University Library

Author/Filing Title ....KOUTSOMYTI, K.

Class Mark ....T

Please note that fines are charged on ALL overdue items.

0403694744

A Configurable Vector Processor for Accelerating Speech Coding Algorithms

By

Konstantia Koutsomyti, MSc, BEng (Hons)

A Doctoral Thesis submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy of Loughborough University

September 2007

Loughborough University
Pilkington Library
Date 9/3/09
Class T
Acc No. 0403694744

To my family

ABSTRACT

The growing demand for voice-over-packer (VoIP) services and multimedia-rich applications has made increasingly important the efficient, real-time implementation of low-bit rates speech coders on embedded VLSI platforms. Such speech coders are designed to substantially reduce the bandwidth requirements thus enabling dense multi-channel gateways in small form factor. This however comes at a high computational cost which mandates the use of very high performance embedded processors.

This thesis investigates the potential acceleration of two major ITU-T speech coding algorithms, namely G.729A and G.723.1, through their efficient implementation on a configurable extensible vector embedded CPU architecture. New scalar and vector ISAs were introduced which resulted in up to 80% reduction in the dynamic instruction count of both workloads. These instructions were subsequently encapsulated into a parametric, hybrid SISD (scalar processor)-SIMD (vector) processor. This work presents the research and implementation of the vector datapath of this vector coprocessor which is tightly-coupled to a Sparc-V8 compliant CPU, the optimization and simulation methodologies employed and the use of Electronic System Level (ESL) techniques to rapidly design SIMD datapaths.

ACKNOWLEDGEMENTS

I would like to thank my supervisors Dr. Sekharjit Datta and especially Dr. Vassilios A. Chouliaras for the continuous guidance and support throughout all the stages of this work. Their advice has been invaluable.

I would deeply like to thank a very special person in my life, Vasilis, for his continuous support and understanding through all these years. He drove me away from my little home in Rafina and gave me the greatest opportunity of all, to open my mind, to learn and believe in myself. Without him none of these would have happened.

I would like to express my deep gratitude and love to my family for always standing by me and supporting me to follow my dreams and especially my beloved mother who taught me to always aim high.

I acknowledge all my colleagues in Loughborough University and especially Tom Jacobs for their support and companionship throughout these years. All the wonderful people I met during my years in Loughborough and have become good friends have contributed even without knowing to this work by making these years special.

Finally, I would like to express my gratitude to the EPSRC for providing me with financial support during the course of this thesis.

TABLE OF CONTENTS

List of Figures ix

List of Tables xiii

List of Abbreviations xv

Chapter 1 Introduction 1

1.1 Problem Formulation 1

1.2 VoIP 4

1.2.1 Description of the VoIP process 5

1.2.2 VoIP Applications 8

1.2.3 Current state of the art 8

1.3 Programmable Architectures 9

1.3.1 General Purpose Processors 9

1.3.2 DSP Processors 10

1.3.3 ASIC (Embedded) processors 10

1.3.3.1 Configurable processors 10

1.3.3.2 Reconfigurable Processors 11

1.3.3.3 Fixed Processors 12

1.4 Hardwired Architectures 12

1.5 Research contribution and overview 13

1.6 Thesis Outline 16

1.7 References 18

Chapter 2 Speech Coding Theory 22

2.1 Introduction 22

2.2 Speech Coding Objectives and Requirements 22

2.3 Speech production system 24

2.4 Coding strategies 26

2.4.1 Waveform Coders 26
2.4.2 Voice Coders (Vocoders) 27
2.4.3 Hybrid Coders 28
2.4.3.1 Analysis by Synthesis 29
2.5 G.729A Speech Coding Standard 31
2.6 G.723.1 Speech Coding Standard 33
2.7 Summary 36
2.8 References 37

Chapter 3 Software and Hardware Parallelism 38

3.1 Overview of Parallelism 38
3.2 Data Dependences 39
3.2.1 Name Dependences 40
3.2.2 Control Dependences 40
3.3 Types of Parallelism 41
3.3.1 Instruction Level Parallelism 42
3.3.1.1 Superscalar Processors 44
3.3.1.2 VLIW Processors 46
3.3.2 Data Level Parallelism 48
3.3.2.1 Advantages of vector architectures 48
3.3.2.2 Vector Processors 50
3.3.3 Thread Level Parallelism 52
3.3.3.1 Shared-Memory Architecture 53
3.3.3.2 Distributed-Memory Architecture 54
3.3.3.3 Multithreading Architecture 55
3.3.4 Hybrid Approaches and Research 56
3.4 Summary 57
3.5 References 58

Chapter 4 Methodology and Architectural Results 62

4.1 Introduction 62

4.2 Simulation Infrastructure 62
4.2.1 SimpleScalar Toolset 66
4.2.2 Customizing the SimpleScalar Toolset 68
4.3 Workload Optimization 69
4.3.1 Profiling 69
4.3.2 Vector ISA Development and Experimentation Methodology 74
4.3.3 Identification of Data Parallel Loops 78
4.3.4 Implementation of vector loop using custom ISA 80
4.3.5 Scalar Optimization 83
4.3.6 Validation Tests 84
4.3.7 The extended ISA (Scalar and Vector Extensions) 86
4.3.8 Inline Assembly 87
4.4 Architectural Results 88
4.5 Summary 102
4.6 References 104

Chapter 5 Vector Processor Architecture 107

5.1 Vector Architectural State 107
5.2 Programmers Model 109
5.3 Vector Processor Instruction Set Architecture 110
5.3.1 Vector ISA 111
5.3.1.1 Load/Store Instructions 111
5.3.1.2 Move Instructions 112
5.3.1.3 Arithmetic Instructions 113
5.3.1.4 Shift Instructions 116
5.3.1.5 Miscellaneous Instructions 117
5.3.2 Scalar ISA 118
5.3.2.1 Load/Store Instructions 118
5.3.2.2 Move Instructions 118
5.3.2.3 Arithmetic Instructions 119
5.3.2.4 Shift Instructions 119

5.3.2.5 Miscellaneous Instructions 120
5.4 Leon3 CPU 120
5.5 Overall System Architecture 124
5.5.1 Processor-coprocessor programmable unit 125
5.5.2 DMA taps 125
5.5.3 PCI I/F 125
5.5.4 External Memory Controller 125
5.5.5 APB Subsystem 126
5.6 Summary 126
5.7 References 127

Chapter 6 Vector Processor Implementation 128

6.1 Overview 128
6.2 Vector Decode Stage (VDEC) 130
6.3 Vector Registers Stage (VREG) 133
6.3.1 Reverse Data Process 134
6.3.2 Splat Data Process 135
6.3.3 Masking Process 135
6.3.4 Bypass process 137
6.3.5 Operands Selection 139
6.3.6 Register enable 139
6.3.7 Vector Register File (gxx_vreg_file) 140
6.3.7.1 Parameterisation 140
6.3.7.2 The vector register file implementation 140
6.3.8 Scalar Register File (gxx_sreg_file) 143
6.3.8.1 Parameterisation 143
6.3.8.2 Scalar register file implementation 143
6.3.9 Vlen register 145
6.3.10 Overflow and Pred Flags 146
6.4 Vector Load/Store Unit (gxx_vlsu) 146
6.5 Vector Datapath Stage (VDP) 149

6.5.1 Vector Adder Unit (gxx_vadd_dp) 152
6.5.2 Vector Multiplier Unit (gxx_vmult_dp) 153
6.5.3 Vector Shifter Unit (gxx_vshift_dp) 155
6.5.4 Vector Miscellaneous Unit (gxx_vmisc_dp) 158
6.5.5 Reverse Data Logic 158
6.5.6 Masking Process Logic 159
6.5.7 Bypassing network of the first VDP stage 160
6.5.8 Register Enable for the input VDP2 registers 160
6.5.9 Second stage adder 160
6.5.10 Vector Accumulator File (gxx_vaccs) 161
6.5.10.1 Parameterisation 162
6.5.10.2 The vector accumulator implementation 163
6.5.11 Vector Adder Tree (gxx_adder_tree) 164
6.5.12 VLSU unit interface with VDP2 165
6.5.13 Overflow and Predicate Flags 165
6.5.14 Bypassing network of the second stage 166
6.5.15 Write Back 166
6.6 Output Register Bunch 166
6.7 Leon3 166
6.7.1 Decode Stage 167
6.7.2 Register Access stage 167
6.7.3 Execute Stage 168
6.7.4 Memory Stage 168
6.7.5 Exception Stage 168
6.8 Summary 170
6.9 References 171

Chapter 7 Vector Processor VLSI Implementation 172

7.1 Design Verification 172
7.2 Synthesis and Place & Route Design Flow 174
7.2.1 Design Compiler Stage (Logical Synthesis) 175

7.2.2 SoC Encounter script Stage (Place and Route) 176
7.2.3 Statistical Power Analysis Stage (Design Compiler) 176
7.3 Implementation Campaign for Vector Datapath 177
7.4 Implementation Campaign for Vector Coprocessor 179
7.5 VLSI Layout 182
7.5.1 Vector Datapath Layout for VLMAX 16 182
7.5.2 Vector Datapath Layout for VLMAX 32 184
7.5.3 Vector Processor Layout for VLMAX 16 185
7.6 ESL Implementation 187
7.6.1 SS_SPARC Platform 187
7.6.2 ESL Methodology 191
7.6.3 Micro-Architecture Results 191
7.7 Summary 194
7.8 References 195

Chapter 8 Conclusions 196

8.1 Contribution of this thesis 196
8.2 Suggestions for future research 198
8.3 References 200

Appendix A Vector and Scalar ISA 201

Appendix B Signal Description 244

Appendix C G.729A and G.723.1 Function Results 246

Author's Publications 260

LIST OF FIGURES

Figure 1-1: Traditional voice and data networks (a) and VoIP network (b) 1
Figure 1-2: The architecture of H.323 protocol stack 2
Figure 1-3: Simplified representation of possible IP telephony network connections 5
Figure 1-4: VoIP signalling and transport flow between endpoints 6
Figure 1-5: Open Systems Interconnection (OSI) and network protocols 7
Figure 2-1: Diagram of the human organs involved in speech production and the Spectral Range of Speech 24
Figure 2-2: General speech production model 25
Figure 2-3: Analysis by Synthesis Code 30
Figure 2-4: G.729A Encoder 32
Figure 2-5: G.729A Decoder 33
Figure 2-6: G.723.1 Encoder 35
Figure 2-7: G.723.1 Decoder 35
Figure 3-1: Code snippet that shows the data dependences 39
Figure 3-2: Multiple-issuing of instructions in an ILP architecture 43
Figure 3-3: Dynamic Instruction Scheduling 45
Figure 3-4: Static Instruction Scheduling 46
Figure 3-5: Basic Vector Processor Architecture 52
Figure 3-6: The basic architecture of a centralised shared-memory multiprocessor system 53
Figure 3-7: The basic architecture of a distributed-memory multiprocessor system 54
Figure 4-1: SimpleScalar Infrastructure 67
Figure 4-2: Machine instruction count for the BASOP.C functions 71
Figure 4-3: Experimentation Methodology 75
Figure 4-4: The extended processor state as defined in the configuration file vstate.h 76
Figure 4-5: Example of a C macro Instruction Definition 77
Figure 4-6: Example of a non-vectorizable loop as the statement S5 depends on a previous result of the S5 execution. The same dependency appears to the statement S9. 79
Figure 4-7: Example of a vectorizable loop with statements S2 and S3 being independent from previous results of their execution. 79
Figure 4-8: Example of loop with DLP within the original C code 80
Figure 4-9: Assign pointers and load the vlen_r register 81
Figure 4-10: Main vector loop 81
Figure 4-11: Strip mining loop 82

Figure 4-12: Scalar optimization example........84
Figure 4-13: Instruction Definition in Vector.def........86
Figure 4-14: Inline Assembly Instruction Definition........87
Figure 4-15: G.729A Encoder (Vector Only) Results........89
Figure 4-16: G.729A Decoder (Vector Only) Results........90
Figure 4-17: G.729A Encoder (Full Optimization) Results........90
Figure 4-18: G.729A Decoder (Full Optimization) Results........91
Figure 4-19: G.723.1 Encoder Vector Optimization Results........92
Figure 4-20: G.723.1 Decoder Vector Optimization Results........92
Figure 4-21: G.723.1 Encoder Full Optimization Results........93
Figure 4-22: G.723.1 Decoder Full Optimization Results........93
Figure 4-23: Cor_h_x (Full Optimization) Results........94
Figure 4-24: Syn_filt (Full Optimization) Results........95
Figure 4-25: Pitch_ol_fast (Full Optimization) Results........96
Figure 4-26: Residu (Full Optimization) Results........96
Figure 4-27: Autocorr (Full Optimization) Results........97
Figure 4-28: Lsp_pre_select (Full Optimization) Results........98
Figure 4-29: Agc (Full Optimization) Results........98
Figure 4-30: Find_Best (Full Optimization) Results........99
Figure 4-31: Estim_Pitch (Full Optimization) Results........100
Figure 4-32: Comp_Lpc (Full Optimization) Results........101
Figure 4-33: Decod_Acbk (Full Optimization) Results........101
Figure 4-34: Comp_Pw (Full Optimization) Results........102
Figure 5-1: Example of an operation that is performed in two vector registers with vector length 64-bits. Each functional unit is driven by the pair of the corresponding slices (vector elements) of the source vector registers. The produced results are stored back to the corresponding slices (vector elements) of the destination vector register........108
Figure 5-2: Vector and Scalar coprocessor programmer's model........109
Figure 5-3: Vector Short Addition........114
Figure 5-4: Vector Short Multiplication for even/odd elements........114
Figure 5-5: Vector multiply-add/sub........115
Figure 5-6: Instruction Formats of Leon3........122
Figure 5-7: Unimplemented Instruction........122
Figure 5-8: Overall system architecture........124

Figure 6-1: The vector speech coprocessor microarchitecture with the four-stage pipeline: Vector Decode Stage (VDEC), Vector Register Access Stage (VREG) and two stages for the Vector Datapath Stage (VDP1 and VDP2) 129
Figure 6-2: The electrical interface of the VDEC Stage 131
Figure 6-3: The Unimplemented instruction format of the Sparc V8 architecture 131
Figure 6-4: Different types of instruction formats of the vector processor ISA 132
Figure 6-5: Vector Register Access Stage (VREG) microarchitecture 134
Figure 6-6: Reverse Data Process 134
Figure 6-7: Splat Data Process 135
Figure 6-8: Mask width function 136
Figure 6-9: Mask extract function 136
Figure 6-10: Vector bypass process for one of the vector source operands and the intermediate result of one of the two VDP stages 138
Figure 6-11: Scalar bypass process for the selection of one of the scalar operands (first) 138
Figure 6-12: Electrical Interface of Vector Register File 141
Figure 6-13: Detailed microarchitecture of the Vector Register File with R/W conflict avoidance 142
Figure 6-14: Electrical Interface of Scalar Register File 143
Figure 6-15: Detailed microarchitecture of the Scalar Register File with R/W conflict avoidance 144
Figure 6-16: VLSU Electrical Interface 147
Figure 6-17: Parallel TAG/DATA configuration and Cascade TAG/DATA configuration caches 148
Figure 6-18: Microarchitecture of VLSU in cascade TAG/DATA configuration 149
Figure 6-19: Microarchitecture of the VDP stage 150
Figure 6-20: Electrical interface of the vector adder unit 152
Figure 6-21: Microarchitecture of a functional unit of the vector adder 153
Figure 6-22: Electrical interface of the vector multiplier unit 154
Figure 6-23: Microarchitecture of a functional unit of the vector multiplier 155
Figure 6-24: Electrical interface of the vector shifter unit 156
Figure 6-25: Two Barrel Shifters connected in series for short or long shift operations 156
Figure 6-26: Microarchitecture of a functional unit of the vector shifter 157
Figure 6-27: Electrical interface of the vector miscellaneous unit 158
Figure 6-28: Masking process logic for low (vrf_opr2_r='1') or high (vrf_opr2_r='0') deposit for the even elements of the input vectors to the accumulator 159
Figure 6-29: Electrical interface of the second VDP stage vector adder 161

Figure 6-30: Electrical Interface of Vector Accumulator File 162
Figure 6-31: Write data and write-enable selection logic for the vector accumulator file 163
Figure 6-32: Adder tree configuration for VLMAX 16 164
Figure 6-33: Leon3 integer unit and vector coprocessor datapath diagram 169
Figure 6-34: Leon3 processor core block diagram 170
Figure 7-1: Example of recording the inputs and the outputs of the L_mult operation C macro .. 172
Figure 7-2: Test bench for the vector mult unit of the vector datapath 173
Figure 7-3: Vector coprocessor testbench configuration 174
Figure 7-4: Script in a pseudocode for the design flow of the vector coprocessor 175
Figure 7-5: Statistical power results of vector datapath for different vector lengths 177
Figure 7-6: Statistical area results of vector datapath for different vector lengths 178
Figure 7-7: Frequency results of vector datapath for different vector lengths 179
Figure 7-8: Statistical power results of vector coprocessor for different vector lengths 180
Figure 7-9: Statistical area results of vector coprocessor for different vector lengths 181
Figure 7-10: Frequency results of vector coprocessor for different vector lengths 182
Figure 7-11: Vector Datapath macrocell for VLMAX 16 183
Figure 7-12: Vector Datapath macrocell for VLMAX 32 185
Figure 7-13: Layout for the whole vector processor (vector datapath and VLSU unit) 186
Figure 7-14: High level view of a 3-instance SS_SPARC kernel 187
Figure 7-15: Superscalar SMT pipeline organisation 188
Figure 7-16: Scalar core (SCORE) pipeline organization 189
Figure 7-17: Dual-pipeline vector unit organization 190
Figure 7-18: ITU VCore Power Results 192
Figure 7-19: ITU VCore Area-Delay Results 193
Figure 7-20: Two-context, 256-bit ITU vector engine 193

LIST OF TABLES

Table 1-1: ITU Standards for Voice Compression 4
Table 4-1: SimpleScalar baseline simulator models 67
Table 4-2: Relative amount of time spent outside the basic instructions 70
Table 4-3: Relative number of total instructions executed outside the DSP emulation instructions 70
Table 4-4: G.723.1 Unmodified Workloads Instruction Count 72
Table 4-5: G.729A Unmodified Workloads Instruction Count 72
Table 4-6: Profiling the G.729A functions by using the speech workload 73
Table 4-7: Profiling the G.723.1 functions by using the 6.3kbits/s workload 74
Table 4-8: G729 Encoder Test Vectors 84
Table 4-9: G729 Decoder Test Vectors 85
Table 4-10: G.723.1 Encoder and Decoder Test Vectors 85
Table 5-1: Vector Load/Store Instructions 112
Table 5-2: Vector Move Instructions 113
Table 5-3: Arithmetic Instructions 116
Table 5-4: Vector Shift Instructions 117
Table 5-5: Vector Miscellaneous Instructions 117
Table 5-6: Scalar Load/Store Instructions 118
Table 5-7: Scalar Move Instructions 118
Table 5-8: Scalar Arithmetic Instruction 119
Table 5-9: Scalar Shift Instructions 120
Table 5-10: Scalar miscellaneous instructions 120
Table 5-11: Enhanced op2 Encoding (Format 2) 123
Table 6-1: Compile-time vector processor parameters for its architectural and microarchitectural state that are contained in gxx_config.vhd file 130
Table 6-2: The allowed silicon technologies that are used for synthesis and place and route contained in gxx_config.vhd file 130
Table 6-3: Compile-time vector register file parameters for its architectural and microarchitectural state that are contained in gxx_config.vhd file 140
Table 6-4: Compile-time vector accumulator file parameters for its architectural and microarchitectural state that are contained in gxx_config.vhd file 163
Table 7-1: VLSI Layout physical parameters for VDP with VLMAX 16 183
Table 7-2: VLSI Layout physical parameters for VDP with VLMAX 32 184

Table 7-3: VLSI Layout physical parameters for VCOP with VLMAX 16 186

LIST OF ABBREVIATIONS

Abbreviation	Expansion
ABI	Application Binary Interface
AbS	Analysis by Synthesis
ADL	Architecture Description Language
ADM	Adaptive Delta Modulation
ADPCM	Adaptive Differential Pulse Code Modulation
AHB	Advanced High-speed Bus
ALU	Arithmetic Logic Unit
AMBA	Advanced Microprocessor Bus Architecture
APB	Advanced Peripheral Bus
ASIC	Application Specific Integrated Circuit
ATC	Adaptive Transform Coding
ATM	Asynchronous Transfer Mode
BASOP	Basic Operations
CAS	Cycle Accurate Simulators
CATV	Cable TV
CCITT	International Telephone and Telegraph Consultative Committee
CELP	Code Excited Linear Prediction
CISC	Complex Instruction Set Architecture
CLB	Configurable Logic Block
CMP	Chip Multi-Processing
CNG	Comfort Generation Noise
CPI	Cycles per Instruction
CPU	Central Processing Unit
CS-ACELP	Conjugate-Structure Algebraic Code Excited Linear Prediction
DLP	Data Level Parallelism
DMA	Direct Memory Access

Abbreviation	Expansion
DSVD	Digital Simultaneous Voice and Data
DSL	Digital Subscriber Line
DSM	Distributed Shared Memory
DSP	Digital Signal Processing
EDA	Electronic Design Automation
EPIC	Explicitly Parallel Instruction Computing
ESL	Electronic System Level
FEC	Forward Error Correction
FLI	Foreign Language Interface
FLOPS	FLoating point Operations Per Second
FPGA	Field Programmable Gate Array
FTTH	Fibre to the Home
ILP	Instruction Level Parallelism
IP	Internet Protocol
ISA	Instruction Set Architecture
ISDL	Instruction Set Description Language
ISPS	Instructions Set Processor Specification
ISS	Instruction-accurate Simulator
ITU	International Telecommunication Union
LAN	Local Area Network
LISA	Language for Instruction Set Architecture
LPC	Linear Predictive Coding
LSP	Line Spectral Pair
MAC	Multiply and Accumulate
MBEN	Multi-Band Excited Vocoder
MELP	Multi-pulse Excited Linear Prediction
MIMD	Multiple Instruction Multiple Data
MIPS	Million Instruction Per Second
MISD	Multiple Instruction Single Data
MOS	Mean Opinion Score
MPEG	Moving Picture Experts Group
MP-MLQ	Multi-Pulse Maximum Likelihood Quantization

Abbreviation	Expansion
NUMA	Non Uniform Memory Access
OS	Operating System
OSI	Open Systems Interconnection
PCI	Peripheral Component Interconnect
PISA	Portable Instruction Set Architecture
PCM	Pulse Code Modulation
PSVQ	Predictive Split Vector Quantizer
QoS	Quality of Service
RAS	Registration/Admission/Status channel
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
PCM	Pulse Code Modulation
PSTN	Public Switched Telephone Network
RELP	Residual Excited Linear Prediction
RTL	Register Transfer Level
RTP	Real Time Protocol
RTCP	RTP Control Protocol
SBC	Sub-Band Coding
SDRAM	Synchronous Dynamic RAM
SISD	Single Instruction Single Data
SIMD	Single Instruction Multiple Data
SIP	Session Initiation Protocol
SMT	Simultaneous Multi-Threading
SMP	Symmetric Multi-Processing
SoC	System on Chip
SPARC	Scalable Processor Architecture
SRAM	Static RAM
SREGS	Scalar Registers
SRF	Scalar Register File
TSMC	Taiwan Semiconductor
TCP	Transport Control Protocol
TLP	Thread Level Parallelism

Abbreviation	Expansion
UART	Universal Asynchronous Receiver Transmitter
UDL/I	Unified Design Language for Integrated circuit
UDP	User Datagram Protocol
ULIW	Ultralong Instruction Word
UMA	Uniform Memory Access
VACC	Vector Accumulator
VDEC	Vector Decode Stage
VDP	Vector Datapath Stage
VHDL	Very high speed integrated circuit HDL
VLIW	Very Long Instruction Word
VLMAX	Vector Length MAXimum
VLSU	Vector Load/Store Unit
VoIP	Voice over Packet Internet
VREG	Vector Register access Stage
VREGS	Vector Registers
VRF	Vector Register File
WAN	Wide Area Network
Wi Fi	Wireless Fidelity
XST	Xilinx Synthesis Technology

CHAPTER 1
INTRODUCTION

1.1 Problem Formulation

Ever-advancing technologies have enabled the worldwide convergence of voice and data communications in a single network infrastructure. This is the domain of packet-switched networks such as the Internet Protocol (IP) which lead to significant savings in cost and infrastructure deployment as well as to bandwidth efficiency [1]. Voice over Internet Protocol (VoIP) is such an example which uses IP to send digitised voice/data as a reliable alternative to traditional circuit-switched communication. In VoIP, the voice network is integrated into the Local Area Network (LAN) and is connected to the traditional Public Switched Telephone Network (PSTN) through a gateway. The gateway is a special piece of equipment which handles the translation of signals from the PSTN into IP packets, required for the transmission across the Internet and vice versa [2]. Figure 1-1 depicts the general model of traditional voice and data networks that are separated (a) and a VoIP network that encompasses both in the same infrastructure.

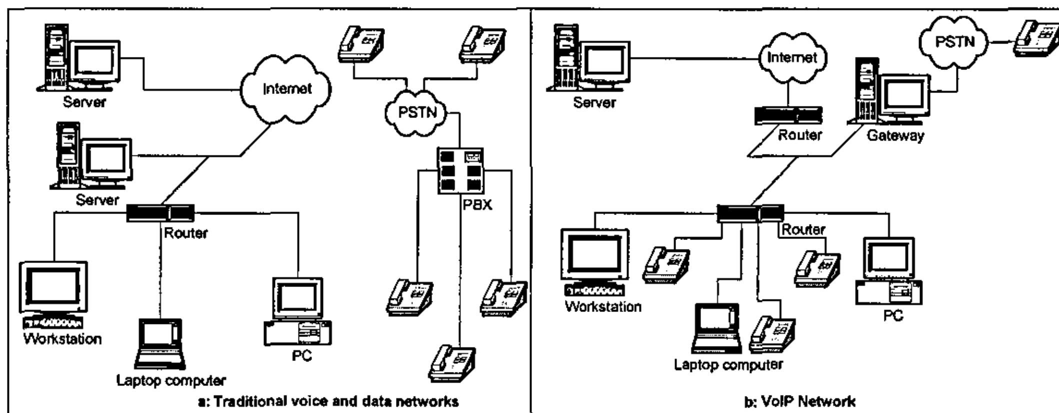


Figure 1-1: Traditional voice and data networks (a) and VoIP network (b)

The transition from circuit-switched to packet-switched networks enables applications that go beyond simple voice transmission, embracing other forms of data and allowing them to all travel over the same infrastructure [2]. Packet-switched networks such as

Internet, Intranets, LANs and WANs encode the message and transmit it in the form of packets that are blocks of data with added header and trailer information. Packet networks don't need a dedicated link between transmitter and receiver hence there is lower cost per communication session as most interconnection charges are avoided. Additionally, the required infrastructure is minimal because all the real-time applications use the existing network. Consolidation of the different networks in one simplifies the equipment, protocols, software and hence enables better service to be provided at low cost and with more efficient use of the resources. In the last few years, there has been a shift in large corporations migrating their communications into a single network infrastructure. The Japanese government decided in 2002 to establish an environment for the widespread use of IP telephony services. This decision initiated the development of key technologies for IP telephony [3]. BT began, since November 2006, to replace its existing telephone network with one based entirely on the Internet Protocol (IP). When this is completed, the telephone system and the internet will share the very same network infrastructure [4].

Since the early days of VoIP it became clear the need for the creation of a common protocol stack in order to enable the development and spreading of the former. In 1996 the H.323 [5] recommendation was issued by the International Telecommunication Union (ITU) and revised in 1998 at which time the framework of an IP network was defined. H.323 was the basis for the first widely used VoIP systems. It specifies a number of protocols for speech coding, call setup, signalling, data transport and other areas [6]. The architecture of the H.323 protocol stack is depicted in Figure 1-2.

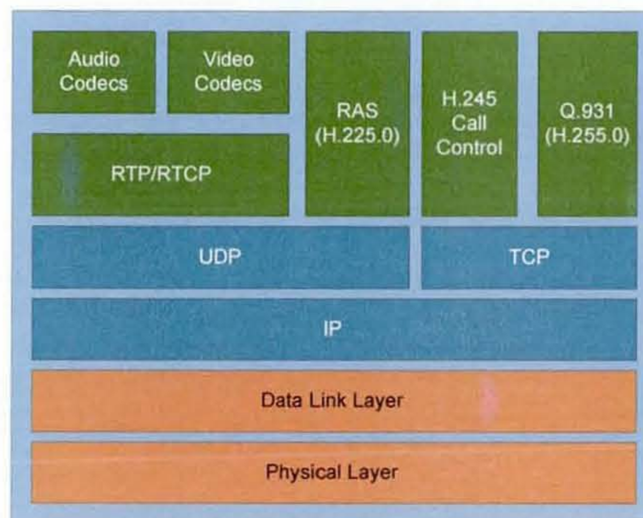


Figure 1-2: The architecture of H.323 protocol stack

The H.323 standard incorporates the following ITU protocols:

- Audio Codecs: G.7xx Series
- Video Codecs: H.26x Series
- RTP: Real Time Transport Protocol
- RTCP: RTP Control Protocol
- RAS (H.225): Registration/Admission/Status channel controlled by the H.225 gatekeeper protocol
- H.245: Call (connection) Control, selects the compression algorithms, bit rate etc
- Q.931 (H.255): Call signalling
- UDP: User Datagram Protocol
- TCP: Transport Control Protocol

H.323 provides a complete protocol stack for real-time multimedia, conferencing (voice and video) and data transfer [2]. It played a key role in the widespread use of VoIP services as H.323 gateways are the interface between the PSTN and packet-switched networks [7]. These gateways employ speech coding algorithms that encode the audio signal prior to transmission and decode it during reception. VoIP specifies a significantly smaller voice bandwidth than a traditional PSTN that operates at a constant 64kbits/sec rate. Speech coding is the process of digitally encoding speech in order to reduce the bit rate of its representation during digital transmission, while maintaining an acceptable speech quality. Speech coding or compression algorithms provide good quality communication over packet based networks and reduce network bandwidth requirements. Hence efficient coding of the human speech is of paramount importance. The H.323 multimedia standard supports a number of common ITU codecs such as G.711 [8], G.726 [9], G.728 [10], G.729A [11] and G.723.1 [12] for interoperability reasons. These codecs have different bit rates, implementation complexity coding delay and voice quality. G.711 is a compulsory recommendation that specifies a simple A/μ-law codec that produces toll quality speech with low computation complexity, typically of 1MIPS, but requires up to 64kbits/s bandwidth. G.729A and G.723.1 are the most popular for bandwidth limited transmission channels. G.729A was designed for simultaneous voice and data applications while G.723.1 was indented for low-bit rate videophones [13]. These speech

coding algorithms are very computationally intensive and consist of a number of sections of code executing in tight loops and processing arrays of data. More details about these codecs and speech coding theory are given in Chapter 2. Table 1-1 shows the characteristics of the aforementioned codecs that are widely employed in VoIP services. With the growing demand for VoIP services, it has become increasingly important to implement efficiently these algorithms. Codec optimization minimizes the processor loading and enables the system to support more voice channels per silicon area, while maintaining low power consumption [7][14].

Table 1-1: ITU Standards for Voice Compression

ITU Specification	Transmission Rate (kbits/s)	Computation Complexity (MIPS)	Mean Opinion Score
G.711	56/64	1	4.1
G.723.1	5.3/6.3	16	3.65/3.9
G.726	32	2	3.85
G.728	16	30	3.61
G.729/G.729A	8	20/11	3.92/3.7

This research presents the design and implementation of a high performance custom vector processor to accelerate these speech coding algorithms that are used typically for voice compression at the gateway of a VoIP network or for multimedia applications. More specifically, a controlling CPU (Leon3) and a closely-coupled, configurable, extensible vector coprocessor was researched and developed as SoC components [15]. A vector processor was selected as it is generally accepted that for multimedia processing, SIMD execution units with wide datapaths are able to achieve significant speedups compared to existing scalar architectures without much of complexity cost [16]. The vector coprocessor is a hybrid SISD (scalar processor)-SIMD (vector processor). The idea of vector coprocessors to be closely coupled to a superscalar CPU has been expressed in the late 80's [17]. This combined scalar/vector architecture can lead to an order of magnitude improvement in workload performance and result in reduced area/power/cost per voice channel compared to the existing solutions.

1.2 VoIP

VoIP supports near-real-time, multidirectional voice exchanges by employing the Internet Protocol as transport technology. VoIP is an exciting technology that has changed the

way that people communicate and its power and versatility make it increasingly pervasive in embedded applications [18]. By merging the two traditional network infrastructures; Data (LAN) and voice (PSTN) the required equipment and expertise for their maintenance is simplified. Figure 1-3 shows possible IP telephony network connections and components of a typical VoIP system.

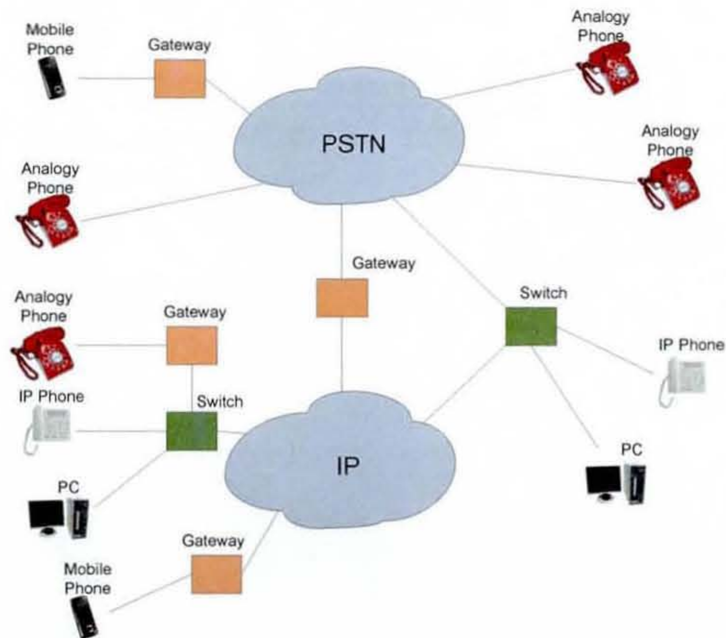


Figure 1-3: Simplified representation of possible IP telephony network connections

1.2.1 Description of the VoIP process

Traditional voice networks such as PSTN employ digital switching technology to establish a dedicated link (circuit) between nodes and terminals for communication [2]. Each such dedicated circuit cannot be used by other callers even if it is not active until it is released and a new call is set up. On the other hand, in packet switched networks, the digital information is encapsulated in packets that are routed between nodes over data links shared with other packet traffic. In each network node, packets are queued or buffered resulting in variable delay whereas in circuit switching there is constant delay and transmission bit rate between the nodes. Packet switching is categorized into datagram (connectionless) such as Ethernet and IP networks and virtual circuit switching (connection orientated) such as Asynchronous Transfer Mode (ATM), X.25 etc [19].

The connection stages between two endpoints in a VoIP system are illustrated in Figure 1-4. These stages incorporate the following functions: signalling process, encoding/decoding, the transport mechanism, and the switching gateway. In the beginning the signalling process takes place and establishes the communication between the handset and the phone network.

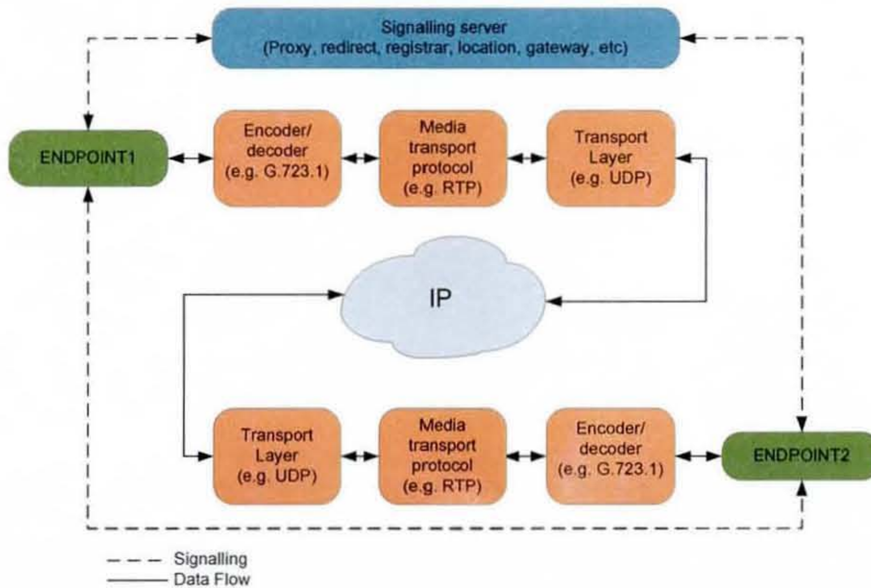


Figure 1-4: VoIP signalling and transport flow between endpoints

The signalling process is responsible for maintaining and terminating the connection between the nodes and hence it is active for the whole duration of the communication. As VoIP transmission is packet-based the data/voice message that is sent during the communication is digitized and separated to frames which are encoded by the chosen speech coder to reduce bandwidth requirements. The resulting bitstream is then packetized and is inserted into the IP network where it follows one or more transport protocols. Afterwards, it goes through a number of switches and eventually reaches the receiving gateway. The switching gateway ensures the packet set's interoperability with a different destination IP-based system or a PSTN system. At the receiving end, the bitstream set is de-packetized, decoded and converted back to an audio signal, after going through the equivalent speech decoder [2].

The communication protocols enable interoperability of the system and are part of H.323 or SIP protocol stacks. SIP or Session Initiation Protocol is an alternative to the large,

complex and inflexible H.323. It was developed specifically for IP telephony and other Internet services but is simpler than H.323 and can adapt more easily to future applications. There are several types of signalling protocols running concurrently at various levels. The various levels of protocol are categorised according to their function in a standardised seven layer model that is called Open Systems Interconnection (OSI) and is depicted in Figure 1-5 [20].

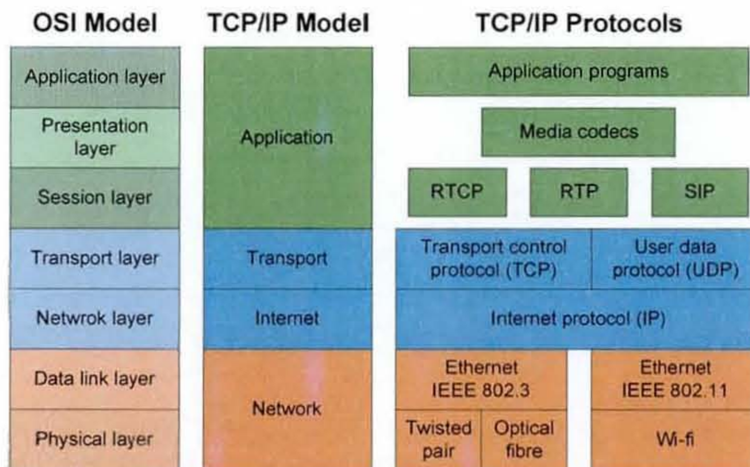


Figure 1-5: Open Systems Interconnection (OSI) and network protocols

The three upper layers (Application, Presentation and Session) support users' applications which are moving through network to be defined in an abstract higher-level way in order to be exchanged between different users. The four lower layers (Physical, Data link, Network and Transport) are used for formatting, encoding and transmission of the data over the network. An IP network operates in the first three layers and the transport layer passes the data from above to the network layer. The transport layer isolates the upper layers from changes of the hardware and controls the movement of the packets, performs error checking etc [6]. Voice and video for real-time communications use UDP (User Datagram Protocol) packet transport instead of TCP (Transmission Control Protocol) as the shortest delivery time is more critical than packet loss. However, the media delivery using UDP is sensitive to packet delay and loss hence QoS (Quality of Service) for multimedia communications is very important [21]. More specifically, the level of intrinsic QoS (latency, jitter, dropped packet rate) for the packet-switched services must be determined in order to assure the adequate perceived QoS [18].

1.2.2 VoIP Applications

Even though VoIP is a technology for transferring voice over IP packets it is not restricted only to that. Broadband IP networks using xDSL (Digital Subscriber Line), FTTH (Fibre-to-the-home), and CATV (Cable TV) lines have increased the available bandwidth and hence the voice quality in VoIP has improved while making additional concurrent visual communication possible [3]. The VoIP infrastructure facilitates an entirely new set of networked real-time applications, such as: videoconferencing, remote video surveillance, analog telephone adapters, Multicasting, Instant messaging, Gaming, Electronic whiteboards etc. Other features added from IP services are automatic rerouting of phone calls on the PSTN to a user's VoIP phone connected to a network node. In this way a global-enabled cellphone network is enabled without roaming charges as the user's location is seen as just another network connection point. IEEE 802.11 enabled VoIP handsets to allow conversation in worldwide WiFi hotspots without compatibility issues [2].

1.2.3 Current state of the art

VoIP implementation depends heavily on the evolution of hardware and software technology. Much effort has focused on developing techniques to meet the QoS requirements and ensuring the performance and reliability of PSTN networks at significantly lower cost. Many protocols and standards have emerged in the last year and made the VoIP feasible. At the same time, many factors need to be balanced to produce a cost effective product with toll quality voice [19].

In a VoIP ASIC, processor selection is very important as this has a direct consequence on the allocation of time critical (speech coding, voice activity detection, echo cancellation) and non-critical (signalling protocols, operating system, user interface) tasks. In addition, it affects significantly the ASIC cost as the core CPU system is typically the most expensive piece of silicon IP. The processor is usually a stand-alone 32-bit RISC engine with a) custom instruction extensions b) large capacity DSP on board processing and c) a loosely-coupled external DSP/coprocessor. The custom instruction extensions or the DSP/coprocessor perform the voice processing operations while the RISC processor

handles only the control functions enabling this way the main processor to support more than one channels.

A popular architecture in VoIP gateways is the dual core processor organization (c) that integrates both RISC and DSP cores within a single package. The software development, debugging and the management of inter-processor communication for this solution is complicated and time consuming. Another popular solution is the RISC/DSP (b) dual execution units but a single instruction set architecture. In this way, there is no need for inter-processor communication and hence smaller overhead and better voice quality [19]. A targeted architecture therefore that can perform efficiently the mathematically intensive operations, has zero-overhead loops, barrel shifters, modulo addressing can improve the system performance dramatically. Dedicated on-chip DSP/coprocessor memories keep the algorithm coefficients and voice sample data on-board, maintaining the processing throughput. Additionally, an integrated solution simplifies the overall complexity and reduces time to market [22].

1.3 Programmable Architectures

1.3.1 General Purpose Processors

In the past, general-purpose processor design was driven mostly for non-real-time, stand-alone applications which were largely nonnumeric with little inherent parallelism. The proliferation of multimedia-rich applications that involve significant real-time processing of continuous media data streams has forced profound changes in computer architecture. Since there are no limitations in the semiconductor technology, general-purpose processors can significantly accelerate media-intensive processing with relatively simple architectural support and the addition of instruction set extensions [16]. Over the last years the major vendors of general-purpose processors have announced the addition of instruction extensions in their ISAs to increase the performance of the multimedia applications. These instruction extensions are based on a subword execution model. This model uses the whole width of the processor datapath by processing smaller data types, typically found in signal processing (8- or 16-bits) in parallel by executing common multimedia operations [23]. Examples of general-purpose processors with added multimedia extensions are Intel's x86 with MMX [24] and SSE [25] extensions, Sun's

UltraSparc enhanced with VIS [26], PowerPC with Altivec [27], Silicon Graphics' MIPS V with MDMX [28], Compaq's Alpha with MVI, and Hewlett-Packard's PARISC with MAX2 [29] extensions.

1.3.2 DSP Processors

The software implementation of speech codecs on DSP processors is a popular choice as these processors are more "tuned" to signal processing algorithms better than general-purpose processors. This is due to the advances in DSP architecture that effectively execute the repetitive computations on data streams present in these algorithms through multiple functional units that operate in parallel and SIMD operations. These techniques are performed using mechanisms with lower complexity than general-purpose processors and speed up significantly the execution of these applications while keeping power consumption low [30]. Several projects [31] [32] [33] employ the Texas Instruments DSPs to implement G.723.1 in real time after applying some iterative refinement and optimization on the reference C code. Motorola implements the G.729A on the StarCore SC140 [34] after optimizing the C reference code of the algorithm and Samsung with its SSP1820 DSP implements the G.723.1 [35]. Another optimized solution is integrating conventional general-purpose RISC processors and DSP cores with dedicated functionality into a single, unified architecture such as the Hitachi SHx-DSP [36] and the Infineon TriCore [37].

1.3.3 ASIC (Embedded) processors

In general, application specific hardware design is the most popular candidate to meet cost, performance and power demands for VoIP applications. ASIC implementations can be divided in to the following three categories:

1.3.3.1 Configurable processors

In high-speed communication system design the simplest and most common architecture use embedded 32-bit processors such as (ARM, MIPS, PowerPC etc) or DSPs in either discrete or integrated form. Though this provides a lot of flexibility and general applicability, the processing of some software-based algorithms limits the system performance to a great extent. In addition, DSPs may not be as attractive in

computationally expensive operations such as error correction algorithms or filters where hardware implementations tend to be more efficient. On the other hand, ASICs achieve very high performance but require significant design cost and effort and offer no flexibility [38, 39]. An alternative architecture that promises high performance, extendibility, flexibility, code size and power dissipation reduction and also lower cost is the configurable processor. Configurable processors can be modified and their ISAs extended to target a specific application domain by changing the processor's feature set in order to accelerate the critical parts of the algorithm. A processor can be configured in three general ways:

- By altering the processor's predefined architectural framework such as cache size, number of registers, multipliers or barrel shifters etc.
- By adding custom, high-performance interfaces and streaming memories
- By adding custom instruction extensions to optimally map to the target application

Configurable processors are typically delivered as synthesizable RTL ready to be synthesized and integrated into an FPGA or SoC design. They usually come with vendor tools, EDA synthesis scripts and verification environments to verify the correct operation on a target system [40]. Examples of configurable processors employed for audio processing apart from the vector processor of the current work are the Tensilica'a Xtensa [41] 32-bit microprocessor that has the ability to run any C or C++ programs and add execution units for the implementation of the instruction extensions to speed the target application. A pioneer in this field is ARC International with its ARC 700 family [42] architecture with 128-bit SIMD configuration. Other vendors include Silicon Hive [43] with UltraLong Instruction Word (ULIW) architecture and so on.

1.3.3.2 Reconfigurable Processors

Reconfigurable processors adapt dynamically their microarchitecture to address the application requirements. This type of processor utilizes microcode and custom configured hardware to improve performance. The microcode is utilized to perform both the reconfiguration process and the execution of the code and its frequently used parts are located permanently in a fixed part of on-chip storage [44]. In the past reconfigurable

architectures referred exclusively to the gate level (fine-grain) with every computation being built up from the Boolean gates. An example of such a device which functions at this level is the FPGA device. An architecture can also be reconfigured on microarchitecture or architecture level. These levels of computational hierarchy are implemented by coarser basic computational units that are incorporated in FPGA devices. The FPGA can contain hard (e.g. multipliers) or soft (e.g. components of a standard library) macros to customize its functionality. The hard macro is a fixed ASIC core embedded into the fabric of the FPGA while the soft macro is a sequence of computations implemented as fixed entities on the FPGA fabric [45]. Examples of reconfigurable architectures used for multimedia applications at the microarchitecture level are the PipeRench [46] and RaPiD [47] processors whereas examples at the architecture level are the RAW project [48] and Pleiades of Berkeley University [49]. Reconfigurable architectures offer flexibility, functional efficiency of hardware and software programmability, logic capacity of programmable devices and advanced automated design techniques.

1.3.3.3 Fixed Processors

This category incorporates fixed architecture processors typically integrated in an ASIC infrastructure (buses, local memories, coprocessors). In order to achieve high performance modifications are usually performed on either the C code or the assembly of the application in order to take full advantage of the processor architecture. Examples of fixed ASIC processor that realise speech codecs are the ARM9 which implements the G.723.1A/G.729AB codecs [50] or the G.729E codec [51] by using optimized ARM assembly code. Another example is a low power DSP core that implements G.723.1 codec within the H.324 standard [52].

1.4 Hardwired Architectures

There are very few instances of research projects focused on the acceleration of the G.723.1 and G.729 standards using configurable, extensible, vector architectures as proposed in this work. A suggested architecture for the hardware implementation of parts of both codecs was proposed by Olausson and Liu [14]. Their paper briefly discusses three hardware structures to accelerate conditional moves and branches before or after the

calculation of the 32-bit absolute value (L_abs) of the 6.3kbits/s G.723.1. Another more focused approach was the hardware/software co-design of the G.723.1 by Mishra et al [7]. In that work parts of the codec (pitch estimator, formant perceptual weighting filter and harmonic noise shaper) were implemented in hardware using a single MAC unit that operates in parallel to a DSP processor which executes the rest of the algorithm. Additionally the normalisation operation is implemented in hardware.

The hardware implementation of the speech codecs is not a common practice as the C reference codes have to be ported to VHDL and this is a quite tedious and time consuming task. Another problem is that the arithmetic logic and especially the multipliers are very complex and their implementation in an FPGA will require many CLBs (160 CLBs on a XILINX Virtex FPGA per multiplier approximately). Since the codecs are typical DSP codes, their execution on DSP processors generally leads to much better performance. On the other hand, ASICs seem a better solution for multi-channel codec implementation but the integration of several DSP cores on an ASIC to offer multiple-channel capabilities is a more effective and appealing solution [53].

1.5 Research contribution and overview

The main objective of this work was to research and develop a configurable, extensible vector embedded CPU architecture for accelerating speech coding algorithms employed in VoIP networks. This research was funded by the Engineering and Physical Sciences Research Council (EPSRC) under grant GR/S44976/01. The contributions of this project are outlined in this section.

At the beginning of this research and in order to investigate the potential acceleration, both C reference codes were profiled to identify the computation workload distribution. This is described in section 4.3.1 of this thesis. The results showed that the most CPU-intensive parts of the code were in the DSP emulation functions of the reference implementation. Further studying of the code revealed that a significant number of the basic operations appear in data-parallel loops. It was apparent that the creation of vector instructions that closely match these basic operations could lead to high performance. This is a major contribution of this work.

The next task was to define the custom vector instructions and the data-level-parallel architecture of the vector coprocessor. Parallel exploitation is essential for the efficient execution of DSP codes. However, the reference implementations have to be fully vectorized in order to benefit from data-parallel processing which is the primary capability of the proposed vector architecture. The custom vector instructions were represented by C macros and were introduced into the C reference codes to implement the data-level-parallel inner loops. As speech coding algorithms consist of small loops or kernels that dominate overall processing time it was important to perform manual vector assembly coding and hand optimization of such tight loops [16]. In order to check the correct operation of the vectorized speech codecs after the vectorization of every loop the codes were verified against the ITU test vectors by comparing the output bitstream of the optimized code with the original one. The vectorization methodology is described in sections 4.3.2 -4.3.4 and the full vectorization of both the G.723.1 and G.729A speech coders and decoders is another major contribution.

The remainder of the code that consists of the non-vectorizable loops and other parts of the code which contain basic DSP operations was optimized through the addition of custom scalar instructions. Again algorithmic equivalence between the optimized and the original (reference) code was established. The scalar optimization and the verification are presented in sections 4.3.5 and 4.3.6 respectively. In addition, both vector and scalar instructions are described in Chapter 5 and are listed in more detail in Appendix A. The joint scalar optimization and vectorization of the reference ITU-T codes is a third contribution of this work.

The next step was to evaluate the performance of the vector architecture before it is implemented in hardware. For this purpose, the SimpleScalar toolset was used to evaluate the coprocessor architecture under study. The simulator was modified and extended to include the added state (coprocessor scalar and vector state) and the scalar and vector extensions. The extended instructions that were represented in C macros were replaced with inline assembly and executed on the simulator. The modifications of the SimpleScalar simulator are described in sections 4.3.7 and 4.3.8. Simulations were run for all ITU-T input vectors and for vector lengths of up to 128 16-bit elements. Results, in the form of relative dynamic instruction count, were taken for the vector only and for full optimization (scalar and vector) of both speech coding algorithms. These results show the

performance metric improvement which the instruction-accurate model of the vector coprocessor achieves. The results are presented on section 4.4 and Appendix C. Methodologies for the introduction of scalar and vector state and addition of instructions in the SimpleScalar infrastructure are another contribution of this work.

Another task of this project was the modelling in SystemC of the vector instruction set extensions and its subsequent synthesis to low-level RTL in order to be introduced to the multi-parallel, configurable SS_SPARC processor. This work was undertaken to study faster routes to silicon of the SIMD extensions, compared to the established RTL flow and is presented in paper [54] and is discussed in section 7.8. The SystemC model is the behavioural description of the same vector instructions that were introduced in the speech codecs. The "packing" of the SIMD ISA was verified by using the ITU test vectors to validate their functionality. The obtained results from the statistical power analysis results for both the SystemC-accelerator and the RTL-accelerator synthesis are presented in section 7.8.3. This is a major contribution of this work as it compares the benefits of synthesizing a configurable, extensible SIMD datapath with that of a highly optimized RTL-based implementation.

The main author's contribution to the research project was the full design and implementation of the proposed vector datapath of the vector processor. The vector datapath was verified by using an FLI-based testbench and this process is described in section 7.1. The vector processor was attached to the fifth stage (memory stage) of the main Leon3 scalar processor. Modifications were made to the pipeline of the scalar core and extra decode logic was added to accomodate the vector unit. The microarchitecture of the vector datapath and its interfacing to the Leon3 is explained in Chapter 6. Finally, statistical power analysis was performed for the vector datapath and the vector coprocessor as a whole for different configurations (VLMAX, frequency) in order to explore their effects on area/power/frequency results. These results along with the layouts of the vector datapath and vector processor are presented in sections 7.4 to 7.7. This is the final and major contribution of this work.

1.6 Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2 a background section in speech coding is given describing the general models of speech representation, coding schemes and types of speech coders that exist. In addition, the characteristics and principles of the two ITU standards that are used in this project namely, the G.729A and G.723.1 standards are presented. Chapter 3 gives an overview of parallelism including the limitations imposed from dependences and description of their types. Additionally the three different types of parallelism are introduced along with the appropriate processor architectures for their efficient exploitation. Emphasis is given to DLP which is the primary form of parallelism addressed in this project. This form of parallelism is most effectively exploited with vector architectures. Chapter 4 discusses the optimization methodology and the performance improvement achieved with the introduction of custom scalar and vector ISA extensions in both speech coding standards. Following that it presents the modifications made to the SimpleScalar instruction-set simulator to incorporate a large number of scalar and vector instruction extensions. Finally this chapter presents the performance benefits achieved via the introduction of the aforementioned instructions for different vector lengths and workloads. In Chapter 5 the vector coprocessor architectural state and programmer's model are presented followed by the introduction of the Leon processor and the overall system architecture. Chapter 6 gives a detailed description of the pipeline organization and its constituent components. This is followed by a brief description of the VLSU which is part of another research work. The modifications to the Leon3 pipeline are then presented to enable the tight-coupling of the vector coprocessor. Chapter 7 deals with the verification, synthesis and back-end flow of the vector datapath and vector processor as a whole. This is followed by the SystemC modelling and the parametric ESL implementation of the vector datapath. The latter was then inserted in the exposed vector engine of the SS_SPARC processor. The Chapter 7 also includes a detailed description of the SS_SPARC ASIC processor. Finally this chapter presents the statistical power analysis results for both the SystemC and RTL-designed vector datapaths. The Conclusions chapter discusses suggestions for further research, potential applications and additions to this work. Appendix A includes the details of the vector processor instruction set. Each instruction is presented individually with its format, a short description of the instruction's operation and a

software example. Appendix B includes the internal control and data signals and their combinations as used in the vector pipeline. Finally, the performance improvement results at function level of both speech codecs obtained from the first year's work are presented in Appendix C.

1.7 References

[1] Todd Wynia, " Laying the foundation for VoIP: A perspective on platforms, protocols and technologies," in *Embedded Computing Design*, Spring 2001.

[2] Jim Doherty and Neil Anderson, *Internet Phone Services Simplified (VoIP)*: Cisco Press, 2006.

[3] M. Mineo, A. Niimura, H. Ooboshi, et al., "IP Telephony Terminal Solutions for Broadband Networks," *Hitachi Review,* vol. 51, June 2002.

[4] Steven Cherry, "Nothing but Net," in *IEEE Spectrum*. vol. 44, January 2007, pp. 18-21.

[5] ITU-T Recommendation H.323, "Packet-based Multimedia communication systems," 1998.

[6] Andrew S. Tanenbaum, *Computer Networks*, 4th ed.: Pearson Education International, pp. 685-691, 2003.

[7] S. M. Mishra and A. Balaram, "Efficient hardware-software co-design for the G.723.1 algorithmtargeted at VoIP applications," in *IEEE International Conference on Multimedia and Expo*, 2000, pp. 1379-1382.

[8] ITU-T Recommendation G.711, "General Aspects of Digital Transmission Systems," 1989.

[9] ITU-T Recommendation G.726, "40, 32, 24, 16 kbit/s Adaptive Differential Pulse Code Modulation (ADPCM)."

[10] ITU-T Recommendation G.728, "Coding of Speech at 16 kbit/s using Low-Delay Code Excited Linear Prediction."

[11] ITU-T Recommendation G.729A, "Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear-prediction (CS-ACELP)," 3/96.

[12] ITU-T Recommendation G.723.1, "Dual Rate Speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s," 3/96.

[13] R. V. Cox and P. Kroon, "Low bit-rate speech coders for multimedia communication," in *IEEE Communications Magazine*. vol. 34, December 1996, pp. 34-41.

[14] M. Olausson and D. Liu, "Instruction and hardware accelerations in G.723.1(6.3/5.3) and G.729," in *the 1st IEEE International Symposium on Signal Processing and Information Technology*, 2001, pp. 34-39.

[15] V. A. Chouliaras, "Vector Coprocessor for Speech Coding: Case of Support," Engineering and Physical Sciences Research Council (EPSRC) – GR/S44976/01, Loughborough University 2002.

[16] K. Diefendorff and P. Dubey, "How Multimedia Workloads Will Change Processor Design," in *IEEE Computer*. vol. 30, September 1997, pp. 43-45.

[17] Francisca Quintana, Roger Espasa, and Mateo Valero, "A Case for Merging the ILP and DLP Paradigms," in *6th Euromicro Workshop on Parallel and Distributed Processing*, Madrid, Spain, 1998, pp. 217-224.

[18] William C. Hardy, *VoIP Service Quality: Measuring and Evaluating Packet-Switched Voice*: McGraw-Hill Networking, 2003.

[19] J. Dionne and B. Davis, "Embedded VoIP implementations using SIP," in *EE Times Asia*, 16 September 2004, www.eetasia.com/ART_8800346844_499491_TA-e55e1221.HTM.

[20] Andy Bateman, *Digital Communications: Design for the real world*: Addison-Wesley, 1999.

[21] Henry Sinnreich and Alan B. Johston, *Internet Communications Using SIP: Delivering VoIP and Multimedia Services with Session Initiation Protocol*, Second ed.: Wiley, 2006.

[22] A. M. Kondoz, "Digital Speech: Coding for Low Bit Rate Communications Systems," John Wiley & sons, 1994, pp. 117-123.

[23] T. M. Conte, P. K. Dubey, M. D. Jennings, et al., "Challenges to Combining General-Purpose and Multimedia Processors," in *IEEE Computer*. vol. 30, December 1997, pp. 33-37.

[24] A. Peleg and U. Weiser, "MMX Technology Extension to the Intel Architecture," in *IEEE Micro*. vol. 16, August 1996, pp. 42-50.

[25] K. Diefendorff, "Pentium III = Pentium II + SSE: Internet SSE Architecture Boosts Multimedia Performance," in *Microprocessor Report*. vol. 13, March 1999.

[26] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, et al., "VIS Speeds New Media Processing," in *IEEE Micro*. vol. 16, August 1996, pp. 10-20.

[27] K. Diefendorff, P. K. Dubey, R. Hochsprung, et al., "AltiVec Extension to PowerPC Accelerates Media Processing," in *IEEE Micro*. vol. 20, March 2000, pp. 85-95.

[28] "MIPS Digital Media Extension," *Instruction Set Architecture Specification, http://www.mips/MDMXspec.pc,* October 1997.

[29] R. B. Lee, "Subword Parallelism with MAX-2," in *IEEE Micro*. vol. 16, August 1996, pp. 51-59.

[30] J. H. Moreno, V. Zyuban, U. Shvadron, et al., "An innovative low-power high-performance programmable signal processor for digital communications," *IBM Journal of Research and Development,* vol. 47, pp. 299-326, 2003.

[31] A.Z.R. Langi, "Rapid development of a real-time speech coder on a TMS320C54x DSP," in *Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering*, 2002, pp. 1045-1048.

[32] Y. Choi, C. Ahn, and T. Kang, "Implementation of a Multi-channel G.723.1 Annex A using DSP," in *International Conference on Consumer Electronics (ICCE)*, 2002, pp. 320-321.

[33] Y. Huang, Y. Juan, S. Zhang, et al., "Implementation of ITU-T G.723.1 Dual Rate Speech Codec based on TMS320C601 DSP," in *the Proceedings of the 5th International Conference on Signal Processing (ICSP)*, Beijing, China, August 2005.

[34] R. Ungureanu, B. Costinescu, and C. Ilas, "ITU-T G.729A Implementation on StarCore SC140," Application Note, Motorola 2001.

[35] S. Lee, S. Park, and Y. Jang, "Cost-effective implementation of ITU-T G.723.1 on a DSP chip," in *Proceedings of 1997 IEEE International Symposium on Consumer Electronics*, December 1997, pp. 31-34.

[36] M. Schlett, "The RISC challenge in signal processing," in *Proceedings of the 3d of the IEEE International Conference onElectronics, Circuits, and Systems*, October 1996, pp. 550-553.

[37] H. Shi, "RISC+SIMD=DSP," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, June 2000, pp. 3211-3214.

[38] A. Wang, E. Killian, D. Maydan, et al., "Hardware/software instruction set configurability for system-on-chip processors," in *Proceedings of the 38th IEEE conference on Design automation*, Las Vegas, United States, 2001, pp. 184-188.

[39] S. Leibson and J. Kim, "Configurable processors: a new era in chip design," in *IEEE Computer*. vol. 38, July 2005, pp. 51-59.

[40] David Fritz, "Configurable Processors: Ready for Prime Time," in *RTC, http://www.rtcmagazine.com/home/article.php?id=100066*, January 2004.

[41] R. E Gonzalez, "Xtensa: A configurable and extensible processor," in *IEEE MIicro*, March/April 2000, pp. 60-70.

[42] "ARC Cores Ltd, www.arc.com/subsystems."

[43] Tom R. Halfhill, "Silicon Hive breaks out," in *Microprocessor Report*, December 2003.

[44] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, "The MOLEN processor prototype," in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004, pp. 296-299.

[45] R. Kastner, A. Kaplan, S. Ogrenci Memik, et al., "Instruction Generation for Hybrid Reconfigurable Systems," *ACM Transactions on Design Automation of Electronics Systems*, vol. 7, pp. 605-627, October 2002.

[46] Y. Chou, P. Pillai, H. Schmit, et al., "PipeRench Implementation of the Instruction Path Coprocessor," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, Monterey, California, 2000, pp. 147-158.

[47] C. Ebeling, D. C. Cronquist, and P. Franklin, "RaPiD-reconfigurable pipelined datapath," in *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, 1996, pp. 126-135.

[48] M. B. Taylor, J. Kim, J. Miller, et al., "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," in *IEEE Micro*. vol. 22, March 2002, pp. 25-35.

[49] M. Wan, H. Zhang, V. George, et al., "Design Methodology of a Low-Energy Reconfigurable Single-Chip DSP System," *Journal of VLSI Signal Processing Systems,* vol. 28, pp. 47-61, May 2001.

[50] Y. Choi and G. Lee, "Real-time implementation of G.723.1A/G.729AB on a RISC processor for personal IP telephony devices," in *Proceedings of the 9th International Symposium on Consumer Electronics(ISCE)*, South Korea, 2005, pp. 20-24.

[51] A. Tripathi, S. Verma, and D. D. Gajski, "G.729E Algorithm Optimization for ARM926EJ-S Processor," University of California, Irvine 2003.

[52] H. Okuhata, M. H. Miki, T. Onoye, et al., "A low-power DSP core architecture for low bitrate speech codec," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, Seattle, USA, May 1998, pp. 3121-3124.

[53] C. Plessl and S. Maurer, "Hardware/Software Codesign in speech compression applications," in *Institut fur Technische Informatik und Kommunikationsnetze* Zurich: Eidgenossische Technische Hochschule, February 2000.

[54] V. A. Chouliaras, K. Koutsomyti, T. Jacobs, et al., "SystemC-defined SIMD instructions for high performance SoC architectures," in *13th IEEE International Conference on Electronics, Circuits and Systems*, Nice, France, December 2006, pp. 822-825.

CHAPTER 2
SPEECH CODING THEORY

2.1 Introduction

As already identified there is a major trend toward integrating voice-related applications in the context of multimedia applications such as VoIP networks, simultaneous voice and data (DSVD) applications, speech recognition, videoconferencing and so on [1]. This is consistent with the growing demand for wireless and satellite communications which require enhanced privacy and high bandwidth. To meet these needs the speech signal is transformed to digital format in order to be processed, stored and transmitted efficiently under software control. Digital speech exhibits flexibility, ability for encryption/decryption and error correction, however requires high transmission bandwidth and storage capacity. To reduce these requirements, speech coding or speech compression has emerged on the research field concerned with efficient digital representations of voice signals for high-quality speech at low data rates [2]. Even though the sampling rate cannot be lower than twice the bandwidth of analog speech, the past decades several methods have been proposed to represent the sampled waveform with a minimum number of bits while preserving its perceptual quality. These methods have been adopted in a number of speech coders standards that are based on an optimum tradeoff between efficient low-bit transmission, perceptual quality for the available bandwidth and a combination of other objectives according to the requirements of every application [2] [1]. In the next sections, a brief description of the speech coding objectives and requirements will be given along with the speech production system. In addition, the main coding strategies will be introduced and the two ITU standards used in this research will be presented.

2.2 Speech Coding Objectives and Requirements

There are several objectives and requirements that a speech coder must meet for specific target applications. These requirements define the basic bitrate, speech perceptual quality,

algorithmic complexity, cost and system delay of the selected speech codec. Therefore, these influencing factors require careful consideration in order to converge towards an optimum compromise between these often conflicting objectives. Speech quality and bit rate are two factors that directly conflict with each other. The lower the bit rate of the speech coder the higher the signal compression and the more the speech quality degradation. Public Switched Telephone Network (PSTN) and associated systems such as CCITT require high quality of encoding usually referred to as 'toll quality'. For private commercial networks and military systems, the quality factor may be reduced to lower processing and bandwidth requirements. Although absolute quality is often specified, sometimes it is compromised for a lower standard if other factors are allocated a higher overall rating. In general, in a mobile radio system it is the overall average quality that is the deciding factor and takes into account both good and bad transmission conditions. Other important factors for the choice of a speech coding algorithm is the coding delay, the immunity to error, the algorithm complexity and the implementation cost [3].

Coding delay includes algorithmic (the buffering of speech for analysis), computational (time taken to process the stored speech samples) and transmission contributions. Only the first two concern the speech coding subsystem though sometimes the transmission can be initiated before the algorithm has completed processing all the information in the analysis frame. In this case, the encoder starts transmission of the spectral parameters as soon as they become available. Low delay is essential if the major issue of echo is to be minimised. For mobile system applications and satellite communication systems echo cancellation is already included as substantial propagation delays exist. In PSTN, where the delay is very small, extra echo cancellers will be required if coders with long delays are introduced [3]. The other problem with the delay is the subjective annoyance factor. Therefore, all the standardised speech coders have specific requirements for the delay. As it is known [4], the speech coding bandwidth occupies only a small fraction of the total channel capacity, the rest is used for Forward Error Correction (FEC) and signalling. For mobile connections which suffer from both random and burst errors, a coding scheme's built-in tolerance to channel errors is essential for acceptable communication quality. By utilising built-in robustness, less FEC can be used resulting in higher source coding capacity. This trade-off between quality and robustness is a difficult task and it is considered from the beginning of the speech coding algorithm design. In order to achieve

a good average overall performance more sophisticated algorithms are created with increased computational complexity. Therefore, the real-time implementation of such algorithms under the additional constraints of size and power consumption is a major issue and several techniques are employed to minimize multiple conflicting objectives [4]. Before we describe the basic model of a vocoder and the various methods that exist, we need to describe the principles of the speech production model.

2.3 Speech production system

The diagram of the main organs of the human anatomy involved in the speech production mechanism is shown in Figure 2-1. The compressed air forced from the lungs to the vocal apparatus pushes apart the vocal cords and creates an opening known as the glottis. When the air passes through the glottis the pressure decreases and the opening closes. The repetition of this process causes the vibration of the vocal cords and a high-energy quasi-periodic speech waveform is produced and sent into the mouth and nose cavities. The excitation of the vocal cords is filtered through the vocal apparatus which operates like a spectral shaping filter with a transfer function that represents the spectral shaping action of the glottis, vocal tract (pharynx and mouth cavity), lip radiation characteristics and so forth [4] [5].

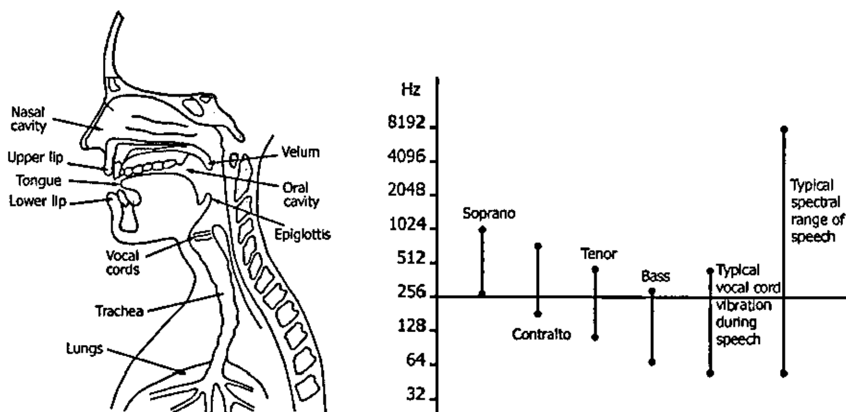


Figure 2-1: Diagram of the human organs involved in speech production and the Spectral Range of Speech

The excitation of the vocal apparatus with glottal vibrations generates voiced sounds and the vibration fundamental frequency is known as pitch frequency. The unvoiced sounds, such as whisper or aspirate, are lower-energy signals as the vocal cords do not participate

and the excitation behaves like noise generator. These sounds are produced by the deliberately constricted air flow through the mouth. Constrictions can be produced by the tongue, the position of the velum, the coupling of the vocal tract with the nasal cavity, the teeth and the lips [4] [5].

Speech can be classified as voiced (e.g. /a/, /k/, etc), unvoiced (e.g. /sh/, /h/ etc) or mixed. As mentioned above, voiced speech is quasi-periodic in the time-domain while unvoiced speech is random-like. The pitch period that is identified by the positions of the largest peaks of the quasi-periodic segments of the voiced signals, consists of approximately 80 samples [2]. Pitch frequency that is used alternatively with the term pitch period, typically ranges for male speakers between 40-120Hz whereas for female speakers is much higher and ranges between 300-400Hz [3]. In the frequency-domain the voiced speech is harmonically structured and its spectrum is characterized by its fine and formant structure. The fine harmonic structure, also known as long-term correlation, is attributed to the vibrating vocal cords. The formant structure or spectral envelope or short-term correlation is attributed to the interaction of the excitation and the vocal tract and is characterized by a set of widened but distinctive spectral needles (peaks) that are multiple of the pitch period and are called formants. Typically for an average vocal tract, three to five spectral envelope peaks can be observed which appear usually around 500Hz, 1500Hz and 2700Hz and represent the resonances of the vocal tract. The amplitudes and locations of the first three formants are vital for the speech synthesis and perception [2]. In contrast, the unvoiced speech does not have a formant structure and exhibits a more high-pass nature with peak around 2500Hz. In addition the energy of unvoiced speech is generally lower than that of voiced speech [4].

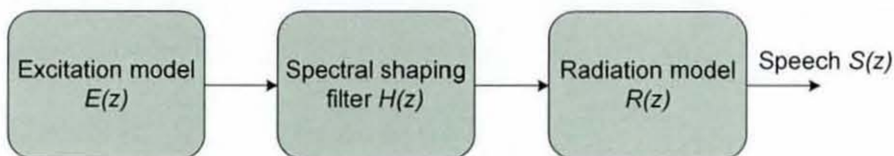


Figure 2-2: General speech production model

The speech reproduction is based on the extraction of the key information of the speech signals [5]. The model of the speech production process is based on digital techniques and a simplified block diagram is shown in Figure 2-2. In this model, the input is the excitation signal which is generally approximated by an impulse sequence for voiced

speech or random noise for unvoiced speech. The excitation signal denoted by *E(z)* is filtered through a time-varying linear digital filter that represents the combined spectral contributions of the glottis, vocal tract and lip radiation characteristics. The filter has a transfer function *H(z)* that can be approximated by an all-pole model and whose coefficients directly depend on the time varying geometry of the vocal apparatus. This speech production model can produce high quality synthetic speech if the underlying model parameters, speech power spectral envelope and the excitation model are appropriately chosen [3] [5]. Even though the process of speech production is known, the perception of the speech by the human auditory system remains a puzzle. It is still unexplained how the recognition between voiced and unvoiced sounds takes place, the ability to locate the position of a sound source (binaural hearing) or to separate a specific voice from a noisy background (cocktail party effect) [2]. Hence there is ongoing research in all these areas.

