High-performance load/store unit for a highly configurable, embedded vector processor

This item was submitted to Loughborough University's Institutional Repository by the/an author.

Additional Information:


Metadata Record: https://dspace.lboro.ac.uk/2134/33579

Publisher: © Simon R. Parr

Rights: This work is made available according to the conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0) licence. Full details of this licence are available at: https://creativecommons.org/licenses/by-nc-nd/4.0/

Please cite the published version.
High Performance Load/Store Unit for a Highly Configurable, Embedded Vector Processor

by

Simon Richard Parr, M Eng MIET

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

February 09

© Simon R. Parr, 2008
ABSTRACT

Voice-over-Internet-Protocol (VoIP) has gained a significant amount of interest due to insatiable demand for improved digital communication in the consumer market. Instead of using the traditional public exchange network telephone companies are now able to offer cheap telephone calls via the internet. The intention of this research is to improve the channel capacity of VoIP networks by researching a novel embedded CPU architecture for accelerating speech coding algorithms. The proposed architecture is a configurable vector coprocessor, closely coupled to a controlling Sparc-V8 compliant CPU. The design is developed as a System-on-Chip (SoC) component utilising high-performance connectivity.

Two speech codecs, provided by the International Telecommunication Union (ITU) and usually found in VoIP applications are the primary workloads studied in this research. By implementing data-level-parallel hardware in the form of custom vector and scalar instructions, the benefit of data-level-parallelism is investigated. The vector instructions are designed to accelerate the inner-loops of the two speech algorithms and results are obtained over a range of vector lengths and different test vectors, provided by the ITU. The final section of this research is the design of a configurable vector Load/Store Unit (LSU). This was implemented at Register Transfer Level (RTL) Very High Speed Integrated Circuit Hardware Description Language (VHDL) and is included in the overall design of the vector coprocessor. Different configurations of the LSU were explored to give the cycle, area and power results across a number of speech workloads.

Keywords:
ACKNOWLEDGMENTS

I first must thank my supervisor and project leader, Dr Vassilios Chouliaras. His industrial expertise in the design and implementation of scalar, superscalar and vector embedded microprocessors is ultimately responsible for this thesis. His assistance and advice has helped guide the project team in the development of the configurable vector coprocessor for accelerating speech coding algorithms. Secondly, I would like to thank my lab partner Konstantia Koutsomyti for her help in getting the excellent results gained from architectural exploration and for the help provided in writing conference/journal papers. I would also like to thank Dr David Mulvaney. As well as being my second supervisor, he also acted as councillor whose door was always open for a chat. My thanks also go to the rest of the research group, Tom Jacobs, Yibin Li and Robert Thomson in providing a pleasant working environment and all their helpful advice.

I am very grateful to the help of Sarah Cooper. Not only has she kept me fed, watered, busy and sane over my PhD years, she took on the tedious task of reading this thesis even though she knows nothing about microprocessors.

Finally, I would like to thank Loughborough District Scouts and in particular 1st Nanpantan Scout Group for keeping me entertained and busy. They enabled me to unwind from working and writing with pleasant distractions.
# CONTENTS

Abstract .................................................................................................................................................................iii
Acknowledgments ......................................................................................................................................................iv
Contents .................................................................................................................................................................v
List of Abbreviations and Symbols ............................................................................................................................vii
List of Figures ..........................................................................................................................................................x
List of Tables ..........................................................................................................................................................xiii

1 Introduction ..........................................................................................................................................................1
  1.1 Problem .........................................................................................................................................................1
  1.2 Voice over Internet Protocol (VoIP) ..................................................................................................................3
  1.3 Speech Compression ........................................................................................................................................7
  1.4 Embedded Processors ...................................................................................................................................10
  1.5 Configurable Extensible Processors .............................................................................................................11
  1.6 Configurable vs. Reconfigurable Hardware .................................................................................................12
  1.7 Contributions ...............................................................................................................................................15
  1.8 References ..................................................................................................................................................17

2 Speech Coding ..................................................................................................................................................21
  2.1 VoIP and Speech Coding .................................................................................................................................21
  2.2 Previous Speech Coding Research ..................................................................................................................22
  2.3 Speech Codecs ..............................................................................................................................................23
  2.4 The G.729A and G.723.1 Speech Codecs ....................................................................................................32
  2.5 Conclusion ...................................................................................................................................................52
  2.6 References ..................................................................................................................................................53

3 Parallelism .........................................................................................................................................................56
  3.1 Introduction ...................................................................................................................................................56
  3.2 Types of Parallelism ......................................................................................................................................58
  3.3 Parallelism Types .........................................................................................................................................60
  3.4 Parallelism Background .................................................................................................................................68
  3.5 Profiling speech codecs ................................................................................................................................71
  3.6 References ..................................................................................................................................................72

4 Architectural Results ...........................................................................................................................................76
  4.1 Introduction ...................................................................................................................................................76
  4.2 The SimpleScalar Toolset .............................................................................................................................76
  4.3 Profiling the Speech Coders ...........................................................................................................................77
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td></td>
<td>Programmer's Model</td>
<td>80</td>
</tr>
<tr>
<td>4.5</td>
<td></td>
<td>Identifying Custom Instructions</td>
<td>81</td>
</tr>
<tr>
<td>4.6</td>
<td></td>
<td>Implementation of the Custom Instructions</td>
<td>87</td>
</tr>
<tr>
<td>4.7</td>
<td></td>
<td>Architecture Exploration Results</td>
<td>94</td>
</tr>
<tr>
<td>4.8</td>
<td></td>
<td>Function Level Architecture Results</td>
<td>97</td>
</tr>
<tr>
<td>4.9</td>
<td></td>
<td>Conclusion</td>
<td>100</td>
</tr>
<tr>
<td>4.10</td>
<td></td>
<td>References</td>
<td>101</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Vector Processor</td>
<td>102</td>
</tr>
<tr>
<td>5.1</td>
<td></td>
<td>Introduction</td>
<td>102</td>
</tr>
<tr>
<td>5.2</td>
<td></td>
<td>The Leon3 System</td>
<td>102</td>
</tr>
<tr>
<td>5.3</td>
<td></td>
<td>Coprocessor Instructions</td>
<td>105</td>
</tr>
<tr>
<td>5.4</td>
<td></td>
<td>Load/Store Units Background</td>
<td>109</td>
</tr>
<tr>
<td>5.5</td>
<td></td>
<td>An Overview of a Cache</td>
<td>110</td>
</tr>
<tr>
<td>5.6</td>
<td></td>
<td>Load/Store Unit Microarchitecture</td>
<td>115</td>
</tr>
<tr>
<td>5.7</td>
<td></td>
<td>Cycle Results</td>
<td>124</td>
</tr>
<tr>
<td>5.8</td>
<td></td>
<td>Power and Area Results</td>
<td>128</td>
</tr>
<tr>
<td>5.9</td>
<td></td>
<td>Conclusion</td>
<td>131</td>
</tr>
<tr>
<td>5.10</td>
<td></td>
<td>References</td>
<td>133</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Conclusion</td>
<td>135</td>
</tr>
<tr>
<td>6.1</td>
<td></td>
<td>Project Findings</td>
<td>135</td>
</tr>
<tr>
<td>6.2</td>
<td></td>
<td>Future Work</td>
<td>139</td>
</tr>
<tr>
<td>6.3</td>
<td></td>
<td>References</td>
<td>140</td>
</tr>
</tbody>
</table>

Appendix A - G.723.1 Results | A1
Appendix B - G.729A Results | B1
Appendix C - G.723.1 Function Results | C1
Appendix D - Author's Publications | D1
**LIST OF ABBREVIATIONS AND SYMBOLS**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbS</td>
<td>Analysis-by-Synthesis</td>
</tr>
<tr>
<td>ACELP</td>
<td>Algebraic CELP</td>
</tr>
<tr>
<td>ADPCM</td>
<td>Adaptive Differential Pulse Code Modulation</td>
</tr>
<tr>
<td>AHB</td>
<td>AMBA High-performance Bus</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>AMBA</td>
<td>Advanced Microcontroller Bus Architecture</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuits</td>
</tr>
<tr>
<td>BASOP</td>
<td>Basic Arithmetic Operations</td>
</tr>
<tr>
<td>CCITT</td>
<td>Comité Consultatif International</td>
</tr>
<tr>
<td>CDFG</td>
<td>Control Data Flow Graph</td>
</tr>
<tr>
<td>CELP</td>
<td>Code Excited Linear Prediction</td>
</tr>
<tr>
<td>CNG</td>
<td>Comfort Noise Generator</td>
</tr>
<tr>
<td>CPI</td>
<td>Clocks-per-Instructions</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CS-ACELP</td>
<td>Conjugate Structure Algebraic CELP</td>
</tr>
<tr>
<td>CS-CELP</td>
<td>Conjugate Structure CELP</td>
</tr>
<tr>
<td>DFT</td>
<td>Discrete Fourier Transform</td>
</tr>
<tr>
<td>DLP</td>
<td>Data Level Parallelism</td>
</tr>
<tr>
<td>DRAM</td>
<td>Dynamic RAM</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>EDA</td>
<td>Electronic Design Automation</td>
</tr>
<tr>
<td>EPSRC</td>
<td>Engineering and Physical Sciences Research Council</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In, First Out</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>FU</td>
<td>Functional Unit</td>
</tr>
<tr>
<td>GPL</td>
<td>General Public License</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Expansion</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
</tr>
<tr>
<td>IIR</td>
<td>Infinite Impulse Response</td>
</tr>
<tr>
<td>ILP</td>
<td>Instruction Level Parallelism</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>ISDN</td>
<td>Integrated Services Digital Network</td>
</tr>
<tr>
<td>ITU</td>
<td>International Telecommunications Union</td>
</tr>
<tr>
<td>ITU-T</td>
<td>ITU-Telecommunications</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>LFU</td>
<td>Least Frequently Used</td>
</tr>
<tr>
<td>LP</td>
<td>Linear Prediction</td>
</tr>
<tr>
<td>LPC</td>
<td>Linear Prediction Coding</td>
</tr>
<tr>
<td>LPC</td>
<td>Linear Predictive Coefficients</td>
</tr>
<tr>
<td>LRU</td>
<td>Least Recently Used</td>
</tr>
<tr>
<td>LSB</td>
<td>Least Significant Bits</td>
</tr>
<tr>
<td>LSF</td>
<td>Line Spectral Frequencies</td>
</tr>
<tr>
<td>LSP</td>
<td>Line Spectrum Pairs</td>
</tr>
<tr>
<td>LSU</td>
<td>Load/Store Unit</td>
</tr>
<tr>
<td>MA</td>
<td>Moving Average</td>
</tr>
<tr>
<td>MELP</td>
<td>Mixed Excitation Linear Predictive</td>
</tr>
<tr>
<td>MIMD</td>
<td>Multiple Instructions, Multiple Data</td>
</tr>
<tr>
<td>MIPS</td>
<td>Million Instructions per Second</td>
</tr>
<tr>
<td>MISD</td>
<td>Multiple Instructions, Single Data</td>
</tr>
<tr>
<td>MOS</td>
<td>Mean Opinion Score</td>
</tr>
<tr>
<td>MPE</td>
<td>Multi-Pulse Excited</td>
</tr>
<tr>
<td>MP-MLQ</td>
<td>Multi-Pulse Maximum Likelihood</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bits</td>
</tr>
<tr>
<td>OSI</td>
<td>Open Systems Interconnect</td>
</tr>
<tr>
<td>PBX</td>
<td>Private Branch Exchange</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
</tr>
<tr>
<td>PCM</td>
<td>Pulse Code Modulation</td>
</tr>
<tr>
<td>PDA</td>
<td>Personal Digital Assistant</td>
</tr>
<tr>
<td>PSTN</td>
<td>Public Switched Telephone Network</td>
</tr>
<tr>
<td>PSVQ</td>
<td>Predictive Split Vector Quantizer</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Expansion</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RAS</td>
<td>Registration, admission and status</td>
</tr>
<tr>
<td>RAW</td>
<td>Read after Write</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>RPE</td>
<td>Regular-Pulse Excited</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>SDRAM</td>
<td>Synchronous DRAM</td>
</tr>
<tr>
<td>SID</td>
<td>Silence Insertion Description</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single-Instruction-Multiple-Data</td>
</tr>
<tr>
<td>SIP</td>
<td>Session Initiation Protocol</td>
</tr>
<tr>
<td>SISD</td>
<td>Single Instruction, Single Data</td>
</tr>
<tr>
<td>SMTP</td>
<td>Simple Mail Transfer Protocol</td>
</tr>
<tr>
<td>SNR</td>
<td>Signal to Noise</td>
</tr>
<tr>
<td>SoC</td>
<td>System on Chip</td>
</tr>
<tr>
<td>SOI</td>
<td>Silicon on Insulator</td>
</tr>
<tr>
<td>SPARC</td>
<td>Scalable Processor ARCHitecture</td>
</tr>
<tr>
<td>SRAM</td>
<td>Static RAM</td>
</tr>
<tr>
<td>TLP</td>
<td>Thread Level Processors</td>
</tr>
<tr>
<td>VAD</td>
<td>Voice Activity Detection</td>
</tr>
<tr>
<td>VHDL</td>
<td>Very High Speed Integrated Circuit Hardware</td>
</tr>
<tr>
<td>VLIW</td>
<td>Description Language</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Long Instruction Word</td>
</tr>
<tr>
<td>VLR</td>
<td>Vector Length Register</td>
</tr>
<tr>
<td>VoIP</td>
<td>Voice over Internet Protocol</td>
</tr>
<tr>
<td>WAR</td>
<td>Write after Read</td>
</tr>
<tr>
<td>WAW</td>
<td>Write after Write</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1-1: Traditional (A) and VoIP (B) Network Infrastructures ........................................... 1
Figure 1-2: Transmission of SID and speech frames ................................................................. 2
Figure 1-3: OSI Reference Model ............................................................................................. 4
Figure 1-4: Typical VoIP Client for Voice Communications .................................................... 4
Figure 1-5: Packet Switched Network ........................................................................................ 5
Figure 1-6: IP Network to PSTN Gateway ............................................................................... 6
Figure 1-7: Protocol Architecture for H.323 ........................................................................... 6
Figure 1-8: The Human Vocal Tract and the Spectral Range of Speech ................................... 8
Figure 1-9: Spectral representations of Phonemes .................................................................. 9
Figure 1-10: Structure of an FPGA .......................................................................................... 13
Figure 1-11: FPGA Interconnect Switch ................................................................................. 14
Figure 2-1: The Spectrum Channel Vocoder ........................................................................... 25
Figure 2-2: The Formant Vocoder ........................................................................................... 26
Figure 2-3: Transfer function of the LPC Vocoder .................................................................... 27
Figure 2-4: Linear Prediction Codec ....................................................................................... 28
Figure 2-5: Analysis by Synthesis Codec ............................................................................... 29
Figure 2-6: CELP Coder Algorithm ........................................................................................ 31
Figure 2-7: G.729 Encoder Schematic .................................................................................... 33
Figure 2-8: LPC Analysis Window for G.729 ........................................................................ 34
Figure 2-9: LP Analysis Window Equation ............................................................................ 34
Figure 2-10: G.729 Vector Quantizer ..................................................................................... 35
Figure 2-11: Transfer Function of the Perceptual Weighted Filter .......................................... 36
Figure 2-12: Closed Loop mean-squared error equation ......................................................... 37
Figure 2-13: Codebook Vector ............................................................................................... 38
Figure 2-14: Modified Codebook Vector for low pitch delays .............................................. 38
Figure 2-15: G.729 Decoder Schematic .................................................................................. 40
Figure 2-16: Signal flow of G.729 Decoder ............................................................................ 41
Figure 2-17: Reconstructed Excitation Signal Equation ............................................................ 42
Figure 2-18: Reconstructed Speech Signal .............................................................................. 42
Figure 2-19: G.723.1 Encoder Schematic .............................................................................. 44
Figure 2-20: Block diagram of the G.723.1 Speech Decoder ................................................... 51
Figure 3-1: Simple 7-stage Integer RISC Pipeline ................................................................... 56
Figure 3-2: Sample Code showing data dependencies ............................................................. 59
Figure 5-7: Two-way set associative cache scheme .............................................. 114
Figure 5-8: Microarchitecture outline for the Load/Store Unit .......................... 115
Figure 5-9: Load/Store Addressing .................................................................. 116
Figure 5-10: Checking TAG .............................................................................. 116
Figure 5-11: Even Unaligned Access ................................................................. 117
Figure 5-12: Odd Unaligned Access ................................................................. 118
Figure 5-13: FSM interface .............................................................................. 118
Figure 5-14: AHB Read Transactions ............................................................... 119
Figure 5-15: AHB Write Transactions ............................................................. 120
Figure 5-16: FSM State Diagram ...................................................................... 121
Figure 5-17: Enabling Data in RAMs ............................................................... 123
Figure 5-18: Even Vector Cache Sort .............................................................. 123
Figure 5-19: Odd Vector Cache Sort ............................................................... 124
Figure 5-20: Testbench Configuration for LSU ............................................... 125
Figure 5-21: G.723.1 Encoder at 5.3 kbps Clocks-per-Instruction Results ....... 126
Figure 5-22: G.723.1 Encoder at 6.3 kbps Clocks-per-Instruction Results ....... 126
Figure 5-23: G.723.1 Mixed Rate Encoder Clocks-per-Instruction Results ...... 127
Figure 5-24: Area results for 16 Kbyte cache at 250MHz .............................. 128
Figure 5-25: Statistical Power Results for 8 Kbytes Cache (High Power) ......... 129
Figure 5-26: Statistical Power Results for 8 Kbytes Cache (Low Power) ........... 129
Figure 5-27: Statistical Power Results for 16 Kbytes Cache (High Power) ....... 130
Figure 5-28: Statistical Power Results for 16 Kbytes Cache (Low Power) ...... 130
Figure 5-29: LSU VLSI Layout - VLEN 8, 4-Way, 2 Kbytes Cache Size .......... 131
Figure 6-1: VLSI Layout for Vector Coprocessor (VLSU and Vector Datapath) 137
LIST OF TABLES

