7] into RF3_UpperHalf	add	R0, R0, R7
0004	move x[n-8]~x[n-11] into RF3_LowerHalf	add	R1, R1, R6
0005	(background housekeeping or idle)	add	R2, R2, R5
0006	(background housekeeping or idle)	add	R3, R3, R4
0007	assign RF3 as active input file		
	assign RF1 as active output file		
0008	(data movement into RF2 for y[n+1])	add	R0, R0, R7
0009	(data movement into RF2 for y[n+1])	add	R1, R1, R6
000a	(background housekeeping or idle)	add	R2, R2, R5
000b	(background housekeeping or idle)	add	R3, R3, R4
000c	assign RF0 as active input file		
	assign RF0 as active output file		
000d	(data movement into RF3 for y[n+1])	multi	R0, R0, R4
000e	(data movement into RF3 for y[n+1])	mac	R0, R1, R5
000f	(background housekeeping or idle)	mac	R0, R2, R6
0010	(background housekeeping or idle)	mac	R0, R3, R7

```
0011  assign RF1 as active input file
      assign RF1 as active output file
0012  (background housekeeping or idle)    multi  R0, R0, R4
0013  (background housekeeping or idle)    mac    R0, R1, R5
0014  move RF0_R0 into RF1_R1              mac    R0, R2, R6
0015  (background housekeeping or idle)    mac    R0, R3, R7
0016  (background housekeeping or idle)    add    R0, R0, R1
0017  move y[n] into memory                (operation for y[n+1])
```

As shown on the right side of the pseudo-operations, the computational codes are very simple. The same assembly codes are used in different places. The same arithmetic operation block processes through different sets of data, while the data management codes on the left side assign the active input and output register files. The computational codes can be reused. This same situation applies to many routines used for wireless communications. On the data management side, all the commands are simple and straightforward. The required instruction word length is minimal. This minimizes the impact on the overhead introduced by the second instruction stream. On the other hand, there are many open slots available for background housekeeping. These can be used to arrange future data in lower level memory hierarchies. This housekeeping is especially useful in the transition period between two tasks.

4.5 Architecture Impacts

The proposed signal processing arrangement and method for processing predictable data can be used to reduce the power consumption of programmable processors for highly deterministic applications such as multiple standard wireless communications processing. In response to the processor factors discussed in Chapter 2, the impacts on traditional

processor architectures are discussed in the following. The architecture modeling and performance evaluation are discussed in Chapter 6.

The operand transfer scheme of the proposed architecture is the primary portion of the innovation. Introducing another instruction dimension distinguishes this scheme from all existing operand transfer schemes. On the computational operation side, the operand transfers are limited to register-to-register operand movements only. The instruction length is short and fixed. This increases the executing performance of computational units and reduces the latency significantly. However, with the support from the data management codes, this does not compromise memory efficiency as is done in conventional RISC architectures. The effective code density is higher than in RISC. The selections of operations are increased by combining the operations in two dimensions. Therefore, the flexibility is increased significantly compared with other architectures that have the same level of code density.

While the number of computational resources is not increased, parallelism is enhanced by executing data transfer operations in the background. The tradeoff is in the difficulty of software programming. Pipelining is achieved in a broader sense. The pipeline is effectively separated into the stages of operand-preparation, arithmetic-operation, and operand-store-back. However, there is no clear pipelining path in the architecture because these stages are handled by the two different portions of the architecture separately. The pipelining operations are scheduled by the two instruction streams.

The memory hierarchy itself is not changed. The memory control, however, is done by software programs. Data movements among different memory layers need to be arranged explicitly. The proposed architecture does not specify the details of memory hierarchy, such as the sizes of cache and main memory. The arrangement of the memory hierarchy needs to be determined by the system in which the proposed architecture is embedded.

The interrupt handling capability is enhanced. While interrupted, the operating system does not need to store the values of all registers as does in most other programmable processors. In wireless communications processing, the interrupting tasks

are usually control-oriented and can be served by a few registers. The proposed architecture uses many small register files instead of a big register file. Therefore, the operating system can assign only a portion of the register bank to the interrupting task and store only the values of those register files utilized by the interrupting task.

The proposed architecture does not have the flexibility constraints of an ASIC; it is still a programmable architecture. However, as discussed above, combining the operations in two dimensions provides more selections of operation. This provides more opportunities for performance optimization and drives the programmable processor architecture towards ASIC in power consumption.

Although the proposed architecture has been created for mobile wireless communications processing, its application is not limited to low power communications protocols processing. The signal processing arrangement and method could be applied to other applications to increase execution speed or energy efficiency. A good example is in general purpose microprocessors pursuing higher performance. Original software codes could be pre-processed to generate redundant data management codes. The data management codes could be processed in separated data pre-fetching blocks to prepare possible operands. This would reduce the overall latency significantly. However, as discussed before, it is not possible to predict future operations accurately in non-deterministic applications. The predictions and data preparations in general purpose microprocessors are often redundant and are not necessarily used. This increases the power consumption dramatically. Unlike in wireless communications processing, there is little opportunity to achieve both low power and high speed by applying the proposed architecture to non-deterministic applications.

4.6 Summary

Data management has a significant impact on the power consumption of all applications. In wireless communications processing, data management is even more important

because large data flows are transferred constantly between the computational units and memory layers.

In this chapter, we present a unique signal processing arrangement and method that can be used to enhance the data management of programmable processors. Separating software routines into computational codes and data management codes provides a new novel data management architecture with higher memory efficiency and software code efficiency for the environment of wireless communications processing. We discuss the building blocks of the architecture and the register file bank for separated data management codes and computational codes. We also describe the procedure for arranging the software codes. This new invention provides direct control over data and a second dimension for optimizing the power consumption. All instructions are simple and straightforward. The combination of operations in the two dimensions provides the flexibility and energy efficiency that are needed in multiple standard mobile wireless communications. A quantitative discussion of performance is presented later in Chapter 6.

In the next chapter, we describe a new overall system architecture using both the innovation discussed in this chapter and the reconfigurable macro-operation signal processing architecture discussed in Chapter 3.

Chapter 5

Overall System Architecture

In previous chapters, two new architectures are presented: reconfigurable macro-operation signal processing architecture, and signal processing arrangement and method for predictable data. These innovations are used to reduce the power consumption of programmable processors without trading off the flexibility needed for multiple standard mobile wireless communications processing. Although both architectures take advantage of the inherent characteristics of wireless communications processing, they focus on different portions of programmable processor architecture. The reconfigurable macro-operation signal processing architecture enhances the efficiency of computational resource utilization. The signal processing arrangement and method for predictable data, on the other hand, enhances the efficiency of data management. In this chapter, we present a new overall system architecture utilizing both innovations to further mitigate the tradeoff between flexibility and energy efficiency. The proposed system architecture also includes a method for merging the functionalities of the microcontroller and the digital signal processor. This method provides a new truly unified microcontroller-DSP architecture as discussed previously in Chapter 2.