2.4 Coding strategies

Speech coding schemes can be broadly divided into three main categories: Waveform coders, Hybrid coders and Vocoders. The general operations that these coding schemes perform are to analyse the signal, remove the redundancy and efficiently code its non-redundant parts in order to preserve its perceptual quality. These coding schemes are classified based on their encoding methodology and each has optimal operation within a certain bitrate region [3].

2.4.1 Waveform Coders

Waveform coders are signal independent as they don't exploit any specific properties of speech. They are designed to work with any input signal that is appropriately limited in amplitude and bandwidth. This has the advantage that waveform coders can also encode other types of information such as signalling tones, voice-band data, or even music. By preserving this generality, their coding efficiency is quite modest and limited to rates above 16kbit/s [4]. However, they are still popular due to their simplicity and ease of implementation. Waveform coders are further divided into time-domain and frequency-domain. The most well known representative for the time-domain is the first speech encoding standard 64kbit/s Pulse Code Modulation (PCM), the 32kbit/s Adaptive

Differential PCM (ADPCM) that has been standardised by the ITU Recommendation in G.721 and the Adaptive Delta Modulation (ADM) [3]. Time-domain coders utilise the redundancy in the speech waveform by exploiting the correlations between adjacent samples and encode only the difference between them. In addition, they use predictors at the receiver end to reduce the variance of the encoded signal and consequently the number of bits needed to represent it. Frequency-domain waveform coders exploit the redundancy of the signal in the transform domain. The signal is split into a number of sub-bands and each sub-band is encoded by using a different number of bits. The various methods differ in the way they represent the short-time power spectrum of speech and also in the perceptual properties of the human ear. The most well known frequency-domain coders are the Sub-Band Coding (SBC) and the Adaptive Transform Coding (ATC) [4].

2.4.2 Voice Coders (Vocoders)

Vocoders lie at the opposite end of waveform coders. They deal with speech-specific signals and in particular the physical principles behind speech and as such they do not attempt to reproduce the input waveform [2]. Hence, the performance of vocoders degrades significantly for nonspeech signals. The design and implementation of a vocoder is based on the speech production model that described in section 2.3. This model represents the human speech production mechanism and specifies the basic parameters needed to be extracted from the input speech signal in order to reproduce it as faithfully as possible [4]. Vocoders traditionally operate at rates below 4.8kbits/s which is their main advantage however the produced speech sounds often crude and synthetic.

The preservation of the speech power spectral envelope and the preservation of the voicing information are the two factors that vocoder engineers use when designing speech codecs. These can then be used to re-synthesise speech sounds [3]. A vocoder consists of two parts: analysis and synthesis. The analysis takes place in the encoder where the parameters that describe the vocal excitation and the vocal transmission are extracted from the speech signal. At the decoder the received information is utilised to synthesize the signal that sounds like the original speech. The concepts that are associated with the vocoders were introduced as early as 1939. These concepts incorporate the two-state excitation (pulse/noise), voicing and pitch detection, and filter-bank representation. The

simple excitation model is related to very low bit-rates but at the same time it is responsible for the synthetic quality of speech that is one of the main disadvantages of vocoders. In addition, the estimation parameters that describe the spectral envelope need reliable envelope estimators. Estimators based on linear prediction and homomorphic signal processing were developed around 1960 and this challenging area provoked further research and spawned the development of several methods with improved quality and increased complexity. Channel vocoder is one of the first vocoding systems. It uses a bank of band-pass filters (typically 16 channels) to represent the speech spectrum. The two-stage excitation is utilised and if it is voiced the fine structure is represented using pitch-periodic pulse-like waves while if it is unvoiced it is reproduced using noise-like excitation. Even though the resulting speech is intelligible, the quality is quite synthetic. The Formant vocoders use a similar method to the channel vocoders but the representation of the spectrum needs only the frequencies and the spectral amplitudes of the formants. As a result, they achieve further band savings. Another category is the Homomorphic vocoders that are based on the idea that convolution of the vocal tract impulse response and the vocal excitation can represent the speech log-magnitude spectrum. The output speech has good quality and by applying predictive encoding the transmission rate can be reduced to 4kbit/s. In general, frequency-domain vocoders are more robust to channel errors and background noise but with low, synthetic speech quality. Time-domain vocoders however such as the Linear-Predictive vocoders produce highly intelligible speech making them one of the most popular techniques for speech coding but they are very sensitive to channel errors and noise [2].

2.4.3 Hybrid Coders

Hybrid coders fill the gap for coding rates between 4.8-16kbit/s by incorporating the advantages of both vocoders and waveform coders in order to provide acceptable and natural speech at lower bit rates [4]. These codecs model the spectral properties of speech and exploit the perceptual properties of the ear for the minimal representation of the voice signal like the vocoders. Hybrid codecs produce more faithful waveform representation and as a result, more robust and better quality speech as the waveform coders [2].

Hybrid coders are broadly divided into two main categories: frequency domain and time domain. The frequency domain coders divide the speech spectrum into frequency bands

or components by using a filter bank or block transform respectively. These coders are based on the assumption that the signal is slowly time-varying. Hence the short-time segment of the input signal can be modelled with a short-time spectrum. The most commonly known coding schemes in this category are Sub-band Coding (SBC) and the Adaptive Transform Coding (ATC) that operate at bit rates between 9.6 to 16kbits/s. Another frequency-domain codec is the Multi-band Excited Vocoder (MBEV). This codec with effective pitch modelling can produce good quality speech for bit rates as low as 4.8kbits/s. A lower bit rate can be achieved by using a modified version of MBEV that represents the harmonic magnitudes by an LPC filter [3].

Time domain hybrids coder are very similar to the Linear-Predictive coders with a portion of the original signal to be transmitted instead of pitch and voicing information. They employ the speech source model described in section 2.3 in which the excitation is represented by a linear time-varying filter with a periodic pulse-train for voiced speech or a random noise for unvoiced speech. Though there are several forms of time domain hybrid coders, the most successful and commonly used are time-domain Analysis-by-Synthesis (AbS) codecs. Examples of AbS codecs are the Residual Excited Linear Prediction (RELP), the Code-Excited Linear Prediction (CELP), the Voice Excited Linear Prediction (VELP) and the Multipulse Excited Linear Prediction (MELP) coders [3].

2.4.3.1 Analysis by Synthesis

Analysis-by-Synthesis speech coders have been widely adopted as they produce good quality speech while maintaining a low bit-rate (between 4.8-16kbit/s) at the cost of high computational complexity [4]. In the AbS approach, the encoder (analysis) incorporates the decoder (synthesis) to determine the excitation signal and uses linear prediction techniques to calculate the coefficients of the speech synthesis filter. The basic structure of an AbS-LPC coding system is depicted in Figure 2-3. There are three main sub-blocks in the model that are used to obtain a good synthesised speech signal [3].

- Time-varying filter (synthesis filter)
- Excitation generator
- Perceptually based minimisation procedure

In the analysis procedure, the input speech is partitioned into blocks of samples (frames) whose length and update rate determines the bit rate of the coding scheme [4]. The decoded speech is produced by filtering the signal produced by the excitation generator through both a long-term (Pitch synthesis) filter and a short-term (LPC synthesis) filter. The excitation signal is found by minimising the mean-squared error over a block of samples. The error signal is the difference between the original and decoded signals and it is perceptually weighted by a weighting filter. In the end, the quantized filter parameters and the vector quantized excitation are transmitted to the decoder. As shown in Figure 2-3 the decoder uses an identical structure with the encoder, where the synthesized speech is generated by filtering the decoded excitation signal through the synthesis filter. The long-term predictor filter models the long-term correlation (spectral fine structure) in the speech signal and its coefficients are adapted at rates varying from 100-200 times/s. An alternative structure for the pitch filter is the adaptive codebook in which the filter is replaced by a codebook that contains the previous excitation at different delays. The resulting vectors are searched and the one that best matches is selected and scaled with an optimal scaling factor. The short-term synthesis filter models the short-term correlation (spectral envelope) in the speech signal. This is an all-pole filter with an order between 8 and 16 and its coefficients are determined using linear prediction techniques for each frame.

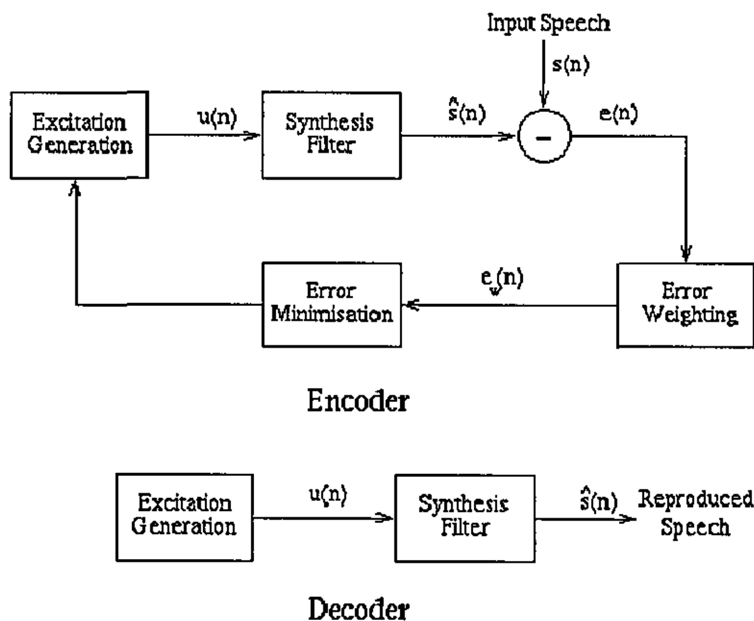


Figure 2-3: Analysis by Synthesis Code

The synthetic speech is generated in the encoder and decoder in order that both ends contain identical conditions in their filter memories. In this way, all the parts of the codec remain synchronised without the need for the memory parameters transmission. Preserving the identical conditions in both ends is one of the biggest challenges as this type of codec is very sensitive to channel errors [3]. Another important factor is the representation of the excitation signal of the time-varying filter. Three main excitation models for Analysis-by-Synthesis Linear Predictive Coding (AbS-LPC) are the multi-pulse model, the regular pulse excitation model and the vector or code excitation model [2].

The International Telecommunication Union (ITU) has created a number of speech coding standards for different voices qualities and bandwidth requirements. All current low-rate speech coders are based on AbS-LPC coding. In the following sections the two ITU standards, G.729A and G.723.1, studied in this research will be presented.

2.5 G.729A Speech Coding Standard

The G.729A [6] speech coding standard is a reduced complexity version of Conjugate-Structure Algebraic-Code-Excited Linear-Prediction (CS-ACELP) coder of the ITU G.729 recommendation [7]. It is designed for multimedia digital simultaneous voice and data (DSVD) applications though its use is not limited to these areas. G.729A grew from the need for low complexity (around 10 MIPS) speech codecs with speech quality equivalent to G.726 at 32kbits/s and operation bitrate of 11.4kbits/s and lower, in 1995. G.729A produces high quality speech (almost toll quality), in most conditions equivalent to G.726 at 32kbits/s, at a low bit rate of 8kbit/s. The complexity of this algorithm is typically 11 MIPS that is 50% less complex than G.729 (22 MIPS) with a small degradation in performance in the case of three tandems and in the presence of background noise [4]. The G.729A has a 5ms look-ahead, 10 ms processing delay, 10 ms transmission delay and the overall one-way system delay is 35ms. The amount of RAM that required is 3000 words [8]. This coder belongs to the time-domain Analysis-by-Synthesis class of speech coders. The encoder and the decoder dataflows of G.729A are depicted in Figure 2-4 and Figure 2-5 respectively [9].

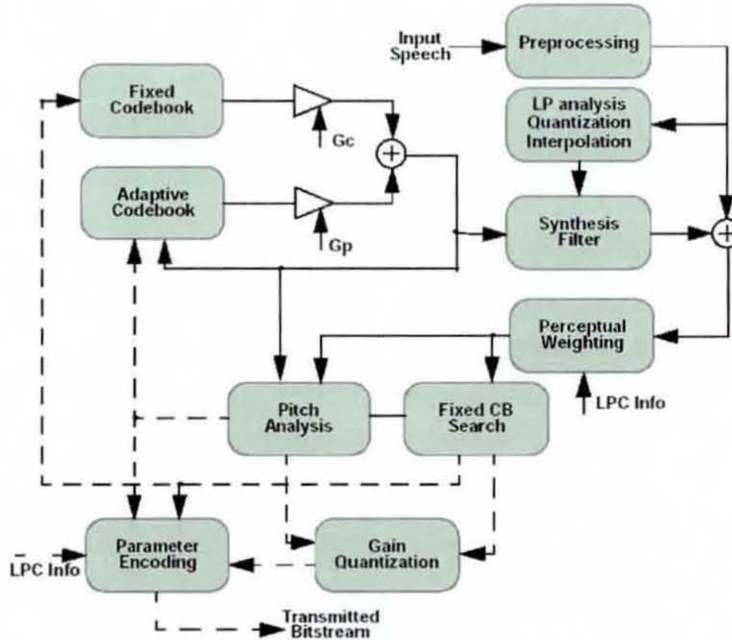


Figure 2-4: G.729A Encoder

The excitation for the synthesis filter is obtained by combining the outputs of two codebooks based on the analysis-by-synthesis search procedure. An adaptive codebook is used to model the long-term periodicities which represent the pitch (fine) structure of voiced speech and a fixed codebook that models the random noise-like unvoiced sounds such as nasal or plosive utterances. The excitation signal is then applied to a tenth-order synthesis filter whose transfer function models the human vocal tract. The residual error between the reconstructed speech produced by the synthesis filter and the original input speech is processed by a perceptual weighting filter in order to produce the perceptually weighted error. The minimization of this error determines the adaptive codebook index and gain for the optimum excitation sequence. The closed-loop search of the fixed codebook is implemented by using an algebraic codebook that simplifies the determination of the codebook parameters and makes real-time operation possible. The index and gains for both codebooks are assembled together with the synthesis filter coefficients to form the bitstream transmitted to the decoder. This entire process is repeated for every 10ms frame of the speech signal [7].

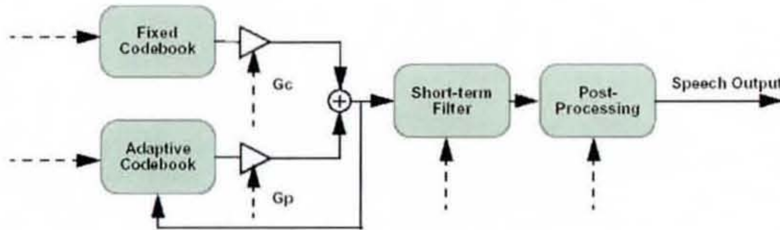


Figure 2-5: G.729A Decoder

At the decoder the received bitstream is used to extract and decode the encoder parameters corresponding to a 10 ms speech frame. These parameters give the synthesis filter coefficients and select the entries for the adaptive and fixed codebooks to represent the excitation to this filter. The excitation is constructed by adding the adaptive and fixed-codebook vectors scaled by their respective gains. The excitation is filtered afterwards by the synthesis filter and the speech is reconstructed. Additional post-processing of the reconstructed speech signal is performed to enhance its perceptual quality [7] [10].

Most of the G.729A codec is identical to G.729 with changes to the following parts of the codec in order to reduce complexity:

- The perceptual weighting filter uses a more traditional error weighting filter.
- The open-loop search for the pitch delay uses for the calculation of the autocorrelation function only the even samples of the weighted input.
- The closed-loop pitch search is achieved by maximizing a simpler (approximated) term than in G.729 that causes some degradation as the chosen adaptive codebook delay differs by 1/3 from the chosen in G.729.
- The algebraic codebook search is simplified by searching only 640 codebook entries per frame compared to 2880 codebook entries in G.729, using a depth-first tree search method.
- The decoder post-processing is simplified by using only integer delays and thus the complexity is reduced to 1 MIPS compared to 2.5 MIPS of G.729 [6].

2.6 G.723.1 Speech Coding Standard

ITU Recommendation G.723.1 [11] was designed for low-bit rate videophone, internet phone and particularly as part of the H.324 multimedia standard. The G.723.1 has two

transmitting bit rates at 5.3 and 6.3kbit/s. The higher bit rate has greater quality while the lower bit rate gives good quality and offers more design flexibility. It is possible to switch between the two rates at any 30ms frame boundary. The G.723.1 dual-rate codec was initially referred to as G.723. However, because under this name coexisted the older ADPCM-based G.723 standard, this scheme was renamed G.723.1 in order to avoid confusion. The G.723.1 is based on Linear Prediction Analysis-by-Synthesis coding carried out for 30 ms or 240-sample speech segments with a look-ahead of 7.5ms, giving a total delay of 37.5 ms [4]. This codec employs Algebraic-Code-Excited Linear-Prediction (ACELP) for its 5.3kbit/s rate and it has algorithmic complexity of 14.6MIPS. For its 6.3kbits/s mode of operation uses Multi-Pulse Maximum Likelihood Quantization (MP-MLQ) excitation and it has complexity of 16 MIPS. Both modes of operation use 2200 words of RAM [8]. Its dual-rate principle is very useful for intelligent multimode transceivers which are reconfigured at each speech frame boundary to provide more robust but lower speech quality or higher speech quality with less immunity to error. In addition, the G.723.1 utilises voice-activity controlled transmission (higher rate for active speech and lower rate for background) and comfort noise generation (CNG) for passive speech intervals [4].

The G.723.1 encoder operates on blocks of 30 ms [11]. Each block is first high-pass filtered to remove the DC components and then divided into 4 subframes of 60 samples each. For every subframe, the coefficients of the 10th order Linear Prediction Coding (LPC) filter are determined. The LP coefficients of the last subframe are converted to Line Spectral Pair (LSP) and quantized using a Predictive Split Vector Quantizer (PSVQ). The other subframes are used to construct the short-term perceptual weighting filter in order to obtain the perceptually weighted speech signal that is used for the open loop pitch period computation. The estimated open loop pitch period is used to construct a harmonic noise shaping filter. Then the combination of the LPC synthesis filter, the formant perceptual weighting filter, and the harmonic shaping filter produces the impulse response. An initial pitch period estimation is derived from the formant-weighted speech signal in an open-loop search. The impulse response along with the pitch period estimation is used for a more accurate closed-loop search which takes place in the fifth-order pitch predictor. Consecutively, the pitch period is calculated as a small differential value around the open loop pitch estimate and the effect of the refined pitch predictor is

removed from the speech signal. Depending on the operation mode, the resultant residual signal is subjected to either MP-MLQ for 6.3kbits/s rate or ACELP for 5.3kbits/s. Finally, the pitch period and the differential value along with the LPC coefficients are transmitted to the decoder [11]. The detailed block diagram of the G.723.1 encoder is depicted in Figure 2-6.

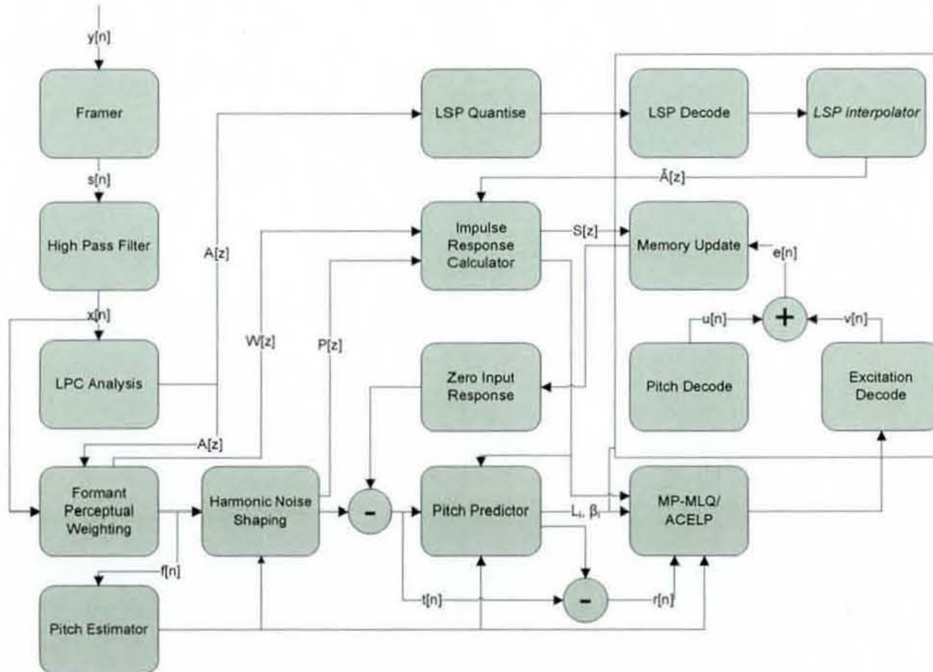


Figure 2-6: G.723.1 Encoder

At the decoder, the quantized LPC indices are decoded and used to construct the LPC synthesis filter. The adaptive codebook excitation and fixed codebook excitation are decoded for every subframe and feed the synthesis filter.

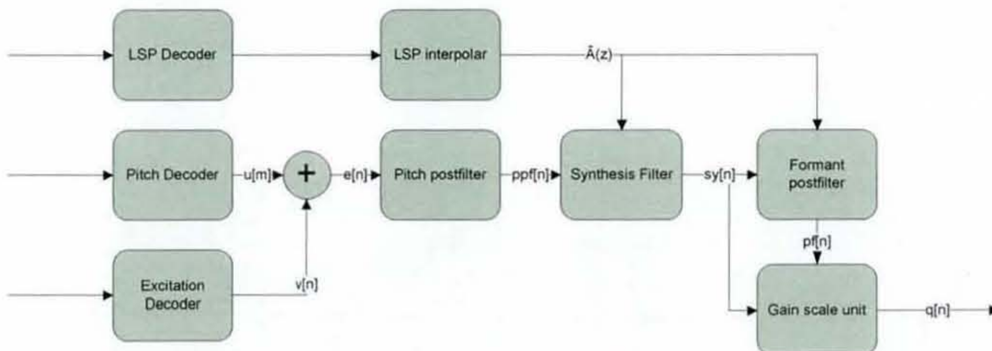


Figure 2-7: G.723.1 Decoder

The excitation signal input the pitch postfilter in order to improve the quality of the synthesized signal and the output of the postfilter feeds the synthesis filter consequently. The output of the synthesis filter feed the formant postfilter whose energy level is maintained by the gain scaling unit. The block diagram of the G.723.1 decoder is shown in Figure 2-7.

2.7 Summary

In this chapter a brief introduction of speech coding was given by discussing the coding objectives and requirements and presenting the basic speech source models. Consequently, the basic principles of the main coding techniques were introduced with more emphasis placed on the analysis-by-synthesis hybrid codecs as this type is employed in low bit-rate speech coders for multimedia applications. Finally, the characteristics and the basic operation of both ITU standards, G.729A and G.723.1, employed in this research were discussed.

2.8 References

[1] K. Diefendorff and P. Dubey, "How Multimedia Workloads Will Change Processor Design," in *IEEE Computer*. vol. 30, September 1997, pp. 43-45.

[2] A. S. Spanias, "Speech Coding: A tutorial review," *Proceedings of the IEEE*, vol. 82, pp. 1541-1582, October 1994.

[3] A. M. Kondoz, "Digital Speech: Coding for Low Bit Rate Communications Systems," John Wiley & sons, 1994, pp. 117-123.

[4] L. Hanzo, C. Somerville, and J. Woodard, "Voice Compression and Communications: Principles and Applications for Fixed and Wireless Channels," Wiley-Interscience, 2001, pp. 3-10, 65-67, 269-274.

[5] R. M. Nickel, "Automatic speech character identification," in *IEEE Circuits and Systems*. vol. 4, Fourth Quarter 2006, pp. 10-31.

[6] ITU-T Recommendation G.729A, "Annex A: Reduced complexity 8 kbits/s CS-ACELP speech codec," 11/96.

[7] ITU-T Recommendation G.729, "Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear-prediction (CS-ACELP)," 3/96.

[8] R. V. Cox and P. Kroon, "Low bit-rate speech coders for multimedia communication," in *IEEE Communications Magazine*. vol. 34, December 1996, pp. 34-41.

[9] ITU-T Recommendation G.729A, "Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear-prediction (CS-ACELP)," 3/96.

[10] K. Koutsomyti, S. R. Parr, V. A. Chouliaras, et al., "Scalar and parametric vector accelerators for the G.729A speech coding standard," in *Proceedings of IEE/ACM SoC Design, Test and Technology Postgraduate Seminar*, Loughborough University, September 2004, pp. 53-57.

[11] ITU-T Recommendation G.723.1, "Dual Rate Speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s," 3/96.

CHAPTER 3
SOFTWARE AND HARDWARE PARALLELISM

3.1 Overview of Parallelism

The proliferation of dynamic multimedia applications such as videoconferencing, image/speech processing and compression, 3D graphics, animation, Virtual Reality Modelling Language, encryption etc has changed the processing workloads of embedded processors significantly [1]. In order to run these multimedia codes efficiently and in real time there is need for high-performance application-specific processors. One approach to improve processor performance is to increase the clock speed. Though this may seem easy at first, the increase of a circuit's clock speed is a direct function of the chosen implementation technology. More importantly, this causes a high increase in the dynamic (switching) power dissipation rendering high-frequency designs unusable for power-constrained consumer applications. An alternative approach to improving processor performance is to increase the number of operations executed per clock cycle [2]. This approach yields very high performance and it is independent of the underlying circuit technology. In order to achieve this, multiple operations must be scheduled to execute in parallel in the extra functional units or processors. To make this, various techniques have been employed to exploit the inherit parallelism in modern applications and speed up their execution. The key to achieving high performance in current and emerging workloads is parallelism. The performance limit is set by the available parallelism in the application and the amount of the adaptation needed on the source code in order to allow the processor to exploit it [3].

The idea of parallelism to increase processor performance has been introduced as early as 1961 with the pipelining technique introduced by Stretch, the IBM 7030 processor [4]. Pipelining is a micro-architectural technique to exploit the parallelism that exists among the actions (steps) needed to complete the execution of an instruction. In this way, different parts of multiple instructions in a sequential instruction stream are overlapped in execution and thus, their completion time decreases [5]. Pipelining is the first form of the Instruction Level Parallelism (ILP) even though it is considered nowadays a low-level

parallelism mechanism. Another form of parallelism (DLP, TLP) was exploited in 1964 with the Control Data Corporation (CDC) 6600 CPU [6]. This processor used ten functional units that could operate in parallel and could perform ten unrelated operations per cycle introducing with this way the concept of Data Level Parallelism (DLP) and ILP. In addition, it had ten identical peripheral processors that operated independently and simultaneously and could execute up to ten programs at the time introducing the idea of Thread Level Parallelism (TLP). Conclusively addressing all forms of parallelism however is a recent achievement enabled by advances in silicon technology and EDA/tools/compilers. In the following sections an overview of the three main techniques of parallelism: ILP, DLP and TLP will be given with more emphasis in DLP as it forms the basic target of this research. Limitations in parallelism exploitation are imposed from dependences that are found in every code sequence. These dependences can cause structural stalls, data hazard stalls or control stalls thus reducing the performance [2]. There are three different types of dependences: data dependences, name dependences, and control dependences [2].

3.2 Data Dependences

An instruction is data dependent on another instruction if its execution uses as input a value created by a previous execution of the latter [7]. The data dependence implies the two instructions cannot execute in parallel or be overlapped as it will affect the correctness of the program [5]. An example of this type of dependence, also known as true data dependence, is illustrated in Figure 3-1.

```
S1    Loop:    ld     r1, [a0];       load array element a
S2             add    r4, r1, r2;     add array element to r2
S3             st     r4, [c, r0];    store result to array b
S4             add    r0, r0, #1;     increment counter
S5             bnez   r0, Loop;       branch to loop if r0!=0
```

Figure 3-1: Code snippet that shows the data dependences

The data hazards caused from data dependence are known as RAW (Read after Write), referring to the order in which instructions are presented in the pipeline. This type of data

hazard occurs when one instruction reads one register operand before that operand is produced from an earlier instruction, resulting in the use of the wrong register operand. Dependences are detected and data hazards are avoided within a processor with pipeline interlocks that force execution stall. In VLIW architectures a compiler performs the instruction scheduling and hides such data dependences rendering the use of interlock logic unnecessary [5].

3.2.1 Name Dependences

There are two types of name dependence: antidependence and the output dependence. An antidependence occurs when an instruction reads from the same register or memory location that another instruction writes. This gives rise to WAR (Write after Read) data hazards as an instruction writes in a destination before this is read from another instruction resulting in the latter reading the new (incorrect) value. This violates program semantics [5].

An output dependence occurs when two instructions write to the same register or memory location. This type of dependence causes a WAW (Write after Write) data hazard which occurs when the value written in the destination was written from the wrong instruction. Again the order is important as the final value must be from the first (in chronological order) instruction. Both types of dependences are not true data dependences and the involved instructions can execute simultaneously or even be reordered as long as the common register name or the memory location are renamed statically by a compiler or dynamically by the hardware [5].

3.2.2 Control Dependences

Control dependence determines the ordering of an instruction with respect to a branch in order for the instruction to execute the correct program order [5]. Hence an instruction that is control dependent on a branch (e.g. in the THEN statement of an IF conditional) cannot be moved before the branch so its execution is no longer controlled by this. In addition, an instruction that is not control-dependent on a branch (e.g. before an IF conditional) cannot be moved after the branch and its execution become controlled by the branch. Therefore, branches limit the ways that code can be re-arranged for optimum

execution performance. According to Intel, 20-30% of the processor performance is left un-tapped due to branch mispredictions [8]. Branch prediction and predication are some of the methods to increase the parallelism without causing any exceptions or changing the data flow [5].

3.3 Types of Parallelism

As mentioned above the objective behind exploiting parallelism at multiple levels is to maximise the execution performance of an application. Several architectural techniques have been employed to exploit effectively these forms of parallelism. Flynn's taxonomy (1986) [9] categorised computer architectures into four categories according to the parallelism in the instruction and data streams that they can handle:

- Single instruction single data (SISD)
- Multiple instruction single data (MISD)
- Single instruction multiple data (SIMD)
- Multiple instruction multiple data (MIMD)

According to Flynn's taxonomy a scalar uni-processor is classified as a SISD system as only one instruction is issued per cycle and that instruction operates on a single piece of data. MISD category hasn't been implemented in any commercial multiprocessor as it does not improve the performance of a system. However, it is expected to have application in fault-tolerant architectures for aerospace. Since it allows a degree of redundancy (it issues multiple instructions on the same dataset) it can be introduced in safety-critical systems. The other two categories correspond to the three different forms of parallelism: Data-Level Parallelism (DLP) is the case of SIMD where identical operations are applied in arrays of data. This form of parallelism is found typically within loops where the same transformations apply to arrays of data. Vector architectures are the most efficient means for exploiting this type of parallelism. Instruction-Level Parallelism (ILP) is a case of MIMD since it issues multiple instructions that operates in multiple data. This can be in the form of a number of microarchitectures differentiated by their instruction scheduling techniques and dispatch width. Finally, Thread-level parallelism (TLP) which is a different aspect of MIMD is regarded as one of the most profound forms of parallelism since it involves multiple processors operating in parallel. Within the TLP

domain, separate instruction streams execute on separate functional units (processor contexts) on separate (multi-programming) or the same (multi-threading) datasets.

Flynn's taxonomy does not apply precisely on today's architectures as modern embedded processors typically belong to more than one category in Flynn's taxonomy. It is a useful framework however in the processor design space. In the following sections an overview of the three different types of parallelism will be given along with the architectural techniques required to exploit these forms effectively.

3.3.1 Instruction Level Parallelism

Instruction-level parallelism (ILP) is the architectural technique that exploits the available parallelism at the instruction (operation) level and executes multiple such operations concurrently [2], [10]. This overlap in the execution of instructions is achieved by extracting independent instructions from a program sequence [11]. The idea of ILP appeared as early as 1960's in the IBM Stretch 7030 (1961) [4] and the Control Data 6600 (1964) [6]. Even though Stretch was a commercial failure, it introduced ideas such as pipelining and dynamic instruction issue mechanism based on Tomasulo's algorithm [12] that are in use even today [4]. Pipelining is a primitive form of ILP as it allows the execution of multiple instructions in different stages of the processor simultaneously. Thus, different multicycle operations may share the same hardware by using different parts of it in different cycles. Nowadays pipelining is considered a low-level mechanism that has contributed significantly to the performance of modern computers and since 1985 it is part of every processor architecture [5], [13]. Seymour Cray's CDC 6600 removed the instructions handling the memory and the I/O from the main CPU and implemented them in a set of peripheral processors. Additionally, it included ten functional units that performed arithmetical-logical instructions at the same time. In this way, the main CPU (arithmetical-logical instructions) and the peripheral processors (memory and I/O instructions) could operate in parallel improving considerably the performance and making it the world's fastest computer until 1969 [6]. In the following years there was a wide range of techniques that extended the idea of ILP and increased the amount of parallelism exploited among instructions.

A processor that employs ILP is typically called multiple-issue and follows a similar execution model as a normal RISC machine [10]. Resources operate in parallel and there may be a multiplicity of functional units that implement the same datapath functions in order to enable more parallelism. Thus, ILP involves two extra factors to accelerate programs: multiple issue and extra functional units. More than one operation can be issued in a given cycle and executed by using replicated or different functional units. ILP is dependent on developments in hardware technology such as circuit speed and power optimization [10] [14]. Since ILP is an architectural technique for achieving higher performance by executing multiple low level operations (such as adds, multiplies, loads etc) at the same time it requires special logic in the fetch stage of the processor [10]. This additional logic unrolls the program sequence and reschedules the order of the instructions in order to arrange multiple operations in a parallel manner before execution and avoid or reduce the stalls caused from data dependences while maintaining the program data flow [5].

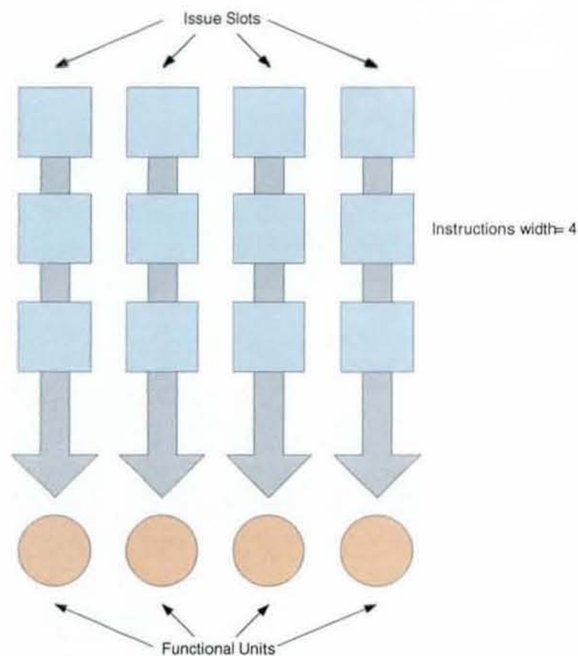


Figure 3-2: Multiple-issuing of instructions in an ILP architecture

These instructions are subsequently issued to the functional units that operate in parallel [15]. The number of the instructions that can be issued and executed each cycle determines the width of the processor. Figure 3-2 shows multiple-issue in a 4-wide ILP architecture. Special logic detects dependences, reorders the instruction sequence,

unrolling loops, and ensures that instructions are committed in order to maintain a precise-exception environment to software [10]. This is achieved using either dynamic or static scheduling [2]. The dynamic scheduling approach uses special logic to identify data dependences and rearrange the instructions dynamically in order to reduce stalls while maintaining the exception and data flow behaviour [2]. The disadvantage of this method is a large amount of extra hardware and thus, extra power consumption compared to an in-order processor. This approach is used in the superscalar and data flow processors [10]. On the other hand, static scheduling uses the compiler instead of dedicated hardware to exploit the available parallelism and keep busy as many functional units as possible. Advances in compiler technology can achieve a similar result thus disposing with all these hardware structures. Very Long Instruction Word (VLIW) [15] [16] processors employ such static, compiler-intensive scheduling. A compiler's ability to perform static scheduling depends on the amount of the available ILP in the program, the latencies of the functional units in the pipeline and the number of registers (storage) in the processors.

3.3.1.1 Superscalar Processors

Superscalar processors are either statically scheduled (using compiler techniques) with in-order execution or dynamically scheduled (using techniques based on Tomasulo's algorithm) with out-of-order execution [2]. Superscalar processors began to appear in the mid-to-late 1980s and for many years they were viewed as the logical next step in RISC movement [14]. There is a wide range of superscalar implementations with different degree of complexity ranging from the DEC Alpha [17] which has a strictly RISC ISA to the Intel X86 [18] that is considered a CISC ISA [14]. The first commercial single-chip superscalar microprocessors were the Intel i960CA (1988) [19] and the AMD 29000-series 29050 (1990) [20].

The statically scheduled approach was used in the early superscalar processors in which instructions were issued in order and all the types of hazards were checked at the issue time [2]. The pipeline control logic detects data or structural hazard only across the instruction packet currently at the decode stage. This type of superscalar processor employs hardware to perform the instruction issuing and hazard detection but scheduling uses software techniques. Sun UltraSPARC II/III are statically scheduled superscalar processors.

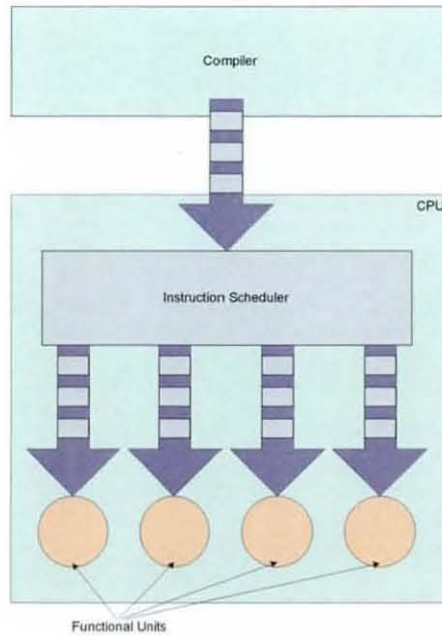


Figure 3-3: Dynamic Instruction Scheduling

Dynamically scheduled superscalar processors employ hardware to rearrange the instruction execution to reduce stalls and simultaneously dispatch multiple instructions per cycle to multiple functional units [14] [2]. This technique handles data dependences unknown at compile time (e.g. a memory reference) simplifying the compiler at the expense of hardware complexity. Additionally, it can reschedule an already compiled code to run on a different pipeline to increase the processor performance [2]. Figure 3-3 depicts the dynamic instruction scheduling of a superscalar processor. Dynamically-scheduled superscalar processors don't require any re-compilation of the source code as they adapt their execution behaviour dynamically according to the application binary. Such processors exhibit limited scalability due to their complexity. Superscalar processors employ speculative execution to overcome the limitation of control dependences caused by branches. Branch prediction is not sufficient in ILP case as, for a wide issue processor, one or more branches may execute in every cycle. Speculative execution combines dynamic branch prediction to select the instruction stream that will be fetched. There are hardware resources dedicated to undoing the effects of a misprediction and/or dynamic scheduling [2]. The dynamic scheduling approach dominates the desktop and server markets and it is used in many successful processors such as Pentium III and IV, MIPS R10000/12000, AMD Athlon, PowerPC etc [2].

3.3.1.2 VLIW Processors

The alternative to the superscalar approach is to employ compiler technology to check for dependences across the instructions of a program sequence, reorder them to minimize the potential hazard stalls and group them into fixed-length packets that will be issued to the processor. Each fixed-length packet resembles a very long instruction that contains multiple independent operations that can execute in parallel and for this reason this type of architecture was named Very Long Instruction Word (VLIW) [2]. Figure 3-4 illustrates static instruction scheduling of a VLIW processor.

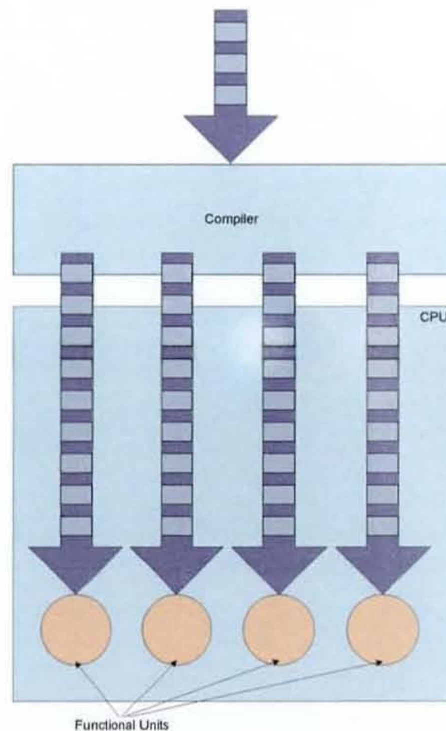


Figure 3-4: Static Instruction Scheduling

An early form of VLIW was processors using horizontal microcode, originally designed for signal processing applications [10]. An example of this type of processor designed to accelerate floating-point computations was the Floating Point Systems [21] FPS-164 and FPS-264 CPUs. These processors were very fast but limited in programmability and application area due to their complexity. The term VLIW was introduced by J. Fisher who developed a compiler that relied on trace scheduling in order to generate horizontal microcode (LIWs) for ordinary programs [22]. Trace scheduling is an optimizing compiler technology that performs loop unrolling and static branch prediction and allows

the processor to exploit the available parallelism beyond basic blocks [22], [2]. Additionally Fisher suggested the co-design of the compiler and the VLIW processor in order to simplify the scheduling algorithms. In 1980's they were three general-purpose VLIWs with varying degrees of parallelism [10]; TRACE from Multiflow Computers Inc [23], Cydra 5 from Cydrome [24] and the Culler-7 from Culler Scientific Systems. These processors, though not commercially successful, developed methods and technologies that influenced the VLIW design philosophy. Current examples of contemporary VLIW CPUs include the TriMedia media processors [25] by NXP (formerly Philips Semiconductors), the SHARC DSP by Analog Devices [26], the C6000 DSP family by Texas Instruments [27], and the STMicroelectronics ST200 [28] family based on the Lx architecture. These contemporary VLIW CPUs are primarily successful as embedded media processors for consumer electronic devices. In addition, the new Intel IA-64 [29] architecture utilizes VLIW techniques to create a scalable instruction-level parallel processor family.

Because of the nature of VLIW processor instructions, they are generally statically scheduled by a compiler removing the need for a complicated scheduling logic. In addition they are highly scalable but require the source code re-compilation across implementations [2]. VLIW is more effective as the number of issues per cycle becomes larger [2]. In the case that they are not enough independent instructions to execute in parallel the fixed-length packet includes NOP instructions which can lead to oversized code. There are several solutions for this problem; Sun MAJC and Tensilica's Xtensa LX2 [30] processor for example utilise variable-length packets to issue per cycle, Trimedia TM3270 compress the code stream in memory and un-compress them when they are loaded in the instruction code and so on.

An advanced form of VLIW that is not used in embedded processors and embodies new principles is the Explicitly Parallel Instruction Computing (EPIC) processors [2][16]. EPIC is a design philosophy that enhances instruction level parallelism and supports explicit parallelism. Explicit parallelism is supported by large parallel execution resources and large register files. EPIC architectures use the compiler to perform full speculative execution and instruction predication to increase parallelism in a program sequence. Speculation is a technique that reduces the effects of memory latency by performing speculative loading. Predication allows conditional execution without branches implying

larger basic blocks [31]. Furthermore, this type of architecture allows some degree of scalability in issue-width implementation to accommodate the resource limits in various applications [16]. The ISA that implements the ideas embodied in EPIC is the IA-64; the first implementation of that ISA was the Merced processor.

3.3.2 Data Level Parallelism

Data Level Parallelism (DLP) is a very important leverage in high performance computing. This paradigm uses vectorization techniques to operate in a large amount of independent data by executing a single instruction (vector instruction) simultaneously on arrays of elements [5]. Multimedia-rich applications involve real-time processing of continuous data streams in the form of vectors of packet 8- 16- and 32-bits integers and floating point numbers and undergo identical processing such as filtering, transformation etc. The microarchitectures capable of extracting this fine-grained data-parallelism are different than those used in fine-grained instruction-level parallelism. The most efficient method to exploit this type of data is by employing machines with SIMD hardware units that can execute whole loops in parallel [1]. These machines are known as vector processors and their advantages over the other architectures are explained in the following sections.

3.3.2.1 Advantages of vector architectures

The exploitation of DLP by the use of vector instruction architectures has many advantages compared to a classic scalar system. First, a vector instruction performs a number of individual operations in parallel thus it contains higher semantic content. Hence the vector program exhibits better code density compared to an equivalent scalar program and therefore smaller instruction fetch overhead. Higher code density implies less instruction fetch bandwidth and thus reduced pressure in the instruction fetch engine. The smaller overhead is due to fewer address computations and loop counter increments as well as branch computations. In addition, relatively simple control can dispatch a large number of operations every time and can better utilize the wide datapath [32]. Examples of the application kernels and the vectorization techniques employed in this work are described in more detail in Chapter 4. Another benefit that DLP machines deliver is better memory system performance than superscalar processors. Despite out-of-order

execution, non blocking caches and pre-fetching mechanisms, the predictive model for the caches is inefficient. This happens because the retrieved data from the previous level in the cache are not necessarily needed. Furthermore, since load/store instructions are mixed with computation and/or conditional execution, possible dependences and resource constraints prevent a memory operation to be performed on every cycle. Therefore the superscalar CPU cannot utilise efficiently the data cache subsystem in vectorized kernels. These problems are avoided in vector memory operations as the requested data usually have stride 1 of the memory pattern. By requesting an array of data with a single memory address a DLP machine uses effectively the available memory bandwidth without requiring extra issue slots and complex decode hardware. Thus by sending a simple address it can achieve a bandwidth of approximately N words per cycle. Finally, the datapath control remains simple as a vector engine can be easily scaled to higher levels of parallelism by replicating the functional units and adding wider paths from the vector registers to the functional units [32]. DLP however, is the least flexible form of parallelism compared to the ILP and TLP. It is also interesting to note that the available DLP in an application can also be exploited from non-DLP architectures by scheduling multiple independent instructions to execute in parallel in a superscalar architecture (ILP) or by computing the elements in parallel instruction streams in a multiprocessor system (TLP) [3]. This inflexibility though makes DLP the easiest form of parallelism that can be exploited with vector machines. These machines are easily scalable to exploit varying amounts of DLP in whole application domains. This is one of the greatest advantages of DLP over ILP since ILP architectures can't scale easily due to dependences between instructions which increase quadratically with the number of the parallel instructions loaded-up; TLP also requires duplicated instruction management logic for each instruction stream, duplicated processor state and suffers overheads from inter-thread synchronization and communication [3]. In addition, superscalar processors with wider issue (>4) exhibit diminishing performance and require large area dedicated to control rather than to datapath. Research has shown that vector processors are able to execute some highly parallel, integer based applications 1.5-7.3 times faster than superscalar processors [33]. Therefore vector processors with wide datapaths could lead to significant performance without increasing the hardware complexity of architectures that exploit the other forms of parallelism.

3.3.2.2 Vector Processors

In this section, the fundamental concepts of vector architectures are provided as this research is based on this processing paradigm. Vector processor architectures made their appearance in the late 1960s and early 1970s to support massive vector and matrix calculations. The first successful implementations of vector processors were the Control Data Corporation (CDC) STAR-100 [34] and the Texas Instruments Advanced Scientific Computer (TI ASC) [35] in 1964. These architectures were memory-to-memory with high bandwidth memory systems centred on a vector processing unit. However, they were not commercially successful due to the long start up overhead of vector instructions and the deep pipelining [36]. They did however presented several innovative ideas that influenced the design of vector supercomputers over the next years. A vector architecture with a different philosophy than the aforementioned was CRAY-1 computer system [37] which introduced in 1976 and it was the first commercially successful vector supercomputer. This machine was centred on scalar processing but it was using vector-register architecture and thus it had significantly lower overhead and less memory bandwidth requirements. CRAY-1 was the fastest processor of its time and its successors CRAY-2 and CRAY X-MP developed by two different groups of Cray Research were amongst the most successful vector machines until 1991. At the same period, CDC continued the development of memory-to-memory vector processors with the Cyber 200 series that was using the same basic architecture as the CDC STAR but offered better performance and wider vector datapaths. Still their performance could not compete with the CRAY machines since they had long memory latencies and could not handle efficiently non-unit strides [36] [38]. In 1980s, CDC created a group called ETA that built the supercomputer ETA-10 that again was based on the same memory-to-memory architecture of Cyber 200 series and had a configuration of up to 10 processors. This processor achieved a performance of 10 GFLOPS but its scalar performance was not as good and in 1989 its production stopped completely. In the 1980s smaller-scale vector processors appeared with the most successful designed by Convex and Alliant. At this time Japanese supercomputers made their appearance starting with the Fujitsu VP100, Hitachi S810 and the NEC SX/2 that were vector-register architectures with similar performance to the CRAY X-MP [36]. These computers continued to evolve with NEC SX/5 which was the fastest vector supercomputer in 2001 with a 16 processors configuration clocking at 312 MHz and Fujitsu VPP5000 with a 128 processors

configuration clocking at 300 MHz. Historically, the fastest supercomputer was the CRAY-4 with 64 processors running at 1 GHz but it was never completed as the company went bankrupt in 1995 [36]. After the appearance of superscalar architectures in the early nineties research was concentrated on superscalar and VLIW architectures as there was the prevailing belief that vector processing would be redundant [3]. Multimedia-rich applications becoming the dominant application domain however has changed the computer architecture and microprocessor design and the interest for vector processing has been revived [1].

Vector architectures can be either memory-to-memory or register-to-register based with the latter being the most dominant type. A typical vector processor consists of pipelined scalar and vector units. The scalar unit handles memory addressing and control where as the vector unit performs the actual processing. Vector architectures are similar to RISC architectures with instruction sets that include arithmetic and memory instructions but instead of processing scalar values they execute the same operation simultaneously on arrays of elements. In other words, a single opcode defines a large number of identical yet independent, operations on the elements of one or more arrays. The arrays of operands are stored in a vector register file in a similar way with the operands in RISC architecture. However the vector register file is a two-dimensional storage array where each row contains all the elements of a single vector [36]. The number of the elements per register is defined by the vector processor ISA/programmer model. A general vector processor architecture is depicted in Figure 3-5. It consists of a number of functional units that operate in parallel. Each unit is fully pipelined and can start a new operation every clock cycle. The vector functional units generate interim results that are used immediately without the time-costly memory references that slowed down the first vector computers [37]. This takes place in combination with the scalar unit which detects structural and data hazards and handles memory accesses.

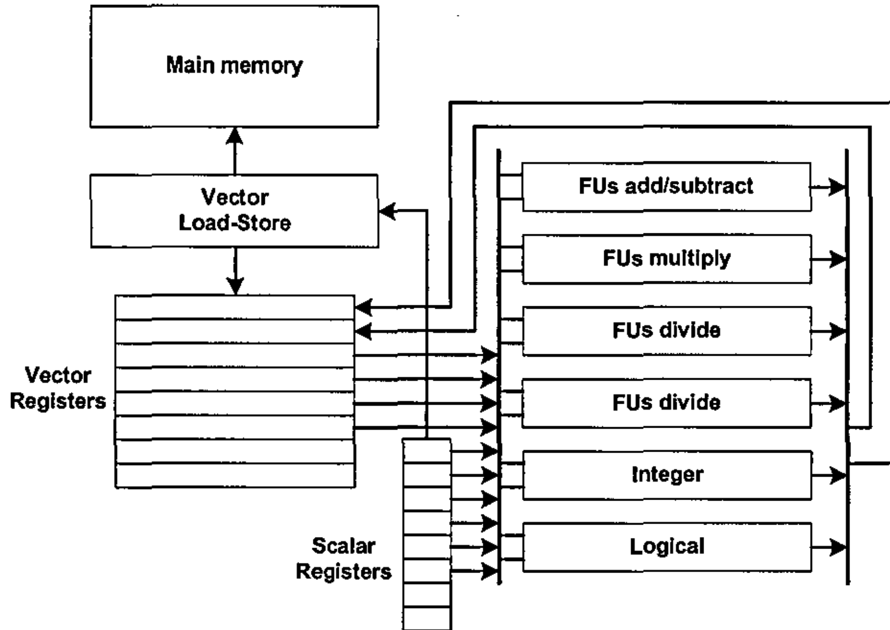


Figure 3-5: Basic Vector Processor Architecture

Each vector register has at least two read ports and one write port in order to allow RISC-like 3-operand execution. Another important component of a vector processor is the load-store unit responsible for loading vectors from and store to memory and it is fully pipelined [39] [36]. A more detailed description of the vector processor architecture and microarchitecture developed in this work is given in Chapter 5 and 6.

3.3.3 Thread Level Parallelism

Another approach to achieve high execution performance is by exploiting the available parallelism at the thread/process level. Architectures that exploit this form of parallelism belongs to the MIMD category [2]. Such architectures consist of a collection of interconnected single-thread processors, with each processor executing independent instructions streams operating on multiple data items. When the processors run independent tasks (programs) this is the case of a multiprogrammed environment. When the multiple processors execute different parts of the same program and share most of their address space this is known as multithreading. The independent parts or processes of the program are called threads. These threads execute concurrently and define another type of parallelism that is known as Thread-Level Parallelism or TLP [2]. TLP is a coarse-grained type of parallelism since each processor works on a specific process and

communicates with the other processors only if necessary. The theoretical performance improvement on n-wide TLP processor is n-fold compared to a single processor where n is the number of the processors that comprise the multiprocessor.

There are two classes of MIMD multiprocessors depending on the number of the processors, the memory organization, and the type of their interconnection: The centralised shared-memory architecture and the distributed-memory architecture [2].

3.3.3.1 Shared-Memory Architecture

Shared-memory architectures consist of a number of processors that share the same memory and are connected via some interconnect scheme typically a bus. When the single main memory has similar (symmetric) access time from all processors this is the case of Symmetric Multiprocessing (SMP) or Uniform Memory Access (UMA) [2].

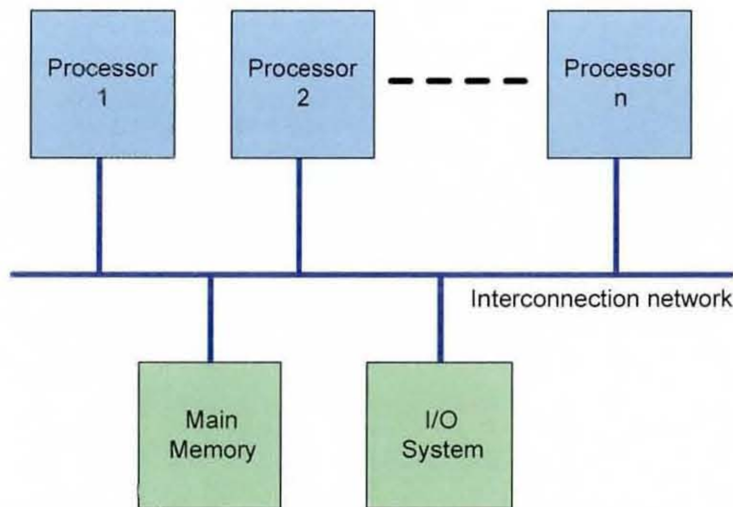


Figure 3-6: The basic architecture of a centralised shared-memory multiprocessor system

In shared-memory architectures it is easier to balance the processor workload efficiently. This class is the most popular organization with a reasonably simple programming model and it is used in tightly-coupled architectures [40]. Support for SMP must be built into the operating system in order to take advantage of the additional processors. SMP was first implemented on the Burroughs B5500 in 1961 and by 2006 has dominated the server and workstation market. With the introduction of dual-core devices, it became prevalent in most new desktops and laptops such as Intel's Xeon and Core Duo, AMD's Athlon64 X2

and Opteron etc that use the x86 instruction set; other non-x86 architectures are Sun Microsystems UltraSPARC, Intel Itanium, Hewlett Packard PA-RISC etc and are used primarily in the server domain. An alternative architecture is the Asymmetric Multiprocessing or ASMP in which only specific locations in memory and specific tasks are allocated for each processor. An example of this architecture can be found in the high-performance 3D chips in modern videocards.

3.3.3.2 Distributed-Memory Architecture

The second class is known as Distributed-Memory architectures in which the memory is physically distributed among a number of processors. This approach can more easily support the bandwidth demands of the individual processors as there is no need to access a centralised resource as the shared memory.

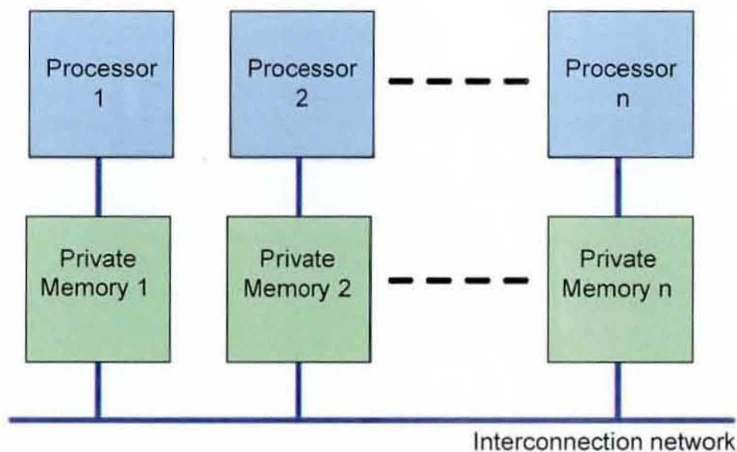


Figure 3-7: The basic architecture of a distributed-memory multiprocessor system

The interconnection between processors and memory can be direct (direct interconnection networks) using for example switches or indirect using typically multidimensional meshes [2]. The distributed-memory architecture can be implemented by using two different approaches for communicating data among processors. In the first approach the communication takes place through a shared address space. This happens by addressing the physical separate memories as one logically shared address space. The multiprocessors that are using this approach are called Distributed Shared-Memory (DSM) multiprocessors. DSM multiprocessors are also known as Non-Uniform Memory Access (NUMA) since the access time depends on the data word location in memory. An

alternative approach is when the address space of the processors consists of multiple and logically disjoint address spaces and the same physical address corresponds to two different locations of two different processors memories. Each processor-memory module is a separate computer and this type of architecture is called a multicomputer. Additionally, a multicomputer can consist of separate computers connected in a local area network, known as a cluster. This approach is very cost effective when little or no communication is required [2].

3.3.3.3 Multithreading Architecture

Multi-threaded processors are based on a hybrid approach that combines ILP and TLP and improve performance by exploiting the pipeline parallelism available through multiplexing independent threads. In this case, multiple threads execute concurrently and share the functional units of a single, wide processor. Each thread has a separate register file, program counter and memory page table that are duplicated in the processor (processor contexts). Multi-threaded processors hide the operation latency by switching threads at appropriate times or by interleaving operations from multiple threads at the same time using superscalar techniques. Apart from successfully hiding operation latency, multi-threaded processors improve processor utilization by keeping active many functional units on every cycle. There is special hardware to switch between different threads [2]. When one thread runs until it is blocked by an event that would cause a long latency stall such as level-2 cache miss (need to access an off-chip memory) execution switches to another thread that was ready to run. This technique is called blocked or coarse-grained multithreading [41]. This is the simplest type of multithreading that issues instructions from only a single thread per cycle and it is effective on high-cost stalls [41] [2]. Another alternative is when the switching between threads takes place on every instruction in order for the execution of multiple threads to be interleaved. This is called interleaved or fine-grained multithreading [41]. This type of switching occurs each clock cycle and eliminates control and data dependence stalls from the execution pipeline since threads are relatively independent from each other. In this technique the processor skips any threads that are stalled at that time and it has a very simple and fast pipeline. Similarly with the blocked multithreading, this type also issues from a single thread. When instructions can be issued from multiple threads per cycle this is the case of Simultaneous Multithreading (SMT). SMT is the most advanced type of multithreading

and it is a variation of the fine-grained multithreading that applies to superscalar processors to exploit the available ILP and TLP across multiple threads [2]. Simultaneous multithreading improves utilization by sharing many of the resources within the processor and can enhance the performance of a superscalar when the available ILP is not enough [41]. This technique was first researched by IBM in 1968 and the first commercial CPU was the DEC 21464 [42]. In another architectural extreme lies the Chip Multiprocessing (CMP). CMP enables multiple cores to share chip resources such as the memory controller, off-chip bandwidth and the L2 cache improving this way the utilisation of these resources [43]. It is an integrated form of Symmetric Multiprocessing and in this configuration, instead of having separate processing units in the computing system, the individual processors (CPU cores) are integrated in a single high performance chip.

3.3.4 Hybrid Approaches and Research

The various forms of machine parallelism are not clearly separated and they can be combined to increase even further the computer performance. For example, the NEC SX-4 vector supercomputer is a pipelined superscalar vector microprocessor architecture which can exploit ILP, DLP, and TLP [3]. Simultaneous multithreading processors employ TLP and ILP in the same time [44]. Another example that combines all the parallelism techniques is the SS_SPARC [45] which is a configurable, extensible, simultaneous multithreaded vector processor. More details of this processor are given in Chapter 7. There has also been a great amount of interest in the addition of extensions in existing instruction sets to accomodate vector processing. Examples of general-purpose microprocessors with vector extensions are Intel's MMX [46], PowerPC's Altivec [47], Sun UltraSparc's VIS [48] and Tarantula [49] that adds to Alpha (EV8) a vector unit. Another interesting combination is the merge of ILP and DLP paradigms in a single architecture [32] and the SMV architecture that combines simultaneous multithreading and DLP [50]. Research is currently underway into the potential performance benefit obtainable through the combination of different forms of parallelism within a single system-on-chip architecture.

3.4 Summary

This chapter presented an overview of parallelism and the performance advantages of exploiting it within given architectures. The limitations and the hazards caused by dependences across instructions in the application binary were also presented along with their main types. In addition, the three basic forms of parallelism were introduced and the processor architectures that exploit them together with their advantages and disadvantages.

3.5 References

[1] K. Diefendorff and P. Dubey, "How Multimedia Workloads Will Change Processor Design," in *IEEE Computer*. vol. 30, September 1997, pp. 43-45.

[2] Kevin W. Rudd, "VLIW Processors: Efficiently Exploiting Instruction-Level Parallelism," in *Electrical Engineering*: PhD Thesis, Stanford University, December 1999.

[3] K. Asanovic, "Vector Microprocessors," PhD Thesis, University of California at Berkeley, May 1998.

[4] W. Buchholz, *Planning a computer system: Project Stretch*: McGraw-Hill Inc, 1962.

[5] John L. Hennessy and David A. Patterson, "Computer Architecture: A Quantitative Approach," 3 ed: Morgan Kaufmann, 2003.

[6] J.E. Thornton, "Parallel Operation in the Control Data 6600," in *Proceedings of the 26th AFIPS Conference*, 1964, pp. 34-40.

[7] R. Allen and K. Kennedy, "Automatic translation of FORTRAN programs to vector form," *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 9, pp. 491 - 542, October 1987.

[8] John Crawford and Jerry Huck, "Motivations and Design Approach for the IA-64 64-Bit Instruction Set Architecture," in *Microprocessor Forum*, San Jose, California, October 1997.

[9] M. J. Flynn, "Some Computer Organisations and Their Effectiveness," *IEEE Transactions on Computers,* vol. 21, pp. 948-960, 1972.

[10] Joseph A. Fisher and Ramakrishna Rau, "Instruction-level Parallel Processing," *Science,* vol. 253, pp. 1233-1241, September 13 1991.

[11] Roger Espasa and Mateo Valero, "Simultaneous Multithreaded Vector Architectures Merging ILP and DLP for High Performance," in *the Proceedings of the Fourth International Conference on High-Performance Computing,* December 1997, pp. 350-357.

[12] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development,* pp. 25-33, January 1967.

[13] Ralph Duncan, "A Survey of Parallel Computer Architectures," in *IEEE Computer*, February 1990, pp. 5-16.

[14] James E. Smith and Gurindar S. Sohi, "The Microarchitecture of Superscalar Processors," in *Proceedings of the IEEE*. vol. 83, December 1995, pp. 1609-1624.

[15] Alexandru Nicolau and Joseph A. Fisher, "Measuring the Parallelism Available for Very Long Instruction Word Architectures," *IEEE Transactions on Computers,* vol. 33, pp. 968-976, November 1984.

[16] J. A. Fisher, P. Faraboschi, and C. Young, "Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools," Morgan Kaufmann, 2005.

[17] R. L. Sites, "Alpha AXP Architecture," in *Communications of the ACM*. vol. 36, February 1993, pp. 33-44.

[18] K. Diefendorff, "Pentium III = Pentium II + SSE: Internet SSE Architecture Boosts Multimedia Performance," in *Microprocessor Report*. vol. 13, March 1999.

[19] Steve McGeady, "Inside Intel's i960CA superscalar processor," in *Microprocessors and Microsystems*. vol. 14, July 1990, pp. 385-396.

[20] Daniel Mann, "Evaluating and Programming the 29K RISC Family," Advanced Micro Devices (AMD), 3d edition 1995.

[21] A. E. Charlesworth, "An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family," in *IEEE Computer*. vol. 14, 1981, pp. 18-27.

[22] Joseph A. Fisher, "Very Long Instruction Word architectures and the ELI-512," in *Proceedings of the 10th annual international symposium on Computer architecture*, Stockholm, Sweden, 1983, pp. 140-150.

[23] R. P. Colwell, R. P. Nix, J. J. O'Donnell, et al., "A VLIW architecture for a trace scheduling compiler," in *ACM SIGARCH Computer Architecture News*. vol. 15, October 1987, pp. 180-192.

[24] G. R. Beck, D. W. L. Yen, and T. L. Anderson., "The Cydra 5 minisupercomputer: Architecture and implementation," *The Journal of Supercomputing*, vol. 7, pp. 143-180, May 1993.

[25] J. W. Van de Waerdt, S. Vassiliadis, D. Sanjeev, et al., "The TM3270 media-processor," in *MICRO '05: Proceedings of the 38th International Symposium on Microarchitecture*, November 2005, pp. 331-342.

[26] Analog Devices, www.analog.com/processors/sharc/.

[27] "Processor Comparison: Texas Instruments C6000 DSP and Motorola G4 PowerPC," http://www.pentek.com/dspcentral/powerpc/articles.cfm.

[28] Benoit Dupont de Dinechin, "From Machine Scheduling to VLIW Instruction Scheduling," *ST Journal of Research Processor Architecture and Compilation for Embedded Systems* vol. 1, September 2004.

[29] Martin Hopkins, "A Critical Look at IA-64: Massive Resources, Massive ILP, But Can It Deliver?," in *Microprocessor Report*, February 2000.

[30] R. E Gonzalez, "Xtensa: A configurable and extensible processor," in *IEEE MIicro*, March/April 2000, pp. 60-70.

[31] R. Arnold, R. Bhatia, and D. Soltis, "Reducing the Physical Cost of Large Register Files in EPIC Architectures with Stacked Register Aliasing," in *Proceedings of the Workshop on EPIC Architectures and Compiler Techniques*, Istanbul, Turkey, November 2002.

[32] Francisca Quintana, Roger Espasa, and Mateo Valero, "A Case for Merging the ILP and DLP Paradigms," in *6th Euromicro Workshop on Parallel and Distributed Processing*, Madrid, Spain, 1998, pp. 217-224.

[33] C. G. Lee and D. J. DeVries, "Initial Results on the Performanc and Cost of Vector Microprocessors," in *the Proceedings of the 30th Annual International Symposium on Microarchitecture*, 171-182, December 1997.

[34] R. G. Hinz and D. P. Tate, "Control data STAR-100 processor design," in *IEEE COMPCON*, September 1972.

[35] W. Watson, "The TI-ASC, A highly modular and flexible super computer architecture," in *American Federation of Information Processing Societies AFIPS*, 1972, pp. 221-228.

[36] John L. Hennessy and David J. Patterson, *Computer Architecture: A Quantitative Approach* 2nd ed.: Morgan Kaufman, 1996.

[37] Richard M. Russell, "The CRAY-1 computer system," *Communications of the ACM* vol. 21, pp. 63-72, 1978.

[38] R. Espasa, M. Valero, and J. E. Smith, "Vector architectures: past, present and future," in *Proceedings of the 12th international conference on Supercomputing*, Melbourne, Australia, 1998, pp. 425-432.

[39] C. Kozyrakis, "A Media-Enhanced Vector Architecture for Embedded Memory Systems," Technical Report: CSD-99-1059, University of California at Berkeley 1999.

[40] Rajkumar Buyya, *High Performance Cluster Computing: Architectures and Systems* vol. 1, 1999.

[41] T. Ungerer, B. Robic, and J. Silc, "A Survey of Processors with Explicit Multithreading," in *ACM Computing Surveys (CSUR)*. vol. 35, March 2003, pp. 29-63.

[42] M. Meswani and P. J. Teller, "Evaluating the Performance Impact of Hardware Thread Priorities in Simultaneous Multithreaded Processors using SPEC CPU2000," in *2nd International Workshop on Operating Systems Interference In High Performance Applications*, Seattle, WA, September 2006.

[43] L. Spracklen and S. G. Abraham, "Chip Multithreading: Opportunities and Challenges," in *Proceedings of the 11th Intel Symposium on High-Performance Computer Architecture*, 2005.

[44] S. J. Eggers, J. S. Emer, H. M. Levy, et al., "Simultaneous Multithreading: A Platform for Next-Generation Processors " in *IEEE Micro*. vol. 17, October 1997, pp. 12-19.

[45] V. A. Chouliaras, K. Koutsomyti, T. Jacobs, et al., "SystemC-defined SIMD instructions for high performance SoC architectures," in *13th IEEE International Conference on Electronics, Circuits and Systems*, Nice, France, December 2006, pp. 822-825.

[46] A. Peleg and U. Weiser, "MMX Technology Extension to the Intel Architecture," in *IEEE Micro*. vol. 16, August 1996, pp. 42-50.

[47] K. Diefendorff, P. K. Dubey, R. Hochsprung, et al., "AltiVec Extension to PowerPC Accelerates Media Processing," in *IEEE Micro*. vol. 20, March 2000, pp. 85-95.

[48] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, et al., "VIS Speeds New Media Processing," in *IEEE Micro*. vol. 16, August 1996, pp. 10-20.

[49] R. Espasa, F. Ardanaz, J. Gago, et al., "Tarantula: A Vector Extension to the Alpha Architecture " in *the Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)* Anchorage, Alaska, 2002, pp. 281-292.

[50] R. Espasa and M. Valero, "Exploiting Instruction- and Data-Level Parallelism," in *IEEE Micro*. vol. 17, September 1997, pp. 20-27.

CHAPTER 4
METHODOLOGY AND ARCHITECTURAL RESULTS

4.1 Introduction

This chapter presents the optimization methodology and the architectural exploration of the ITU G.729A and G.723.1 speech coders for a data parallel processor. The methodology addresses target-independent optimizations of both reference codes. Thus, these workload optimizations presented here can be utilised on any DSP code with data-parallel infrastructure for acceleration. As instruction level simulation was the base for the adopted experimentation methodology, the description of the software tools and their development to suit the purpose of this research is given in the following sections along with a briefly survey of computer systems simulators. Both speech coding algorithms were benchmarked using the SimpleScalar toolset [1], before and after the data-parallelization and optimization to obtain the instruction count (also called dynamic instruction count). The instruction count is the total number of instructions executed by an ideal scalar processor when running the codes. Using this information, the vectorization of the speech algorithms was performed and performance improvement recorded after every new vector instruction was introduced. Finally, the architecture of the coprocessor was defined.

4.2 Simulation Infrastructure

There is a growing need for efficient techniques to predict the performance of future computer systems and evaluate candidate, novel microarchitectures in the research phase of a new computer before implementing them in hardware [2]. Simulation has been essential for the research and design of processors, compilers or any hardware that comprises a computer system or platform. It accelerates the hardware development process by employing software models for the proposed hardware. Simulation can reveal the dynamic characteristics of the hardware model and the software system that executes on it and allows for rapid design space exploration. Such models can be implemented in

traditional programming language such as C/C++ or hardware description language such as Verilog and VHDL and then, exercised with appropriate workloads to validate the performance and correctness of the proposed hardware at very early stage. In addition, computer system models allow the developing and testing of the software before the hardware is available [3], [4], [5]. Typically, such software models are substantially slower than the equivalent hardware; however they can be built in very short time [1]. The implementation of the software model can vary in the following quality features/requirements:

Performance: The performance depends on the amount of the workload that can be exercised. The greater the number of workloads that can exercise the model, the more thorough the model study and verification can be, ensuring this way increased probability of correct-by-construction design. Performance has to do also with the speed (simulated MIPS) of the actual simulation model and therefore with the speed of each of the component that comprise the latter.

Detail: The detail or simulation model accuracy determines the level of abstraction of the implemented model's components. It can describe the simulated system from a purely functional processor state level all the way to cycle-accurate timing including memory wait states and interrupt latency of all its components. Different levels of abstraction provide complementary amounts of information to the system designer however, at increased execution time.

Flexibility: Flexibility indicates how well structured is the simulator to easily modify or add design variants of the simulated system in order to re-use it for slightly or completely different models.

There is a trade-off between these three aspects of the computer system simulators. A highly detailed model can faithfully simulate all aspects of the system's operation but does so at low execution speeds and has reduced flexibility. On the other hand, a simpler model is less accurate but faster and certainly more flexible. Thus, there are several different simulator implementation models that meet different requirements in terms of performance, detail and flexibility. There is ongoing research in this area as researchers strive to achieve a reasonable trade-off [1, 6].