Table 1-1: ITU Standards for Voice Compression ................................................................. 3
Table 1-2: Exclusive OR Truth Table ...................................................................................... 13
Table 2-1: Structure of the Fixed Codebook for G.729 .......................................................... 38
Table 2-2: Bit Allocation of the 8 kbits G.729 and G.729A codecs ........................................ 40
Table 2-3: ACELP excitation codebook for G.723.1 ............................................................. 47
Table 2-4: Bit allocation of the 6.3 kbit/s coding algorithm for G.723.1 ................................... 49
Table 2-5: Bit allocation of the 5.3 kbit/s coding algorithm for G.723.1 ................................... 49
Table 2-6: List of transmitted Parameters from the G723.1 Coder ........................................ 50
Table 3-1: 7-Stage RISC Pipeline .......................................................................................... 57
Table 3-2: Relative number of total instructions executed outside the DSP emulated instructions simulated on Linux 86 ................................................................. 72
Table 3-3: Relative number of total instructions executed outside the DSP emulated instructions simulated on the SimpleScalar Instruction Set ......................................................... 72
Table 4-1: Unmodified G723.1 instruction count .................................................................. 77
Table 4-2: Unmodified G.729A instruction count ................................................................. 77
Table 4-3: Profiling functions for the G.723.1 encoder at 6.3 kbits/s ........................................ 78
Table 4-4: Profiling for worst DSP functions for G.723.1 encoder at 6.3 kbits/s ...................... 79
Table 4-5: SimpleScalar Vector Instructions ........................................................................ 92
Table 4-6: SimpleScalar Scalar Instructions .......................................................................... 93
Table 4-7: Profiling functions for G.723.1 encoder at 6.3 kbits/s with coprocessor instructions .......................................................................................................................... 97
Table 5-1: Coprocessor Opcodes for Vector Instructions ...................................................... 107
Table 5-2: Coprocessor Opcodes for Scalar Instructions ..................................................... 108
Table 6-1: VLSI Physical Layout Properties ........................................................................... 137
1 INTRODUCTION

1.1 Problem

In recent years there has been an increasing move from an existing two-tier network infrastructure to a single network infrastructure. Traditionally, the two separate networks included the data network, known as the Local Area Network (LAN) and the voice network, known as the Private Branch Exchange (PBX), as depicted in figure 1-1A. Each network is connected to the outside world via separate interfaces which span multiple protocols, require a multitude of equipment and call for high maintenance expertise. With the deployment of the single packet based network infrastructure, the PBX is integrated into the LAN where the connection to the Public Switched Telephone Network (PSTN) is made through the Gateway, as shown in figure 1-1B. This type of network is a Voice over Internet Protocol (VoIP).

![Figure 1-1: Traditional (A) and VoIP (B) Network Infrastructures](image)

Voice compression [1-1] is key to the success of such VoIP networks. The traditional PBX network operates at a reserved 64 kbits/s whereas the voice bandwidth requirement for VoIP networks is dramatically reduced down to a few kbits/s via advanced speech compression. Algorithmic techniques such as Voice Activity Detection (VAD) can often reduce the bandwidth even further, [1-2].

The main purpose of a VAD algorithm is to detect the presence or absence of speech and prevent the unnecessary encoding/decoding of silent sound. This can be a difficult task due to the immense spectral range of speech sounds while trying to isolate background noise [1-3 1-4]. A VAD algorithm can not just rely upon simple level detection as speech can often be buried below background noise. The decision on whether to activate the speech codec is made
upon certain characteristics of the input speech against a time-varying threshold value. This threshold value is determined during silence periods.

Some VAD algorithms also come with the ability to transmit idle characteristics of the background noise, often called the Comfort Noise Generator (CNG), [1-3]. This information can then be used to reconstruct artificial background noise giving the impression of a constant transmission stream to the listener. The CNG uses information from the VAD algorithm to calculate the parameters needed to synthesise the artificial noise which are then encoded into a Silence Insertion Description (SID) frame, [1-3], and transmitted. Transmission of this frame has very low bitstream, as low as 4 bytes which is significantly lower than transmitted full speech frames. The SID frame is not transmitted in a periodic fashion. VAD algorithms determine whether to send a SID frame or not. This gives the ability to change the background noise during silent periods. This is achieved by comparing the current SID frame with previous transmitted frames. If the spectrum of the noise has changed, then the current SID frame is transmitted.

Figure 1-2 shows the output frames from a typical speech codec with a VAD algorithm and CNG support [1-5]. The addition of the algorithms reduces the overall transmitted data during silent periods by not transmitting encoded non-voice frames.

The International Telecommunications Union (ITU) has produced a number of recommendations which define how telecommunication networks operate and interlink. These recommendations are not compulsory but are universally adopted to ensure interconnectivity between world wide telephone networks. A few of these standards are summarised in table
Introduction

1-1 and are part of the H.323 protocol family, [1-6 1-7 1-8 1-9 1-10] (Note: this list is not exhaustive and only contains a few of the ITU Recommendations).

Table 1-1: ITU Standards for Voice Compression

<table>
<thead>
<tr>
<th>ITU Specification</th>
<th>Transmission Rate (kb/s)</th>
<th>Computation Complexity (MIPS)</th>
<th>Usage</th>
<th>Mean Opinion Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>G.711</td>
<td>56/64</td>
<td>1</td>
<td>ISDN</td>
<td>4.1</td>
</tr>
<tr>
<td>G.723.1</td>
<td>5.3/6.3</td>
<td>18</td>
<td>Near Toll Quality</td>
<td>3.9/3.65</td>
</tr>
<tr>
<td>G.728</td>
<td>16</td>
<td>30</td>
<td>Quality Close to G.7.11</td>
<td>3.65</td>
</tr>
<tr>
<td>G.729/G.729A</td>
<td>8</td>
<td>20/11</td>
<td>Near Toll Quality</td>
<td>3.92/3.7</td>
</tr>
</tbody>
</table>

1.2 Voice over Internet Protocol (VoIP)

VoIP based commercial services, such as "free" telephone calls over the internet, are becoming common place. These services include 'Voice over Internet', 'Telephone IP' and 'Voice over IP'. All these products rely on the same technology; the advanced compression of speech and the transport of the bitstream over Internet Protocol (IP) based networks. VoIP is not a new concept and has been around since the early 1970s when experiments were being carried out for military applications [1-11]. However, the complex processing needed for real-time applications called for costly hardware platforms and certainly did not exist in the consumer market. It was not until the mid-1990s that the first commercial VoIP software hit the market, produced by VocalTec [1-12]. This was enabled by the development of processing power in the form of System-on-Chip (SoC) based computing platforms, data network architecture and speech processing enhancements enabling it to now be affordable for the end user.

One of the main advantages of an IP network is that it has no understanding of the data being transmitted. An IP network is a transport technology for moving partial data from point-to-point. The protocol operates on the network layer of the Open Systems Interconnect (OSI) [1-13] reference model, depicted in figure 1-3.
There are seven layers to the OSI model. The upper three layers (Application, Presentation and Session Layers) are primarily used to interact with users for applications running over a network. The four lower layers (Physical, Link, Network and Transport Layers) are intended to be used for formatting, encoding and transmitting the data over the network. IP networks operate on the first three of the lower layers. The transport layer is only concerned with end-to-end management, checking if packets have arrived and performing error checking.

Figure 1-4 shows a typical end point for voice communications. This is considered as the three upper most layers with the Network Interface containing the lower four layers.
The basic functionality of the audio VoIP client is to digitally sample the speech signals obtained from a handset or other audio device connected to the system. The digital signal can then be encoded using a codec and framed into IP Packets. The packets are then transmitted over the network to another client on the network. Audio is received in the same manner as sending with addition of a Buffer to store incoming packets. During transmission packets may be lost or wrongly ordered. This is due to the way packets travel across the network; there is no fixed path for them to travel so it is possible for each packet to take a different path as depicted in figure 1-5.

In this diagram, the originating device sends two packets across the network and each one takes different routes. The first packet travels along a route which is heavily congested (shown by the dotted line) while the second packet travels along a non-congested route. In this case, the second packet arrives before the first. Generally, the routers would alter the direction that packets take to free up the congested parts of the network. In a worse case scenario, the route the first packet takes may go down during transmission resulting in the packet being lost. The buffer can re-order the packets so the decoder can decode the packets in the correct order. The buffer introduces a delay in the speech signals as the packets are essentially placed in a queue before decoding takes place. Transmitting and receiving speech packets happens simultaneously.

VoIP based communications between clients enables a wide range of multimedia possibilities, however there is still a need for users to connect to the outside world, using the PSTN. This introduces the need for a network component which incorporates the functions of an IP
Client, a conventional telephone which can bridge the two networks together, and is called a Gateway, [1-11].

![Gateway](image)

**Figure 1-6: IP Network to PSTN Gateway**

As with any communication technology, there needs to be well defined standards for VoIP technology to work. Early implementations of VoIP solutions were privately owned and were difficult to interface with. The ITU-Telecommunications (ITU-T) provided a good starting point for an industry wide standard for multimedia conferencing and later produced the H.323 recommendation which specified the framework for an IP network. Figure 1-7 shows the architecture of the H.323 protocol.

<table>
<thead>
<tr>
<th>Conference Management</th>
<th>Media Agents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conference Set-up and discovery</td>
<td>Data Conference and Application Sharing</td>
</tr>
<tr>
<td>H.255 RAS</td>
<td>H.255 Call Signalling</td>
</tr>
<tr>
<td>RSVP</td>
<td>T.120</td>
</tr>
<tr>
<td>H.263</td>
<td>G.711</td>
</tr>
<tr>
<td>G.723.1</td>
<td>H.261</td>
</tr>
<tr>
<td>G.729A</td>
<td>H.263</td>
</tr>
<tr>
<td>RTP/RTCP</td>
<td>UDP</td>
</tr>
<tr>
<td>IP and IP Multicast</td>
<td>Integrated Services Forwarded</td>
</tr>
</tbody>
</table>

**Figure 1-7: Protocol Architecture for H.323**

The H.323 standard incorporates the following ITU recommendations:
- H.255 – Registration, admission and status (RAS) and Call signalling.
- T.12x Series – Multipoint Graphics Control
- H.26x Series – Video Codecs
- G.7xx Series – Audio Codecs
- RTP – Real Time Transport Protocol
- RTPCP – RTP Control Protocol
• RSVP – Resource Reservation Protocol
• TCP – Transmission Control Protocol
• UDP – User Datagram Protocol

Additional services which are also offered by the H.323 protocol and not shown in Figure 1-7 include:

• H.245 – Control Signalling
• H.235 – Privacy and Encryption
• H.450 – Supplementary Services (such as PSTN services like call forwarding, call holding and call park)

Other standards have also been created, such as the Session Initiation Protocol (SIP) produced by the Internet Engineering Task Force (IETF) [1-14 1-15] with an “internet” perspective. The H.323, developed by the ITU, reflects the heritage of PSTN by using binary encoding and reusing parts of the Integrated Services Digital Network (ISDN) signalling [1-16] where as the SIP is a text-based protocol, like the Hypertext Transfer Protocol (HTTP) and Simple Mail Transfer Protocol (SMTP). A text based system is less complex and can be easily implemented on networks. The main advantage the H.323 protocol has over the SIP is that message sizes are smaller. This comes at the price of increased complexity.

The H.323 protocol is currently the most used in telecommunication systems. It has been widely adopted as an alternative to PSTN to handle simple telephone calls and video conferencing. The SIP protocol design is still in the early days and it is simpler compared to the H.323 but currently does not support all the features that the H.323 has, such as video conferencing.

1.3 Speech Compression

Speech coding algorithms produce a compact digital representation of speech sounds suitable for digital processing, storage and transmission. When the digital signal is reconstructed, the resulting speech should be sufficiently similar to the original sound for the purposes of the application. Robust, low bit-rate speech coders are key in meeting current communication demands [1-17 1-18]. Substantial progress has been made in recent years and modern speech coders produce high-quality speech sounds at low-bit rates (currently around 8 kbits/s).

Even in inexpensive optical fibre networks, there is a growing need for bandwidth conservation in wired communications, especially with the explosion in bandwidth of the
Introduction

Internet. Therefore, speech compression is vital for good performance in VoIP networks where the available voice bandwidth is often significantly reduced due to transmission constraints.

1.3.1 Speech Production System

The human vocal apparatus consists of a pair of lungs, trachea (wind pipe), larynx (two folds of skin called vocal cords which vibrate when air is forced through) and the oral tract nasal passage. These components cooperate to produce different sounds by making the air vibrate at different frequencies. The typical spectral range of speech is from 30 Hz up to 8 kHz, [1-13].

Speech sounds are produced by exciting the vocal tract which is effectively an acoustical tube. It is terminated by the lips at one end and by the constriction of the vocal cords at the other. The frequency response of the tube shows the resonant peaks (called formants) corresponding to different multiples of the acoustic quarter wavelength. The cross-sectional area of the vocal tract is controlled by the articulators (lips, jaw, tongue and the velum). To produce sound, air is drawn into the lungs by enlarging the chest cavity. The vocal cords which form a constriction to the air flow are forced to oscillate producing quasi-periodic pulses of air. Depending upon the position of the velum, the vocal tract can be coupled with the nasal cavity. This nasal coupling can substantially influence the character of a sound radiated from the mouth. Articulators are used to change the acoustical characteristics of the vocal tract. The spectrum of the quasi-periodic excitation can be changed and the final various sounds are produced. The sounds that are produced are called "voiced" sounds. The rate of the vocal cord vibration is known as the "pitch" frequency. "Unvoiced" sounds are produced by the turbulent flow of air through constricted spaces in the vocal tract.
1.1 Figure 1-9: Spectral representations of Phonemes

Figure 1-9 shows a range of sound wave plots. Each plot shows different characteristics of some sounds. Looking at the plot /k/, there is random noise before settling to a regular period. The random noise part of the plot is termed unvoiced where the regular period is termed voiced. The sounds /p/, /t/ and /k/ are called plosive sounds as they are formed by abruptly releasing air pressure which has been built behind a closure in the vocal tract; the sounds /s/ and /sh/ are unvoiced sounds. The energy of voiced sounds is generally higher than that of unvoiced sounds. 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 [1-19].

1.3.2 Speech Coding History

Homer Dudley [1-20 1-21] developed and demonstrated the first analysis-synthesis method for speech coding while working for Bell Telephone Laboratories. Over 50 years ago, there was a need for the development of systems that could transmit speech over low-bandwidth telegraph cables. Dudley developed a system that analysed speech in terms of its pitch and spectrum. The speech signal was synthesized by exciting a bank of 10 analogue band-pass filters, representing the vocal tract (the voiced sounds). The unvoiced sounds were represented by aperiodic or random excitation. Dudley’s method, called the Channel Vocoder, is described in more detail in the next chapter.

During the Second World War, a large body of research was carried out on Channel Vcoders to investigate the potential for transmitting encrypted speech. During the 1950s and 1960s improvements were made in the performance of channel vocoders [1-22]. Formant coders [1-23] and pattern matching [1-24] were also developed. Formant coders track the movement of the formants by using the resonant characteristics of the filter banks, while pattern matching coders find the best match between the short time spectrum of speech and a set of stored
frequency response patterns. The speech is then reproduced by exciting the channel filter associated with the selected pattern.

Early vocoders used analogue speech representation, but with the introduction of digital technology much of the research interest transferred to digital representations because of their potential for encryption, ease of transmission and storage. Work had already been done on Pulse Code Modulation (PCM) in the early 1940s and was made easier with the flexibility of digital speech. This method is used for discrete-time, discrete approximation of analogue waveforms.

Itakura and Saito [1-25] and Atal and Schroeder [1-26] were among the first to apply Linear Prediction (LP) techniques to speech with the use of digital computers. LP analysis involves predicting the present speech sample using the linear combination of previous samples. Atal and Schroeder proposed a new linear algorithm based on the LP algorithm with vector excitation which they called “Code Excited Linear Prediction” (CELP) [1-27], an approach still used in modern speech coders and discussed in the next chapter.

1.4 Embedded Processors

Embedded processors are primarily designed for specific tasks in constrained environments, unlike general-purpose PC platforms which handle a range of applications with different processing requirements; such platforms consist of a hardware component and associated software. Problems arise when a task executing on an embedded system needs changing or upgrading. This often leads to redesigning or scrapping the hardware platform. Embedded processors tend to focus on size, power consumption and price. Embedded systems typically run at much lower frequency compared to desktops as they are not needed to perform the range of tasks desktops perform, reducing the power and price. Taking a Personal Digital Assistant (PDA) as an example, it does not have a floating point co-processor (which reduces the cost) as software emulation is sufficient. The majority of applications running on a PDA are not computational intensive but are interactive and display orientated so can run at a slower clock speed. By keeping the device small and having limited functionality, the fabrication cost is reduced as well as increasing battery life.

Another embedded solution which focuses on performance, size, power consumption and price is to use embedded Digital Signal Processors (DSP). DSP’s have specialised functional units, optimised memory and addressing which allows them to perform repetitive calculations
Introduction

in a pipelined fashion, usually out performing general purpose processors running at higher speeds.