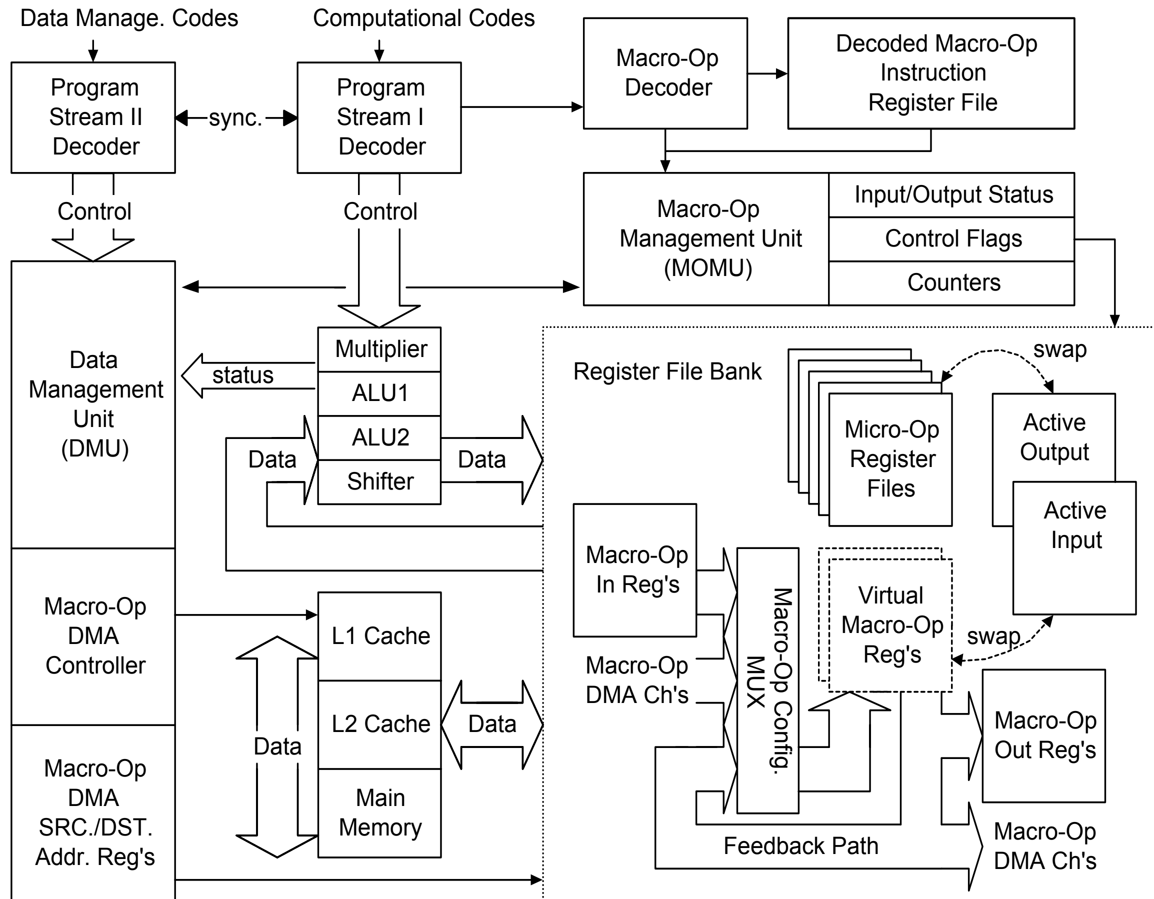


Figure 16 - The diagram of the system architecture combining the reconfigurable macro-operation signal processing architecture shown in Figure 4 and the data management architecture for predictable data shown in Figure 10.

5.1 System Architecture

Figure 16 shows the proposed system architecture that includes the reconfigurable macro-operation signal processing architecture and the data management architecture for predictable data. As shown, the major structure is composed of the combination of the diagrams shown in Figure 4 and Figure 10. The two architectures proposed previously are connected in two places: the program decoders and the computational units. Since the macro-operations are computational instructions, they are dispatched from the computational codes. The program stream I decoder used for the computational codes determines whether or not the issued instruction is a macro-operation by decoding the first byte of the instruction macro-operation. The first byte of a macro-instruction, macro-operation indication, distinguishes a macro-operation from a micro-operation. If the issued instruction is a macro-operation, the remaining nine bytes of the macro-operation are passed to the macro-operation decoder for further processing. This macro-operation processing is described previously in Chapter 3. The procedure is straightforward. On the other hand, the connection in the computational units demands more consideration. This involves the register file structure and the execution of macro-operations. Figure 17 shows an enlarged picture of the special register file bank used in the proposed system architecture.

5.1.1 Block Description for Register File Bank

- *Active Input/Output Register File*: are only logical register files. They do not exist physically. In micro-operation executions, as discussed in Chapter 4, one of the micro-operation register files can be assigned as the active input register file. Either the same micro-operation register file or another micro-operation register file can be assigned as the active output register file. In addition, the macro-operation input register file can also be assigned as the active input/output register file so that constant parameters can be pre-arranged for macro-operations. For the same reason,

the macro-operation output register file can be assigned as the active input register file so that the results of macro-operations can be used by micro-operations. However, there is no need to assign the macro-operation output register file as the active output register file. In macro-operation executions, the virtual macro-operation register files are assigned as the active input/output register files. The computational units always take inputs from the active input register file and store output(s) to the active output register file in both micro- and macro-operation executions.

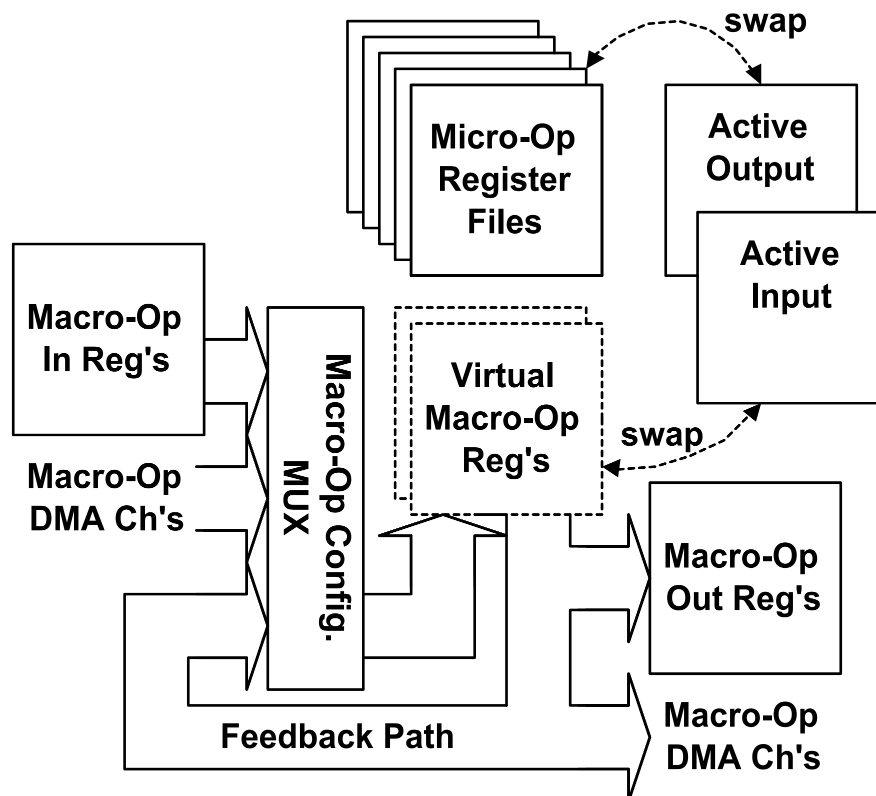


Figure 17 - Register file bank. The building blocks include active input register file, active output register file, micro-operation register files, virtual macro-operation register files, macro-operation input registers, macro-operation input DMA channels, macro-operation configuration multiplexer, macro-operation output registers, and macro-operation output registers.

- *Micro-Operation Register Files*: are of the same structure as discussed in Chapter 4. Orthogonal block data transfers are supported.

- *Virtual Macro-Operation Register Files*: are only logical register files. They do not exist physically. While macro-operations are being executed, these files are assigned as the active input/output register files to provide the necessary paths for inputs and outputs for the computational units. Physically, the virtual macro-operation register files provide the outputs from the macro-operation configuration multiplexer and the buses connecting the feedback path, the macro-operation output register files, and the macro-operation output DMA channels.

- *Macro-Operation Input/Output Registers*: store the constant parameters as inputs and the final outputs in macro-operation executions. While an interrupt occurs, the macro-operation output registers also store the intermediate results for the datapath. However, as discussed above, the macro-operation input registers can be read and written into by a micro-operation. This is different from the architecture discussed in Chapter 3, where micro-operation read for macro-operation input/output registers is not necessary and not supported. On the other hand, the macro-operation output registers remain readable only as described in Chapter 3.

- *Macro-Operation Input/Output DMA Channels*: are of the same structure as discussed in Chapter 3. However, the details of operations are different. Although macro-operations are utilized by computational codes, DMA operations involve those functional blocks mainly used by data management codes. While DMA operations are requested by a macro-operation, the data management unit (DMU) is responsible for the operations. Since computational codes can utilize register-to-register data movements, the DMA source and destination addresses are set up in data management codes instead of computational codes.

- *Macro-Operation Configuration Multiplexer*: is of the same structure as discussed Chapter 3. The inputs are from three sources: macro-operation input registers, macro-operation DMA input channels, and the feedback path. The outputs are physically

directed to computational units logically through the virtual macro-operation register files.

5.1.2 Universal Operand Access

In order to provide smooth transitions between micro-operations and macro-operations physically and logically, a universal method of operand access is used in the proposed system architecture. In both micro-operations and macro-operations, the operands are fetched from the assigned active input register file and are stored into the assigned active output register file. From the perspective of the computational units, there is no difference between micro-operations and macro-operations. The multiplier, shifter, and ALU's start to perform their operations as soon as the inputs to them are ready on the input data buses from the active input register file. After they finish the operations, the results are available on the output data buses to the active output register file. All the data routing and pipelining are arranged in the special register file bank. The details are hidden from the computational units through the arrangement of active register files. The switching between micro-operations and macro-operations becomes the switching of the active register files.

In the proposed architecture, the macro-operation input/output register files and the virtual macro-operation register files are treated the same logically and are also named the same in the instructions, although they serve different types of operations physically. The macro-operation input/output register files are used in both micro-operations and macro-operations. In micro-operations, there is no difference between the macro-operation input/output register files and the micro-operation register files, except that the macro-operation output register files are only readable. The macro-operation input register file can be assigned as the active input or output register file. The macro-operation output register file can be assigned as the active input register file. In macro-operations, the macro-operation input register file provides necessary inputs and the macro-operation output register file stores the outputs. The virtual macro-operation register files are used only in macro-operations. While a macro-operation is dispatched

from the computational codes, they are assigned automatically as the active input and output register files to provide the necessary data paths for the macro-operation. By using this special arrangement, the complexity of using different types of operations is reduced and the method for accessing operands becomes universal. Table 4 summarized the discussion in this paragraph.