Certain types of simulators are described with an Architecture Description Language (ADL) [7], [8]. ADLs are computer languages designed specifically for representing and analysing system's microarchitecture. Such formal descriptions of architecture and microarchitecture have been the subject of research for years [9] and several models and techniques have been proposed on this front in an effort to facilitate architecture and microarchitecture description and space exploration. ADL-based simulators belong primarily to one of the two categories depending on whether the ADL captures the behaviour (instruction-set) or the structure (microarchitecture) of the system [10]. Recently, a third category has emerged which combines effectively both, behaviour and microarchitecture [7]. Behaviour-centric [7] also known as instruction set [10], [11] simulators describe instruction functionality but don't allow detailed pipeline and control-path specification. They are primarily used during the development phase of architecture (before the actual hardware specification is written and implementation begins) providing an execution model of the system and thus, writing the first programs and testing the compiler code generation [11]. Such ADLs are good for regular architectures and provide programmer's model but they are tedious for irregular architectures [8]. This type of simulators is simple, relatively fast (low MHz range) and can be easily retargeted to various ISAs. Examples of ADLs generated behaviour-centric simulators are nML [12], ISDL [13], ISPS [14]. nML is based on the concept that the majority of instructions share common properties. By exploiting these common properties, a hierarchy scheme is developed to describe instruction sets. The instructions are the topmost elements in the hierarchy and partial instructions are the intermediate elements. Each instruction definition in nML can be in the form of an AND-OR tree of intermediate elements that has a few attributes [10], [12]. The Instruction Set Description Language (ISDL) [13] was developed at MIT in order to express parallelism with explicit specification and it targets mainly VLIW processors [10]. The Instruction Set Processor Specification (ISPS) [14] appeared in the early 1970's and has been the basis for many design tools [15]. ISPS was used to model the architecture of processors and analyse their performance rather than to describe a complete computer system [15], [10]. On the other hand, structure-centric [7] also known as cycle-accurate [11] simulators simulate the microarchitecture of a system and provide performance metrics such as cycle counts, cache hit ratios and resource utilization statistics amongst others. Examples of structure-centric simulators are MIMOLA and UDL/I. MIMOLA [10] describes application programs with a Pascal-like

syntax while the processor model has the form of a component netlist. UDL/I [16] stands for the Unified Design Language for Integrated circuit. It is a Register Transfer level description language for simulation and logic synthesis. The techniques that are used in this ADL category to describe in detail the computer microarchitectures are very complex, quite slow and sometimes architecture specific [11]. Mixed type of simulators such as LISA and EXPRESSION capture both the structure and behaviour of the architecture. The Language for Instruction Set Architecture (LISA) [17] explicitly models both the datapath and control that are necessary for cycle accurate simulation. This description comprises two types of declaration: resources and operations. Resources refers to hardware structures such as registers, pipelines and memory systems whereas operations are the basic objects that represent the programmer's view of the behaviour, structure and the instruction set of the architecture. EXPRESSION [8] describes a processor as a netlist of functional units and storage elements and automatically generates Reservation Tables (RT) based on that netlist. Thus netlist representation is at a higher level of abstraction, similar to a block-diagram level description.

Simulators are also classified depending on whether they are trace driven [5] or execution driven [2],[15]. Trace-based simulation is a more traditional simulation technique that uses a stream of pre-recorded instructions to drive a hardware timing model. It employs a variety of techniques, both hardware and software, in order to obtain the instruction traces. Such techniques include hardware monitoring, binary instrumentation that inserts probe functions at various location in the to-be-traced code in order to collect event traces or trace synthesis [1]. Trace-based is faster than execution driven simulation but requires large amount for storage of traces and incurs large time overheads as traces can contain billions of references. In addition, it can be less accurate because of the difficulty in characterizing the behaviour of real programs stochastically meaning that it can capture only a part of processor behaviour e.g. cache misses. Since a trace is obtained from logical execution paths of a workload it can't model speculative execution such as branch directions or load addresses [2]. On the other hand, execution-driven simulation permits greater accuracy as the execution of the program and the simulation of the architecture are closely related and interleaved. It can reproduce a device's internal operation by replicating the execution of instructions on the simulated machine. In this way it provides all the data produced or consumed inside all microarchitecture components. The typical

output of this type of simulation is a large number of statistics that can help to understand how the components of the simulated system behave and a precisely-estimated execution time. Execution-based simulation can be also employed for dynamic power analysis as it can precisely record the change in the inputs of microarchitecture blocks and calculate relative dynamic power metrics accordingly. The drawbacks of the execution driven simulation are the high model complexity and the difficulty in reproducing experiments [1]. There is ongoing research to overcome these issues such as retargetable instruction set simulators [18], where the goal is to generate a simulator automatically from a machine description language. Additionally, traces can record the precise system state and can help to recreate the record-of-execution [1].

Finally, simulators can be classified depending on the amount of detail that they employ for system representation from Instruction-accurate simulators (ISS) [4], [18], [1] to Cycle-accurate simulators (CAS) [6], [19]. ISS imitates the behaviour of a mainframe or microprocessor by "executing" instructions and maintaining internal variables which represent the processor's registers. The ISS represents the system at a higher level of abstraction allowing the development of this simulator in short time. It is preferred from the cycle-accurate simulators in the early stages of a project to model fast the architectural features of the system but it can be also used in later stages to validate the functionality of the system since it can rapidly run the complete benchmark. The ISS however can't be used for performance analysis as they don't contain pipeline detail or timing issues [4], [18]. The Cycle-accurate simulators on the other hand, can perform timing (CPI) analysis and give quite accurate performance estimates. They are more complex to develop because of the great amount of detail and thus more time-consuming and lower speed than the ISS. Additionally, different CAS need to be developed for any new implementation of an architecture whereas the ISS undergo only minor changes between implementations of the same architecture [19].

4.2.1 SimpleScalar Toolset

The primary architecture exploitation was carried out on the Version 3.0 of the SimpleScalar tool set that is publicly available. Since its release in the Opensource (1995) SimpleScalar has been widely used for research in the computer architecture community [18]. The toolset provides an infrastructure for simulation and architectural modelling that

simplifies the implementation of hardware models for simulation of complete applications [1], [18]. It can perform program performance analysis, measure the dynamic characteristics of the hardware model and contribute to the software-hardware co-verification and co-optimization. It comprises of a compiler, assembler, linker and simulation tools for a range of modern processors architectures. SimpleScalar comprises several simulator models ranging from a simple functional instruction emulator (sim-safe) to a detailed microarchitectural model with dynamic scheduling (sim-outorder). Table 4-1 lists the seven simulators at different level of microarchitectural abstraction that are contained in the current release (version 3.0) of SimpleScalar. These simulator models are Instruction Set Simulators (ISS), also called functional, apart from the sim-outorder which is full cycle-accurate simulator and provides detailed microarchitectural timing [1].

Table 4-1: SimpleScalar baseline simulator models

Simulator	Description	Code Lines	Typical Speed
sim-safe	Simple functional simulator	320	6 MIPS
sim-fast	Speed-optimized functional simulator	780	7 MIPS
sim-profile	Dynamic program analyser	1,300	4 MIPS
sim-bpred	Branch predictor simulator	1,200	5 MIPS
sim-cache	Multilevel cache memory simulator	1,400	4 MIPS
sim-fuzz	Random instruction generator and tester	2,300	2 MIPS
sim-outorder	Detailed microarchitectural timing model	3,900	0.3 MIPS

Figure 4-1 illustrates the SimpleScalar infrastructure and its main components. The behaviour of the simulator depends on the processor model that is defined at three levels: ISA, ABI (Application Binary Interface) and microarchitecture [13].

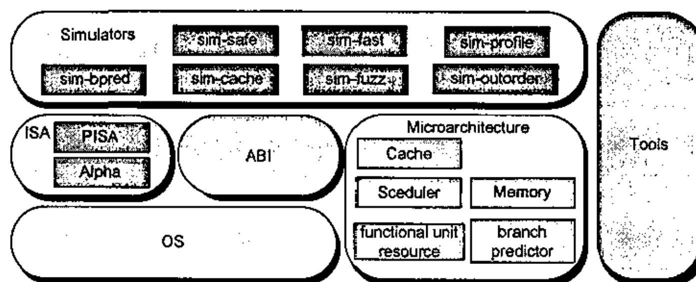


Figure 4-1: SimpleScalar Infrastructure

Only two ISA's are supported in the current release, the Portable Instruction Set Architecture (PISA) and the Alpha instruction set architecture. The instructions have a specific format that comprises the assembly format, binary opcode, register source and destinations, execution unit, instruction class and enum opcode that are assigned from the infrastructure [13]. Each instruction is associated with a semantic action statement that provides a comprehensive mechanism for describing how the instructions modify the state of the registers and memory. The OS handles only the trap instructions with the help of the system call simulation. The Application Binary Interface (ABI) establishes the communication between the simulated system and the external I/O. The instructions are loaded on a binary file format of the machine code after they are linked and relocated statically as no dynamic linking is supported. SimpleScalar uses the provided COFF binary file loader or the GNU's binary file descriptor library [13]. Since the SimpleScalar toolset is an execution-driven simulator, there is no need for instruction trace files as all the instructions are generated dynamically [1]. It models several microarchitectural components such as cache, memory, functional unit resource, scheduler and branch predictor. Its microarchitectural modelling ability can be extended easily due to its simple design that allows the addition of more components [13].

4.2.2 Customizing the SimpleScalar Toolset

For this research the SimpleScalar PISA instruction set was used. This is an extension of Hennessy and Patterson's DLX instruction set [20] that it also includes a number of instructions and addressing modes from the MIPS-IV [21] and RS/6000 (IBM pSeries). It utilizes a 64-bit instruction encoding to provide an easily extensible, research environment for instruction-set and system design. This extended encoding can support modification or addition of instructions, variation of the number of the program used registers etc [22]. The simulation tool utilised was the sim-fast that is a speed-optimized functional simulator that provides instruction accurate simulation but no timing. It executes all the instructions serially without assuming the existence of a cache. Based on this simulation tool, sim-vector was created that incorporates apart from the existing PISA, a file with the proposed vector ISA (`vector.def`). The `vector.def` file contains the definition of all the instruction extensions (scalar and vector) of the proposed coprocessor. The development of the coprocessor ISA and its introduction in the

`vector.def` file are described in more details in section 4.3.7. The specific two target workloads (G.729A and G.723.1) run on the model using execution-driven simulation. They use the statistical package which tracks updates to statistical counters and produces a detailed report. Sim-system is another tool based on the sim-fast that was created to model a shared memory multiprocessor environment. The sim-system simulator also called a PRAM model (Parallel RAM), is multithreaded and allows the execution of shared-memory applications. Sim-system was not utilised in this research as sim-vector provided the entire infrastructure in terms of single processor and the ability to add scalar-vector extensions. It has however been part of another closely linked research project in which the identified scalar-vector extensions were implemented in SystemC and attached to the vector unit of a high performance configurable extensible processor [23]. This research project and its results are detailed in Chapter 7.

4.3 Workload Optimization

4.3.1 Profiling

As mentioned previously, ITU-T provides reference C code for a number of speech coders. Every such reference implementation defines a set of universal, basic arithmetic operations (functions), essential for the implementation of speech coding algorithms. For the purpose of this research and in order to investigate the potential acceleration, the ITU G.729A reference code was profiled initially in native mode (Intel X86) in order to identify the computation workload distribution in these basic functions [24]. This was achieved by compiling the code with the compile flag **-pg** (for embedding profile instrumentation in the resulting binary) and running it with one of the ITU-T supplied test vectors, to produce a single profile data file. Subsequently, this was processed by the **gprof** Linux utility. Profiling revealed that the average relative amount of time spent outside the basic-op functions in reference code was 30.4% and 26.9% for the G.729A coder and decoder respectively as it shown in Table 4-2 [24]. The same profiling was also performed for the ITU G.723.1 reference code and the results are depicted also in Table 4-2.

Table 4-2: Relative amount of time spent outside the basic instructions

Algorithm	Relative CPU Time (%) in Native Mode
G.729A Coder	30.4
G.729A Decoder	26.9
G.723.1 Coder	31.3
G.723.1 Decoder	22.8

As general applicability and consistency of the profiling data were desirable, the workloads were profiled again in the SimpleScalar environment which is our simulation infrastructure. Table 4-3 depicts the highest percentage of the dynamic instruction count spent outside the basic operations of both the application codes, for encoding and decoding [25].

Table 4-3: Relative number of total instructions executed outside the DSP emulation instructions

Algorithm	Relative Instructions(%, simulated)
G.729A Coder	34.2
G.729A Decoder	37.2
G.723.1 Coder	34.5
G.723.1 Decoder	33.3

Even though two fundamentally different instruction set architectures and profiling collection/execution environment were used, both respective profiling metrics of the codecs were within 5% of one another. Therefore the experiments were continued by using the simulated infrastructure as the produced results are reasonably independent of the sampling issues of profiling in native mode and closer to real implementations of RISC/DSP processing kernels for multimedia applications [25]. The profiling results, as it was expected, revealed that the workloads spend a significant amount of time/instructions executing the basic emulation functions. Table 4-3 reveals that a 66.7% of the total machine instructions executed is inside the set of basic functions. A further, very important observation relates to parallelism exploitation within the right DSP loops utilising these basic operations. In general, visual inspection of the code suggests a significant number of the basic operations appear in data-parallel loops [24]. It was apparent that efficient implementation of the basic operations via a configurable microprocessor with a targeted, data-parallel architecture, that closely matches these basic

operations, could lead to high performance. These basic instructions are listed in the chart of Figure 4-2 along with the number of the executed machine instructions that they need. Therefore the creation of vector instructions was based primarily on the profiling information selecting the most machine instruction consuming.

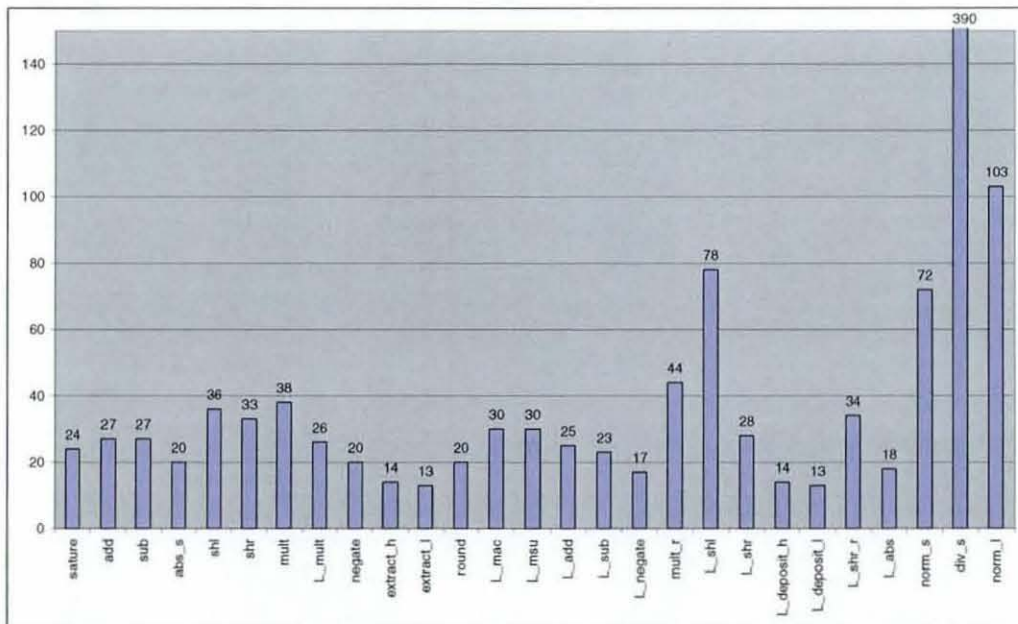


Figure 4-2: Machine instruction count for the BASOP.C functions

Additional acceleration of these computationally expensive operations can be achieved by taking advantage of the Data Level Parallelism (DLP) to create vector operations, based on the DSP emulation instructions, into a data parallel form. As it was explained in the previous chapter vector instructions are a simple yet, very powerful mechanism to significantly improve the performance of the system [26]. The unmodified speech coders G.729A and G.723.1 were profiled once more using the SimpleScalar toolset for all ITU-T test vectors. The results of the comprehensive profiling for both codecs are shown in Table 4-5 and Table 4-4 respectively.

Table 4-4: G.723.1 Unmodified Workloads Instruction Count

Workloads	Instruction Count	Frames
Encoder		
Dtx63.tin	10,159,684,865	864
Dtx53mix.tin (r53)	925,852,798	120
Dtx53mix.tin (mixed)	1,062,686,614	120
Decoder		
Dtx63.rco	680,066,056	864
Dtx53.rco	90,359,083	120
Dtxmix.rco	90,305,154	120
Dtx63e.tco	925,852,811	120
Dtx63b.tco	9,093,395	11

These results were used as a baseline during the research and optimization phases of the scalar and vector ISA in order to precisely quantify the benefit.

Table 4-5: G.729A Unmodified Workloads Instruction Count

Workloads	Instruction Count	Frames
Encoder		
Algthm	62,613,638	34
Fixed	213,961,855	119
Lsp	3,977,183,269	2231
Pitch	3,253,175,283	1834
Tame	230,917,008	127
Test	311,692,276	175
Speech	6,656,624,952	3749
Decoder		
Algthm	13,456,279	34
Fixed	45,865,491	119
Lsp	865,256,672	2231
Pitch	706,161,011	1834
Tame	49,456,050	127
Speech	1,440,402,972	3749
Erasure	114,722,597	299
Overflow	148,851,504	383
Parity	115,390,78	288

Table 4-6 depicts the top ten most computationally intensive functions of the G.729A speech coder. As it can be seen the most demanding function is the `cor_h_x` that computes the correlation of the input response with the target vector.

Table 4-6: Profiling the G.729A functions by using the speech workload

Function	No of call	Dynamic Instruction Count	DLP	Description
`Cor_h_x`	15,000	247,349,024	High	Compute correlation of target vector
`Syn_filt`	30,000	236,497,500	High	Linear Prediction synthesis filter
`D4i40_17_fast`	7,500	217,172,751	Low	Algebraic codebook with 4 nonzero pulses
`Pitch_ol_fast`	3,750	213,394,987	High	Compute the open pitch lag
`Autocorr`	3,750	203,658,564	High	Find autocorrelations of signal with windowing
`Lsp_pre_select`	7,500	199,979,638	High	First stage quantizer using LSP codebook
`Residu`	7,500	58,402,500	High	Compute the LPC residual
`Pit_pst_filt`	7,500	43,319,433	Low	Find Pitch period and perform Postfiltering
`Copy`	63,755	41,693,130	High	Copy input to output vector
`Agc`	7,500	25,141,988	High	Scale postfilter output by automatic control

The next function, `Syn_filt`, implements the synthesis filtering [27]. Visual inspection of these functions identified the amount of the Data Level Parallelism (DLP) that can be effectively exploited and this is also shown in Table 4-6. Table 4-7 shows the top ten most computationally intense functions of the G.723.1 for the 6.3kbits/s workload. In this case, the most demanding function is the `Find_Best` that performs the fixed codebook search for the high rate encoder [28]. It contains a significant DLP and thus has high vectorization potential. The next function in the list, `Find_Acbk`, computes the adaptive

codebook contribution in the closed-loop around the open-loop pitch lag. This function unfortunately does not posses sufficient DLP [28].

Table 4-7: Profiling the G.723.1 functions by using the 6.3kbits/s workload

Function	No of call	Dynamic Instruction Count	DLP	Description
`Find_Best`	4,408	1,370,009,644	High	Fixed Codebook Search
`Find_Acbk`	2,772	915,225,959	Low	Adaptive Codebook Calculation
`Estim_Pitch`	1,728	430,602,013	High	Open-loop pitch estimation
`Lsp_Svq`	926	141,876,220	Medium	Search for the LSP indices
`Comp_Lpc`	864	126,386,784	High	Computes the LPC filter coefficients
`Upd_Ring`	3,456	98,506,368	Medium	Update memory of the filters
`Sub_Ring`	2,772	78,871,716	Low	Computes the zero-input response and target speech vector
`Comp_Ir`	2,772	78,048,432	Low	Computes the combined impulse response from the filters
`Error_Wght`	864	66,604,896	Low	Implements the formant perceptual weighting filter
`Decod_Acbk`	6,228	31,267,516	High	Computes the adaptive codebook contribution

This functional profiling in conjunction with visual inspection indicates that a vector implementation of the basic operations can lead to a high performance processing platform for these workloads. This is the basic premise around which a vector ISA and microarchitecture have been defined in this work.

4.3.2 Vector ISA Development and Experimentation Methodology

This section describes the optimization methodology adopted for both ITU G.729A and G.723.1 reference codes on the vector coprocessor software model. The main steps of the software optimization process are depicted in Figure 4-3. The selection of the kernels for optimization was based primarily on the profiling information for both ITU speech coding algorithms and focused on the most time/instruction-critical functions.

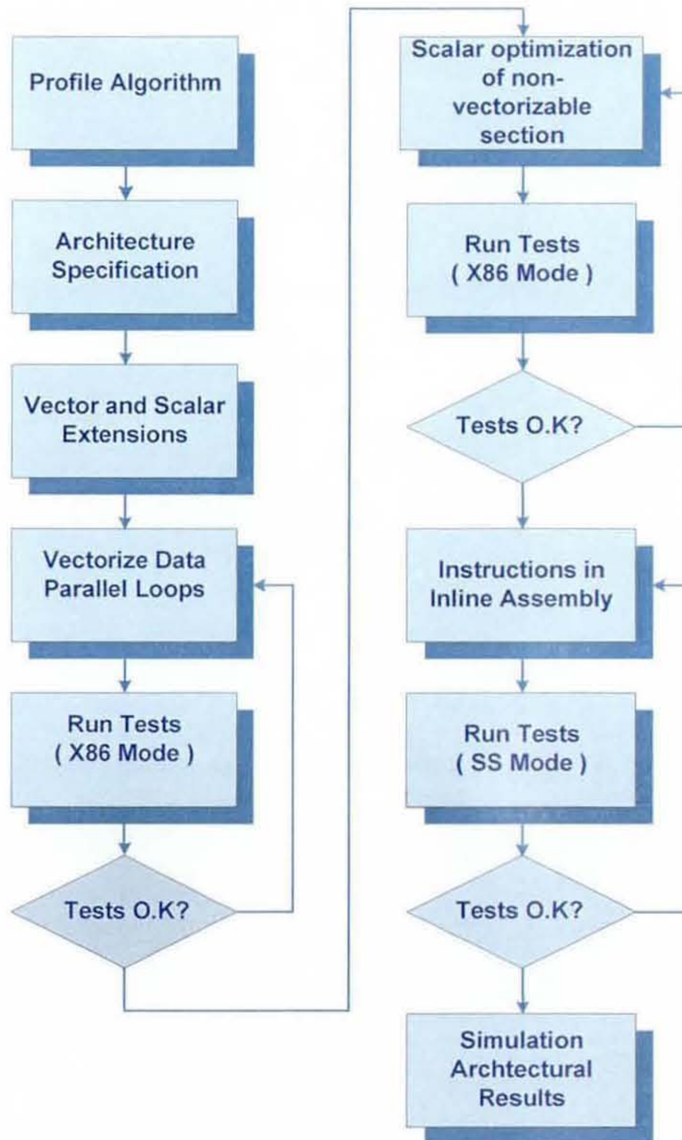


Figure 4-3: Experimentation Methodology

The architectural state of the proposed vector accelerator was defined in the architectural configuration file `vstate.h`. That file precisely describes the extended processor state, on top of the existing one (SimpleScalar specified processor state). Figure 4-4 illustrates the contents of the `vstate.h` that relates to the extended vector state (`vstateT` structure). The `#define` directives specify the number of the vector and scalar registers, the vector accumulators and the predication and overflow flag bits. As the coprocessor is uniquely parametric, parameter VLMAX is defined at the beginning of the file and

determines the maximum vector length of the vector components. In this particular case VLMAX is equal to 8. This means that a vector register will include 8x16-bit elements and a vector accumulator will have (8/2)x32-bit elements respectively. The structure `vstateT` encapsulates the total vector coprocessor state. It includes the definition of all the above mentioned programmer-visible registers as two dimensional arrays apart from the overflow flag that is a single dimension array.

```
//*****************
#define VLMAX 8
//*****************
 typedef signed short int VECTOR[VLMAX];

 #define VECTOR_REGS 16
 #define VACCUMULATORS 2
 #define PRED_REGS 1
 #define SCALAR_REGS 16

 typedef struct
 {
      // Vector length register
      int VLEN;
      // Vector register file
      signed short int VRF[VECTOR_REGS][VLMAX];
      // Vector accumulators
      signed int VACC[VACCUMULATORS][VLMAX/2];
      // Predicate registers
      unsigned short int PRED[PRED_REGS][VLMAX];
      // Scalar registers
      signed int SRF[SCALAR_REGS];
      // Vector overflow
      unsigned short int V16[VLMAX];
} vstateT;
```

Figure 4-4: The extended processor state as defined in the configuration file vstate.h

Subsequently, vector instruction extensions were developed that match the basic operations of the speech coding algorithms as these were proved to be the most critical. In order to check the coprocessor at the functionality level without the need to specify any underlying technology, C macros were created to represent the vector instruction extensions. This resulted in a new codebase which included these new instructions and thus can benefit from the power of the vector hardware. With this method, the instruction-accurate model of the coprocessor was verified with the help of the test vectors by mapping directly the output of the modelled coprocessor with that of the original scalar one.

In order to be able to run the algorithm in vector mode, it was essential to re-write the data level parallel loops of the code in vector assembly in such a manner that no semantic difference exists between the vectorized and the original code. At this point, it must make clear that the architecture specification and the ISA development are interlocked and both evolved during the vectorization of the workloads. The remaining (non-vectorizable part) of the code was also optimized by re-writing it in scalar assembly by using Scalar Instruction Set extensions. Initially, all the created instructions modelled in C and were included in the `x86_visa.h` header file located in the source directory of the ITU codecs. This step allowed for at-speed validation, in native mode, of the custom instructions with the original instructions replaced by the instruction extensions. An example of such instruction, as defined in the `x86_visa.h` file, is shown in Figure 4-5.

```
#ifdef X86
 //Vector register shift right
 /*--------------------*/
      #define vshri(vrf,amount)\
 /*--------------------*/\
 ({\
      extern vstateT vstate;\
      int index;\
      stats_start;\
      update_stats("vshri");\
      for (index = 0; index < vstate.VLEN ; index ++)\
      {\
            putv\
            if (vrf!=0)\

      vstate.VRF[vrf][index]=shr_simple(index,vstate.VRF[vrf]
                                        [index],(Word16)amount);\
            orv;\
      }\
      regv;\
      stats_end;\
 });
```

Figure 4-5: Example of a C macro Instruction Definition

The pre-processor directive `#ifdef x86` at the beginning of the instruction is used as a switch to enable or disable the C macros when executing Linux x86. The `vshri` instruction performs an arithmetic shift right of the source vector register, `vrf`, by as many positions as the variable amount defines. It calls the `shr_simple` function for each vector element, to perform the shift right operation. The result is stored in the destination register `vrf`. The shift is performed within a loop of `vlen` iterations, which is the

dynamic number of elements (16-bits each) that comprise the operand vector (`vrf`). This number is specified in the `vlen_r` register. A similar format was followed for all the instructions. The only main difference between the scalar and vector instructions is that the former does not contain a loop as the length of the scalar operands is constant while the length of the vector operands is parametric (run-time). After all the identified DLP loops were replaced with vector assembly the optimized workloads were validated by running the test vectors to ensure that there is no semantic difference between the vectorized and the original code. The remaining of the code was optimised by using scalar assembly and again it was verified by running the same ITU test vectors. When the optimization of the workloads was complete the vector and scalar instructions re-written in inline assembly and inserted in the SimpleScalar simulation infrastructure to extend its functionality and thus, the architectural simulation results. The steps that are mentioned above are described in more details in the following sections of this chapter.

4.3.3 Identification of Data Parallel Loops

As it already discussed, parts of both C reference codes had to be re-written in vector assembly in order to run efficiently on the vector accelerator. The replacement of scalar operations by vector extensions is called vectorization. Vectorization takes place in functions that can exhibit Data Level Parallelism (DLP). Such functions typically operate iteratively on blocks of data without the presence of data dependences (loop-carried dependences). By carefully examining the code it became apparent that the main area of interest is the loops as, in their overwhelming majority, perform DSP-type operations on arrays of data. These loops were therefore targeted and their bodies were replaced with vector operations semantically equivalent to the original code. Any mismatch in the output bitstreams between the original (ITU-T) and vectorized (as above) codes is attributed to loop-carried dependences which can't be eliminated [29]. In chapter 3 were described all the types of data dependences that can be detected in a program. In this case only the true dependences between statements in a loop were considered. More specifically, every loop was examined to determine whether a statement depends upon itself (loop-carried dependences) or if a statement that writes a memory location precedes a statement that uses that memory location as an input [29]. Figure 4-6 and Figure 4-7 illustrate the case of data dependent loop (non-vectorizable) and a data independent loop

(vectorizable) respectively. In Figure 4-6 the loop calculates the Line Spectral Pair (LSP) coefficients in G.729A encoder and shows interstatement (iteration-carried) dependences. This loop can't be vectorized. As it can be seen statements S5 and S9 depend upon input values that were created by previous execution (iteration) of S5 and S9 respectively.

```
   for (i = 0; i< NC; i++)
   {
S2    t0 = L_mult(a[i+1], 8192);    /*x=(a[i+1]+a[M-i])>>1*/
S3    t0 = L_mac(t0, a[M-i],8192); /*-> From Q11 to Q10*/
S4    x  = extract_h(t0);
S5    f1[i+1] = sub(x, f1[i]);    /*f1[i+1]=a[i+1]+a[M-i]-f1[i]*/
S6    t0 = L_mult(a[i+1], 8192); /* x = (a[i+1]-a[M-i]) >> 1 */
S7    t0 = L_msu(t0, a[M-i],8192); /*-> From Q11 to Q10 */
S8    x  = extract_h(t0);
S9    f2[i+1] = add(x, f2[i]);    /*f2[i+1]=a[i+1]-a[M-i]+f2[i]*/
   }
```

Figure 4-6: Example of a non-vectorizable loop as the statement S5 depends on a previous result of the S5 execution. The same dependency appears to the statement S9.

Figure 4-7 presents a loop that subtracts the unquantized LSP frequencies for the current frame in order to compute the VQ weighting vectors. It selects the frequencies that are closer in value with each other in order to produce weights of greater precision. As shown, this loop is vectorizable as both statements (S2 and S3) are independent from previous results of their execution (producer/consumer iteration indexes are linear combination of one another and independent). The inputs of these statements are arrays of the currents frequencies that can be loaded from assigned pointers to the vector registers.

```
   for ( i = 1 ; i < LpcOrder-1 ; i ++ )
   {
S2    Tmp0 = sub( CurrLsp[i+1], CurrLsp[i] ) ;
S3    Tmp1 = sub( CurrLsp[i], CurrLsp[i-1] ) ;
S4     if ( Tmp0 > Tmp1 )
S5          Wvect[i] = Tmp1 ;
S6     else
S7          Wvect[i] = Tmp0 ;
   }
```

Figure 4-7: Example of a vectorizable loop with statements S2 and S3 being independent from previous results of their execution.

The same methodology was followed for all the loops in both C reference codes for both encoder and decoder, whenever iteration-carried data dependences didn't arise between

loop statements, loops were re-written in vector assembly as described in the following section.

4.3.4 Implementation of vector loop using custom ISA

Figure 4-8 shows a loop that quantizes the difference between the computed and predicted coefficients at the first-stage vector quantizer in the LP analysis of the G.729A encoder. This vectorizable loop has M iterations (value is specified at compile time) that performs subtraction (`sub`) of two arrays, multiply the subtraction result with itself and adds the product to the accumulator (`L_mac`), for the entire current frame M. As it can be seen these two operations are data independent as the iterated statements are not using values computed in some previous iterations. Therefore they can safely be replaced and directly converted to vector form. The pre-processor directive `#ifded ORIGINAL` selects the conditional compilation of the code to run this non-optimized part when the original mode is selected in the `compile.h` header file.

```
/*****************LOOP1******************/
 #ifdef ORIGINAL
   for ( j = 0 ; j < M ; j++ )
    {
      tmp = sub(rbuf[j], lspcb1[i][j]);
      L_tmp = L_mac( L_tmp, tmp, tmp );
    }
```

Figure 4-8: Example of loop with DLP within the original C code

Figure 4-9 depicts the first part of the transformed loop with the introduction of vector assembly. Having identified that the loop is vectorizable, it is necessary to identify the inputs and outputs of the loop that have to be loaded or stored in vectors. By associating these I/O vectors with 16-bit or 32-bit pointers, this allows the data to be represented using the 16-bit elements of the vector registers or the 32-bit elements of the vector accumulators respectively. The newly created pointers point to the first values in both data arrays (inputs). All the intermediate values are stored temporarily into the vector registers or accumulators, depending on the instruction. When the pointers are set the vector length register (`vlen_r`) needs to be loaded with the maximum vector length (VLMAX). The last instruction `vsplatacci(...)` in this code snippet loads the value zero to the vector accumulator zero in order to clear it before any calculations take place.

```
#else
    {
      //Set Pointers
      signed short int *from1=rbuf;
      signed short int *from2=&lspcb1[i][0];

      //Load VLMAX into vlen_r register
      ldvlen_r(VLMAX);
      //Clear accumulator
      vsplatacci(0,0);
```

Figure 4-9: Assign pointers and load the vlen_r register

Figure 4-10 illustrates the main vectorized loop (modulus part). This is true while executing the loop since the loop only deals with whole vector lengths. In the figure the original loop range is decreased by dividing the initial iteration number by the maximum number of vector elements available, VLMAX. Doing this, in combination with incrementing the vector pointers, `from1` and `from2` by `VLMAX`, allows for each iteration of the loop the pointers to point to new set of vector data. This part of the code will be performed as many times as the quotient of this division.

```
          //Modulus Part
          for (i=0; i < M/VLMAX; i++)
           {
            //Load vector register from rbuf
S1          vldw(1, from1);
            //Load vector register from &lspcb1
S2          vldw(2,from2);
            //Perform subtraction to vr1, vr2
S3          vitu_sub_r(3,1,2);
            //Multiply even word and add to VACC0
S4          vmace(0,3,3);
            //Multiply odd word and add to VACC0
S5          vmaco(0,3,3);
            //Increase address pointers
S6          from1 += VLMAX;
S7          from2 += VLMAX;
          }
```

Figure 4-10: Main vector loop

Within this loop five custom vector instructions are executed. The first two (statements S1, S2) are vector loads which load the data from the pointer addresses `from1` and `from2` and deposit them in the vector registers 1 and 2 respectively. The next three instructions (S3, S4, S5) perform the main functionality of the loop, that is vector subtraction and multiply-accumulate operations. First the subtraction is executed on

vector source registers 1 and 2 and the result is stored into vector register 3. The multiply-accumulate calculation is performed as a pair of instructions, for the even and odd elements respectively. Each of these instructions multiplies the register 3 with itself and adds the product to the corresponding even or odd elements of accumulator 0. The last two instructions increment the pointers by VLMAX to prepare the data for the next loop iteration.

Since the original loop parameter, in this case M, may not be exactly divisible by `VLMAX` a remainder section (loop strip mining code) is required to ensure that all the original data is processed. Strip mining is the process of running the loop with a number of iterations that does not divide exactly the VLMAX architecture constant. This code is only executed if there is a remainder from the modulus operation, `M%VLMAX`. If this is the case, the `vlen` register (dynamic vector length) is loaded with the new vector length `M%VLMAX` in order to indicate in which elements the vector instructions will be performed during the strip mined section. This loop is executed only once, for the specified vector elements and thus, only a subset of the vector datapath is achieved during this section.

```
//Remainder Part
      if (M % VLMAX)
      {
S1      ldvlen_r(M % VLMAX);
        //Load vector register from rbuf
S2      vldw(1, from1);
        //Load vector register from &lspcb1
S3      vldw(2,from2);
        //Perform subtraction to vr1, vr2
S4      vitu_sub_r(3,1,2);
        //Multiply even word and add to VACC0
S5      vmace(0,3,3);
        //Multiply odd word and add to VACC0
S6      vmaco(0,3,3);
      }
S7    ldvlen_r(VLMAX);
      // Do ADD reduction of VACC0
S8    vaccaddreduce(0);
      //Store accumulator value in element 0 to L_tmp
S9    vstacc(0,0,&L_tmp);
      }
#endif
```

Figure 4-11: Strip mining loop

The last section of the code snippet in Figure 4-11 restores the dynamic vector length register to the maximum vector length for the vector accumulator to perform an add-reduce operation in all its elements and produce a final 32-bits scalar result. This result is deposited in the lowest element (element 0) of accumulator 0 and it is stored into memory at the pointer's address `L_tmp` with statement S9. In the vectorized code the arithmetic instructions calculating the displacement from the index base are reduced by (VLMAX+1) times as the number of iterations is divided with VLMAX plus the modulus calculation for loop strip mining.

4.3.5 Scalar Optimization

All the data-parallel loops constructs that didn't exhibit any data (iteration-carried) dependences were re-written in vector assembly, using vector instruction extensions and the techniques discussed previously. The remaining of the code that comprises non-vectorizable loops and parts that contain BASOP instructions was optimized through the addition of custom scalar instructions. Figure 4-12 depicts an example of scalar assembly that replaces part of the original code. This loop transforms back the LPC from the LSP coefficients. As it can been seen this loop presents data dependency as both statements of the original code depend upon input values that were created by previous execution/iteration (`f1[i-1]` and `f2[i-1]`). Therefore this loop is not vectorizable and can be only optimized by replacing the BASOP operations with scalar instructions. The pre-processor directive `#ifdef METHOD2` is used at compile time to allow this scalar-optimized part of the reference code to run and it is activated by the `METHOD2` switch in the `compile.h` header file. In a similar manner with the vector-optimized loops, the operands are loaded to the coprocessor scalar registers. The difference is that these registers are scalar and the loop iteration is the same as the original one. The next instructions perform long addition (`L_add`) and long subtraction (`L_sub`) to the scalar registers and results are stored back to the memory.

```
for (i = 5; i > 0; i--)
 {
/******************** METHOD2 ********************/
#ifdef METHOD2

      //Load variable f1[i] in register[1]
      m2sld32(1,f1[i]);
      //Load variable f1[i-1] in register[2]
      m2sld32(2,f1[i-1]);
      //Load variable f2[i] in register[3]
      m2sld32(3,f2[i]);
      //Load variable f2[i-1] in register[4]
      m2sld32(4,f2[i-1]);
      //Perform L_add
      m2sladd(1,1,2);
      //Perform L_sub
      m2slsub(3,3,4);
      //Store to f1[i]
      m2sst32(1,f1[i]);
      //Store to f2[i]
      m2sst32(3,f2[i]);

#else  //ORIGINAL CODE

      f1[i] = L_add(f1[i], f1[i-1]);     /* f1[i] += f1[i-1]; */
      f2[i] = L_sub(f2[i], f2[i-1]);     /* f2[i] -= f2[i-1]; */

#endif
```

Figure 4-12: Scalar optimization example

4.3.6 Validation Tests

Every time a vector or scalar assembly instruction was added in one of the C reference codes, tests were run, using the test vectors provided by the ITU-T. This was to verify the full algorithmic equivalence between the optimized and the original (reference) codes. The test vectors employed for both algorithms are listed in Table 4-8 below.

Table 4-8: G729 Encoder Test Vectors

Input vector	ITU Reference output	Description
Algthm.in	Algthm.bit	Conditional parts of the algorithm
Fixed.in	Fixed.bit	Fixed codebook search
Lsp.in	Lsp.bit	Lsp quantization
Pitch.in	Pitch.bit	Pitch search
Speech.in	Speech.bit	Generic speech file

It is important to note that these vectors are not exhaustive and thus can only be part of a more comprehensive validation suite.

Table 4-9: G729 Decoder Test Vectors

Input vector	ITU Reference output	Description
Algthm.bit	Algthm.pst	Conditional parts of the algorithm
Fixed.bit	Fixed.pst	Fixed codebook search
Lsp.bit	Lsp.pst	Lsp quantization
Pitch.bit	Pitch.pst	Pitch search
Speech.bit	Speech.pst	Generic speech file
Tame.bit	Tame.pst	Taming procedure
Erasure.bit	Erasure.pst	Frame erasure recovery
Overflow.bit	Overflow.pst	Overflow detection in synthesizer
Parity.bit	Parity.pst	Parity test

Passing these vectors can be considered a minimum requirement, and is not a guarantee that the implementation is correct for every possible input signal.

Table 4-10: G.723.1 Encoder and Decoder Test Vectors

Input vector	ITU Reference output	Description
Encoder		
dtx63.tin	dtx63.rco	Encoder input / 6.3 rate
dtx53mix.tin	dtx53.rco	Encoder input / 5.3 and mixed rate
dtx53mix.tin / dtxmix.rat	dtxmix.rco	Encoder rate input
Decoder		
dtx63.rco	dtx63.rou	Decoder input / rate 6.3
dtx53.rco	dtx53.rou	Decoder input / rate 5.3
dtxmix.rco	dtxmix.rou	Decoder input / mixed rate
dtx63e.tco/ dtx63e.crc	dtx63e.rou	Decoder input / rate 6.3 with Cyclic Redundancy Check (CRC) input
dtx63b.tco	dtx63b.rou	Decoder input / rate 6.3

For the purpose of this research these ITU supplied test vectors were used to ensure compliance of the reference speech coders throughout the optimization phase. The

compiler used to compile the vectorized reference code was gcc 3.3.2 (linux x86) [30] and the gcc 2.7.3 [30] cross-compiler for the SimpleScalar ISA.

4.3.7 The extended ISA (Scalar and Vector Extensions)

This section, describes the modifications that took place in the core SimpleScalar toolset in order to emulate the coprocessor architecture under study. The sim-vector tool that was used is an extended simulator based on the sim-fast simulator but modified with added state (coprocessor scalar and vector state) and instructions (coprocessor scalar and vector instructions). This code includes the extra processor state and the instructions that operate on that extended state. The extended state specifies the additional registers on top of the existing architectural state (SimpleScalar processor state). The vector.def file includes the definition of all the existing instructions of the SimpleScalar along with the extended instruction set architecture. The vector.def file contains the PISA.def which includes C macro implementations of all the basecase SimpleScalar instructions.

```
Vector.def example

switch ((inst.b>> CATEGORY_LSB) & CATEGORY_MASK)\
 {\
  /**************************/\
   case 2: /* CATEGORY 2 */\
  /**************************/\
   switch (OPCODE)\
    {\
     case 1:\
    {\
     switch ((inst.b >> EXT_OPCODE_LSB) & EXT_OPCODE_MASK)\
      {\
       case 3:\
        {\
         /* VSHRI */\
         extern vstateT vstate;\
         enum md_fault_type _fault;\
         int index;\
         Word16 amount;\
         amount=GPR(IMM9_ADDR);/*(Word16)IMM8;*/\
          for (index=0; index< vstate.VLEN; index++)\
           {\
            if (RD_ADDR !=0 )\
             vstate.VRF[RD_ADDR][index]=my_shr_simple
            (index,vstate.VRF[RS1_ADDR][index],(Word16)amount);\
           }\
      break;
```

Figure 4-13: Instruction Definition in Vector.def

The vector.def file contains the opcode definitions of the whole extended instruction set. Extended opcodes are split into 3 parts; the opcode bits 20-24, category bits 25-28 and the extended opcode bits 29-31. A typical opcode is implemented with 3 levels of switch statements. The first level is the category switch, the second level the opcode switch and the final level is the extended opcode switch. Figure 4-13 shows the C description of the vector shift right coprocessor command and, as it can be seen, is similar to the C macro definition of the instruction in Figure 4-5. The main difference is how the source/destination registers are decoded. They have been extracted from the instruction opcode in an earlier stage. Inside the loop the vector instruction is performed and every loop iteration represents a vector datapath lane. This replicates the functionality of the vector processor. When the extended SimpleScalar toolset is running, the extended vector instruction count is added to the default instruction count to derive precise execution statistics for the whole (base and extended) processor architecture.

4.3.8 Inline Assembly

The C representation of the extended instructions (macro-based) adds a lot of time-overhead as every opcode corresponds to a number of instructions and is thus used only to model the execution of these instructions. Therefore, in order to derive a final optimized implementation, the extended instructions were inserted with inline assembly.

```
      #ifdef SS
            // simplescalar
            #define vshri(vrf,amount) \
            ({\
      asm volatile ("addu $10,%0,$0" : :"r"(amount):"$10");\
                  asm volatile (".word 0x00010000");\
                  asm volatile (".word \
                  3 << 29 |          /* EXT_OPCODE */\
            2 << 25 |          /* CATEGORY */\
            1 << 20 |          /* OPCODE */\
            "#vrf"<< 15 |      /* VRD = VRF */\
            "#vrf"<< 10 |      /* VRS1=10 */\
            10 << 5");         /* RS2 = HOST REG */ \
            });
      #else
            // Sparc
      #endif
#endif
```

Figure 4-14: Inline Assembly Instruction Definition

With this method, every scalar/vector opcode corresponds to one instruction only and the added instructions can run in the SimpleScalar mode to produce reliable statistics. Figure 4-14 illustrates an example of inline assembly for the `vshri` vector instruction; that its C macro was showed in Figure 4-5. The asm volatile statement is divided in three parts. The first part is the code section where the first (source) operand (%0) is added to source register 0 and stored into the target register 10 ($10) [30]. Since there are not output operands two consecutives colons are added on the place where the output operands would go. The "r" (amount) signifies that is the other input register operand. The "r" is a constraint string which indicates that the following C variable (amount) is placed in a general register. The last part of the asm instruction, the clobber list, is utilised to inform the compiler about which register is clobbered (modified) by the assembly code. In this example "10" indicates to the compiler that register 10 has been modified by the inline assembly. The next two assembly lines comprise a 64-bit opcode that will be dynamically decoded by sim-vector during run time. The first part of the opcode which is 32-bits (0x00010000) represents the `nop` instruction annotation 1 (flag) whereas the second part builds the remaining 32-bits (word) which is the actual vector instruction [30]. This word is the binary pattern of the instruction set extensions. The compiler composes the opcode binary according to the above inline assembly statements. During runtime SimpleScalar encounters the `nop` opcode and checks the binary pattern. If it is an extended instruction, it performs the transformation on the processor state as specified by the extended opcode.

4.4 Architectural Results

As it was described in the previous sections, both workloads were optimized with the development of vector and scalar extended ISAs. Throughout the experimentation phase, the modified workloads were validated by using the ITU-T test bitstreams to ensure compliance with the reference speech coders. In order to study the optimization benefits, simulations were run for all ITU-T input vectors and for vector lengths of up to 128 16-bit elements. During compile time, the user can select which mode the coprocessor will run. A special file (`compile.h`) contains all necessary switches for compilation in order to be able to select the mode that the code will run. By selecting x86 (native mode) or SS (SimpleScalar mode) the compiler is using the C-macros (Figure 4-5) or the inline assembly implementation (Figure 4-14) of the extended instructions respectively. The

ORIGINAL switch selects if the code will run in original or vector mode while the METHOD2 adds the scalar features. The results are segmented in two major groupings with the first group showing the induced performance of the vector ISA only. The second group reflects the performance of the full optimizations and exposes the additional performance benefits of the scalar ISA. Figure 4-15 and Figure 4-16 show the results of the extended, architecture-level performance simulation of the G.729A encoder and decoder respectively for vector optimizations only. The performance metric used is the relative dynamic instruction count which in both cases is approximately 59.1% and 60.7% respectively at a vector length of sixteen 16-bit elements.

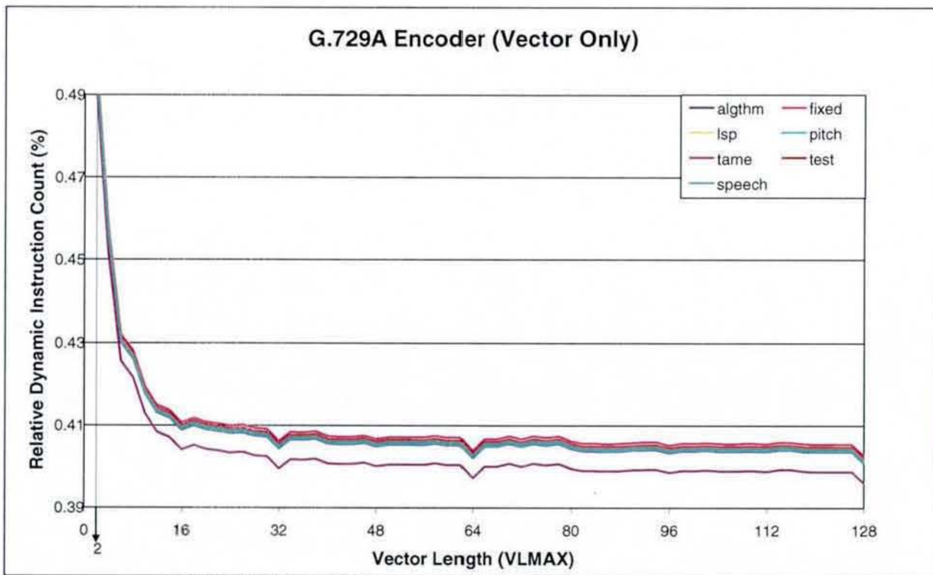


Figure 4-15: G.729A Encoder (Vector Only) Results

This essentially means that the vectorized G.729A encoder executes 59.1% fewer instructions compared to the reference C implementation when the vector ISA comes in effect, for a VLMAX of 16. In the case of the G.729A decoder, this figure is 60.7%.

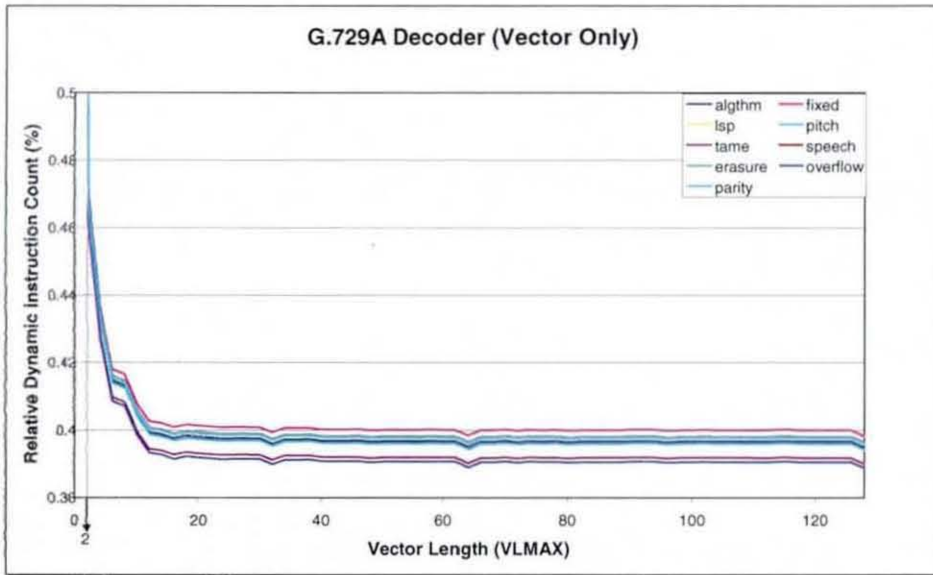


Figure 4-16: G.729A Decoder (Vector Only) Results

The slope of the graphs clearly demonstrates that the most significant performance benefits are realized at shorter vector lengths, in the range of 2 to 16 16-bit elements, while no further significant reduction is measured beyond that configuration. This observation has the benefit of restricting the microarchitecture design space to shorter vector lengths as configurations with vector lengths greater than 16-bit elements are in practice unrealistic, due to the large silicon overhead incurred by such wide datapaths and the need for very long cache fill bursts [31].

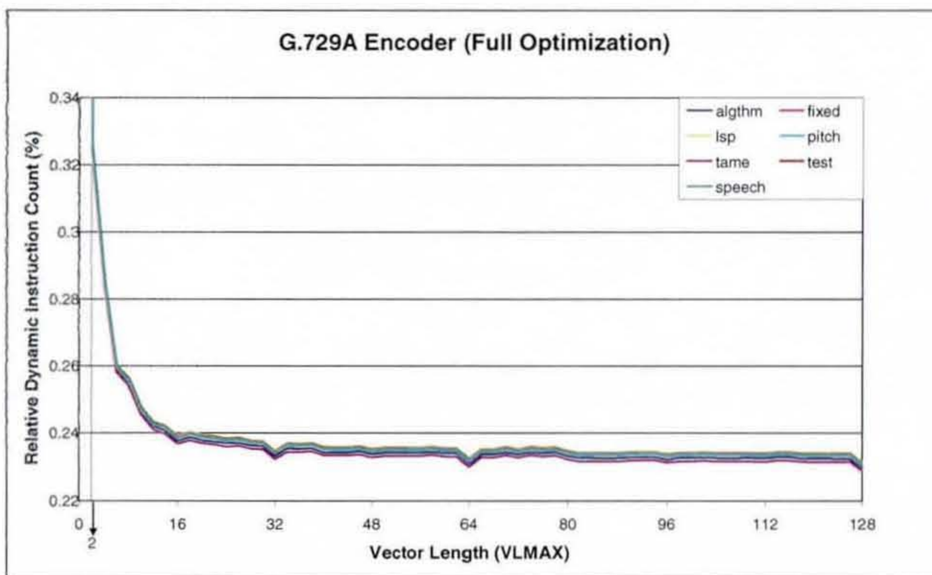


Figure 4-17: G.729A Encoder (Full Optimization) Results

Figure 4-17 and Figure 4-18 depict the relative algorithmic complexity of the G.729A encoder and decoder obtained with all the optimizations.

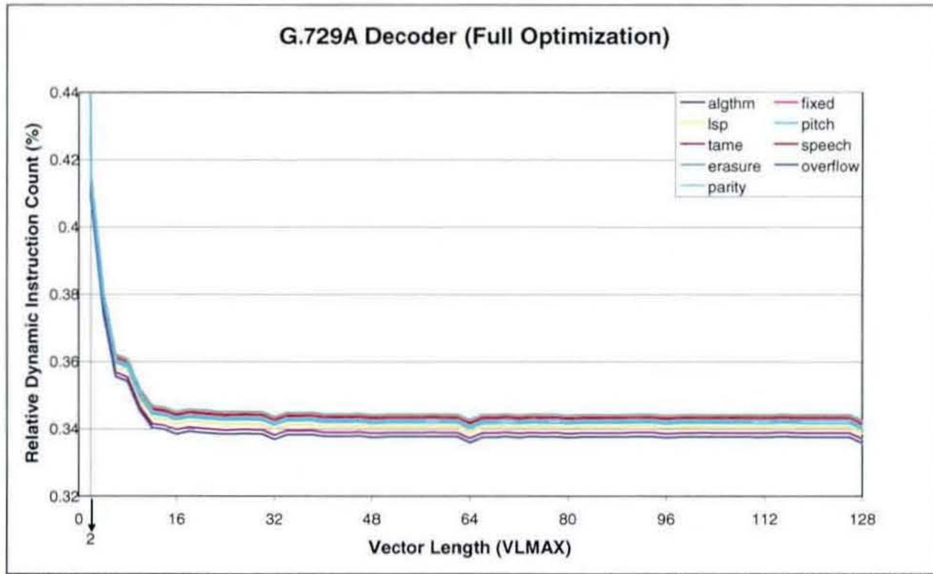


Figure 4-18: G.729A Decoder (Full Optimization) Results

In this particular case, both the data-parallel as well the non-vectorizable sections of the code were optimized. The achieved performance metric improvement for the encoder and decoder is 76.2% and 65.9% respectively for vector length of 16 (256 bits) and no further improvement appears for larger vector lengths. It is clear from the results that there is significant improvement in the dynamic instruction compared to the original execution. These data indicate that both speech coding standards benefit substantially from combined, scalar and vector accelerator.

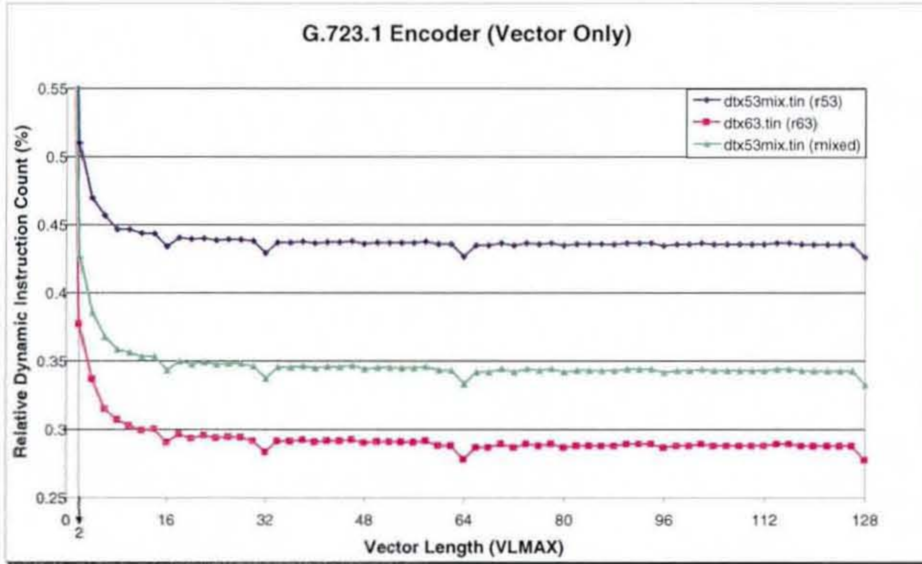


Figure 4-19: G.723.1 Encoder Vector Optimization Results

Figure 4-19 and Figure 4-20 illustrate the relative dynamic instruction count reduction of the G.723.1 encoder and decoder respectively for the vector optimization only.

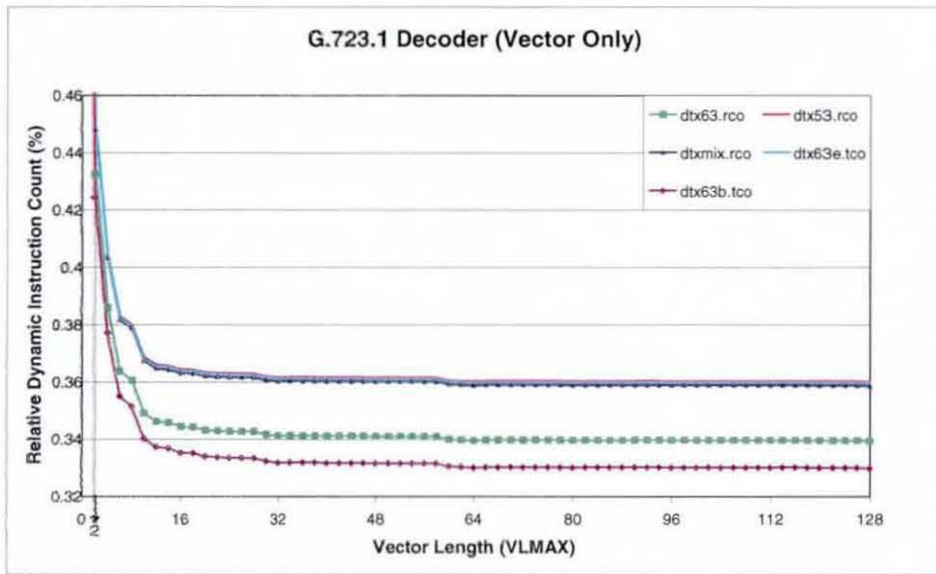


Figure 4-20: G.723.1 Decoder Vector Optimization Results

The performance metric for the encoder and decoder is approximately 70% and 67% respectively at a vector length of 16 16-bit elements. This maximum improvement appears at a vector length of 16 (256 bits) and no significant improvement emerges beyond that. Performance saturation clearly indicates that wider DLP configurations are

not needed and that most of the inherent DLP of the algorithms can be exploited by a 256-bit wide vector coprocessor.

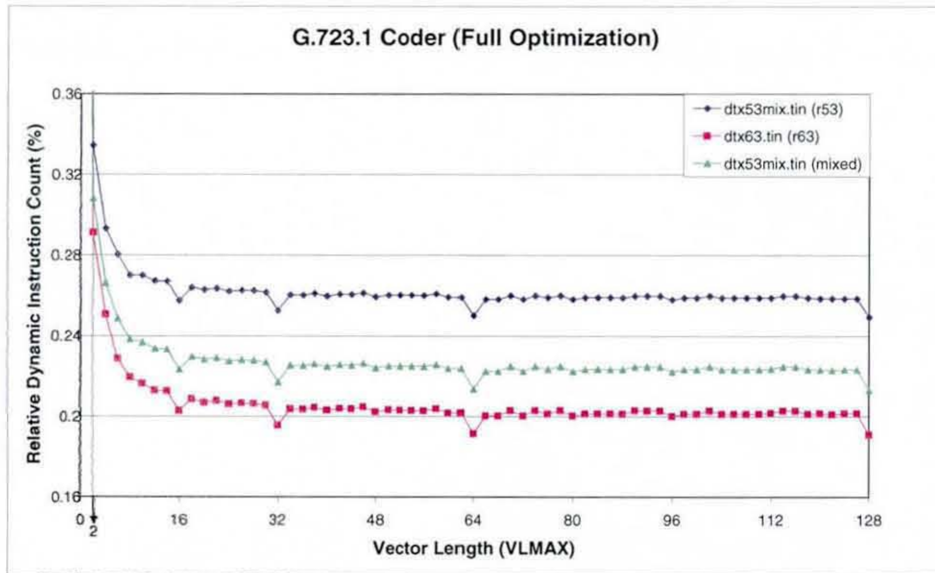


Figure 4-21: G.723.1 Encoder Full Optimization Results

Figure 4-21 and Figure 4-22 depict the performance metric of the G.723.1 encoder and decoder with full scalar and vector optimizations. In this case, the dynamic instruction count is reduced approximately to 79.7% and 73.6% at a vector length of 16 (256 bits).

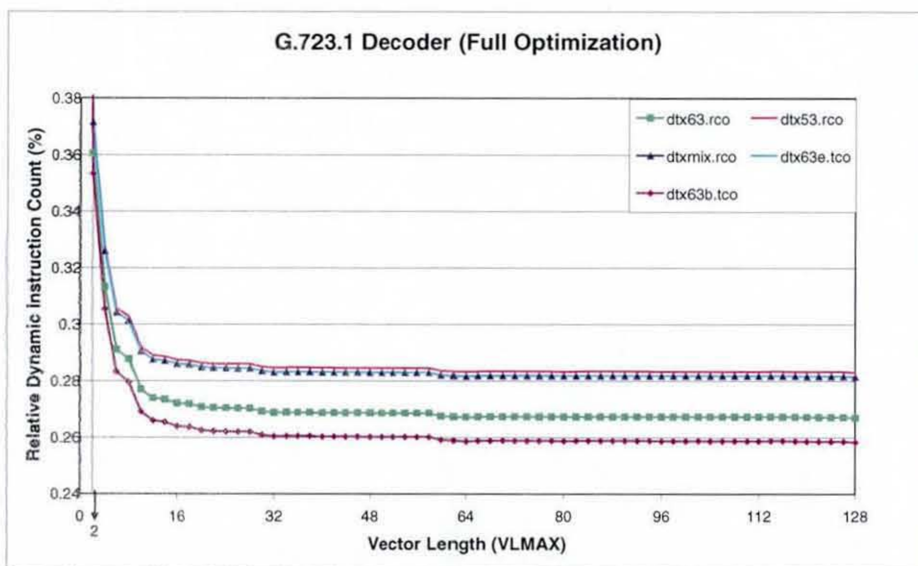


Figure 4-22: G.723.1 Decoder Full Optimization Results

This additional decrease in the dynamic instruction count of both speech codecs shows considerable improvement with the introduction of the scalar instructions. It is clear that the combination of scalar and vector optimized code, via the two proposed extended ISAs yields better performance metrics and for this reason the design implementation includes both coprocessors. The next set of graphs (Figure 4-23 up to Figure 4-34) illustrates the performance improvement of the most compute-intensive functions as they appear in Table 4-6 and Table 4-7, for both speech codecs. These results can be used to see the specific sections of the speech codec which have been improved. Figure 4-23 shows the results for the G.729A encoder function `Cor_h_x` under full-optimization. This function computes the correlation of the input response with the target vector in the algebraic codebook (fixed codebook) search procedure [27]. The fractional performance improvement of the optimized codebook search at vector length of 16 16-bit elements (256 bits) is 75.5% and it reaches 78.5% at vector length of 2048 bit over the range of reference input bitstreams.

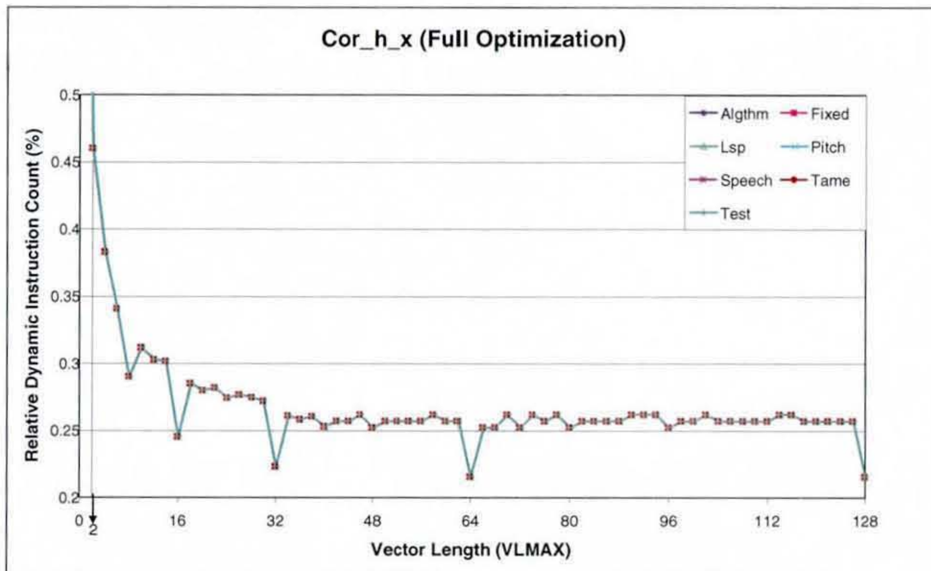


Figure 4-23: Cor_h_x (Full Optimization) Results

Figure 4-24 presents the performance metric (relative instruction count) of the G.729A function `Syn_filt`. This function implements the 10th order Linear Prediction (LP) synthesis filter (1/A(z)) [27]. The performance improvement of the synthesis filter at vector length of sixteen is 73.5% over the range of the reference input bitstreams. As it can be seen no further improvement is evident beyond this vector length. This is

explained from the fact that the number of iterations for the internal loop of this function is 10.

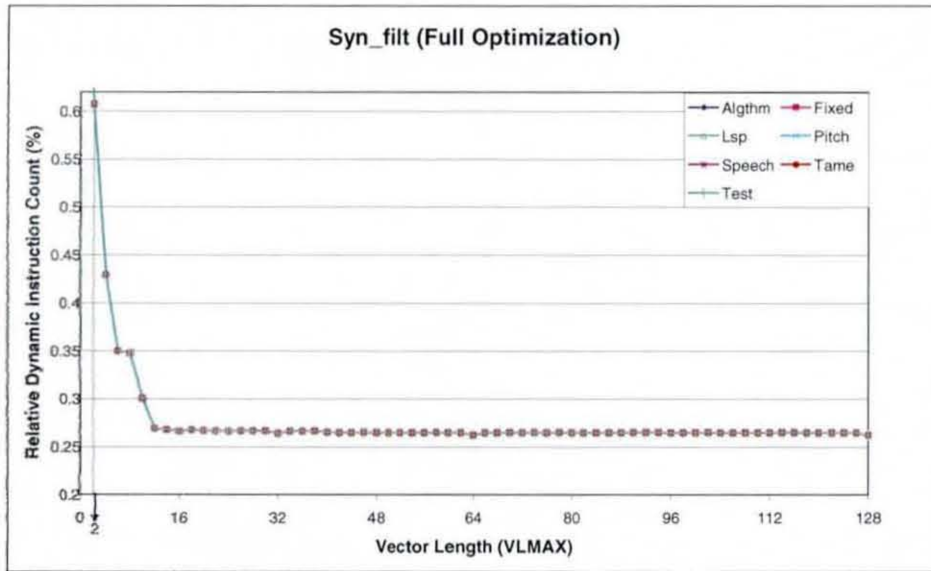


Figure 4-24: Syn_filt (Full Optimization) Results

Figure 4-25 depicts the relative instruction count of the G.729A function `Pitch_ol_fast` under full-optimizations. This function estimates the open-loop pitch delay based on the perceptually weighted speech signal. This open-loop delay is used as an indication from the closed-loop analysis to find the adaptive-codebook delay and gain [27].

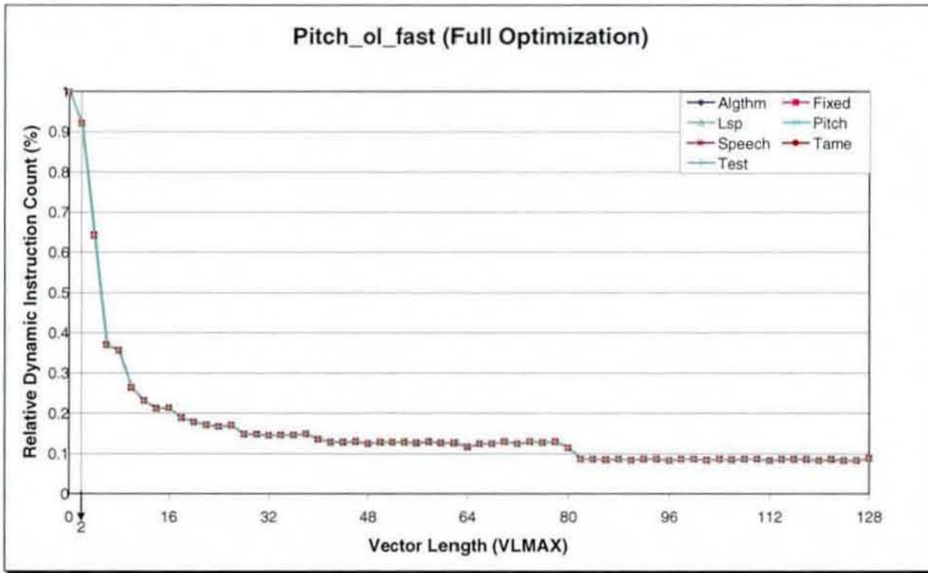


Figure 4-25: Pitch_ol_fast (Full Optimization) Results

In this case the performance-metric reduction (relative dynamic instruction count) is approximately 78.7% at a vector length of sixteen 16-bit elements. The next graph in Figure 4-26 shows the relative performance improvement of the G.729A function `Residu`. This function computes the LP residual signal by filtering the input speech through the LP synthesis filter. The LP residual signal is used to find the target vector for the adaptive-codebook search [27].

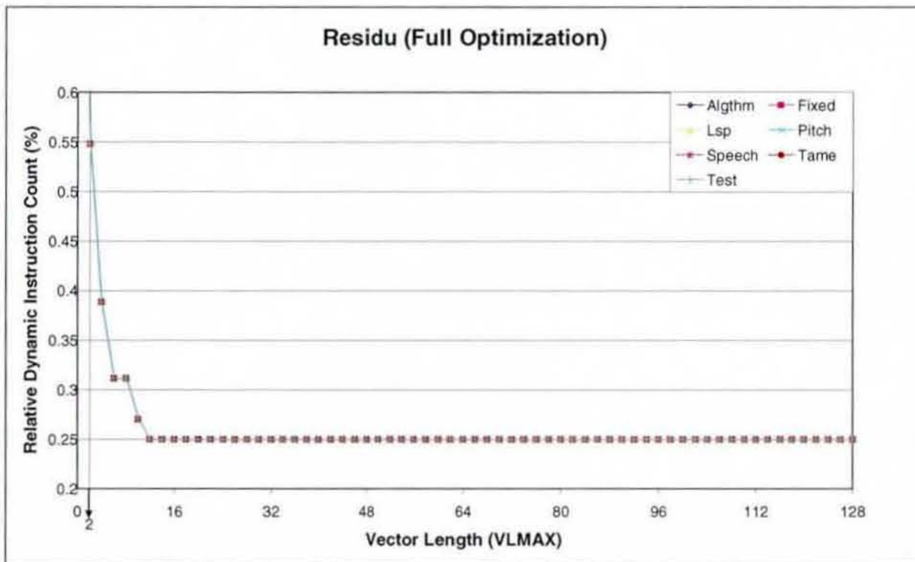


Figure 4-26: Residu (Full Optimization) Results

In this case the achieved instruction count reduction for this function is of the order 75.1%. No further improvement is evident beyond this vector length as the number of iterations for the internal loop of this function is 10. Figure 4-27 depicts the performance improvement for the G.729A function `Autocorr`. This function computes the autocorrelation of the signal with a 30ms asymmetric window in order to perform linear prediction (short-term) analysis. Later the autocorrelation coefficients of the windowed speech are computed and converted to the LP coefficients using the Levinson algorithm [27].

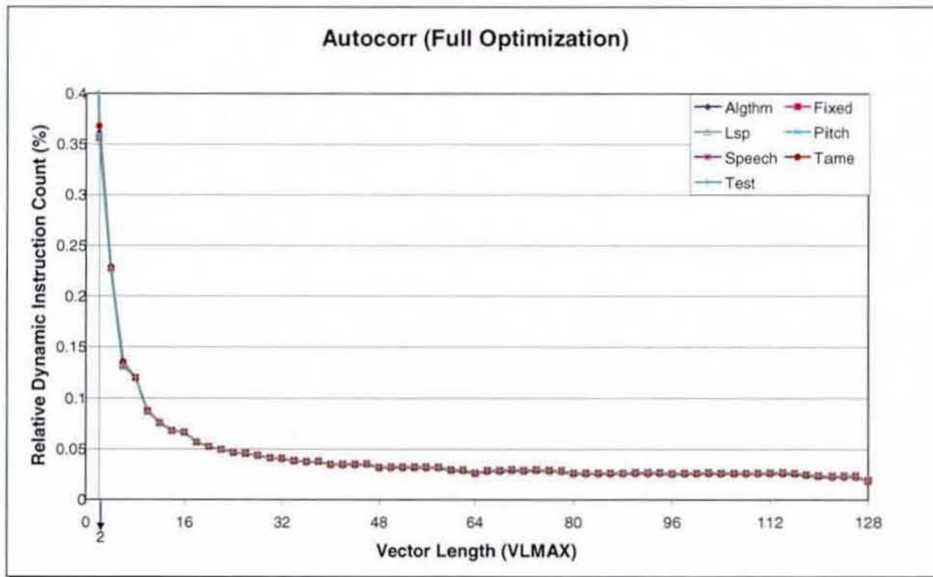


Figure 4-27: Autocorr (Full Optimization) Results

This function demonstrates excellent performance stability and experiences a dynamic instruction count reduction of approximately 93.4% at a vector length sixteen. The `Autocorr` function contains a large number of data-parallel loops that were vectorized successfully.