Embedded processors are being developed for high performance applications, such as network, video and audio processing. These are highly specialised processors optimised for size and speed, in order to perform real-time data manipulation. They are complex, expensive to design, cheap to produce and have a number of embedded purpose built-in devices. Because embedded processors are designed for specific tasks they have very focused functionality and consume a low amount of power.

Embedded System architects perform Hardware/Software partitioning, a technique that decides which part of an application to execute on an embedded processor or to implement on custom hardware such as a coprocessor or some other streaming processing unit. Such optimised systems achieve significant improvement in performance compared to software-only solutions [1-28] and can even reduce the energy consumption. The authors report in [1-29] that they have improved performance and reduced the power consumption by moving critical regions of the software from the microprocessor into hardware. Hardware/Software partitioning has been made attractive by companies producing single-chip platforms incorporating microprocessors and Field Programmable Gate Arrays (FPGAs) [1-30 1-31].

1.5 Configurable Extensible Processors

Since the industry trend is towards faster, cheaper, and more capable integrated solutions, (Central Processing Unit) CPU designers have added additional resources to the embedded processor to enable it to perform more relevant work per clock cycle. However, implementing in Register Transfer Level (RTL) and validating logic blocks takes time. Such blocks are rigid making it difficult to accommodate new features and change requirements.

To tackle this problem, Configurable Processors [1-32] have been proposed as very potent processing platforms in order to meet the real-time performance requirements of the embedded application. They consist of a base-case processor architecture with the ability to adapt that processor to specific applications by implementing software functions in hardware. This includes the addition of separate execution units (which have been designed to meet the system requirements), registers and buses.

Extensible Processors are a superset of configurable processors [1-33]. They allow the system designer to add instructions to the base Instruction Set Architecture (ISA) that have not been
Introduction

previously thought of in the original design. These highly customised instructions can be targeted to specific applications thus achieving significant performance improvements. The current technique for producing customised instructions is labour-intensive for design engineers as they have to manually select parts of the program which would justify the addition of custom instructions (though a number of industries and academic groups are working towards automating this process). The engineer would then add the instructions into the processor simulator and implement them in hardware. This is clearly a daunting task for large applications in the embedded domain. Tensilica’s XTensa 32-bit processor enables ASIC design engineers to extend its original instruction set by allowing them to add custom instructions and associated execution units in order to accelerate the targeted application [1-33]. ARC International provides a family of cores which can be configured to suit the SoC designers needs to build highly competitive products. The speed of their processors can be further increased by adding ARC’s 128-bit SIMD sound subsystem which adds a further 60 audio specific instructions and shares the same memory as the main system CPU across the system bus interface [1-34]. Other vendors include Intel’s MMX/SEE who have extended their instruction set to use SIMD architecture [1-35], ARM’s Neon 64/128 bit SIMD instruction set [1-36] and IBM’s Cell which consists of their 54-bit Power Architecture core and eight specialised coprocessors based on a SIMD architecture [1-37].

Although designing configurable extensible processors is labour intensive, they offer efficient flexible options while keeping the design turn-around shorter compared to hardwired solutions. This is partly down to the use of pre-verified Intellectual Property (IP) blocks.

1.6 Configurable vs. Reconfigurable Hardware

Reconfigurable hardware, such as FPGAs allow for the dynamic matching of the hardware platform and the software application. Reconfigurable computing has been around since the early 1960s when Gerald Estrin presented a paper featuring a standard processor with an array of reconfigurable hardware logic blocks [1-38]. This work formed the basis of current reconfigurable hardware implemented as FPGA.
A typical FPGA device is depicted in figure 1-10 and consists of the following:

- A number of logic blocks consisting of AND, OR, NOT and EXCLUSIVE OR gates and a flip-flop - usually implemented as Look-Up-Tables.
- A hierarchical matrix of programmable interconnects
- General purpose memory block
- Input/Output Ports
- Configuration memory

The designer can arrange these elements for the application specification through configuring the structure of an FPGA during system boot time.

The logic block gates and other combinatorial functionality are not actually implemented as traditional, dedicated gates but are represented by look-up tables which reproduce their functionality. Each implementation of a gate will have a small amount of dedicated memory inside the logic block storing the truth table of the gate i.e. table 1-2 depicts the lookup table for a simple two-port exclusive OR (XOR) gate.

Table 1-2: Exclusive OR Truth Table

<table>
<thead>
<tr>
<th>Exclusive OR (XOR)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>Q</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
The structure of this XOR gate only has two ports, this can be extended to any number of input ports. In most applications, logic blocks have around five input ports and two independent output ports. The advantage of this is that the logic block can be configured to implement a range of logical functions and when combined can create more complex combinational systems such as encoders/decoders and mathematical functions.

![Interconnect Switch](image)

**Figure 1-11: FPGA Interconnect Switch**

A key part of reconfigurable systems are the programmable interconnects and switches which connect the logic blocks together. Input and output pins on the logic blocks are connected to wiring segments. The wiring segments generally span a logic block width before being terminated by a programmable switch box (as seen in figure 1-10). Wherever a horizontal interconnect wire crosses a vertical interconnect wire there will be a switch box. This contains a number of switches and can control the direction of the data flow on the wires. Figure 1-11 shows a simple 3 wide interconnect wire. In this architecture, only wire 1 in the horizontal plane is connected to wire 1 in the vertical plane. In more complex FPGAs, wire 1 in the horizontal plane could also be connected to wires 2 and 3 in the vertical plane. Figure 1-11 also shows a detailed view of a switch box. Within the box there are 6 different switches covering the different pathways data can travel e.g. if data is travelling from ‘W’ to ‘S’, switch 4 would be activated. Similarly, if is data travelling from ‘W’ to ‘E’, switch 3 would be activated.

The logic blocks and the routing of the switch boxes are controlled by the configuration memory. The configuration bitstream is produced via established Electronic Design Automation (EDA) flows, usually starting with the compiler from a schematic design or a hardware description of the design in a Hardware Description Language (HDL).
Early FPGAs were limited in use due to their restricted capacity. They were also more expensive than Application Specific Integrated Circuits (ASIC) for mass production. As costs have dropped and computation for finding routes has improved, FPGAs have become a cheap alternative.

The industry has matured and embedded CPU cores on high-capacity FPGAs are now commonplace [1-39], [1-40]. However, certain applications can suffer from poor design and offer reduced flexibility and exploitation of logic blocks. This can lead to long interconnects, increased area due to the amount of routing required and additional consumption in power resulting in poor performance compared to fabricated embedded CPU cores.

The terms reconfigurable and configurable hardware are commonly used interchangeably. In this work, reconfigurable hardware refers to the platform that can alter its functionality in the field. Some FPGAs have the ability to perform partial reconfiguration during run-time. This makes them highly reconfigurable and suitable for a range of applications.

Configurable hardware refers to one-off designs, typically compile-time configurable that once committed can not be changed. ASIC are often classed as configurable hardware due to the nature in which they are designed.

Conserving power is becoming a significant concern within the embedded market. Embedded system constraints require them to last longer and consume less power. ASIC designers often embed power saving techniques into their hardware to make their application power efficient. Reconfigurable hardware has a shorter time to market compared to configurable hardware. However, they consume more power, are generally slower and have larger area due to the routing needed compared to configurable hardware.

1.7 Contributions

The work carried out was to research novel embedded CPU architecture for accelerating speech coding algorithms in VoIP networks. This research was funded by the Engineering and Physical Sciences Research Council (EPSRC) - GR/S44976/01. The author of this thesis contributed to this project in many different ways which are outlined in this section.

The first contribution to this work was in the profiling of the two speech codecs. This is presented in section 3.5 of this thesis. This enabled the project group to identify where the speech codecs spent most of their time executing code. On studying the speech codecs in
further detail the code appears to be executing DSP type functions in data-parallel loops. With the addition of a vector processor there is potential for a vast amount of improvement with regards to instructions actually executed.

The next part of the project was to identify and define the custom instruction architecture of the vector coprocessor. This is outlined in chapter 4.5. After the data-parallel instructions were identified, they were introduced into the speech codecs reference codes, provided by the ITU. Section 4.5 describes how this was achieved.

The custom instructions were also implemented on the SimpleScalar toolset – chapter 4.2. The toolset enabled the group to explore different architectural options of the coprocessor such as vector register width. A range of different bitstreams, provided by the ITU, were examined by the research group who were able to explore the coprocessor’s effect on the dynamic instruction count (the number of instructions executed) while maintaining bit stream equivalency. These results are presented in section 4.7. SimpleScalar also provided results in terms of improvements to particular aspects of the codecs to identify enhancement could be made and where the coprocessor could be used more effectively. Section 4.8 and appendixes B and C illustrate this.

Another aspect of this project dealt with modelling parts of the coprocessor using SystemC [1-41]. SystemC is a hardware-oriented language based on C++. It enables fast modelling of SoC platforms and silicon IP, all the way to system level modules. Within a SystemC model the behavioural descriptions of the custom vector instructions introduced in the speech codecs were instantiated. The SystemC datapath was then synthesised to RTL Very High Speed Integrated Circuit Hardware Description Language (VHDL) using a commercial SystemC synthesiser. There are a very few cases of research projects for the design of custom Single-Instruction-Multiple-Data (SIMD) instructions using SystemC. The authors in [1-42] present work which focused on the functional simulation level and developed a SystemC-based language, called ArchC, for describing CPU architectures. Using SystemC for CPU design is also discussed in [1-43]. In that paper, the authors describe a methodology for CPU design and optimisation using SystemC transaction-level modelling. In this technique, the models can be refined to meet performance requirements. In [1-44], the authors have integrated SIMD extensions written in SystemC through a co-processor interface to a CPU simulator. The instructions have been developed gradually, improving the performance of the application by using system-level simulation. Work carried out during this research is similar
to that presented in the latter paper with the important contribution of providing a route to silicon implementation of the SIMD extensions.

The microarchitecture of the system is explained in section 5. This includes the design of the vector processor and the main Reduced Instruction Set Computer (RISC) CPU.

The final part which the author contributed to the research project was the full design and implementation of a high bandwidth Load/Store Unit in section 5.6 using RTL. This was designed in a two stage serial cache design where the TAGs were accessed in one clock cycle and data in the following. This was attached to the vector datapath of the coprocessor and the AMBA Bus of the Leon 3 system. The Load/Store Unit (LSU) was then examined to determine the number of cycles per instruction based on the custom Load/Store instructions implemented in the two speech codecs workloads. Different cache configurations and vector lengths were explored to decide on the optimum design. Finally, the statistical power analysis was performed for the LSU over different configurations including different vector size, cache sizes over different clock frequencies in order to determine optimum size, associated with the cycle results, while not consuming too much power and area.

1.8 References


[1-34] ARC International www.arc.com
Introduction

Multiprocessor system-on-chip platforms,” IEE Proceedings on Computers and Digital Techniques, vol. 152, no. 1, pp.3-11, 2005


2 SPEECH CODING

2.1 VolP and Speech Coding

In recent years, speech coding has drawn a lot of attention due to the explosion of VolP networks. Although the technology has been around since the 1970s, VolP has gained interest due to the high speed internet connections that are currently available, especially in residential homes. Early questions were asked whether the public switched networks could actually compete with the cost of internet telephones in the future [2-1]. Companies are starting to invest in VolP technologies because they have lower costs for the service provider which can then be passed onto the consumer. VolP is also attractive as the infrastructure needed is already in place since it uses the same equipment that is currently driving the internet. The author in [2-2] made a comparison between VolP networks with the traditional PSTN. His study focused on Quality of Service (QoS) that each network provides. The drawback of VolP is the risk of data packets being lost over the network during transmission, especially on a heavily congested network. Lost packets reduce the quality of the reconstructed speech. PSTNs have a clear advantage over VolP networks in this respect as data is not as easily lost and therefore the quality of speech is better. The author in [2-2] asks the question “Is the internet ready for the amount of traffic that will be unleashed with the introduction of VolP in residential homes?”

The crucial part of any VolP network is the voice coder, often referred to as a vocoder or speech codec. The purpose of the vocoder is to transform the sampled analogue speech into a compressed digital signal bitstream for transmission. There are many issues to take into account when choosing the best vocoder [2-3]. Different vocoders offer different qualities of the reconstructed speech and have different bandwidth requirements. The vocoders with high bandwidth requirements often offer better voice quality compared to those with lower bandwidth. Bit-rates of today’s speech coders range from 1.2 to 64 kbps; the most efficient codec available operates as low as 5 kbps and still gives “toll quality” or better. Toll quality is defined as having a high Mean Opinion Score (MOS) [2-4]. Users are asked to rate the connection they have just used with 1 being Poor and 5 being excellent. Anything above an average MOS of 4 is considered toll quality. As more sophisticated vocoders are designed and implemented, the bit-rate requirement will decrease further allowing more samples to be efficiently coded while ensuring the reconstructed speech quality remains the same.
The ITU codecs that are commonly used in VoIP networks are summarised in table I-I. The current trend in industry is to develop speech coders which have a lower bit rate. There is a clear bit-rate difference in the ITU’s G.711 and the G.723.1 recommendation, with almost a one-twelfth saving in bandwidth. Is this actually a saving as quality often degrades with lower bit-rates? This is reflected in the MOS. Bringing this together with previously mentioned problems how quality can degrade with heavily used VoIP networks; can the low bit-rate codecs reproduce comprehensible speech on heavily congested networks? The authors in [2-4] examine if too much time is being spent on creating low bit-rate codecs to conserve bandwidth instead of addressing the quality of vocoders.

2.2 Previous Speech Coding Research

This section presents previous research into speech coding ranging from hardware implementations to algorithm development. The authors in [2-5] have designed and implemented an ASIC architecture for speech synthesis at 1.6 kbps. Their proposed architecture is similar to speech codecs implementing codebook searches. The architecture requires less hardware resources than other speech codecs. However, due to the low bit-rate that is offered, quality is poor and the performance does not compare well to the CELP and Mixed Excitation Linear Predictive (MELP) speech codecs cousins.

Examples of algorithm development for low bit-rate speech codecs are presented [2-6] and [2-7] where the authors report on an enhanced waveform interpolative speech coder at 4 kbps and 2.8 kbps respectively. They compare their speech coder with the G.723.1 recommendation and find that the performance is better than the G.723.1 standard. The waveform interpolative speech coder has been researched further in [2-8] demonstrating that this technique is a promising candidate in future multimedia communication applications.

It was during the 1990s that the ITU released their low bit-rate speech coders for multimedia communication. These speech coders drew a lot of attention in the research community for their potential to be the coders for internet communications as reported in [2-9]. The speech coders were further researched in [2-10] which compared the G.729, G.723.1 and G.729 Annex A for a range of applications. Early research started to improve the speech coding standards by reducing the complexity of the encoder [2-11], optimising the codecs and suggesting new bit-rates that they can work at [2-12]. Authors have tried different methods to improve the codebook search to make it faster [2-13], have attempted to improve the quality of the reconstructed speech by modifying the codecs [2-14] and have developed a new speech coder based on the ITU-T standards [2-15].
The ITU-T speech codecs have also been used in a number of research projects as example algorithms due to their popularity. Examples of this can be found in research projects like [2-16] and [2-17] which addressed conversion algorithms between speech codecs, a process known as transcoding. The ITU-T speech coders have also been used in research projects involving off-the-shelf DSP chips [2-18 2-19 2-20 2-21]. These involve implementing the codecs in hardware using a range of commercial DSP chips for real-time applications and optimising the code in order to map the codecs to the microarchitecture of the processors. More recent work has focused on designing targeted processors to execute the speech codecs in hardware. The authors in [2-22] examine the G.729 and find that the main drawback of the codec is during the autocorrelation computation. They then specify a hardware design for this part with the remainder of the speech processing being implemented on a DSP.

More applicable background research to this project is the work performed by the authors in [2-23] and [2-24]. The first paper discusses hardware accelerators and custom instructions for accelerating the G.723.1 and G.729 speech coders. The second paper reports on the improvement of the performance of the G.723.1 codebook search with segmentation addressing. The authors then evaluate a DSP with respect to speech coding and their previous research [2-25]. The Blackfin DSP uses a 32-bit architecture which differs from traditional 16-bit DSPs. This allows for two operations to be executed in the same cycle including 32-bits loads and stores from memory. This is effectively a style of long-instruction word processing. The authors also report on the speed-up in performance the processor has made.

2.3 Speech Codecs

There are many different types of speech codecs which encode digital speech with a range of different bit-rates. The selection of the type of speech codec is dependent on the type of target application ranging from landline telephones to satellite telephony [2-26]. This section discusses the history of speech codecs and then presents the different types of speech codecs that are currently used today.

2.3.1 The Channel Vocoder

The Spectrum Channel Vocoder, designed by Dudley, is the oldest method for speech analysis-synthesis [2-28]. An example of the Channel Vocoder is shown in figure 2-1. The system relies on representing the shape of the short-time amplitude spectrum and the vocal tract excitation. The vocal tract envelope is represented by a bank of band-pass filters
(typically between 10 and 20). The spectral representation becomes more accurate as the number of band-pass filters are increased.

At the analyser, the input speech is analysed by a bank of band-pass filters and the measured power in each channel is used to control the inputs to the corresponding synthesiser filters. A decision is made with regards to the vocal excitation as to whether the speech is voiced or unvoiced. If the speech is voiced, then the pitch period is measured and sent with the voiced/unvoiced information to the receiver. The synthesiser also has a bank of band-pass filters connected in parallel. They estimate the discrete power spectrum of the speech signal. The envelope of the power spectrum is controlled with variable attenuators at the input to each filter.

The synthesised speech may be understandable but there is perceptible degradation of the speech quality. There are many factors responsible for this degradation:

- The discrete amplitude spectrum is not an efficient way of preserving all the important spectral details, due to the limitation placed on the achievable resolution of the number, bandwidth and spacing of the band-pass filters.
- The large range of the spectrum may not be represented due to practical limitations.