Table 4 - The relations between operation type, active register file assigned in the instruction (logically), and physical arrangement for the active register input/output registe files.

Operation type	***Active register file assigned in the instruction (logically)***	***Physical arrangement for the active input or output register file***
micro-operation	micro-operation register file	one of the micro-operation register files
micro-operation	macro-operation register file	macro-operation input or output register file
macro-operation	macro-operation register file	virtual macro-operation register file

Table 5 - Software program partitioning: different types of operations in the data management code and in the computational code.

Data Management Codes
• data transfers among memory layers
• data transfers between register file bank and other memory locations

• assign active register files • assign DMA source/destination addresses for macro-operations
Computational Codes
• arithmetic micro-operations • data transfers from register to register • macro-operations • conditional/unconditional jumps

5.2 Software Program

As discussed in Chapter 4, a software program is divided into two portions: computational codes and data management codes. Table 5 lists different types of operations that are included in the computational codes and the data management codes.

The operations in the data management codes include data transfers among memory layers, data transfers between register file bank and other memory locations, assigning the active register input and output files, and assigning the DMA source and destination addresses for macro-operations. The instructions for data transfers among memory layers are to replace the dynamic memory management handled by hardware in conventional processor architectures. The dynamic memory management scheme used by conventional processor architectures is based on the rate of data missing in the cache. It exploits the temporal locality of data from past executions. It does not receive any information or support from software programs for future executions. Therefore, it does not take advantage of the determinism property of wireless communications processing. By using direct control over data movements among memory layers, the software structure for the proposed system architecture can exploit the spatial and temporal locality of data globally. The instructions for data transfers between the register file bank and other memory locations are for preparing future operands for the computational

codes and for storing the past results to other memory layers. These operations are done in the background while the computational units are executing operations on other data. This reduces the operation latency. In conjunction with the special structure of the register file bank, this arrangement also makes the arithmetic operations very simple and highly efficient. The active file swapping commands control the data flows inside the register file bank. In order to keep the simplicity of a signal register file for the computational codes, these commands are issued in the data management codes instead of the computational codes. For the same reason, the assignments for the DMA source/destination addresses in macro-operations are also handled by the computational codes.

The operations in the computational codes include arithmetic micro-operations, data transfers from register to register, macro-operations, and conditional/unconditional jumps. As discussed in the previous paragraph, the computational codes are supported by the data management codes in many ways. Therefore, arithmetic micro-operations become very simple. All operands, except the intermediates explicitly stated in the instructions, are in registers and are logically named by the register numbers without the register file numbers. The instructions are more compact than the instructions in conventional RSIC architectures. Register-to-register data movements can also be performed without any arithmetic operations. The data are directly moved from the active input register file to the active output register file. However, the data cannot be moved into or out of an inactive register file. This constraint does not reduce the performance of the proposed architecture although it makes the software programming more difficult if a significant number of register-to-register data movements are needed in the routines. Macro-operations are issued by the computational codes. Although macro-operations may utilize direct memory access (DMA) operations, they do not conflict with the arrangement of a second stream for data management. DMA requests are forwarded to the data management unit (DMU) where corresponding control signals are generated. Conditional/unconditional jumps are used to control the program flow. A system architecture with two instruction streams requires special considerations for jump operations. A jump operation requested in one instruction stream changes the program

flow in both instruction streams. In the proposed architecture, a jump operation is always issued in the computational codes. Because both instruction streams share the same program counter (PC), a jump issued in the computational codes will change the program flow of the data management codes automatically. The instruction scheduling immediately before a conditional jump needs to be more conservative to prevent unnecessary data movements. Unconditional jumps do not cause the same problems because the operations around an unconditional jump can be optimized as they are in an unfolded program flow. Figure 18 illustrates the software programming procedure for the proposed system architecture. The details of each step are explained in the following:

Step 1: Write application routines in high-level programming language such as C/C++. Although the software programming could start directly with assembly code, it is much easier to verify the operations of high-level language routines. High-level languages also have more support such as function libraries and debugging tools.

Step 2: A preliminary compiler needs to be created. This compiler is not for generating the final assembly code for the proposed system architecture. Instead, this compiler is used to generate preliminary assembly code for a pseudo system architecture that presents the system architecture without all the inventions discussed previously. In other words, none of the reconfiguarble macro-operations signal processing architecture, the data management architecture, and the register file bank architecture is used in the pseudo system architecture.

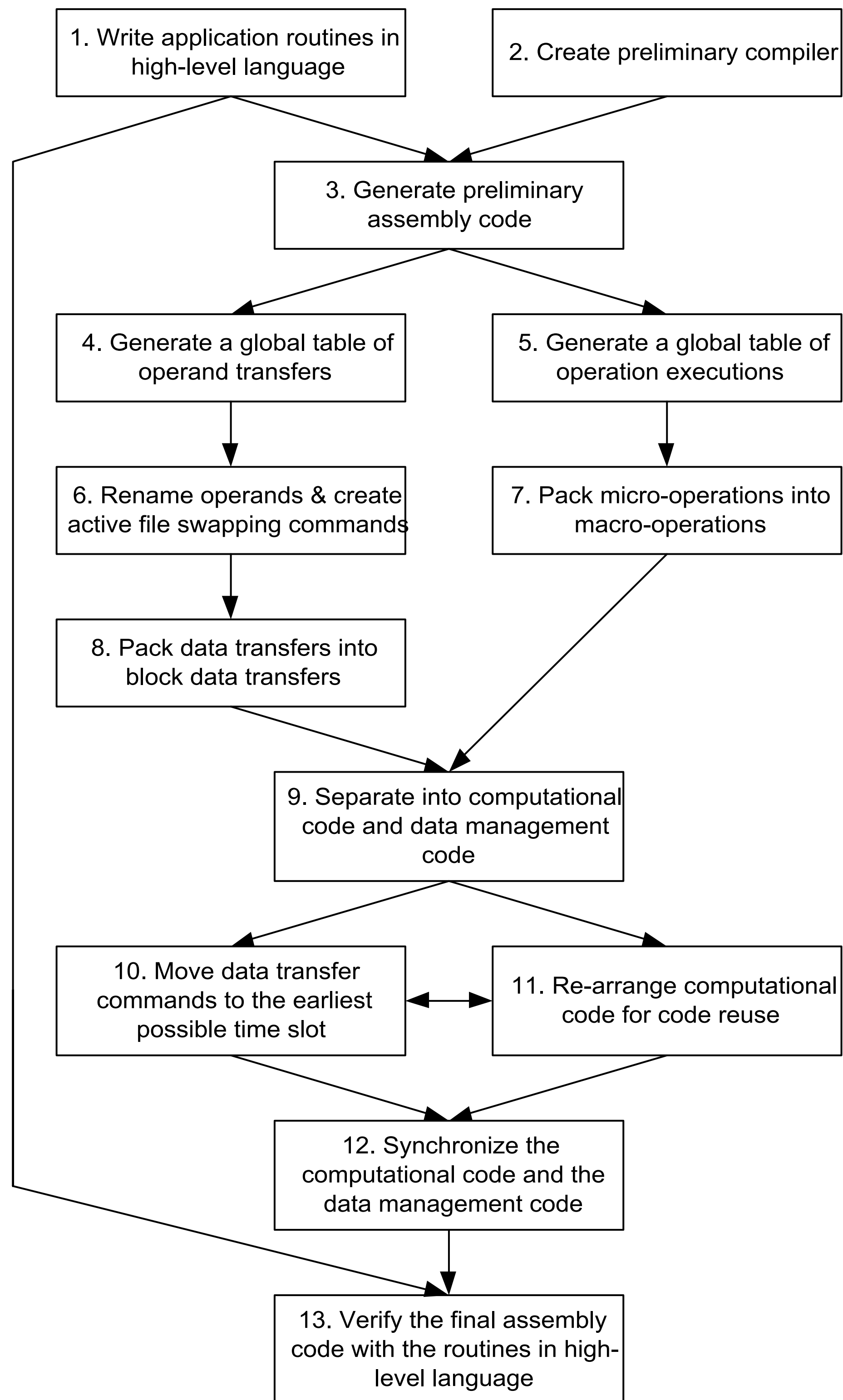


Figure 18 - Software programming procedure for the proposed system architecture.

This preliminary compiler is very similar to regular compilers for conventional processor architectures. In this research, we select an existing compiler for the Intel x86 architecture [Int95][Int96] as the preliminary compiler.