Figure 4-28 shows the relative instruction count of the G.729A function `Lsp_pre_select`. This function implements the first stage quantizer that quantizes the difference between the computed and predicted LSF coefficients of the current frame. This quantizer is a 10-dimensional Vector Quantizer (VQ) that uses a codebook with 128 entries (7 bits) [27].

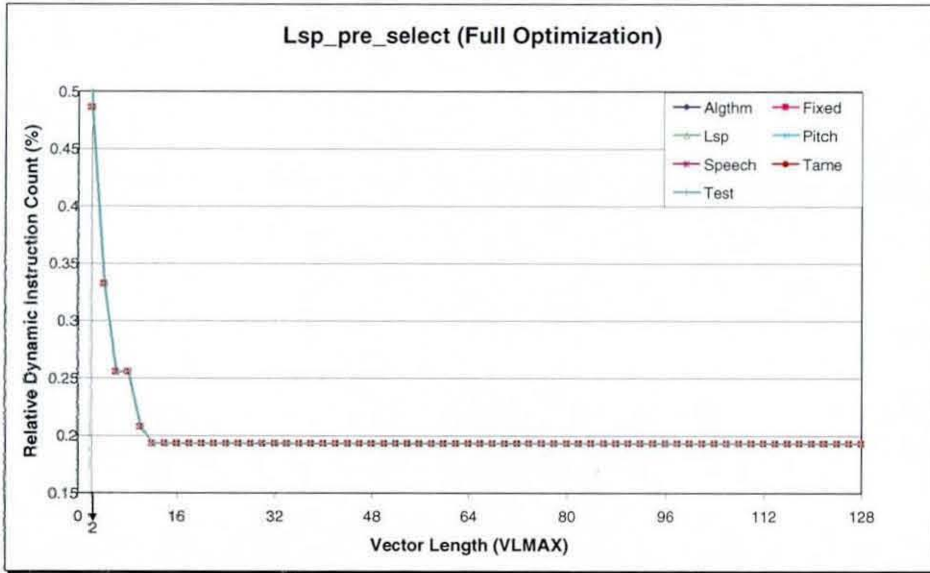


Figure 4-28: Lsp_pre_select (Full Optimization) Results

The improvement in performance under full-optimizations is of the order of 80.7%. After this vector length an expected performance saturation is observed since the number of iterations for the internal loop of this function is 10.

Figure 4-29 represents the results of the G.729A decoder function `Agc`. This function implements the Automatic Gain Control (AGC) procedure that takes the output of the adaptive post filter and scales it to match the energy of the reconstructed signal [27].

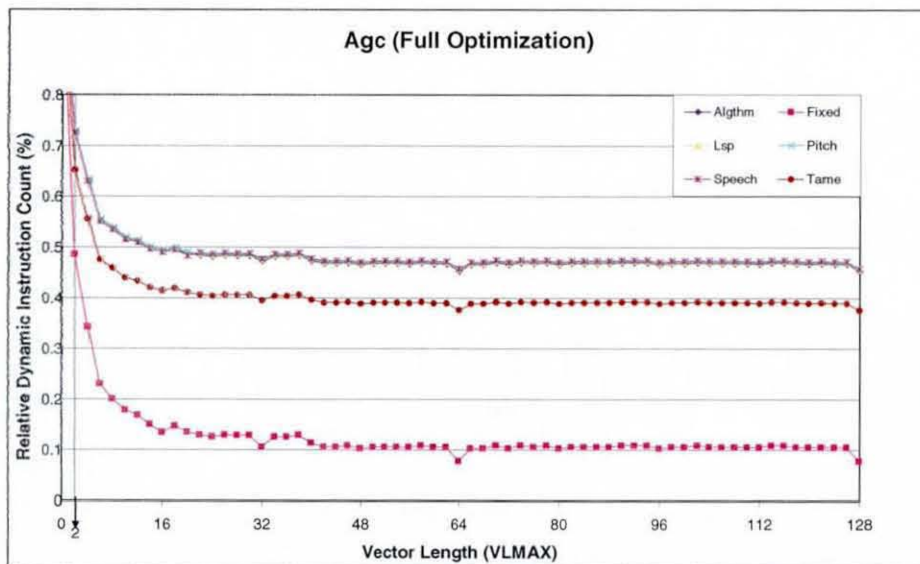


Figure 4-29: Agc (Full Optimization) Results

The dynamic instruction count reduction ranges between 50.6% and 86.6% at vector length sixteen 16-bit elements over the range of reference input bitstreams.

Figure 4-30 illustrates the results of the full optimization of the G.723.1 `Find_Best` function. This function implements the fixed codebook search for the high rate encoder by performing quantization on the residual signal in the MP-MLQ block [28].

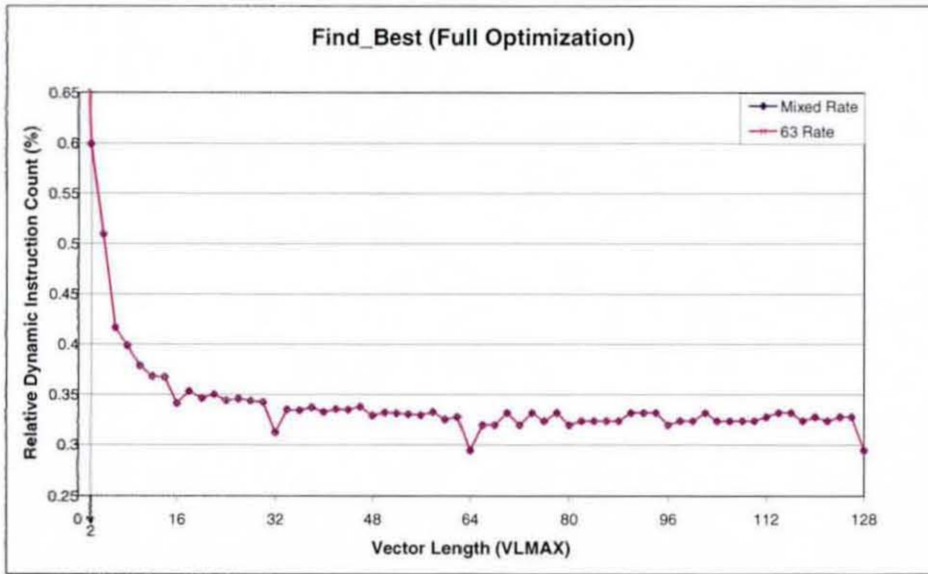


Figure 4-30: Find_Best (Full Optimization) Results

It is interesting to note that this graph only shows results for two workloads: Mixed Rate and 5.3 kbits/s. This is because the codebook search is only done at lower bit rates. The quantization process is approximating the target vector (residual signal) and the excitation is made by positive or negative pulses multiplied by a gain and whose positions can be either all odd or even. The fractional instruction count improvement of this codebook search is 66% at vector length of sixteen (256 bits).

Figure 4-31 illustrates the results for the relative algorithmic complexity of the G.723.1 function `Estim_Pitch`. This function implements the open-loop pitch estimation that is performed twice per frame, one for the first two subframes and one for the last two. The open loop estimate is computed using the perceptually weighted speech that is selected by the maximization of the cross-correlation of the speech method [28].

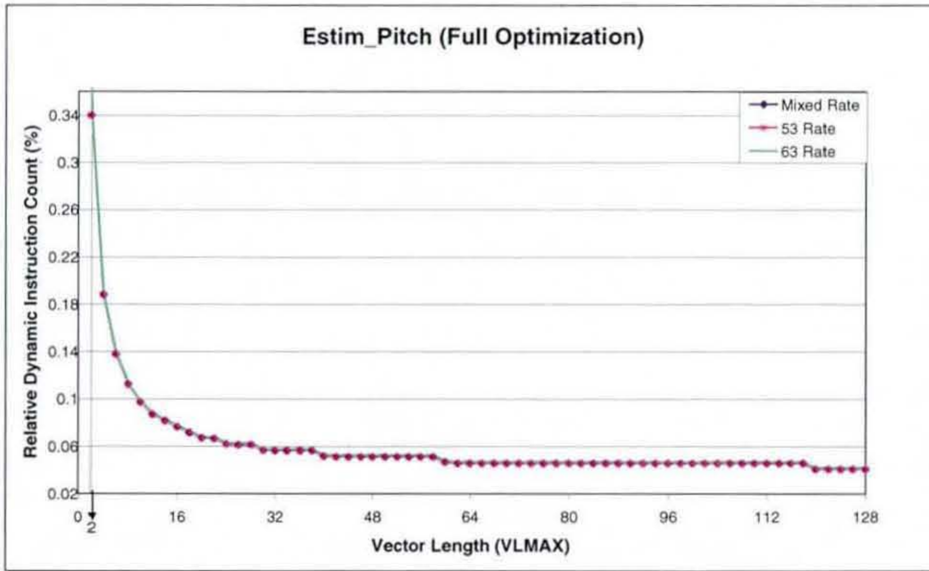


Figure 4-31: Estim_Pitch (Full Optimization) Results

The overall improvement appears for a vector length sixteen and full-optimizations and is of the order of 92.3%. The next plot in Figure 4-32 shows the architecture-level results of the G.723.1 encoder function `Comp_Lpc` that computes the 10th order LPC filter coefficients for every frame. A Hamming-windowed block is centred on the subframe and is used to compute the eleven autocorrelation coefficients that are inputs in the Levinson-Durbin algorithm that generates the LPC coefficients. The produced LPC sets are constructing the short-term perceptual weighting filter that performs the synthesis [28]. This function demonstrates excellent performance scalability and experiences a reduction in dynamic instruction count of approximately 88% at a vector length sixteen (256 bit).

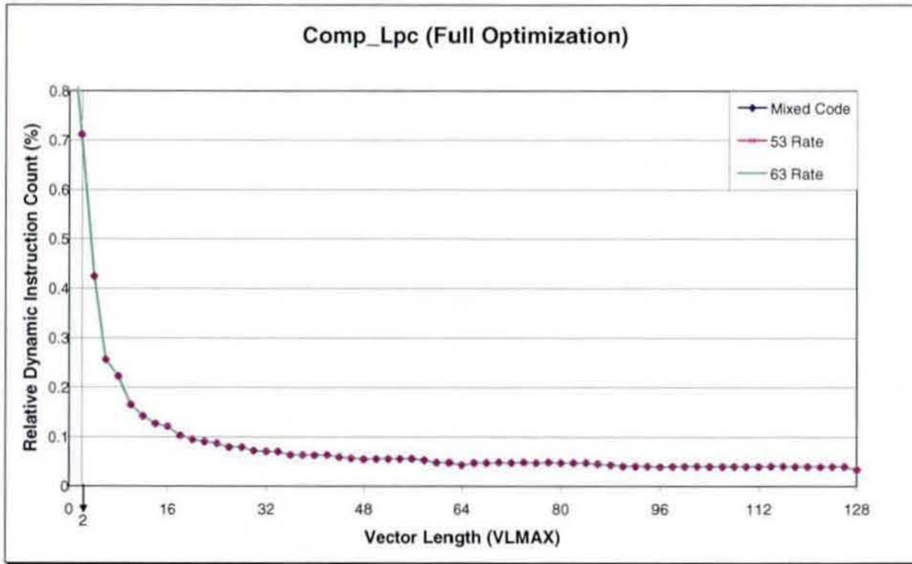


Figure 4-32: Comp_Lpc (Full Optimization) Results

Figure 4-33 shows the relative algorithmic complexity results of the G.723.1 function `Decod_Acbk` that computes the adaptive codebook contribution from the previous excitation vector in the pitch predictor [28].

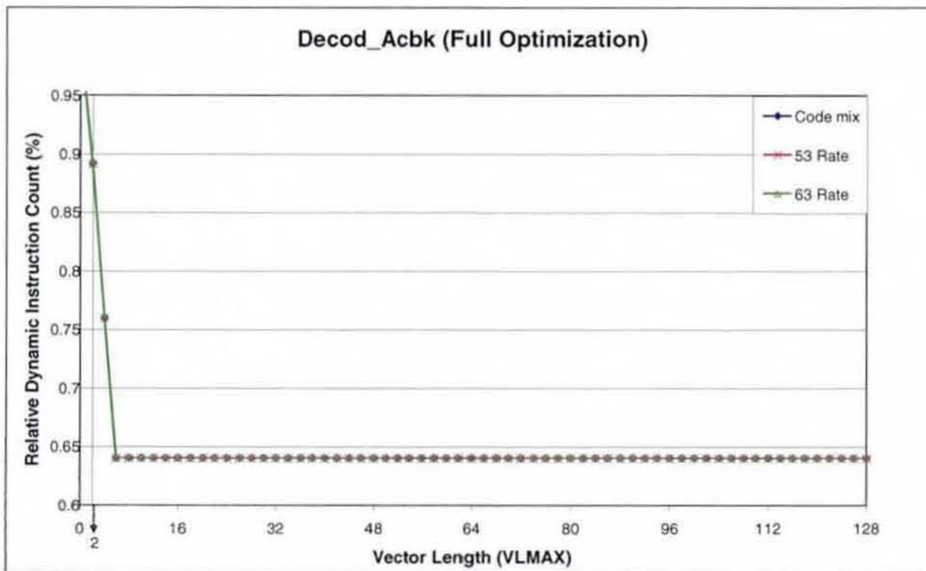


Figure 4-33: Decod_Acbk (Full Optimization) Results

The reduction in the dynamic instruction count is 36% at vector length of sixteen (256 bit) and full-optimizations. As it can be seen there is no further improvement beyond

vector length of 6 (96 bits) as the number of iterations for the internal loop of this function is 6.

Figure 4-34 depicts the simulation results for the G.723.1 function `Comp_Pw` that computes the harmonic noise filter coefficients. The optimal lag for this filter is searched around the open loop pitch lag that maximises the positive correlation [28].

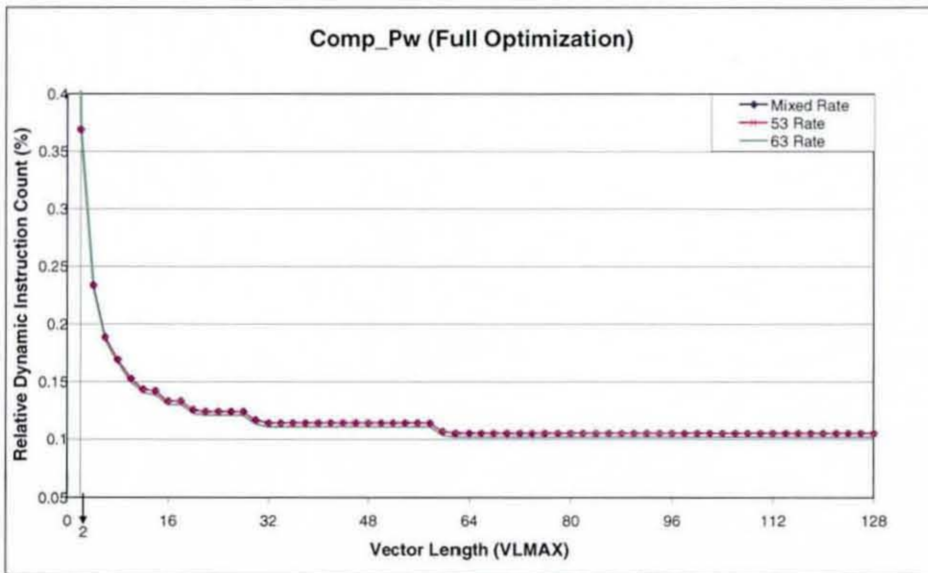


Figure 4-34: Comp_Pw (Full Optimization) Results

The results show that the improvement in the dynamic instruction count performance metric is of the order of 87.6% at a vector length of 16. In the following table are the most compute-intensive functions of G.732.1 in order to have an overall view of the performance improvement.

4.5 Summary

This chapter described the optimization methodology and the performance improvements that were achieved via custom vector and scalar ISA extensions and optimizations of both speech coding standards. During this process the workloads were profiled over a range of vector lengths to identify the enhancement the custom ISA extensions have produced. The architectural results are very promising, demonstrating a reduction in the dynamic instruction count metric of 58% and 71% for G.729A and G.723.1 speech coders respectively when the vector instructions were introduced and a further 18% and 9%

reduction in dynamic instruction count when the scalar instructions were applied. These results show the potential benefit of applying custom instructions and having associated coprocessor vector functional units. The overall simulation results indicate that the area/performance points of interest lie in between 64-bit to 256-bit wide configurations. In addition both sets of results reveal that the maximum benefit is achieved by a combination of custom vector and scalar architectures. From this, the microarchitecture can be designed and attached to a generic RISC CPU. This is explained in more detail in the next chapters.

4.6 References

[1] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," in *Computer.* vol. 35 - no. 2, February 2002 pp. 59-67.

[2] S. Dwarkadas, J. R. Jump, and J. B. Sinclair, "Execution-Driven Simulation of Multiprocessors: Address and Timing Analysis," *ACM Transactions on Modeling and Computer Simulation (TOMACS),* vol. 4, pp. 314 - 338 October 1994.

[3] J. L. Peterson, P. J. Bohrer, and e. al, "Application of full-system simulation in exploratory system design and development," vol. 50, pp. 321-332, March 2006.

[4] S. Hangal and M. O'Connor, "Performance Analysis and Validation of the picoJava Processor," in *IEEE Micro.* vol. 19, May 1999, pp. 66-72.

[5] T. A. Diep, C. Nelson, and J. P. Shen, "Performance evaluation of the PowerPC 620 microarchitecture," in *Proceedings of the 22nd annual international symposium on Computer architecture,* S. Margherita Ligure, Italy, 1995, pp. 163-174.

[6] L. Guerra, J. Fitzner, D. Talukdar, C. Schläger, B. Tabbara, and V. Zivojnovic, "Cycle and phase accurate DSP modeling and integration for HW/SW co-verification," in *in Proceedings of the 36th ACM/IEEE conference on Design automation,* New Orleans, Louisiana, United States, 1999, pp. 964 - 969

[7] P. Mishra, N. Dutt, and H. Tomiyama, "Architecture Description Language driven Validation of Dynamic Behavior in Pipelined Processor Specifications," CECS Technical Report #03-25, Center for Embedded Computer Systems, University of California, Irvine July 2003.

[8] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: A languagefor architecture exploration through compiler/simulator retargetability," in *in Proceedings of Design Automation and Test in Europe (DATE),* 1999, pp. 485-490.

[9] F. S.-H. Chang, "Fast Specification of Cycle-Accurate Processor Models," in *Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors,* 2001, pp. 488-492.

[10] G. Zimmermann, "The MIMOLA design system a computer aided digital processor design method," in *in Proceedings of the 16th ACM IEEE Conference on Design automation* San Diego, CA, United States, 1979, pp. 53-58.

[11] M. Reshadi and N. Dutt, "Generic Pipelined Processor Modeling and High Performance Cycle-Accurate Simulator Generation," in Proceedings of the conference on Design, Automation and Test in Europe, 2005, pp. 786 - 791.

[12] M. Freericks, "The nML machine description formalism," Technical Report 1991/15, Technische Fachbereich Informatik, Berlin University, Berlin 1991.

[13] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An Instruction Set Description Language for Retargetability," in *in Proceedings of the 34th annual conference on Design automation,* Anaheim, California, United States, 1997, pp. 299-302.

[14] M. Barbacci, "Instruction Set Processor Specifications (ISPS): The Notation and Its Applications," *IEEE Transactions on Computers,* vol. 30(1), pp. 24-40, 1981.

[15] G. Mulley, "Using Ismene to Debug and Predict the Performance of an Embedded System Device Driver," University of Glamorgan, Technical report 2004.

[16] T. Hoshino, "UDL/I version Two: A New Horizon of HDL Standards," *IFIP Transactions: Proceedings of the 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications,* vol. A-32, pp. 437 - 452 1993.

[17] V. Zivojnovic, S. Pees, and H. Meyr, "LISA-machine description language and generic machine model for HW/SW co-design," *in IEEE Workshop on VLSI Signal Processing,* pp. 127-136, 1996.

[18] W. S. Mong and J. Zhu, "A retargetable micro-architecture simulator," in *Proceedings of the 40th ACM IEEE conference on Design automation*, Anaheim, CA, USA, 2003, pp. 752-757.

[19] G. Maturana, J. L. Ball, J. Gee, and e. al, "Incas: A Cycle Accurate Model of UltraSPARC," in *Proceedings of the 1995 International Conference on Computer Design: VLSI in Computers and Processors*, Los Alamitos, California, October 1995, pp. 130-135.

[20] J. L. Hennessy and D. J. Patterson, *Computer Architecture: A Quantitative Approach* 2nd ed.: Morgan Kaufman, 1996.

[21] D. Martin, "Vector Extensions to the MIPS-IV Instruction Set Architecture (The VIRAM Architecture Manual) Revision 3.7.5.," March 2000.

[22] T. M. Austin, "SimpleScalar 3.0a pre-release," SimpleScalar LLC: http://www.simplescalar.com.

[23] V. A. Chouliaras, K. Koutsomyti, T. Jacobs, S. Parr, D. Mulvaney, and R. Thomson, "SystemC-defined SIMD instructions for high performance SoC architectures," in *13th IEEE International Conference on Electronics, Circuits and Systems*, Nice France, December 10-13, 2006.

[24] V. A. Chouliaras and J. L. Nunez, "Scalar Coprocessors for accelerating the G723.1 and G729A Speech Coders," *IEEE Transactions on Consumer Electronics,* vol. 49, pp. 703-710, August 2003.

[25] V. A. Chouliaras, J. Nunez, S. R. Parr, K. Koutsomyti, D. J. Mulvaney, and S. Data, "Development of custom vector accelerator for high-performance speech coding," *IEE Electronics Letters,* vol. 40, pp. 1559-1561, Nov 2004.

[26] K. Asanovic, "Vectorizing SPECint95," in *Computer Science Division*. vol. Unpublished manuscript extracted from PhD Thesis California: Berkeley, March 1998.

[27] ITU-T Recommendation G.729A, "Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear-prediction (CS-ACELP)," 3/96.

[28] ITU-T Recommendation G.723.1, "Dual Rate Speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s," 3/96.

[29] R. Allen and K. Kennedy, "Automatic translation of FORTRAN programs to vector form," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, pp. 491 - 542, October 1987.

[30] "http://gcc.gnu.org/onlinedocs/."

[31] K. Koutsomyti, S. R. Parr, V. A. Chouliaras, and J. Nunez, "Applying Data-Parallel and Scalar Optimizations for the efficient implementation of the G.729A and G.723.1 Speech Coding Standards," in *Proceedings of the 7th IASTED International Conference, Signal and Image Processing*, Honolulu, Hawaii, USA, August 2005, pp. 40-45.

CHAPTER 5
VECTOR PROCESSOR ARCHITECTURE

5.1 Vector Architectural State

The vector-scalar coprocessor is attached to the Sparc-V8 compliant CPU core via a custom, pipelined coprocessor interface. The accelerator consists of two major unit microarchitectures: One parametric microarchitecture that implements the vector ISA and a second that implements the scalar ISA. The coprocessor attaches to the integer unit of the Leon CPU in the fifth pipeline stage which is the memory stage. It was not designed as a stand-alone AHB coprocessor because, though the workloads perform a lot of work on blocks of data (samples), there where many more instances where custom assembly code (scalar) needed to be inserted into irregular (non-iterative) blocks. Therefore a very tightly-coupled configuration was pursued which accommodates efficiently both cases [1]. The coprocessor is connected to the memory stage in order to avoid the majority of the exceptions and interruptions of the Leon CPU and to have enough time to transfer data to/from the main processor if requested. Therefore, when a valid vector coprocessor instruction is encountered and there is no exception or pipeline stall then the vector/scalar instruction along with a valid signal is sent to the first stage (decode) of the vector coprocessor pipeline for execution. By defining coprocessor extension instructions instead of a full stand-alone instruction set allows taking advantage of any developments in the Leon architecture and use of the development tools available for the latter. In addition, the coprocessor can be imported into any other embedded CPU architecture with very little modifications.

The vector pipeline is a SIMD array of functional units (FUs). The functional units are organised in four groups: Addition (`vadd`), multiplication (`vmult`), shift (`vshift`) and miscellaneous (`vmisc`). Each group has a parametric number of functional units equal to half the maximum vector length (VLMAX/2) where VLMAX can take values that are power of 2. On every cycle, only one of the aforementioned FU groups is active. The subdivision of the vector pipeline into the four vector FU groups is detailed in the next chapter in section 6.4. The VLMAX/2 vector FUs are driven by the corresponding slices

of the operand registers (vector elements), stored in the vector register file. These slices provide, per unit, two read ports (2 x 32-bits) and a write port (32-bits). Each functional unit has a dynamically configurable 2-way SIMD or scalar organisation, depending on whether the instruction produces 2x16-bit results or 32-bit. The vector length is located in the vector length (`vlen`) register and defines the width of the vector registers and the number of FU that are utilised to perform an operation. It does not alter any of the hardware resources. All vector operations are governed by the current vector length and a vector mask. The current vector length is taken from the `vlen` register and the vector mask is implemented by a combinational logic that differs for each specific instruction. The FUs take their source operands from either vector registers, scalar registers or vector accumulators and can perform both vector and scalar operations. Each functional unit of the group active in the current cycle accepts 32-bit source operands and produces a 32-bit result except from the FUs of the vmult group that can handle 16-bit input operands and produce a 32-bit result. Figure 5-1 illustrates an example of a vector operation that is performed in two source vector registers.

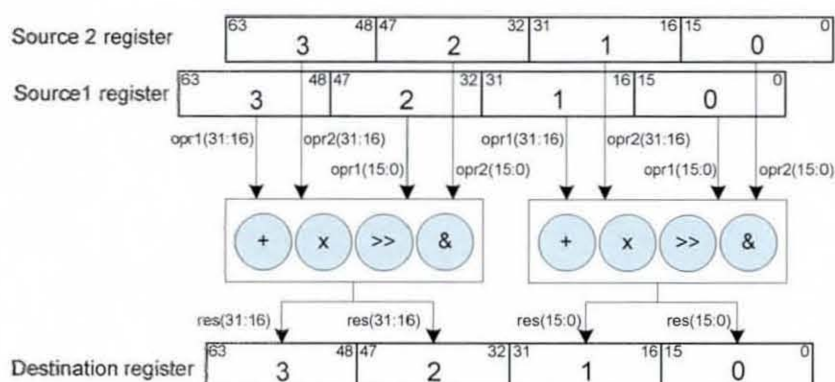


Figure 5-1: Example of an operation that is performed in two vector registers with vector length 64-bits. Each functional unit is driven by the pair of the corresponding slices (vector elements) of the source vector registers. The produced results are stored back to the corresponding slices (vector elements) of the destination vector register.

In the case of scalar instructions only the first (FU0) functional unit from the active group operates whereas the others do not change state (via clock gating and combinational logic gating) in order to save power. The control/status flag and registers have two uses: to support predicated execution and to store exception bits that are implicitly set by instructions that may produce the relevant exceptions [2].

5.2 Programmers Model

The user programming model is shown in Figure 5-2. Along with the instruction set it completes the portion of the architecture that is visible to software. The programmer's model contains two types of registers, the general-purpose registers and the control/status registers. The general-purpose registers consist of the vector and scalar register files. The vector register file contains VREGS vector registers of statically-configurable length VLMAX of scalar 16-bits elements with two read ports and one write port. The VREGS configuration constant can take values from 2 to 32 and in this instance of the architecture that value is 16. The 16 vector registers are individually designated by the symbols VR0, VR1,..., VR15 as illustrated. The scalar register file contains SREGS general-purpose scalar registers of 32-bits width and it has three read ports and one write port. The SREGS configuration constant is 16 in the current implementation but can be any value between of 2-32. The 16 scalar registers are individually designated by the symbols SR0, SR1,..., SR15, as illustrated in the model of Figure 5-2. The scalar registers can serve a number of purposes including use as address pointer registers, for scalar memory references, provide data values for vector and scalar operations, store final or intermediate results etc.

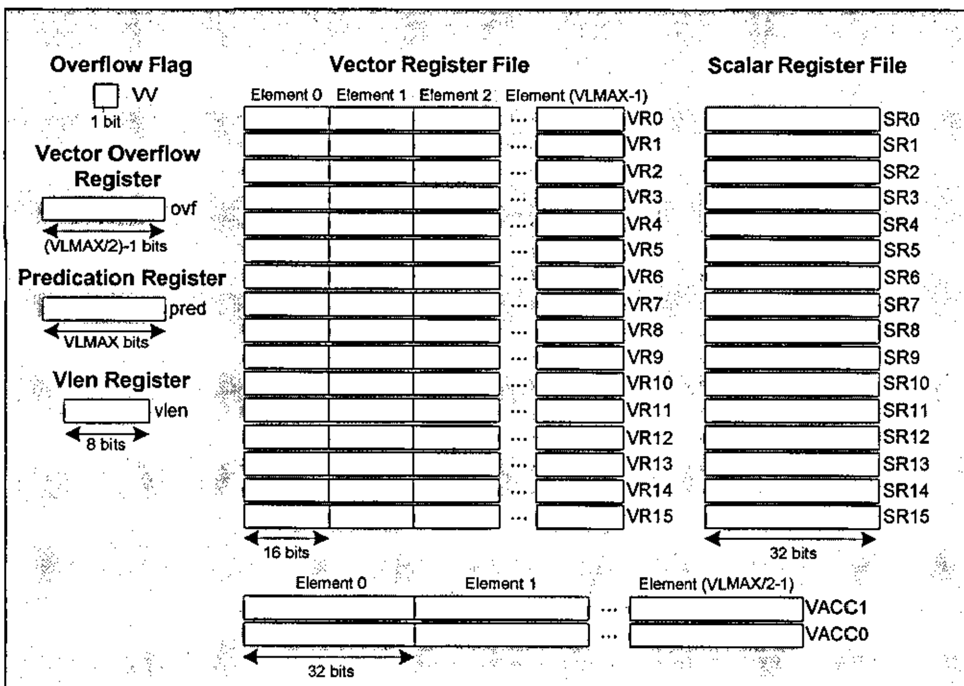


Figure 5-2: Vector and Scalar coprocessor programmer's model

There are also ACC_NUMBER vector accumulators consisting of VLMAX/2 scalar elements (32-bit). The ACC_NUMBER configuration constant in this case is 2 (VACC0, VACC1) but can be any value of the range of 2-32 with the restriction for the long instructions which access the accumulators, except the multiply-add/sub instructions, that can use only the first two accumulators (VACC0, VACC1) as source operands. There are special move instructions that exchange data between the vector, scalar and Leon general purpose registers and the vector accumulators. The control/status registers include a vector length register (`vlen`), a predication register (`pred`), a vector overflow register (`ovf`) and an overflow flag (VV). The `vlen` register has maximum value of VLMAX and defines the width of the data that will be processed by the vector datapath. The predication register is a type of mask register with VLMAX bits where each bit corresponds to a vector element. It is set when a comparison instruction takes place and it is utilised during merge operations to select the appropriate vector elements that comprise the vector comparison result (merge operation). The overflow flag (VV) is a single bit and it is set whenever an overflow happens during arithmetic instructions. Internally, multiple overflow flags are generated where each such flag corresponds to one vector element, and they are combined in a single overflow flag by using an or-reduce operation. In addition, there is a vector overflow register (`ovf`) that is VLMAX/2-bit long where every bit is the overflow result of each functional unit of the group that performed the particular operation. The only vector mask register is the predication register that is employed for the comparison and merge operations. All the other masking processes are implemented on the run by combinational logic obeying the current vector length value.

5.3 Vector Processor Instruction Set Architecture

The instruction set defines the transformations the software component can perform in the architectural state, including both memory and register file. Instructions define one or more operations for a scalar set of data. The vector instruction set, on the other hand, allows software to express, with a single opcode, multiple independent operations on arrays of data [3]. This section describes the instruction set that implements most of the basic DSP operations on the target, G.729A and G.723.1, ITU-T speech coding algorithms. These operations are more complicated than the basic operations of a RISC architecture and are described in this document in two levels of detail. The first level of

detail is presented in the remaining of the chapter which is divided into sections that present and briefly describe groups of instructions of similar types. Each group is expanded into a more detailed description for each instruction that comprises it. This is the second level of detail that is contained in the Appendix A and contains for every instruction, its format, a short description of the instruction's operation and a software example. In the proposed processor architecture all the coprocessors instructions are 22 bits wide and include 2 and 3-address formats (1 or 2 source operand registers and the destination register, all independently specified). The instruction set is divided into two main categories; the vector instructions and the scalar instructions.

5.3.1 Vector ISA

The vector instruction set described in this document comprises 43 instructions which are divided into groups of instructions of similar types. Every type is detailed by showing assembly formats and giving a short description of the instruction's operation. More detail is contained in Appendix A, where each instruction is presented separately. The vector instructions can be grouped into five categories: load/store, move, arithmetic, shift and miscellaneous. The assembly language format of an instruction is written with a shorthand notation and few examples of the vector and scalar assembly are given. In vector mode the coprocessor can process in parallel VLMAX 16-bit operations or VLMAX/2 32-bit operations.

5.3.1.1 Load/Store Instructions

Vector load /store instructions are the only instructions that access memory via the Vector Load/Store Unit (VLSU) and are illustrated in Table 5-1. This table also includes the instruction that loads the `vlen` register (`ldvlen_r`) with an immediate even if it is not regarded as a load instruction in a typical sense.

Table 5-1: Vector Load/Store Instructions

No	Instruction	Assembly	Brief Description
1	**ldvlen_r**	ldvlen_r(imm)	Load Vector Length Register with immediate
2	**vldw**	vldw(vrd,srs1)	Load vector register from memory address
3	**vldwn**	vldwn(vrd,srs1)	Load vector register downward from memory
4	**vstw**	vstw(vrs2,srs1)	Store vector register back to memory address
5	**vstwn**	vstwn(vrs2,srs1)	Store vector register downwards to memory
6	**vldaccw**	vldaccw(vaccd,srs1)	Load vector accumulator from memory address
7	**vstacc**	vstacc(vacc,velem,srs1)	Store vector accumulator element to memory

Vector load/store operations use a scalar register (`srs1`) that contains the memory address in which data is loaded from/stored to. For the load instructions the destination can be a vector register (`vldw`) or a vector accumulator (`vldaccw`) while for the store instructions (`vstw` or `vstacc`) these registers are the data sources. The load/store instructions are strided. A strided load takes a base address, in this case the `srs1`, and a signed stride, and loads a vector of values starting at the base address, where each element is separated by the stride amount. The stride is in units of elements, not bytes and can take the values 1 (`vldw`) and –1 for load downward (`vldwn`). A similar method applies for the strided store in which a vector of values is stored starting from the base address and be separated from 1 (`vstw`) or –1 stride for store downward (`stwn`) [2]. Store instructions have one cycle latency and are performed in the Vector Register Access (VREG) stage where the store data, along with the memory address, are sent to the VLSU unit. Load instructions have latency of two cycles as the VLSU unit has a cascade TAG/DATA configuration. During a load operation the load address is sent to VLSU unit at the VREG stage and the memory data are obtained at the second Vector Datapath (VDP2) stage. It is clear that the load/store latency depends on the VLSU implementation. A parallel TAG/DATA configuration for the VLSU microarchitecture will reduce the load instruction latency from two cycles to one cycle at the expense of increased power consumption.

5.3.1.2 Move Instructions

The vector move instructions are used to exchange data between the vector, scalar and Leon general purpose registers as well as the vector accumulators. They comprise move instructions (`mvvr2gpr` or `mvgpr2vr`) that transfer data between coprocessor's vector registers and the main CPU's (Leon) general-purpose registers.

Table 5-2: Vector Move Instructions

No	Instruction	Assembly	Brief Description
8	**vaccclr**	vaccclr(vacc)	Set the value in the vector accumulator to zero
9	**vsplatacci**	vsplatacci(vaccd,srs1)	Load vector accumulator with a scalar value
10	**vldacceli**	vldacceli (vaccd,velem,value)	Load immediate value into vector accumulator element
11	**vsplat_h_r**	vsplat_h_r(vrd,srs1)	Splat a 16-bit scalar value to all elements of vector register
12	**vmvacctre**	vmvacctre (vrd,vacc1,amount)	Extract high (amount=0) or low (amount=16) the even elements of vector accumulator and load them to vector register
13	**vmvacctro**	vmvacctro (vrd,vacc1,amount)	Extract high (amount=0) or low (amount=16) the odd elements of vector accumulator and load them to vector register
14	**vmvrtacce**	vmvrtacce (vaccd,vrs1,amount)	Deposit high (amount=16) or low (amount=0) the even elements of vector register to the vector accumulator
15	**vmvrtacco**	vmvrtacco (vaccd,vrs1,amount)	Deposit high (amount=16) or low (amount=0) the even elements of vector register to the vector accumulator
16	**mvgpr2vr**	mvgpr2vr (vrd,velem,grs1)	Moves a value (32-bit) from the general purpose register (Leon) to the vector register element
17	**mvvr2gpr**	mvvr2gpr (grd,velem,vrs1)	Moves the vector register element to the general purpose register (Leon)

Splat instruction (`vsplat_h_r`) "splats" a scalar value in a vector register and deposit instructions (`vmvrtacce` and `vmvrtacco`) deposit low or high data from a vector register to a vector accumulator. The extract instructions (`vmvacctre` and `vmvacctro`) are utilized to extract high or low data from vector accumulators into vector registers. Finally, they comprise instructions that set to zero (`vaccclr`), splat scalar data (`vsplatacci`) or load an immediate value (`vldacceli`) into a vector accumulator. All the move instructions are summarised in Table 5-2.

5.3.1.3 Arithmetic Instructions

The vector arithmetic instructions include short and long addition, subtraction and multiplication. All the arithmetic instructions are performed in a single cycle apart from the multiply-add (`vmace`/`vmaco`) and the multiply-sub (`vmsue`/`vmsuo`) which take two cycles. The short addition (`vaddh`) and subtraction (`vitu_sub_r`) take as inputs two vector registers and perform a 16-bit addition operation. The long addition (`vaddacc`) and subtraction (`vsubacc`) take as inputs vector accumulators and perform 32-bit addition operations. Figure 5-3 shows a vector addition for a vector length of 2 that specifies two vectors as input operands and produces a vector result by executing the

same operation on each pair of elements from the input arrays. The multiply instructions are implemented as pairs for the even and odd elements of the vector registers as the multiplier for every vector functional unit takes as input two 16-bits and produces a 16-bit (short multiplication) or 32-bit (long multiplication) product. Figure 5-4 illustrates a short multiplication of two vectors with vector length 2.

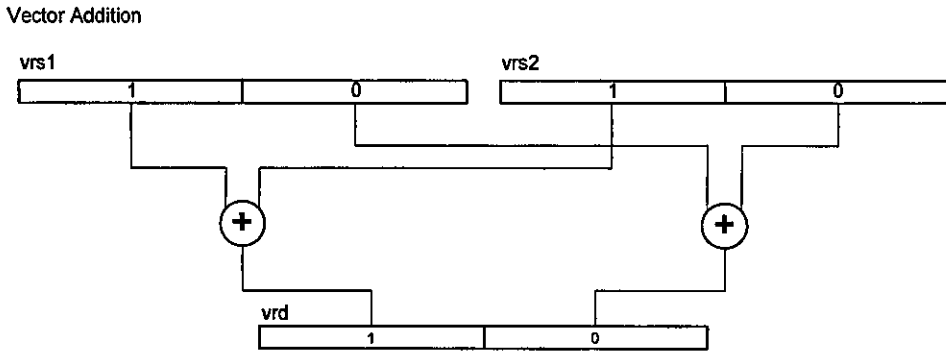


Figure 5-3: Vector Short Addition

It takes as inputs the even elements of the pair of vector registers (elements 0) and the product is placed in the even element of the destination register.

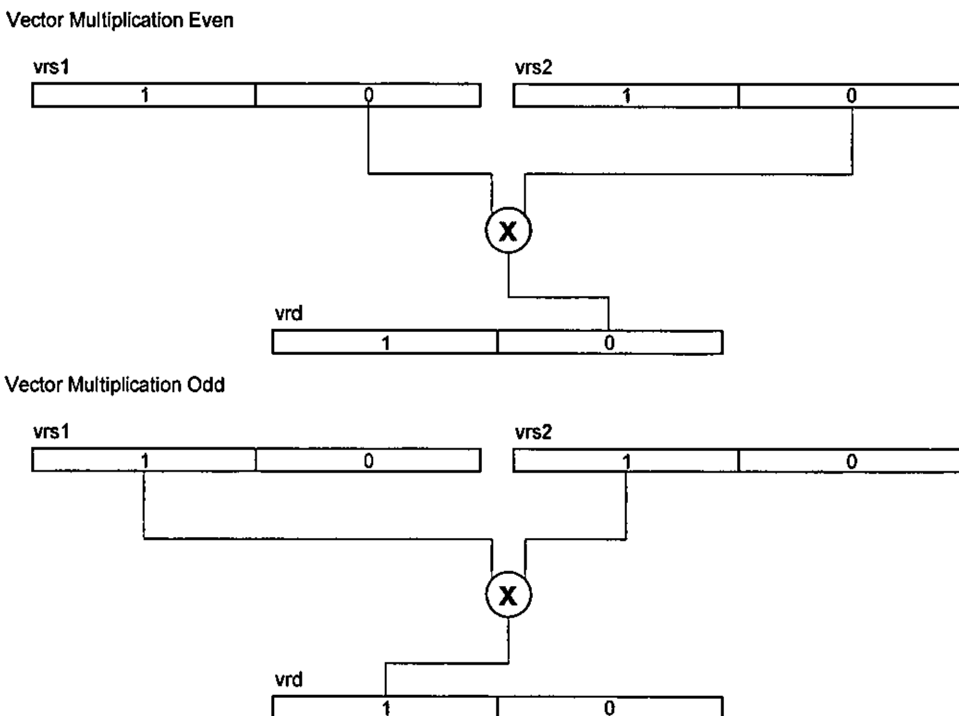


Figure 5-4: Vector Short Multiplication for even/odd elements

Then it takes as inputs the odd elements of the pair of vector registers (elements 1) and the product is placed to the odd element of the destination register. The short multiplication involves simple multiplication (mult), multiplication with rounding (mult_r) and integer multiplication (imult). All these multiply instructions perform a signed or unsigned 16 × 16 → 16-bit operation. The long multiplication performs a signed 16 × 16 → 32-bit operation and, along with the multiply-add, is executed from the pair of instructions vmace/vmaco but without the accumulation part. The multiply-add (vmace/vmaco) and the multiply-sub (vmsue/vmsuo) instructions are performed in the even and odd elements respectively of the vector registers vrs1 and vrs2 and add or subtract the product to the even and odd elements of the vector accumulator vacc.

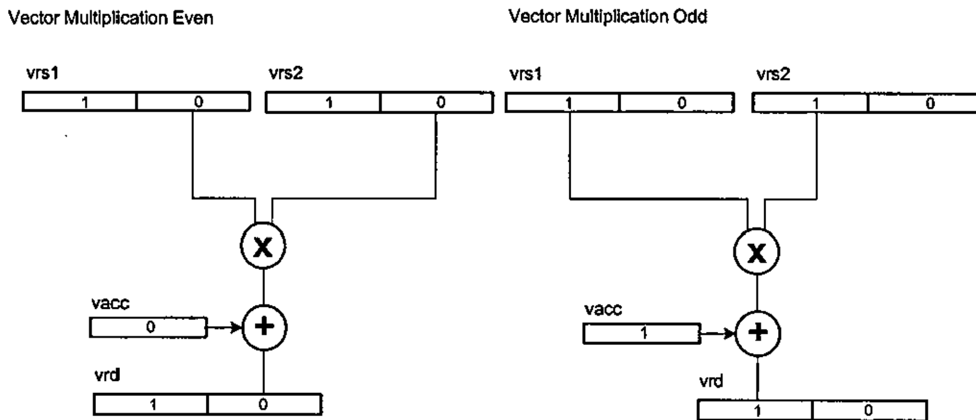


Figure 5-5: Vector multiply-add/sub

Finally, the vaccaddreduce is used after the execution of the pair instructions that involve the accumulator and perform add-reduce to the elements of the accumulator. With the use of an adder tree, a 32-bit final result is obtained and it is placed to the element 0 of the vector accumulator. All the vector arithmetic instructions are summarized in Table 5-3.

Table 5-3: Arithmetic Instructions

No	Instruction	Assembly	Brief Description
18	**vaddh**	vaddh(vrd,vrs1,vrs2)	Vector short addition (16-bit) of vector registers
19	**vitu_sub_r**	vitu_sub_r(vrd,vrs1,vrs2)	Vector short subtraction (16-bit) of vector registers
20	**vaddacc**	vaddacc(vaccd,vacc1,vacc2)	Vector long addition (32-bit) of vector accumulators
21	**vsubacc**	vsubacc(vaccd,vacc1,vacc2)	Vector long subtraction (32-bit) of vector accumulators
22	**vaccaddreduce**	vaccaddreduce (vacc)	Vector accumulator add-reduce
23	**vitu_mult_e_r**	vitu_mult_e_r (vrd,vrs1,vrs2)	Vector signed short multiply of the vector registers even elements
24	**vitu_mult_o_r**	vitu_mult_o_r (vrd,vrs1,vrs2)	Vector signed short multiply of the vector registers odd elements
25	**vitu_mult_r_e_r**	vitu_mult_r_e_r (vrd,vrs1,vrs2)	Vector short multiply with rounding of the vector register even elements
26	**vitu_mult_r_o_r**	vitu_mult_r_o_r (vrd,vrs1,vrs2)	Vector short multiply with rounding of the vector registers odd elements
27	**vitu_i_mult_e_r**	vitu_i_mult_e_r (vrd,vrs1,vrs2)	Vector short integer multiply of the vector registers even elements
28	**vitu_i_mult_o_r**	vitu_i_mult_o_r (vrd,vrs1,vrs2)	Vector short integer multiply of the vector elements odd elements
29	**vmace**	vmace (vacc,vrs1,vrs2)	Vector mutliply-add (L_mac) of the vector registers even elements
30	**vmaco**	vmaco (vacc,vrs1,vrs2)	Vector mutliply-add (L_mac) of the vector registers odd elements
31	**vmsue**	vmsue (vacc,vrs1,vrs2)	Vector mutliply-sub(L_msu) of the vector registers even elements
32	**vmsuo**	vmsuo (vacc,vrs1,vrs2)	Vector mutliply-sub(L_msu) of the vector registers odd elements

5.3.1.4 Shift Instructions

The shift instructions implement the 16 and 32-bit ITU shift operations. These operations have also the ability to specify negative shift amounts resulting in a positive shift in the opposite direction. In addition they saturate the result in the range of 0xffff8000-0x00007fff in case of overflows or underflows. The short (16-bit) shifts are performed in a vector register with an immediate or with the shift amount being in the second vector register. The long (32-bit) shifts are implemented in vector accumulator with an immediate value or with the amount stored in a vector register. All the shift instructions are summarized in Table 5-4.

Table 5-4: Vector Shift Instructions

Page	Instruction	Assembly	Brief Description
33	**vshli**	vshli (vrd,vrs1,amount)	Vector short (16-bit) shift left by amount
34	**vshri**	vshri (vrd,vrs1,amount)	Vector short (16-bit) shift right by amount
35	**vshlr**	vshlr (vrd,vrs1,vrs2)	Vector short shift left with register
36	**vshrr**	vshrr (vrd,vrs1,vrs2)	Vector short shift right with register
37	**vlshlacc**	vlshlacc (vaccd,vacc1,amount)	Vector long (32-bit) shift left by amount
38	**vlshracc**	vlshracc (vaccd,vacc1,amount)	Vector long (32-bit) shift right by amount
39	**vlshlaccr**	vlshlaccr (vaccd,vacc1,vrs1)	Vector long (32-bit) shift left with register
40	**vlshraccr**	vlshraccr (vaccd,vacc1,vrs2)	Vector long (32-bit) shift right with register

5.3.1.5 Miscellaneous Instructions

The miscellaneous instructions for the vector ISA perform only comparison operations between vector registers (16-bit) or vector accumulators (32-bit) and comparison with zero. The compare instruction compares the two operands together by subtracting the one from the other. If the result is positive (first operand is greater than or equal to the second operand register, accumulator or zero) the predication flag (`pred`) is set to '1'. If the result is negative (first operand is less than the second) the predication flag is set to '0'. Finally the merge instructions are utilised to select the vector register or accumulator value that satisfies the given equation, on a per-element basis.

Table 5-5: Vector Miscellaneous Instructions

Page	Instruction	Assembly	Brief Description
41	**vcmp**	vcmp(vacc1,vacc2)	Compare vector accumulators and update Predication flag (pred)
42	**vrcmp**	vrcmp(vrs1,vrs2)	Compare vector registers and update Predication flag (pred)
43	**vcmp_h_ge**	vcmp_h_ge(vrs1)	Check vector register if it is greater than or equal to zero and update Predication flag
44	**vmerge_t_h_r**	vmerge_t_h_r (vrd,vrs1,vrs2)	Merge two vector registers according to the Predication flag value
45	**vmerge**	vmerge (vaccd,vacc1,vacc2)	Merge two vector accumulators according to the predication flag value

This is a multiplexer-style operation that selects between two values which one to pass to the output result, according to the predication flag value. The miscellaneous instructions are depicted in the above table.

5.3.2 Scalar ISA

The scalar instruction set comprises 36 instructions which are grouped into five categories: load/store, move, arithmetic, shift and miscellaneous. Each category is presented to the following sections whereas a more detailed description for every scalar instruction is given in Appendix A. In scalar mode the coprocessor can accommodate one 16-bit or 32-bit operation.

5.3.2.1 Load/Store Instructions

The scalar load/store instructions access memory via the VLSU unit. The load instructions can load 16 or 32-bit data from the memory location that is contained in scalar register (`srs1`) into the destination register (`srd`). The store instructions store the 16 or 32-bit data of the scalar register (`srs2`) into the memory location stored in scalar register (`srs1`). All the scalar load/store instructions are summarized to the Table 5-6.

Table 5-6: Scalar Load/Store Instructions

Page	Instruction	Assembly	Brief Description
46	**m2sld16**	m2sld16(srd,srs1)	Load scalar register with 16-bit from memory
47	**m2sld32**	m2sld32(srd,srs1)	Load scalar register with 32-bit from memory
48	**m2sst16**	m2sst16(srs2,srs1)	Store 16-bit word of scalar register to memory
49	**m2sst32**	m2sst32(srs2,srs1)	Store 32-bit word of scalar register to memory

5.3.2.2 Move Instructions

The scalar move instructions offer a flexible way to transfer data between the coprocessor's scalar registers and the main CPU's (Leon) general-purpose register file. These instructions comprise the address of the source register (`srs1` or `grs1`) and the address of the destination register (`grd` or `srd`) and are listed in Table 5-7.

Table 5-7: Scalar Move Instructions

Page	Instruction	Assembly	Brief Description
50	**mvgpr2sr**	mvgpr2sr(srd,gpr1)	Moves contents from general purpose register to scalar register
51	**mvsr2gpr**	mvsr2gpr(gprd,srs1)	Moves contents from scalar register to general purpose register

5.3.2.3 Arithmetic Instructions

The scalar arithmetic instructions include short and long addition, subtraction and multiplication. All these arithmetic instructions take as inputs two scalar registers and perform a 16 or 32-bit operation. When the result exceeds the range of 0x80000000-0x7fffffff an overflow bit is produced. In the case of multiply-add and multiply-sub the role of the accumulator is played by a third scalar register that is used both as a source and as destination register. This was also the reason that the scalar register file has three read ports instead of two as the main vector register file has. The scalar arithmetic instructions are listed in Table 5-8.

Table 5-8: Scalar Arithmetic Instruction

Page	Instruction	Assembly	Brief Description
52	**m2sladd**	m2sladd(srd,srs1,srs2)	Scalar Long (32-bit) Addition
53	**m2slsub**	m2slsub(srd,srs1,srs2)	Scalar Long (32-bit) Subtraction
54	**m2sadd**	m2sadd(srd,srs1,srs2)	Scalar Short (16-bit) Addition
55	**m2ssub**	m2ssub(srd,srs1,srs2)	Scalar Short(16-bit) Subtraction
56	**m2slmac**	m2slmac(srd,srs1,srs2)	Scalar multiply-accumulate (L_mac)
57	**m2slmsu**	m2slmsu(srd,srs1,srs2)	Scalar multiply-subtract (L_msu)
58	**m2slmult**	m2slmult(srd,srs1,srs2)	Scalar long (32-bit) multiplication
59	**m2smult**	m2smult(srd,srs1,srs2)	Scalar short (16-bit) multiplication
60	**m2smult_r**	m2smult_r(srd,srs1,srs2)	Scalar multiplication with rounding
61	**m2simult**	m2simult(srd,srs1,srs2)	Scalar short integer multiplication

5.3.2.4 Shift Instructions

Shift instructions are used to shift the contents of a scalar register left or right by a given amount. The shift amount can be specified by a constant (`amount`) in the instruction or by the contents of a scalar register (`srs2`). As with the vector shift instructions, short and long scalar shifts are supported. The scalar shift instructions are summarized in Table 5-9.

Table 5-9: Scalar Shift Instructions

Page	Instruction	Assembly	Brief Description
62	**m2slshl**	m2slshl (srd,srs1,amount)	Scalar long 32-bit shift left by immediate
63	**m2slshr**	m2slshr (srd,srs1,amount)	Scalar long shift right by immediate
64	**m2slshl_rg**	m2slshl_rg (srd,srs1,srs2)	Scalar long shift left with register
65	**m2slshr_rg**	m2slshr_rg (srd,srs1,srs2)	Scalar long shift right with register
66	**m2sshl**	m2sshl (srd,srs1,amount)	Scalar short shift left by amount
67	**m2sshr**	m2sshr (srd,srs1,amount)	Scalar short shift right by amount
68	**m2sshl_rg**	m2sshl_rg (srd,srs1,srs2)	Scalar short shift left with register
69	**m2sshr_rg**	m2sshr_rg (srd,srs1,srs2)	Scalar short shift right with register

5.3.2.5 Miscellaneous Instructions

The miscellaneous instructions perform the remaining instructions that comprise the basic operations of the ITU standard algorithms. They include short and long negate, absolute value, normalization, deposit, extract and rounding.

Table 5-10: Scalar miscellaneous instructions

Page	Instruction	Assembly	Brief Description
70	**m2slnegate**	m2slnegate (srd,srs1)	Scalar long negate (L_negate)
71	**m2slabs**	m2slabs (srd,srs1)	Scalar long absolute value (L_abs)
72	**m2snorm_l**	m2snorm_l (srd,srs1)	Scalar long normalisation (norm_l)
73	**m2sldeposit_l**	m2sldeposit_l (srd,srs1)	Deposits 16 LSB into the LSB of scalar register the remain are sign extended
74	**m2sldeposit_h**	m2sldeposit_h (srd,srs1)	Deposits 16 LSB into the MSB of scalar register the remain are zero extended
75	**m2snegate**	m2snegate (srd,srs1)	Scalar short negate (negate)
76	**m2sabs_s**	m2sabs_s (srd,srs1)	Scalar short absolute value (abs_s)
77	**m2sextract_h**	m2sextract_h (srd,srs1)	Extracts the 16 MSB from scalar register
78	**m2sextract_l**	m2sextract_l (srd,srs1)	Extracts the 16 LSB from scalar register
79	**m2sround**	m2sround (srd,srs1)	Rounds a 32-bit value to 16-bit

They use one scalar register (`srs1`) as source operand and calculate the result that place into the destination register (`srd`). Table 5-10 lists all the miscellaneous instructions.

5.4 Leon3 CPU

Leon3 is an open-source synthesisable VHDL model of a 32-bit processor core implementing the SPARC V8 architecture (standard IEEE-1754) [4]. The model is highly configurable, and particularly suitable for system-on-a-chip (SoC) designs. It is designed for embedded applications that require a high performance, low complexity and low

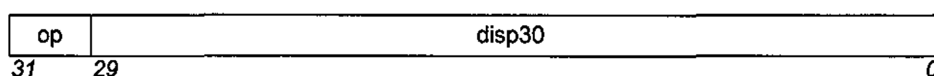
power consumption programmable engine. The Leon3 CPU has a 7 stage pipelined integer unit with a pseudo-Harvard architecture (separate instruction and data caches):

- Fetch Stage: In this stage the instruction is fetched from the instruction cache if it is enabled else a request sent to the memory controller. In addition, the value of the program counter is updated. At the end of this stage the valid instruction and the value of the program counter are latched to the next stage.
- Decode Stage: The instruction is decoded and extracts the addresses for both source operands and the destination operand. Also it generates the addresses for branch and CALL instructions and the control signals for the next stages.
- Register Access Stage: The source operands are read from the register file or from bypassed intermediate results.
- Execute Stage: All the arithmetical, shift and miscellaneous operations are performed. For memory load or store and jump/return operations the address is generated and sent to the memory unit.
- Memory Stage: At this stage the data cache is accessed and the store operation is performed.
- Exception Stage: All the traps and interrupts signals are processed and the data are aligned in the case of a data cache load.
- Write Back Stage: The result from any arithmetical, logical, shift or cache operation is written back to the register file.

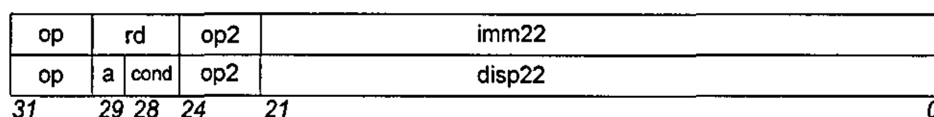
It has an on-chip debug support unit and interfaces to a Floating-point unit (FPU) and a custom coprocessor. The Leon3 processor implements the full SPARC V8 Reference Memory Management Unit (SRMMU) and its interrupt model recognises and handles 15 asynchronous interrupts. The number of the registers in the register file is configurable within the range of 2 to 32 with a default value of 8. The cache system is highly configurable as well and is connected to two independent cache controllers for the instruction and data caches respectively (icache.vhd and dcache.vhd) [4]. In addition, there is an interface between the two caches controllers and the Amba AHB bus (acache.vhd). Both caches are configured to be direct-mapped or multi-set with set associativity of 2-4 sets, where every set can be 1-256 Kbytes and be divided into cache lines (blocks) of 16-32 bytes each. The Leon3 includes a hardware multiplier, with optional 16x16 bit MAC and 40-bit accumulator, and a divider. In this research, we will

consider the integer unit of the Leon3 processor in which the vector processor is attached in a closely coupled configuration. Leon3 can be configured to provide a generic interface to a user-defined co-processor. The interface allows the operation of the coprocessor in parallel increasing this way the performance. The vector coprocessor is a hardware component that will run in parallel with the Leon3 and will exchange data with it. In order to perform this, the coprocessor-allocated opcodes must be ignored by the decode logic of the pipeline of Leon3. This means that the Leon3 should treat these instructions in a benign way, as is the case of a nop instruction. From the SPARC architecture manual it can be seen that the instructions are encoded in three major 32-bit formats as illustrated in Figure 5-6.

Format 1 (op = 1): CALL



Format 2 (op = 0): SETHI & Branches (Bicc, FBfcc, CBccc)



Format 3 (op = 2 or 3): Remaining instructions

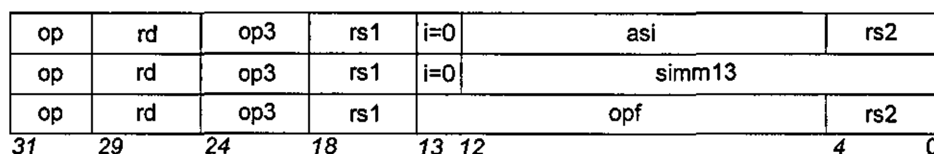


Figure 5-6: Instruction Formats of Leon3

The format that can be used for the vector coprocessor and will demand only few modifications of the Leon3 decode logic is the unimplemented instruction (UNIMP). The values of the UNIMP instruction are not reserved by the architecture for any future use and the *const22* value is ignored by the hardware [5]. The UNIMP instruction is an instruction with unimplemented opcode that causes an illegal_instruction trap and its format is shown in Figure 5-7.

Format 2 (op = 0): UNIMP

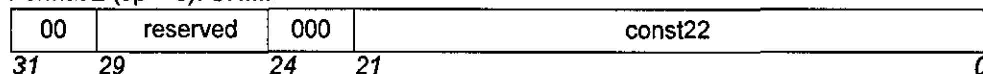


Figure 5-7: Unimplemented Instruction

Because the UNIMP instruction causes an illegal_instruction trap at the exception detection stage additional decode logic and modifications in the existing decode logic prevent the exception process from setting the ***illegal_inst*** signal. Furthermore, the Leon3 was modified to perform add with zero when the allocated opcode is decoded. In this way, the Leon3 IU performs a nop instruction while the 22-bits of the UNIMP opcode (*const22*) are sent for further decoding in the vector coprocessor. Therefore the available 22-bits are utilised for encoding the vector and scalar instruction set. More detailed description, for the Leon3 modifications and the way that the vector coprocessor is attached to it, is given in Chapter 6.

As mentioned the UNIMP instruction cause an *illegal_instruction* trap. Traps are vectored transfer of program control caused from events that should not occur during normal program execution. Traps can be induced by an exception related to an instruction or by an external interrupt. If a defined trap condition occurs, the system trap handler is invoked to handle the program interruption through a special trap table. The base address is defined in the trap base register (TBR) and the displacement within the table is calculated in combination with the trap ID. There are three trap categories: the precise trap that is caused from a particular instruction and takes place before any program-visible state is altered; the deferred trap that is like the precise one but occurs after the program-visible state changes and the interrupting trap that is induced by an external interrupt request. The default trap model that is implemented in Leon3 comprises precise traps apart from the FPU or coprocessor traps and the "Non-resumable machine-check" exceptions. The table that contains the 3-bit field (op2) that encode the format 2 instructions is shown in Table 5-11.

Table 5-11: Enhanced op2 Encoding (Format 2)

Op2	Instructions	Description
0	UNIMP	Vector Processor Instruction
1	unimplemented	unimplemented
2	Bicc	Branch on Integer Condition Codes
3	unimplemented	unimplemented
4	SETHI	Set High 22 bits of an *r* register instruction
5	unimplemented	unimplemented
6	FBfcc	Branch on Floating-point Condition Codes
7	CBfcc	Branch on Coprocessor Condition Codes

5.5 Overall System Architecture

The vector coprocessor microarchitecture is currently being implemented in RTL VHDL as a tightly coupled coprocessor for the Leon Sparc-V8 CPU. It has private vector and scalar register files as this method promises significantly better performance. Detailed microarchitecture analysis followed by trial synthesis confirmed that all instructions can fit in a single high frequency cycle resulting in a latency of 1 and an initiation rate of 1. Exceptions to this are the Multiply-add/subtract instructions and the short divide with latency/initiation rate of 2/1 and 17/17 respectively. In particular, it was decided that due to the very low improvement, the iterative divider block would not be utilized [6]. The overall system architecture is depicted in Figure 5-8.

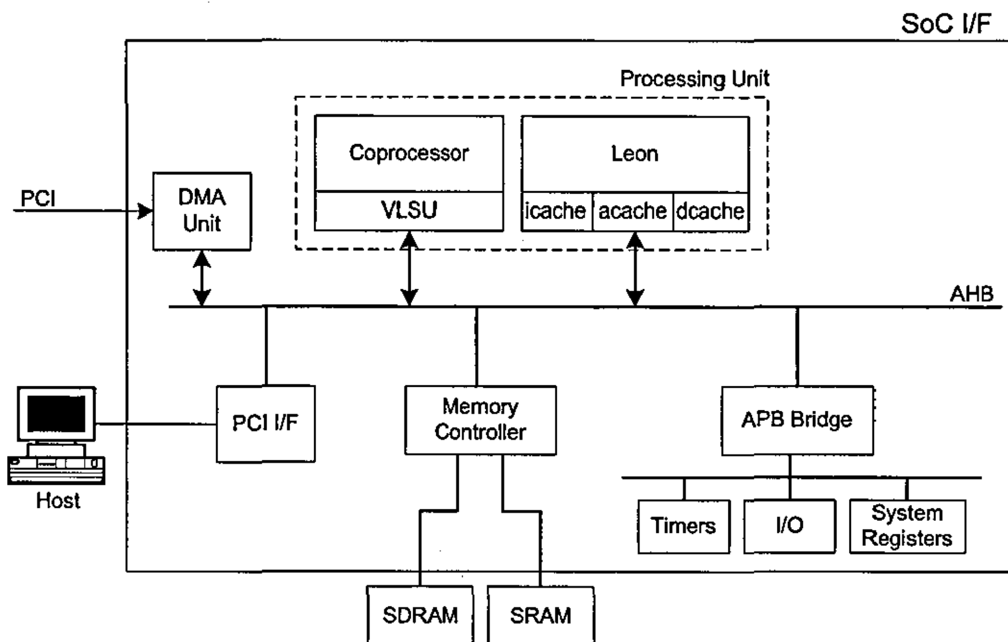


Figure 5-8: Overall system architecture

It consists of the backbone interconnect (32-bit AHB bus), a configurable number of processor-coprocessor units, a DMA (Direct Memory Access) unit, a PCI I/F (Peripheral Component Interconnect Interface), the external memory controller a low-speed (non-streaming) peripheral bus (APB) subsystem which houses miscellaneous units such as timers, interrupt controllers, I/O and memory-mapped registers.

5.5.1 Processor-coprocessor programmable unit

The main processing unit is the vector processor (Leon3/vector coprocessor combination). This unit has two AHB taps, one used for refilling the scalar processor caches (Instruction, Data) and the second for refilling the coprocessor data cache. Both main processor and coprocessor caches remain consistent via i) using a write-through configuration and ii) uses a write-invalidate mechanism which ensures that writes to a cache block from either processor invalidates the same block in the other processor. Thus the latter processor will have to go to the main memory if it accesses that location and recover the up-to-date contents instead of using its own stale data.

5.5.2 DMA taps

These are the input ports to the SoC. An external agent requests the DMA unit for transferring PCM (frames) data into the SoC address space. The DMA unit has AHB mastering capability and is also used to transfer the compressed bitstream (processed frames) from the SoC address space to the environment.

5.5.3 PCI I/F

An Opencores [7] PCI I/F is used to transfer data between the host system (host PC) and the FPGA board.

5.5.4 External Memory Controller

This unit is responsible for all memory accesses in the SoC addresses space. It directly interfaces to a 133MHz DDR (Double Data Rate) memory component and a standard asynchronous RAM component. These external memories are address-range enabled (0x60000000 for SDRAM, 0x40000000 SRAM). The optimized speech coder and the frames to be processed are transferred with DMA from the host PC to the SDRAM memory of the RISC/Coprocessor FPGA board. After that, the RISC CPU/coprocessor combination processes the frames and stores the compressed frames in local memory (SDRAM). The compressed frames are transferred back to the PC memory for comparison with the ITU-T test vectors [6].

5.5.5 APB Subsystem

The final subsystem includes all non-streaming components (internal and external) such as timers, I/O ports, interrupt controllers and UARTS. This subsystem also houses memory mapped registers.

5.6 Summary

This chapter introduced the architectural state and programmer's model of the vector processor. The vector and scalar instruction extensions were presented, divided into groups of instructions of similar types. Every type was detailed by showing assembly formats and giving a short description of the instruction's operation. More details of the instructions are contained in Appendix A. Finally a description for the overall system architecture was given.

5.7 References

[1] K. Koutsomyti, S. R. Parr, V. A. Chouliaras, J. Nunez, D. J. Mulvaney, and S. Data, "Scalar and parametric vector accelerators for the G.729A speech coding standards," in *Proceedings of IEE/ACM SoC Design, Test and Technology Postgraduate Seminar*, Loughborough University, September 2004, pp. 53-57.

[2] D. Martin, "Vector Extensions to the MIPS-IV Instruction Set Architecture (The VIRAM Architecture Manual) Revision 3.7.5.," March 2000.

[3] C. Kozyrakis, "Scalable Vector Media-processors for Embedded Systems," in *Computer Science* University of California: Berkeley, 2002.

[4] "GRLIB IP Core User's Manual, Version 1.0.7," Gaisler Research February 2006.

[5] "The Sparc Architecture Manual Version 8 ", www.sparc.com.

[6] V. A. Chouliaras and J. L. Nunez, "Scalar Coprocessors for accelerating the G723.1 and G729A Speech Coders," *IEEE Transactions on Consumer Electronics,* vol. 49, pp. 703-710, August 2003.

[7] http://www.opencores.org/.

CHAPTER 6
VECTOR PROCESSOR IMPLEMENTATION

6.1 Overview

This chapter describes the vector processor along with a number of implementation details and the general principles of its operation. In addition, it details the way that the vector speech coprocessor is attached to the main Leon3 scalar processor. The vector processor consists of the *Vector Datapath* (VDP) and the *Vector Load/Store Unit* (VLSU). In the sections that follow only the Vector Datapath is discussed in detail as the VLSU is addressed as part of another thesis [1]. The vector processor fully implements the Vector and Scalar ISAs that were described in the previous chapter. The vector pipeline comprises four-stage pipeline: the *Vector Decode Stage* (VDEC), the *Vector Register Access Stage* (VREG) and the *Vector Datapath Stage* (VDP) which consists of a two stage pipeline (VDP1 and VDP2). All vector/scalar instructions are fully-pipelined with a latency of one and an initiation rate of one instruction per cycle, with the exception of multiply-add and multiply-sub instructions which have a latency of two cycles and an initiation rate of one.

The organization of the speech coprocessor with the 4-stage pipeline is depicted in Figure 6-1. The vector coprocessor is parameterised along both the architecture and the microarchitecture axes. The architectural parameterisation refers to the number of registers including accumulators and the extensible vector ISA. The microarchitectural parameterisation refers to the extensible, non programmer visible state of the processor. This includes the number of scalar datapaths (functional units), maximum data width and internal flop-based state. This parameterisation is defined from a number of compile-time parameters that specify the various architectural and microarchitectural characteristics of the coprocessor.

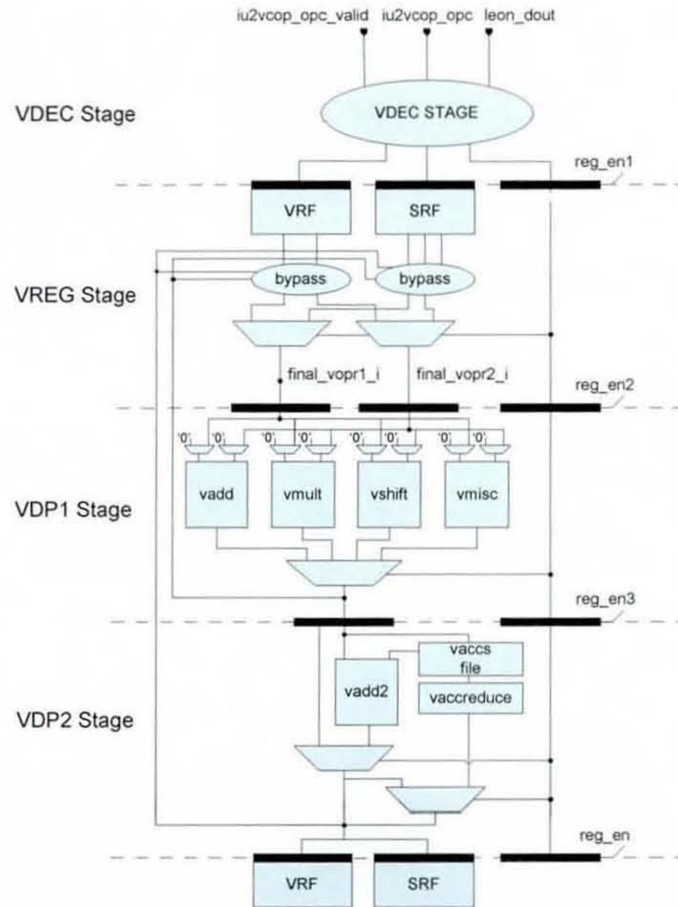


Figure 6-1: The vector speech coprocessor microarchitecture with the four-stage pipeline: Vector Decode Stage (VDEC), Vector Register Access Stage (VREG) and two stages for the Vector Datapath Stage (VDP1 and VDP2)

The choice of compile-time configuration puts the combined processor/vector coprocessor firmly in the domain of configurable, extensible CPUs. The compile-time parameters are listed in Table 6-1.

This table indicates the valid values and the maximum number of the vector/scalar registers, the accumulators and the vector units (VLMAX/2). Exceeding these limits or choosing other values than the valid will generate errors during the RTL simulation.

Table 6-1: Compile-time vector processor parameters for its architectural and microarchitectural state that are contained in gxx_config.vhd file

Parameter	Allowed range	Default	Description
VLMAX	2,4, 8, 16,32,64,128	2	Maximum Vector Length
VREGS	4, 8, 16, 32	16	Number of Vector Registers
SREGS	4, 8, 16, 32	16	Number of Scalar Registers
ACC_NUMBER	2, 4, 8, 16, 32	2	Number of Vector Accumulator
ACC_WIDTH	(VLMAX/2)*32	(VLMAX/2)*32	Width of Vector Accumulator

The code is parameterised as to target a number of technologies easily. This has been achieved through the use of fully technology independent VHDL constructs as well as using generic RAM components. The allowed silicon technologies are listed in Table 6-2.