The structure of the voiced spectrum is represented using pitch-periodic pulse-like waves whose characteristics can be different from those of glottal pulses.
2.3.2 Formant Vocoder

Formant vocoders can be divided into two different categories depending on the synthesiser's structure, namely the 'cascade' and the 'parallel' form. The example shown in figure 2-2 is the parallel form.

As with the Channel Vcoders, the short term amplitude spectrum of the speech is effectively sampled, coded and transmitted together with the vocal excitation. Transmitting such detailed information is unnecessary, as its adjacent values are highly correlated. It is possible to define
the shape of speech by only transmitting the frequencies and the spectral amplitudes of the formants therefore saving bandwidth. Vocoder systems that transmit to the synthesiser only the formant and the vocal excitation data are known as formant vocoders.

Figure 2-2: The Formant Vocoder
In the parallel formant vocoder shown in figure 2-2, the formant characteristics obtained during analysis are used by the synthesiser to control three variable resonant filters which represent the first three speech formants. The filters are adjusted according to the formant characteristics, and their input is excited either by a noise source (for unvoiced speech) or a pulse generator (for voiced speech).

The performance of formant vocoders depends principally on the implementation method adopted at the analysis stage when obtaining the formant and the voiced/unvoiced information. The method for identifying formants is generally regarded to use a bank of band pass filters (as in the earlier channel vocoder) and to pick the frequencies at which the filter output is the greatest. Current vocoders tend to use digital techniques such as Discrete Fourier Transform (DFT) followed by a peak picking procedure, homomorphic filtering or inverse linear filtering.

### 2.3.3 Linear Prediction Coding (LPC) Vocoders

Linear Prediction Coding (LPC) is the most widely researched technique for speech coding. It involves time domain methods and avoids the difficulties in finding formant locations. During frequency domain analysis certain formants seem to diminish in certain sounds or become over emphasised during others.

The LPC vocoder produces speech by using an adaptive $P^{th}$ order linear digital filter (as shown in figure 2-4) and was proposed by Atal. This models the vocal tract characteristics, the radiation characteristics and the pulse shape of the vocal excitation. The transfer function of this model is assumed to be an “all-pole” approximation of the shape of the original speech spectrum and is defined by the equation:

$$H(z) = \frac{1}{1 - \sum_{i=1}^{P} a_i z^{-i}}$$

*Figure 2-3: Transfer function of the LPC Vocoder*

The order, $P$, in this equation is defined as $P=2L$ where $L$ is the number of formants that are needed to characterise the speech amplitude spectrum. The complex roots of the equation provide the formants and the bandwidth of the modelled speech spectrum. The excitation function is a signal constructed from discrete pulses, random noise or a combination of the two. This represents the voiced or unvoiced sounds of the synthesised speech.
LPC vocoders generate an excitation at the receiver only. This is based on the transmitted pitch period and voicing information. Vocoders which provide extra information about the spectral shape of the speech waveform are based on LPC and generally operate between 2.0 and 5.8 kbps.

2.3.4 Analysis-by-Synthesis Codecs

Analysis-by-Synthesis (AbS) codecs are based on LPC vocoders. The difference being that these codecs test a large set of excitations which closely match the speech waveform. The codec then selects the best excitation candidate. AbS codecs are becoming more widely used as they can be implemented easily onto a DSP chip. The first AbS codec was invented by Atal and Remde in 1982 with the Multi-Pulse Excited (MPE) Codec [2-29]. Later, the Regular-Pulse Excited (RPE) [2-30] and the CELP codecs were introduced. The main differences with these codecs are the way that they represent the excitation signals. These will be covered later in this section. The general model of an AbS codec is shown in figure 2-5.
AbS vocoders are good at creating intelligible speech at 2.4 kbps but this is not natural sounding. The traditional waveform coders are capable of producing good quality speech at rates down to 16 kbps but are not usable below that bit rate. To bridge the gap between bit rates and quality, hybrid codecs have been introduced which use the linear prediction filter model of the vocal tract as found in the LPC vocoders. Instead of using the simple two-state model (voiced/unvoiced) to find the required input to the filter, the excitation signal is chosen by attempting to match the reconstructed speech waveform as closely as possible to the original speech waveform.

The input speech is split into frames, typically 20 ms each. The parameters for the synthesis filters and the excitation of the filters are determined for each frame. This is achieved by finding the excitation signal which, when passed to the given synthesis filter, minimises the error between the input speech and the reconstructed speech. Thus, the name "Analysis-by-Synthesis" is given as the encoder analyses the input speech by synthesising many different approximations of it. The encoder transmits information representing the synthesis filter parameters and the excitation to the decoder for each frame. The decoder then takes the information and passes it through the synthesis filter to give the reconstructed speech. The error weighting block is used to shape the spectrum of the error signal in order to reduce the subjective loudness of this error. This is possible because the error signal in frequency regions where the speech has high energy will be partially masked by the speech. The weighting filter
then emphasises the noise in the frequency regions where the speech content is low. The weighted error block is therefore used in frequency regions where the speech signal has high energy in order to minimise the weighted error and mask the error signal by the speech signal. This procedure significantly improves the quality of the reconstructed speech compared to the original AbS codec.

AbS codecs synthesis filters may include a pitch filter to model the long-term periodicities present in the voiced speech. MPE and RPE codecs often work satisfactorily without a pitch filter, but performance improves when it is included. However, for CELP codecs pitch filters are necessary to provide acceptable performance. The main differences with these codecs are the way that they represent the excitation signals.

2.3.4.1 Multi-Pulse Excited (MPE) Codecs

The MPE codec excitation signal is given by a fixed number of non-uniform pulses for every speech frame. The coder encodes the position of these pulses along with their amplitude during the analysis stage which are then transmitted to the decoder. The pulse positions are such so that the weighted mean square error is minimised. There are usually between 4 and 6 pulses for every 5 ms. Good quality speech is reconstructed at a bit rate of around 10kbits/s. MPE codecs are generally more expensive to implement than classical linear predictive vocoders due to the need to calculate the amplitude and the location of pulses. The performance of the MPE codec deteriorates with high-pitch speakers, but this problem can be resolved by adding a pitch prediction filter.

2.3.4.2 Regular-Pulse Excited (RPE) Codecs

The RPE codec differs from the MPE codec by using regular spaced pulses at fixed intervals. The analysis stage only needs to find the first pulse, its amplitude and the amplitude of the second pulse, as the pulses are located at fixed time intervals. The pulse spacing does not need to be coded so no information needs to be transmitted regarding relative pulse position. The pulse spacing factor is around 3 to 4 for the RPE codec, and the location of the first pulse is updated every 5 ms. Typically, there are 10 to 13 pulses per 5 ms (compared to around 4-6 pulses for the MPE codec). This gives the RPE codec slightly better quality speech than the MPE codec; this tends to mean that the RPE codecs are less complex than MPE codecs.

2.3.4.3 Code Excited Linear Prediction (CELP) Codecs

MPE and RPE codecs exhibit acceptable performance at rates of 10 kbps and above. Quality degrades below this due to the information that needs to be transmitted about the pulses' position and amplitude. If the bit-rate is to be reduced, the number of pulses should be
reduced or the amplitude quantized more coarsely. The CELP algorithm overcomes this problem and is the most commonly used speech codec today. The speech coder was proposed by Schroeder and Atal in 1985 [2-31] and the block diagram of the algorithm is shown in figure 2-6.

This method differs from the MPE and RPE codecs in that the excitation signal is effectively vector quantized using a codebook. The codebook index is represented with 10 bits, giving a codebook size of 1024 entries (vectors) with each vector being 40-samples (5ms). The gain is coded with 5 bits therefore the bit rate necessary to transmit the excitation information is greatly reduced. The optimum vector is selected such that the weighted mean squared error is minimised. The codebook in the original CELP algorithm was filled with Gaussian sequences. The problem with the original codebook was that it was very complex and could not be implemented in real-time. Much work has been done to optimise the speech codec including reducing the complexity of the codebook (as seen in section 2.2). With the improvement of DSP chips, these CELP codecs can now be implemented on standard DSPs. Speech coding
standards, including the ITU recommendations G.729A and G.723.1, have been derived from the principles defined by CELP coding.

2.4 The G.729A and G.723.1 Speech Codecs

The G.729A [2-32] and the G.723.1 [2-33] recommendations are the primary Speech Coders studied in this project. This section describes how they work and which category, as discussed in the previous sections, they belong to.

2.4.1 The ITU G729 Speech Coder

During the 1990s, the Comité Consultatif International Téléphonique et Télégraphique (CCITT) committee invited candidates to design a low-delay 8kbits/s speech coding standard with a frame length of 5 ms. It was not until 1992 that two codecs were submitted, one with a length of 13 ms using Conjugate Structure CELP (CS-CELP) and the other 12 ms using Algebraic CELP (ACELP). The committee decided that a frame length of 10 ms would be preferable and both codecs were merged to form a Conjugate Structure Algebraic CELP (CS-ACELP) to provide a toll quality speech at 8 kbits/s with a 10 ms frame length.

2.4.1.1 Encoding Principles

The ITU recommendation G.712 [2-34] generates a digital signal from the analogue input signal by performing telephone bandwidth filtering. The signal is then sampled at 8000 Hz followed by a conversion to a 16-bit linear PCM signal for the input of the coder for the G.729. The output of the decoder should be converted back by similar means.

The encoding schematic is shown in figure 2-7, which is similar to that in figure 2-6. The encoder operates on speech frames of 10 ms (80 samples). The input frames are pre-processed which serves as the input signal for all subsequent calculations. The Linear Predictive Coefficients (LPC) are calculated for every frame and are then converted to Line Spectrum Pairs (LSP) (a derivative of the coefficients and all-zero filter resulting in a set of spectral frequencies) due to their statistical properties they have and help ensure the stability of the filter. These are quantized with 18 bits using two-stage vector quantization. The excitation signal is chosen by using an analysis-by-synthesis search where the error between the original and reconstructed speech is minimised by filtering the error signal with a perceptual weighted filter. The speech frame is split into two 5 ms subframes where the excitation parameters are determined for each subframe. The LPC (both quantized and unquantized) are interpolated and used for the first subframe, while the second subframe uses the original quantized and unquantized LPC. The open-loop pitch delay is estimated for each
10 ms frame based on the weighted speech signal. Closed-loop pitch analysis is done by searching around the open-loop pitch delay to find the adaptive-codebook delay and gain. The pitch delay is encoded using 8-bits in the first subframe and differentially encoded using 5-bits in the second subframe. A parity bit is added to the adaptive-codebook index of the first subframe to detect channel faults so error concealment can be applied at the decoder. To calculate the excitation of the signal, a 17-bit algebraic codebook is used as the fixed codebook. The adaptive and fixed-codebook gains are vector quantized using 7-bits. Finally, the filter memories are updated using the determined excitation signal.

Forward adaptation of the speech signal is used to determine the synthesis filter parameters for every 10 ms frame. The filter parameters are then converted to Line Spectral Frequencies (LSFs) and quantized using a predictive two-stage vector quantization. Each frame is then split into two 5 ms subframes and the excitation for the synthesis filter of each subframe is calculated.

2.4.1.2 Pre-Processing

A simple two stage pre-processing is applied to the input before the encoding process starts. The 16-bit PCM signal is initially scaled by dividing by a factor of 2 to reduce the possibility...
of overflows in the fixed-point implementation of the codec. Next, the signal is high-pass filtered using a second order pole/zero filter with a cut-off frequency of 140 Hz. This will remove any undesired low-frequency content in the input signal. The pre-processed signal acts as the input speech signal to the rest of the encoder.

2.4.1.3 LPC Analysis and Quantization

LPC analysis is used to derive the filter coefficients for the tenth-order synthesis and weighting filters for every 10 ms frame using the autocorrelation method with a 30 ms asymmetric window, shown in figure 2-8 [2-35].

![LPC Analysis Window](image)

**Figure 2-8: LPC Analysis Window for G.729**

The LP analysis window is given by:

\[
\begin{align*}
    w(n) &= \begin{cases} 
    0.54 - 0.46\cos\left(\frac{2\pi n}{399}\right) & n = 0\ldots\ldots199 \\
    \cos\left(\frac{2\pi (n-200)}{159}\right) & n = 200\ldots\ldots239
    \end{cases}
\end{align*}
\]

**Figure 2-9: LP Analysis Window Equation**

The window consists of half a Hamming window for the first 120 samples (15 ms) and a quarter of a cosine function cycle for the final 40 samples (5 ms). The sample indices 0 to 79 in figure 2-8 correspond to the present frame. Previous speech frames are also applied (-120
to 0) included in the window as well as a further 40 samples from future speech frames. This translates to the extra algorithmic delay of 5 ms. The windowed speech is used to compute 11 autocorrelation coefficients which are then modified to add a white-noise correction factor and to avoid problems with low-level input signals. The autocorrelation coefficients are then used to calculate the Linear Prediction (LP) filter coefficients using the Levinson-Durbin algorithm. The LPC are converted to LSF before quantization and interpolation. For the first subframe, the average quantized LSFs from the current and previous frames are interpolated whereas the second subframe only uses the current frame.

2.4.1.4 Quantization of the LSP Coefficients

A 4th order Moving Average (MA) prediction is used to predict the LSF coefficient of the current frame. The difference between the calculated and the predicted coefficients is quantized using a two-stage vector quantizer. The process can be seen in figure 2-10.

The MA predictor is used to predict the set of LSFs of the current frame from the values of the previous quantized LSFs. Both of the MA predictors are invoked simultaneously. The predictor which minimises the LSF prediction error is signalled to the decoder via the 1-bit output and is employed in the prediction process. The first stage of the prediction is to search the 10-dimensional codebook which minimizes the mean squared error. The difference between the codebook entry and the prediction errors are then quantized in the second stage.
The algorithm then tries to match the LSF vector by minimising the weighted squared error. The second stage uses two 32-entry codebooks to quantize the LSFs. The LSFs are split into two where the first 5 are quantized using one codebook, the final 5 LSF using the other. These are then coded into 5 bits for each totalling 18 bits for the quantization process. Once the LSF coefficients have quantized and interpolated, these are converted back to LPC.

2.4.1.5 Perceptual Weighting

The weighting filter is based on the unquantised LP filter coefficients calculated above. The transfer function of the weighting filter is given by:

$$W(z) = \frac{A(\frac{z}{\gamma_1})}{A(\frac{z}{\gamma_2})}$$

$$W(z) = \frac{1 + \sum_{i=0}^{10} \gamma_1 i z^{-i}}{1 + \sum_{i=0}^{10} \gamma_2 i z^{-i}}$$

Figure 2-11: Transfer Function of the Perceptual Weighted Filter

The amount of weighting is controlled by $\gamma_1$ and $\gamma_2$. These variables are based on the spectral shape of the input signal. In order to improve the performance of the codec for signals with low frequency response the weighting has been made adaptive. The variables are updated for every 10 ms frame and the adaptation is interpolated for each first subframe to smooth the process. The spectral shape is obtained as a by-product from the Levinson-Durbin recursion computed earlier and passed through a 2nd order filter to characterise the speech. The control variables $\gamma_1$ and $\gamma_2$ are updated depending on whether the speech is flat or tilted.

2.4.1.6 The Adaptive Codebook

In order to reduce the complexity of the adaptive codebook search, open-loop pitch analysis on the input weighted speech is carried out. This is calculated over a 10 ms frame and attempts to maximise the autocorrelation. These maximum values are found in three different ranges which are then normalised. They are then used to calculate the best open-loop pitch delay.

The open-loop pitch delay is subsequently used in the closed-loop search for the adaptive codebook index. The index search is limited to 6 samples for the first subframe around the open-loop pitch delay and is coded with 8 bits (P1) which takes fractional delay values between $19\frac{1}{3}-84\frac{2}{3}$ and integer delay values between 85-143. The second subframe closed-
loop is concentrated around the delay found in the first subframe and is coded with 5 bits (P2).

The closed-loop pitch search for both subframes is achieved by minimising the mean-squared weighted error between the original and reconstructed speech. This is achieved for the integer pitch search by maximising the term in figure 2-12.

\[
R(k) = \frac{\sum_{n=0}^{39} x(n) y_k(n)}{\sqrt{\sum_{n=0}^{39} y_k(n) y_k(n)}}
\]

Figure 2-12: Closed Loop mean-squared error equation

The term \(x(n)\) is the target signal and \(y_k(n)\) is the past filtered excitation at a delay \(k\). For the reduced complexity version of the G.729, i.e the G.729A, only the numerator in the equation found in figure 2-12 is evaluated.

For the fractional part of the pitch search, the interpolated values of \(R(k)\) are used to search for the maximum. This is done by using a Finite Impulse Response (FIR) filter based upon a Hamming windowed sinc(x) function.

Once the pitch delay has been calculated for both subframes, the output of the adaptive codebook vector is calculated for both the encoder and decoder. This is done by using the past excitation signal for integer delays or the interpolated past excitation signal for fraction delays. A Hamming windowed sinc(x) function is used for the interpolation.

The coder is made more robust against random bit errors by introducing a parity bit, P0. This is computed against the delay index in P1. By using an XOR operation on the 6 Most Significant Bits (MSB) in P1 the parity bit can be calculated. At the decoder, the parity can be recalculated, by using the same XOR function, on the 6 MSB in P1. If the parity bits do not match then error concealment can be applied.

The final part of the adaptive-codebook delay search is to calculate the adaptive-codebook gain, \(g_p\).
2.4.1.7 The Fixed Algebraic Codebook

The G.729 uses a 17-bit fixed codebook, based on an algebraic codebook structure. The closed-loop search of these codebooks can be complex due to their large size, and the delay imposed on this search would deliver poor results in real-time applications. However, in this codebook, each subframe contains four non-zero pulses. Each pulse can have either the amplitude $+1$ or $-1$ and have the positions defined in table 2-1.