Step 3: Generate the preliminary assembly code by using the preliminary compiler from Step 2. This step shortens the programming cycle by using the preliminary assembly code as reference.

Step 4: A global table of operand transfers is created. The spatial and temporal localities of the operands are examined.

Step 5: A global table of operation executions is created for the instruction scheduling in Step 9.

Step 6: The operands from Step 4 are renamed in order to expand from operations using a single register file to operations using the full register file bank. The commands for swapping the active input and out register files are created in this step.

Step 7: Micro-operations are packed into macro-operations for all possible situations. In order to minimize the frequency of reconfiguring the datapath, the selections of the macro-operations to be used are optimized globally.

Step 8: Data transfer commands are packed into block data transfer commands for all possible situations.

Step 9: This is the most critical step. In this step, we separate the preliminary assembly code that has been modified in previous steps into computational code and data management code. These codes are re-scheduled and further optimized for the proposed system architecture by the information from Step 4 and Step 5.

Step 10: All data transfer instructions in the data management code are moved to the earliest possible time slots. All the empty time slots in the data management code are noted for other background operations if needed.

Step 11: In this step, we re-arrange the computational code for possible code reuse. Obviously, any change made in this step has an impact in the data management code. Therefore, Step 10 and Step 11 are not independent. The codes for both sides are considered and modified together.

Step 12: The computational code and the data management code are synchronized by the instruction line numbers that will be pointed to by the same program counter (PC) in real-time execution.

Step 13: The final assembly code needs to be verified for timing and functionalities. The operations dispatched by the assembly code are compared to the original software routines in the high-level programming language.

5.3 Summary

Arithmetic operation processing and data transfer are the two primary concerns for programmable processor architectures. They identify the focus of an architecture and they also determine the performance of the architecture.

In this chapter, we present a new novel overall system architecture that addresses the challenges in both arithmetic operation processing and data transfers. The proposed system architecture merges the structures and the benefits of the reconfigurable macro-operation signal processing architecture discussed in Chapter 3 and the signal processing arrangement and method for predictable data discussed in Chapter 4. The novel arrangement inside the register file bank provides smooth transitions between macro-operations and micro-operations. This fulfills the requirements for a true unified microcontroller-DSP architecture. The different arrangements for logical access and physical access also reduce the system complexity.

The next chapter presents a quantitative discussion of the performance of the proposed architectures.

Chapter 6

Performance Evaluation

When we presented the proposed architectures for reconfigurable macro-operation signal processing and data management in Chapter 3 and Chapter 4, we discussed the impacts of these architectures on system performance from theoretical perspectives. In this chapter, a quantitative discussion is presented. The chapter begins with a brief discussion of the power consumption of integrated circuits implemented in complementary metal oxide silicon (CMOS) technology. The approach for power reduction in this research is also explained. In Section 6.2, the architecture modeling and simulation methodology are described. The simulation results are presented in Section 6.3 and Section 6.4. Then, the system-level performance of the proposed system architecture is discussed in Section 6.5.

6.1 Power Consumption of CMOS Circuits

Many device technologies are used to implement digital integrated circuits: Bipolar, MOSFET, NMOS, PMOS, CMOS, and others. CMOS technology has become the most popular one in the last decade because of its high integration capability, low power dissipation, scalability, and low cost for manufacturing [Wes88][Won97]. Although all device technologies can be used to implement the proposed architectures, CMOS

technology is chosen in this research because of its popularity. Equation 6.1 presents the total power consumption in CMOS circuits.

Equation 6.1: $P_{total} = \frac{1}{2} C_L V_{DD}^2 f_{clock} N_{activity} + Q_{SC} V_{DD} f_{clock} N_{activity} + I_{leak} V_{DD}$

where P_{total}: total power consumption

C_L: average node capacitance

V_{DD}: power supply voltage with respect to ground

f_{clock}: clock frequency

$N_{activity}$: average number of voltage transitions per clock cycle

Q_{SC}: average short-circuit charge per voltage transition

I_{leak}: total leakage current

The first term of Equation 6.1 is the power consumption resulting from charging and discharging circuit nodes. In most VLSI CMOS circuits, this term is the primary source of dynamic power consumption and counts for approximately 90% of the total power consumption. The second term is another source of dynamic power consumption. It is due to a short current pulse during the transient state of PMOS and NMOS transistors which generates a short circuit from power supply voltage to ground. The third term is the static power consumption resulting from the leakage current in CMOS devices. The second and third terms of Equation 6.1 are determined mainly by the inherent properties of CMOS devices. The contribution from these two terms is also less significant compared with the first term of Equation 6.1. Therefore, for considerations at circuit and system levels, most effort towards reducing power consumption is aimed at the first term. Hence, the primary factors become clock frequency, power supply voltage, node capacity, and switching activity of the circuit.

Clock frequency is the easiest factor to change. Power consumption is reduced by decreasing clock frequency. However, reducing clock frequency also means slower operation and reduced performance. The reduction of clock frequency is limited by the minimum computation required to finish a given tasks in time. In wireless

communications processing, the system has to be fast enough to catch up with the continuously incoming and outgoing signals.

Power supply voltage plays an important role in power consumption because the power consumption is proportional to the square of power supply voltage. This is not an independent factor. It has significant impact on the inherent speed and leakage current of CMOS devices. Increasing power supply voltage makes the devices switch faster and provide better performance. The tradeoff is higher power consumption. On the other hand, reducing power supply voltage reduces the computing power of the system. The benefit is lower power consumption. Since the power consumption is proportional to the square of power supply voltage, it is beneficial to reduce power supply voltage and increase the system parallelism to finish a given task within its required time limit. This reduces the overall power consumption. However, there are two factors that prevent this approach from proceeding too far. First, the logic function performance of CMOS devices needs to be retained. Decreasing power supply voltage reduces the error margin for logic functions. If the voltage is reduced too much, the CMOS devices will not work. Secondly, the leakage current is increased as the power supply voltage is reduced. The third term of Equation 6.1 becomes more significant and eventually cannot be ignored. More detailed consideration of these two factors is beyond the consideration of this dissertation.

The average node capacitance strongly depends on the circuit design and the layout of the system. Clock lines and global buses contribute to the major portion of the total capacitance loading. This average capacitance also depends on the system architecture and the algorithms used to finish a given task. In general, the fewer resources in the system architecture that are involved in finishing an algorithm, the smaller the average node capacity. For example, if a routine utilizes a large amount of main memory access, the average capacitance to be charged or discharged would be quite high. The problem can be mitigated by changing the algorithm or the system architecture.

The switching activity also depends on the system architecture and the algorithms to be executed. In order to reduce power consumption, an operation should not be executed unless it is really necessary. Unnecessary data movements, extra control

overhead, and less optimized algorithms increase the switching activity without improving system performance.

The four factors discussed above can be adjusted at several different levels to reduce the power consumption. These include device, circuit, logic, system architecture, and algorithm levels. This research addresses the power reduction at the system architecture level. Two reasonable assumptions are made regarding algorithms and technology in this research:

- The mathematical equations of the algorithms remain the same. Therefore, the number of high-level arithmetic operations included in a mathematical function is unchanged. For example, if a filter function contains a certain number of high-level multiplication and accumulation operations, the number of these high-level operations remains the same when the filter function is executed in a new system architecture. However, this does not mean that low-level operations are also kept the same. The low-level assembly codes can be different.
- The device technology and circuit design of the building blocks are not changed. Devices of different scales, for example, 0.35 micro-meters and 0.18 micro-meters, have different power consumption. Even in the same device scale, many circuit techniques can make a function block consume different amounts of power. In this research, we focus on the improvement that can be achieved at the architecture level. Therefore, we assume the power consumption of the fundamental building blocks, such as a 16-bit multiplier and an 8-bit decoder block, remains unchanged for a fixed power supply voltage.

Under these two assumptions, power consumption can be reduced mainly by eliminating unnecessary data movements, decoding, and controlling operations. The details of the low-level operations are re-arranged by the proposed reconfigurable macro-operation signal processing architecture and the proposed data management architecture. Fewer low-level operations are executed for a given task. In other words, the overall switching activity level is reduced. Also, the local data bus structures are changed in the

proposed architectures. Thus, the effective node capacity is decreased. On the other hand, the data management architecture achieves lower latency in execution. In the just-in-time execution environment of wireless communications processing, the latency reduction can be traded to further reduce power consumption by reducing the power supply voltage and/or the clock rate.