Table 6-2: The allowed silicon technologies that are used for synthesis and place and route contained in gxx_config.vhd file

Parameter	Description
GEN	Technology independent RAM macros
XST	Xiling FPGA Technology (Spartan3)
TSMC018	Taiwan Semiconductor Manufacturing Company (TSMC) 0.18μm standard-cell technology
TSMC013	TSMC 0.13μm standard-cell technology

6.2 Vector Decode Stage (VDEC)

This is the first stage of the pipelined vector coprocessor datapath. In this stage the instruction from the Leon3 opcode register is decoded and all the datapath control signals for the following pipeline stages are produced. The instruction for the decoding is coming pipelined from the Decode stage of the Leon3 to the Memory stage where the coprocessor is attached along with few control signals. More specifically in this stage the following operations are performed:

- The opcode is decoded and control signals are produced ready to be pipelined in subsequent stages.
- The addresses for the source and destination register operands are produced and access of the vector and the scalar register files starts (split over two stages).
- The write enables for all the pipeline registers of the vector pipeline are produced.

The electrical interface of the VDEC stage is depicted in Figure 6-2. The input signals are coming from the Leon3 processor and vector load/store unit (VLSU). The output signals that are of `vdec2vregs` type are going to the input of the VREG stage.

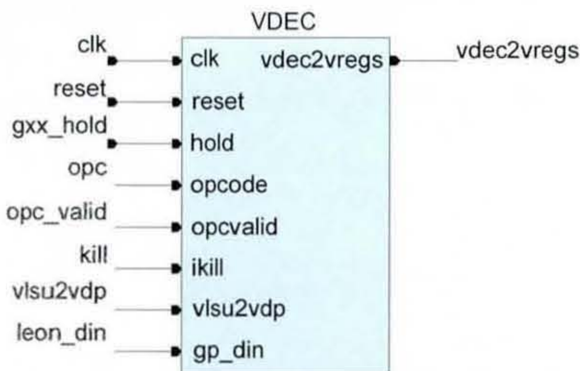


Figure 6-2: The electrical interface of the VDEC Stage

As mentioned in the previous chapter, the selected instruction format for the vector processor is included in the Unimplemented Instruction [2] of the Sparc V8 architecture and it is depicted in Figure 6-3. This instruction is architecturally not implemented and generates an exception if encountered.

In the Leon3 the *const22* bitfield is completely ignored by the decoding logic of the processor. Additional combinational logic has been inserted in Leon3 to extract the *const22* field and sent it to the vector coprocessor decode unit as the input vector opcode.

Figure 6-3: The Unimplemented instruction format of the Sparc V8 architecture

In the decode stage of the coprocessor the opcode-valid signal is asserted if the 22 bits are a valid vector instruction and datapath control signals, addresses for the vector/scalar register operands and enables are produced. In the case of a 3-address format (Figure 6-4) the extracted addresses fields along with the produced read enable signals are used to access the synchronous register file, in parallel with the decoding of the latched instruction. In this way, the depth of the pipeline of the coprocessor is reduced by one stage compared to a purely cascade decode/register access organisation and this has an

additional beneficial effect during the transfer of data from the coprocessor to and from the scalar processor.

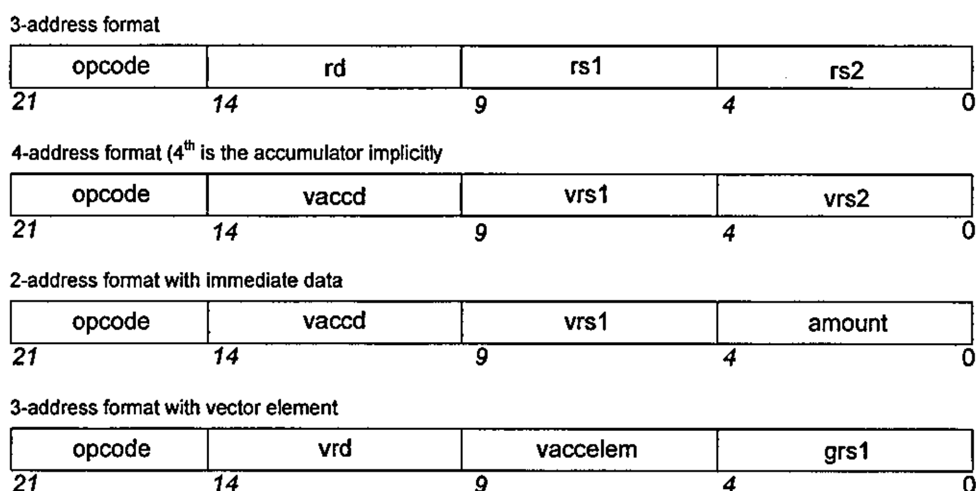


Figure 6-4: Different types of instruction formats of the vector processor ISA

A similar process is performed in the case of multiply-add and multiply-sub instructions that are a 4-address format instructions (the accumulator is an implicit source and destination operand). In this case the accumulator address and read enable are sent during the decoding of the latched instruction in order to obtain the accumulator operand for the next stage. Another difference for these instructions is that they are always implemented in pairs of even and odd elements. A combinational logic (evod16_en) asserts the appropriate read enable bits for the even or odd operands. In the case that one of the source operands is immediate data, this is included in the instruction field [4:0] which is extracted and sent to the next stage where it is zero extended to 32-bits. A detailed description of the extension process will be given in the VREG stage. In the case where one register operand is used to select a vector element (vaccelem) for load or store operations then the instruction field [9:5] is extracted and used to calculate the write or read enables respectively for the specific vector element (16-bit word). The same method is followed in the case of the move from or to Leon3 instructions to or from an element of a vector register. When a move from Leon3 instruction is performed, the operand is coming from the main scalar CPU register file and it is pipelined to the next stage (VREG). At that stage it is selected as a source operand and enters the appropriate lane of the coprocessor vector pipeline to finally commit to either the coprocessor scalar or vector register file. In the case of a move to Leon3 instruction, the selected 16-bit element

of the vector register is zero extended to 32-bits and it is sent to the Leon3 register file. More detail description is given in the VREG stage. The custom instruction formats for the previously mentioned cases are depicted in Figure 6-4. All the datapath data and control signals are latched at the end of the VDEC stage to the set of registers of type vdec2vregs. The pipeline enable (`reg_en1`) of these registers is asserted when the following conditions are true:

- the main CPU is not halted (`holdn='1'`)
- no exception takes place in the main CPU (`ikill='0'`)
- there is no cache miss in VLSU (`vlsu2vdp.hold='0'`)
- the coprocessor instruction is valid (`opcvalid='1'`)

6.3 Vector Registers Stage (VREG)

The Vector Registers Access Stage selects the source operands from the vector/scalar register files or from the accumulator file or from the bypassed results of the first and second stage of the vector datapath. In the latter case, the results are made available, from any of the other downstream stages, to the VREG stage in order to be used as source operands if this is required. This bypassing of intermediate results is established practice in CPU architecture [3] and is the only way to resolve data dependences without stalling the pipeline. As mentioned in Chapter 3, data dependences happen when an instruction needs to use the result of a previous instruction prior to its commit to the register file [4]. In addition, it is the stage where store instruction takes place and the memory address for the load instruction is sent to the VLSU unit in order for load data to be ready and be sent to the second stage of the VDP. Furthermore the vector length (`vlen_r`) register and the overflow and predication (`pred`) registers are updated. The detailed schematic of the VREG stage is illustrated in Figure 6-5.

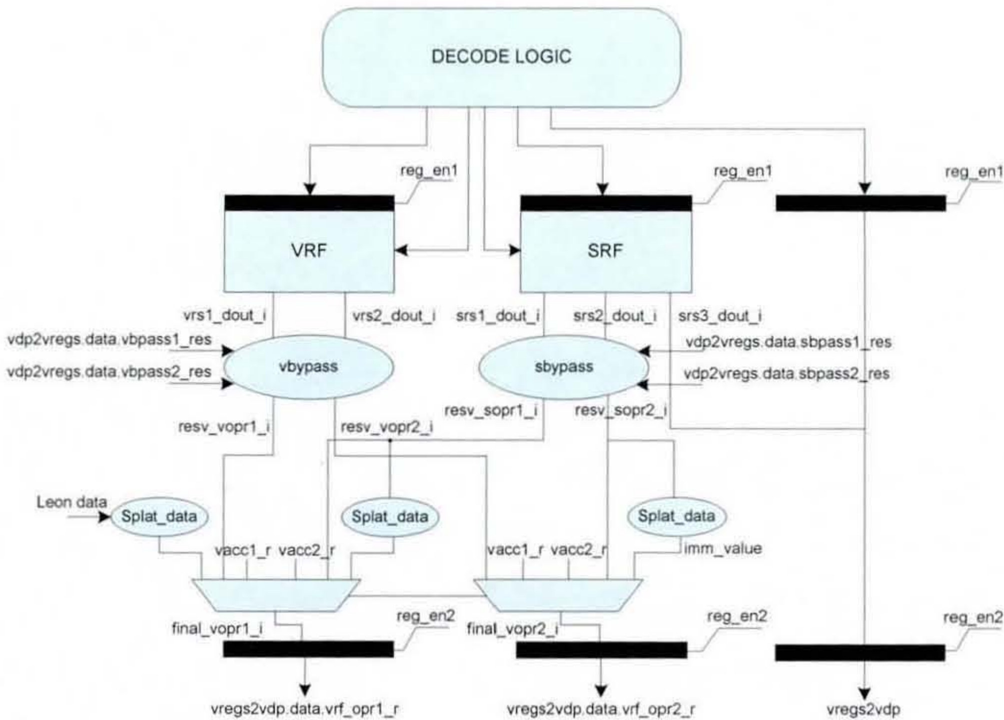


Figure 6-5: Vector Register Access Stage (VREG) microarchitecture

6.3.1 Reverse Data Process

When a load or store instruction with a negative stride is performed a special control signal (`vdec2vregs.lst_neg_r`) is asserted. In the case of a negative stride store (`vstwn`) the data to be written to the memory that comes from the bypass logic of the second register file read port (`resv_vopr2_i`) needs to be reversed.

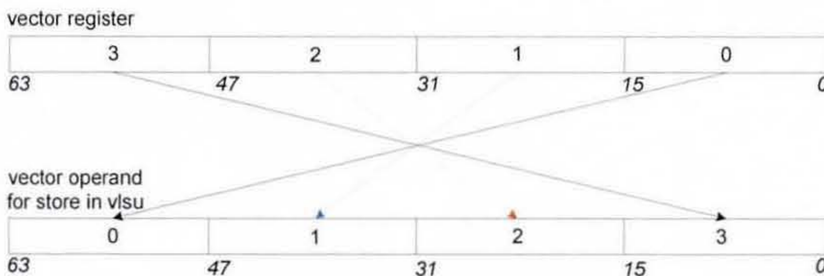


Figure 6-6: Reverse Data Process

This is performed with the use of the reverse data function logic (`reverse_data`) which swaps the order of the elements as they are placed within the final vector, from the most significant element to the least significant element. The output of this function is sent to

the input of the VLSU (sregs2vlsu.data_in) as the data that will be written to the vector data cache. The data-reversing process is shown in Figure 6-6. The reverse process for the load instruction is the same but it is performed in the VDP2 stage of the vector processor. For this reason is described in section 8.4.5.

6.3.2 Splat Data Process

There are instructions that need to replicate a 16-bit (vsplat_h_r) or 32-bit (vsplatacci) scalar value to all the elements of a vector register or accumulator respectively. This "splat" operation is performed in the splat function (splat_data) in the VREG stage. The splat logic takes as inputs a 32-bit word and the width of the vector operand in which the value will be copied. If the value that is to be "splated" is 16-bit, it is duplicated in order to produce the necessary 32-bit value that acts as the 32-bit input of the function. The resulting vector is sent to the multiplexer responsible for the first operand selection in the VREG stage. The schematic for this function is depicted in Figure 6-7.

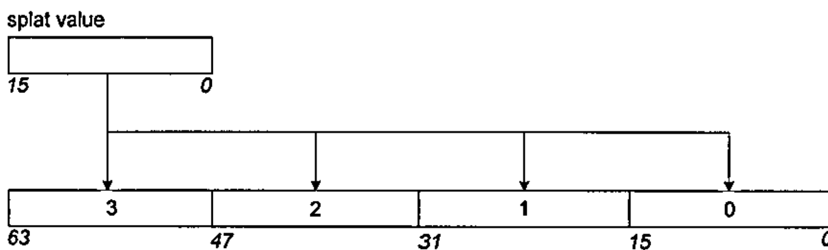


Figure 6-7: Splat Data Process

6.3.3 Masking Process

There are two masking processes that are implemented in the VREG stage: these are the mask_width and the mask_extract. The mask_width logic takes as an input a value that indicates the width of the vector to be processed and produces a mask bit-vector that is VLMAX*16 bits long. The produced mask defines a set of bits that are used as a selector in order to extract the desired scalar elements from the vector that the mask is applied. The input value (width) gives the number of the mask's bits that will be '1' while the remaining bits will be '0'. The functionality of this masking operation is depicted in

Figure 6-8. This type of mask is used in the bypass logic for the selection and formulation of the input operands to the vector ALU stage.

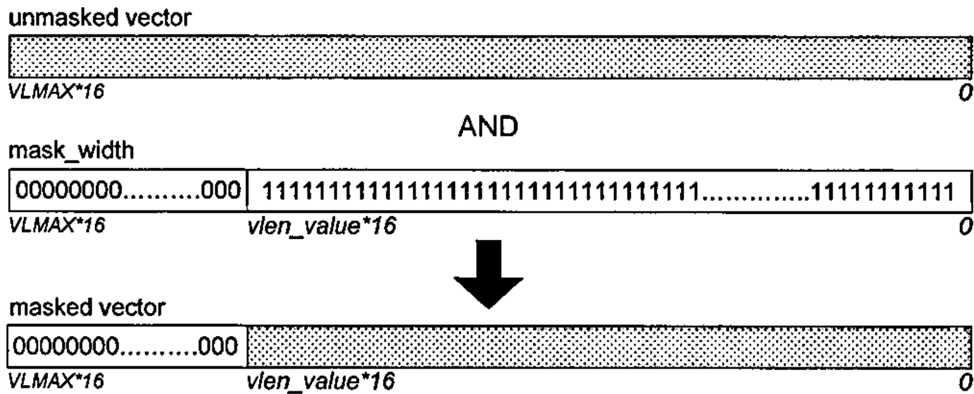


Figure 6-8: Mask width function

The `mask_extract` logic (function) takes as an input a value that indicates which vector element of 32-bits should be selected and produces a mask that is VLMAX*16 long. This second mask comprises sets of '0's and '1's that are structured in a way to extract the desired 32-bits from a given input vector. The input value to this function resembles a "read-enable" that selects the 32-bits element that will be extracted from the vector in which the mask is applied. The `mask_extract` functionality is illustrated in Figure 6-9.

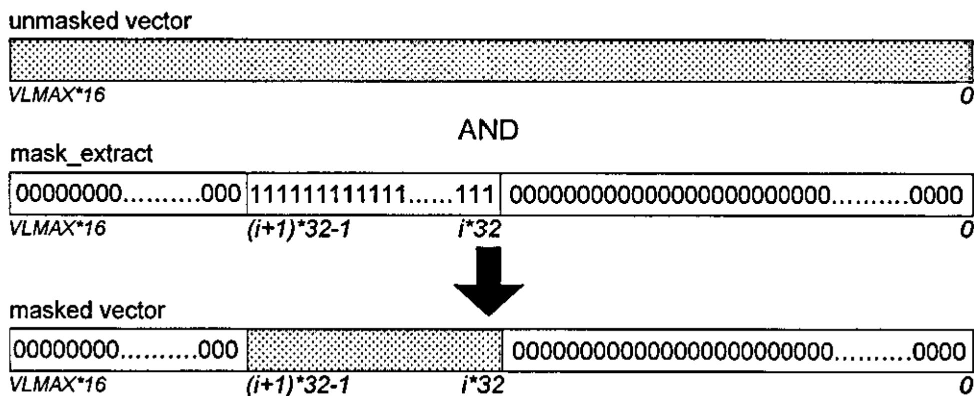


Figure 6-9: Mask extract function

This type of mask is used to select a scalar element from an accumulator register for load or store operations.

6.3.4 Bypass process

The bypass process is critical for the efficient operation of pipelined processors. In the VREG stage it selects the operands from the vector/scalar register files, the vector accumulators or the intermediate results produced in the vector datapath (before they are written to the register files) from the first and second stages of the VDP. There are actually two bypass processes: the vector_bypass and the scalar_bypass. In the vector_bypass process, the two vector-operand read addresses (`vdec2vregs.vrs1_rdaddr_a`, `vdec2vregs.vrs2_rdaddr_a`) for the vector register file are compared respectively with the write address (`vdp2vregs.ctrl.vbpass1_vwr_addr_r`) of the instruction currently executing at the first VDP stage (VDP1). If either of them is equal with the VDP write-back address and the valid signal (`vdp2vregs.ctrl.vbpass1_valid`) of the bypass result is asserted, the vector length of the bypassed result (`vdp2vregs.data.vlen_cvalue_r`) is compared with the current (architected) vector length (`vlen_r`) of the coprocessor that is located in the vlen register. In the case that the result from the VDP1 stage has a vector length smaller than the vector length of the resolved operand, then the bits from 0 to vdp2vregs.data.vlen_cvalue_r*16-1 are containing in the bypassed result (`vdp2vregs.data.vbpass1_res`) while the remaining bits up the vlen_r*16 are filled with the outputs of the corresponding read ports (`vrs1_dout_i` or `vrs2_dout_i`) of the vector register file. If the vector length of the bypassed result is larger then the operand's (`resv_vopr1_i` or `resv_vopr2_i`) bits are filled with one of the outputs of the read ports (`vrs1_dout_i` or `vrs2_dout_i`) from bits 0 to vlen_r*16-1. In the case that both vector lengths are equal, the resolved operand comprises the bypassed result of the first VDP stage. The same process is followed for the bypassed result of the second VDP stage (VDP2) and both read ports of the vector register file in the case where there is a mismatch in the target register of the first VDP stage and the source register in VREGS in order the appropriate operands to be selected. The formulation of the resolved operands is always performed with the use of the masking process (`mask_width`). The schematic for the vector bypass process for one of the vector source operands and the intermediate result of one of the two VDP stages is illustrated in Figure 6-10.

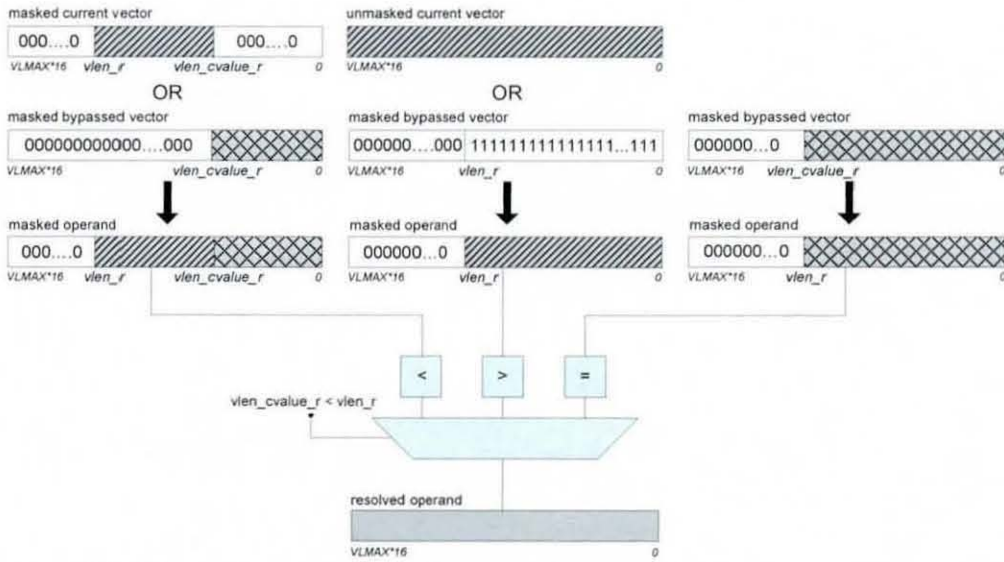


Figure 6-10: Vector bypass process for one of the vector source operands and the intermediate result of one of the two VDP stages

The scalar bypass process is much simpler than the vector bypass as there is no need for masking of the operands. The three scalar read addresses (`srs1_rdaddr_a`, `srs2_rdaddr_a` and `srs3_rdaddr_a`) for the scalar register file are compared respectively with the write address of the bypassed scalar result (`vdp2vregs.ctrl.sbpass1_swr_addr_r`) of the first VDP stage. If they are the same and the valid signal (`vdp2vregs.ctrl.sbpass1_valid`) of the result is asserted, the corresponding resolved operand (`resv_sopr1_i`, `resv_sopr2_i`, `resv_sopr3_i`) is assigned from the scalar bypassed result (`vdp2vregs.data.sbpass1_res`) else with the output of the corresponding read port of the scalar register file (`srs1_dout_i` or `srs2_dout_i` or `srs3_dout_i`). The same process is followed for the bypassed scalar result of the second VDP stage and the three read ports of the scalar register file. Figure 6-11 depicts the scalar bypass process for one of the scalar operands.

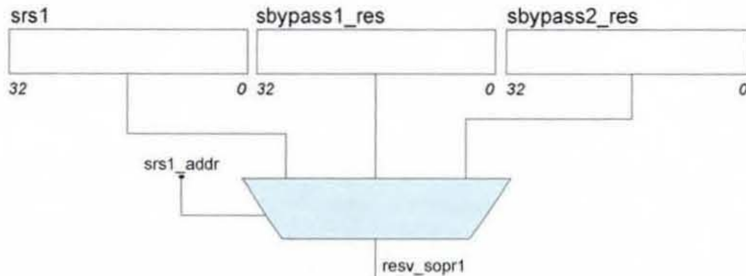


Figure 6-11: Scalar bypass process for the selection of one of the scalar operands (first)

6.3.5 Operands Selection

The two source operands (vector or scalar) are selected after the bypass process, prior to the end of the VREG stage and committed to the output registers of vregs2vdp type. In the case of a move-from-coprocessor instruction, the requested 32-bit data from the Leon3 are extracted from the selected first source operand prior to committing and sent back to the main CPU write-back stage via the coprocessor-CPU custom interface. The third operand, that is always scalar, is driven directly from the output of the third read port (srs3_dout_i) of the scalar register file to the corresponding output register. The selection of the two operands is performed via two large multiplexers as they depicted in the detailed schematic in Figure 6-5. The first operand (final_vopr1_i) can be the 16-bit (vdec2vregs.sel_width_r = '1') or 32-bit output of the scalar bypass process (resv_sopr1_i) or the output of the vector bypass process (resv_vopr1_i). It can also be one of the vector accumulator file hardwired read ports (vdp2vregs.data.vacc1_r or vdp2vregs.data.vacc2_r) or the Leon3 general purpose registers (gpdata) or the "splated" data formulated from the splash function using data from a scalar register. Similarly, the second operand can come from the output (16-bit or 32-bit) of the scalar bypass process (resv_sopr2_i) or the output of the vector bypass process (resv_vopr2_i) or one of the hardwired read ports of the accumulator file or the immediate data (imm_value) that have been extracted from the coprocessor instruction at the decode stage.

6.3.6 Register enable

The register enable (reg_en2) for the output registers of the VREG stage is asserted when both hold signals that are coming from the Leon3 (hold) and the VLSU unit (vlsu2vdp.hold) are not asserted. In addition, the pipelined register enable (vdec2vregs.reg_en2_r) should be asserted and the signal vdec2vregs.sel_st_r must be set to zero in order to prevent any store instruction from taking place. The later condition is necessary because the store instruction is performed and completed at the VREG stage so the next stages are not used for this instruction. Therefore, when the store is completed no change in the state of the following datapath stages flip flops should take place in order to avoid unnecessary power consumption. The reg_en2 is the pipeline

enable of the output registers (`vreg2vdp`) in which all the datapath data and control signals of the VREG stage are latched.

6.3.7 Vector Register File (gxx_vreg_file)

The coprocessor stores the results of the vector computations in a vector register file that is a two-dimensional storage array where every row holds all the scalar elements of a single vector. The vector register file is parametric so its dimensions are specified from compile-time parameters in both axes. The width is defined from the number of the vector elements (16-bits each) and is equal to the maximum vector length (VLMAX) while the number of such entries is VREGS, equal to the architectural vector registers. The vector register file provides two read ports and one write port that translates to two vector read and one vector write operations per cycle.

6.3.7.1 Parameterisation

The vector register file is fully-configurable design. The number of register windows (VREGS) is within the range of 2 to 32, with a default setting of 16. These parameters are specified in gxx_config.vhd and are shown in Table 6-3:

Table 6-3: Compile-time vector register file parameters for its architectural and microarchitectural state that are contained in gxx_config.vhd file

Parameter	Default
VLMAX	2, 4, 8, 16, 32, 64, 128
VREGS	16
Technology	GEN, TSMC013

6.3.7.2 The vector register file implementation

The electrical interface of the vector register file is shown in Figure 6-12.

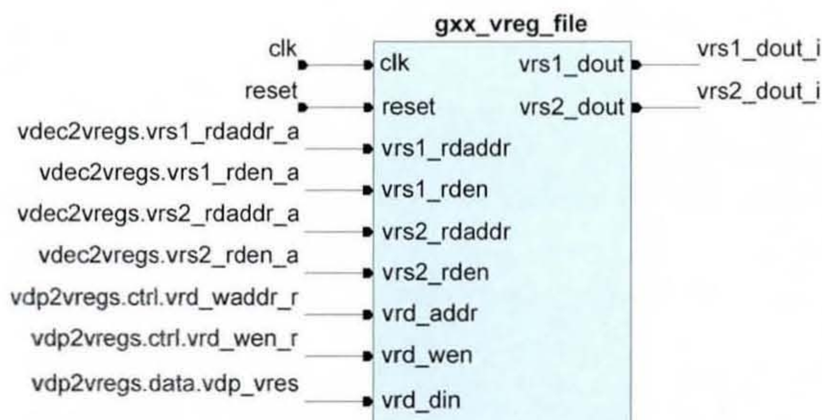


Figure 6-12: Electrical Interface of Vector Register File

It has two read address ports (`vdec2vregs.vrs1_rdaddr_a`, `vdec2vregs.vrs2_rdaddr_a`) that are driven unlatched from the vector decode stage in parallel with the decoding of the latched opcode, in order to initiate the register file access which in turn, will return the operands before the end of the VREG stage. The write address port (`vdp2vregs.ctrl.vrd_waddr_r`) is coming pipelined from the end of the second stage of the VDP in order to commit the vector result. The register file is technology-independent and allows two reads and one write to be performed on the same cycle. In the case where a read of a register is required at the same cycle that it is written, a R/W conflict occurs. When this condition is detected the read-port is disabled and the data are bypassed from the write-port write-data. This ensures that the memory cell does not get corrupted when doing a simultaneous R/W operation at the same address. This behaviour has been observed in the TSMC 0.13μm dual-port RAMs and the above solution ensures that this extreme case never causes corruption of data. Figure 6-13 details the organisation of the vector register file with R/W conflict avoidance. This performed by the `conflict_process` logic in which each read address is compared with the incoming write address and if are the same and any of the bits of the read or write enable signals are asserted then a conflict signal is produced. Because there are two read ports there are two conflict signals (`conflict1_i`, `conflict2_i`) and when one of them is asserted the output data are coming from the write-port data via the output multiplexer.

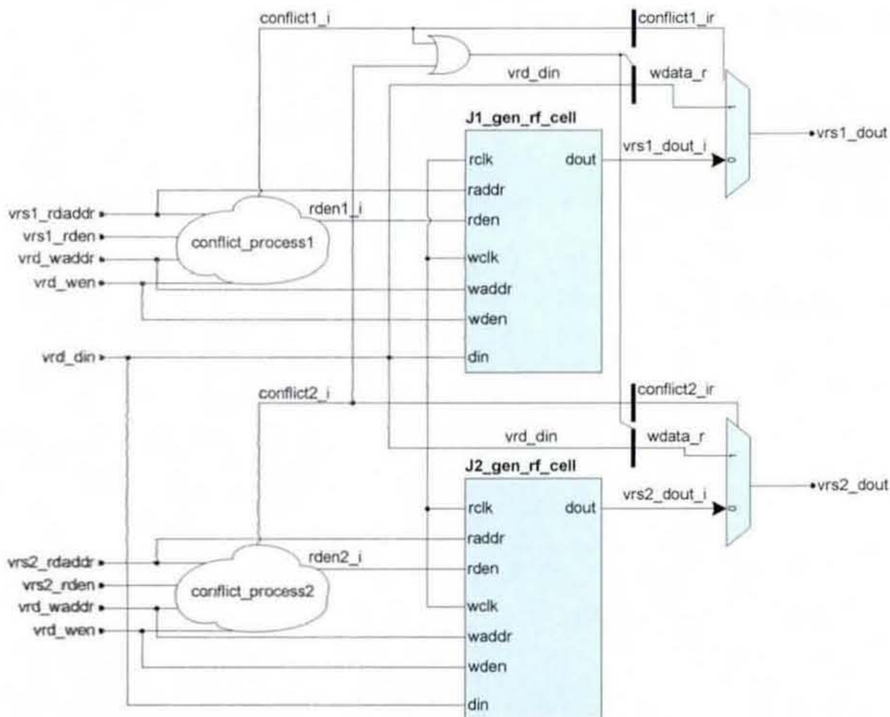


Figure 6-13: Detailed microarchitecture of the Vector Register File with R/W conflict avoidance

In addition, there are two read enable ports (`vdec2vregs.vrs1_rden_a`, `vdec2vregs.vrs2_rden_a`) that are coming from the decode stage at the same time as the read addresses. The read enable signal is a bit vector in which every bit enables the read operation at byte-granularity from the selected register. In this specific case every 2 bits of the read enable signal correspond to a 16-bit element from the source vector register. When the read addresses are valid and the read enables are set to '1', the corresponded data are read and sent to the outputs of the register file (`vrs1_dout_i`, `vrs2_dout_i`). The write enable strobes (`vdp2vregs.ctrl.vrd_wen_r`) arrive pipelined from the end of the VDP2 stage and are based in the same principle as the read enable strobes. When the write address is valid and the write enable is asserted the input data (`vdp2vregs.data.vdp_vres`) are written to the selected register.

6.3.8 Scalar Register File (gxx_sreg_file)

The scalar operands are stored to the scalar register file that is again a two-dimensional storage array. It contains sixteen registers of 32-bit width and it supports three reads and one write operations per cycle.

6.3.8.1 Parameterisation

The scalar register file has SREGS registers that can be in the range of 2 to 32 and with default setting of 16. The compile-time parameters with their default values that specify the structure of the scalar register are shown in Table 6-4:

Table 6-4: Compile-time scalar register file parameters for its architectural state that are contained in gxx_config.vhd file

Parameter	Default
SREGS	16
Technology	GEN, TSMC013

6.3.8.2 Scalar register file implementation

The electrical interface of the scalar register file is depicted in Figure 6-14.

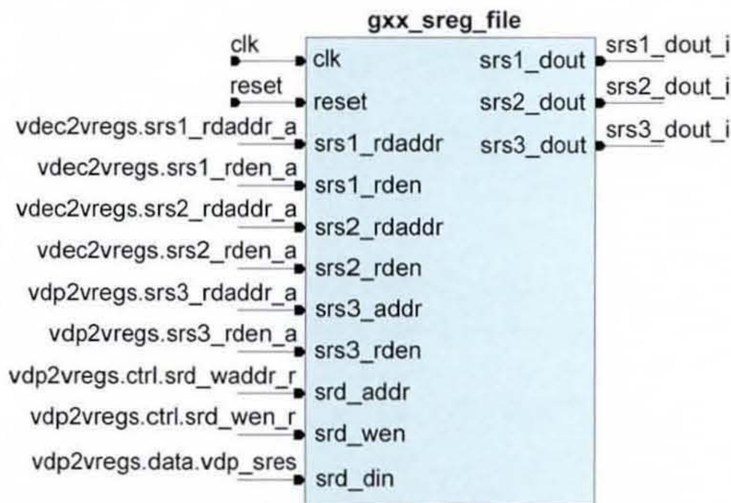


Figure 6-14: Electrical Interface of Scalar Register File

The scalar register file has three read address ports (vdec2vregs.srs1_rdaddr_a, vdec2vregs.srs2_rdaddr_a, vdec2vregs.srs3_rdaddr_a) and three read enable ports (vdec2vregs.srs1_rden, vdec2vregs.srs2_rden, vdec2vregs.srs3_rden) that are coming

unlatched from the vector decode stage (VDEC). This is happening in order the scalar register file to be accessed during the decoding and produce the scalar operands before the end of the VREG stage in time for bypassing. It was decided to attach an additional read port to the scalar register file as a third operand was needed to play the role of the accumulator for the multiply-add/sub instructions.

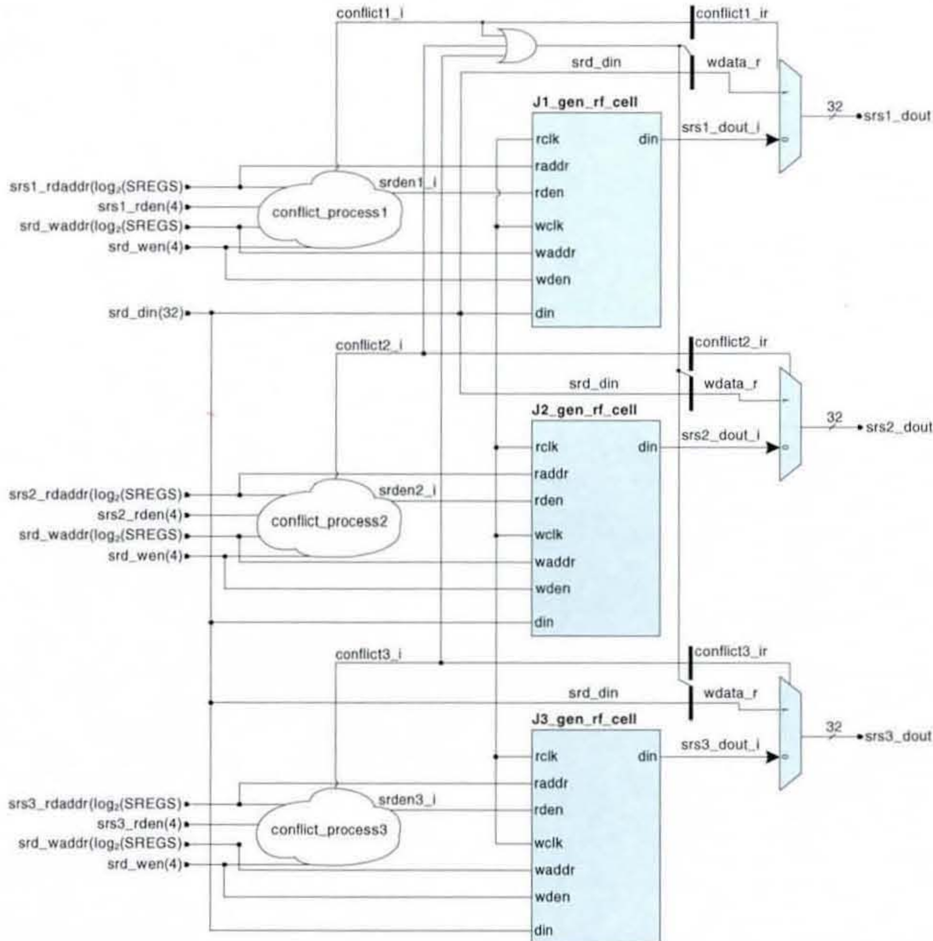


Figure 6-15: Detailed microarchitecture of the Scalar Register File with R/W conflict avoidance

The write address (vdp2vregs.ctrl.srd_waddr_r) and the write enable (vdp2vregs.ctrl.srd_wen_r) are coming pipelined from the end of the VDP2 stage. The scalar register file is described in a technology-independent way and supports three reads and one write operations per cycle. R/W conflict avoidance happens with three conflict signals (conflict1_i, conflict2_i, conflict3_i) that are produced from the conflict_process 1, 2, 3 in order to prevent a read and write operation to happen simultaneously to the

same register address. In this case the particular read-port is disabled and the data are coming instead bypassed from the write-port data. The detailed schematic is depicted in Figure 6-15.

6.3.9 Vlen register

The degree of data-level parallelism that the vector coprocessor can exploit on every cycle is defined by the vector length register (`vlen_r`). This control register stores the value of the dynamic vector length that determines the number of the 16-bit elements in which the vector operations will be performed. For example, a vector short addition with vector length of four will only add the first four pairs (4x16-bit) of elements of the input vector registers and will ignore the rest. The value of the vector length is stored to `vlen_r` register before any other instruction takes place in order to reconfigure the hardware. The vector length can take any value that is multiple of two; 2, 4, 16, 32, 64 up to the maximum vector length (VLMAX) that in this case is 128 (2048 bits). If the data-level parallelism in a particular loop of the speech algorithm, which corresponds to the number of times a loop body is executed, is greater than the VLMAX then the `vlen_r` is loaded with the maximum value and performs a sequence of identical operations that comprise the loop. At the end the `vlen_r` is loaded with the remaining of the modulus division of the number of repetitions of the loop with the VLMAX (loop strip mining) and one more iteration of identical operations is performed but this time with a shorter-than-VLMAX vector length. The instruction that is responsible for loading the `vlen_r` register with a value for vector length is `ldvlen_r(value)`. When this instruction is encountered, the value is extracted from the instruction opcode and the `vlen_r` write-enable (`vdec_i.vlen_wen`) is set at the decode stage. Subsequently they are latched to the VREG stage as `vdec2vregs.vlen_nvalue_r` and `vdec2vregs.vlen_wen_r` respectively and pipelined to the following stages of the vector datapath coprocessor. At the VREG stage the pipelined write-enable (`vdp2vregs.ctrl.vlen_wen_r`) is checked and if asserted the pipelined value of the vector length (`vdp2vregs.data.vlen_nvalue_r`) from the VDP2 stage is committed to `vlen_r`. In every cycle the `vlen_r` is read and the current value (`vregs2vdp.data.vlen_cvalue_r`) is pipelined to next stages to dynamically reconfigure the vector pipeline [5].

6.3.10 Overflow and Pred Flags

When an arithmetic instruction produces a result that is greater than the value a register can store or represent then an overflow bit is asserted and written to the overflow flag. The overflow flag is set to indicate a problem so the software can be aware of this condition and act accordingly to compensate or mitigate the error. More specifically, both ITU-T speech coding algorithms that execute on the coprocessor deal with this problem by using the saturation instruction that limits the output to the allowed range for 16-bit or 32-bit numbers. The coprocessor has a vector overflow register (`ovf`) and an overflow flag (`VV`). The vector overflow register (`ovf`) is VLMAX/2 bits long, one overflow bit per 32-bits of the vector length, and it is updated at the VREG stage when the overflow enable (`vdp2vregs.ctrl.ovf_wen`) that is forwarded from the end of the second stage of VDP is set. In this case, the vector overflow register takes the new value (`vdp2vregs.data.ovf_r`) that is coming pipelined from the VDP2 stage. The overflow flag (`VV`) is 1-bit and it changes when the same overflow enable as before is asserted. The new value of the overflow flag is the or-reduce result of the vector overflow value (`vdp2vregs.data.ovf_r`). In the case of a vector comparison instruction the predicate bits are set according to the result of the comparison and written to the `pred` flag. The comparison is performed on pairs of vector operand elements and every produced predicate bit corresponds to a 16-bit comparison. The `pred` register is VLMAX bits long. The `pred` register is updated at the VREG stage when the predicate write-enable (`vdp2vregs.ctrl.pred_en`) that is coming pipelined from the end of the VDP2 stage is set and the `pred` register can take the result predicate bits (`vdp2vregs.data.pred_r`) from the comparison.

6.4 Vector Load/Store Unit (gxx_vlsu)

At the VREG stage the Vector Load Store Unit (VLSU) is accessed and the load/store instruction along with the store data (in the case of store) and the control information are sent from the vector coprocessor to the former. The electrical interface of the VLSU is illustrated in Figure 6-16.

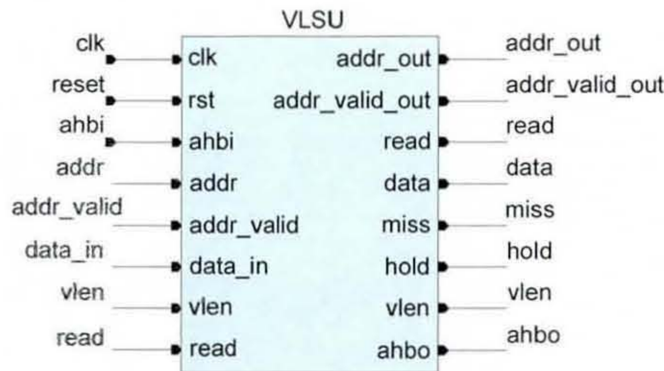
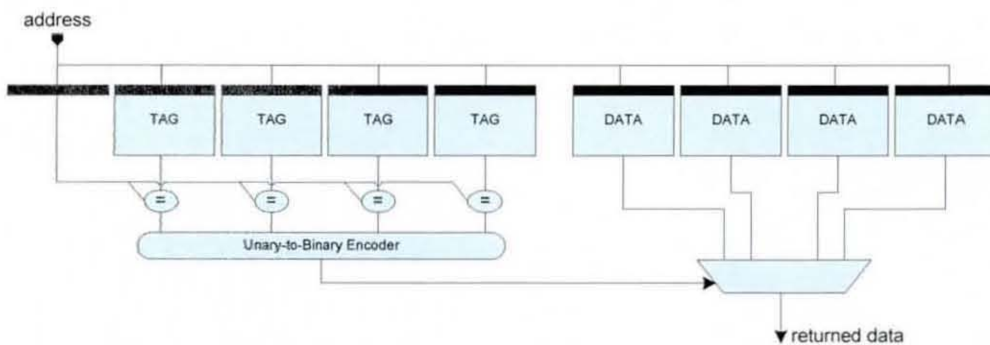


Figure 6-16: VLSU Electrical Interface

In the case of a load instruction (`sregs2vlsu.read='1'`), the VLSU takes the read address (`sregs2vlsu.addr`) for the memory along with the valid signal (`sregs2vlsu.addr_valid`) and the vector length (`sregs2vlsu.vlen`) that determines the width of the vector data in order to prepare that vector and return it to the second stage of VDP. When a store instruction (`sregs2vlsu.read='0'`) is performed, again the address with the control signals are sent from the VREG stage to the VLSU along with the data for storing (`sregs2vlsu.data_in`). The VLSU has a cascade TAG/DATA configuration resulting in one latent Load-Use cycle through the bypass logic of the vector coprocessor. This means that the TAG array is checked one cycle before accessing the DATA array, on the following cycle, resulting in the load data being ready at the second stage of the VDP. Even though this configuration results in increased latency than the more traditional parallel TAG/DATA organization, it leads to substantially lower power consumption; in a multi-way configuration, all TAG and one (selected) DATA arrays are powered up in consecutive cycles whereas in the parallel TAG/DATA case, all TAG and all DATA arrays are powered up concurrently, resulting in higher power consumption. Whereas in the cascade TAG/DATA configuration all TAG RAMs are power-up during cycle1 but only the selected way of the DATA RAM is powered up on cycle 2.

Parallel TAG/DATA Configuration Cache



Cascade TAG/DATA Configuration Cache

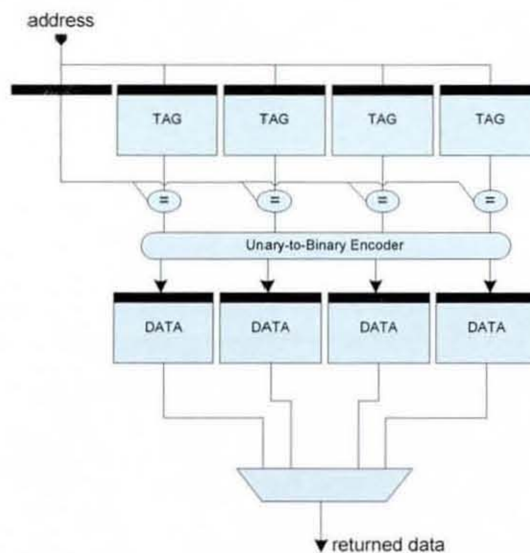


Figure 6-17: Parallel TAG/DATA configuration and Cascade TAG/DATA configuration caches

For example, a cascade 4-way set associative data cache has four TAG RAMs (cycle 1) and one DATA RAM (cycle 2) powered up while a parallel data cache will have four TAG RAMs and four DATA RAMs that makes eight RAMs in total powered up. Figure 6-17 depicts parallel TAG/DATA configuration and cascade TAG/DATA configurations caches.

Figure 6-18: Microarchitecture of VLSU in cascade TAG/DATA configuration

The microarchitecture of the VLSU is parameterised and is depicted in Figure 6-18. There are a number of compile-time parameters that specify the number of ways and the size of each cache and are defined in the coprocessor configuration file (gxx_config.vhd). In addition, there is a Finite-State-Machine (FSM) that handles the communication with the AMBA bus [6].

6.5 Vector Datapath Stage (VDP)

The Vector Datapath Stage forms the execution core of the vector coprocessor and is the most complicated piece of logic. The VDP is divided in two stages: Stage one performs all the arithmetic, shift and miscellaneous vector operations that are of single cycle latency and the multiplication part of the multiply-add and multiply-sub instructions.

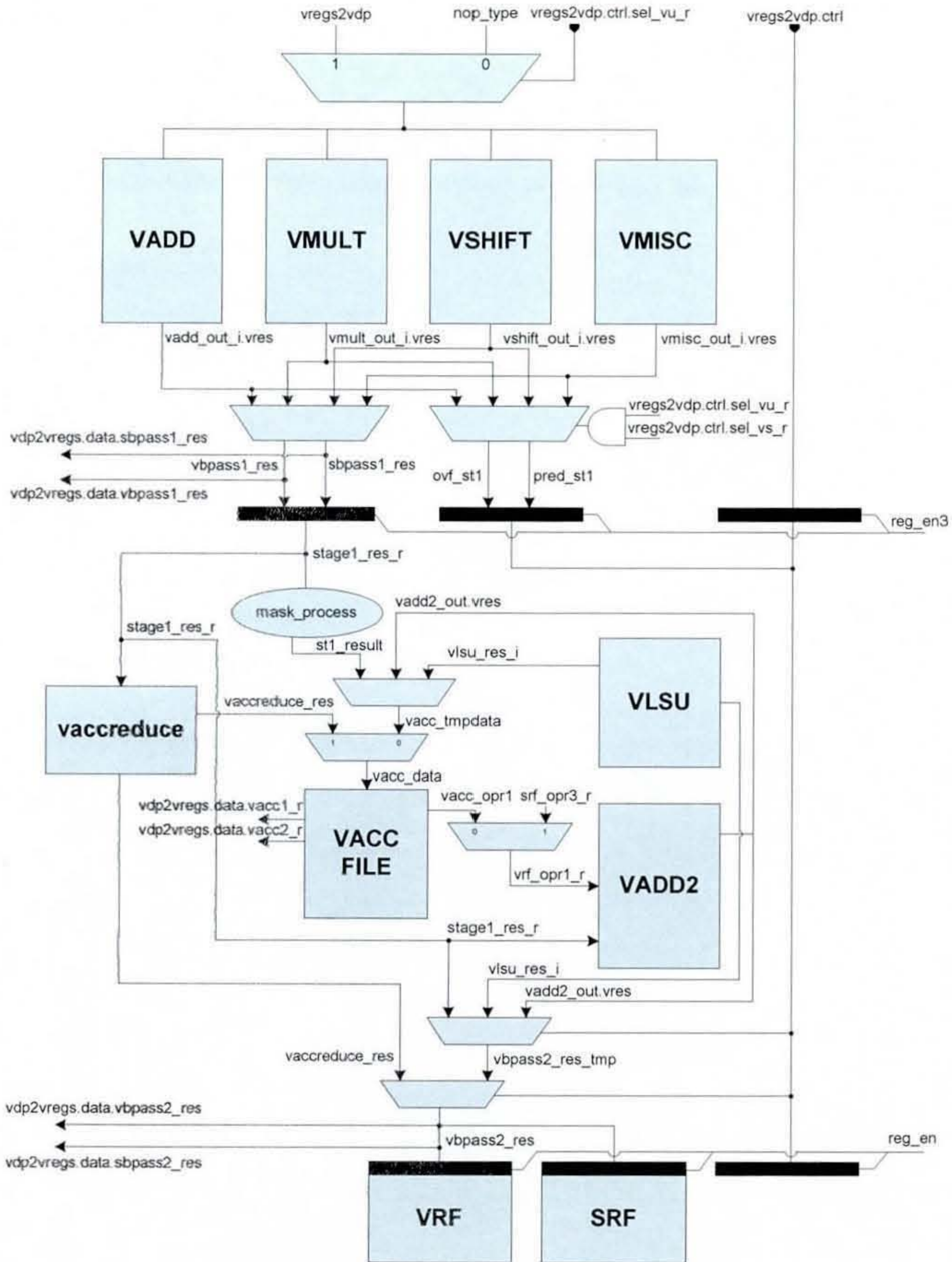


Figure 6-19: Microarchitecture of the VDP stage

The second stage accepts returning loads from the VLSU and performs the addition/subtraction part of the multiply-add/sub as well as the setting-up of the write data to the register files. At the end of each of the VDP stages and right before they are latched in to the corresponding output (VDP1 stage) or architectural (VDP2 stage) registers, the results are bypassed to the VREG stage in order to be available to dependent instructions

and avoid stalls due to Read-After-Write (RAW) dependences. The detailed schematic of the VDP stage is shown in Figure 6-19. Stage one consists of four vector datapath units: The vector adder (`vadd`), the vector multiplier (`vmult`), the vector shifter (`vshift`) and the vector miscellaneous (`vmisc`) unit. Each such vector unit consist of VLMAX/2 replications of their corresponding scalar unit that produces a 32-bit result. At the input of the vector units there are multiplexers that select which vector unit will accept the input operands and the control signals that are coming from the output registers of the VREG stage. The vector units not participating in the current computation cycle execute a `nop` instruction. This input operand gating is applied to eliminate redundant switching activity in the multiple functional units of the vector datapath. This ensures that unused functional units are kept in a quiescent state by maintaining constant inputs. This minimizes switching activity and as a result, dynamic power consumption. In the case that the coprocessor is performing a scalar instruction, the scalar operands along with the signal (`vregs2vdp.ctrl.sel_vs_r ='0'`) that indicates that it is a scalar operation are the inputs to the vector units. Special logic activates scalar lane (lower 32-bits) of the particular unit that comprises the selected vector path instead of implementing a dedicated scalar datapath. This results in reduced silicon area and control logic overheads and also to less verification effort [6]. At the output of the vector units there is another multiplexer that selects the vector result (`vbpass1_res`) or the scalar result (`sbpass1_res`) to be passed on to VDP2 stage, depending on the operation. At the same time, the result is bypassed to the VREG stage as an intermediate result and also it is written to the output registers of the first stage (`reg_st1`). At the second VDP stage the latched result from stage one (`stage1_res_r`) or the load data (`vlsu_res`) returned from the VLSU, are sent directly for writing back at the end of the cycle. In the case of the multiply-add/sub instruction the latched result from stage one (`stage1_res_r`) is used as the second input operand to the vector adder unit (`vadd_snd_stage`) of the second stage for the addition/subtraction part of the operation. When other instructions that employ accumulators occur, the registered result (`stage1_res_r`) is driven as the input data to the accumulator file with the exception of the `vaccreduce` instruction where the latched result of stage one is sent as an input to the adder tree (reduction unit). At the end of the second VDP stage, there is a multiplexer that selects the final result from the vector adder, the adder tree, the vector accumulator file, the load data from the VLSU or the pipelined result from the previous stage to commit to the vector/scalar

register files. The final result is bypassed as discussed previously to the VREG stage, in order that dependent instructions don't stall the vector pipeline.

6.5.1 Vector Adder Unit (gxx_vadd_dp)

The vector adder unit (`vadd`) is an array of VLMAX/2 identical units, where every such functional unit takes two 32-bit operands and produces a 32-bit result. The `vadd` unit can perform short (16-bit) or long (32-bit) addition, subtraction, comparison or 32-bit to 16-bit round operation. The electrical interface of the vector adder unit is depicted in Figure 6-20. As shown, every such functional unit takes as operands the correspondent elements of the input vector and produces a 32-bit vector result along with the overflow and predicate bits.

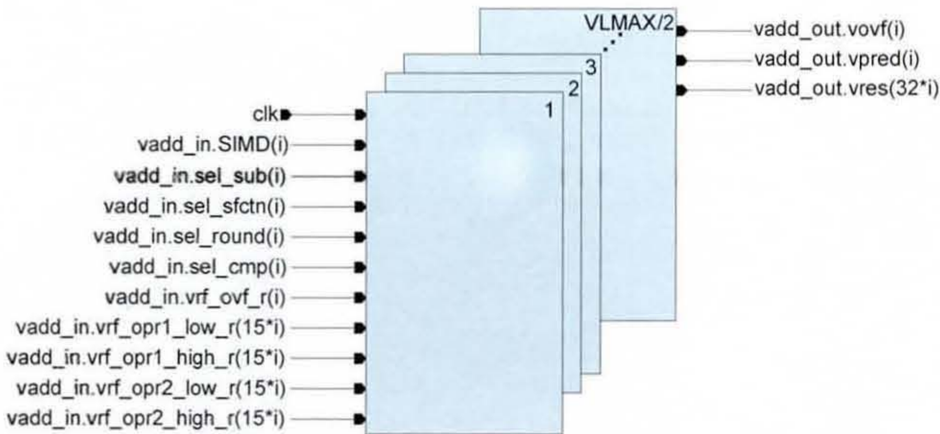


Figure 6-20: Electrical interface of the vector adder unit

The `vadd` unit comprises two mirrored combinational logic blocks that are called the "low" and "high" part of the unit. The low part calculates the least-significant 16-bits of the 32-bit result and the high part calculates the most-significant 16-bits. Additional logic exists between the low and high part that combines them in order to perform a long (32-bit) instruction. When a short operation is performed (`vadd_in.SIMD(i)='0'`) the two blocks work in parallel and produce two 16-bit results along with separate overflow and predicate bits. When a long operation takes place then the two blocks are linked together e.g. the carry out of the low part is driven to the carry in of the high part of the functional unit. A detailed schematic of the `vadd` functional unit's microarchitecture is illustrated in

Figure 6-21. The remaining control signals that define which operation the vector adder unit will execute are described in more detail in Appendix B.

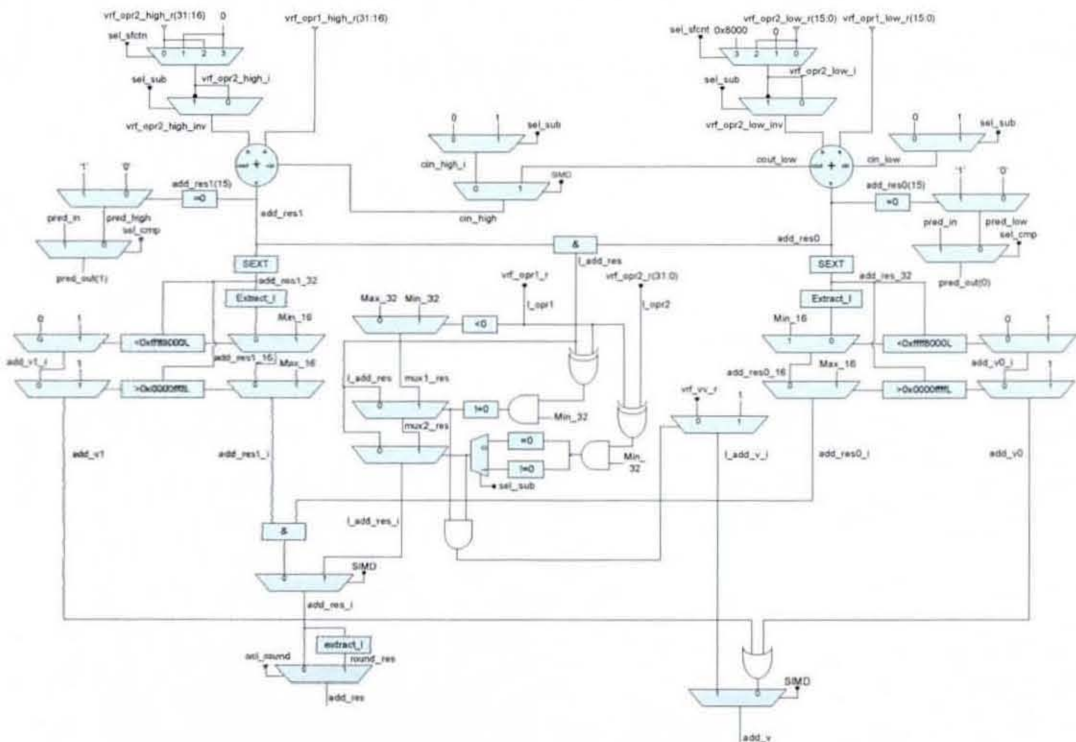


Figure 6-21: Microarchitecture of a functional unit of the vector adder

6.5.2 Vector Multiplier Unit (gxx_vmult_dp)

The vector multiplier unit (`vmult`) is an array of VLMAX/2 identical datapath units, where each such datapath takes two 16-bit operands and produces a 32-bit result. The `vmult` can execute all kinds of multiplications that the target speech coding workloads require. The electrical interface of the `vmult` is illustrated in Figure 6-22.

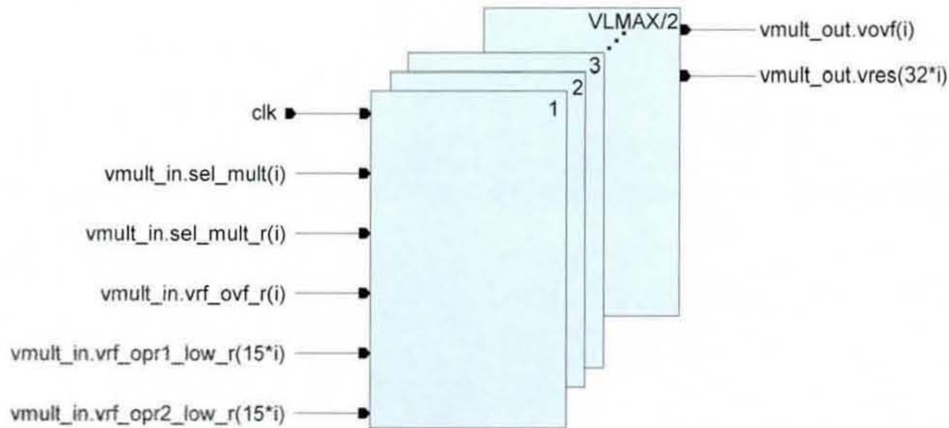


Figure 6-22: Electrical interface of the vector multiplier unit

Every functional unit takes as inputs the corresponding 16-bit elements, even or odd, of the full input vector operands and produces a 32-bit result for long multiplication or a zero extended 16-bit result for result consistency with the other types of multiplication along with an overflow bit. This is because every such unit comprises a 16x16 signed multiplier and every multiplication operation executes in instruction pairs for the even and odd elements of the input vector operands. The reason behind this choice comes from previous simulation studies which showed an improvement in the dynamic instruction count metric of the order of 2 - 4% when multiplying the even and odd elements of the input operands in parallel. Therefore by utilising a single multiplier per 32-bit scalar datapath, the number of multipliers is halved and at the same time the performance penalty is very little. The complete schematic of the microarchitecture of a functional unit of the vector multiplier is shown in Figure 6-22.

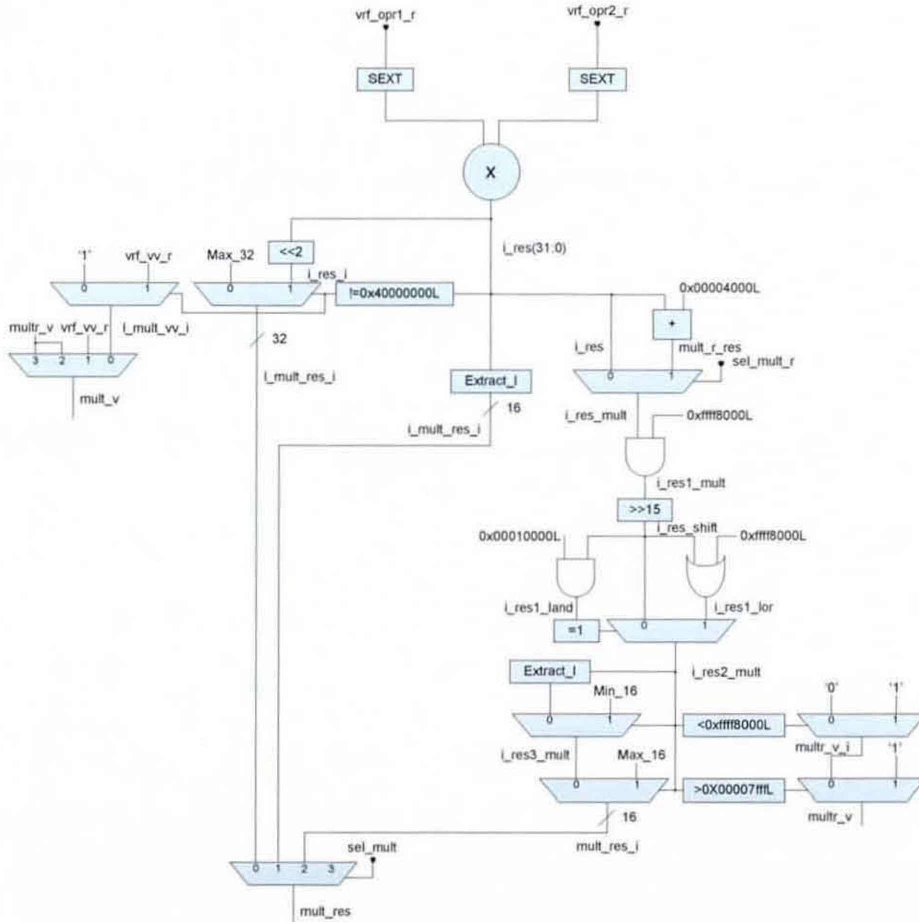


Figure 6-23: Microarchitecture of a functional unit of the vector multiplier

6.5.3 Vector Shifter Unit (gxx_vshift_dp)

The vector shifter unit (`vshift`) is an array of VLMAX/2 identical units, where each such functional unit takes as inputs two 32-bit operands and produces a 32-bit result and overflow bit. The `vshift` can perform short and long shift left and shift right operations to all or the even/odd elements of the input vector operands. The electrical interface of the vector shifter unit is depicted in Figure 6-24.

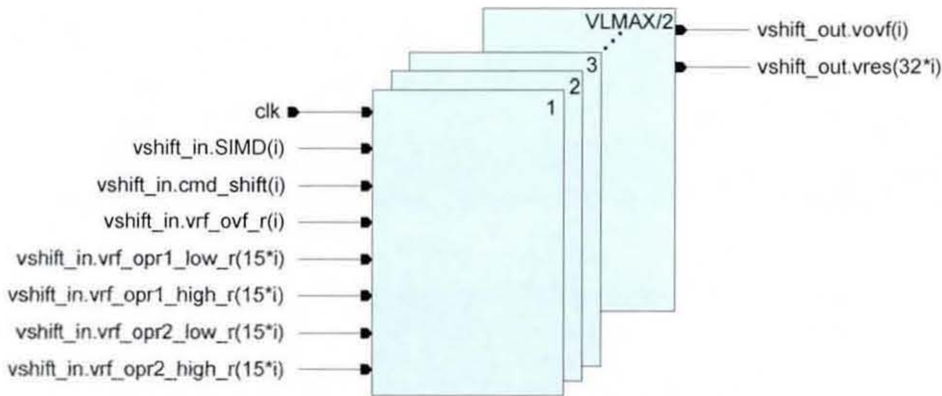


Figure 6-24: Electrical interface of the vector shifter unit

Every functional unit comprises two mirrored combinational logic blocks for the "low" and "high" part for the input vectors. The corresponded 16-bit elements of the input vector operands drive each of them respectively. Additional logic links the two parts of the unit in order to execute the long shift (`SIMD_i='1'`). Each of the logic blocks contains a specialised barrel shifter that implements the core functionality of the ITU-T shift operations. A barrel shifter is a common digital circuit that can shift or even rotate a data word by any number of bits in a single cycle [7]. For this particular design, a 16-bit bi-directional barrel shifter is implemented that comprises a network of multiplexers and can shift up to 15 positions on either direction. Each functional unit contains two such 16-bit barrel shifters that are connected in series in order to execute two short shifts or one long shift. Figure 6-25 shows the connections of the two barrel shifters ports.

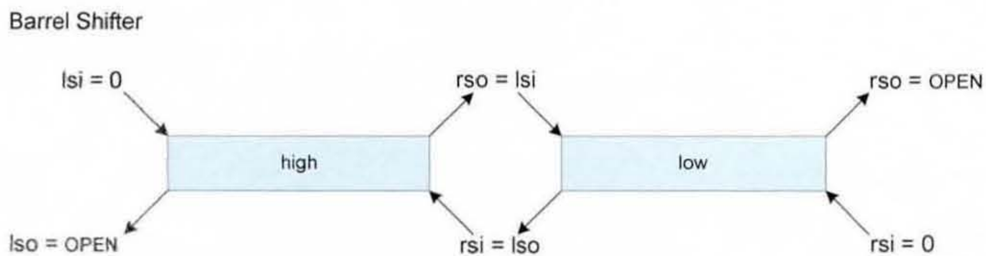


Figure 6-25: Two Barrel Shifters connected in series for short or long shift operations

The right shift output (`rso`) for the "low" barrel shifter and the left shift output (`lso`) for the "high" barrel shifter are left open as no rotation is specified in the coprocessor ISA. Due to this reason the right shift input (`rsi`) of the "low" shifter and the left shift

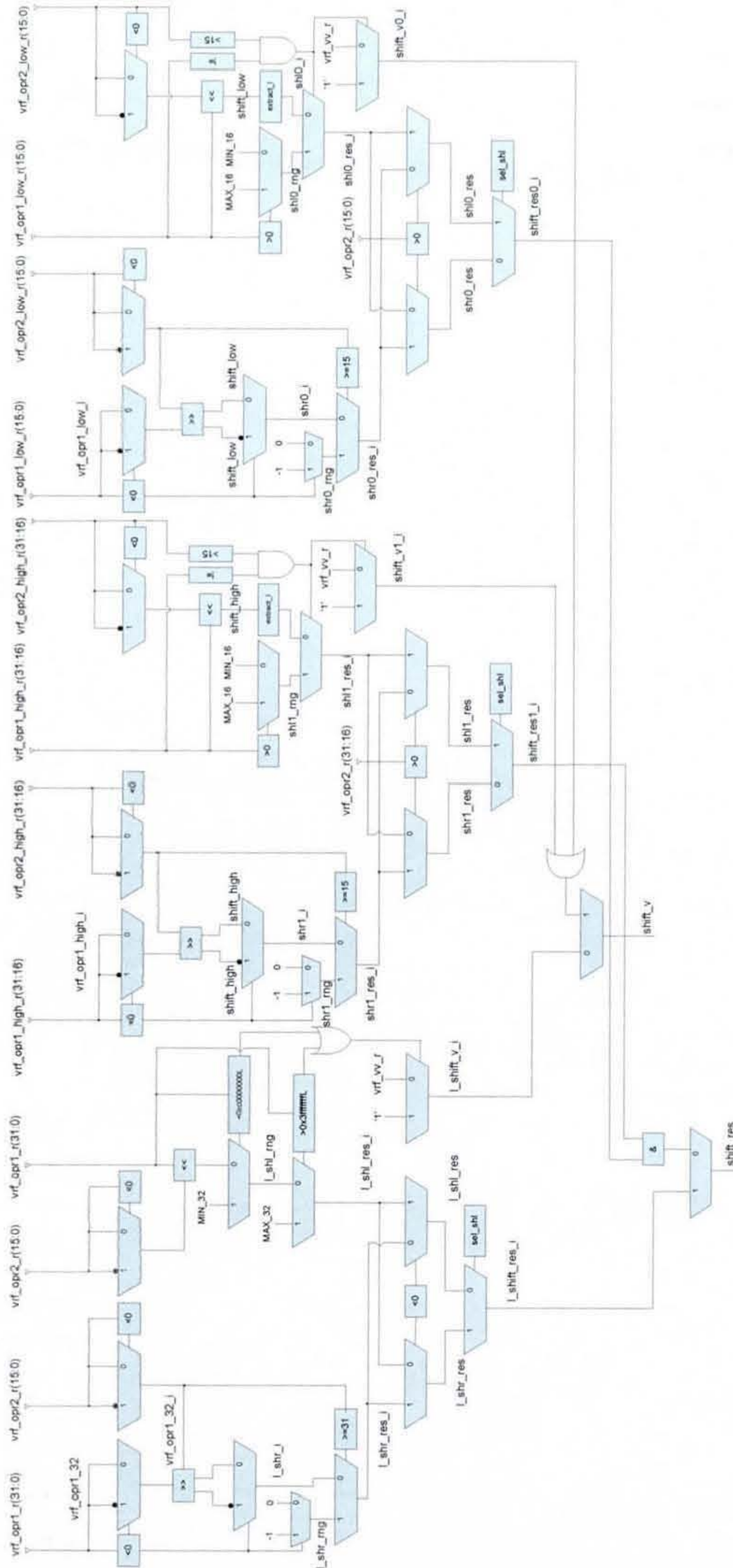


Figure 6-26: Microarchitecture of a functional unit of the vector shifter

input (`lsi`) of the "high" shifter are permanently tied to value zero. The remaining ports are linked together in order to execute the long shifts. The additional combinational logic around the barrel shifter saturates the shift result in case of underflows or overflows and checks if the shift amount is negative in order to perform the opposite-direction shift. A detailed schematic of the microarchitecture of a functional unit of the vector shifter is shown in Figure 6-26.

6.5.4 Vector Miscellaneous Unit (gxx_vmisc_dp)

The vector miscellaneous unit (`vmisc`) contains the logic that implements the miscellaneous vector operations of the coprocessor ISA. Every functional unit of the `vmisc` accepts two 32-bit input vector operands in case of vector operations or one 32-bit scalar operand for scalar operations and produces a 32-bit result (or 16-bit result zero-extended to 32-bits). The electrical interface of the `vmisc` is depicted in Figure 6-27.

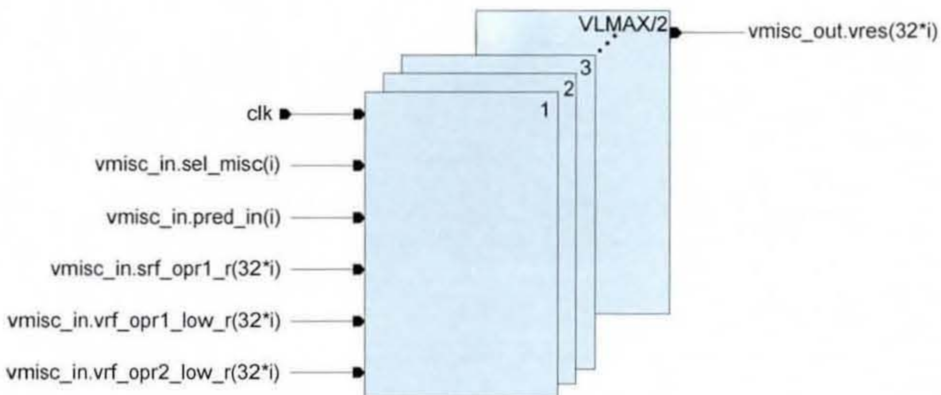


Figure 6-27: Electrical interface of the vector miscellaneous unit

6.5.5 Reverse Data Logic

As previously mentioned for the case of store operations with a negative stride, the store data are reversed in the VREG stage prior to sending them to the VLSU. The same operation is performed for data that return from the VLSU (`vlsu2vdp.data`) in the case of a load instruction with negative stride (`vldwn`). This is achieved with the reverse data logic (function). The function (`reverse_data`) output drives a multiplexer which selects

amongst the reversed (load with negative stride) and no reversed (standard load) for the returning result at the end of the second VDP stage.

6.5.6 Masking Process Logic

The masking process logic that is implemented in the second VDP stage selects the appropriate elements of a vector input and places them in the target vector accumulator. The input control signal (`reg_st1.ctrl.sel_evod_r`) defines which elements, even or odd, should be extracted from the vector input and be placed at the corresponding even or odd elements of the vector accumulator. The second input operand (`reg_st1.ctrl.vrf_opr2_r`) determines whether the sixteen bits of the even or odd elements of the vector input should be placed at the MSB (deposit high operation) or LSB (deposit low operation) of the 32-bits elements of the vector accumulator.