<table>
<thead>
<tr>
<th>Pulse</th>
<th>Sign</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i_0$</td>
<td>$s_0: \pm 1$</td>
<td>$m_0: 0, 5, 10, 15, 20, 25, 30, 35$</td>
</tr>
<tr>
<td>$i_1$</td>
<td>$s_1: \pm 1$</td>
<td>$m_1: 1, 6, 11, 16, 21, 26, 31, 36$</td>
</tr>
<tr>
<td>$i_2$</td>
<td>$s_2: \pm 1$</td>
<td>$m_2: 2, 7, 12, 17, 22, 27, 32, 37$</td>
</tr>
<tr>
<td>$i_3$</td>
<td>$s_3: \pm 1$</td>
<td>$m_3: 3, 8, 13, 18, 23, 28, 33, 38$</td>
</tr>
</tbody>
</table>

The codebook vector, $c(n)$, is constructed by taking the four non-zero unit pulses at their locations and multiplying them with their corresponding sign:

$$c(n) = s_0 \delta(n - m_0) + s_1 \delta(n - m_1) + s_2 \delta(n - m_2) + s_3 \delta(n - m_3)$$

where $\delta(0)$ is a unit pulse. The codebook also includes an adaptive pre-filter which enhances the harmonic components in the reconstructed speech and improves the performance of the speech. The equation in figure 2-13 is modified to the following if the pitch delay is lower than 40:

$$c(n) = \begin{cases} 
  c(n) & n=0,...,39 \\
  c(n) + \beta(n-T) & n=T,...,39 
\end{cases}$$

The fixed codebook is searched by minimizing the mean-squared error between the perceptual weighted speech and the weighted reconstructed speech. The G.729 uses a series of four nested loops to search for the pulse positions. A focused search is also used which computes a
threshold from the first three pulses. If this threshold is exceeded then the final loop is not entered. The final loop is more complex, having 16 possible positions for the fourth pulse. The threshold is pre-computed for each subframe before the codebook is searched; this procedure reduces the amount of time that the codebook is searched. In the G.729A, the search for the pulse positions is different. Instead of a nested-loop search, an iterative depth-first, tree search is used. This method reduces the number of pulse position combinations that are tested. This method also has a fixed complexity compared to the G.729 because of the threshold condition on the forth pulse.

The codebook contribution is represented by 17-bits. Each pulse has a fixed size (+1 or -1) and is encoded with 1-bit, \(s=1\) if the sign is positive and \(s=0\) if negative. The first three pulses have eight possible positions and are encoded with 3-bits. The final pulse has 16 possible positions and is encoded with 4-bits which makes a total of 17-bits to represent the fixed-codebook index signal.

The next stage in the process is to quantize the two codebook gains. This is done by using a predictive, two stage conjugate structured vector quantizer. A fourth-order moving-average prediction is used to calculate an intermediate fixed codebook gain. This is based upon the energies of the previous gain fixed codebook signal. A correction factor is then applied to calculate the optimum gain. The adaptive codebook gain and the correction factor are quantized using a two-stage conjugate structured codebook. The first stage consists of a 3-bit two-dimensional codebook while the second stage is a 4-bit two-dimensional codebook. This gives the adaptive codebook index as \(GAI\) for the first stage and \(GBI\) for the second; similarly for the fixed codebook, \(GA2\) for the first stage and \(GB2\) for the second.

The last stage of the encoder is to save the current state of the filters in memory. This is needed as the next computed target signal is determined from the previous excitation signals.

2.4.1.8 Bit Allocation

The transmitted parameters are listed in table 2-2 and are the same in both the G.729 and G.729A codecs.
Table 2-2: Bit Allocation of the 8 kbits G.729 and G.729A codecs

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Codeword</th>
<th>Subframe 1</th>
<th>Subframe 2</th>
<th>Total per frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line Spectrum Pairs</td>
<td>L0, L1, L2, L3</td>
<td>18</td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>Adaptive-codebook delay</td>
<td>P1, P2</td>
<td>8</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>Pitch-delay parity</td>
<td>P0</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Fixed-codebook index</td>
<td>C1, C2</td>
<td>13</td>
<td>13</td>
<td>26</td>
</tr>
<tr>
<td>Fixed-codebook sign</td>
<td>S1, S2</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Codebook gains (stage 1)</td>
<td>GA1, GA2</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Codebook gains (stage 2)</td>
<td>GB1, GB2</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td>80</td>
</tr>
</tbody>
</table>

2.4.1.9 Decoder Principles

The decoding principle of the G.279 is shown in figure 2-15. The next few sections discuss the decoding process.

![Decoder Schematic](image)

Figure 2-15: G.729 Decoder Schematic

The first step in the decoding process is to decode the transmitted parameters listed in table 2-2 and then extract the LP coefficients, adaptive-codebook vector, fixed-codebook vector and gains from the transmitted data. The digital speech is then reconstructed. Figure 2-16 shows the signal flow of the decoding process.
2.4.1.10 Decode the LP Filter Parameters

The transmitted line spectrum pairs, L0, L1, L2 and L3 are used to reconstruct the quantised LSP coefficients using the technique described in section 2.4.1.4 for each frame. The LSP coefficients are interpolated to obtain two sets of coefficients corresponding to two subframes. These coefficients are converted into LP filter coefficients and are used for synthesising the reconstructed speech for each subframe.

2.4.1.11 Decode the adaptive-codebook vector

The parity bit is first computed using the transmitted index value P1. If this is not identical to the value P0 then there are likely to be bit errors in the received data. If there are no errors then the integer and fractional parts of the pitch delay values are calculated for each subframe using P1 and P2. If there are bit errors, the integer pitch delay value of the first subframe is set to the pitch delay of the previous second subframe. The pitch-delay is then used to determine the adaptive-codebook vector from interpolating the previous excitation signals. The adaptive-codebook index gain is also calculated.

2.4.1.12 Decode the fixed-codebook vector

The fixed-codebook indexes, C1 and C2 are taken straight from the transmitted values in table 2-2. These give the positions of the excitation pulses. The signs of the pulses, S1 and S2 are
also extracted to calculate the fixed-codebook vector which is also dependent on the pitch delay value. The fixed-codebook gain is also calculated.

2.4.1.13 Reconstruct the Speech

The excitation signal is reconstructed by using the quantized adaptive and fixed-codebook gains ($g_p$ and $g_c$), the adaptive-codebook ($v(n)$) and the fixed-codebook vectors. This is given by the equation in figure 2-17.

$$u(n) = \hat{g}_p v(n) + \hat{g}_c c(n) \quad n = 0 \ldots 39$$

Figure 2-17: Reconstructed Excitation Signal Equation

The LP synthesis filter then reconstructs the speech using the calculated excitation signal, $u(n)$, and the interpolated LP filter coefficients, $\hat{d}_n$, for the current subframe.

$$s(n) = u(n) - \sum_{i=1}^{10} \hat{a}_i s(n-i) \quad n = 0 \ldots 39$$

Figure 2-18: Reconstructed Speech Signal

The equation in figure 2-18 shows how the speech signal is reconstructed. The sum of the product of LP filter coefficients and previous speech samples is subtracted from the excitation signal. The reconstructed signal is then processed further by the post processor which consists of an adaptive post filter, high-pass filtering and signal upscaling.

2.4.1.14 Post Filtering

The adaptive post filter improves the perceptual quality of the decoded speech. This is done by emphasising the formant and pitch peaks and smoothing the troughs between peaks. It is the noise energy within the troughs that are most likely to cause audible noise when reconstructing the speech. Attenuating these troughs minimises the distortion in the perceived reconstructed speech.

The adaptive post filter consists of a long-term post filter, a short-term post filter and a tilt compensation which is used to balance the spectral tilt introduced by the short-term post filter. The final part of the adaptive post filter is the adaptive gain control which is used to compensate for the energy differences in the reconstructed signal, $s(n)$, and the post-filter signal, $S(n)$.
2.4.1.15 High-Pass filtering and upscaling

This post-filtered signal is passed through a high-pass filter using a second-order pole-zero filter with a cut-off frequency of 100 Hz. The final stage of the speech codec is to multiply the filtered signal by a factor of 2 to restore the input signal level.

2.4.2 The ITU G.723.1 Dual Rate Speech Codec

The principal application of the G.723.1 was intended for the H.324 multimedia compression and transmission standard which includes the H.263 video codec. The speech coder was originally referred to as the G.723 but was renamed the G.723.1 as an older Adaptive Differential Pulse Code Modulation (ADPCM) based G.723 standard already existed. The coder was designed to operate with a digital signal acquired by performing telephone bandwidth filtering of the analogue signal using the G.712 standard. The digital signal was sampled at 8000 Hz and converted to 16-bit linear PCM for the input to the encoder. The output of the decoder should be converted back to analogue using similar methods.

The G.723.1 is a dual-rate speech coder and belongs to the category of linear-prediction analysis-by-synthesis speech coders which attempt to minimise a perceptually weighted error signal. The ITU standard can operate at either 5.3 kbit/s or 6.3 kbit/s. The higher bit rate results in better speech quality than the lower rate but the lower rate provides flexibility to design engineers, being less complex. Both rates are a compulsory part of the standard and it is possible to switch between rates at any frame boundary. The 5.3 kbit/s uses algebraic CELP technique which is also used by the ITU G.729 codec. For the 6.3 kbit/s operation, Multi-Pulse Maximum Likelihood Quantisation (MP-MLQ) excitation is used.

The encoder operates on frames of 240 speech samples. Since the digital signal is sampled at 8 kHz, this gives the total time of the encoder process to be 30 ms. There is also a look ahead of 7.5 ms resulting in a total algorithmic delay of 37.5 ms.

2.4.2.1 Encoding Principles

The dataflow of the G.723.1 encoder is shown in figure 2-19.
Speech Coding

2.4.2.2 Framer

The first part of the encoding process is buffering the speech samples, $y[n]$, into frames of 240 samples. Each frame is then divided into two parts (120 samples each) for pitch estimation. Each subframe is further divided once more into 2 subframes each having 60 samples.

2.4.2.3 High Pass Filter

Before further processing of the signal can be done, the speech signal is passed through a high-pass-filter which removes the remaining DC elements from the input speech, $s[n]$.

2.4.2.4 Linear Predictive Coefficient Analysis

Linear Predictive Coefficient (LPC) analysis, with a filter order of 10, is then performed on the signal $x[n]$. A 180 sample Hamming window is applied centred on each subframe and 11 autocorrelation coefficients are determined before using the Levinson-Durbin recursion to compute four LPC sets, one for each subframe. These coefficients are then used in the formant-based perceptual weighting filter which computes perceptual filter coefficients and applies a weighting filter.
2.4.2.5 Line Spectrum Pair Quantizer
The next part of the encoder is to transform the LPC coefficients into Line Spectrum Pair (LSP) format. A technique called Predictive Split Vector Quantizer (PSVQ) is used after a bandwidth expansion of 7.5Hz. The selected indices from the PSVQ are then transmitted.

2.4.2.6 LSP Decoder and Interpolator
Most codecs today decode and interpolate the LSP coefficient in a local simulated decoder which is implemented in sub components, as shown in figure 2-19. The LSP decoder inverses the quantization of LSP and then linear interpolation is performed between the decoded LSP vector and the previous LSP vector. Four interpolated LSP vectors are converted to LPC vectors. The quantized LPC synthesis filter, $\hat{A}(z)$, is used for generating the decoded speech signal.

2.4.2.7 Formant Perceptual Weighting Filter, Pitch Estimator and Harmonic Noise Shaping
The formant perceptual weighting filter is also followed by a harmonic noise shaping filter and an open loop pitch estimation. Two pitch estimates are computed for every frame, one for the first two subframes and the other for the last two subframes. The open loop pitch period is calculated using the weighted speech signal $f[n]$ for the range of 18 to 142 samples. The harmonic noise shaping filter computes the harmonic filter coefficients and then, for each subframe, searches for the optimum lag around the open-loop pitch lag based on positive correlation values only. A harmonic noise filter is then applied using an $L^{th}$ order FIR filter (where $L$ is the lag of the filter).

2.4.2.8 Impulse Response Calculator
The impulse response calculator in figure 2-19 computes the combined impulse response for the formant perceptual weighting filter, harmonic noise shaping filter and synthesis filter for each subframe, $S_i(z)$.

2.4.2.9 Zero Input Response
The next block for the G.723.1 is the zero input response of the combined filter, $S_i(z)$. The zero input response is obtained by setting the input signal to all zero-value samples. A ringing subtraction is then performed by subtracting the zero-input response from the harmonic noise weighted speech vector to produce the target speech vector.
2.4.2.10 Pitch Predictor

The open-loop pitch estimation calculated earlier is used for a more accurate closed-loop analysis-by-synthesis search in a limited range and takes place in the pitch predictor block. The contribution of the pitch predictor is treated as a conventional adaptive codebook and is a fifth order pitch predictor. For the first and third subframes, the closed loop pitch lag is refined around the open-loop pitch lag in the range ±1 and is coded using 7 bits. The open loop pitch lag is not coded or transmitted. For the second and fourth subframes only the difference with previous lag is encoded using 2 bits in the range of -1 to +2.

The pitch predictor gains are vector quantized using a 170 entry codebook for the 5.3kbit/s rate and either an 85 or 170 entry codebook for the higher 6.3kbit/s rate. The 170 entry codebook is the same for both rates. The 85 entry codebook is used for pitch gain quantization when the associated pitch lag value is less than 58 otherwise the pitch gain is quantized using the 170 entry codebook. The output from the pitch predictor can then be subtracted from the target speech vector to acquire the residual signal.

2.4.2.11 MP-MLQ

Depending on the bit rate, the residual signal is subjected to either MP-MLQ or ACELP excitation optimization. For the higher bit rate, MP-MLQ is used and the residual signal is transformed into a new target vector and is then quantized for one subframe of 60 samples. The algorithm uses an excitation of multiple non-uniformly spaced pulses. The locations of these pulses are determined using the Dirac function and the gain of the pulses are estimated by minimizing the mean square error of the signal. To improve the quality of speech with a short pitch period, a train of Dirac functions is used instead of a single Dirac function.

The number of excitation pulses is 6 for even subframes or 5 for the odd subframes. There is a restriction on the number pulse positions which can be either all odd or all even. This is indicated by a grid bit. There are 30 different pulse locations that the 6 excitations pulses can have. Hence there are \( \binom{30}{6} = 593775 \) different positions for the even subframes. Similarly for the odd subframes, there are \( \binom{30}{5} = 142506 \) different positions for the pulses. In the recommendation, the four Most Significant Bits (MSBs) from each subframe pulse position index are combined to 16-bits and then encoded to a 13-bit index. This reduces the number of bits needed to encode the pulse positions from 76 to 73 and also reduces the total number of bits transmitted. All these parameters are encoded and transmitted to the decoder.
2.4.2.12 ACELP

For the lower bit rate an ACELP excitation model is used. This model is similar to the one used in the 8 kbps G.729 codec and uses a 17-bit ACELP codebook. The code vector contains no more than four non-zero pulses which can have signs and positions given in table 2-3.

Table 2-3: ACELP excitation codebook for G.723.1

<table>
<thead>
<tr>
<th>Sigma</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>±1</td>
<td>0, 8, 16, 24, 32, 40, 48, 56</td>
</tr>
<tr>
<td>±1</td>
<td>2, 10, 18, 26, 34, 42, 50, 58</td>
</tr>
<tr>
<td>±1</td>
<td>4, 12, 20, 28, 36, 44, 52, (60)</td>
</tr>
<tr>
<td>±1</td>
<td>6, 14, 22, 30, 38, 46, 54, (62)</td>
</tr>
</tbody>
</table>

The pulses can occupy either even or odd positions in the subframe, like before. The bracketed pulse positions in the table are outside the subframe boundary and signify that the pulse is not present. Each pulse position is encoded with 3 bits and each pulse sign with 1 bit. Therefore, to encode the four excitation pulses, 16 bits are required. A further additional bit is also used so that the positions of the pulses can be simultaneously shifted by one to occupy odd positions. This makes the 17-bit codebook.

The structure of the codebook allows for a very fast search since the vector excitation only contains 4 non-zero pulses. The search is structured into 4 nested loops which corresponds to each pulse position. The search procedure is simplified by applying a precomputed threshold which is tested with the absolute correlation from the previous three pulses before entering the last loop. This is to see if it is worth continuing the search in terms of synthesized speech. The last loop is only entered if the threshold is exceeded. There is also a limit that the last loop is entered. This last step for the encoder is to quantize the gain. This is performed in the same way as the MP-MLQ.

2.4.2.13 Excitation Decoder

The decoding of the pulses in the excitation decoder is performed in the following way:

- The maximum gain index is derived
- The positions of the pulses are decoded depending on the bit rate (combinatorial decoding for the high bit rate and direct coding of the positions for the low rate)
- The grid positions are derived from the grid bit
- The pulse signs are derived from the sign bits
- Decoding of the pulse train is performed only for the high bit rate and only when there is a short pitch period
- The vector is reconstructed using the decoded parameters
- The pitch contribution, \( u[n] \), and the pulse contributions, \( v[n] \), are summed together to form the excitation vector, \( e[n] \).

2.4.2.14 Pitch Decoder

The pitch decoder, decodes the lag of the pitch predictor for both even and odd subframes. The gain vector of the pitch predictor of each of the subframes is derived from the gain index. These are then decoded and used to compute the pitch contribution.