6.2 Simulation

The proposed architectures are modeled at register-transfer-level (RTL) with the hardware description language Verilog [Mad95][Ste93] to demonstrate their feasibility. In order to evaluate the performance of the proposed architectures, the hardware modeling Verilog codes are then mapped to high level language (C/C++) codes [Kru91][McC93][Mey92][Str91]. Power consumption and execution latency are evaluated by the C/C+ simulations. Figure 19 shows the procedure flow of the evaluation. The details of each step are explained in the following:

1. *Define global specifications*: The global specifications of the Verilog modeling are defined first: datapath word length, clock scheme, hierarchy of different modules, connections among all building blocks, and etc. In this modeling, a 16-bit datapath and a one-phase clock are chosen. Eight register files with eight registers each are used to construct the register file bank.
2. *Define specifications of building blocks*: At this stage, we define the scope of each

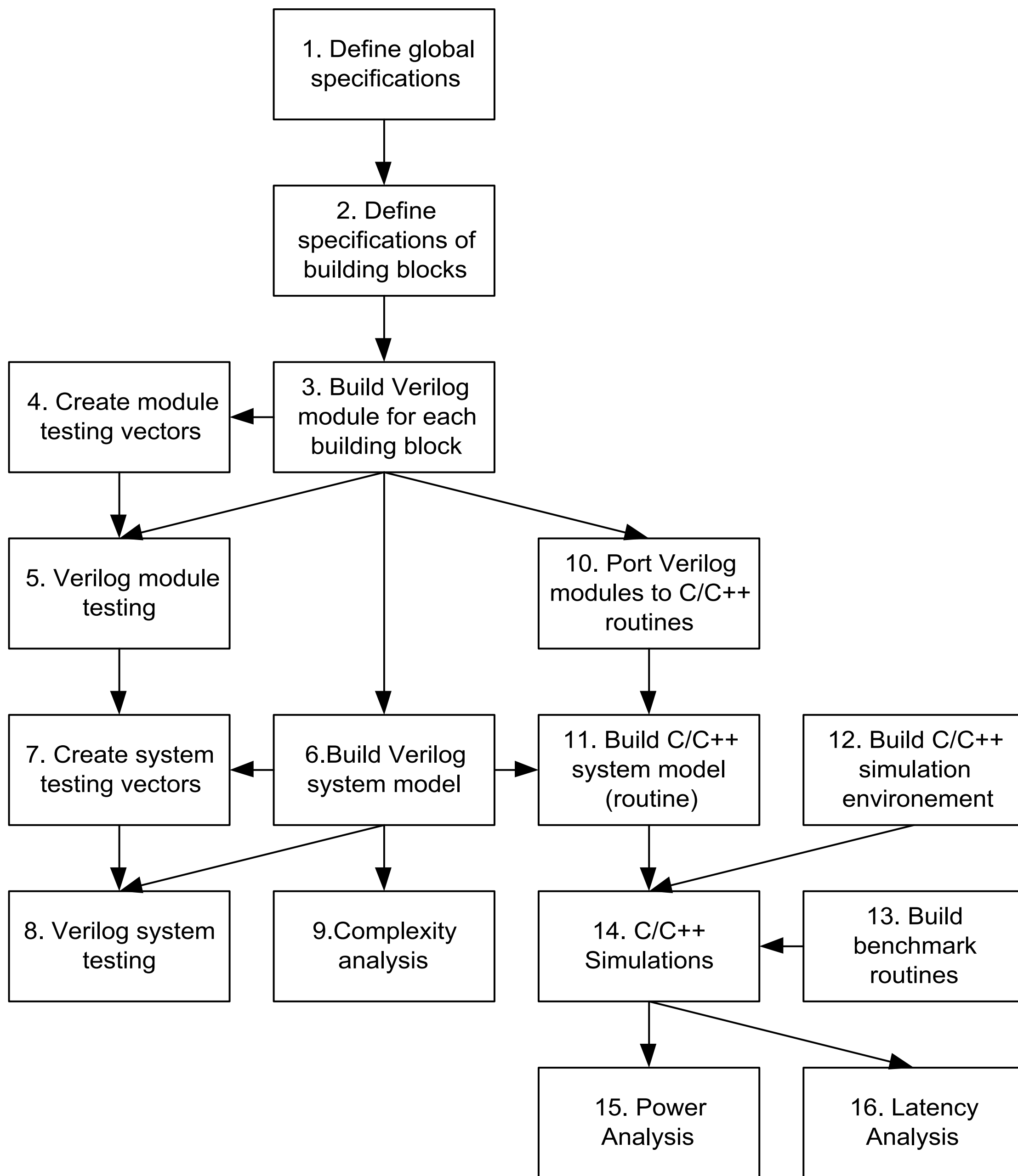


Figure 19 - Procedure flow for performance evaluation. The flow starts at defining global specifications (block 1) and ends at complexity analysis (block 9), power analysis (block 15), and latency analysis (block 16).

Verilog module. The necessary inputs and outputs are determined. The behavior of these blocks are defined.

3. *Build Verilog module for each building block*: According to the specifications defined in Step 1 and Step 2, we model the building blocks by Verilog.
4. *Generate module testing vectors*: In order to test the Verilog module created in Step 3, we generate module testing vectors by software programs written in C language. The software programs generate the inputs to the modules for all possible situations.
5. *Verilog module testing*: The testing vectors generated in Step 4 are ported to Verilog programs to provide inputs and outputs to the Verilog modules written in Step 3. The Verilog modules are verified for their correct behaviors.
6. *Build Verilog system model*: After all Verilog modules are verified, we create a system model using those modules.
7. *Generate system testing vectors*: Similar to Step 4, we need to generate inputs to the Verilog system model written in Step 6. We write software programs in C language to create inputs for all possible situations.
8. *Verilog system testing*: At this step, we test the Verilog system modeled in Step 6 using the system testing vectors generated in Step 7. The Verilog system model is verified to demonstrate the feasibility of the proposed architectures.
9. *Complexity analysis*: From the Verilog modeling, we analyze the hardware complexity of the proposed architectures. The transistor gate count of each building module can be estimated well by comparing the block with standard hardware synthesis libraries. The overall complexity can be estimated by combining the transistor gate counts of all building blocks plus the overhead from global connections.
10. *Port Verilog modules to C/C++ routines*: In order to simulate more complicated operations, C/C++ routines are built based on the Verilog modules written in Step 3. Since these modules have been verified, the C/C++ routines can be created directly by porting the Verilog codes to the C/C++ codes.

11. *Build C/C++ system model (routine)*: The C/C++ routines from Step 10 are combined to create a system model for the proposed architectures. This model is used as a routine in the simulation program.
12. *Build C/C++ simulation environment*: In order to use the routine for the system model from Step 11, a simulation program is needed to provide the simulation environment and record the operation statistics, such as the number of times a particular operation is executed in a routine.
13. *Build benchmark routines*: The routines extensively used in wireless communications processing are built in C/C++ language. These routines are different from regular C/C++ routines. They are specially written from the simulation environment created in Step 12.
14. *C/C++ simulation*: At this stage, we combine the work done in Step 11, Step 12, and Step 13. The benchmark routines are simulated for the proposed architectures. The statistics of the operations executed are recorded.
15. *Power analysis*: Power analysis is done by translating the statistics of the operations to power consumption. The power consumption of each building block is first estimated by comparing the block with standard hardware synthesis libraries and the associated power consumption statistics. Then, the power consumed in each operation is determined by the resources that the operation utilizes. All the values of power consumption are normalized by the power consumption of an 8-bit addition operation in order to compare with the power consumption of conventional architectures.
16. *Latency analysis*: In order to simplify the comparison between the proposed architectures and conventional architectures, we assume all basic operations, such as multiply and operand move, take one clock cycle to be finished in all architectures. The latency of complicated operations, such as macro-operations, are determined by how many basic operations that they include. The latency of a routine is determined by the latencies of the operations executed in the routine.

In order to compare the proposed architectures with conventional architectures in Steps 9, 15, and 16, a system model for conventional architectures is created. This

conventional system model is based on traditional one-instruction-stream programmable processor architectures. The resources inside this conventional system model are normalized to the resources inside the new novel system architecture proposed in this chapter. Similar to the proposed system architecture, this conventional system model has one multiplier, one shifter, and two ALU's. All of these computational units have 16-bit word lengths. A single register file is used in this conventional system model. In order to match the total number of registers with the proposed system architecture, there are sixty-four registers in this single register file. The data buses and other supporting structures are based on the designs of several of the most popular programmable processors such as TMS320C54x from Texas Instruments [Tex94][Tex96], ADSP-2106x from Analog Devices [Ana95], and others. Although this model is different from each individual design of the conventional architectures that are used as references, it presents well the considerations of the conventional architectures. This representative conventional system model is used to obtain the average performance for conventional architectures that is used in the performance analyses to be discussed in the following sections.