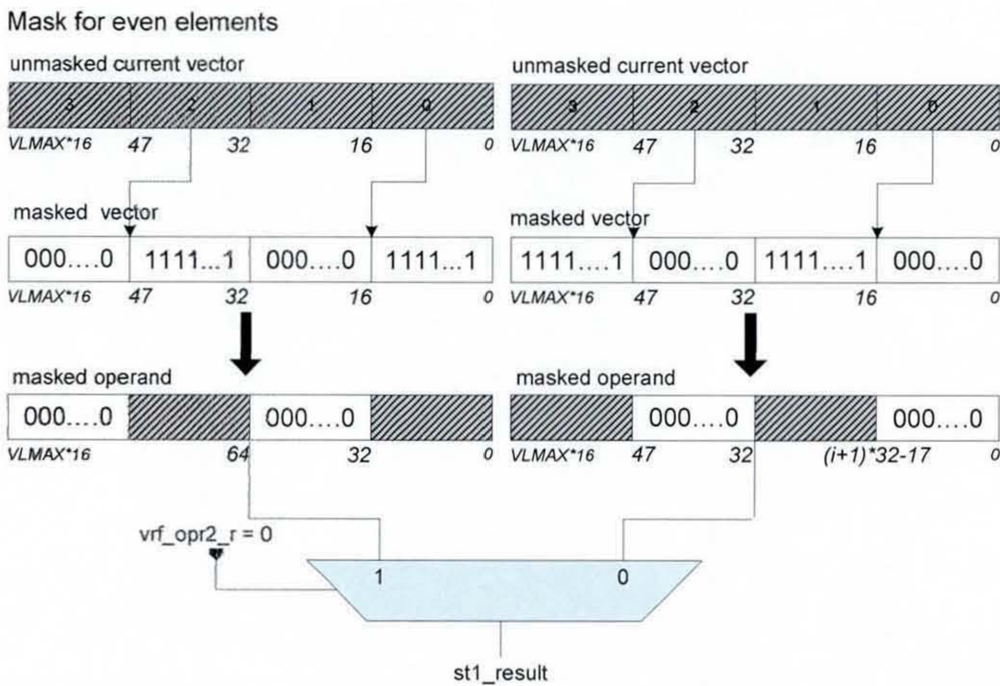


Figure 6-28: Masking process logic for low (vrf_opr2_r='1') or high (vrf_opr2_r='0') deposit for the even elements of the input vectors to the accumulator

This is implemented by shifting left by the amount specified in the second operand of the 16-bit scalar elements (within the vector input) that will be placed inside the corresponding 32-bit elements of the vector accumulator; the amount can take only the

values of zero or sixteen. The remaining bits of each element of the accumulator are filled with zeros. Figure 6-28 depicts the masking process for the even elements of a vector value with the second operand being zero and sixteen respectively.

6.5.7 Bypassing network of the first VDP stage

At the end of the first stage and prior to clocking the results into the VDP2 input registers, the intermediate vector and scalar results from the first execution stage (`vdp2vregs.data.vbpass1_res` and `vdp2vregs.data.sbpass1_res`) are forwarded to the VREG stage as inputs to the bypass logic process for the source vector and scalar operands selection. In addition, the destination write-addresses for the vector (`vdp2vregs.ctrl.vbpass1_vwr_addr_r`) and the scalar (`vdp2vregs.ctrl.sbpass1_swr_addr_r`) register files are sent along with the valid signals. The bypass-valid vector and scalar signals (`vdp2vregs.ctrl.vbpass1_valid`, `vdp2vregs.ctrl.sbpass1_valid`) are asserted in the same way as the register enable signals. For the vector bypass result, the current vector length (`vdp2vregs.data.vbpass1_vlen_r`) is also sent to the bypass logic to determine the extend that the intermediate result will comprise the source operand. The bypass logic for the second VDP stage is described in section 6.4.15.

6.5.8 Register Enable for the input VDP2 registers

The register enable (`reg_en3`) for the registers of the first VDP stage is asserted when both hold signals that come from the Leon3 (`hold`) and the VLSU unit (`vlsu2vdp.hold`) are set to zero. In addition, the latched register enable of the previous stage (`vregs2vdp.ctrl.reg_en3_r`) should be asserted. The registers at the end of the first VDP stage is of `reg_st1` type.

6.5.9 Second stage adder

When a multiply-add or a multiply-sub is executed, the multiplication is performed at the first VDP stage while the addition or subtraction part of the instruction is performed in the vector adder in the second VDP stage.

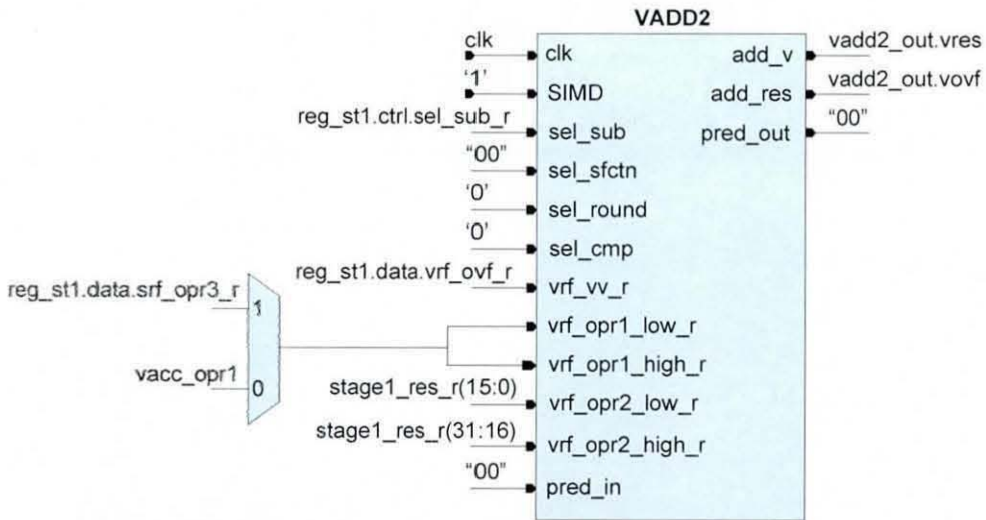


Figure 6-29: Electrical interface of the second VDP stage vector adder

This vector adder is identical to the vector adder unit of the previous stage apart from the fact that the control signals are pre-set to perform long addition or subtraction. This pipeline scheme was chosen to allow single cycle operations in VDP1 stage and at the same time compound (pipelined) operations such as multiply-add and multiply-sub to be fully pipelined by using the multiplier in the first stage and the second instance of the vector adder in the second VDP stage since the vector adder unit is reasonably cheap. The first input operand comes from a vector accumulator (`vacc_opr1`) or a scalar register (`reg_st1.data.srf_opr3_r`), depending if the instruction is a vector or scalar one. The second input operand is the registered multiplication result from the previous stage (`stage1_res_r`). The result from the vector adder (`vadd2_out.vres`) is written in the target vector accumulator and is also bypassed to the end of the second stage for the dependent instructions.

6.5.10 Vector Accumulator File (gxx_vaccs)

When the coprocessor is executing long operations (32-bits elements) or instructions that access the accumulator, one or two vector operands are read from the vector accumulator file. The vector accumulator file is a two-dimensional storage array parameterised as to the number of accumulators and their width. The number of the elements per accumulator is always equal to half the maximum vector length (VLMAX), 32-bit elements and there are ACC_NUMBER vector accumulators. In this particular instance of the architecture

the number of vector accumulators is set to 2 but can increase till 32, as the available opcode bits allow, for the multiply-add and multiply-sub operations.

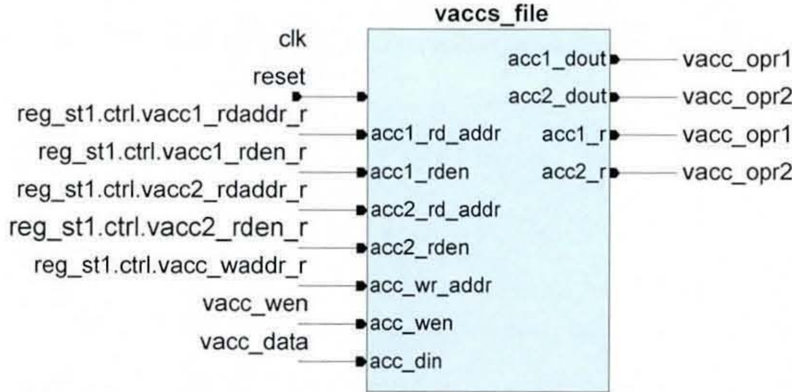


Figure 6-30: Electrical Interface of Vector Accumulator File

The only restriction is that the remaining long operations can use for source operands only the accumulator zero and accumulator one as these are hardwired to VREG stage for the source operands selection. The vector accumulator file implementation is flip flops-based and has two asynchronous read ports and one synchronous write port. In addition, there are an extra two hardwired read ports with the accumulators that are used in the VREG stage to retrieve the source operands when an accumulator source is specified. The accumulator file is located physically in the second stage of the VDP and its electrical interface is depicted in Figure 6-30.

6.5.10.1 Parameterisation

The vector accumulator file is a fully-configurable design. The number of accumulators (ACC_NUMBER) is within the range of 2 to 32, with a default setting of 2. The accumulator width (ACC_WIDTH) is always equal to half the maximum vector length (VLMAX), 32-bit elements. The compile-time parameters with their default values that specify the structure of the vector accumulator are specified in gxx_config.vhd and are listed to the Table 6-4.

Table 6-4: Compile-time vector accumulator file parameters for its architectural and microarchitectural state that are contained in gxx_config.vhd file

Parameter	Default
VLMAX	2, 4, 8, 16, 32, 64, 128
ACC_NUMBER	2
ACC_WIDTH	(VLMAX/2)*32

6.5.10.2 The vector accumulator implementation

In the vector accumulator file both read addresses (`reg_st1.ctrl.vacc1_rdaddr_r`, `reg_st1.ctrl.vacc2_rdaddr_r`) and read-enable strobes (`reg_st1.ctrl.vacc1_rden_r`, `reg_st1.ctrl.vacc2_rden_r`) are coming pipelined from the vector decode stage as well as the write address (`reg_st1.ctrl.vacc_waddr_r`). The write enable (`vacc_wen`) is set when the register enable (`reg_en`) of this stage is set in order to implement the synchronous write when the result is ready at the end of the second VDP stage. The accumulator write data (`vacc_data`) are coming from the second vector add unit when multiply-add/sub operation is performed or from the VLSU unit in the case of load or from the adder tree.

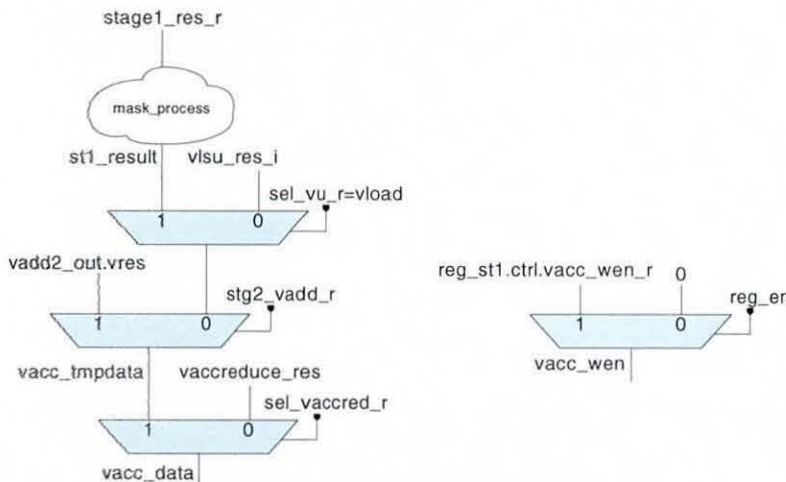


Figure 6-31: Write data and write-enable selection logic for the vector accumulator file

Otherwise, the write data originate from the masked vector output (`st1_result`) of the VDP1. The masked value is formulated with the use of a masking logic to implement the pair of instructions, even and odd, for deposit high (amount 16) and deposit low (amount 0) operations and it is described in more detail in section 6.5.6. In the case of other long

instructions the vector output is unchanged. Figure 6-31 illustrates the selection logic for the write data and write-enable for the vector accumulator file.

6.5.11 Vector Adder Tree (gxx_adder_tree)

The adder tree is utilised in the `vaccreduce` instruction in which all the scalar elements in an accumulator are add-reduced to a final 32-bit result. The adder tree is a parameterised two-dimensional matrix of adders $\log_2$(VLMAX) rows deep and VLMAX/2 adders at the row zero that are decreased by half in every row. Figure 6-32 shows an adder tree configuration for vector length of 256-bit elements (VLMAX 16). At the beginning (row zero) there are four (VLMAX/4) adders that perform 32-bit additions. The 33-bit results are added in pairs from two adders that comprise the second row (row1). The two 34-bit results are added with each other to form the final 35-bit result whose least-significant 32-bits are passed to the output (`adder_tree_out`) of the adder tree.

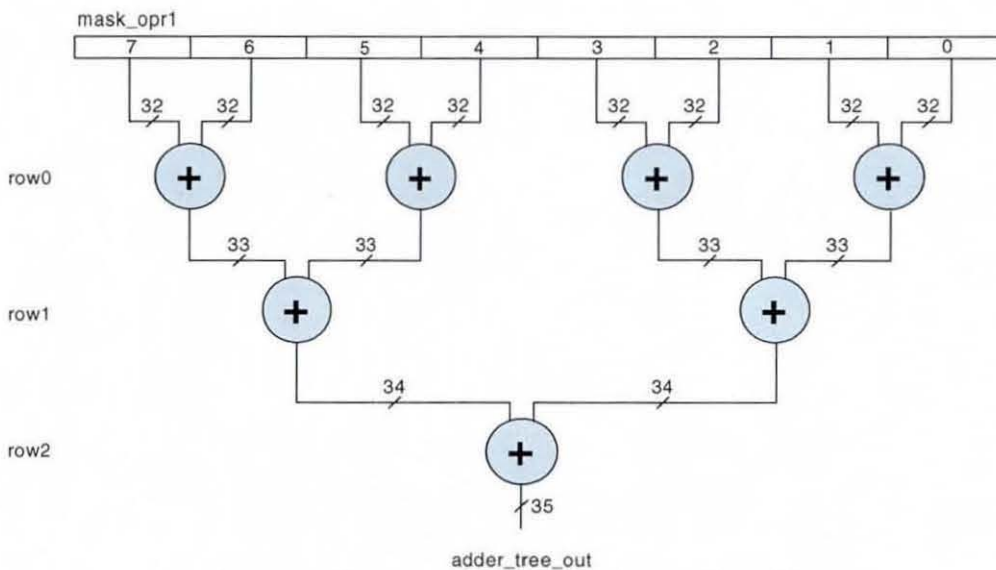


Figure 6-32: Adder tree configuration for VLMAX 16

The input operand (`adder_tree_in.data.vrf_opr1_r`) is masked to the current vector length (`adder_tree_in.data.vlen_cvalue_r`) in order only the necessary vector elements to be processed as the remaining vector elements till VLMAX are set to zero. This process is performed for reducing power consumption as the non used flops are not switching.

6.5.12 VLSU unit interface with VDP2

When a load instruction is performed (reg_st1.ctrl.sel_vu_r=vload) the requested data from the memory is returned, if valid (vlsu2vdp.data_valid='1'), by the VLSU in the second VDP stage as shown in Figure 6-14. As previously mentioned the VLSU has a cascade TAG/DATA configuration which translates to a minimum of 2-cycle load/use latency if no cache miss takes place. The returned data (vlsu_res) has vector length of VLMAX*16 bits for vector load or 32 bits zero extended to VLMAX*16 bits for the scalar load.

6.5.13 Overflow and Predicate Flags

At the end of the first VDP stage a multiplexer selects the result overflow (ovf_st1) from the vector unit that executed the coprocessor operation. In the case of a miscellaneous or shift operation, the overflow takes the pipelined value of the overflow register (vregs2vdp.data.vrf_ovf_r) as no new overflow value is produced by either operation. In the second VDP stage another multiplexer selects the overflow (ovf_st2) from the latched overflow of the previous stage (reg_st1.data.vrf_ovf_r) and the produced overflow (vadd2_out.vovf) of the second vector adder in the case of multiply-add/sub operation. The write enable signal (vdp2vregs.ctrl.ovf_wen) for the overflow flag/register is asserted when the instruction is valid and no exception is detected (reg_en='1') and it is pipelined along with the overflow value to the VREG stage to update the overflow flag/register. A predicate value is produced only in the first VDP stage from the vector adder unit (vadd) in the case of a comparison instruction. A multiplexer selects, at the end of the first stage, the predicate value (pred_st1) from the pipelined value of the predication register (vregs2vdp.data.pred_r) or the predicate result of the vadd. At the second stage the latched predicate value (reg_st1.data.pred_r) along with the write enable (vdp2vregs.ctrl.pred_wen) are sent to the VREG stage in order to update the predicate register. The write enable of the predicate register is controlled by the same conditions that apply to the overflow flag/register write enable signal.

6.5.14 Bypassing network of the second stage

Prior to writing back to the register files, the results (`vdp2vregs.data.vbpass2_res` and `vdp2vregs.data.sbpass2_res`) are forwarded again to the VREG stage as inputs to the bypass process for source operands selection. The valid signals (`vdp2vregs.ctrl.vbpass2_valid` and `vdp2vregs.ctrl.sbpass2_valid`) that are sent along with the bypassed results and the target register write addresses are asserted when the instruction is valid and no exception is detected from the previous stages (`reg_en='1'`). Again the current vector length (`vdp2vregs.data.vbpass2_vlen_r`) is sent to determine which part of the source operand will contain the forwarded vector result.

6.5.15 Write Back

This is the final stage prior to committing a result to the vector or scalar register files or the vector accumulators. This stage is actually incorporated in the end of the second VDP. It includes combinational logic that selects the results of the operations that took place in the second stage of the vector datapath along with the results from the previous stage. The result thus can be derived from the VLSU unit (load operation) or from the accumulator file (L_mac/L_msu operation) or the adder tree unit (vaccareduce operation) or the registered result of the first stage of the vector datapath. The pipelined addresses and write enables are sent to VREG stage to select the destination registers for the results.

6.6 Output Register Bunch

At the end of each stage of the vector coprocessor pipeline, there are the output registers which contain the control and data signals that enter into the following stage. The signals for all the pipeline stages are listed analytically in Appendix B.

6.7 Leon3

As discussed in the previous chapter, the vector coprocessor is tightly-coupled to the Leon3 32-bit CPU which was chosen as the basecase CPU. A number of modifications took place in the Leon3 pipeline in order to attach the vector coprocessor and its control

and data channels. These changes will be described in the order they appear in every stage of the pipeline of the Leon3 in the following paragraphs.

6.7.1 Decode Stage

In the Decode stage the latched instruction from the Fetch stage (`de_inst`) is decoded in parallel from both the CPU and an additional combinational logic that inspects if the current instruction is for the vector processor or not. As mentioned above the instruction opcode that is targeted for the vector processor is the one embedded in the lower 22 bits of the UNIMP instruction (Figure 6-3). The additional combinational logic checks the bits 31:30 and 24:21 of the latched instruction and if equal to zero an opcode valid signal (`v.a.opc_valid`) is asserted and the bits 21:0 are pipelined as the coprocessor opcode (`v.a.opc`). In the case of a move data instruction from Leon3 to coprocessor (`mvsr2gpr` or `mvvr2gpr`) a data enable signal (`v.a.vcop_data_en`) is also asserted. Additionally in this stage, the addresses of the source and the destination operands are extracted from the latched instruction in parallel with decoding. This allows the concurrent access of the register file in order to prepare the operands for the next stage. When the address of the first source operand is calculated additional logic checks from which field to extract it depending on whether it is a move instruction from the main CPU to coprocessor or not. Similar combinational logic selects the destination address and sets the write enable signals according to whether a move instruction from the coprocessor to the CPU has been decoded or not.

6.7.2 Register Access stage

In this stage the operands are read from the register file or from intermediate data bypass networks. When a coprocessor instruction is performed the selected default operation in Leon3 will be addition. This in combination with the zero operands passed to the next stage, will cause Leon3 to perform a NOP operation when the executed instruction is targeting the coprocessor. However, this is still a valid instruction packet and can be interrupted like any other Sparc V8 instruction. In the case of a move from the CPU to the coprocessor instruction the first source operand (`v.e.op1`) is pipelined as data input (`v.e.leon_data`) to the latter. The opcode (`v.e.opc`), the opcode valid (`v.e.opc_valid`) and the data enable (`v.e.vcop_data_en`) signals for the

coprocessor are pipelined to the next stage if there is no exception and the opcode valid of the previous stage (`r.a.opc_valid`) is asserted. In addition, at the exception_detect process, the VCOP logic was added to deactivate the `illegal_inst` signal when the Leon3 decoder detects the UNIMP format that is the case of a coprocessor instruction. This ensures that all UNIMP opcodes are "hijacked" and passed to the coprocessor for execution.

6.7.3 Execute Stage

In the Execute stage all the arithmetic, logical, shift and miscellaneous operations are performed along with the load/store address calculation. When a coprocessor instruction is executed the source operands are set to zero. Therefore, the Leon3 will perform an addition with zero operands and this will emulate a `nop` instruction. Similarly to the previous stage, the coprocessor signals (`v.m.opc`, `v.m.opc_valid`, `v.m.vcop_data_en`) are pipelined to the next stage in the case of no exception and the opcode valid of the previous stage (`r.e.opc_valid`) is asserted.

6.7.4 Memory Stage

In the Memory stage the data cache is accessed and the store operation is performed. It is this stage where the vector coprocessor is attached to the Leon3 pipeline in order to avoid the majority of the exceptions and interruptions of the Leon3 and to have enough time to transfer data to/from the main processor (write stage) if requested. Therefore, when a coprocessor instruction is performed and there is no exception and the opcode valid of the previous stage (`r.m.opc_valid`) is asserted the vector/scalar instruction (`iu2vcop_opc`) along with the valid signal (`iu2vcop_opc_valid`) is sent to the decode stage of the vector coprocessor. In addition, the other control signals (`v.x.opc_valid` and `v.x.vcop_data_en`) are pipelined to the next stage.

6.7.5 Exception Stage

In this stage, all the traps and interrupts are resolved and the data are aligned for data cache read. Even though the full functionality of Leon3 supports single issue, seven stage

pipeline, in this application the write back (7th stage) is not implemented and the outputs from the exception stage are going straight to the register file.

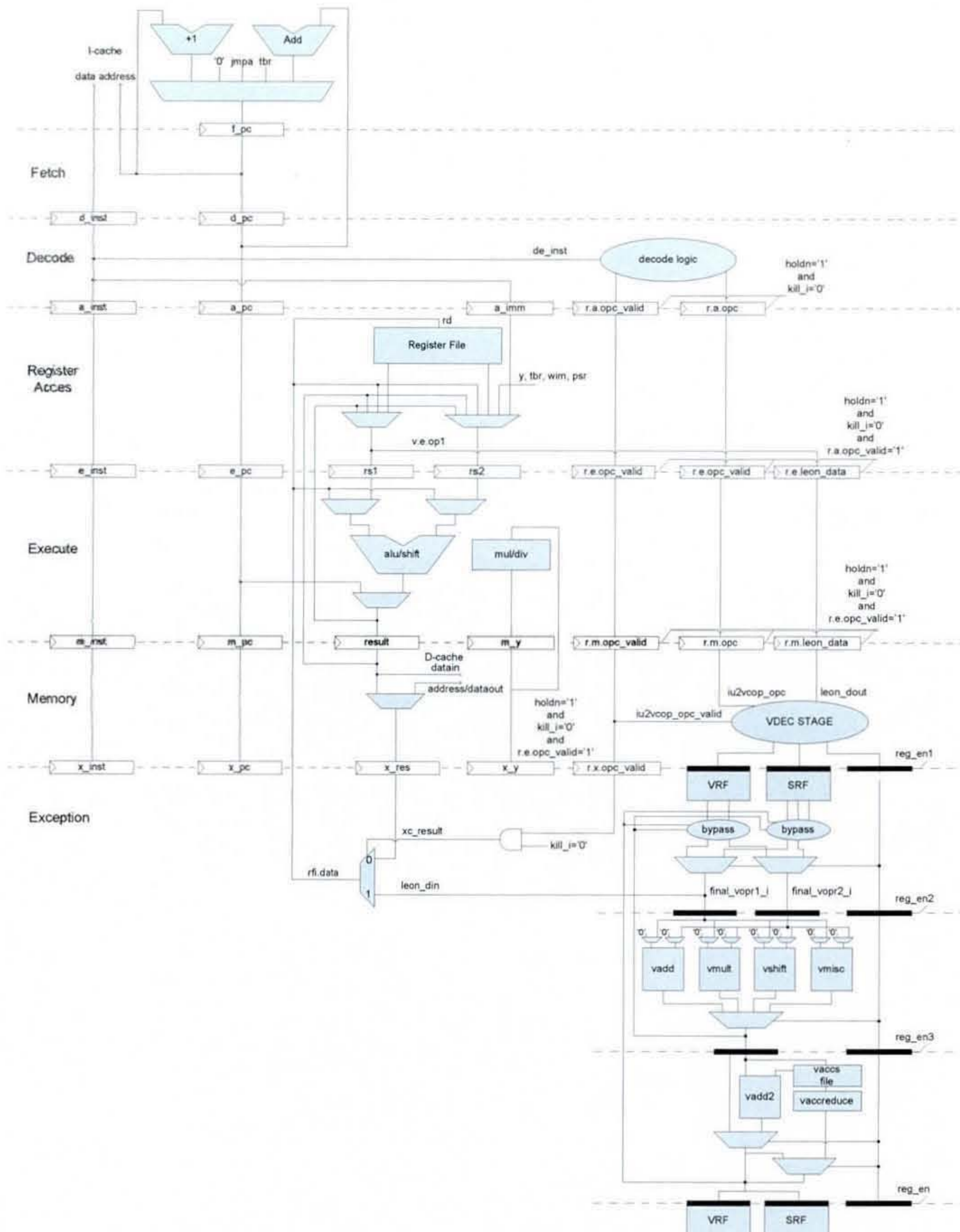


Figure 6-33: Leon3 integer unit and vector coprocessor datapath diagram

The write data, prior to commit to the register file (`rfi.wdata`), is the output of a final multiplexer which selects the result from the main CPU (`xc_result`) or the coprocessor data (`leon_din`) from the VREG stage in the case of a move instruction from the VCOP

to the Leon3. The latter is performed only if there is no exception and the opcode valid of the previous stage (`r.x.opc_valid`) is asserted. The detailed schematic of the Leon3 with the attached vector coprocessor is illustrated in Figure 6-33.

The vector processor was added in the proc3 hierarchy and connected with the interface of the integer unit. In this hierarchy, the Leon3 processor core with the integer unit and the complete cache sub-system with controllers and rams are contained. It also comprises the multiply and divide units hardware. Figure 6-34 depicts the proc3 hierarchy that includes the vector processor.

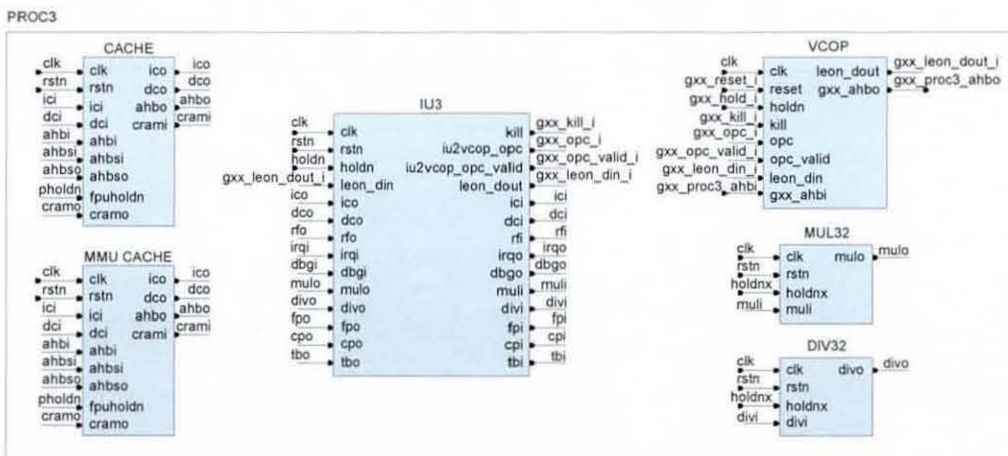


Figure 6-34: Leon3 processor core block diagram

6.8 Summary

In this chapter, the design and implementation of the vector datapath was described. The pipeline organization and its constituent components were presented along with a brief description of the VLSU. In addition, the modifications to the Leon3 pipeline to enable its tight-coupling to the vector processor were detailed.

6.9 References

[1] S. R. Parr, "High Performance Load/Store Unit for a highly configurable, embedded vector processor," in *Electronic and Electrical Engineering*: Loughborough, 2007.

[2] "The Sparc Architecture Manual Version 8 ", www.sparc.com.

[3] J. L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach," 3 ed: Morgan Kaufmann, 2003.

[4] S. Furber, "ARM: System-on-Chip Architecture," Second ed: Addison-Wesley, 2000, pp. 80-81.

[5] C. Kozyrakis, "Scalable Vector Media-processors for Embedded Systems," in *Computer Science* University of California: Berkeley, 2002.

[6] S. R. Parr, K. Koutsomyti, and V. A. Chouliaras, "A High Bandwidth Configurable Load/Store Unit for an Embedded VectorProcessor," in *Postgraduate Workshop on Embedded Systems* Birmingham, UK, 2006.

[7] P. A. Beerel, S. Kim, P.-C. Yeh, and K. Kim, "Statistically optimized asynchronous barrel shifters for variable length codecs," in *International symposium on Low power electronics and design*, San Diego, California, 1999, pp. 261 - 263.

CHAPTER 7
VECTOR PROCESSOR VLSI IMPLEMENTATION

7.1 Design Verification

The vector datapath was verified using test vectors that were produced by recording the inputs operands, the state of the global overflow flag and output results from each of the C macros that implement the basic operations. The recording process was performed by inserting pre-processor directives to every basic operation in their definition file as it is shown in the code snippet of Figure 7-1. The figure depicts the C macro of the basic operation L_mult and as it can be seen the pre-processor directive (#ifdef GEN_TVEC_L_MULT) uniquely identifies the name of the operation under test and selects the inputs and the outputs for recording which are then piped to a file. The whole process was controlled by a Perl script.

```
Word32 L_mult(Word16 var1,Word16 var2)
{
 Word32 L_var_out;
#ifdef GEN_TVEC_L_MULT
 int Overflow_in=Overflow;
#endif
 L_var_out = (Word32)var1 * (Word32)var2;
  if (L_var_out != (Word32)0x40000000L) {
    L_var_out *= 2L;
 }
  else {
   Overflow = 1;
   L_var_out = MAX_32;
 }
#ifdef GEN_TVEC_L_MULT
  fprintf (tv ,"%x,%x,%x,%x,%x\n",var1 , var2 , Overflow_in , L_var_out , Overflow);
#endif
  return(L_var_out);
}
```

Figure 7-1: Example of recording the inputs and the outputs of the L_mult operation C macro

The two scripts for producing test vectors for the speech coding algorithms run the workloads by using the architecture-level simulator for all the ITU-supplied bitstreams. The produced test vectors are subsequently applied to the vector datapath via a FLI-based testbench. The testbench is a self-contained VHDL model in a testing system and is designed to perform an automatic sequence of operations to validate the functionality of a design-under-test. The latter is instantiated and driven with a long sequence of the test vectors, created during the normal execution of the ITU-T workloads. These vectors are imported into the testbench and read by a Foreign Language Interface (FLI)-based stimulus process. The FLI provides a way for software components written in a high-level language, such as C, to interact with components written in VHDL or Verilog. In this particular case, the FLI allows for the C code, which reads in the test vectors from the stimulus file, to be used within the VHDL simulation environment and for each variable in the test vector to drive the correct signals of the VHDL testbench. The designs were simulated and verified with Mentor Graphic's ModelSim [2]. This software package allows for the event driven simulation of a VHDL or Verilog design and performs direct comparison between the outputs from the design-under-test and the expected (golden) results, stored also in the stimulus file. From the comparison an error report is produced that is used to validate the functionality of the design-under-test.

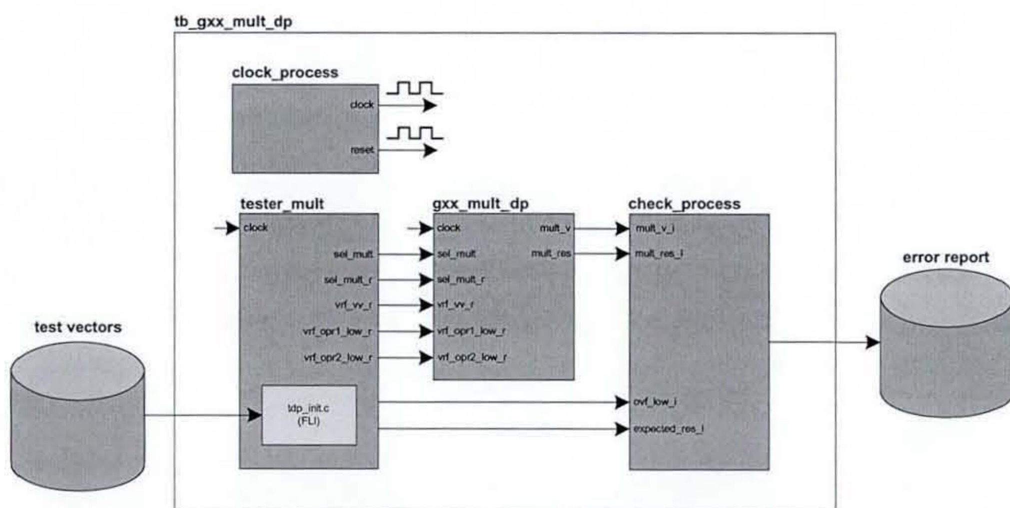


Figure 7-2: Test bench for the vector mult unit of the vector datapath

The functionality is based on the specifications imposed on the design and can be confirmed by producing the expected results. The vector datapath testbench consists of four testbenches, one for each vector unit. Each such testbench was designed for the

particular datapath blocks and their functionality was validated on a per-workload basis. Figure 7-2 shows the configuration of the testbench for the vmult unit. It comprises the clock process, the stimulus process (`tester_mult`) that reads the test vector from the stimulus file via the FLI (`tdp_init.c`), the `vmult` unit (design-under-test: `gxx_mult_dp`) and the check process that performs the comparison of the outputs of the `vmult` unit and the expected golden results from the FLI. The testbenches for the other vector units of the datapath have similar configurations. This kind of verification is called block-level verification. After the block-level verification, system level verification was performed. Figure 7-3 depicts the configuration of the testbench for the overall design of the vector coprocessor.

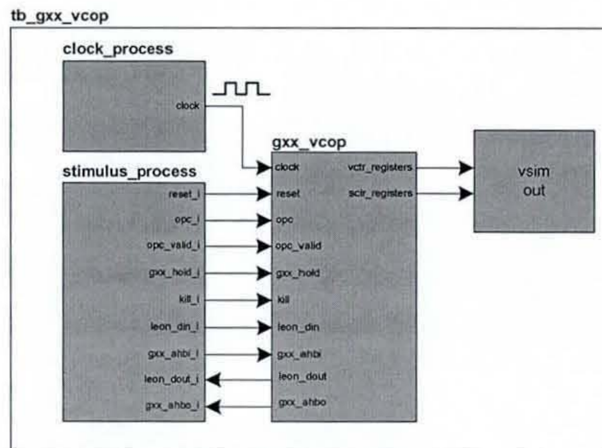


Figure 7-3: Vector coprocessor testbench configuration

The full coprocessor testbench consists of the clock process that generates the periodic clock signal, a hardwired stimulus process and the VHDL simulator output. The hardwired stimulus process drives the inputs of the vector coprocessor interface and the produced response is observed on the VHDL simulation environment.

7.2 Synthesis and Place & Route Design Flow

The design flow of the vector coprocessor is completed via a fully-automated synthesis/place-and-route campaign. This process is driven by using a grand (master) script whose pseudocode is depicted in Figure 7-4. This script runs Design Compiler for logical synthesis (statement S9), Cadence SoC Encounter for place and route (S10) and again DC (S11) for statistical power analysis. These are performed for different vector

lengths (VLMAX) and different periods in order to have a complete view of the vector coprocessor.

```
        Main driver script
        {
S3      for each VLMAX
          {
S5         for each period
             {
               Change period;
S8             Modify processor configuration;
S9             DC run1: Logical Synthesis;
S10            Encounter run: Place and Route;
S11            DC run2: Power Analysis;
             }
          }
         } end;
```

Figure 7-4: Script in a pseudocode for the design flow of the vector coprocessor

These steps of the master script and the produced results are described in more detail in the following sections.

7.2.1 Design Compiler Stage (Logical Synthesis)

After the design verification the next step is the synthesis phase. Synthesis is the automatic transformation of a Register Transfer Level (RTL) design description to a gate level netlist implementation. The synthesis process takes as inputs the RTL HDL description, timing constraints and attributes for the design and a technology library and produces a fully-mapped gate level netlist. Synthesis is an iterative process that starts by defining the constraints for each RTL block of the design and optimising the gate-level netlist for area, timing and power [1]. The synthesis tool that was used for the coprocessor design is the industry standard Synopsys Design Compiler (DC) [3]. The target standard-cell technology chosen for the design was Taiwan Semiconductor Manufacturing Company's (TSMC) 0.13μm standard-cell library (1Poly, 8 Copper) [4]. Using this technology, each design was synthesised varying both VLMAX and target clock frequency (period). The design constraints that contain the timing and the area information are defined in the design compiler's TCL (Tool Control Language) driven script. This script is used to guide the synthesis and optimization process of the design with the ultimate aim of meeting the user-specified constraints. The output of the DC run

are design timing constrains in Synopsys design constraints (*.sdc) format in addition to the new netlist representing the mapped and optimised design.

7.2.2 SoC Encounter script Stage (Place and Route)

After Logical Synthesis with Design Compiler, it is the turn of the Place-and-Route encounter script to run. This script drives the place and route process which produces the necessary files for statistical power analysis. The script starts by running Cadence First Encounter (FE) [5] in batch mode and by reading in the physical view of the RAMs and the library along with their timing view. The optimised verilog netlist (*.v) from the previous stage and the Synopsys Design Constraints (*.sdc) file that specifies the timing constraints are then imported. The place and route tool performs floor planning, power grid specification (power/ground ring and stripes), placement of RAM macros and standard cells, and clock tree synthesis. These are followed by global and detail routing (multithreaded mode), extraction of RC data and post clock tree synthesis timing optimization to fix the setup time. This is achieved by the tool inserting to the setup-violating paths buffers or inverters and doing gate resizing (including flip-flops) and instance cloning. After that step, filler cells (dummy cells) are added to fill the area between the placed and routed standard cells and connect their VDD and VSS rails to the power ring. When the final layout is ready it needs to be checked against the verilog netlist (Layout vs. Schematic LVS). In addition, Design Rule Check (DRC) takes place that checks the enforcement of the technology library design rules in the final layout. The outputs from this stage include area, and maximum frequency reports, of the design along with path delays, timing constraint values, interconnect delays in standard delay format (*.sdf) file, standard parasitic extraction format (*.spef) file and a new gate-level netlist representing the very final placed-and-routed design. These outputs are then read back into DC for the final stage of statistical power analysis.

7.2.3 Statistical Power Analysis Stage (Design Compiler)

Power analysis in statistical mode is run immediately after the end of place and route. The post placed-and-routed verilog netlist is loaded along with the timing constraints (.*sdc) and the standard parasitic extraction format (*.spef). When the statistical power analysis is performed several files are created which include average power dissipation, area,

worst IR drop etc. The final such results for the vector datapath, coprocessor and the overall system are presented in the following sections.

7.3 Implementation Campaign for Vector Datapath

For the vector datapath three different metrics were obtained, namely power, area and maximum operating frequency f_{max}. Figure 7-5 depicts the statistical power consumption observed for varying VLMAX and clock period of the vector datapath design. Here each requested period is plotted against its corresponding power. Observing this set of results it is obvious that the general shape of the graph is of a similar nature. It can be seen that as VLMAX increases the amount of power increases proportionally. This is due to the fact that the higher the VLMAX the higher becomes the physical number of gates placed on the silicon. This rise in the number of the gates inevitably leads to an increase in the power consumption. In addition, this plot reveals how the consumed power has a direct relationship with the speed that the design can operate. As the design is pushed into operating at higher frequencies the power dissipated at these frequencies increases as well. This is an expected result as the design's clock frequency affects the number of switching gates thus leading to a rise in dynamic power.

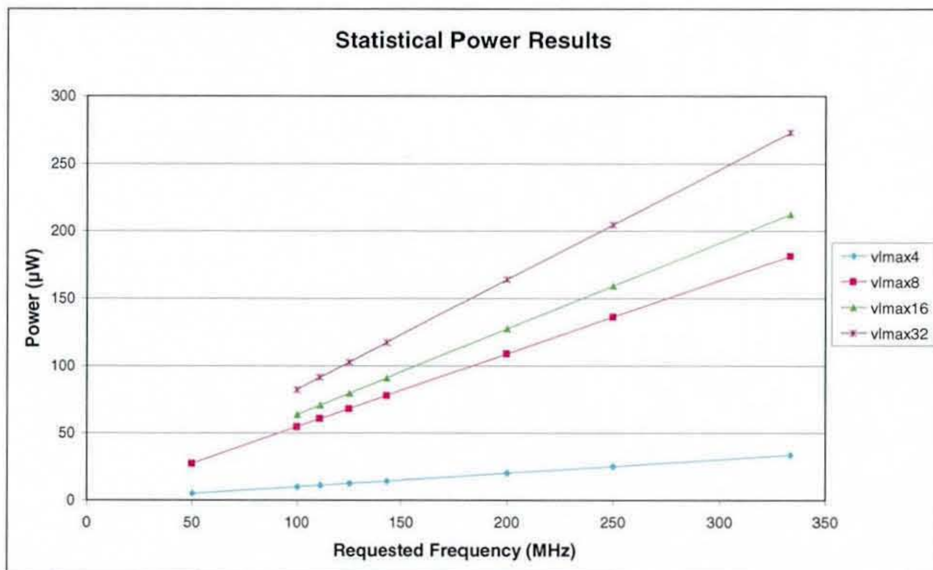


Figure 7-5: Statistical power results of vector datapath for different vector lengths

Another interesting observation is the significant difference in power consumption that observed for all VLMAX and maximum frequencies. At a vector length of 4 and a

frequency range of 100 to 333MHz the vector datapath power consumption ranges from 9.94 to 82.17mW whereas for vector length 32 at the same frequency range the power consumption ranges between 33.69 to 272.96mW. This can be seen as a fairly constant three fold increase in power consumption. In addition to studying the power consumption of each design methodology the physical area of each design was recorded. Figure 7-6 shows how this area changes for different values of VLMAX and at different frequencies (periods) for the vector datapath design.

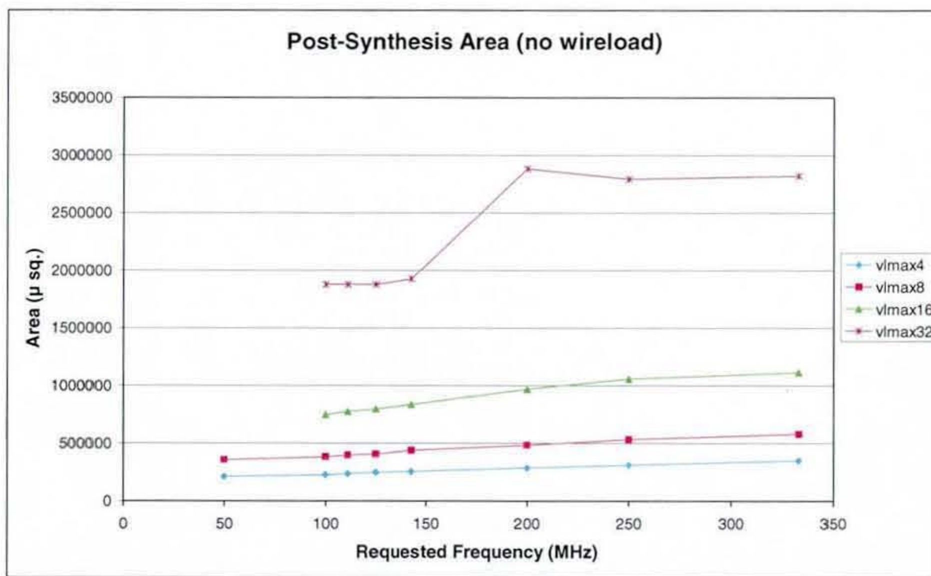


Figure 7-6: Statistical area results of vector datapath for different vector lengths

As it can be observed from the graph the required area for a given VLMAX shows a marginal change for the studied frequency range. The vast majority of the silicon area within the chip is used by the logic gates that perform the functionality of the design. As the frequency requirement increases various synthesis optimization methods are automatically applied to allow for the design to operate at this higher frequency. These methods often lead to an increase in silicon area as they employ faster and larger buffers for timing optimization of the critical paths and consequently affect the whole system layout. All these methods for pushing the design to achieve ever increasing speeds have an adverse affect on both power and area. Another observation that can be made from the above graph is that the area of the device is directly related to the vector length (VLMAX). This is due to the effect of the vector length on the quantity of the design logic as each increase in VLMAX involves additional vector element instantiations. The increase in area

required for higher vector length completely overshadows the increase due to the operation at higher frequencies. This effect of the operating frequency on the area of a device is less apparent as the effect of the dramatic increase in logic required for each change in VLMAX. Due to these reasons the graph shows a near parallel set of lines for the vector datapath.

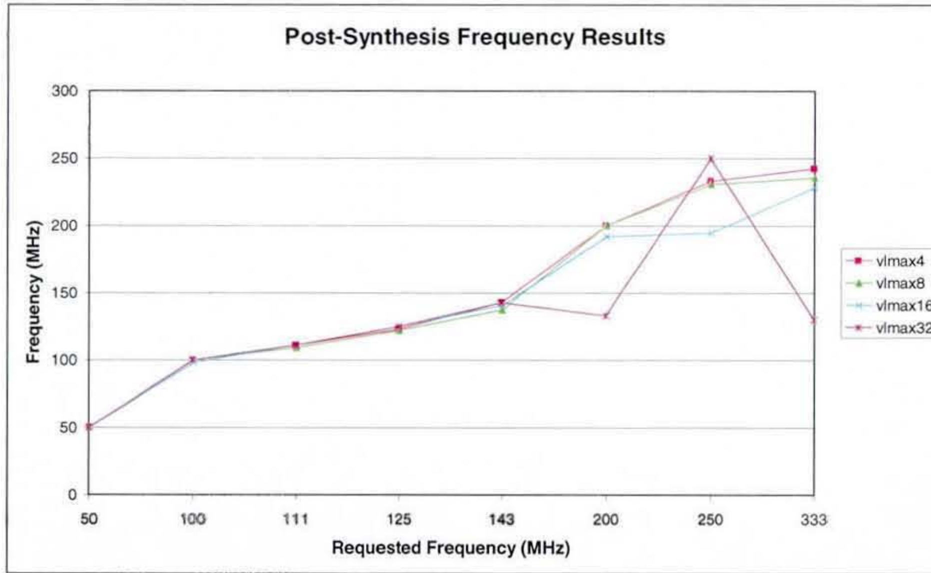


Figure 7-7: Frequency results of vector datapath for different vector lengths

Figure 7-7 illustrates the maximum achievable frequency against the requested frequency for different vector lengths. It is observed that the relationship between the achieved frequency and the requested is near-linear for frequencies up to 333 MHz and vector lengths from 4 to 16. For VLMAX 32 and frequencies below 200 MHZ the same near-linear relationship is observed. As the requested frequency is increased above 200 MHz, the achieved frequency becomes more unpredictable due to the enormous size of the netlist optimised by DC in a top-down mode.

7.4 Implementation Campaign for Vector Coprocessor

The statistical power analysis was performed for the vector coprocessor as a whole. This includes the vector datapath (previous section) and the VLSU (other project[1]) unit. Figure

[1] This is a parallel running project, addressing the design of the Vector Load/Store Unit of the processor.

7-8 illustrates the power consumption for different VLMAX and periods (frequencies). From the results it can be seen that as the requested period (frequency) increases the amount of the power dissipated increases proportionally. This direct relationship is due to the number of switching gates and their size, as the latter is affected with increasing the requested clock frequency. The higher the frequency, the higher the capacity load switching, which leads to the rise in the dynamic power.

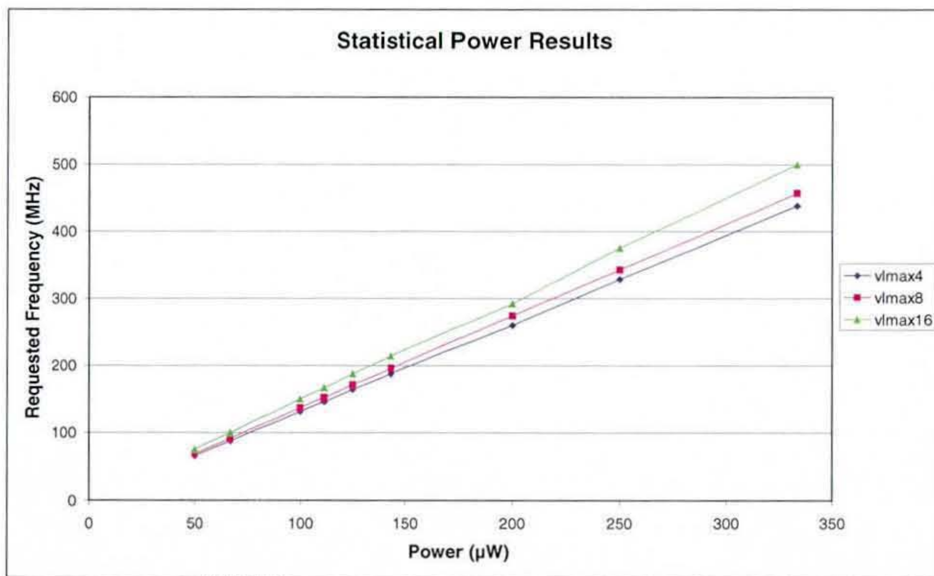


Figure 7-8: Statistical power results of vector coprocessor for different vector lengths

For different VLMAX the graph shows a marginal change for the dissipated power. This is because the size of the VLSU is much larger than the vector coprocessor and consequently the power it dissipates. At low requested periods the difference in statistical power between VLMAX 4 and VLMAX 16 is approximately 12.3% which seems constant over the frequency range. The statistical power results were obtained up to VLMAX 16 and reveal that the power consumption increase is a fairly constant 6.6 fold for the period range of 20ns (50MHz) to 3ns (333MHz). No results were obtained for higher VLMAX as the design was too large for the synthesis run to complete successfully. Apart from the power consumption the physical area of the vector coprocessor design was also recorded. Figure 7-9 depicts the area for different values of VLMAX and for various requested periods (frequencies) for the vector coprocessor design. Again the required area for a given VLMAX shows a marginal change for all the frequency range, as the silicon area is proportional to the number of gates that perform

the functionality of the design. As the frequency increases however there is a slight rise in the silicon area as the timing optimization methods affect the whole design layout.

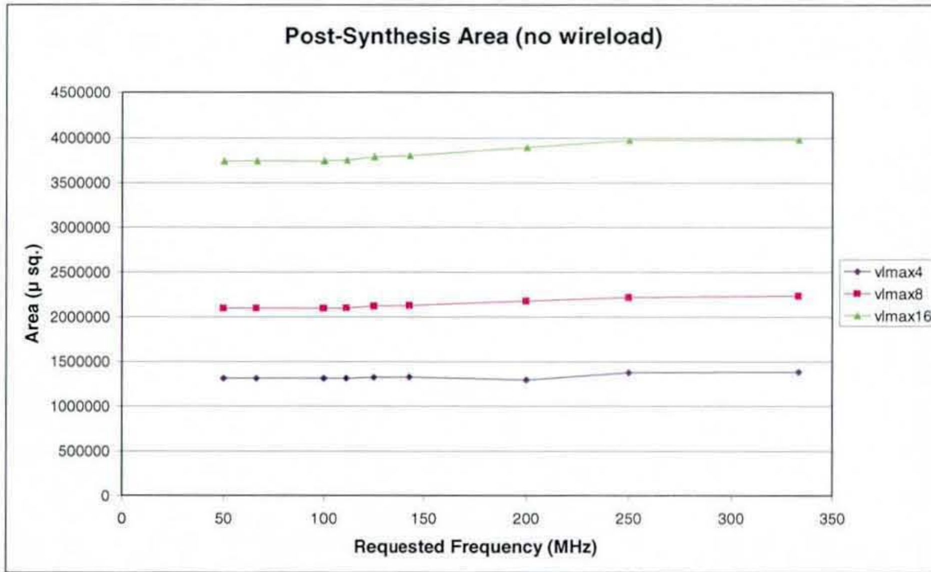


Figure 7-9: Statistical area results of vector coprocessor for different vector lengths

Additionally from the graph it can be seen that the area is directly related to the vector length (VLMAX). This was expected as the vector length affects the number of the functional units in the vector datapath along with the size of the register files and the VLSU unit. The last graph in Figure 7-10 illustrates the achievable frequency against the requested frequency for different vector lengths of the whole vector coprocessor. As it can be seen the achievable frequency matches or even is higher than the requested for frequencies up to 200 MHz and vector lengths from 4 to 16. For higher frequencies however logical synthesis is unable to achieve the requested frequency an effect exacerbated at higher vector lengths. This is due to the increased design size which can't be handled efficiently by the synthesis tool.

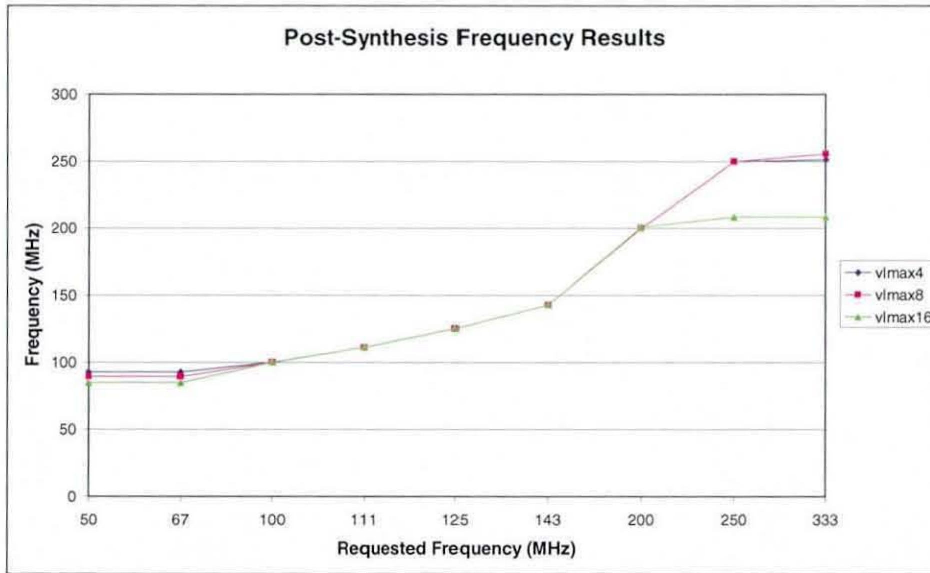


Figure 7-10: Frequency results of vector coprocessor for different vector lengths

From the graph it can be observed that the maximum operational frequency for the vector coprocessor is 256 MHz for vector lengths up to 8 and 208 MHz for a vector length of 16. These figures fall well within the acceptable range of high performance industrial-level ASIC design for the given silicon technology.

7.5 VLSI Layout

The following sections present the resulting VLSI macrocells along with their physical characteristics for the Vector Datapath and the Vector Processor designs respectively.

7.5.1 Vector Datapath Layout for VLMAX 16

The vector datapath with VLMAX=16 (256-bit length) was taken through the full front end (logical synthesis) and the back end (Place and Route) flows. The design was read into Synopsys design compiler and synthesized for a target frequency of 250 MHz, targeting the TSMC 0.13µm (1 Poly, 8 Copper) process. A top-down flow and no wireload models were used. This flow was chosen as our experience shows that the back end tool (Cadence SoC Encounter) is capable of very advanced netlist re-synthesis thus making the use of front end wireload models unnecessary. After synthesis the optimized netlist of the vector datapath with length 256 bits was imported into SoC Encounter and

the flat physical flow was carried out. The physical characteristics of the VLSI cell are given in Table 7-1.

Table 7-1: VLSI Layout physical parameters for VDP with VLMAX 16

Parameters	Value
X dim (μm)	1010
Y dim (μm)	1010
Area (mm sq)	1.02
Cells (RAMs)	63945 (7)
Cell rows	279
Speed (MHz)	186.2

The VLSI results show a worst case (0.9V, 125 C) maximum frequency of 186.2 MHz post-route. The achieved frequency is well within the domain of high performance implementations of wide parallel processors.

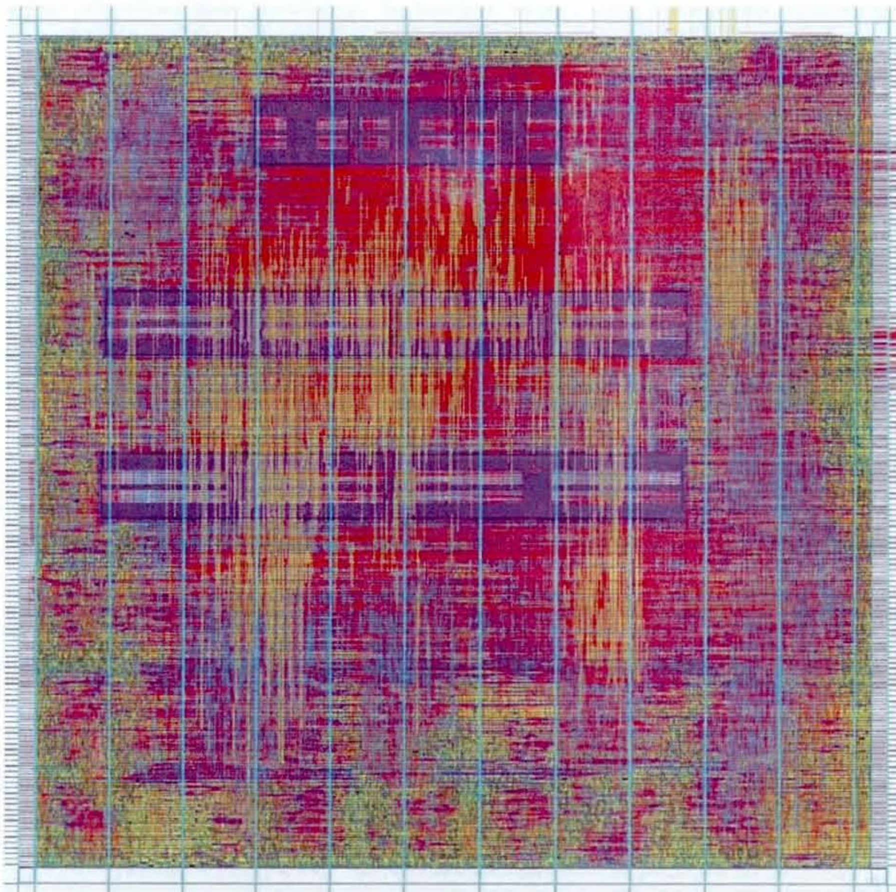


Figure 7-11: Vector Datapath macrocell for VLMAX 16

It is anticipated that further work at the back-end will result in a substantially faster cell. The power consumption based on statistical activity (not workload-based) of the cell is also moderate; at 61.3 mW when optimized for 4ns period. The design includes approximately 64 K gates, 7 RAM macros in 279 standard cells rows. The cell area is 1.01 by 1.01 mm^2. The resulting VLSI macrocell is shown in Figure 7-11.

7.5.2 Vector Datapath Layout for VLMAX 32

The same methodology was followed for the vector datapath with VLMAX=32 (512-bit length). The design was synthesized for a target frequency of 200 MHz, targeting the TSMC 0.13µm (1 Poly, 8 Copper) process. The physical characteristics of the VLSI cell are given in Table 7-2.

Table 7-2: VLSI Layout physical parameters for VDP with VLMAX 32

Parameters	Value
X dim (µm)	1801
Y dim (µm)	1800
Area (mm sq)	3.24
Cells (RAMs)	209809 (11)
Cell rows	453
Speed (MHz)	126.7

Again no wireload models were used and the physical flow was carried out for the vector datapath with length 512 bits. The resulting VLSI macrocell is shown in Figure 7-12.

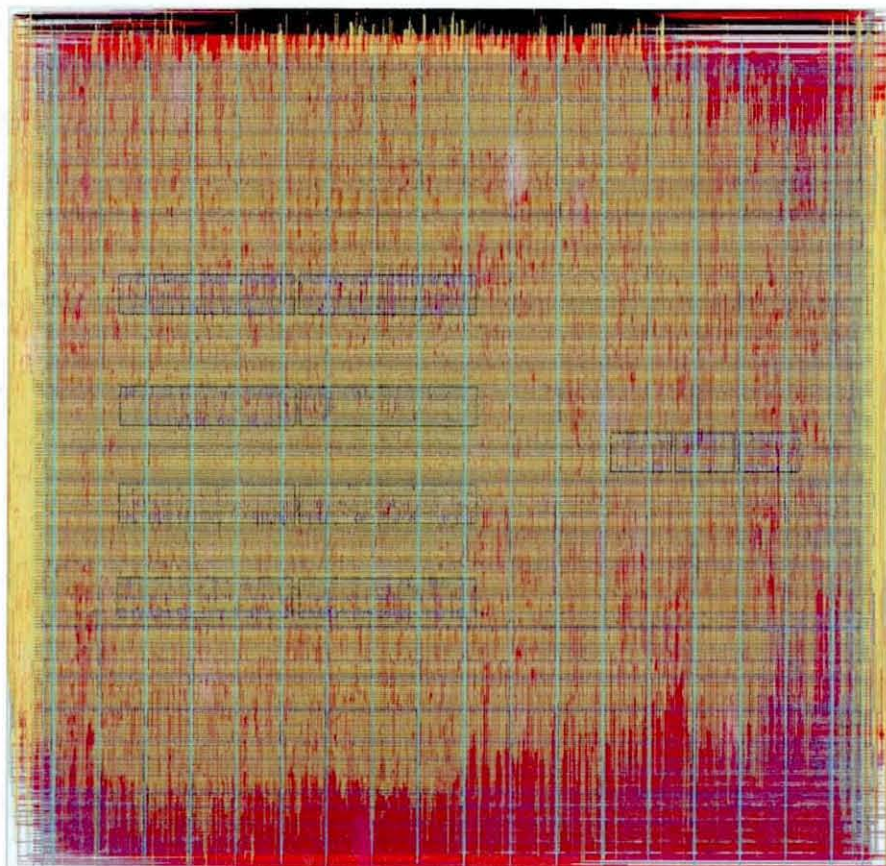


Figure 7-12: Vector Datapath macrocell for VLMAX 32

The design achieved a much lower frequency of 126.7 MHz post-route, worst case (0.9V, 125 C) maximum frequency when optimised for 5ns period. This discrepancy between logical synthesis (200MHz) and final post-route speed (126.7MHz) is attributed to very wide datapath (512 bits) which resulted in a substantially congested VLSI macro. The VLSI macro includes approximately 210 K gates, 11 RAM macros in 453 standard cells rows. The cell area is 1.8 by 1.8 mm^2.

7.5.3 Vector Processor Layout for VLMAX 16

Finally, the full vector processor (incorporating the Vector Datapath and the VLSU) with VLMAX=16 and Vector Data Cache configuration 4-way, 8Kbytes, 128 bytes block length and 2 sub-blocks per block, was taken through the full front end (logical synthesis) and the back end (Place and Route) flows. The design synthesized for a target frequency of 200 MHz, targeting the TSMC 0.13μm (1 Poly, 8 Copper) process.

Table 7-3: VLSI Layout physical parameters for VCOP with VLMAX 16

Parameters	Value
X dim (μm)	1802
Y dim (μm)	3491
Area (mm sq)	6.29
Cells (RAMs)	257308 (22)
Cell rows	921
Speed (MHz)	182

A top-down flow and no wireload models were used. After synthesis the optimized netlist was imported into SoC Encounter and the physical flow was carried out with the two major partitions being the vector datapath and Vector Load/Store Unit (VLSU). The resulting VLSI macrocell is shown in Figure 7-13.

Figure 7-13: Layout for the whole vector processor (vector datapath and VLSU unit)

The physical characteristics of the VLSI cell are given in Table 7-3. The design includes approximately 257 K gates, 22 RAM macros in 921 standard cells rows. The cell area is 1.8 by 3.5 mm^2. The design achieved 182 MHz post-route, worst case (0.9V, 125 C) maximum frequency that clearly indicates that the critical path lies within the Vector Datapath.

7.6 ESL Implementation

This section discusses briefly the SystemC-based methodology, which automatically generates a technology independent Verilog netlist from the vector instructions of a vectorized application. This application involves both ITU-T speech coders, G.729A and G.723.1. The vector instruction set extensions, which were described in Chapter 5, were formed by C-source vector macro-opcodes and were introduced to a next-generation multi-parallel, configurable application-specific processor known as SS_SPARC. The SS_SPARC platform along with the ESL methodology and the statistical power analysis results obtained from the SystemC-accelerator synthesis and the handed-code RTL synthesis are presented in the following sections [6].

7.6.1 SS_SPARC Platform

SS_SPARC is a configurable, extensible, chip multi-processor where each processor is a 5-issue, simultaneous multithreaded vector processor [6]. A high-level view of a 3-instance SS_SPARC kernel is depicted in Figure 7-1.

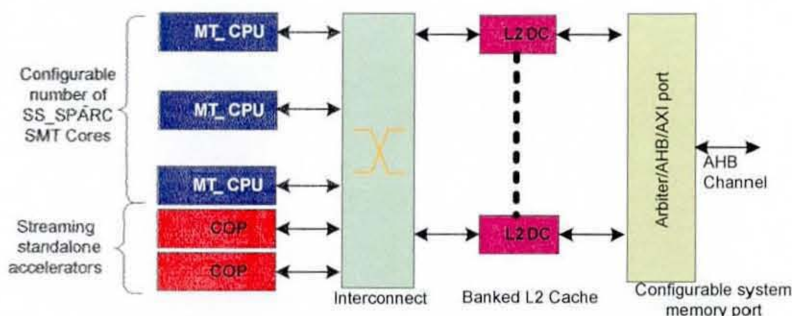


Figure 7-14: High level view of a 3-instance SS_SPARC kernel

The SS_SPARC platform consists of a configurable number of SMT processing units, a number of user-defined, loosely-coupled coprocessors, a pipelined switch matrix, and a

multi-banked, level-2 memory system with a standard AHB interface. Additionally, a generic, transaction-level-pipelined memory interface which connects to the next generation AMBA 3 Advanced eXtensible Interface (AXI) [7] standard is available. The design is parameterized as to the number of SMT processing units, the number of contexts per processor unit, the vector infrastructure, the instruction and data caches configuration and buffering schemes and the switch matrix configuration [6]. Figure 7-15 illustrates the schematic diagram of the superscalar pipeline of a SMT processing unit. It comprises the instruction Front-End (IFE), the scalar core (SCORE), the vector core (VCORE) and the load/store unit (LSU).

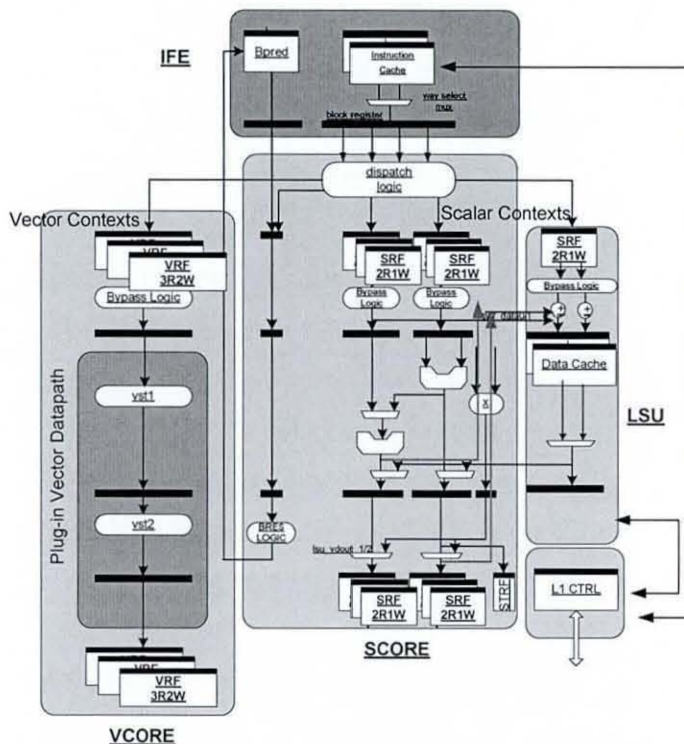


Figure 7-15: Superscalar SMT pipeline organisation

The IFE consists of a configurable, multi-way instruction cache (ICache) and supplies an instruction block (5 instructions) per cycle to the per-context instruction buffers. A programmable arbitration mechanism is employed to select one of the non-blocked contexts. The ICache services one block request per cycle and supports pipelined transactions to the main memory. In case of a cache miss only the particular context is blocked while the remainder are allowed to proceed. The employed branch predictor is configurable as to the numbers of branches it can predict per cache block and it is

relatively simple with good prediction rate in the computationally intensive loops dominant workloads of the telecoms domain. After the instruction buffers there is a dispatch logic which checks the buffered instruction per processing unit to resolve data dependences and prepare the instruction packet for execution. The instruction packet is dispatched to the register/bypass stage in the SCORE block, for subsequent context prioritization and transfer to the execution block [6].

The scalar core (SCORE) block consists of the microarchitectural units equal with the number of supported contexts, the context selection unit (CCU) and a 3-stage pipeline that implements the Sparc V8 ISA [8]. The instructions that were dispatched in the previous cycle access per-context the register files. These instructions are prioritized by the CCU, and progress to the registers of the execution datapath. The datapath comprises two 32-bit integer ALUs in a cascade configuration. Figure 7-16 illustrates the SCORE pipeline organization [6].

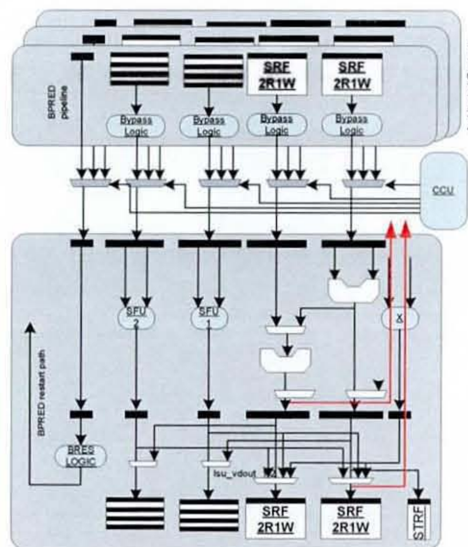


Figure 7-16: Scalar core (SCORE) pipeline organization

The dual-pipeline vector core (VCORE) is highly configurable and extensible for the architecture (programmer's model and ISA) as well as the microarchitecture (width of vector registers, number of stages of the vector pipeline, bypassing etc) and it is the primary DSP engine [6]. In the first pipeline, custom instructions can be easily inserted as 'plug-in datapaths' in the vector core by using the exposed interface of the latter. The second pipeline is dedicated to returning vector loads from the high-bandwidth LSU and

it is not accessible from the system architect. As shown in Figure 7-17 the vector core comprises the architected state (one per context), the vector bypass logic, and a configurable number of vector execute stages for the custom datapath. In a multi-context configuration, multiple threads access the architected state of the processor. In the case that there are no register or resource dependences, multiple contexts are prepared to be dispatched to the single-issue vector datapath. The CCU arbitrates the ready CPU contexts by using context arbitration algorithm and issues one to the vector pipeline. The results are made available (via bypassing) to dependent vector instructions. The exposed microarchitecture allows the system architect to design and implement custom instructions using a number of methodologies including RTL-based and, ESL-based. The interfaces that facilitate this are: a) the Dispatch IF that is the input interface to the user defined vector datapath b) the Bypass IF consists of the vector result buses, one per stage, vector masks and valid strobes to determine the bypass paths c) the LSU return path IF is the entry point of the return vector load from the LSU d) the write-back IF is the point where the produced vector results (two per cycle) from the vector datapath are passed to the vector register file of the specific context for writing [6].

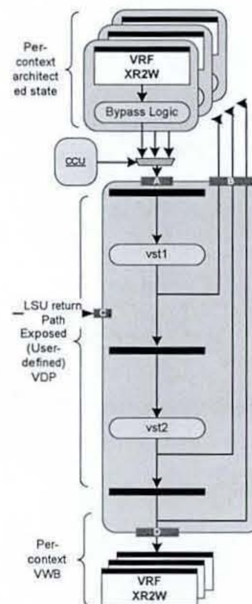


Figure 7-17: Dual-pipeline vector unit organization

7.6.2 ESL Methodology

The input of the flow of the developed methodology is the vectorized source code of the ITU-T G.729A and G.723.1 speech coders. The vectorization was performed by using a number of assembly-like C-macros. The C-level macros define precisely the vector instruction set extensions that were described in Chapter 5. The custom flow parses these C-macros and creates a SystemC module that instantiates these SIMD instructions. The SystemC model is verified by using the test vectors that were produced by running the vectorized algorithm in order to ensure that this "packing" of the SIMD ISA hasn't change the functionality of the operations. A number of pipeline registers and the bypass taps are specified in the synthesis tool. The SystemC datapath is then synthesized to technology independent gates RTL-VHDL using a commercial SystemC synthesizer. Afterwards the RTL model is validated again by the same test vectors (as they applied before) to ensure that the SystemC-RTL transformation was successful. The resulting RTL datapath is instantiated in the exposed vector unit of the SS_SPARC processor and further decoding logic is added to the core processor to enable the execution of these extensions [6]. The combined RTL (vector extensions and SS_SPARC platform) goes to the standard design flow which was described in sections 7.2.1 to 7.2.3. The results from the statistical power analysis results for both the SystemC-accelerator and the RTL-accelerator synthesis along with a VLSI layout are presented in the following section.