2.4.2.15 Memory Update

The last task of the encoder is to update the memories of the synthesis filter, \( \hat{A}_i(z) \), the formant perceptual weighting filter, \( W_i(z) \), and the harmonic noise shaping filter, \( P_i(z) \), before encoding the next subframe. The complete response of the combined filters, \( S_i(z) \), is calculated by passing the excitation vector through this filter. The memory of the combined filter is then saved and used to compute the zero input response for the encoding of the next speech vector.

2.4.2.16 Bit Allocation

The bit allocation for the high bit rate excitation is shown in table 2-4 and similarly for low bit rate excitation is shown in table 2-5. The major difference has to do with how the two rates encode the pulse positions and amplitudes. For the higher rate the pulse positions and pulse sign requires \( 73 + 22 = 95 \) bits per \( 30 \text{ms} = 3.1 \text{ kbps} \) in total. For the lower rate codec, this requires \( 48 + 16 = 64 \) bits per \( 30 \text{ms} = 2.1 \text{ kbps} \). This results in the difference of 1 kbps for the two algorithms.
Table 2.4: Bit allocation of the 6.3 kbit/s coding algorithm for G.723.1

<table>
<thead>
<tr>
<th>Parameters Coded</th>
<th>Subframe 0</th>
<th>Subframe 1</th>
<th>Subframe 2</th>
<th>Subframe 3</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPC indices</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>Adaptive codebook lags</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>Excitation and pitch</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>48</td>
</tr>
<tr>
<td>gains combined</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pulse Positions</td>
<td>20</td>
<td>18</td>
<td>20</td>
<td>18</td>
<td>73*</td>
</tr>
<tr>
<td>Pulse signs</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>22</td>
</tr>
<tr>
<td>Grid Index</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>189</td>
</tr>
</tbody>
</table>

* The first four MSBs of each subframe pulse position index can be combined saving 3 bits

Table 2.5: Bit allocation of the 5.3 kbit/s coding algorithm for G.723.1

<table>
<thead>
<tr>
<th>Parameters Coded</th>
<th>Subframe 0</th>
<th>Subframe 1</th>
<th>Subframe 2</th>
<th>Subframe 3</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPC indices</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>Adaptive codebook lags</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>Excitation and pitch</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>48</td>
</tr>
<tr>
<td>gains combined</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pulse Positions</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>48</td>
</tr>
<tr>
<td>Pulse signs</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Grid Index</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>158</td>
</tr>
</tbody>
</table>

The list of parameters that are transmitted from the encoder to the decoder are listed in table 2-6.
Table 2-6: List of transmitted Parameters from the G723.1 Coder

<table>
<thead>
<tr>
<th>Name</th>
<th>Transmitted Parameters</th>
<th>Number of Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>6.3kbps</td>
</tr>
<tr>
<td>LPC</td>
<td>LSP VQ index</td>
<td>24</td>
</tr>
<tr>
<td>ACL0</td>
<td>Adaptive CodeBook Lag</td>
<td>7</td>
</tr>
<tr>
<td>ACL1</td>
<td>Differential Adaptive CodeBook Lag</td>
<td>2</td>
</tr>
<tr>
<td>ACL2</td>
<td>Adaptive CodeBook Lag</td>
<td>7</td>
</tr>
<tr>
<td>ACL3</td>
<td>Differential Adaptive CodeBook Lag</td>
<td>2</td>
</tr>
<tr>
<td>GAIN0</td>
<td>Combination of adaptive and fixed gains</td>
<td>12</td>
</tr>
<tr>
<td>GAIN1</td>
<td>Combination of adaptive and fixed gains</td>
<td>12</td>
</tr>
<tr>
<td>GAIN2</td>
<td>Combination of adaptive and fixed gains</td>
<td>12</td>
</tr>
<tr>
<td>GAIN3</td>
<td>Combination of adaptive and fixed gains</td>
<td>12</td>
</tr>
<tr>
<td>POS0</td>
<td>Pulse Position Index</td>
<td>20*</td>
</tr>
<tr>
<td>POS1</td>
<td>Pulse Position Index</td>
<td>18*</td>
</tr>
<tr>
<td>POS2</td>
<td>Pulse Position Index</td>
<td>20*</td>
</tr>
<tr>
<td>POS3</td>
<td>Pulse Position Index</td>
<td>18*</td>
</tr>
<tr>
<td>PSIG0</td>
<td>Pulse Sign Index</td>
<td>6</td>
</tr>
<tr>
<td>PSIG1</td>
<td>Pulse Sign Index</td>
<td>5</td>
</tr>
<tr>
<td>PSIG2</td>
<td>Pulse Sign Index</td>
<td>6</td>
</tr>
<tr>
<td>PSIG3</td>
<td>Pulse Sign Index</td>
<td>5</td>
</tr>
<tr>
<td>GRID0</td>
<td>Grid Index</td>
<td>1</td>
</tr>
<tr>
<td>GRID1</td>
<td>Grid Index</td>
<td>1</td>
</tr>
<tr>
<td>GRID2</td>
<td>Grid Index</td>
<td>1</td>
</tr>
<tr>
<td>GRID3</td>
<td>Grid Index</td>
<td>1</td>
</tr>
</tbody>
</table>

* - The 4 MSB of these are combined to form a 13-bit index saving 3 bits
2.4.2.17 Decoder Principles

The schematic of the G.723.1 decoder is shown in figure 2-20.

![Block diagram of the G.723.1 Speech Decoder](figure2-20.png)

2.4.2.18 LSP Decoder, LSP Interpolator, Pitch Decoder and Excitation Decoder

The LSP decoder, LSP Interpolator, pitch decoder and excitation decoder blocks are exactly the same design as those in the simulated decoder in the encoder and are performed on a frame-by-frame basis.

2.4.2.19 Pitch Postfilter

The pitch postfilter is used to improve the quality of synthesized signal. It is performed on every subframe but the whole frame excitation signal is saved within this block. The quality of the signal is improved by increasing the Signal to Noise (SNR) at multiples of the pitch period. The computation of the gain and the delay of the excitation signal is decoded during this stage. This is done based on a forward and backward crosscorrelation analysis.

2.4.2.20 Synthesis Filter

The LPC synthesis filter implements a 10th order Infinite Impulse Response (IIR) filter to the signal $\hat{A}(z)$ and which is used to synthesize the signal $sy[n]$ from the decoded pitch postfilter residual $ppf[n]$ for each subframe.

2.4.2.21 Formant Postfilter

The formant postfilter applies a 10-pole, 1-zero Autoregressive Moving-Average filter to the synthesized signal and also calculates the energy of the subframe vector.
The last part of the decoder is to scale the postfiltered output vector, \( pf[n] \). This is done by first calculating the amplitude ratio between the synthesized speech vector, \( sy[n] \), and the postfiltered output vector, \( pf[n] \).

### 2.4.2.22 Gain Scaling Unit

The final output vector, \( q[n] \), is obtained by scaling the postfiltered signal, \( pf[n] \), and the gain, which is calculated using the amplitude ratio and the previous gain value.

### 2.5 Conclusion

Speech coding is a key concept in making VoIP networks work. There are two speech codecs that are commonly used in VoIP networks and these are mainly the G.729A, operating at 8 kbit/s, and the dual rate G.723.1, operating at either 5.3 or 6.3 kbit/s.

This chapter first explored the different types of speech codecs that are available for encoding speech, with the oldest method being the Channel Vocoder designed by Homer Dudley. Another vocoder system explored was the Formant Vocoder where only the formant and the vocal excitation data are transmitted. The main difficulty of the Formant Vocoder is finding formants locations. To overcome this problem the Linear Prediction Coding Vocoder is introduced and are the most commonly used vocoders. The final type of speech coder explored is the Analysis-by-Synthesis Codecs. These speech codecs are based on the LPC codecs with the exception that they analyse a large set of excitations and select one which closely matches the original speech waveform. These speech codecs are further split into different categories:

- Multi-Pulse Excited Codec
- Regular-Pulse Excited Codec
- Code Excited Linear Predication Codec

The both the G.729A and the G.723.1 speech codecs fall into the Analysis-by-Synthesis codecs and are described in section 2.4.1 and 2.4.2 respectively. The G.729A and the G.723.1 at 6.3 kbit/s is a code excited linear predication codec while the G.723.1 at 5.3 kbit/s is a multi-pulse excited codec.
2.6 References

Speech Coding


[2-33] ITU Recommendation G.723.1, "Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s", 3/96


3 PARALLELISM

3.1 Introduction

Parallelism is a micro-architectural technique for improving the performance of processors via the simultaneous execution of multiple instructions. Most processors from the late 80s have parallelism amongst instructions initially through pipelining [3-1]. This method allows overlapping the execution of instructions, reducing the amount of time taken to complete a stream of instructions. The important issue in pipelining is ensuring instructions do not depend on the previous instruction; otherwise the pipeline has to be stopped (stalled) while the previous instruction completes. The first known pipelined processor was the IBM 7030 and was called Stretch. It followed from the IBM 704 processor in the 1950s and was 100 times faster [3-5, 3-6]. The processor employed a 4-stage pipeline.

Most RISC CPU instructions can be classified into three groups [3-1]:

- Arithmetic Logic Unit (ALU) operations – typically subtraction, addition, shifting or logical operations
- Load/Store Instructions – Loads/Store data from a memory address
- Branch – Change of program control flow

For a 7 stage integer RISC pipeline, as shown in figure 3-1, every instruction can be executed in at least 7 clock cycles. The 7 pipeline stages are:

1. **Fetch** – The Program Counter (PC) is used to fetch the instruction to be executed from either memory or the instruction cache. The PC is updated to the next address by adding 4 (since each instruction is 4 bytes).

2. **Decode** – The instruction is decoded and the control fields are set for the following clock cycles. The target addresses for all branches are also calculated.

3. **Register Access** – The registers are accessed to obtain
Parallelism

the operands needed.

4. **Execution** – ALU operates on the operands read in the previous cycle. For Load/Store instructions the address is calculated.

5. **Memory** – For a load instruction, memory is accessed using the address calculated in the ALU. For a store, the data calculated is passed on to the data cache.

6. **Exception** - Any traps and interrupts are evaluated during this stage.

7. **Write Back** – Data is written back into the register file whether it comes from memory or from the main integer pipeline.

---

Table 3-1: 7-Stage RISC Pipeline

<table>
<thead>
<tr>
<th>Instruction</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>Clock Cycles</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction 1</td>
<td>F</td>
<td>D</td>
<td>R</td>
<td>X</td>
<td>M</td>
<td>X</td>
<td>MEM</td>
<td>EX</td>
<td>C</td>
<td>W</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 1+1</td>
<td>F</td>
<td>D</td>
<td>R</td>
<td>X</td>
<td>M</td>
<td>X</td>
<td>MEM</td>
<td>EX</td>
<td>C</td>
<td>W</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 1+2</td>
<td>F</td>
<td>D</td>
<td>R</td>
<td>X</td>
<td>M</td>
<td>X</td>
<td>MEM</td>
<td>EX</td>
<td>C</td>
<td>W</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 1+3</td>
<td>F</td>
<td>D</td>
<td>R</td>
<td>X</td>
<td>M</td>
<td>X</td>
<td>MEM</td>
<td>EX</td>
<td>C</td>
<td>W</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 1+4</td>
<td>F</td>
<td>D</td>
<td>R</td>
<td>X</td>
<td>M</td>
<td>X</td>
<td>MEM</td>
<td>EX</td>
<td>C</td>
<td>W</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction 1+5</td>
<td>F</td>
<td>D</td>
<td>R</td>
<td>X</td>
<td>M</td>
<td>X</td>
<td>MEM</td>
<td>EX</td>
<td>C</td>
<td>W</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If= Fetch  
DE= Decode  
RA= Register Access  
X= Execute  
MEM= Memory  
EXC= Exception  
WB= Write Back

Table 3-1 shows a typical way of drawing a pipeline structure. For every clock cycle, a new instruction is issued and the CPU can be executing up to 7 different instructions, each at various stages of completion. Instructions can not use the same resources at the same time i.e. a single ALU can not be asked to compute a memory address and subtract two different operands on the same cycle and is referred to as a structural hazard. This is done inherently due to the nature of RISC instructions. All major blocks of the RISC pipeline are separated via pipeline registers.

Although pipelining increases performance by improving the throughput, it has no effect in the time it takes for an instruction to execute as it will still take 7 clock cycles. The program will run faster due to many instructions executing in parallel. From table 3-1 it can be seen that 7 instructions can be completed before the fourteenth clock cycle. If this processor was not pipelined, less than 2 instructions could be executed by that time.
3.2 Types of Parallelism

So far, the discussion has been about pipelined processors; a basic form of Instruction Level Parallelism (ILP). There are different types of parallelism which fall into four different possible categories as defined by Michael Flynn [3-2, 3-3].

- Single Instruction, Single Data (SISD) – Most embedded processors belong to this category where one instruction is issued and operates on a single piece of data.

- Single Instruction, Multiple Data (SIMD) – One instruction operates on different pieces of data, decreasing the instruction bandwidth of a processor. This is known as Data Level Parallelism (DLP) and is often found in vector processors. It is an abundant form of parallelism in multimedia and communications cases where most computations are carried out are in data-parallel loops.

- Multiple Instructions, Multiple Data (MIMD) – In this type of parallel architecture, multiple instructions operate on different data segments. An example of this would be Thread Level Processors (TLP). Typically, there would be a number of processors running different instructions, operating on different parts of the data algorithm set.

- Multiple Instructions, Single Data (MISD) – In this final category, many instructions operate on the same piece of data. There are not many instances of this type of architecture due to the scalability and functional resources of SIMD and MIMD processors. However, an MISD processor can be useful in fault detection systems where the application needs a high level reliability. Two functional units operate on the same piece of data and check the results for consistency e.g. in safety critical systems.

These categories describe the execution architecture during every clock cycle. For SISD architecture, one instruction would operate on one specific piece of data every clock cycle. Therefore the pipeline processor discussed in the previous section would fall into this category. The MIMD architecture would issue multiple instructions and operate on a range of data segments per clock cycle.

Section 3.3.1 explores the different approaches used to process multiple datasets. These are Instruction Level Parallelism (ILP), Data-Level Parallelism (DLP) and Thread-Level Parallelism (TLP).
3.2.1 Dependencies and Hazards

Determining where the instructions are dependent is crucial to the execution of parallel instructions. Generally, instructions are written sequentially and are executed as such. Dependencies happen when an instruction has to be executed before the following instruction. If the instructions are independent of each other than they can be executed in parallel. If they are dependent, they must be executed in order to preserve the program correctness flow, however, they may be partially parallelised.

1) loop: 
   ldr r1, [a, r0] ; load array element a

2) add r4, r1, r2 ; add array element to r2

3) str r4, [c, r0] ; store result to array c

4) add r0, r0, #1 ; increment counter

5) cmp r0, #8 ; test counter

6) bnez r0, loop; ; loop

Figure 3-2: Sample Code showing data dependencies

Figure 3-2 depicts a simple code sequence which loads a value stored in memory, adds the value in the second register and stores the results back in memory and repeats eight times. The arrows in the code show the instructions dependencies. In this example, some of the instructions depend on the result from a previous instruction i.e. line 1 loads data into register r1 and this is then used in line 2 for the addition. The result from the addition is then stored back to memory. There is parallelism in this code, lines 1-3 are independent of lines 4-6 and may be executed in parallel.

There are a number of methods which can be employed to remove dependencies and maximise the benefit of parallelism such as pipelining [3-4], bypassing [3-5, 3-6] and instruction reordering [3-7]. In software code, there can be false dependencies where the software code is written in such a way that the dependencies need not be there. These sections of code can be re-written reordering the instructions and removing the dependencies.

Preserving the program semantics is crucial when reordering instructions. The instructions can now be executed earlier/later and can have a possible consequence of affecting the execution of previous/following instructions. These hazards can be categorised in the following way:
• Read after Write (RAW) — Instruction 2 tries to read an operand before instruction 1 commits the result. This means that the second instruction obtains the old value. This is a common type of data hazard and is often found when a load instruction followed by ALU instruction that uses the load result in a scalar processor.

• Write after Write (WAW) — Instruction 2 writes to a location before it is written by instruction 1. This means that data has been written in the wrong order and now contains the wrong value written by instruction 1 instead of instruction 2. This type of hazard happens in architectures which allow write operations in more than one stage; an instruction is allowed to continue down the pipeline even if the pipeline is stalled, this is a particular hazard in processors with a parallel architecture.

• Write after Read (WAR) — Instruction 2 writes to a location before it is read by instruction 1. This means that instruction 1 has obtained the new value. This hazard can not happen in static pipelines as all read operations happen early on in the pipeline during the Register Access stage, whereas the write operations happen late on during the Write Back. This hazard generally occurs if the instructions have been reordered or in multiprocessors where instructions finish too early.

3.3 Parallelism Types

This section explores the different architectures used when processing multiple datasets. The first method, Instruction Level Parallelism (ILP), issues multiple instructions per clock cycle increasing the throughput of instructions and is classified MIMD using Flynn’s taxonomy. Data Level Parallelism (DLP) is a SIMD design where a single instruction can operate on multiple data items. The programmer needs to analyse the code to find instances where a single instruction can replace instances where the same operation is performed on a dataset, often in data-parallel loops (depicted in figure 3-5). The final method of parallelism is Thread Level Parallelism (TLP) and comes under the classification of MIMD. This form of parallelism uses multiple processors operating in parallel.