6.3 Reconfigurable Macro-Operation Signal Processing Architecture

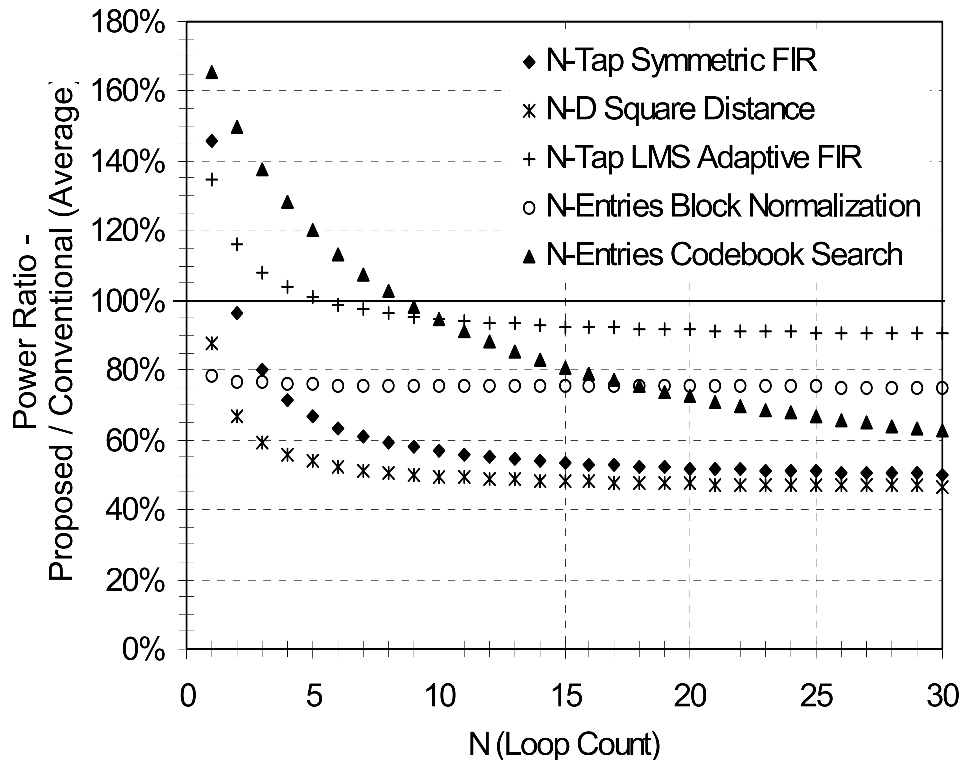


Figure 20 - Power consumption reduction achieved by using reconfigurable macro-operation signal processing architecture. Five benchmark routines are plotted: N-tap symmetric FIR filter, N-dimension square distance, N-tap LMS adaptive FIR filter. N-entries block normalization, and N-entries VSELP codebook search. When the power ratio is lower than 100%, the proposed architecture consumes less power than traditional architectures.

Figure 20 shows the simulation results for the reconfigurable macro-operation signal processing architecture. The Y-axis is the power consumption ratio between the proposed architecture and the average of conventional architectures. Where the ratio is lower than 100%, the proposed architecture consumes less power. The X-axis is the

variable N, that indicates how many times a loop is executed. Five benchmark routines are plotted: N-tap symmetric FIR filter, N-dimension square distance, N-tap LMS adaptive FIR filter. N-entries block normalization, and N-entries codebook search. As Figure 20 shows, all curves start with higher power consumption ratios when the variable N is small, As N increases, the ratios reduce and approach some constant values asymptotically. The initial higher power consumption ratio results from the extra effort needed for setting up a macro-operation. The power consumption ratio of the N-tap symmetric FIR filter starts at 146% when N is equal to one. It drops quickly to a little below 100% power consumption when N is equal to two. This indicates the proposed architecture saves power for a symmetric FIR filter as long as the filter has more than two taps. The power consumption ratio approaches 52% for N larger than 15. Since most, if not all, of the symmetric FIR filters used in wireless communications have more than fifteen taps, it is clearly a good choice to pack several micro-operations into a single macro-operation to execute a symmetric FIR filter. The N-tap adaptive FIR filter has a similar curve. The power consumption ratio starts at 135%. It drops below 100% when N is larger than 5. However, the ultimate power reduction is not as good as for the N-tap symmetric FIR filter. The power consumption ratio approaches 90% for large N. This indicates that only a 10% power reduction is achieved. This is because the software code optimized for an N-tap adaptive FIR filter is too complicated to be handled in a single macro-operation. A macro-operation and a few micro-operations are mixed in a loop to execute the N-tap adaptive FIR filter. On the other hand, it is not complicated enough to yield more improvement. However, the energy efficiency is still enhanced for N larger than 5.

Although extra power is consumed in setting up a micro-operation, some routines using macro-operations have lower power consumption even if we only execute them once. N-dimension square distance and N-entries block normalization are good examples. For N equal to one, the power consumption ratios of both routines are lower than 100%. The flat curve for N-entries block normalization indicates that the routine for N-entries block normalization is relatively less dependent on the number of the entries. The power

reductions for N-entries block normalization and N-dimension square distance approach 55% and 23% separately.

The last routine shown in Figure 20 is an N-entries codebook search. Compared with other routines discussed above, it is much more complicated. The overhead for setting up macro-operations needed in the routine is quite high. The power consumption ratio starts at above 160% for N equal to one. More loops of execution are required to recover from the extra power consumed in set up. When N is larger than 9, it becomes beneficial to utilize macro-operations. Because the routine is complicated, it has more margin to be improved. As N increases, the power consumption ratio drops rapidly to 60%. The performance of the five routines shown in Figure 20 are summarized in Table 6.

Table 6 - Performance summary of reconfigurable macro-operation signal processing architecture. Crossover point is defined as the minimum N for which the power consumption ratio is lower than 100%. Asymptotic power reductions listed are the asymptotic values for large N.

Routine	*Crossover Point*	*Asymptotic Power Reduction*
Symmetric FIR Filter	N=2	48 %
LMS adaptive FIR Filter	N=6	10 %
Square Distance	(N<1)	55 %
Block Normalization	(N<1)	23 %
Codebook Search	N=9	40 %

6.4 Signal Processing Arrangement and Method for Predictable Data

The power reduction achieved by the proposed signal processing arrangement and method for predictable data are listed in Table 7. Eight benchmark routines are listed: 8x-

tap symmetric FIR, symmetric FIR of other tap numbers, 4x-tap LMS adaptive FIR, sum of squares, maximum vector value, discrete cosine transform, radix-2 FFT (256 points), Viterbi decoding (GSM), and VSELP codebook search. Unlike for the proposed macro-operation signal processing architecture, there is no extra power consumption due to initial setup. The eight benchmark routines gain 6% to 29% power reductions from the proposed data management architecture. In general, the proposed architecture can save more power if more data are moved among different memory locations in a more complicated way. This is because there are more unnecessary data movements in such a routine written for conventional architectures. The total number of data movements can be reduced significantly in the proposed architecture. On the other hand, a routine with very simple data movements cannot be modified much. There is little margin for improvement. For example, the routine for sum of squares is relatively simple. The proposed architecture can provide only 6% power reduction. However, there are many complicated routines in wireless communications processing. In most cases, splitting a software program into two instruction streams provides a great deal of extra margin to improve the power consumption.

As mentioned in Chapter 4, we limit the block data transfers to several patterns: full register file, half register file, full row of register files, and half row of register files. Therefore, the size of a single register file and the number of register files in the register bank have important impact on the overall performance. In general, a routine can be better optimized if the number of operand transfers to or from the register file bank is equal to a multiple of the size of a single register file or the number of register files in the register bank. In such cases, neither zero padding nor extra micro-operations are needed.

Table 7 - Summary of power reduction achieved by the proposed signal processing arrangement and method for predictable data.

Routine	*Power Reduction*
Symmetric FIR (8x-tap)	12 %
Symmetric FIR (others)	9 %

LMS Adaptive FIR (4x-tap)	17 %
Sum of Squares	6 %
Maximum Vector Value	11 %
Discrete Cosine Transform	13 %
Radix-2 FFT (256 points)	24 %
Viterbi Decoding (GSM)	29 %
VSELP Codebook Search	21 %

Table 8 - Summary of latency reduction achieved by the proposed signal processing arrangement and method for predictable data.

Routine	*Latency Reduction*
Symmetric FIR (8x-tap)	25 %
Symmetric FIR (others)	21 %
LMS Adaptive FIR (4x-tap)	21 %
Sum of Squares	7 %
Maximum Vector Value	19 %
Discrete Cosine Transform	25 %
Radix-2 FFT (256 points)	33 %
Viterbi Decoding (GSM)	28 %
VSELP Codebook search	37 %

If the numbers from hardware structures and software routines do not match, extra effort is needed to finish the routine. Therefore, the power consumption is less optimized. This effect is illustrated by the symmetric FIR filters listed in Table 7. Because we use eight register files with eight registers each in the simulation, a symmetric FIR filter with tap numbers of multiples of eight gains more power reduction than a symmetric FIR filter with other numbers of taps. The power reductions listed in Table 7 are 12% and 9% for these two cases.