7.6.3 Micro-Architecture Results

In this work the statistical power consumption and the area were obtained for the SystemC-defined accelerators as well as the RTL-accelerators. Figure 7-18 depicts the power consumption of both implementations for all the configurations: vector length 256-bit (VLMAX 16) and 512-bit (VLMAX 32), vector contexts 1, 2, 4 and 8 for different clock periods. In this figure each requested period is plotted against its corresponding power. From the set of the results it is obvious that the general shape of the graphs is of a similar nature.

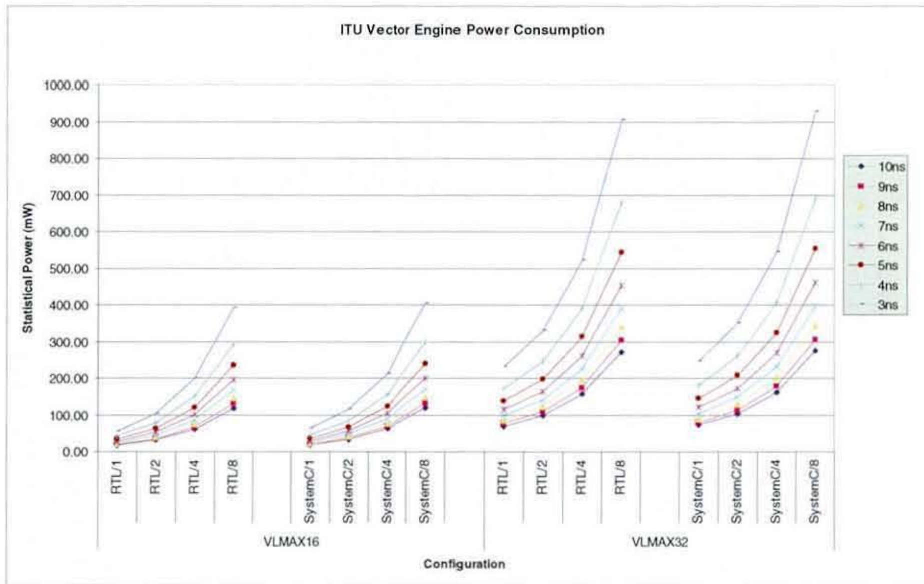


Figure 7-18: ITU VCore Power Results

The SystemC-accelerators shows a pre-route overhead of 3% to 15% compared to the hand-coded (RTL) designs over the synthesis campaign. These results demonstrate that the SystemC synthesis is fairly reliable and can achieve power consumption close to the traditional RTL synthesis [6]. Additionally, from the RTL results it can be seen that the power consumption is affected significantly from the vector length (VLMAX). The power consumption shows a 4 fold increase for context 1 between VLMAX 16 and VLMAX 32 whereas the increase for context 8 is 2 fold between these vector lengths.

Figure 7-19 shows the pre-route area of both sets of accelerators also for all configurations. In this case, the SystemC-implementation exhibited even better area usage characteristics with a reduction in the range of 2% to 18% compared to the hand-coded (RTL) designs. This is due to the fact that the SystemC synthesizer that makes more intelligent resource allocation compared to the traditional RTL design flow [6]. Additionally from the graph it can be seen that the area is directly related to the vector length (VLMAX). From the results it can be seen that there is a fairly constant two fold increase in area allocation between VLMAX 16 and VLMAX 32 and for all the range of contexts. This was expected as the vector length affects the number and the width of the vector datapaths that for VLMAX 32 is double.

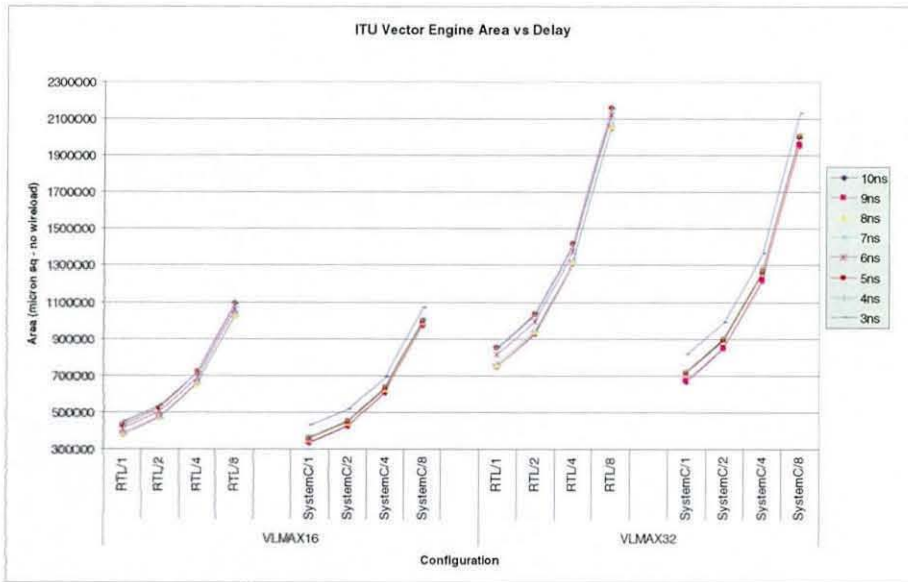


Figure 7-19: ITU VCore Area-Delay Results

The SystemC-defined datapath configuration (VLMAX=32, T_{arget}=250 MHz) was through the entire flow to a VLSI macro. The resulting VCORE (including the datapath, the vector contexts, all multiplexing/bypassing and the LSU return path) is shown in Figure 7-20. The design includes approximately 70K gates and six 16x128-bit dual-port RAM macros, three for each vector register file of the two CPU contexts. A two-stage pipelined architecture was specified which resulted in a worst-case (0.9V, 125C) maximum frequency of 213 MHz [6].

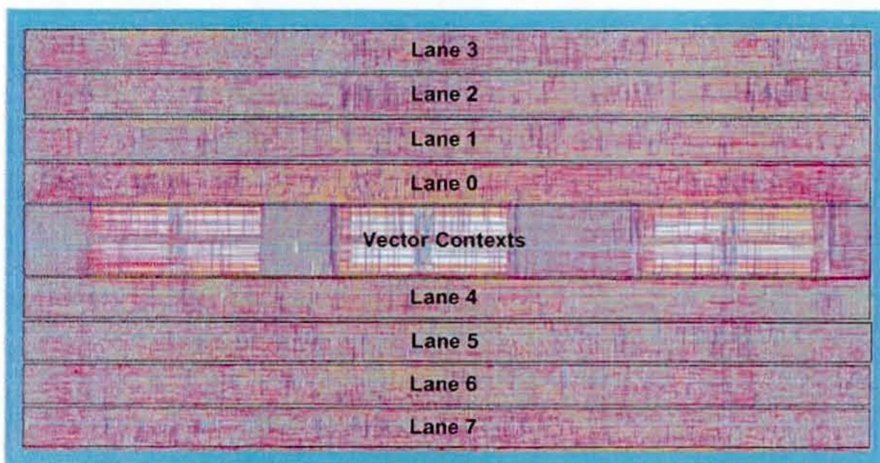


Figure 7-20: Two-context, 256-bit ITU vector engine

7.7 Summary

This chapter discussed the verification methodology used to validate the vector processor and its associated units along with the synthesis and back-end flow of the vector datapath. Statistical power/area/frequency results were presented for the vector datapath and the vector coprocessor as a whole for different configurations (VLMAX, frequency) after a scripted synthesis/place-and route campaign. The VLSI layouts and their physical parameters of the vector datapath and the vector processor were also illustrated. This was followed by the description of the SS_SPARC ASIC platform, the SystemC modelling of the vector instruction set extensions and their subsequent synthesis to low-level RTL. The ESL-implemented of the vector extensions was inserted after to the exposed vector engine of the SS_SPARC processor and statistical power analysis results for both the SystemC-accelerator and the RTL-accelerator datapaths were presented and compared.

7.8 References

[1] S. Akella, "Guidelines For Design Synthesis Using Synopsys Design Compiler," Department of Computer Science Engineering, University of South Carolina, Columbia, December 2000.

[2] G. R. Beck, D. W. L. Yen, and T. L. Anderson., "The Cydra 5 minisupercomputer: Architecture and implementation," *The Journal of Supercomputing*, vol. 7, pp. 143-180, May 1993.

[3] "Design Compiler 2003.06," Synopsys Inc., 2003.

[4] "Advanced Logic Technology - 0.13μm," Taiwan Semiconductor Manufacturing Company, 2006.

[5] C. Kozyrakis, "A Media-Enhanced Vector Architecture for Embedded Memory Systems," Technical Report: CSD-99-1059, University of California at Berkeley 1999.

[6] V. A. Chouliaras, K. Koutsomyti, T. Jacobs, et al., "SystemC-defined SIMD instructions for high SystemC-defined SIMD instructions for high," in *13th IEEE International Conference on Electronics, Circuits and Systems*, Nice, France, 2006, pp. 822-825.

[7] "AMBA AXI Specification," http://www.arm.com/armtech/AXI.

[8] "The Sparc Architecture Manual Version 8 ", www.sparc.com.

CHAPTER 8
CONCLUSIONS

The aim of this thesis was to study the potential acceleration of both speech coding algorithms, namely G.729A and G.723.1, through their efficient implementation on a configurable extensible vector embedded CPU architecture. The outcome of this work was the optimization of both C reference codes and the design and implementation of a parametric (configurable) vector processor, to explore the effects of different configurations (VLMAX, number of registers and accumulators) and thus, probe the microarchitecture space. The optimized reference codes and the vector architecture were fully validated with the use of the ITU-supplied test vectors. This chapter presents the main contributions of this research and proposes further work which leads on from this project.

8.1 Contribution of this thesis

At the beginning of this work and in order to investigate the potential acceleration of both speech codecs, the profiling of both C reference codes was performed to identify the computation workload distribution. This revealed that the most CPU-intensive parts of the codes were in the DSP emulation functions (e.g. in G.723.1 decoder 66.7% of the total machine instructions) of the reference implementations. Additionally, these algorithms exhibited a large amount of data-level parallelism. Therefore it was decided that efficient implementation of these basic operations in the form of a configurable vector processor with a targeted, data-parallel architecture, could achieve a leading area/power/cost result.

An optimization methodology was developed, in which custom vector and scalar ISA extensions were identified and inserted into both reference codes in place of the DLP-loops and other non-vectorizable parts of the codes respectively. The optimized codes were verified and run on the SimpleScalar toolset for all ITU-T test vectors, over a range of vector lengths, to evaluate the performance of the vector architecture prior its implementation in hardware. For this purpose the simulator was modified and extended to

include the added state (coprocessor scalar and vector state) and the scalar and vector extensions.

The architectural results were very promising, demonstrating a reduction in the dynamic instruction count metric of 58% and 71% for G.729A and G.723.1 speech coders respectively when the vector instructions were introduced and a further 18% and 9% reduction in dynamic instruction count when the scalar instructions were applied. The overall simulation results indicated that the area/performance points of interest lie in between 64-bit (VLMAX 4) to 256-bit (VLMAX 16) wide configurations as there was not much more improvement over a vector data length of 16 (256 bits) due to the size of the speech frames. These speech codecs operate on frames (blocks) of 240 samples and these frames are also divided into subframes of 60 samples and hence fast performance improvement can be seen for lower vector lengths. At vector length of 4, the coprocessor would save 71.6% of the dynamic instruction count of the G.729A encoder and almost 75% for the G.723.1 encoder. For vector length 16, the coprocessor would only save another 4.4% and 5% for G.729A and G.723.1 respectively and no significant improvement emerges beyond that. In addition both sets of results revealed that the maximum benefit is achieved by the combination of custom vector and scalar architectures. These results conclusively showed the potential benefit of applying custom instructions and having associated coprocessor vector functional units.

Another aspect of this work was the SystemC modelling of the vector instruction set extensions and their subsequent synthesis info low-level RTL. This work was undertaken to explore faster routes to silicon for SIMD extensions, compared to the established RTL flow. These ESL-implemented vector extensions were inserted into the exposed vector engine of the SS_SPARC ASIC processor and statistical power analysis results, for both the SystemC-accelerator and the RTL-accelerator datapaths, were presented and compared. From the synthesis results it was shown that the SystemC synthesis was fairly reliable and achieved power consumption close to the traditional RTL synthesis.

The main contribution of this research project was the full design and implementation of the proposed vector datapath of the vector processor. The vector pipeline is a SIMD array of functional units with a configurable 2-way SIMD or scalar organization. It has a four stage-pipeline organization and it is parameterised along both the architecture and the

microarchitecture axes. Few modifications took place to the Leon3 pipeline to enable its tight-coupling to the vector processor.

The vector datapath was verified by using an FLI-based testbench that applied the ITU-supplied test vectors. Finally, statistical power/area/frequency results were obtained for the vector datapath and the vector coprocessor as a whole for different configurations (VLMAX, frequency) after a scripted synthesis/place-and route campaign. In addition, the VLSI layouts and their physical parameters of the vector datapath and the vector processor were obtained. From these results, the vector datapath with VLMAX=16 configuration showed a worst case (0.9V, 125C) maximum frequency of 186.2MHz, area 1.02 mm^2 and power of 61.3 mW. The whole vector coprocessor with VLMAX=16 and vector data cache configuration 4-way, 8Kbytes, 128 bytes block length and 2 sub-blocks per block achieved maximum frequency of 182MHz, area of 6.29 mm^2 and power of 74.97 mw.

8.2 Suggestions for future research

The vector processor was developed to efficiently execute the G.729A and G.723.1 speech coding standards in an embedded application. Since its vector and scalar ISA are based on the basic operations of these algorithms, all the ITU G.7xx speech coding standards which share the same (or a subset) emulation operations such as G.711, G.726, G.727, G.728 and G.729 can also be accelerated by adapting them for this vector processor. This adaptation involves optimization with the insertion of vector and scalar extensions.

The developed vector processor can be attached to any scalar CPU with very little modifications in its interface. This gives it the great advantage of being able to interface to different architectures and ASIC platforms. Thus allows further research on novel multimedia architectures that incorporate VoIP/speech coding functionality.

Since the VLSU unit is also parametric, several different configurations can be implemented and their performance in terms of area and power dissipation investigated. In addition, entirely different VLSUs can be attached to the vector datapath with cascade or parallel TAG/DATA organization with few modifications to their interface with the

vector datapath. As the current VLSU has cascade TAG/DATA organization an extra signal in the output multiplexer of the VDP1 stage needs to be added. This signal will select the return load data from the VLSU at the end of the VDP1 stage as the load takes only one cycle for a parallel TAG/DATA configuration instead of two which is the case for the cascade configuration.

As already discussed, the vector coprocessor implementation is technology independent therefore it can be re-targeted to different silicon technologies. The multiple configurations (VLMAX, number of registers and accumulators) lead to different statistical power/area/frequency points thus covering a large part of the implementation spectrum.

Another area of research would be to investigate the benefits of ESL techniques instead of programmable architectures by coupling the ESL-implemented vector datapath to other ESL defined architectures.

As multimedia applications consist of more than one time-critical execution threads there is a significant amount of coarse-grained parallelism. Therefore by attaching the vector coprocessor to a multithreaded architecture could accelerate even more multimedia-rich applications that incorporate speech coding [1].

Another interesting approach will be an architecture that combines the best of ILP and DLP techniques for an optimal implementation. This architecture would combine vector instructions with out-of-order execution with register renaming and even simultaneous multithreaded execution. Such implementations are very promising according to Espasa [2] and Quintana [3] and the Tarantula project [4] in which a vector unit is attached to the superscalar Alpha engine. This is also the domain of the SS_SPARC processor [5].

8.3 References

[1] K. Diefendorff and P. Dubey, "How Multimedia Workloads Will Change Processor Design," in *IEEE Computer*. vol. 30, September 1997, pp. 43-45.

[2] R. Espasa and M. Valero, "Exploiting Instruction- and Data-Level Parallelism," in *IEEE Micro*. vol. 17, September 1997, pp. 20-27.

[3] Francisca Quintana, Roger Espasa, and Mateo Valero, "A Case for Merging the ILP and DLP Paradigms," in *6th Euromicro Workshop on Parallel and Distributed Processing*, Madrid, Spain, 1998, pp. 217-224.

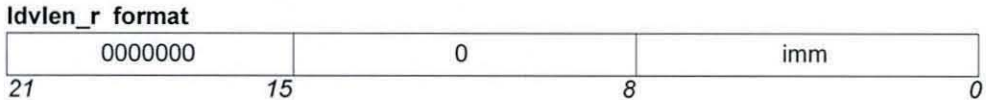
[4] R. Espasa, F. Ardanaz, J. Gago, et al., "Tarantula: A Vector Extension to the Alpha Architecture " in *the Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)* Anchorage, Alaska, 2002, pp. 281-292.

[5] V. A. Chouliaras, K. Koutsomyti, T. Jacobs, et al., "SystemC-defined SIMD instructions for high performance SoC architectures," in *13th IEEE International Conference on Electronics, Circuits and Systems*, Nice, France, December 2006, pp. 822-825.

APPENDIX A VECTOR AND SCALAR ISA

ldvlen_r

Instruction Format



Syntax

`ldvlen_r(`*`imm`*`)`

where:
imm is a numeric constant

Description

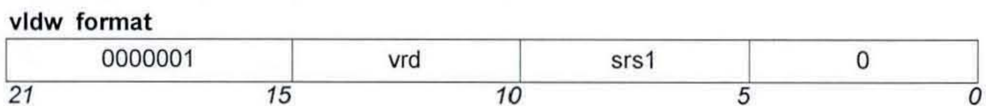
The `vlen_r` instruction loads an immediate into the Vector Length Register

Example

```
ldvlen_r(16); //Vector Length Register is set to 16
```

vldw

Instruction Format



Syntax

`vldw(`*`vrd, srs1`*`)`

where:
vrd is the destination vector register
`srs1` is the address of the variable in memory

Description

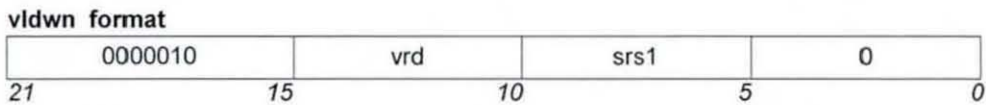
The `vldw` instruction loads the vector register `vrd` from memory address given in scalar register `srs1`.

Example

```
vldw(2, 3); //Load vreg2 from address given in sreg3
```

vldwn

Instruction Format



vldwn format			
0000010	vrd	srs1	0
21 — 15	10	5	0

Syntax

`vldwn(vrd, srs1)`

where:

vrd	is the destination vector register
srs1	is the address of the variable in memory

Description

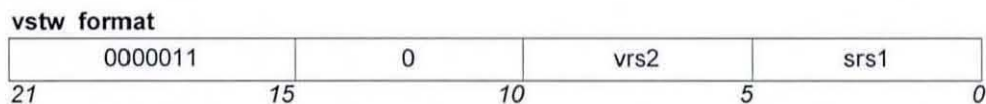
The `vldwn` instruction loads vector register `vrd` downward from memory address given in scalar register `srs1`.

Example

```
vldwn(2, addr); //Load vreg2 downwards from address given
                   in sreg3
```

vstw

Instruction Format



vstw format			
0000011	0	vrs2	srs1
21 — 15	10	5	0

Syntax

`vstw(vrs2, srs1)`

where:

vrs2	is the source vector register
srs1	is the memory address

Description

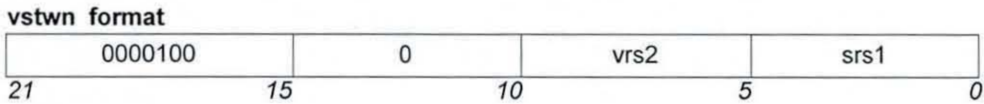
The `vstw` instruction stores the vector register `vrs2` to memory address given from scalar register `srs1`.

Example

```
vstw(3, 1); //Store vreg3 to memory address given in sreg1
```

vstwn

Instruction Format



Syntax

vstwn(*vrs2, srs1*)

where:

vrs2	is the source vector register
srs1	is the memory address

Description

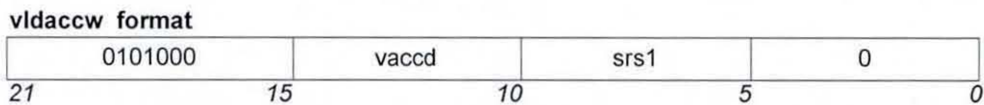
The vstwn instruction stores a vector register downward to memory address

Example

```
vstwn(3, addr); //Store vreg3 downwards to addr
```

vldaccw

Instruction Format



Syntax

vldaccw(*vaccd, srs1*)

where:

vaccd	is the destination vector accumulator
srs1	is the address of the variable in memory

Description

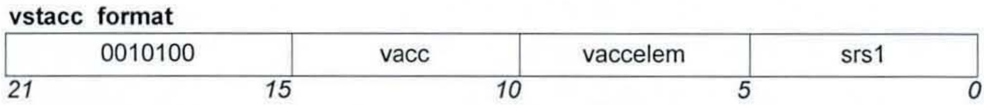
The vldaccw instruction loads 32-bit word to the vector accumulator from memory

Example

```
vldaccw(0, addr); //Load vacc0 from addr
```

vstacc

Instruction Format



Syntax

`vstacc(vacc, velem, srs1)`

where:

vacc	is the source vector accumulator
velem	is the element of the source vector accumulator
srs1	is the memory address

Description

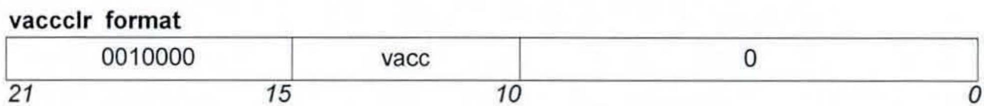
The `vstacc` instruction stores a vector accumulator element (32-bit) to memory

Example

```
vstacc(1,0,addr); //Store element 0 of vacc1 to addr
```

vaccclr

Instruction Format



Syntax

`vacclr(vacc)`

where:

vacc	is the vector accumulator

Description

The `vaccclr` instruction sets the value in the vector accumulator vac to zero (clear)

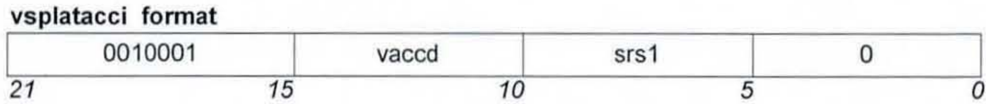
Example

```
vaccclr(1); //Set vacc1 to zero
```

vsplatacci

Instruction Format

vsplatacci format



Syntax

```
vsplatacci(vaccd, srs1)
```

where:

`vaccd`	is the destination vector accumulator
`srs1`	is the value (32-bits) that is splated into the vector accumulator

Description

The `vsplatacci` instruction splats the 32-bit word scalar value into the vector accumulator.

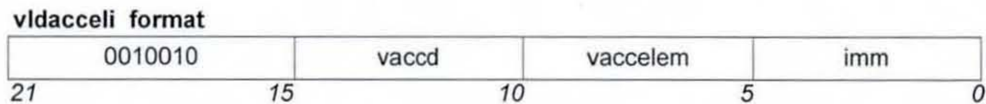
Example

```
vsplatacci(0, 3); //Splat vacc0 with the value of the
                    scalar register 3
```

vldacceli

Instruction Format

vldacceli format



Syntax

```
vldacceli(vaccd, velem, imm)
```

where:

`vaccd`	is the destination vector accumulator
`velem`	is the destination element of the vector accumulator
`imm`	is the immediate to be loaded

Description

The `vldacceli` instruction loads an immediate value into a vector accumulator element

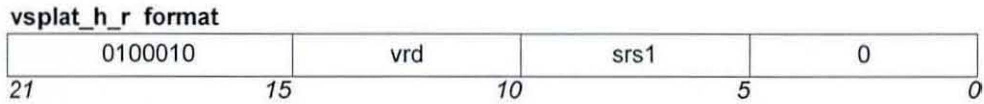
Example

```
vldacceli(1,0,16); //Load immediate 16 into element 0 of
                     vacc1
```

vsplat_h_r

Instruction Format

vsplat_h_r format



Syntax

```
vsplat_h_r(vrd, srs1)
```

where:

vrd	is the destination vector register
srs1	is the scalar register value

Description

The `vsplat_h_r` instruction splats a 16-bit word of scalar register `srs1` to all the elements of vector register `vrd`.

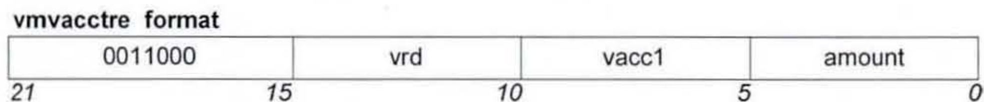
Example

```
vsplat_h_r(1,3); //Splat 16-bit value of sreg3 to vreg1
```

vmvacctre

Instruction Format

vmvacctre format



Syntax

```
vmvacctre(vrd, vacc1, amount)
```

where:

vrd	is the destination vector register
vacc	is the vector accumulator
amount	is the shift amount

Description

The `vmvacctre` instruction extracts high (amount=0) or low (amount=16) the even elements of vector accumulator and loads them into the even elements of the vector register `vrd`

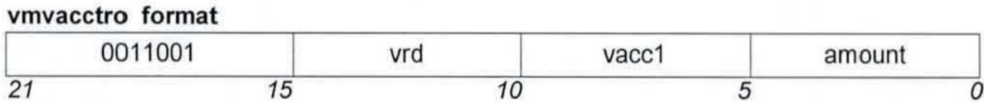
Example

```
vmvacctre(2,1,16); //Extracts high the even elements of
```

```
                    vacc1 and loads them to vreg2
```

vmvacctro

Instruction Format



Syntax

`vmvacctro(vrd, vacc1, amount)`

where:

vrd	is the destination vector register
vacc	is the vector accumulator
amount	is the shift amount

Description

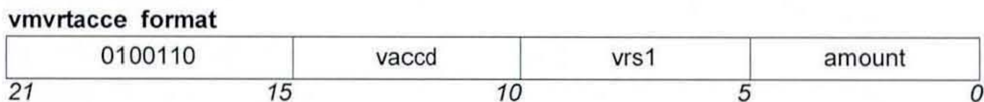
The vmvacctro instruction extracts high (amount=0) or low (amount=16) the odd elements of vector accumulator and loads them into the even elements of the vector register vrd.

Example

```
vmvacctro(3,0,0); //Extracts low the odd elements of
                    vacc0 and loads them to vreg3
```

vmvrtacce

Instruction Format



Syntax

`vmvrtacce(vaccd, vrs1, amount)`

where:

vaccd	is the destination vector accumulator
vrs1	is the destination vector register
amount	is the shift amount

Description

The `vmvrtacce` instruction deposits high (amount=16) or low (amount=0) the even elements of vector register to the vector accumulator.

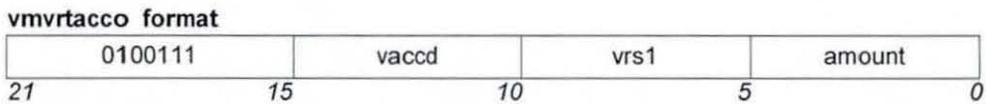
Example

```
vmvrtacce(0,3,16); //Deposits high the even elements of
                     vreg3 to vacc0
```

vmvrtacco

Instruction Format

vmvrtacco format



Syntax

`vmvrtacco(`*`vaccd, vrs1, amount`*`)`

where:

vaccd	is the destination vector accumulator
vrs1	is the destination vector register
amount	is the shift amount

Description

The `vmvrtacco` instruction deposits high (amount=16) or low (amount=0) the odd elements of vector register to the vector accumulator.

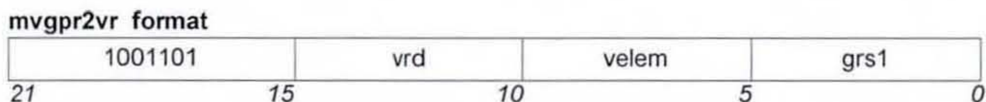
Example

```
vmvrtacco(1,3,0); //Deposits low the odd elements of
                    vreg3 to vacc1
```

mvgpr2vr

Instruction Format

mvgpr2vr format



Syntax

`mvgpr2vr(`*`vrd, velem, grs1`*`)`

where:

vrd	is the destination vector register

`velem` is the destination element of the vector register
`grs1` is the source general purpose register (Leon)

Description

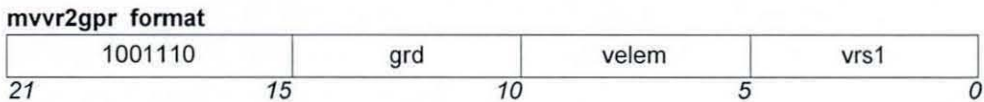
The `mvgpr2vr` instruction moves the scalar contents (32-bit) of the general purpose register to the vector register element.

Example

```
mvgpr2vr(1,2,5); //Move the contents of the general
                   purpose register 5 to the 2nd element
                   of vreg1
```

mvvr2gpr

Instruction Format



mvvr2gpr format

1001110	grd	velem	vrs1

21 … 15 … 10 … 5 … 0

Syntax

`mvvr2gpr(`*`grd, velem, vrs1`*`)`

where:
`grd` is the destination general purpose register (Leon)
`velem` is the vector register element
`vrs1` is the source vector register

Description

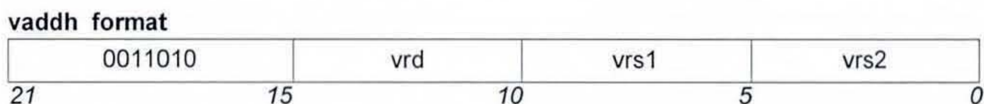
The `mvvr2gpr` instruction moves the contents of the vector register element to the general purpose register (Leon).

Example

```
mvvr2gpr(2,3,5); //Move the contents of the 3d element
                   of vreg5 to the general purpose
                   register 2
```

vaddh

Instruction Format



vaddh format

0011010	vrd	vrs1	vrs2

21 … 15 … 10 … 5 … 0

Syntax

```
vaddh(vrd, vrs1, vrs2)
```

where:

vrd	is the destination vector register
vrs1	is the first source vector register (operand 1)
vrs2	is the second source vector register (operand 2)

Description

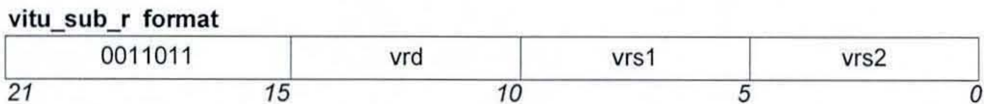
The vaddh instruction performs short addition (16-bit) of source vector registers vrs1 and vrs2 and places the result to the destination vector register vrd.

Example

```
vaddh(5,2,3); //vreg5=vreg2+vreg3 (16-bits)
```

vitu_sub_r

Instruction Format



Syntax

```
vitu_sub_r(vrd, vrs1, vrs2)
```

where:

vrd	is the destination vector register
vrs1	is the first source vector register (operand 1)
vrs2	is the second source vector register (operand 2)

Description

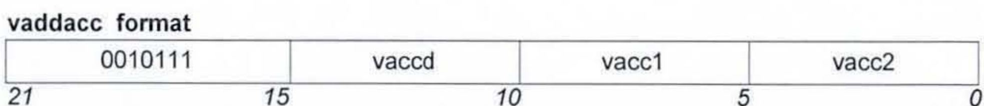
The vitu_sub_r instruction performs short subtraction (16-bit) of source vector registers vrs1 and vrs2 and places the result to the destination vector register vrd.

Example

```
vitu_sub_r(5,2,3); //vreg5=vreg2-vreg3 (16-bits)
```

vaddacc

Instruction Format



Syntax

```
vaddacc(vaccd, vacc1, vacc2)
```

where:

`vaccd` is the destination vector accumulator
`vacc1` is the first source vector accumulator (operand 1)
`vacc2` is the second source vector accumulator (operand 2)

Description

The `vaddacc` instruction performs long addition (32-bit) of source vector accumulators `vacc1` and `vacc2` and places the result to the destination vector accumulator `vaccd`.

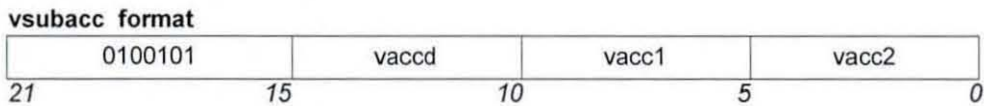
Example

```
vaddacc(0,0,1); //vacc0=vacc0+vacc1 (32-bits)
```

vsubacc

Instruction Format

vsubacc format



Syntax

```
vsubacc(vaccd, vacc1, vacc2)
```

where:

`vaccd` is the destination vector accumulator
`vacc1` is the first source vector accumulator (operand 1)
`vacc2` is the second source vector accumulator (operand 2)

Description

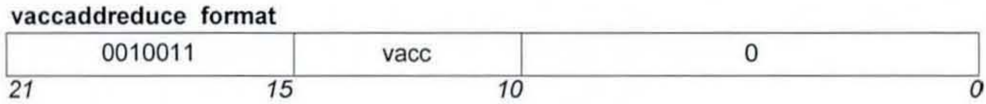
The `vsubacc` instruction performs long subtraction (32-bit) of source vector accumulators `vacc1` and `vacc2` and places the result to the destination vector accumulator `vaccd`.

Example

```
vsubacc(0,0,1); //vacc0=vacc0-vacc1 (32-bits)
```

vaccaddreduce

Instruction Format



Syntax

```
vaccaddreduce(vacc)
```

where:

`vacc` is the vector accumulator

Description

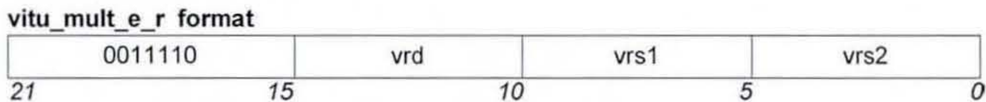
The `vaccaddreduce` instruction add-reduces all the elements of the vector accumulator `vacc` to a 32-bit value that is placed to its zero element.

Example

```
vaccaddreduce(1); //Add-reduce vector accumulator 1
```

vitu_mult_e_r

Instruction Format



Syntax

```
vitu_mult_e_r(vrd, vrs1, vrs2)
```

where:

`vrd` is the destination vector register
`vrs1` is the first source vector register (operand 1)
`vrs2` is the second source vector register (operand 2)

Description

The `vitu_mult_e_r` instruction performs signed short multiplication (16-bit) to the even elements of the source vector registers `vrs1` and `vrs2` and places the result to the even elements of the destination vector register `vrd`.

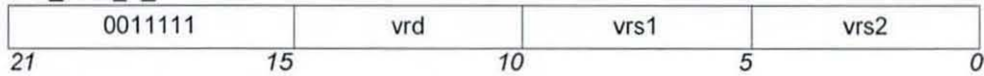
Example

```
vitu_mult_e_r(3,1,2); //vreg3=vreg1*vreg2 (even elements)
```

vitu_mult_o_r

Instruction Format

vitu_mult_o_r format



0011111	vrd	vrs1	vrs2

21 15 10 5 0

Syntax

`vitu_mult_o_r(vrd, vrs1, vrs2)`

where:

vrd	is the destination vector register
vrs1	is the first source vector register (operand 1)
vrs2	is the second source vector register (operand 2)

Description

The `vitu_mult_o_r` instruction performs signed short multiplication (16-bit) to the odd elements of the source vector registers `vrs1` and `vrs2` and places the result to the even elements of the destination vector register `vrd`.

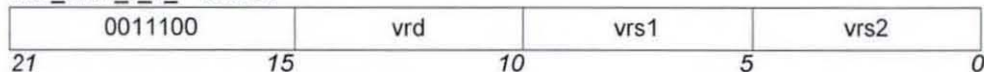
Example

```
vitu_mult_o_r(3,1,2); //vreg3=vreg1*vreg2 (odd elements)
```

vitu_mult_r_e_r

Instruction Format

vitu_mult_r_e_r format



0011100	vrd	vrs1	vrs2

21 15 10 5 0

Syntax

`vitu_mult_r_e_r(vrd, vrs1, vrs2)`

where:

vrd	is the destination vector register
vrs1	is the first source vector register (operand 1)
vrs2	is the second source vector register (operand 2)

Description

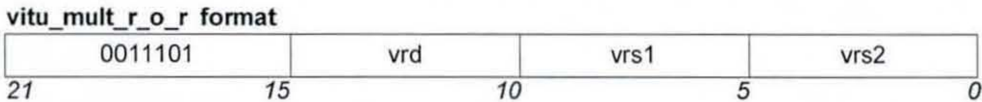
The `vitu_mult_r_e_r` instruction performs signed short multiplication (16-bit) with rounding to the even elements of the source vector registers `vrs1` and `vrs2` and places the result to the even elements of the destination vector register `vrd`.

Example

```
vitu_mult_r_e_r(3,1,2); //vreg3=vreg1*vreg2 (with
                          rounding - even elements)
```

vitu_mult_r_o_r

Instruction Format



Syntax

`vitu_mult_r_o_r(`*`vrd, vrs1, vrs2`*`)`

where:

vrd	is the destination vector register
vrs1	is the first source vector register (operand 1)
vrs2	is the second source vector register (operand 2)

Description

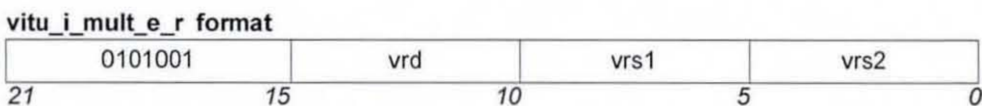
The `vitu_mult_r_o_r` instruction performs signed short multiplication (16-bit) with rounding to the odd elements of the source vector registers `vrs1` and `vrs2` and places the result to the odd elements of the destination vector register `vrd`.

Example

```
vitu_mult_r_o_r(3,1,2); //vreg3=vreg1*vreg2 (with
                          rounding - odd elements)
```

vitu_i_mult_e_r

Instruction Format



Syntax

`vitu_i_mult_e_r(`*`vrd, vrs1, vrs2`*`)`

where:

vrd	is the destination vector register
vrs1	is the first source vector register (operand 1)
vrs2	is the second source vector register (operand 2)

Description

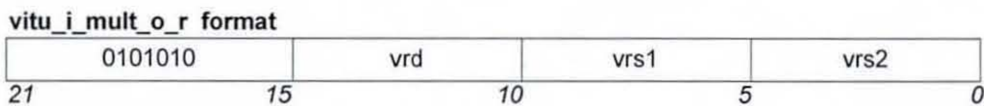
The `vitu_i_mult_e_r` instruction performs integer short multiplication (16-bit) to the even elements of the source vector registers `vrs1` and `vrs2` and places the result to the even elements of the destination vector register `vrd`.

Example

```
vitu_i_mult_e_r(3,1,2); //vreg3=vreg1*vreg2 (integer-
                          even elements)
```

vitu_i_mult_o_r

Instruction Format



vitu_i_mult_o_r format			
0101010	vrd	vrs1	vrs2
21 15	10	5	0

Syntax

`vitu_i_mult_o_r(`*`vrd, vrs1, vrs2`*`)`

where:

vrd	is the destination vector register
vrs1	is the first source vector register (operand 1)
vrs2	is the second source vector register (operand 2)

Description

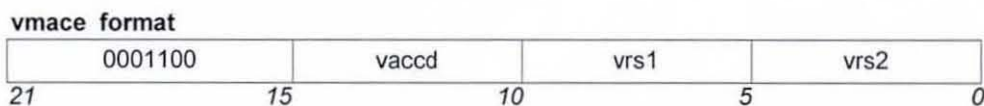
The `vitu_i_mult_o_r` instruction performs integer short multiplication (16-bit) to the odd elements of the source vector registers `vrs1` and `vrs2` and places the result to the odd elements of the destination vector register `vrd`.

Example

```
vitu_i_mult_o_r(3,1,2); //vreg3=vreg1*vreg2 (integer-
                          odd elements)
```

vmace

Instruction Format



vmace format			
0001100	vaccd	vrs1	vrs2
21 15	10	5	0

Syntax

`vmace(`*`vaccd, vrs1, vrs2`*`)`

where:

vaccd	is the destination vector accumulator
vrs1	is the first source vector register (operand 1)
vrs2	is the second source vector register (operand 2)

Description

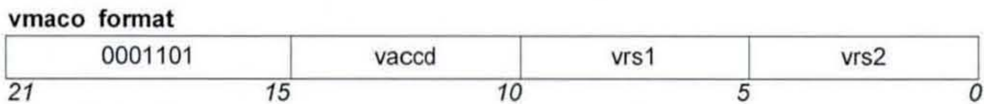
The `vmace` instruction performs long multiplication (32-bit) to the even elements of the source vector registers `vrs1` and `vrs2` and adds the product to the even elements of the destination vector accumulator `vaccd`.

Example

```
vmace(0,1,2); //Perform mac to even elements of vacc0,
                vreg1 and vreg2
```

vmaco

Instruction Format



Syntax

`vmaco(vaccd, vrs1, vrs2)`

where:

vaccd	is the destination vector accumulator
vrs1	is the first source vector register (operand 1)
vrs2	is the second source vector register (operand 2)

Description

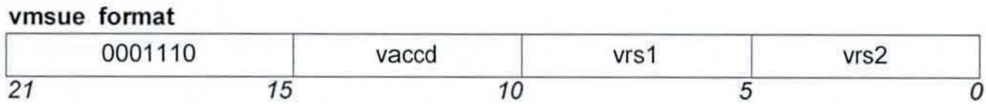
The `vmaco` instruction performs long multiplication (32-bit) to the odd elements of the source vector registers `vrs1` and `vrs2` and adds the product to the odd elements of the destination vector accumulator `vaccd`.

Example

```
vmaco(0,1,2); //Perform mac to odd elements of vacc0,
                vreg1 and vreg2
```

vmsue

Instruction Format



Syntax

vmsue(*vaccd, vrs1, vrs2*)

where:

vaccd	is the destination vector accumulator
vrs1	is the first source vector register (operand 1)
vrs2	is the second source vector register (operand 2)

Description

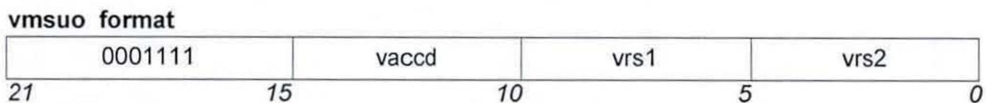
The vmsue instruction performs long multiplication (32-bit) to the even elements of the source vector registers vrs1 and vrs2 and subtracts the product to the even elements of the destination vector accumulator vaccd.

Example

```
vmsue(0,1,2); //Perform multiply-subtract to even elements
               of vacc0, vreg1 and vreg2
```

vmsuo

Instruction Format



Syntax

vmsuo(*vaccd, vrs1, vrs2*)

where:

vaccd	is the destination vector accumulator
vrs1	is the first source vector register (operand 1)
vrs2	is the second source vector register (operand 2)

Description

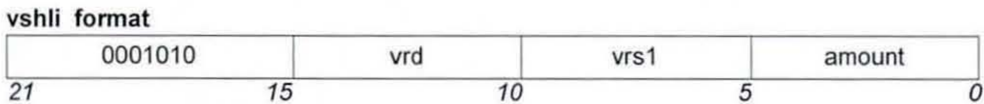
The vmsuo instruction performs long multiplication (32-bit) to the odd elements of the source vector registers vrs1 and vrs2 and subtracts the product to the odd elements of the destination vector accumulator vaccd.

Example

```
vmsuo(0,1,2); //Perform multiply-subtract to odd elements
                of vacc0, vreg1 and vreg2
```

vshli

Instruction Format



Syntax

`vshli(vrd, vrs1, amount)`

where:

vrd	is the destination vector register
vrs1	is the vector register (operand 1) to be shifted
amount	is the shift amount (immediate)

Description

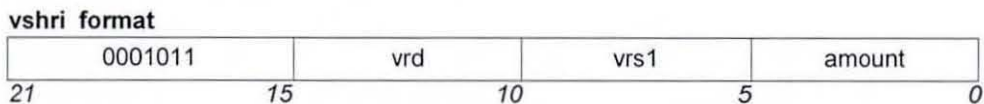
The `vshli` instruction performs short shift left (16-bit) to the vector register `vrs1` by immediate (`amount`).

Example

```
vshli(3,1,4); //Shift left vreg1 by 4 and put result to
                vreg3
```

vshri

Instruction Format



Syntax

`vshri(vrd, vrs1, amount)`

where:

vrd	is the destination vector register
vrs1	is the vector register (operand 1) to be shifted
amount	is the shift amount (immediate)

Description

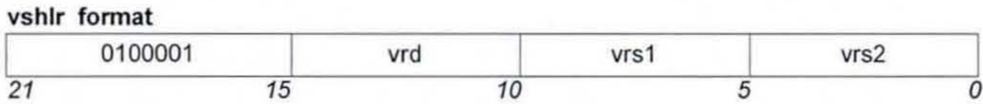
The `vshri` instruction performs short shift right (16-bit) to the vector register `vrs1` by immediate (`amount`).

Example

```
vshri(3,1,4); //Shift right vreg1 by 4 and put result to
                vreg3
```

vshlr

Instruction Format



Syntax

`vshlr(vrd, vrs1, vrs2)`

where:

vrd	is the destination vector register
vrs1	is the vector register (operand 1) to be shifted
vrs2	is the vector register (operand 2) that contains the shift amount

Description

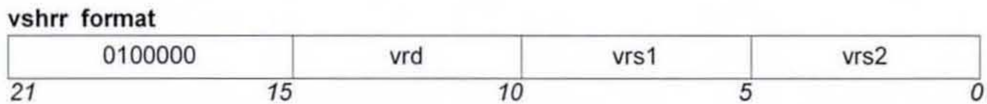
The `vshlr` instruction performs short shift left (16-bit) to the vector register `vrs1` by the amount of the vector register `vrs2`.

Example

```
vshlr(5,1,3); //Shift left vreg1 by amount that is in
                vreg3 and put result to vreg5
```

vshrr

Instruction Format



Syntax

`vshrr(vrd, vrs1, vrs2)`

where:

vrd	is the destination vector register

`vrs1` is the vector register (operand 1) to be shifted
`vrs2` is the vector register (operand 2) that contains the shift amount

Description

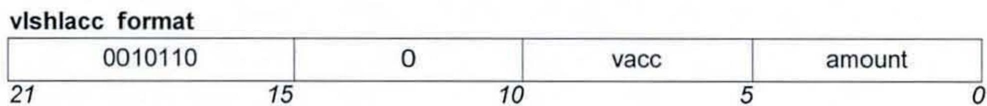
The `vshrr` instruction performs short shift right (16-bit) to the vector register `vrs1` by the amount of the vector register `vrs2` and places the result to the destination scalar register `vrd`.

Example

```
vshrr(5,1,3); //Shift right vreg1 by amount that is in
                vreg3 and put result to vreg5
```

vlshlacc

Instruction Format



Syntax

```
vlshlacc (vacc, amount)
```

where:
vacc is the vector accumulator
`amount` is the shift amount

Description

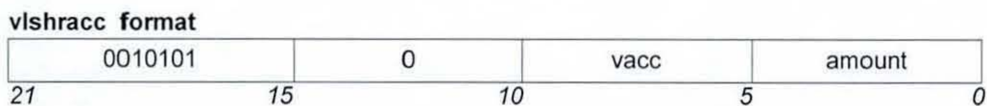
The `vlshlacc` instruction performs long (32-bit) shift left to the vector accumulator `vacc` by `amount` (immediate).

Example

```
vlshlacc(1,3); //Long shift left vacc1 by 3 and put result
                 to vacc1
```

vlshracc

Instruction Format



Syntax

```
vlshracc (vacc, amount)
```

where:

vacc is the vector accumulator
amount is the shift amount

Description

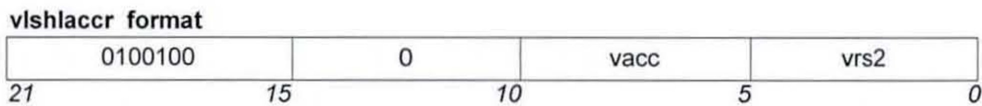
The vlshracc instruction performs long (32-bit) shift right to the vector accumulator vacc by amount (immediate).

Example

```
vlshracc(1,3); //Long shift right vacc1 by 3 and put
                 result to vacc1
```

vlshlaccr

Instruction Format



Syntax

```
vlshlaccr (vacc, vrs2)
```

where:

vacc is the vector accumulator
vrs2 is the vector register with the shift amount

Description

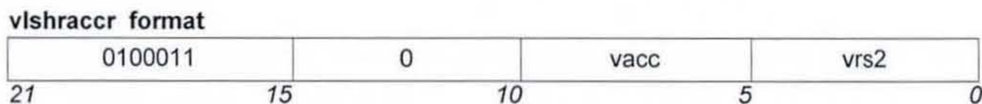
The vlshlaccr instruction performs long (32-bit) shift left to the vector accumulator vacc by the amount of the vector register vrs2.

Example

```
vlshlaccr(1,2); //Long shift left vacc1 by amount that is
                  in vreg2 and put result to vacc1
```

vlshraccr

Instruction Format



Syntax

```
vlshraccr (vacc, vrs2)
```

where:

vacc	is the vector accumulator
vrs2	is the vector register with the shift amount

Description

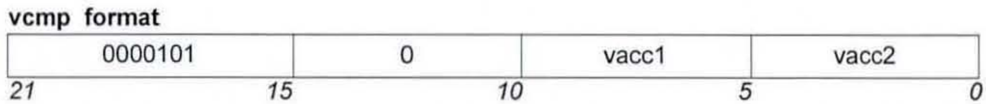
The `vlshraccr` instruction performs long (32-bit) shift right to the vector accumulator `vacc` by the amount of the vector register `vrs2`.

Example

```
vlshraccr(1,2); //Long shift right vacc1 by amount that is
                  in vreg2 and put result to vacc1
```

vcmp

Instruction Format



Syntax

```
vcmp (vacc1, vacc2)
```

where:

vacc1	is the first source vector accumulator (operand 1)
vacc2	is the second source vector accumulator (operand 2)

Description

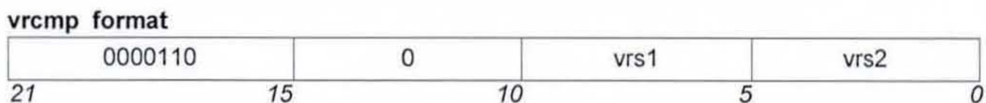
The `vcmp` instruction compares two vector accumulators `(vacc1, vacc2)`. If `vacc1` is greater than `vacc2` then the predication flag becomes 1 (true) else 0 (false).

Example

```
vcmp(0,1); //Compares vacc0 with vacc1
```

vrcmp

Instruction Format



Syntax

```
vrcmp (vrs1, vrs2)
```

where:

`vrs1` is the first source vector register (operand 1)
`vrs2` is the second source vector register (operand 2)

Description

The `vrcmp` instruction compares two vector registers (`vrs1, vrs2)`. If `vrs1` is greater than `vrs2` then the predication flag becomes 1 (true) else 0 (false).

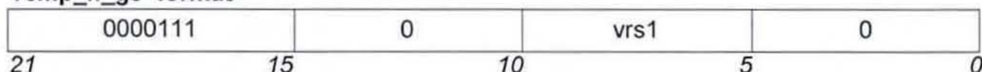
Example

```
vrcmp(1,2); //Compares vreg1 with vreg2
```

vcmp_h_ge

Instruction Format

vcmp_h_ge format



Syntax

```
vcmp_h_ge (vrs1)
```

where:

`vrs1` is the vector register to be compared

Description

The `vcmp_h_ge` instruction checks if vector register `vrs1` is greater than or equal to zero and if it is true sets the predication flag to 1 (true) else 0 (false).

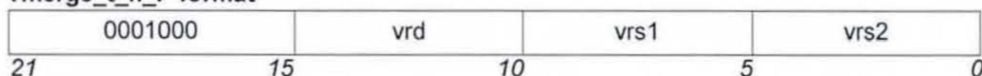
Example

```
vcmp_h_ge(2); //Compares vreg2 with zero
```

vmerge_t_h_r

Instruction Format

vmerge_t_h_r format



Syntax

```
vmerge_t_h_r (vrd, vrs1, vrs2)
```

where:

vrd	is the destination vector register
vrs1	is the first source vector register (operand 1)
vrs2	is the second source vector register (operand 2)

Description

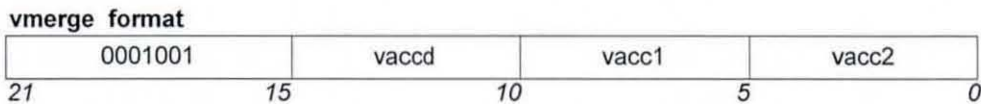
The `vmerge_t_h_r` instruction merges two vector registers (`vrs1, vrs2`) according to the predication flag value. If `pred` is 1 then `vrd=vrs1` else `vrd=vrs2`.

Example

```
vmerge_t_h_r(3,4,2); //if pred=1 vreg3=vreg4 else vrd=vreg2
```

vmerge

Instruction Format


vmerge format

0001001	vaccd	vacc1	vacc2

21 15 10 5 0

Syntax

```
vmerge (vaccd, vacc1, vacc2)
```

where:

vaccd	is the destination vector accumulator
vacc1	is the first source vector accumulator (operand 1)
vacc2	is the second source vector accumulator (operand 2)

Description

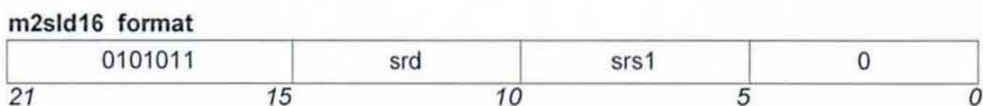
The `vmerge` instruction merges two vector accumulators (`vacc1, vacc2`) according to the predication flag value. If `pred` is 1 then `vaccd=vacc1` else `vaccd=vacc2`.

Example

```
vmerge(0,0,1); //if pred=1 vacc0=vacc0 else vacc0=vacc1
```

m2sld16

Instruction Format


m2sld16 format

0101011	srd	srs1	0

21 15 10 5 0

Syntax

```
m2sld16(srd, srs1)
```

where:

srd	is the destination scalar register
srs1	is the address of the variable in memory

Description

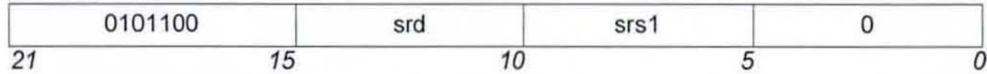
The `m2sld16` instruction loads a 16-bit value to scalar register `srd` from memory address given in scalar register `srs1`.

Example

```
m2sld16(2, 3); //Load (16-bit) to sreg2 from address that
                 is in sreg3
```

m2sld32

Instruction Format



Syntax

```
m2sld32(srd, srs1)
```

where:

srd	is the destination scalar register
srs1	is the address of the variable in memory

Description

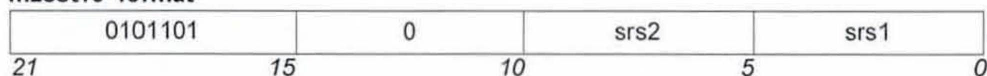
The `m2sld32` instruction loads a 32-bit value to scalar register `srd` from memory address given in scalar register `srs1`.

Example

```
m2sld16(4, 3); //Load (32-bit) to sreg4 from address that
                 is in sreg3
```

m2sst16

Instruction Format



Syntax

`m2sst16(srs2, srs1)`

where:

srs2 is the source scalar register
srs1 is the memory address

Description

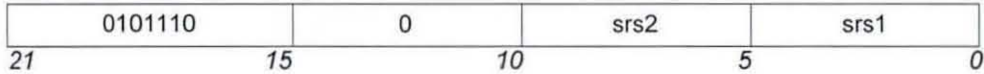
The `m2sst16` instruction stores a 16-bit value of scalar register `srs2` to memory address given from scalar register `srs1`.

Example

```
m2sst16(4, 3); //Store (16-bit) sreg4 to memory address
                 that is in sreg3
```

m2sst32

Instruction Format



m2sst32 format

0101110	0	srs2	srs1
21	15	10	5 0

Syntax

`m2sst32(srs2, srs1)`

where:

srs2 is the source scalar register
srs1 is the memory address

Description

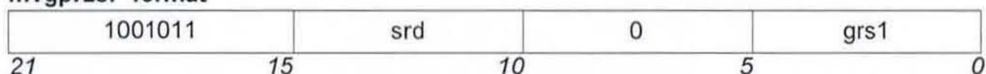
The `m2sst32` instruction stores a 32-bit value of scalar register `srs2` to memory address given from scalar register `srs1`.

Example

```
m2sst32(2, 1); //Store (32-bit) sreg2 to memory address
                 that is in sreg1
```

mvgpr2sr

Instruction Format



mvgpr2sr format

1001011	srd	0	grs1
21	15	10	5 0

Syntax

`mvgpr2sr(srd, grs1)`

where:

srd	is the destination scalar register
grs1	is the source general purpose register (Leon)

Description

The `mvgpr2sr` instruction moves the scalar contents (32-bit) of the general purpose register `grs1` to the scalar register `srd`.

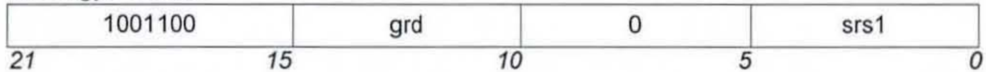
Example

```
mvgpr2sr(1,2); //Move the contents of the greg2 to sreg1
```

mvsr2gpr

Instruction Format

mvsr2gpr format



Syntax

`mvsr2gpr(grd, srs1)`

where:

grd	is the destination general purpose register (Leon)
srs1	is the source vector register

Description

The `mvsr2gpr` instruction moves the contents of the scalar register `srs1` to the general purpose register (Leon) `grd`.

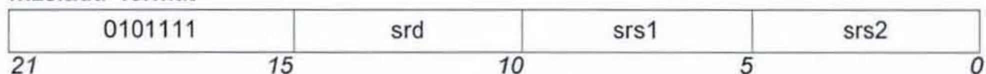
Example

```
mvsr2gpr(2,3); //Move the contents of sreg3 to greg2
```

m2sladd

Instruction Format

m2sladd format



Syntax

`m2sladd(`*`srd, srs1, srs2`*`)`

where:

srd	is the destination scalar register
srs1	is the first source scalar register (operand 1)
srs2	is the second source scalar register (operand 2)

Description

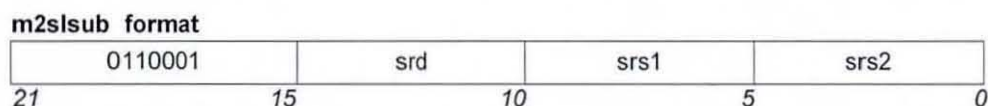
The `m2sladd` instruction performs long addition (32-bit) of source scalar registers `srs1` and `srs2` and places the result to the destination scalar register `srd`.

Example

```
m2sladd(4,2,3); //sreg4=sreg2+sreg3 (32-bit)
```

m2slsub

Instruction Format



Syntax

`m2slsub(`*`srd, srs1, srs2`*`)`

where:

srd	is the destination scalar register
srs1	is the first source scalar register (operand 1)
srs2	is the second source scalar register (operand 2)

Description

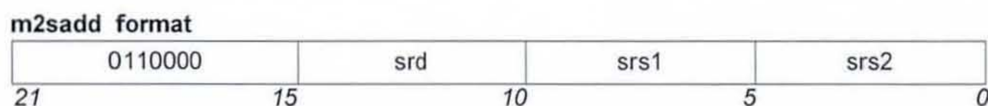
The `m2slsub` instruction performs long subtraction (32-bit) of source scalar registers `srs1` and `srs2` and places the result to the destination scalar register `srd`.

Example

```
m2slsub(4,2,3); //sreg4=sreg2-sreg3 (32-bit)
```

m2sadd

Instruction Format



Syntax

`m2sadd(srd, srs1, srs2)`

where:

srd	is the destination scalar register
srs1	is the first source scalar register (operand 1)
srs2	is the second source scalar register (operand 2)

Description

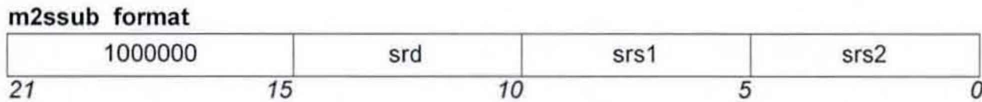
The `m2sadd` instruction performs short addition (16-bit) of source scalar registers `srs1` and `srs2` and places the result to the destination scalar register `srd`.

Example

```
m2sadd(4,2,3); //sreg4=sreg2+sreg3 (16-bit)
```

m2ssub

Instruction Format



Syntax

`m2ssub(srd, srs1, srs2)`

where:

srd	is the destination scalar register
srs1	is the first source scalar register (operand 1)
srs2	is the second source scalar register (operand 2)

Description

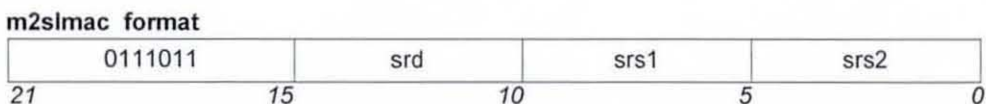
The `m2ssub` instruction performs short subtraction (16-bit) of source scalar registers `srs1` and `srs2` and places the result to the destination scalar register `srd`.

Example

```
m2ssub(4,2,3); //sreg4=sreg2-sreg3 (16-bit)
```

m2slmac

Instruction Format



Syntax

```
m2slmac(srd, srs1, srs2)
```

where:

srd	is the destination scalar register
srs1	is the first source scalar register (operand 1)
srs2	is the second source scalar register (operand 2)

Description

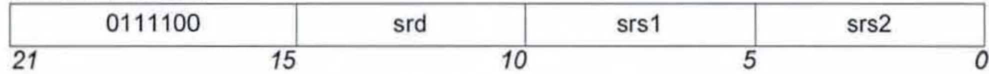
The `m2slmac` instruction performs long multiplication (32-bit) to source scalar registers `srs1` and `srs2` and adds the product to the destination scalar register `srd`.

Example

```
m2slmac(4,2,3); //sreg4=sreg4+(sreg2*sreg3)
```

m2slmsu

Instruction Format


m2slmsu format

0111100	srd	srs1	srs2
21 15	10	5	0

Syntax

```
m2slmsu(srd, srs1, srs2)
```

where:

srd	is the destination scalar register
srs1	is the first source scalar register (operand 1)
srs2	is the second source scalar register (operand 2)

Description

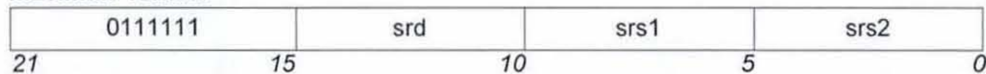
The `m2slmsu` instruction performs long multiplication (32-bit) to source scalar registers `srs1` and `srs2` and subtracts the product to the destination scalar register `srd`.

Example

```
m2slmsu(4,2,3); // sreg4=sreg4-(sreg2*sreg3)
```

m2slmult

Instruction Format


m2slmult format

0111111	srd	srs1	srs2
21 15	10	5	0

Syntax

`m2slmult(srd, srs1, srs2)`

where:

`srd`	is the destination scalar register
`srs1`	is the first source scalar register (operand 1)
`srs2`	is the second source scalar register (operand 2)

Description

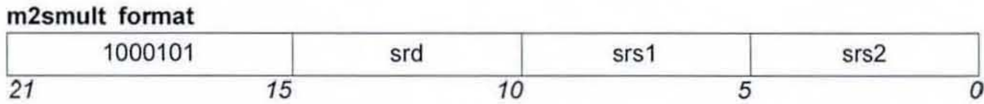
The `m2slmult` instruction performs long multiplication (32-bit) to source scalar registers `srs1` and `srs2` and places the result to the destination vector register `srd`.

Example

```
m2slmult(4,2,3); //sreg4=sreg2*sreg3 (32-bit)
```

m2smult

Instruction Format



Syntax

`m2smult(srd, srs1, srs2)`

where:

`srd`	is the destination scalar register
`srs1`	is the first source scalar register (operand 1)
`srs2`	is the second source scalar register (operand 2)

Description

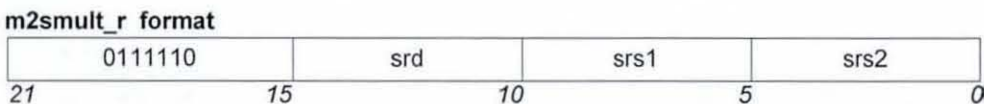
The `m2smult` instruction performs short multiplication (16-bit) to source scalar registers `srs1` and `srs2` and places the result to the destination vector register `srd`.

Example

```
m2smult(4,2,3); //sreg4=sreg2*sreg3 (16-bit)
```

m2smult_r

Instruction Format



Syntax

```
m2smult_r(srd, srs1, srs2)
```

where:

srd	is the destination scalar register
srs1	is the first source scalar register (operand 1)
srs2	is the second source scalar register (operand 2)

Description

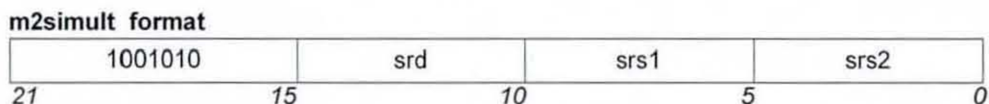
The `m2smult_r` instruction performs short multiplication (16-bit) with rounding to source scalar registers `srs1` and `srs2` and places the result to the destination vector register `srd`.

Example

```
m2smult_r(4,2,3); //sreg4=sreg2*sreg3 (with rounding)
```

m2simult

Instruction Format



Syntax

```
m2simult(srd, srs1, srs2)
```

where:

srd	is the destination scalar register
srs1	is the first source scalar register (operand 1)
srs2	is the second source scalar register (operand 2)

Description

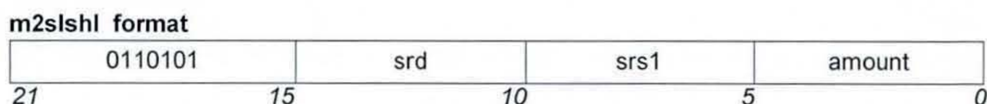
The `m2simult` instruction performs short integer multiplication (16-bit) to source scalar registers `srs1` and `srs2` and places the result to the destination vector register `srd`.

Example

```
m2simult(4,2,3); //sreg4=sreg2*sreg3 (integer)
```

m2slshl

Instruction Format



Syntax

`m2slshl(srd, srs1, amount)`

where:

srd	is the destination scalar register
srs1	is the scalar register (operand 1) to be shifted
amount	is the shift amount (immediate)

Description

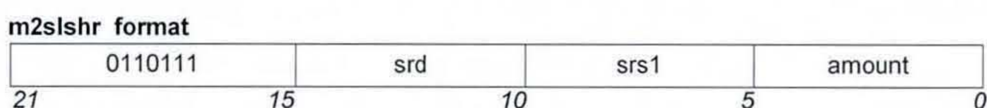
The `m2slshl` instruction performs long shift left (32-bit) to the scalar register `srs1` by immediate (`amount`) and places the result to the destination scalar register `srd`.

Example

```
m2slshl(3,1,4); //Long shift left sreg1 by 4 and put
                  result to sreg3
```

m2slshr

Instruction Format



Syntax

`m2slshr(srd, srs1, amount)`

where:

srd	is the destination scalar register
srs1	is the scalar register (operand 1) to be shifted
amount	is the shift amount (immediate)

Description

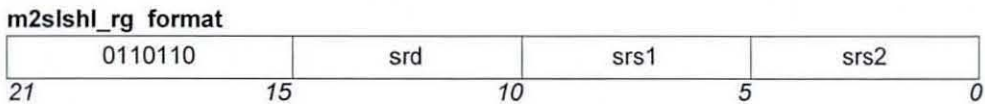
The `m2slshr` instruction performs long shift right (32-bit) to the scalar register `srs1` by immediate (`amount`) and places the result to the destination scalar register `srd`.

Example

```
m2slshr(3,1,4); //Long shift right sreg1 by 4 and put
                  result to sreg3
```

m2slshl_rg

Instruction Format



Syntax

m2slshl_rg(*srd, srs1, srs2*)

where:

srd	is the destination scalar register
srs1	is the scalar register (operand 1) to be shifted
srs2	is the scalar register (operand 2) with the shift amount

Description

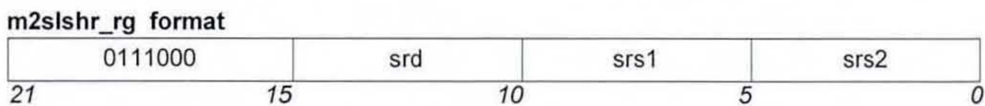
The m2slshl_rg instruction performs long shift left (32-bit) to the scalar register srs1 by the amount of the vector register srs2 and places the result to the destination scalar register srd.

Example

```
m2slshl_rg(3,1,4); //Long shift left sreg1 by amount that
                     is in sreg4 and put result to sreg3
```

m2slshr_rg

Instruction Format



Syntax

m2slshr_rg(*srd, srs1, srs2*)

where:

srd	is the destination scalar register
srs1	is the scalar register (operand 1) to be shifted
srs2	is the scalar register (operand 2) with the shift amount

Description

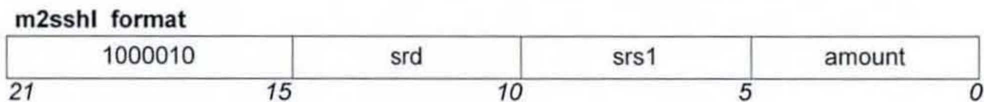
The `m2slshr_rg` instruction performs long shift right (32-bit) to the scalar register `srs1` by the amount of the vector register `srs2` and places the result to the destination scalar register `srd`.

Example

```
m2slshr_rg(3,1,4); //Long shift right sreg1 by amount that
                     is in sreg4 and put result to sreg3
```

m2sshl

Instruction Format



m2sshl format			
1000010	srd	srs1	amount
21 – 15	15 – 10	10 – 5	5 – 0

Syntax

`m2sshl(srd, srs1, amount)`

where:

srd	is the destination scalar register
srs1	is the scalar register (operand 1) to be shifted
amount	is the shift amount (immediate)

Description

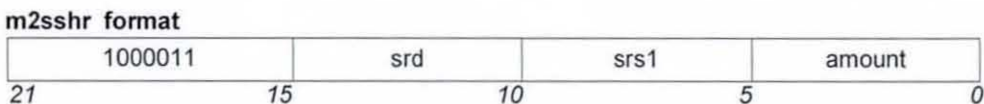
The `m2sshl` instruction performs short shift left (16-bit) to the scalar register `srs1` by immediate (`amount`) and places the result to the destination scalar register `srd`.

Example

```
m2sshl(3,1,4); //Short shift left sreg1 by 4 and put
                 result to sreg3
```

m2sshr

Instruction Format



m2sshr format			
1000011	srd	srs1	amount
21 – 15	15 – 10	10 – 5	5 – 0

Syntax

`m2sshr(srd, srs1, amount)`

where:

srd	is the destination scalar register
srs1	is the scalar register (operand 1) to be shifted
amount	is the shift amount (immediate)

Description

The `m2sshr` instruction performs short shift right (16-bit) to the scalar register `srs1` by immediate (`amount`) and places the result to the destination scalar register `srd`.

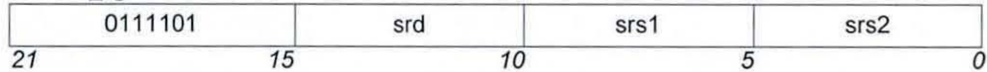
Example

```
m2sshr(3,1,4); //Short shift right sreg1 by 4 and put
                 result to sreg3
```

m2sshl_rg

Instruction Format

m2sshl_rg format



Syntax

m2sshl_rg(*srd, srs1, srs2*)

where:

srd	is the destination scalar register
srs1	is the scalar register (operand 1) to be shifted
srs2	is the scalar register (operand 2) with the shift amount

Description

The `m2sshl_rg` instruction performs short shift left (16-bit) to the scalar register `srs1` by the amount of the vector register `srs2` and places the result to the destination scalar register `srd`.

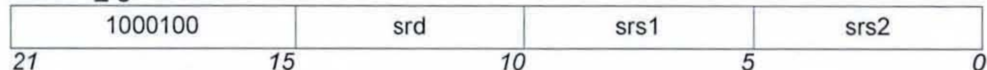
Example

```
m2sshl_rg(3,1,4); //Short shift left sreg1 by amount that
                    is in sreg4 and put result to sreg3
```

m2sshr_rg

Instruction Format

m2sshr_rg format



Syntax

```
m2sshr_rg(srd, srs1, srs2)
```

where:

srd	is the destination scalar register
srs1	is the scalar register (operand 1) to be shifted
srs2	is the scalar register (operand 2) with the shift amount

Description

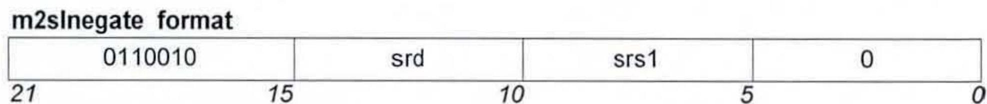
The m2sshr_rg instruction performs short shift right (16-bit) to the scalar register srs1 by the amount of the vector register srs2 and places the result to the destination scalar register srd.

Example

```
m2sshr_rg(3,1,4); //Short shift right sreg1 by amount that
                    is in sreg4 and put result to sreg3
```

m2slnegate

Instruction Format



m2slnegate format

0110010	srd	srs1	0
21	15	10	5 0

Syntax

```
m2slnegate(srd, srs1)
```

where:

srd	is the destination scalar register
srs1	is the source scalar register (operand 1)

Description

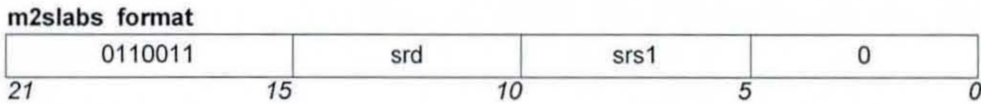
The m2slnegate instruction negates the 32-bit value in scalar register srs1 with saturation and stores the result to the destination scalar register srd.