3.3.1 Instruction Level Parallelism (ILP)

A basic form of ILP has already been discussed in section 3.1. This can be exploited further by the means of issuing multiple instructions on every clock cycle operating on multiple data sets. In the previous section, only one instruction was being issued for every clock cycle. By issuing multiple instructions on every clock cycle there will be dramatic increase in performance, exploiting the parallelism found in most codes. For ILP processors, there also needs to be a set of Functional Units (FU) to execute the instructions.
Parallelism

Figure 3-3: Multiple Instruction Issue System

Figure 3-3 illustrates an ILP processor with an issue width of 3. This means that the processor has the potential of executing 3 instructions every clock cycle. The organisation of the FU depends on the application to achieve high performance i.e. The FU in the first lane could contain a floating point unit, a barrel shifter and rounding units whereas the FU in the second lane could contain three Multiply-Accumulate units. For ILP to achieve the best possible performance, the processor needs to effectively issue instructions to every lane on every clock cycle. Due to dependencies, hazards and FU layouts, this is not always the case. The efficiency of the processor decreases when a lane is not operating on an instruction.

Multiple-Issue processors come in two different forms – Superscalar [3-8] and Very Long Instruction Word (VLIW) [3-9].

3.3.1.1 Superscalar

Superscalar processors are able to issue a range of instructions per clock cycle. They can be either statically scheduled, where the instructions are executed in order defined by the compiler, or dynamically scheduled where the processor uses buffers to reorder the instructions, this is often called out-of-order execution. In the latter option, the programmer and compiler have no knowledge of the parallelism in the processor. The instructions produced are examined by a scheduler embedded into the processor to check for dependencies and hazards. This information is used to schedule the instructions for executing the instructions on the appropriate FU.

With the throughput of instructions greatly increased and accompanied with the portability of the processor, the instructions executed on superscalar processors are similar if not the same, superscalar processors are becoming popular with modern consumer processors. The mid
1980s saw superscalar processors appearing with the IBM RISC 6000 processor [3-10] and the ZS-1 Central Processor [3-11], issuing multiple instructions per clock cycle for multiple pipelines.

3.3.1.2 Very Long Instruction Word

VLIW processors issue a fixed number of instructions per clock cycle. As the name suggest, the instructions to be executed are formatted in to one large instruction termed as a fixed instruction packet. Parallelism among the instructions can be redefined by a larger instruction. Because of the nature of VLIW processor instructions, they are generally statically scheduled by a compiler, removing the need for complicated scheduling logic. This in turn complicates the compiler as it needs to know specific details about the processor i.e. the number of FU and the type and order of instructions to be executed. As a result, binaries produced are not as portable as is the case with superscalar processors.

The earliest VLIW processors were designed to accelerate floating-point computations, Floating Point Systems FPS-164 and FPS-264 [3-12]. These processors were limited in programmability due to their complexity. They were effectively coprocessors and controlled by a host processor.

3.3.1.3 Multiple Issue Processor Research

There has been considerable research into Superscalar and VLIW processors which has shown significant improvement into their performance. In [3-13], they discuss the typical microarchitecture of a number of superscalar processors and their effect on performance is discussed. They round up their research by focusing on superscalar processors in the market including the MIPS R10000 [3-14] and the AMD K5 [3-15] processors. Although superscalar processors (and VLIW processors) are good at improving performance they are still susceptible to control hazards. A method to eliminate hazards is presented in [3-16]. The authors describe how they focus their design on the control dependencies using a combination of hardware software techniques while extracting the ILP from different applications. A different method of ILP is reported in [3-17] with Instruction Level Distributed Processing and its performance compared with superscalar processors. Instead of very deep pipelines, their microarchitecture consists of a number of distributed processing elements each with a simple in-order pipeline.

Studies have been performed to determine the performance of superscalar processors. In [3-18], the author reports on the performance improvement that ILP can have on programs, but the costs involved to gain this performance are too expensive to produce. Good branch
prediction in hardware (dynamic) and software (static) are needed to make the ILP a viable solution in improving processor performance, [3-19].

Although superscalar and VLIW processors achieve very good performance, the complexity at the decoding stage becomes more involved due to calculating the dependency of instructions and needing a slower clock speed. There is also a significant increase in power consumption due to the issue/bypass logic being so complex and expensive to implement. Even with regards to memory, power is also major issue [3-20]. To achieve the high clock frequency, deep pipelines are needed however this trades in the performance by having more hazards to negotiate and a lower number of instructions being issued per cycle.

3.3.2 Data-Level Parallelism (DLP)

Data Level Parallelism (DLP) is another method used to achieve high performance by focusing on the data level where multiple data computations run concurrently using a single instruction. The computations are performed independently on individual functional units all operating in lockstep, connected together and executing the same operation. The parallelism exploited by DLP processors is dependent on having large data sets, found in most multimedia applications - graphics, video, speech, encryption, where the same operation is executed over and over again. The first DLP machine shortly appeared after pipelining was introduced with the Cray-I Computer System [3-21] in 1976. The machine introduced vector style instructions which specified a series of operations to be performed on vector-linear arrays. Each operation is independent of each individual data element and is therefore highly parallel. Each operation is performed on separate functional units. One vector instruction can be equivalent to an entire data-parallel loop dramatically reducing the instruction bandwidth and conditional branches eliminating the control hazards.

3.3.3 Vector Architecture

There are two types of vector architecture: vector-register processors and memory-memory processors. In vector-register processors all operations (excluding load/store instructions) happen within vector registers. With memory-memory processors, operations are performed on data which resides in normal or cached memory.
The basic architecture of a vector processor is shown in figure 3-4. The main components of the architecture are the vector and scalar registers, functional units and the load/store unit. Each vector register is of fixed length and accommodates a single vector. Designers often need to determine the best length and an appropriate number of registers to achieve maximum performance while reducing overhead costs. For example, The Cray-I system had 8 vector registers each having 64-bit elements, whereas the Hitachi 8820 processor [3-22], introduced in 1983, has 32 vector registers and 256-bit elements in each. The vector processors have a Vector Length Register (VLR) to control the data being operated on. The VLR value cannot exceed the length of the vector registers (64 bits for the Cray-I processor and 256 bits for the Hitachi 8820 processor).

The functional units execute the vector instructions and the number of them depends on the application. The Cray-I processor has 6 functional units: floating point add, multiply, reciprocal, an integer adder, logical block and shift unit. The Hitachi 8820 has only 4 functional units; floating point multiply-add, multiply/divide-add unit and 2 integer add/logic blocks. The multiply-add functional unit performs a floating point multiplication followed by either an addition or a subtraction. The multiply/divide-add functional unit performs floating point multiplication or division followed by addition or subtraction.
The final part of the vector architecture is the Load/Store Unit. This loads data from memory and creates the vectors to be operated on. The scalar registers can provide inputs to the functional units, but they also compute data addresses for the vector load-store unit.

Consider the following code segment:

\[
\text{for}(i=0; i<8; i++) \\
\quad c[i] = a[i] + b[i];
\]

**Figure 3-5: Data-parallel loop code segment for 8-element array addition**

The code shown in figure 3-5 is an ideal code segment which can be implemented on DLP processors where the elements are 16-bits wide. This code is highly parallel where each of the array elements does not depend on previous or future values and is therefore not data dependent.

![Figure 3-6: 16-bit 8-element array addition](image)

**Figure 3-6: 16-bit 8-element array addition**

Figure 3-6 shows the implementation on a general purpose processor for the code segment. Each addition instruction will be operated on every clock cycle. There are further overheads as control and data hazards which may cause pipeline stalls. To compare vector instructions, the code segment is best viewed in assembler and is shown below:

1) \text{mov} r0, #0 ; reset loop counter
2) \text{loop:} \text{ldr} r1, [a, r0] ; load element from a
3) \text{ldr} r3, [b, r0] ; load element from b
4) \text{add} r1, r3, r1 ; add elements from a and b
5) \text{str} r1, [c, r0] ; store result to element from c
6) \text{add} r0, r0, #1 ; increment and test loop counter
7) \text{cmp} r0, #8
8) \text{bnez} r0, \text{loop;}

**Figure 3-7: Assembler for 16-bit 8-element array addition**

It can be seen that to complete addition of all array elements the processor executes 57 instructions. By increasing the number of iterations of the loop, there will be a proportional increase in the number of lines of code to be executed. Although the processor will execute
the 57 instructions, there will be more clock cycles due to the branch control and memory latency in the load instructions on lines 2 and 3.

The benefits of a vector processor can be seen in the assembler code in figure 3-8.

1) ldrv v1, a; load array a to vector reg
2) ldrv v3, b; load array b to vector reg
3) addv v4, v2, v3; add vector registers
4) strv v4, c; store vector result

Figure 3-8: Vector Assembler Code

Assume that the vector processor has a vector length of 128-bits. The vectors are loaded into the registers on lines 1 and 2. The addition on line 3 will take place in the functional units of the vector processor and the final results will be written back to main memory. The total number of instructions executed has now dramatically reduced to 4 as the loop overhead is no longer present. However, line 3 has a data hazard due to the data needed is being loaded into the vector register on the previous line. Here, the pipeline of the processor will need to be stalled till the load has finished. Vector processors also help reduce the memory latency compared to general processors. If data sets are adjacent, fetching vectors from a heavily interleaved memory is quicker due to sequential accesses compared to general processor which load single elements at a time and incur set-up delays for every load instruction (16 load instruction in figure 3-7 compared to 2 in figure 3-8). High-speed processors will often cache data to avoid too many memory accesses and so help reduce the long latency times. For computationally intensive systems which require large data sets, this is not always possible.

3.3.4 Thread-Level Parallelism (TLP)

Thread Level Parallelism (TLP) is a method of distributing the workload across a number of processing elements. Each element will process a different part of the Control Data Flow Graph (CDFG) of a single application (multi threaded workload) or process multiple CDFGs (multi-programming workload). By spreading the processing across a multiple of processing elements, performance can be improved.
Figure 3-9: Thread Level Parallelism System

Figure 3-9 depicts the concept behind a TLP system. On a standard CPU, software code is executed sequentially. If the code is independent, then it can be implemented in a parallel configuration as shown. This will increase the throughput of instructions as there are now multiple processing elements operating on different parts of software code. By achieving higher throughput, TLP systems can process more data in the same time compared to a standard CPU.

Problems arise in TLP systems when a particular processing element tries to reference a piece of data that is being operated on by another. This can cause RAW, WAW and WAR hazards as discussed in section 3.2.1. Methods can be produced to ensure program order, but this increases code complexity and reduces performance as a thread is effectively “put to sleep” until another thread has finished with the data.
Two different architectures have been developed to store data, depicted in figure 3-10. Since TLP operates on the same programme, it is important that the data between the processing elements are consistent. The obvious method to implement would be to have a single central memory accessible to all processors over a bus, illustrated in figure 3-10 A. With this type of architecture, processors can be in contention over the bus if more than one processor requires access to memory. Therefore the number of processors needs to be limited on this type of architecture before there is too much contention on the bus. To address the problem of contention, a distributed memory system can be employed, figure 3-10 B. Each processor has its own private memory. This way processors can operate on data located in their own memory space reducing the need to access the bus. The drawback in this situation is when a processor requires data from a different memory location. Using special transfer instructions, data can be transferred over a bus from one private memory to another. There is little contention with this architecture but the time it takes to pass data from one processor to another can be substantial.

### 3.4 Parallelism Background

There has been significant research into parallelism. Early investigation reported the limitations of ILP processors [3-23] especially due to the number of instructions that can be issued before a control hazard occurs. This can be overcome by good branch prediction either in hardware or software and reordering of the instructions. Another method of improving the performance of ILP processors is presented in [3-24] where the authors decompose the program into small fragments of code which are dynamically scheduled. VLIW processors have also been studied in the field of general-purpose applications with additional hardware to improve the performance [3-25] which can handle events at run-time that are not computed by the compiler. Memory problems can occur when introducing ILP processors. Hardware must check to see if a load instruction targets the same address as an earlier store instruction. It is possible that the load instruction completes before the store instruction is actually issued. The
load instruction must use the value in the earlier store instruction (store-load forwarded). Another memory problem is when a store instruction targets the same address as a later load instruction. Again, it is possible for the load instruction to be ordered before the store instruction. Program order must be maintained and checked. In [3-26], the authors present a simple method of address matching with other load or store instructions to overcome this memory problem.

There has also been research into combining ILP and DLP where the authors in [3-27] report an increased performance at low complexity by merging the two techniques. This has been investigated further in [3-28] by introducing a processor which can be configured for different types of parallelism: ILP, DLP and TLP. The authors in [3-29] use TLP and manually select parts of the software code to execute on different CPUs for MPEG4 and H.264.

Early research into DLP and vector processor microarchitecture has shown their potential in different applications, especially in multimedia applications [3-30]. Vector operations are essentially a collection of scalar instructions. Because these instructions execute in parallel, there are no data hazards associated with them so hardware does not need to check. Loop controls are implicit; therefore, there are no control hazards to worry about. Vector processors are more power efficient than superscalar processors [3-31]. They fetch, decode and execute fewer instructions and have efficient custom hardware function units whereas superscalar processors have more hardware to check for hazards. Vector processors can operate at high clock frequency and have deep pipelines. Due their nature, they have high number instructions per clock cycle. Due to the nature of vector processors, it is possible to reorder the instructions without much difficulty as long as program execution order is maintained. However, vectorising code is not easy.

1) \[
\text{for } (i=1; \ i<N; \ i++)
\]
2) \[
b[i] = a[i] + b[i-1];
\]

**Figure 3-11: Vectorising Code Sample 1**

Figure 3-11 shows an example of code which cannot be replaced by vector instructions. Although it appears that the code can be replaced by vector instructions, the problem lies with the second line of the code. The second operand in the addition equation is data dependent on the previous element. This value had to be calculated in a previous loop cycle and is therefore not parallel.
Parallelism

1) \[ \text{for } (I = 0; I < N; I++) \]
2) \[ \text{for } (J = 0; J < N; J++) \]
3) \[ \text{for } (K = 0; K < N; K++) \]
4) \[ C[I][J] += A[I][K] \ast B[K][J]; \]

Figure 3-12: Vectorising Code Sample 2

The code in figure 3-12 looks like it can not be replaced by vector instructions. The code implements a matrix multiplication, as shown in figure 3-13.

\[
\begin{array}{c}
\text{C} \\
\downarrow \\
\rightarrow j
\end{array}
\quad = 
\quad \begin{array}{c}
\text{C} \\
\downarrow \\
\rightarrow j
\end{array}
\quad + 
\quad \begin{array}{c}
\text{A} \\
\downarrow \\
\rightarrow k
\end{array}
\quad \times 
\quad \begin{array}{c}
\text{B} \\
\downarrow \\
\rightarrow k
\end{array}
\]

Figure 3-13: Matrix Multiplication

In this example, the rows in A are being multiplied by the columns in B. They are then adding to the result to the element in C. To be able to vectorise the code in figure 3-12, the code needs to be reordered.

1) \[ \text{for } (I = 0; I < N; I++) \]
2) \[ \text{for } (K = 0; K < N; K++) \]
3) \[ \text{for } (J = 0; J < N; J++) \]
4) \[ C[I][J] += A[I][K] \ast B[K][J]; \]

Figure 3-14: Vectorising Code Sample Reordered

Figure 3-14 shows the reordered code. The overall program execution order is still the same but it is easier to replace the code with vector instructions. Instead of multiplying a row by column, the code now multiplies a row in array B by a single element in A followed by addition of a row in C, as shown in figure 3-15.

\[
\begin{array}{c}
\text{C} \\
\downarrow \\
\rightarrow i \\
\quad = 
\quad \begin{array}{c}
\text{C} \\
\downarrow \\
\rightarrow i \\
\quad + 
\quad \begin{array}{c}
\text{A} \\
\downarrow \\
\rightarrow k
\end{array}
\quad \times 
\quad \begin{array}{c}
\text{B} \\
\downarrow \\
\rightarrow j
\end{array}
\end{array}
\end{array}
\]

Figure 3-15: Matrix Multiplication Reordered

The code can now easily be replaced by vector instructions. However, due to the nature of vector processors, only lines 3 and 4 can be substituted by them. Vector processors can only
handle inner parallel loops and therefore lines 1 and 2 can not be replaced. Due to the nature of this kind of optimization tied in with the complication of data dependencies, it is very difficult to design an automatic vectorisation compiler and thus, vectorisation is often done by hand.

3.5 Profiling speech codecs

The ITU provides a set of reference C code for a range of speech codecs. This research looks at the G.723.1 dual rate speech codecs for multimedia communications transmitting at 5.3 & 6.3kbits/s and the G.729A coding of speech at 8kbits/s using conjugate-structure algebraic-code excited linear-prediction.

Within the supplied reference code, a set of universal, basic arithmetic operations (BASOP) are defined relating to simple DSP related functions, e.g. addition, subtraction, multiplication etc. By using the SimpleScalar toolset [2-32], the instruction count distribution can be seen in the pie chart in figure 3-16 which depicts the instruction count distribution of all functions when running the G.729A coder with one of the sets of test vectors supplied by the ITU-T.

![G729 Coder Instruction Distribution](image)

**Figure 3-16: Percentage of overall instruction count spent within BASOP functions**

It is clear that 78% of the total machine instructions executed are inside the BASOP functions. A number of the operations in these functions appear in data-parallel loops. Similar results can be seen for the G.723.1. The workloads were executed and profiled in native mode (Linux X86). Table 3-2 shows the relative time spent outside the DSP emulation instructions. To research the potential acceleration of the algorithms when executed on an embedded microprocessor, the workload has been recompiled for the SimpleScalar instruction set architecture.
Table 3-3 illustrates the simulated processor profiling results in terms of the number of instructions executed.