The latency reductions for the same benchmark routines used above are listed in Table 8. The reductions range from 7% to 37%. Similar to the power reduction, the more complicated the data structures are, the more reduction can be achieved in the execution latency. Similar effects from the size of register and the number of register files in the register file bank also exist in the latency reduction. The mismatch between hardware specifications and software routines results in somewhat longer execution latency.

6.5 System Performance

This section summarizes the performance analyses from the simulations for the proposed overall system architecture. Compared with the average for conventional architectures obtained from the conventional system model described in Section 6.2, the proposed system architecture takes advantage of the inherent characteristics of wireless communications processing and achieves low power, low latency, and low complexity in tradeoff with software programming complexity.

6.5.1 Low Power

Combining the advantages provided by the reconfigurable macro-operations and the special data management scheme, the proposed overall system architecture reduces power consumption up to 55% for the routines extensively used in wireless communications processing.

6.5.2 Low Latency

The execution latency is reduced up to 37% for the routines extensively used in wireless communications processing. The latency reduction mainly results from the proposed data management architecture used in the overall system architecture. The proposed

reconfigurable macro-operation signal processing architecture has less impact on the execution latency of the overall system architecture. Packing several micro-operations into a single macro-operation does not reduce the total execution latency except for the portion contributed by unnecessary data movements in conventional architectures.

6.5.3 Low Complexity

Compared with conventional programmable processor architectures with the same computational resources, the complexity analysis shows 25% hardware overhead resulting from the extra building blocks for the reconfigurable macro-operations and the second instruction stream. However, the proposed system architecture achieves much higher efficiency. If we compare the proposed system architecture with those architectures with the same level of efficiency, for example, single-instruction-multiple-data (SIMD) architectures, the proposed system architecture is much simpler. It achieves similar efficiency without sacrificing hardware simplicity. The proposed architecture also merges the functionalities of micro-controller and digital signal processor. It eliminates the need for using two processors. Compared with the complexity of a simple conventional digital signal processor plus a microcontroller, the complexity of the proposed architecture is much lower.

6.5.4 Software Programming Complexity

The reconfigurable macro-operations and the background data management increase the complexity of software programming. In order to take advantage of macro-operations, software routines need to be analyzed and micro-operations need to be re-arranged and packed into macro-operations. In addition, all data movements need to be controlled by software programs. Each processing routine needs a counterpart for data management. As illustrated in Section 4.3 and Section 5.2, the software programming procedure is more complicated than in conventional processor architecture. However, wireless

communications software for protocol processing is programmed only once and embedded into read-only-memory. It is seldom modified in a particular wireless transceiver unless the protocol changes. For mobile wireless communications processing, the importance of lower power consumption surpasses the extra burden on software programming. Therefore, we have the opportunity to tradeoff the software programming complexity to gain performance in real-time execution.

6.6 Summary

An innovation is not practical unless it can be implemented in the real world. Theoretical discussions also need to be supported by quantitative studies.

This chapter presents our work on demonstrating the feasibility of the proposed architecture and quantitatively evaluating performance through simulations. Compared with conventional architectures which are represented by the conventional system model discussed in Section 6.2, up to 55% power reduction and up to 37% latency reduction are achieved by the proposed architectures for representative routines extensively used in wireless communications processing. The proposed architectures also reduce the overall complexity by merging the functionalities of microcontroller and digital signal processor.

In the next chapter, we conclude this dissertation by summarizing the previous chapters and listing several topics that might be of interest for future research.

Chapter 7

Conclusions

In the past, wireless communications standards were adopted to focus on some specific user sectors and technologies. These standards were developed independently without the consideration of being compatible with other standards. As the technologies in wireless communications and integrated circuits advanced, the popularity of wireless communications increased dramatically. It became the fastest growing segment of the telecommunications market. The usage of different wireless communications applications started to overlap and create a need for an integrated environment across different geographical locations and technologies. Thus, multiple standard wireless communications is now in demand. In this dissertation, we preset some new innovations that can be used to provide a low-power flexible hardware platform for multiple standard wireless communications processing. In Section 7.1, we summarize the discussions in each chapter. In Section 7.2, we list several topics that might be of interest for future research.

7.1 Dissertation Summary

Chapter 2 of this dissertation describes general backgrounds of programmable processors and wireless communications. From a historical point of view, the development of the

general purpose microprocessor had the biggest influence on the evolution of programmable processors. These microprocessor architectures were invented in order to process general computational tasks as fast as possible. Digital signal processors (DSP's) were first invented as co-processors to microprocessors and were used to break through the bottleneck created by extensive computing requirements that were beyond the capability of microprocessors. Because of the consideration of executing speed, DSP structures were less flexible and generally unsuitable for process controlling. Therefore, the unified microcontroller-DSP was introduced recently. No matter what category a programmable processor falls into, the primary concern in the architecture has been the execution speed. Although power consumption also was discussed for these architectures, most attention was paid to reducing the power consumption of an existing architecture which was designed for high speed. In mobile wireless communications processing, however, power consumption is more important than high speed since the modern VLSI technologies have created computing power more than enough for wireless protocol processing. In order to provide low power technologies for programmable processors for mobile wireless communications, we discussed the inherent characteristics of wireless communications processing: determinism, repetition, just-in-time process, limited but more complicated subset of operations, less control needed, large internal dynamic range but low output resolution, input/output intensive and heavy internal data flows, and one time programming software. By taking advantage of these characteristics, we can achieve lower power consumption without losing the programmability and flexibility, which are needed for handling a number of different protocols in a single mobile unit. In Chapter 2, we also point out some factors that distinguish different processor architectures: operand transfer scheme, instruction length and code density, parallelism, pipelining, memory hierarchy, interrupt handling, and ASIC-orientation. With these factors in mind, we can realize what is changed when we propose a new processor architecture.

In Chapter 3, we present the first new innovation in this research. A reconfigurable macro-operation signal processing architecture is described. The idea of reconfigurable macro-operations takes advantage of the determinism and repetition in wireless communications processing. We pre-arrange several arithmetic micro-operations

and pack them to create a more complicated but more efficient macro-operation. Because of the determinism, we can decide in advance what the optimized macro-operation is in a particular time slot and when we should switch from one macro-operation to another. The instruction format of the proposed macro-operations is simple and compact. This reduces the effort in decoding several full micro-operations. Following the assignments in a macro-operation, the operands are fed back to computational resources without being stored into registers redundantly. This eliminates unnecessary data movement. The tradeoff is the need to reconfigure the datapath for macro-operations. However, because of the repetition property of wireless communications processing, we usually use a macro-operation many times before we switch to another macro-operation. The overhead from macro-operation setup becomes insignificant compared to the efficient repeating of the processing steps. In Chapter 3, we describe the building blocks of the proposed architecture and the instruction format for macro-operations. The building blocks of the proposed architecture include macro-operation instruction decoder, decoded macro-operation instruction register file, macro-operation management unit (MOMU), computational units, macro-operation connection multiplexer, macro-operation input registers, macro-operation output/interrupt registers, 8-channel memory loading DMA controller, 4-channel memory storing DMA controller, DMA source address registers, and DMA destination address registers. The reconfigurable feature is achieved by a special arrangement of the data feedback path and the controlling mechanism included in the 10-byte macro-operation instruction format. A single compact macro-operation provides all information needed for hardware configurations, data movements, operation timing, loop control, etc. The code density is increased significantly. Operands are also transferred more efficiently. We discuss how the procedure goes and how different operation modes can be utilized. The four different operation modes - new-instruction loop, stored-instruction loop, new-instruction discrete, and stored-instruction discrete modes - provide the flexibility for software code optimization for different signal processing tasks. Before we conclude Chapter 3, we discuss the impact of the proposed architecture on those architecture factors mentioned in Chapter 2. The proposed architecture takes advantage of the high code density of CISC architectures to reduce

memory traffic. It also makes the processors more ASIC-oriented and drives the power consumption down. Even with these improvements, this innovation still retains simplicity of system hardware in the tradeoff with software complexity.

In Chapter 4, we present the second new innovation in this research, a signal processing arrangement and method for predictable data. This provides a low power data management architecture for multiple standard mobile wireless communications processing. In the proposed signal processing arrangement and method, the architecture is separated into two portions controlled by two separated instruction streams: computational codes and data management codes. This changes the philosophy away from the one-instruction-stream architectures that have been used for a long time. Two-instruction-stream architecture becomes practical because the tasks in wireless communications processing are highly deterministic and future operands are highly predictable. The synchronization of two instruction streams can be pre-arranged. This eliminates the need for a sophisticated instruction scheduler, which is usually essential in non-deterministic environments. The parallel processing of two instruction streams gives us another dimension for optimization. In the proposed architecture, the instructions for computational codes are simplified by limiting the data movements to register-to-register movements only. On the other hand, by assigning active input and output register files, we separate logical access and physical access to those register files. This not only further simplifies the instructions of computational codes, but also increases the reuse frequency of computational codes. Furthermore, the proposed architecture enhances the flexibility of matrix and vector operations by providing special orthogonal accesses to the register file bank. In Chapter 4, we describe the building blocks of the proposed architecture. We explain how the data management codes and the computational codes work together and the special features of the register file bank. We describe the procedure for arranging the software codes for the proposed architecture. Similar to Chapter 3, we discuss the impact of the proposed architecture on those architecture factors at the end of Chapter 4. Separating software routines into computational codes and data management codes provides a new novel data management architecture with higher memory efficiency and software code efficiency for the environment of wireless

communications processing. This new invention provides direct control over data and a second dimension for optimizing the power consumption. All instructions are simple and straightforward. The combination of operations in the two dimensions provides the flexibility and energy efficiency that are needed in multiple standard mobile wireless communications.