Example

```
m2slnegate(3,1); //Negates (32-bit) value sreg1 and put
                   result to sreg3
```

m2slabs

Instruction Format



Syntax

```
m2slabs_s(srd, srs1)
```

where:

srd	is the destination scalar register
srs1	is the source scalar register (operand 1)

Description

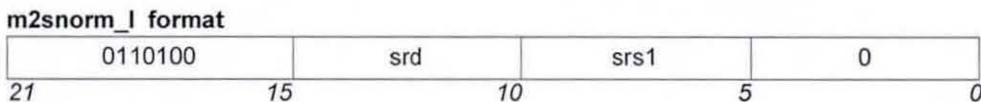
The `m2slabs` instruction produces the absolute value of the 32-bit value in scalar register `srs1` and places the result to the destination scalar register `srd`.

Example

```
m2slabs(3,1); //Absolute (32-bit) value of sreg1 and put
                result to sreg3
```

m2snorm_l

Instruction Format



Syntax

```
m2snorm_l(srd, srs1)
```

where:

srd	is the destination scalar register
srs1	is the source scalar register (operand 1)

Description

The `m2snorm_l` instruction produces the number of left shifts needed to normalise the 32-bit value in scalar register `srs1` and places the result to the destination scalar register `srd`.

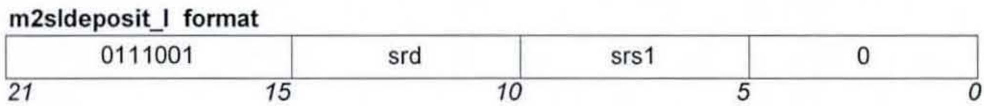
Example

```
m2snorm_l(3,1); //Normalise value (32-bit)of sreg1 and put
```

```
                    result to sreg3
```

m2sldeposit_l

Instruction Format



Syntax

m2sldeposit_l(*srd, srs1*)

where:

srd	is the destination scalar register
srs1	is the source scalar register (operand 1)

Description

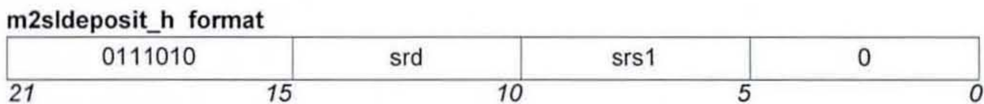
The m2sldeposit_l instruction deposits the 16 LSB of scalar register srs1 into the LSB 32-bit of destination scalar register srd. The 16 MSB of srd are sign extended.

Example

```
m2sldeposit_l(3,1); //Deposit 16 LSB of sreg1 into 16 LSB
                      of sreg3
```

m2sldeposit_h

Instruction Format



Syntax

m2sldeposit_h(*srd, srs1*)

where:

srd	is the destination scalar register
srs1	is the source scalar register (operand 1)

Description

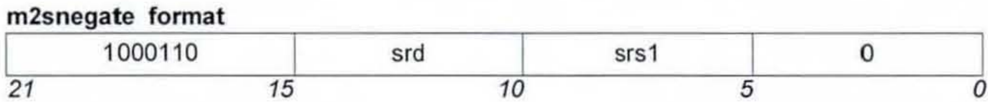
The m2sldeposit_h instruction deposits the 16 LSB of scalar register srs1 into the MSB 32-bit of destination scalar register srd. The 16 LSB of srd are zero extended.

Example

```
m2sldeposit_h(3,1); //Deposit 16 LSB of sreg1 into 16 MSB
                      of sreg3
```

m2snegate

Instruction Format



Syntax

`m2snegate(srd, srs1)`

where:

srd	is the destination scalar register
srs1	is the source scalar register (operand 1)

Description

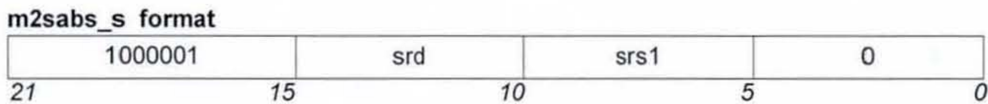
The `m2snegate` instruction negates the 16-bit value in scalar register `srs1` and places the result to the destination scalar register `srd`.

Example

```
m2snegate(4,2); //Negate value (16-bit)of sreg2 and put
                  result to sreg4
```

m2sabs_s

Instruction Format



Syntax

`m2sabs_s(srd, srs1)`

where:

srd	is the destination scalar register
srs1	is the source scalar register (operand 1)

Description

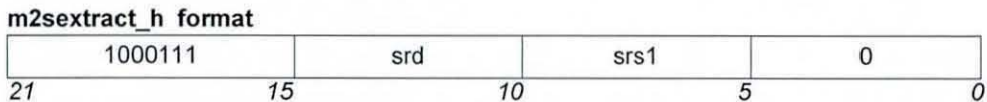
The `m2sabs_s` instruction produces the absolute value of the 16-bit value in scalar register `srs1` and places the result to the destination scalar register `srd`.

Example

```
m2sabs_s(4,2); //Absolute value (16-bit)of sreg2 and put
                 result to sreg4
```

m2sextract_h

Instruction Format



Syntax

`m2sextract_h(`*`srd, srs1`*`)`

where:

srd	is the destination scalar register
srs1	is the source scalar register (operand 1)

Description

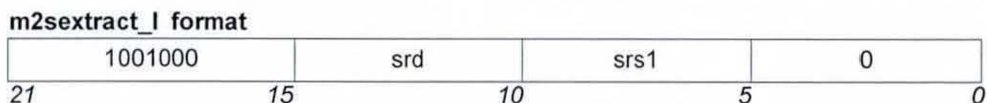
The `m2sextract_h` instruction extracts the 16 MSB of the 32-bit value of scalar register `srs1` and places them into the 16 LSB of the destination scalar register `srd`. The 16 MSB of `srd` are zero extended.

Example

```
m2sextract_h(4,2); //Extract 16 MSB of sreg2 and put
                     them to sreg4
```

m2sextract_l

Instruction Format



Syntax

`m2sextract_l(`*`srd, srs1`*`)`

where:

srd	is the destination scalar register
srs1	is the source scalar register (operand 1)

Description

The m2sextract_l instruction extracts the 16 MSB of the 32-bit value of scalar register srs1 and places them into the 16 LSB of the destination scalar register srd. The 16 MSB of srd are zero extended.

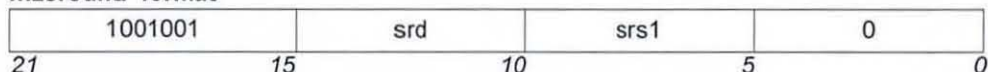
Example

```
m2sextract_l(4,2); //Extract 16 LSB of sreg2 and put
                        them to sreg4
```

m2sround

Instruction Format

m2sround format



1001001	srd	srs1	0
21 15	10	5	0

Syntax

m2sround(*srd, srs1*)

where:

srd	is the destination scalar register
srs1	is the source scalar register (operand 1)

Description

The m2sround instruction rounds the 16 LSB of the 32-bit value of scalar register srs1 into its most significant 16-bits with saturation. The result is shifted right by 16 and placed in the destination scalar register srd.

Example

```
m2sround(4,2); //Round 32-bit value of sreg2 and put
                  the result to sreg4
```

APPENDIX B SIGNAL DESCRIPTION

Signals for Vector Datapath

Signal	Type	Width	Brief Description
SIMD_r	Control	1 bit	Selects two 16-bit (when = '0') operations or one 32-bit (when = '1')
sel_sub_r	Control	1 bit	Selects addition (when='0') or subtraction (when='1')
sel_sfctn_r	Control	2 bits	Selects function for vadd unit
sel_round_r	Control	1 bit	Selects round operation (when='1')
sel_cmp_r	Control	1 bit	Selects compare operation (when='1')
sel_mult_r	Control	2 bits	Selects multiplication type for vmult unit
sel_mult_r_r	Control	1 bit	Selects mult (when='0') or mult_r ('1')
cmd_shift_r	Control	cmd_shift_type	Selects shift operation
sel_misc_r	Control	4 bits	Selects miscellaneous operation
sel_vu_r	Control	sel_vu_type	Selects vector unit for operation
vrs1_rdaddr_r	Control	Log2(VREGS)	Source 1 vector register address
vrs1_rden_r	Control	VLMAX*2	Source 1 vector register read-enable
vrs2_rdaddr_r	Control	Log2(VREGS)	Source 2 vector register address
vrs2_rden_r	Control	VLMAX*2	Source 2 vector register read-enable
vrd_addr	Control	Log2(VREGS)	Destination vector register address
vrd_wen	Control	VLMAX*2	Destination vector register write-enable
srs1_rdaddr_r	Control	Log2(SREGS)	Source 1 scalar register address
srs1_rden_r	Control	4 bits	Source 1 scalar register read-enable
srs2_rdaddr_r	Control	Log2(SREGS)	Source 2 scalar register address
srs2_rden_r	Control	4 bits	Source 2 scalar register read-enable
srs3_rdaddr_r	Control	Log2(SREGS)	Source 3 scalar register address
srs3_rden_r	Control	4 bits	Source 3 scalar register read-enable
srd_waddr_r	Control	Log2(SREGS)	Destination scalar register address
srd_wen_r	Control	4 bits	Destination scalar register write-enable
vacc1_rdaddr_r	Control	Log2 (ACC_NUMBER)	Source 1 vector accumulator address
vacc1_rden_r	Control	ACC_WIDTH/32	Source 1 vector accumulator read-enable
vacc2_rdaddr_r	Control	Log2 (ACC_NUMBER)	Source 2 vector accumulator address
vacc2_rden_r	Control	ACC_WIDTH/32	Source 2 vector accumulator read-enable
vacc_waddr_r	Control	Log2 (ACC_NUMBER)	Destination vector accumulator address
vacc_wen_r	Control	ACC_WIDTH/32	Destination vector accumulator write-enable
vlen_wen_r	Control	1 bit	Write enable for the vlen register
ovf_wen	Control	1 bit	Write enable for the overflow register
pred_wen	Control	1 bit	Write enable for the predicate register
vlen_nvalue	Data	Log2(VLMAX)	New value for the vlen register
lst_neg	Control	1 bit	Selects load/store negative stride (when='1')
opc_valid	Control	1 bit	Valid signal for the register output
addr_valid	Control	1 bit	Signal to indicate the address is valid
read	Control	1 bit	Selects load ('1') or store ('0') instruction
sel_vs	Control	1 bit	Selects vector ('1') or scalar ('0') ruction

Signal	Type	Width	Brief Description
sel_width	Control	1 bit	Selects 16 ('1') or 32 ('0') bits data width
sel_evod	Control	even_odd_type	Selects even or odd or normal operation
sel_opr1	Control	opr_type	Selects the type of the first operand
sel_opr2	Control	opr_type	Selects the type of the second operand
stg2_vadd	Control	1 bit	Stage 2 vadd unit enable
sel_vaccred	Control	1 bit	Vaccreduce unit enable
gpdata	Data	32 bits	Data from Leon register file
sel_st	Control	1 bit	Selects store instruction (when='1')
sel_mask	Control	1 bit	Selects to mask the result (when='1')

Control signal for Vadd Unit

sel_sfctn	Instruction
00	add/sub/vrcmp
01	vcmp_h_ge
10	vcmp
11	round

Control signal for Vmult Unit

sel_mult	Instruction
00	L_mult
01	i_mult
10	mult
11	mult_r

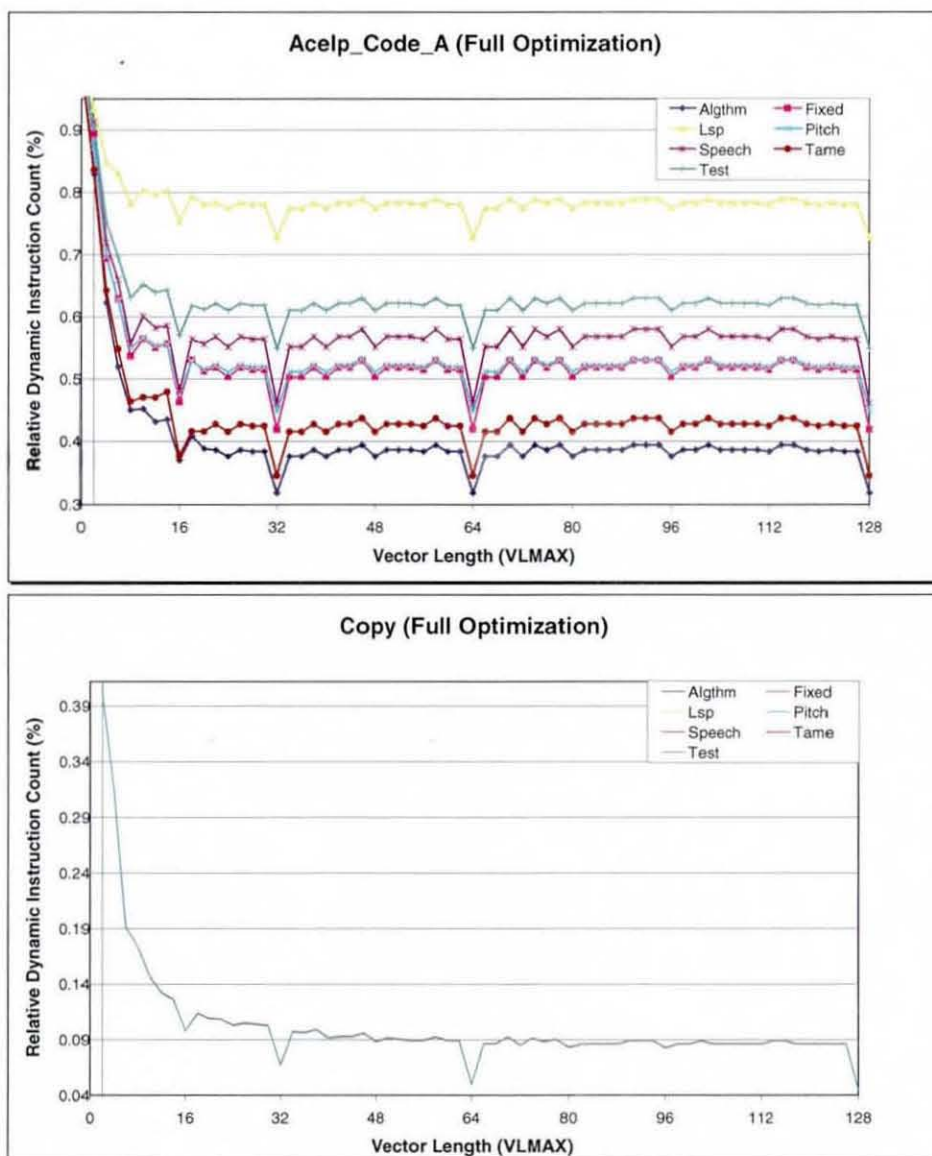
sel_mult_r	Instruction
0	mult
1	mult_r

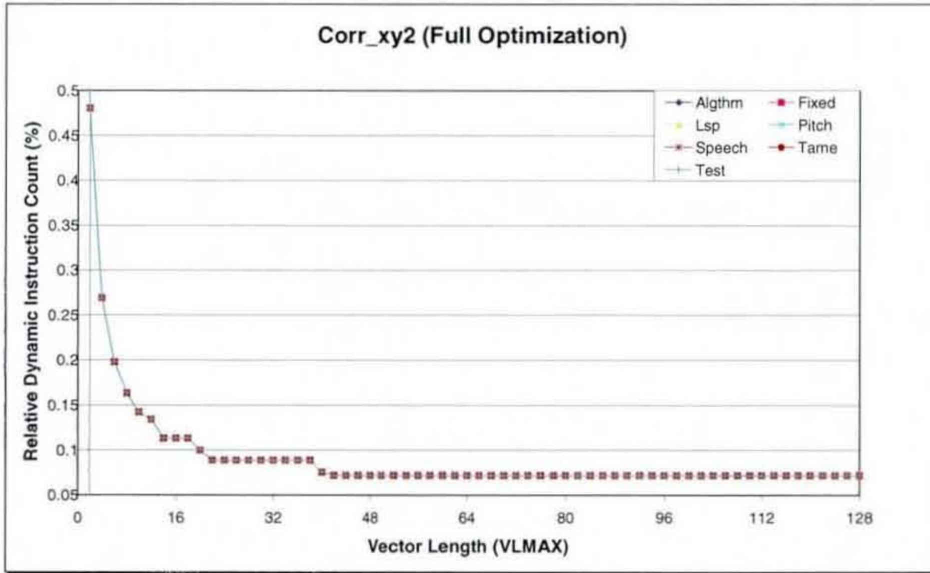
Control signal for Vmisc Unit

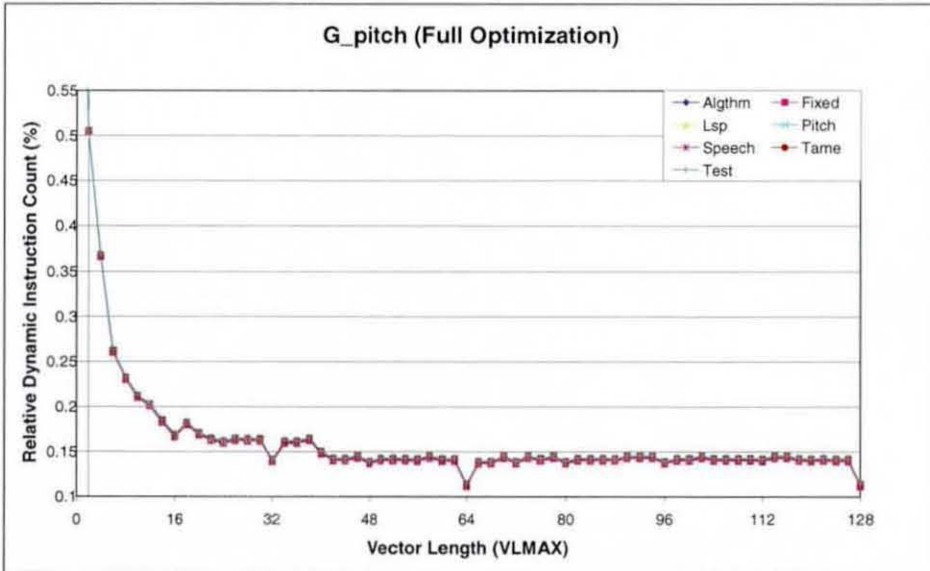
sel_misc	Instruction
0000	L_negate
0001	negate
0010	norm_l
0011	L_abs
0100	abs_s
0101	extract_l
0110	extract_h
0111	l_deposit_l
1000	l_deposit_h
1001	merge
1010	merge_t_h

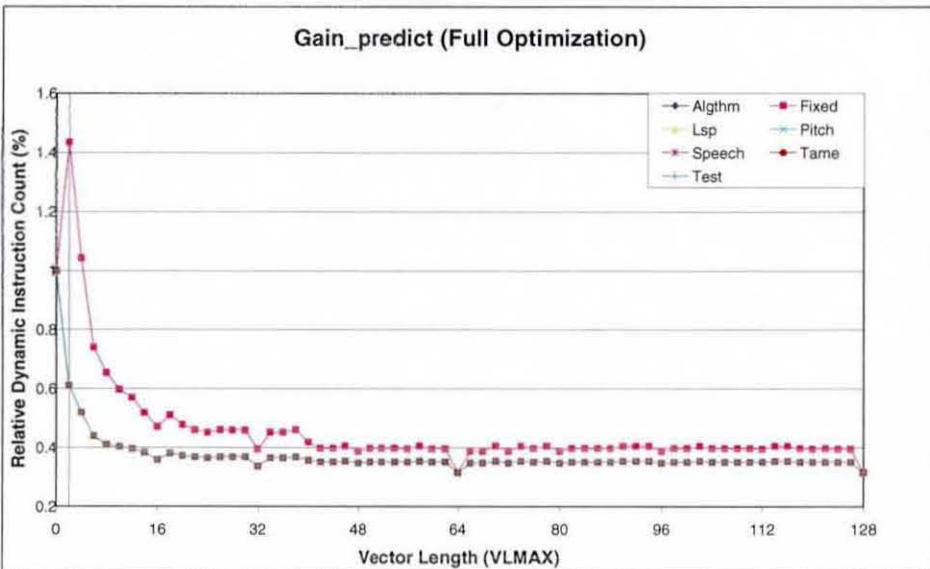
APPENDIX C G.729A AND G.723.1 FUNCTION RESULTS

This section presents the results from the G.729A speech codec showing the improvement made from a function perspective.




Corr_xy2 (Full Optimization)
Relative Dynamic Instruction Count (%)
Vector Length (VLMAX)
Algthm
Fixed
Lsp
Pitch
Speech
Tame
Test
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5
0
16
32
48
64
80
96
112
128


G_pitch (Full Optimization)
Relative Dynamic Instruction Count (%)
Vector Length (VLMAX)
Algthm
Fixed
Lsp
Pitch
Speech
Tame
Test
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5
0.55
0
16
32
48
64
80
96
112
128


Gain_predict (Full Optimization)
Relative Dynamic Instruction Count (%)
Vector Length (VLMAX)
Algthm
Fixed
Lsp
Pitch
Speech
Tame
Test
0.2
0.4
0.6
0.8
1.2
1.4
1.6
0
16
32
48
64
80
96
112
128

Get_wegt (Full Optimization)

Relative Dynamic Instruction Count (%)

0.8 0.75 0.7 0.65 0.6 0.55 0.5 0.45 0.4 0.35 0.3

Algthm, Fixed, Lsp, Pitch, Speech, Tame, Test

0 16 32 48 64 80 96 112 128

Vector Length (VLMAX)

Int_qlpc (Full Optimization)

Relative Dynamic Instruction Count (%)

0.5 0.45 0.4 0.35 0.3 0.25 0.2 0.15 0.1

Algthm, Fixed, Lsp, Pitch, Speech, Tame, Test

0 16 32 48 64 80 96 112 128

Vector Length (VLMAX)

Lag_window (Full Optimization)

Relative Dynamic Instruction Count (%)

1.25 1.15 1.05 0.95 0.85 0.75 0.65 0.55 0.45 0.35 0.25

Algthm, Fixed, Lsp, Pitch, Speech, Tame, Test

0 16 32 48 64 80 96 112 128

Vector Length (VLMAX)

Lsp_get_quant (Full Optimization)

Relative Dynamic Instruction Count (%)

Algthm, Fixed, Lsp, Pitch, Speech, Tame, Test

Vector Length (VLMAX)

Lsp_get_tdist (Full Optimization)

Relative Dynamic Instruction Count (%)

Algthm, Fixed, Lsp, Pitch, Speech, Tame, Test

Vector Length (VLMAX)

Lsp_prev_compose (Full Optimization)

Relative Dynamic Instruction Count (%)

Algthm, Fixed, Lsp, Pitch, Speech, Tame, Test

Vector Length (VLMAX)

Lsp_prev_extract (Full Optimization)

Relative Dynamic Instruction Count (%)

0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8

0 16 32 48 64 80 96 112 128

Vector Length (VLMAX)

Algthm, Fixed, Lsp, Pitch, Speech, Tame, Test

Lsp_select_1 (Full Optimization)

Relative Dynamic Instruction Count (%)

0.25 0.3 0.35 0.4 0.45 0.5 0.55 0.6

0 16 32 48 64 80 96 112 128

Vector Length (VLMAX)

Algthm, Fixed, Lsp, Pitch, Speech, Tame, Test

Lsp_select_2 (Full Optimization)

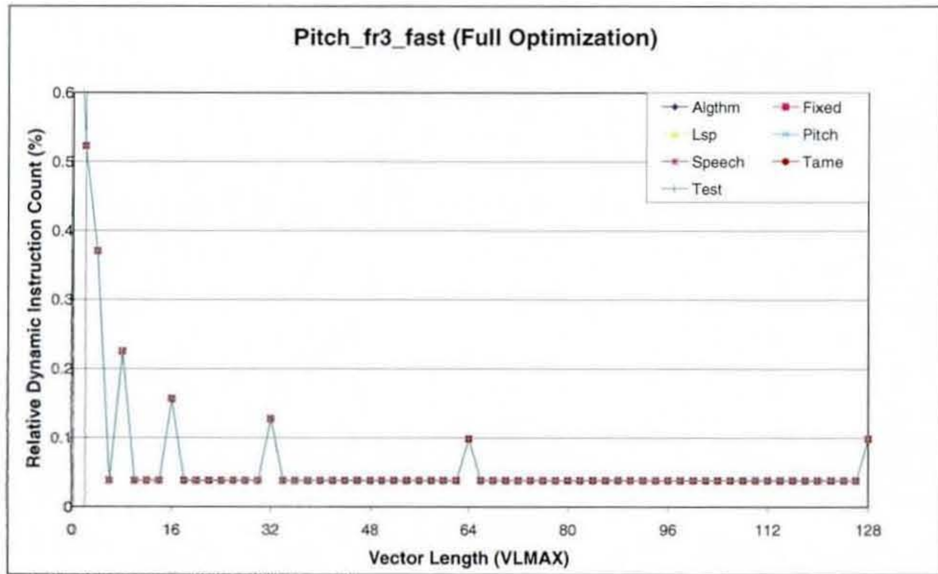
Relative Dynamic Instruction Count (%)

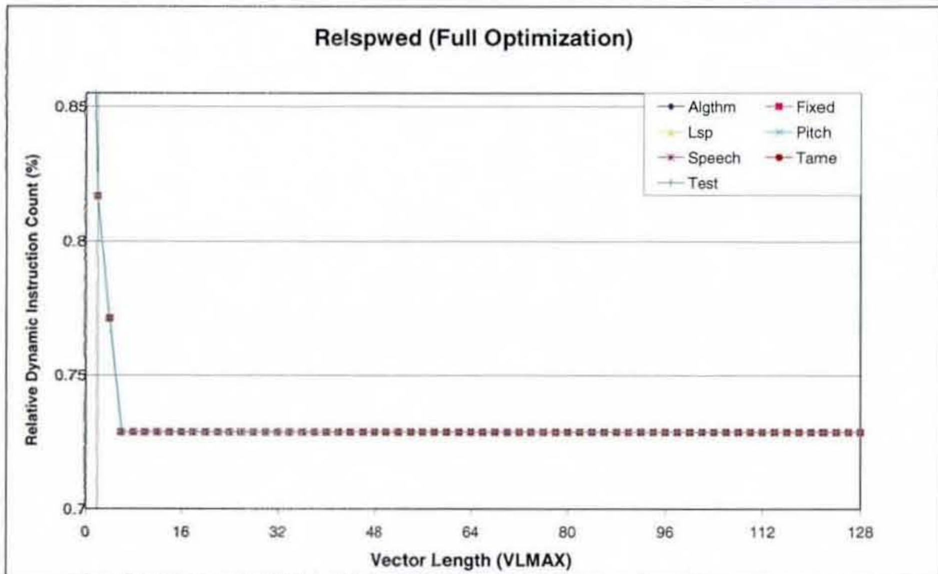
0.25 0.3 0.35 0.4 0.45 0.5 0.55 0.6

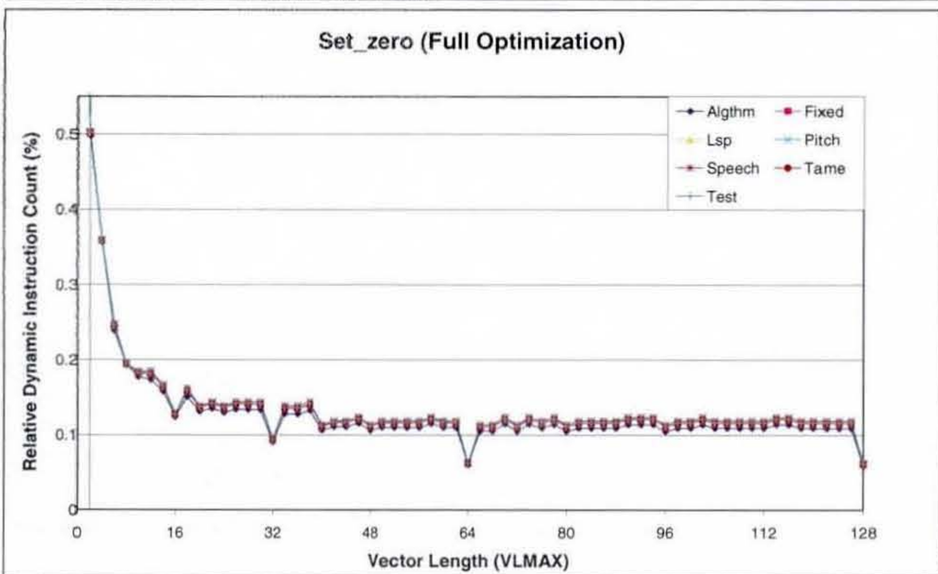
0 16 32 48 64 80 96 112 128

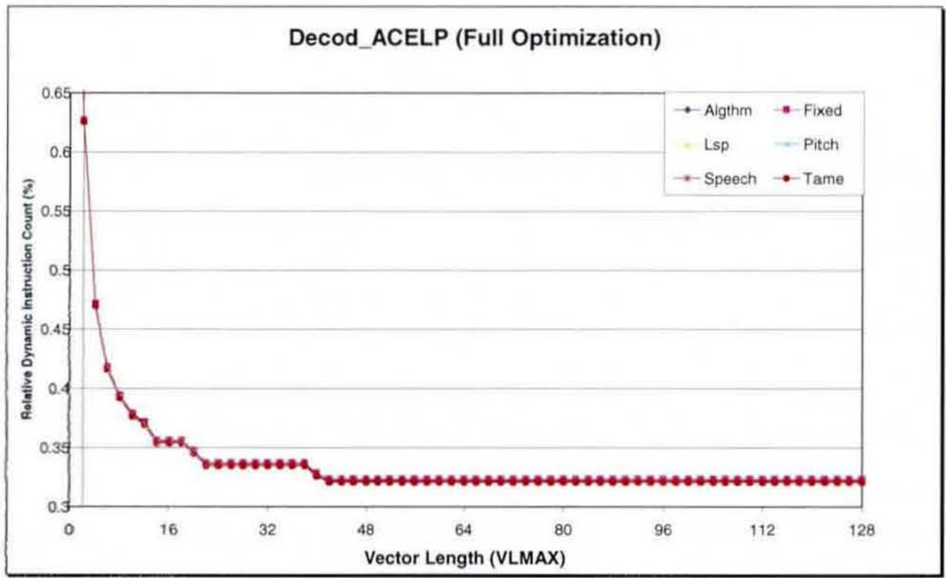
Vector Length (VLMAX)

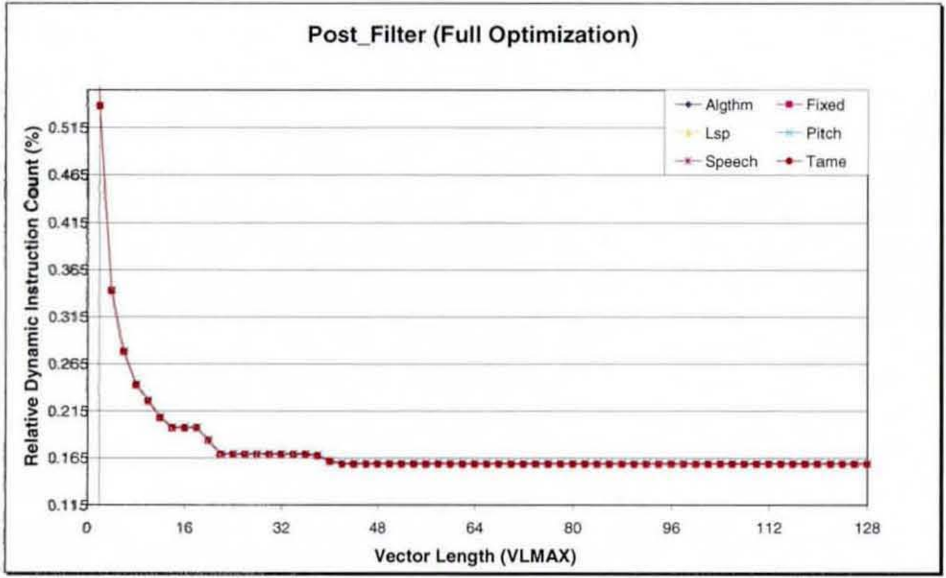
Algthm, Fixed, Lsp, Pitch, Speech, Tame, Test


Pitch_fr3_fast (Full Optimization)
Relative Dynamic Instruction Count (%)
Vector Length (VLMAX)
Algthm
Fixed
Lsp
Pitch
Speech
Tame
Test
0
0.1
0.2
0.3
0.4
0.5
0.6
16
32
48
64
80
96
112
128

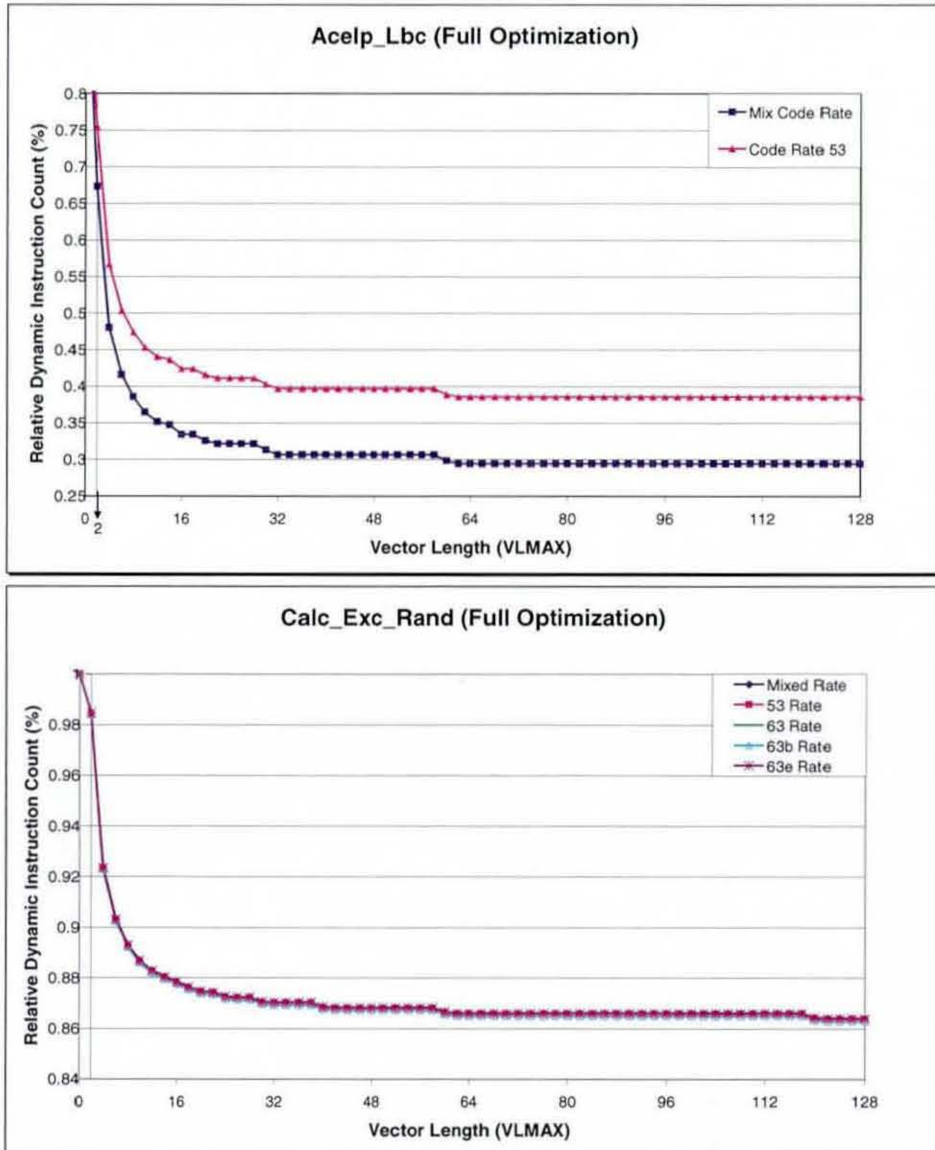

Relspwed (Full Optimization)
Relative Dynamic Instruction Count (%)
Vector Length (VLMAX)
Algthm
Fixed
Lsp
Pitch
Speech
Tame
Test
0.7
0.75
0.8
0.85
0
16
32
48
64
80
96
112
128

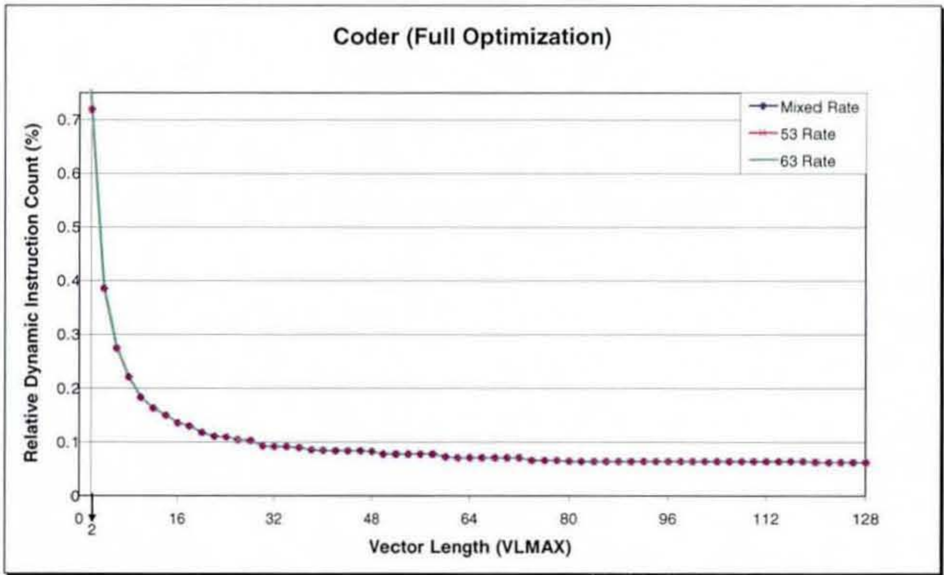

Set_zero (Full Optimization)
Relative Dynamic Instruction Count (%)
Vector Length (VLMAX)
Algthm
Fixed
Lsp
Pitch
Speech
Tame
Test
0
0.1
0.2
0.3
0.4
0.5
16
32
48
64
80
96
112
128

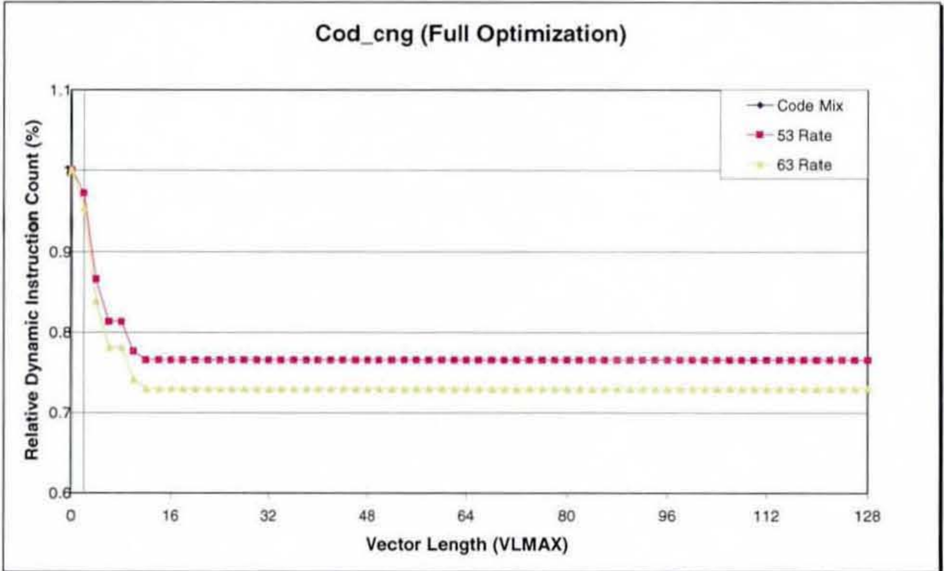

Decod_ACELP (Full Optimization)
Relative Dynamic Instruction Count (%)
0.65
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0
16
32
48
64
80
96
112
128
Vector Length (VLMAX)
Algthm
Fixed
Lsp
Pitch
Speech
Tame

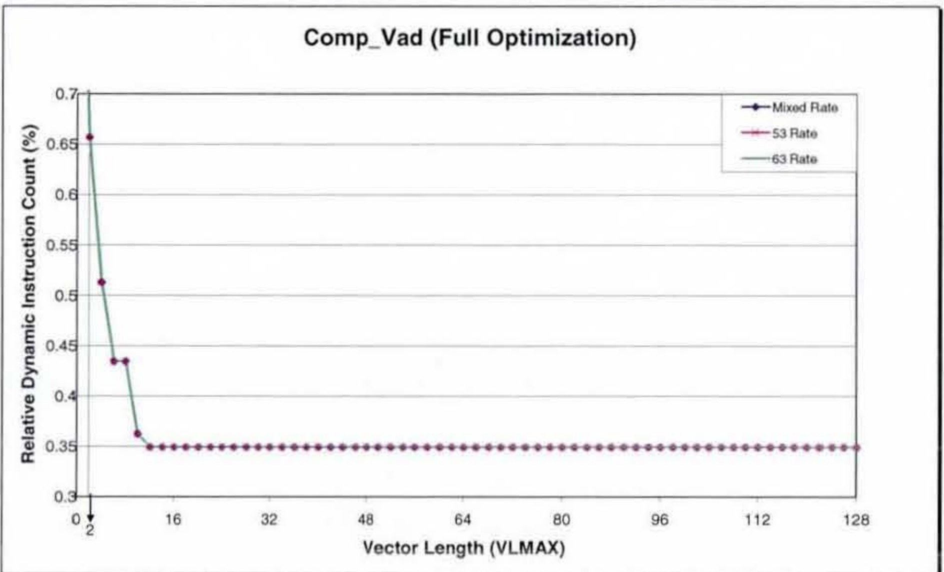

Post_Filter (Full Optimization)
Relative Dynamic Instruction Count (%)
0.515
0.465
0.415
0.365
0.315
0.265
0.215
0.165
0.115
0
16
32
48
64
80
96
112
128
Vector Length (VLMAX)
Algthm
Fixed
Lsp
Pitch
Speech
Tame

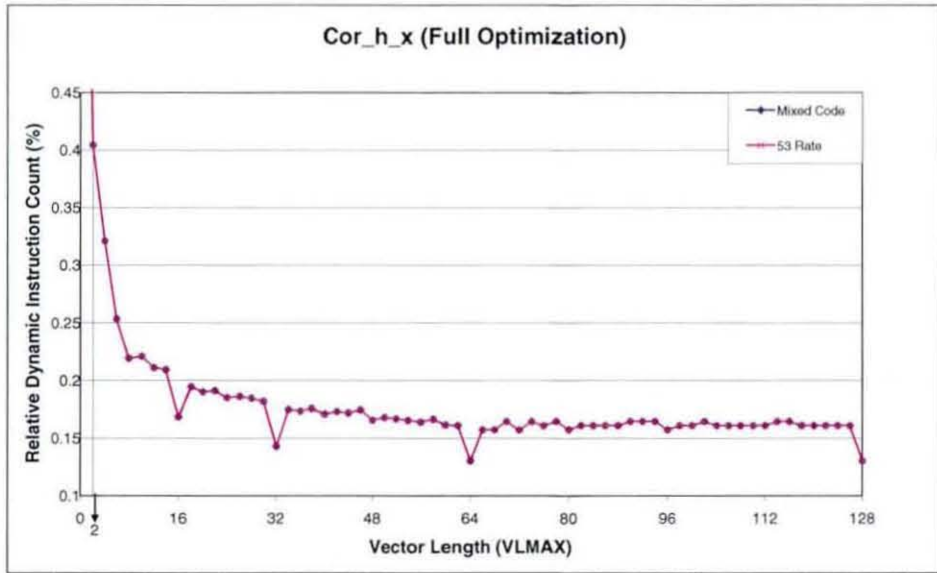
This section presents the results from the G.723.1 speech codec showing the improvement made from a function perspective.

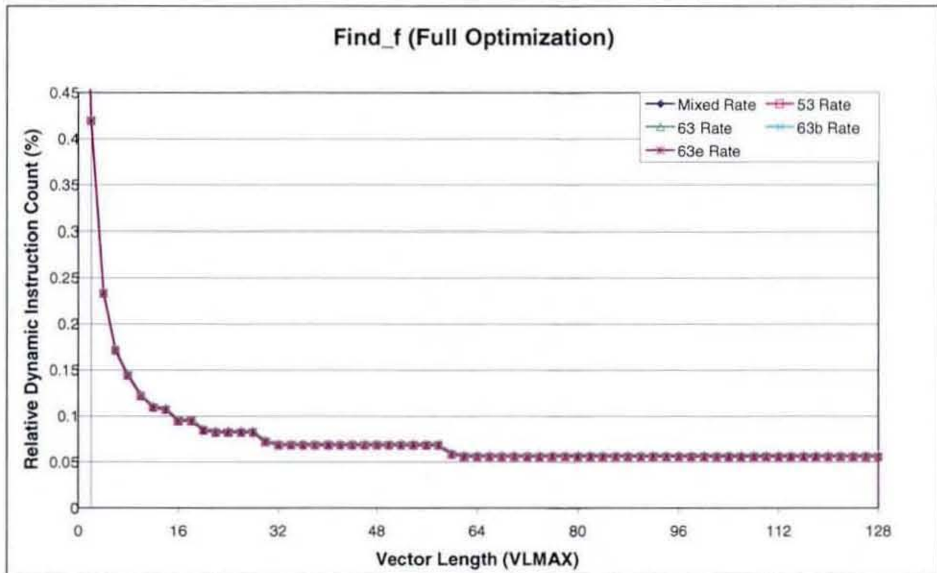


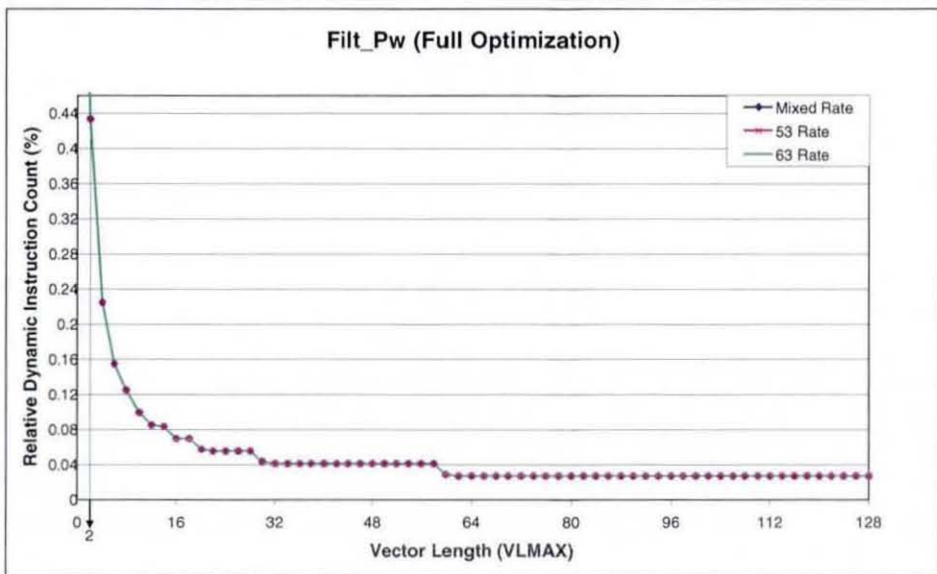

Coder (Full Optimization)
Mixed Rate
53 Rate
63 Rate
Relative Dynamic Instruction Count (%)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
2
16
32
48
64
80
96
112
128
Vector Length (VLMAX)

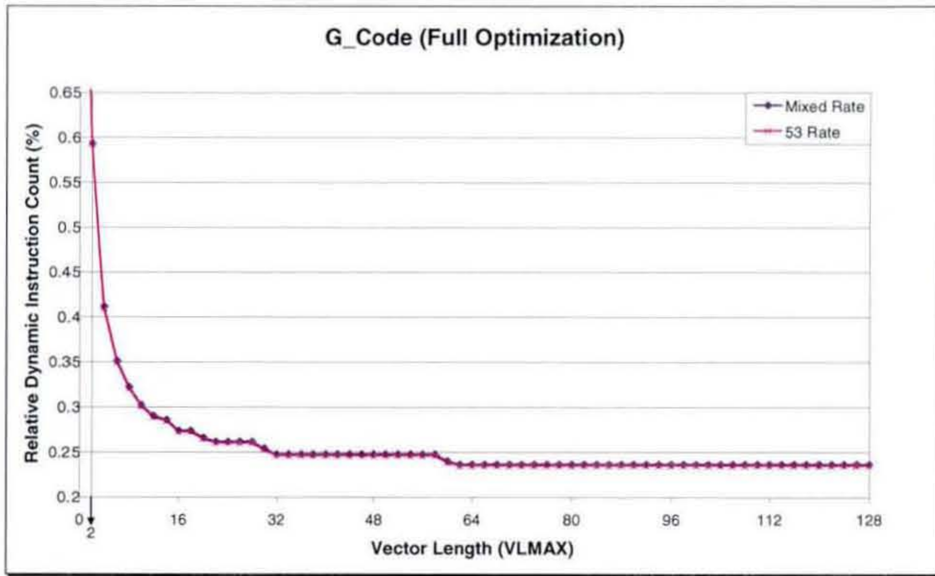

Cod_cng (Full Optimization)
Code Mix
53 Rate
63 Rate
Relative Dynamic Instruction Count (%)
0.6
0.7
0.8
0.9
1
1.1
0
16
32
48
64
80
96
112
128
Vector Length (VLMAX)

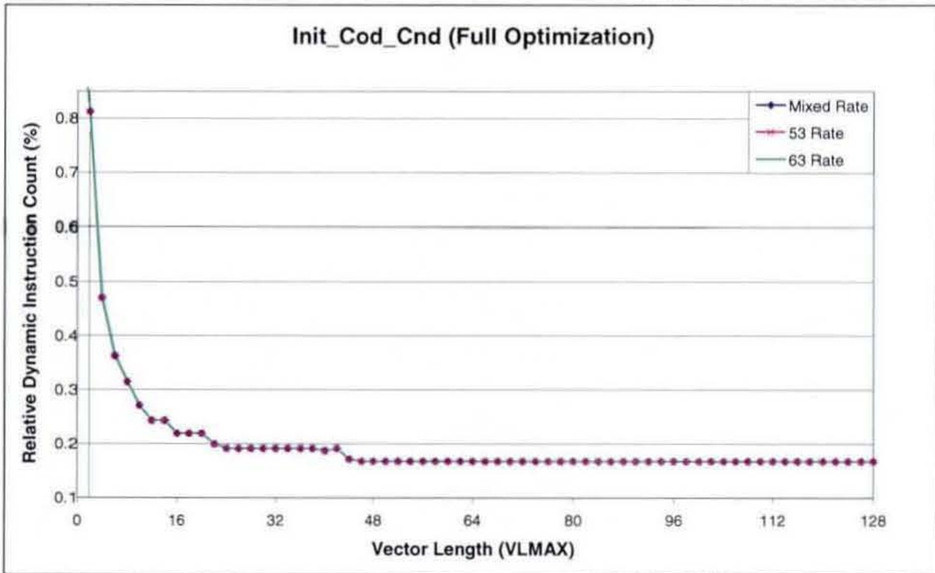

Comp_Vad (Full Optimization)
Mixed Rate
53 Rate
63 Rate
Relative Dynamic Instruction Count (%)
0.3
0.35
0.4
0.45
0.5
0.55
0.6
0.65
0.7
0
2
16
32
48
64
80
96
112
128
Vector Length (VLMAX)

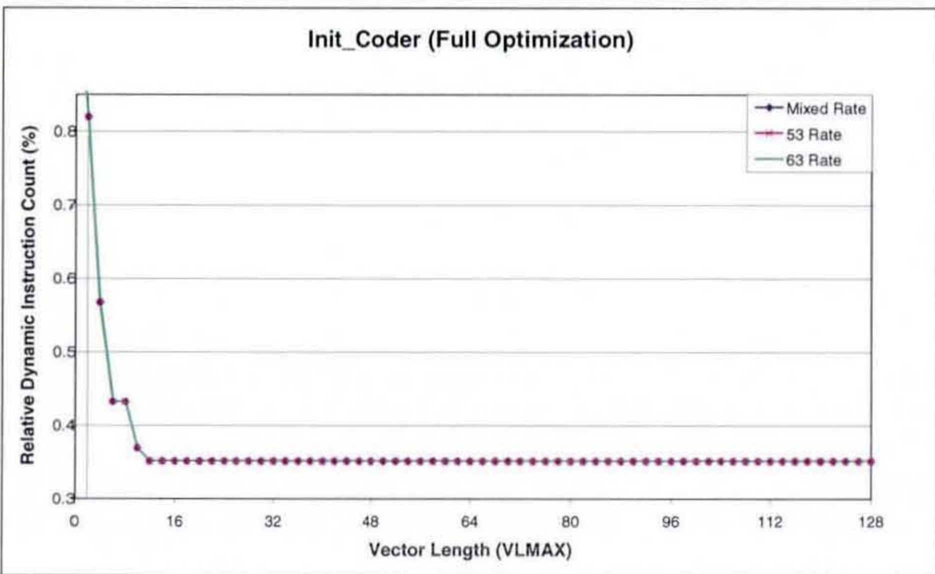

Cor_h_x (Full Optimization)
Relative Dynamic Instruction Count (%)
Vector Length (VLMAX)
Mixed Code
53 Rate
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0
2
16
32
48
64
80
96
112
128

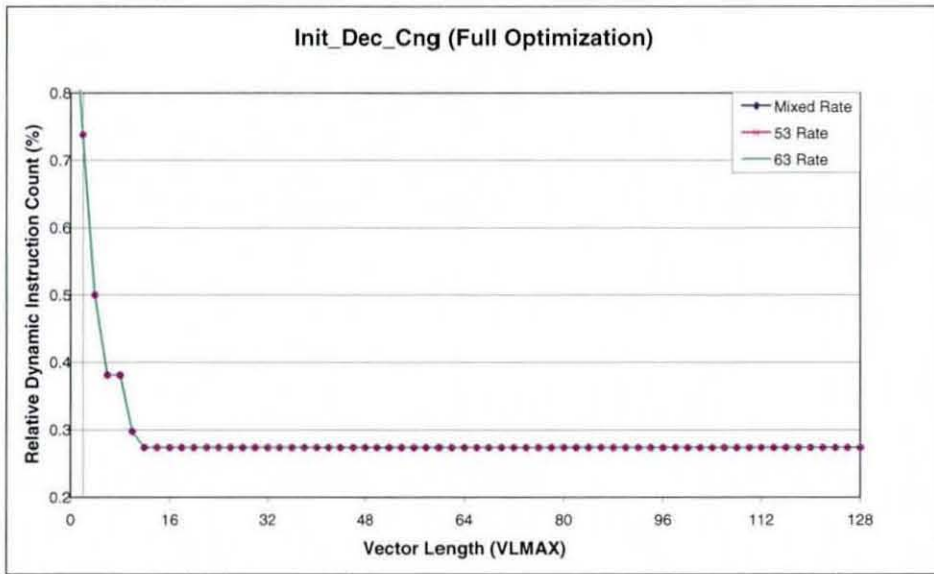

Find_f (Full Optimization)
Relative Dynamic Instruction Count (%)
Vector Length (VLMAX)
Mixed Rate
53 Rate
63 Rate
63b Rate
63e Rate
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
16
32
48
64
80
96
112
128

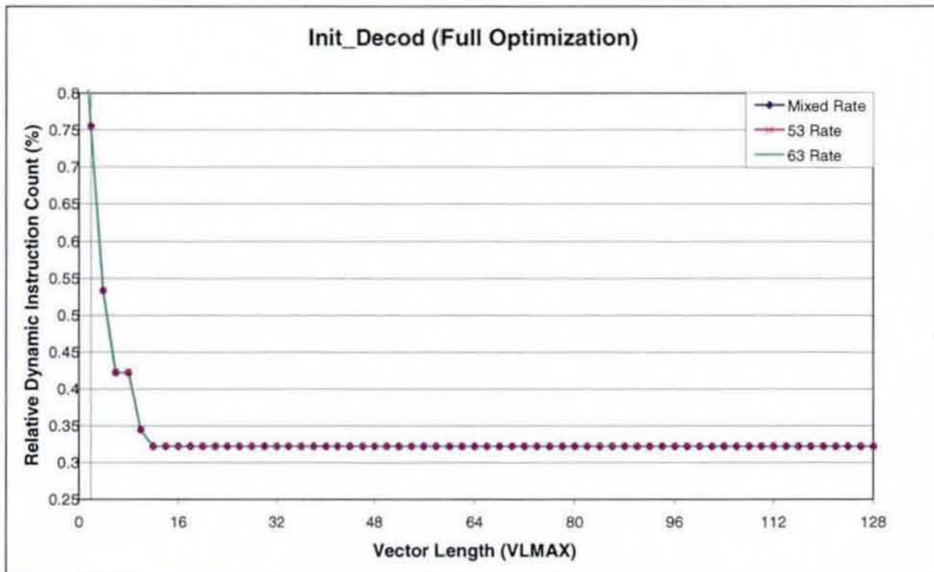

Filt_Pw (Full Optimization)
Relative Dynamic Instruction Count (%)
Vector Length (VLMAX)
Mixed Rate
53 Rate
63 Rate
0.44
0.4
0.36
0.32
0.28
0.24
0.2
0.16
0.12
0.08
0.04
0
0
2
16
32
48
64
80
96
112
128

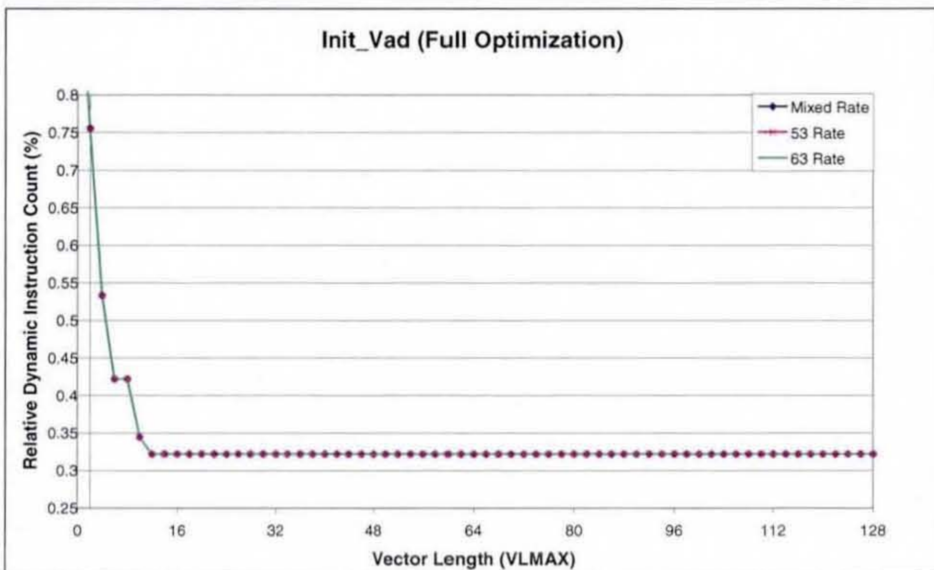

G_Code (Full Optimization)
Relative Dynamic Instruction Count (%)
Vector Length (VLMAX)
Mixed Rate
53 Rate
0.65
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0
2
16
32
48
64
80
96
112
128

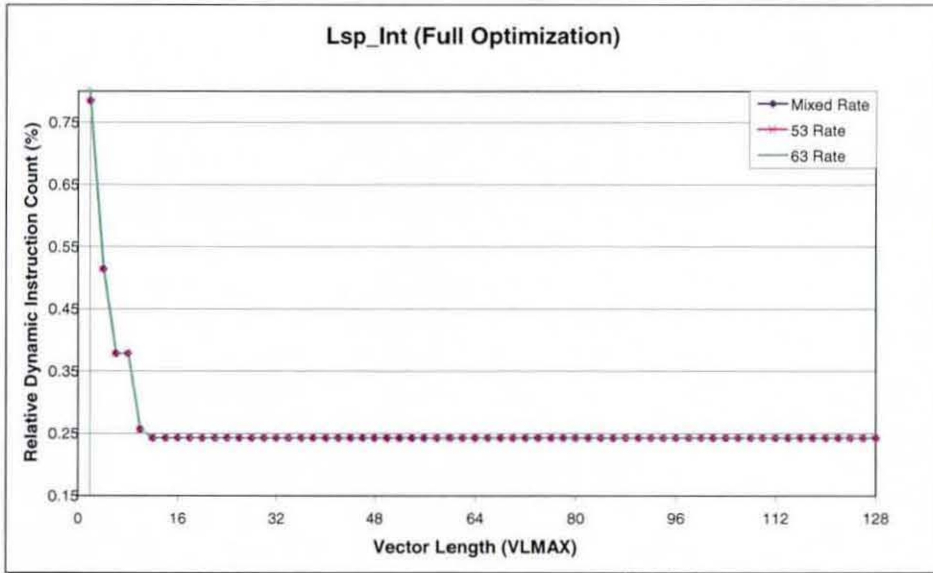

Init_Cod_Cnd (Full Optimization)
Relative Dynamic Instruction Count (%)
Vector Length (VLMAX)
Mixed Rate
53 Rate
63 Rate
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
16
32
48
64
80
96
112
128

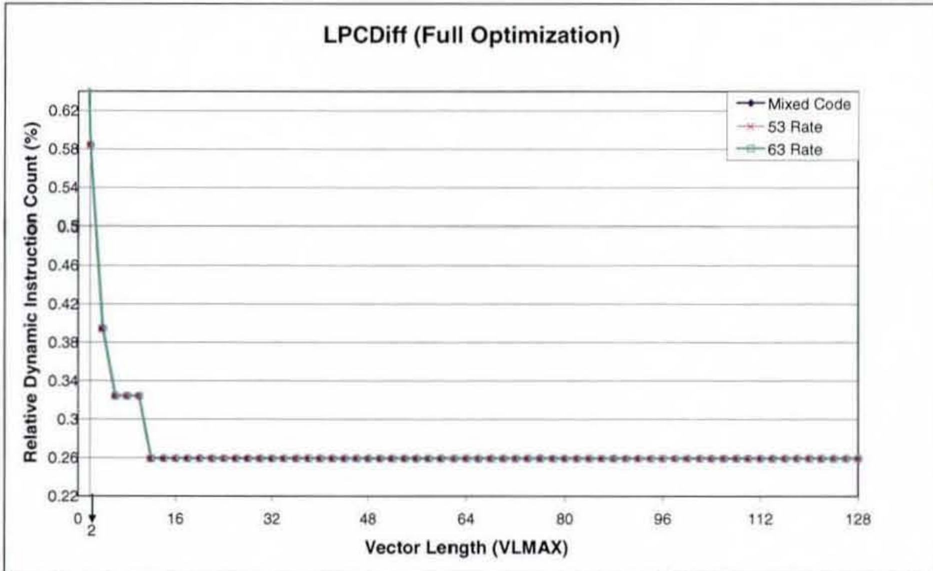

Init_Coder (Full Optimization)
Relative Dynamic Instruction Count (%)
Vector Length (VLMAX)
Mixed Rate
53 Rate
63 Rate
0.8
0.7
0.6
0.5
0.4
0.3
0
16
32
48
64
80
96
112
128

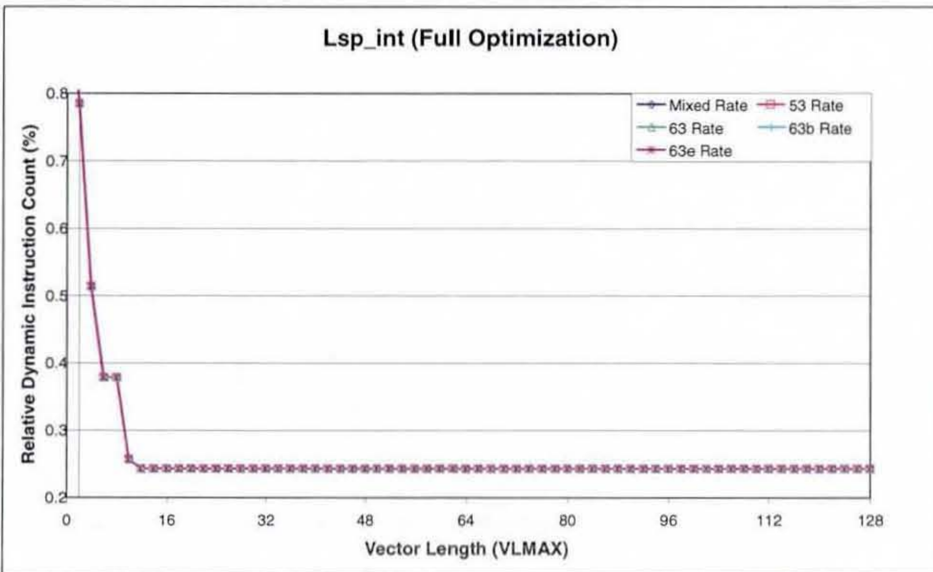

Init_Dec_Cng (Full Optimization)
Relative Dynamic Instruction Count (%)
0.8
0.7
0.6
0.5
0.4
0.3
0.2
Mixed Rate
53 Rate
63 Rate
0
16
32
48
64
80
96
112
128
Vector Length (VLMAX)

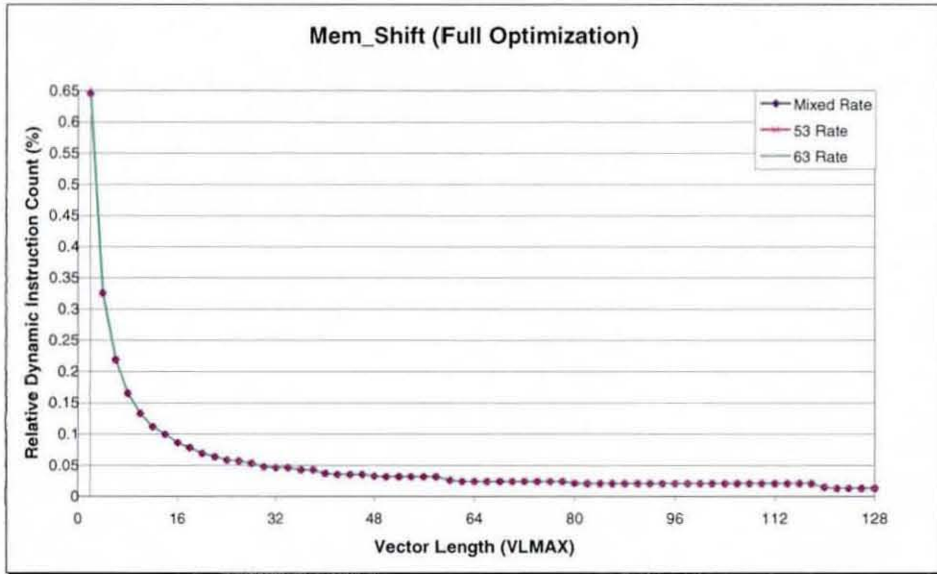

Init_Decod (Full Optimization)
Relative Dynamic Instruction Count (%)
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0.25
Mixed Rate
53 Rate
63 Rate
0
16
32
48
64
80
96
112
128
Vector Length (VLMAX)

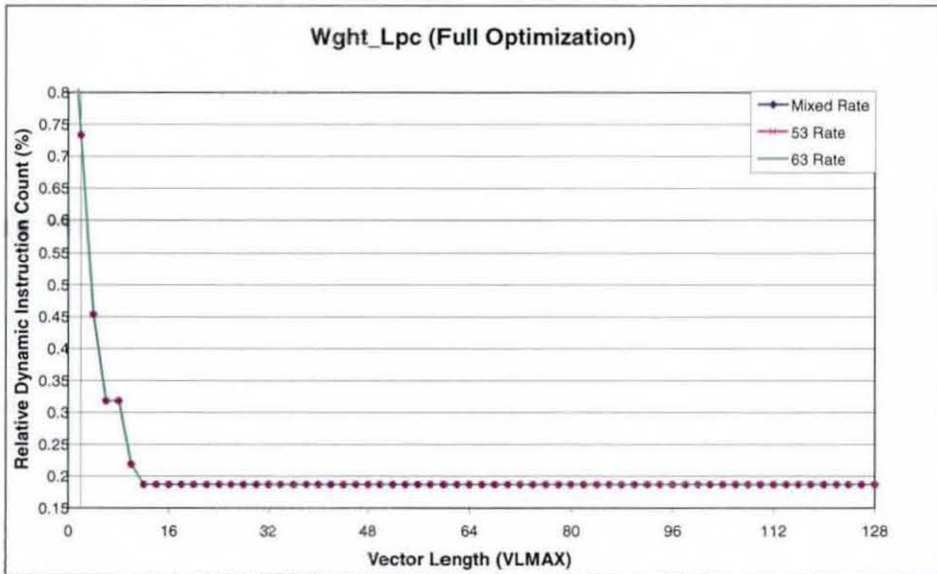

Init_Vad (Full Optimization)
Relative Dynamic Instruction Count (%)
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0.25
Mixed Rate
53 Rate
63 Rate
0
16
32
48
64
80
96
112
128
Vector Length (VLMAX)


Lsp_Int (Full Optimization)
Mixed Rate
53 Rate
63 Rate
Relative Dynamic Instruction Count (%)
0.75
0.65
0.55
0.45
0.35
0.25
0.15
0
16
32
48
64
80
96
112
128
Vector Length (VLMAX)


LPCDiff (Full Optimization)
Mixed Code
53 Rate
63 Rate
Relative Dynamic Instruction Count (%)
0.62
0.58
0.54
0.5
0.46
0.42
0.38
0.34
0.3
0.26
0.22
0
2
16
32
48
64
80
96
112
128
Vector Length (VLMAX)


Lsp_int (Full Optimization)
Mixed Rate
53 Rate
63 Rate
63b Rate
63e Rate
Relative Dynamic Instruction Count (%)
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
16
32
48
64
80
96
112
128
Vector Length (VLMAX)


Mem_Shift (Full Optimization)
Relative Dynamic Instruction Count (%)
0.65
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
16
32
48
64
80
96
112
128
Vector Length (VLMAX)
Mixed Rate
53 Rate
63 Rate


Wght_Lpc (Full Optimization)
Relative Dynamic Instruction Count (%)
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0
16
32
48
64
80
96
112
128
Vector Length (VLMAX)
Mixed Rate
53 Rate
63 Rate

AUTHOR'S PUBLICATIONS

The following are the publications that have resulted from the work in this thesis.

[1] V. A. Chouliaras, J. L. Nunez, S. R. Parr, K. Koutsomyti, D. J. Mulvaney and S. Datta, "Development of custom vector accelerator for high-performance speech coding", *IEE Electronic Letters*, vol. 40, November 2004, pp.1559-1561.

[2] K. Koutsomyti, S. R. Parr, V. A. Chouliaras, J. L. Nunez, D. J. Mulvaney and S. Datta, "Scalar and Parametric Vector Accelerators for the G.729A Speech Coding Standard", in *Proceedings of IEE/ACM Soc Design, Test and Technology Postgraduate Seminar*, Loughborough University, September 2004 , pp. 53-57.

[3] K. Koutsomyti, S. R. Parr, V. A. Chouliaras, J. L. Nunez, D. J. Mulvaney and S. Datta, "Configurable Scalar and Vector Accelerators for the G.729A and G.723.1 Speech Coding Standards", in *Proceedings of Postgraduate Research Conference in Electronics, Photonics, Communications and Networks, and Computing Science (PREP2005)*, Lancaster University, March 2005, pp. 62-63.

[4] S. R. Parr, K. Koutsomyti, V. A. Chouliaras, J. L. Nunez and D. J. Mulvaney, "Configurable scalar and Vector Coprocessors for accelerating the G.723.1 and G.729.A speech coders", in *Proceedings of the International Conference on Signal and Image Processing*, Novosibirsk, Russia, June 2005, pp.340-344.

[5] K. Koutsomyti, S. R. Parr, V. A. Chouliaras and J. L. Nunez, "Applying Data-Parallel and Scalar Optimizations for the efficient implementation of the G.729A and G.723.1 Speech Coding Standards", in *Proceedings of the 7th IASTED International Conference on Signal and Image Processing (SIP 2005)*, Honolulu, USA, August 2005, pp. 40-45.

[6] V. A. Chouliaras, K. Koutsomyti, T. R. Jacobs and S. R. Parr, D. J. Mulvaney and R. Thomson, "SystemC-defined SIMD instructions for high performance SoC architectures", in *13th IEEE International Conference on Electronics, Circuits and Systems*, Nice, France, December 2006, pp. 822-825.

[7] V. A. Chouliaras, K. Koutsomyti, T. R. Jacobs, S. R. Parr, D. J. Mulvaney and R. Thomson, "SystemC-defined SIMD instructions for a CMP/SMT ASIC platform", in *Proceedings of the 24th IEEE Norchip conference in ASIC design*, Linkoping, Sweden, November 2006, pp. 285-288.

[8] K. Koutsomyti, V. A. Chouliaras, S. R. Parr, J. L. Nunez and S. Datta, "Accelerating speech coding standards through SystemC synthesized SIMD and Scalar accelerators", in *Proceedings of the IEEE International Conference on Consumer Electronics (ICCE06)*, Las Vegas, USA, pp. 279-280.

[9] S. R. Parr, K. Koutsomyti, V. A. Chouliaras, "A High Bandwidth Configurable Load/Store Unit for an Embedded Vector Processor", in Postgraduate *Workshop on Microelectronics and Embedded Systems*, Birmingham, UK, October 2006.