In order to utilize both innovations discussed in Chapter 3 and Chapter 4, a new overall system architecture is introduced in Chapter 5. The universal operand access method significantly reduces the effort for switching between micro-operations and macro-operations. With the novel arrangement inside the register file bank, the operation of switching between micro-operations and macro-operations becomes simply re-assigning the active register files. In Chapter 5, we discuss the building blocks of the register file bank and how it provides an universal operand access method for both micro-operations and macro-operations. We also explain the software partition between the computational codes and the data management codes. The software programming procedure is discussed briefly before we conclude Chapter 5. This new overall system architecture merges the structures and the benefits of the reconfigurable macro-operation signal processing architecture discussed in Chapter 3 and the signal processing arrangement and method for predictable data discussed in Chapter 4. The novel arrangement inside the register file bank provides smooth transitions between macro-operations and micro-operations and fulfills the requirements for a true unified microcontroller-DSP architecture. The different arrangements for logical access and physical access also reduce the system complexity.

Chapter 6 describes the performance evaluation of the proposed low power architectures. The proposed architectures are modeled at register-transfer-level (RTL) by the hardware description language, Verilog, to demonstrate the feasibility. The routines extensively used in wireless communications processing are simulated in high level language C/C++. The discussion of overall performance is also presented in Chapter 6. Compared with conventional programmable processor architectures, up to 55% power reduction and up to 37% latency reduction are achieved by the proposed architectures for those representative routines extensively used in wireless communications processing.

The proposed architectures also reduce the overall complexity by merging the functionalities of microcontroller and digital signal processor.

7.2 Future Work

Several topics that might be of interest for future research are listed in this section.

7.2.1 Compiler Technology

One of the most challenging problems in utilizing the innovations presented in this dissertation is the difficulty of writing efficient software programs for the proposed system. Although it can be done and it is not very unusual to hand code each line of a software program in low-level programming language [Ana95], it is certainly painful and time-consuming. An intelligent software compiler that can compile software programs written in high-level languages into machine codes for the proposed system is highly desirable.

7.2.2 Memory Layer Structure

In this dissertation, we discuss the method of transferring data among these different memory spaces. We use the proposed register file bank and assume several memory layers such as level one-cache, level-two cache, and main memory. We did not specify the detailed arrangements of these memory layers, for example, the size of the level-one cache. Some topics that need to be investigated are listed in the following.

- How many layers of memory hierarchy should be used in a system? What is the size of each layer? What is the speed requirement for each layer?
- What is the bus structure between two memory layers?

- Which portion of data transfer should be assigned by software and which portion can be handled by hardware as before? What are the occasions for which a software command can overrule a hardware decision?

7.2.3 Variations and Scalability

In order to minimize the system complexity, the philosophy behind the proposed architecture is not to add more resources to the system. We use only one multiplier, one shifter, and two ALU's in the proposed system. Very limited resources are added to handle the second instruction stream and the special register bank. Although the proposed technologies are based on these limited resources, the ideas are scalable. This creates many possible variations if more resources are added. For example, more computational units can be added to provide more complicated and more efficient macro-operations. Adding different kinds of computational units also creates different sets of macro-operations. Also, the instruction streams can be further split into more sub-streams to serve more dedicated functionalities. Different variations provide different sets of criteria for optimization. The investigation of the tradeoffs among these variations is a valuable topic for future research.

Bibliography

[Abn96] A. Abnous and J. Rabaey, "Ultra-Low-Power Domain-Specific Multimedia Processor", *VLSI Signal Processing*, Vol XI, pp. 461-470, IEEE, 1996

[Ana95] *ADSP-2106x SHARC User's Manual*, Analog Devices, 1995

[Bai95] R. Baines, "The DSP Bottleneck", *IEEE Communications Magazine*, Vol. 33, No. 5, pp. 46-54, May 1995

[Cox87] D. C. Cox, "Universal Digital Portable Radio Communications", IEEE Proceedings, Vol. 75, No. 4, pp.436-477, April 1987.

[Cox91] D. C. Cox, "A Radio System Proposal for Widespread for Low-Power Tetherless Communications", *IEEE Transactions on Communications*, Vol. 39, No. 2, pp. 324-335, February 1991.

[Cox92] D. C. Cox, "Wireless Network Access for Personal Communications", *IEEE Communications Magazine*, Vol. 30, No. 12, pp. 96-115, December 1992.

[Cox95] D. C. Cox, "Wireless Personal Communications: What Is It*", IEEE Personal Communications Magazine*, Vol. 33, No. 4, pp. 20-35, April 1995.

[Cra97] J. Crawford, "Next Generation Instruction Set Architecture", *Microprocessor Forum*, San Jose, USA, October 1997.

[Fle97] R. Fleck, "A Truly Unified Microcontroller-DSP Architecture", *Microprocessor Forum*, San Jose, USA, October 1997.

[Goo91] D. J. Goodman, "Trends in Cellular and Cordless Communications", *IEEE Communications Magazine*, Vol. 29, No. 6, pp. 31-40, June 1991

[Hen95] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach*, Morgan Kaufmann, 1995

[Int95] *Pentium® Processor Family Developer's Manual Volume 1~3*, Intel, 1995

[Int96] *Intel Architecture MMX™ Technology - Programmer's Reference Manual*, Intel, 1996

[Kru91] R. L. Kruse, B. P. Leung, and C. L. Tondo, *Data Structures and Program Design in C*, Prentice-Hall, 1991

[Lac95] R. J. Lackey and D. W. Upmal, "Speakeasy: The Military Software Radio", *IEEE Communications Magazine*, Vol. 33, No. 5, pp. 56-61, May 1995

[Lee97] Y. Lee, *Adaptive Equalization and Receiver Diversity for Indoor Wireless Data Communications, Ph.D. Dissertation*, Stanford University, 1997

[Mad95] V. K. Madisetti*, VLSI Digital Signal Processors - An Introduction to Rapid Prototyping and Design Synthesis*, IEEE Press, 1995

[McC86] E. J. McCluskey, *Logic Design Principles with Emphasis on Testable Semicustom Circuits,* Prentice-Hall, 1986

[McC93] S. C. McConnell*,* *Code Complete - A Practical Handbook of Software Construction*, Microsoft Press, 1993

[Mey92] S. Meyers, *Effective C++ - 50 Specific Ways to Improve Your Programs and Designs*, Addison-Wesley, 1992

[Mic92] *Microsoft MASM Programmer's Guide*, Microsoft, 1992

[Mit95] J. Mitola, "The Software Radio Architecture", *IEEE Communications Magazine*, Vol. 33, No. 5, pp. 26-38, May 1995

[Nat97] *The Evolution of Untethered Communications,* Computer Science and Telecommunications Board, National Research Council, National Academy Press, 1997

[Nor95] P. Norton, *Inside the PC*, Sams Publishing, 1995

[Rap96] T. S. Rappaport, *Wireless Communications -Principles and Practice*, Prentice-Hall, 1996

[Raz94] B. Razavi, "Low-Power Low-Voltage Design - An Overview", International Journal of High Speed Electronics and Systems, Vol. 5, No. 2, pp. 145-157, June 1994.

[Skl88] B. Sklar, *Digital Communications Fundamentals and Applications*, Prentice-Hall, 1988

[Ste93] E. Sternheim, R. Singh, R. Madhavan, and Y. Trivedi, *Digital Design and Synthesis with Verilog HDL*, Automata, 1993

[Str91] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1991

[Tex94] *Telecommunications Applications with the TMS320C5x DSP's*, Texas Instrument, 1994

[Tex96] *TMS320C54x DSP Reference Set Volume 1~3*, Texas Instruments, 1996

[Ver96] I. Verbauwhede, M. Touriguian, K. Gupta, J. Muwafi, K. Yick, and G. Fettweis, "A Low Power DSP Engine For Wireless Communications", *VLSI Signal Processing*, Vol XI, pp. 471-480, IEEE, 1996

[Wes88] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley, 1988

[Won97] P. B. Wong, *Low-Power Low-Complexity Diversity Combining Algorithm and VLSI Circuit Chip for Hand-Held PCS Receivers*, Ph.D. Dissertation, Stanford University, 1997