Table 3-2: Relative number of total instructions executed outside the DSP emulated instructions simulated on Linux 86

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Relative Time (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>G723 Encoder</td>
<td>31.3</td>
</tr>
<tr>
<td>G723 Decoder</td>
<td>22.8</td>
</tr>
<tr>
<td>G729A Encoder</td>
<td>30.4</td>
</tr>
<tr>
<td>G729A Decoder</td>
<td>26.9</td>
</tr>
</tbody>
</table>

Table 3-3: Relative number of total instructions executed outside the DSP emulated instructions simulated on the SimpleScalar Instruction Set

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Relative Time (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>G723 Encoder</td>
<td>34.5</td>
</tr>
<tr>
<td>G723 Decoder</td>
<td>33.3</td>
</tr>
<tr>
<td>G729A Encoder</td>
<td>34.2</td>
</tr>
<tr>
<td>G729A Decoder</td>
<td>37.2</td>
</tr>
</tbody>
</table>

The workloads spend a significant proportion of their time and instructions executing DSP emulation functions. A total of 67% machine instructions were executed inside these BASOP. On visual inspection of the G.723.1 and the G.729A code, it suggested that most of these BASOP are executed in data parallel loops which can be exploited in parallelism architecture. If the DSP emulation instructions could be executed by configurable extensible vector microprocessor with a targeted data parallel architecture which closely matches the BAOSP, there is the potential to achieve a significant reduction in execution time and higher performance [3-33]. A suitable high-performance, targeted-architecture for executing the workloads could reduce the form-factor and power consumption, making it a very attractive candidate for replication and integration in a SoC ASIC.

3.6 References


4 ARCHITECTURAL RESULTS

4.1 Introduction

This chapter presents the architectural exploration of the G.729A [4-1] and G.723.1 [4-2] speech codecs. The codecs were profiled using an internally developed simulator, based on the SimpleScalar toolset [4-3] to calculate the dynamic instruction count (the total number of instructions executed) at varying bands of granularity. It was then used to find where the processor was spending most of its time when executing these codecs. Using this information as the optimisation metric and by visual inspection of the speech coders code, custom instructions (both scalar as well as vector) were investigated, in an effort to minimise the performance metric without substantially increasing the silicon cost.

4.2 The SimpleScalar Toolset

The SimpleScalar toolset enables low-level system simulation of applications, providing detailed statistics of the computer system behaviour under an algorithmic load. The toolset has a range of execution-driven processor simulators including a flexible superscalar processor, which makes it ideal for use in this project for virtual architecture evaluation simulation. There are five different simulation suites provided with the SimpleScalar toolset. The first is known as ‘sim-fast’ and has the greatest execution speed running at 10+ Million Instructions per Second (MIPS) on a current generation Linux X86 workstation under nominal load. All timing measurements have been removed from this simulator which concentrates on modelling the architectural functionality of the processor. Moving up the complexity scale of the simplescalar simulators, the execution speed is decreased but more detailed statistics are available about the simulated processor. The last simulator in the package executes a more detailed view of the processor and is called ‘sim-outorder’. This is the most complicated simulator executing at approximately 1MIPS on a Linux X86 workstation. This simulator also allows significant customisation, altering processor and cache hierarchy including the use of out-of-order instruction issuing, branch prediction and multiple functional units.

Research projects have used the SimpleScalar toolset in order to study the performance results in [4-4, 4-5 and 4-6]. Each project used different parts of the toolset to aid its research ranging from memory activity, to obtain traces from simulating applications or to allow modifications to the toolset architecture to closely match the target processor. The semantics of the SimpleScalar ISA is based on the MIPS-IV ISA [4-7]. The toolset allows for easy alteration to
Architectural Results

the baseline microarchitecture and instructions can be easily added making it an extendable ISA. Using the simulator it is possible to measure the computational activity of a programme. For this research, it be can used to help to find "hotspots" of the speech algorithms – where the programme spends most of the time executing code. Using this information coupled with visual inspection of the code, SIMD and scalar style instructions can then be designed and the functional behaviour verified before adding them to the toolset ISA. When the codecs are simulated, the performance metric can then be evaluated and a choice is made as to whether to include the vector instruction in the ISA.

4.3 Profiling the Speech Coders

The speech coders were profiled using the SimpleScalar proposed toolset without any modification. Table 4-1 and table 4-2 show the unmodified instruction count for the G.723.1 and the G.729A respectively on the simulated processor for various workloads provided by the ITU-T. These results were used as a baseline to show the improvement brought by the accelerator by the reduction of the total instruction count.

Table 4-1: Unmodified G723.1 instruction count

<table>
<thead>
<tr>
<th>Workloads</th>
<th>Instruction Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoder</td>
<td></td>
</tr>
<tr>
<td>Rate 5.3 kbits/s</td>
<td>925,853,310</td>
</tr>
<tr>
<td>Rate 6.3 kbits/s</td>
<td>10,159,685,901</td>
</tr>
<tr>
<td>Mixed Rate</td>
<td>1,062,686,809</td>
</tr>
<tr>
<td>Decoder</td>
<td></td>
</tr>
<tr>
<td>Rate 6.3 kbits/s</td>
<td>680,067,420</td>
</tr>
<tr>
<td>Rate 5.3 kbits/s</td>
<td>90,359,906</td>
</tr>
<tr>
<td>Mixed Rate</td>
<td>90,305,750</td>
</tr>
<tr>
<td>Rate 6.3 kbits/s (e)</td>
<td>89,548,422</td>
</tr>
<tr>
<td>Rate 6.3 kbits/s (b)</td>
<td>9,093,869</td>
</tr>
</tbody>
</table>

Table 4-2: Unmodified G.729A instruction count

<table>
<thead>
<tr>
<th>Workloads</th>
<th>Workload Description</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoder</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alghthm</td>
<td>Conditional parts of the algorithm</td>
<td>62,613,675</td>
</tr>
<tr>
<td>Fixed</td>
<td>Fixed Codebook Search</td>
<td>213,961,885</td>
</tr>
<tr>
<td>Lsp</td>
<td>LSP Quantization</td>
<td>3,977,183,504</td>
</tr>
<tr>
<td>Pitch</td>
<td>Pitch Search</td>
<td>3,253,175,471</td>
</tr>
<tr>
<td>Tame</td>
<td>Taming Procedure</td>
<td>230,917,008</td>
</tr>
<tr>
<td>Decoder</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rate 6.3 kbits/s</td>
<td>680,067,420</td>
<td></td>
</tr>
<tr>
<td>Rate 5.3 kbits/s</td>
<td>90,359,906</td>
<td></td>
</tr>
<tr>
<td>Mixed Rate</td>
<td>90,305,750</td>
<td></td>
</tr>
<tr>
<td>Rate 6.3 kbits/s (e)</td>
<td>89,548,422</td>
<td></td>
</tr>
<tr>
<td>Rate 6.3 kbits/s (b)</td>
<td>9,093,869</td>
<td></td>
</tr>
</tbody>
</table>
Table 4-3 illustrates the ten most computationally intensive functions where the G.723.1 speech encoder, at 6.3 kbits/s, spends most of its time executing instructions i.e. the execution hotspots. The dynamic instruction count gives the total number of instructions that are executed by that function during operation and does not include the instruction count to calls to other functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Number of Calls</th>
<th>Dynamic Instruction Count</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find_Best</td>
<td>4408</td>
<td>1370009644</td>
<td>Fixed Codebook Search</td>
</tr>
<tr>
<td>Find_Acbk</td>
<td>2772</td>
<td>915225959</td>
<td>Adaptive Codebook Calculation</td>
</tr>
<tr>
<td>Estim_Pitch</td>
<td>1728</td>
<td>430602013</td>
<td>Open-loop pitch estimation</td>
</tr>
<tr>
<td>Lsp_Svq</td>
<td>926</td>
<td>141876220</td>
<td>Search for the LSP indices</td>
</tr>
<tr>
<td>Comp_Lpc</td>
<td>864</td>
<td>126386784</td>
<td>Computes the LPC filter coefficients</td>
</tr>
<tr>
<td>Upd_Ring</td>
<td>3456</td>
<td>98506368</td>
<td>Update memory of the filters</td>
</tr>
<tr>
<td>Sub_Ring</td>
<td>2772</td>
<td>78871716</td>
<td>Computes the zero-input response and target speech vector</td>
</tr>
<tr>
<td>Comp_Ir</td>
<td>2772</td>
<td>78048432</td>
<td>Computes the combined impulse response from the filters</td>
</tr>
<tr>
<td>Error_Wght</td>
<td>864</td>
<td>66604896</td>
<td>Implements the formant perceptual weighting filter</td>
</tr>
<tr>
<td>Decod_Acbk</td>
<td>6228</td>
<td>31267516</td>
<td>Computes the adaptive codebook contribution</td>
</tr>
</tbody>
</table>
The worst procedure is the \texttt{Find\_Best} function which performs the fixed codebook search, quantization and approximates the best possible residual signal. The next function, \texttt{Find\_Acbk}, performs the computation of the adaptive codebook contribution. These two procedures are expected to be the worst due to the complexity of the codebook search.

Within these functions, there are many data parallel loops executing DSP type operations and are found in the file \texttt{basop.c}. The code shown in figure 4-1 illustrates a multiply and accumulate data parallel loop during the fixed codebook search. This code multiples two array elements and adds the result to the accumulator for the entire subframe length (40 samples).

```
1) for ( i = 0 ; i < SubFrLen ; i ++ )
2) Acc0 = L\_mac( Acc0, OccPos[i], OccPos[i] );
```

Figure 4-1: Example of a Data Parallel Loop

Table 4-4 shows the top ten most executed DSP instructions with respect to their dynamic instruction count when executed on the SimpleScalar simulator. The DSP instruction which has the most impact to the dynamic instruction count is the 32-bit multiply and accumulate function followed by a 32-bit addition instruction (L\_mac). These instructions are counted as basic DSP operations.

<table>
<thead>
<tr>
<th>DSP Functions</th>
<th>Number of Calls</th>
<th>Dynamic Instruction Count</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>L_mac</td>
<td>153775312</td>
<td>2460404992</td>
<td>Long Multiply &amp; Accumulate</td>
</tr>
<tr>
<td>L_add</td>
<td>200172139</td>
<td>1941851871</td>
<td>Long Addition</td>
</tr>
<tr>
<td>L_shr</td>
<td>42534233</td>
<td>614668817</td>
<td>Long Shift Right</td>
</tr>
<tr>
<td>L_mult</td>
<td>42593996</td>
<td>511120752</td>
<td>Long Multiply</td>
</tr>
<tr>
<td>L_msu</td>
<td>21573705</td>
<td>345179280</td>
<td>Long Multiply &amp; Subtract</td>
</tr>
<tr>
<td>L_sub</td>
<td>23936234</td>
<td>219894706</td>
<td>Long Subtraction</td>
</tr>
<tr>
<td>L_shl</td>
<td>6270421</td>
<td>217893577</td>
<td>Long Shift Left</td>
</tr>
<tr>
<td>mult_r</td>
<td>3143127</td>
<td>81539187</td>
<td>Multiply with rounding</td>
</tr>
<tr>
<td>abs_s</td>
<td>5344001</td>
<td>55797485</td>
<td>Absolute</td>
</tr>
<tr>
<td>Round</td>
<td>6869733</td>
<td>54957864</td>
<td>Round</td>
</tr>
</tbody>
</table>

These DSP instructions can be efficiently implemented on a configurable vector processor as they mostly appear in data-parallel form. Performance of the main RISC CPU would be improved by reducing the number of instructions executed as the vector processor would be executing the instructions for a wider data range. This will inherently reduce the overall dynamic instruction count.
It is possible to simulate this on SimpleScalar by introducing the custom vector instructions and associated functional units to execute the DSP instructions. The toolset can be used in order to study the custom instruction effect on the algorithms workloads. This can be done by recording the total number of instructions executed for a range of different vector lengths of the coprocessor, it will then be possible to quantify the improvement the coprocessor has on the performance metric.

4.4 Programmer's Model

The proposed programmer's model for the coprocessor is depicted in figure 4-2. There are 16 vector registers (VR0-VR15), each consisting of a parametric number (VMAX) of scalar (16-bit) elements and referred to as the maximum vector length. There are two vector accumulators (VACCO, VACC1) consisting of VMAX/2 scalar elements (32-bits) and two vector mask registers of length VMAX bits. Finally, there are 16 32-bit scalar registers e.g. for a maximum vector length of 4, there would be 16 vector registers each being 64-bits wide.

Since the SimpleScalar toolset is an open source C based simulator, the vector architecture can be added easily into the toolset. Sim-fast has been altered to include the new architecture creating a new simulator known as sim-vector. The definition of the vector architecture within the toolset is depicted in figure 4-3.
Architectural Results

1) #define VLMAX 32
2) #define VECTOR_REGS 16
3) #define VACCUMULATORS 2
4) #define PRED_REGS 2
5) #define SCALAR_REGS 16 // 4 scalar regs-used as addr ptrs
6) typedef struct
7) {
8)   // Vector length register
9)   int VLEN;
10)  // Vector register file - 16-bits wide
11)  signed short int VRF[VECTOR_REGS][VLMAX];
12)  // Vector accumulators - 32-bits wide
13)  signed int VACC[VACCUMULATORS][VLMAX/2];
14)  // Predicate registers
15)  unsigned short int PRED_T[PRED_REGS][VLMAX];
16)  unsigned short int PRED_F[PRED_REGS][VLMAX];
17)  // Scalar registers
18)  signed int SRF[SCALAR_REGS];
19)  unsigned short int VV;
20) } vstateT;

Figure 4-3: X86 Coprocessor Definition

Line 1 defines the maximum vector length allowed, $VLMAX$, in this case is 32. This means that there are 32 elements each being 16-bits wide resulting in a vector length of $32 \times 16 = 512$-bits. Lines 4-12 defines the programmer's model as depicted in figure 4-2 indicating the number of registers and accumulators. The vector coprocessor state is encapsulated in a structure, vstateT, in lines 14-36.

4.5 Identifying Custom Instructions

This section explains in detail how the custom instructions are introduced in to the speech codecs. In order to run parts of the speech algorithms on the simulated vector coprocessor the
C code needed to be modified to include vector assembly instructions. By carefully studying the code using profiling (as in table 4-4) to identify "hot spots" and visual inspection it came apparent that the DSP emulation functions operate iteratively on blocks of data without dependencies. The main area of interest was to target these DSP loops by replacing the main body by custom vector assembly. Figure 4-4 depicts a section of code from the open loop pitch search function. During this function the pitch is estimated twice for every 20 ms frame. This is calculated using the perceptual weighted speech signal (Dpnt). The original code is depicted lines 2 and 3. Other segments of the code have been added.

1) #ifdef ORIGINAL
2)     for ( j = 0 ; j < 2*SubFrLen ; j ++ )
3)         Accl = L_mac( Accl, Dpnt[Pr+j], Dpnt[Pr+j] ) ;
4) #else
5) {
6)     Word16 *from = Dpnt + Pr;
7)     // Load Vector Length
8)     ldvlen_r(VLMAX);
9)     // Clear Accumulator 0
10)    vaccclr(0);
11)
12)    // Modulus Part
13)    for(j=0; j < ((2*SubFrLen) / VLMAX); j++)
14)    {
15)        // Load Dpnt into register one
16)        vldw(1,from);
17)        // Do even and odd L_mac
18)        vmace(0,1,1);
19)        vmaco(0,1,1);
20)    }
21)    // Increment Pointer
22)    from += VLMAX;
23) }
24)    // Remainder Part
25)    if((2*SubFrLen) % VLMAX)
26)    {
27)        // Load Remainder Vector Length
28)        ldvlen_r((2*SubFrLen) % VLMAX);
The original code executes a multiply and accumulate instruction across twice the subframe length, executing another 120 samples. This is an ideal data-parallel loop to introduce custom vector instructions. However, there are data dependencies in this calculation due to the accumulate part i.e. the variables require previous calculated values. This can be removed by manipulating the equation to add all the vector results together. Lines 1, 4 and 46 contain conditional statements and define which parts of the code to include at compile time. If the token “ORIGINAL” has been defined in the compile header file, then the original code on lines 2-3 will be compiled. If the token has not been included then lines 5-45 will be compiled. This switch gives the capability to turn off the custom instructions to see the overall enhancement that they can make. It also enables the use of debugging by turning off the instruction macros in certain files.

In the above example, the additional code added can be split into four separate segments. The first part is the setup and is depicted on lines 6-10. The first instruction creates a pointer to the first value in the data array. The second instruction loads the maximum vector length into the vector length register. The final instruction in this section loads the value zero to the accumulator setting it to a known value.

The next section of code performs the main calculation and can be seen on lines 12-24. This section loops until no more work can be done utilising the maximum vector length. The first instruction on line 16 loads data, from the address given by the pointer, into the coprocessor’s
register 1. Lines 19-20 performs the multiply accumulate calculation for both even and odd elements respectively. These instructions multiply the value in register 1 and then add the result to accumulator 0 across all vector elements. The final instruction on line 23 increments the pointer to the next vector data element for the next loop iteration.

The remainder part of the calculation is contained in lines 26-38 which have similar format to the previous segment. The main difference is that this part of the code is executed only once for a smaller data length. E.g. for a subframe length 60 the original code would perform 120 iterations. With a maximum vector length of 16, the modulus part of the code would only execute 7 times performing calculations on 112 samples utilizing the whole vector data length. The remaining 8 samples also need calculating are performed during this section. Line 30 loads this remainder vector length into the vector register informing the coprocessor not to use all of its data lanes.

The final section handles how the final result is to be stored back. Line 40 restores the vector register with the maximum value. Line 42 performs an add-reduce function. Since each of the results are currently split across all accumulator lanes, there needs to be one final 32-bit result to store back to the main processor. This instruction adds all accumulator lanes together producing one single result in the lowest element of the accumulator. Line 44 finally stores the value in the accumulator in element 0 back to the variable address.

Note, multiplication operations are performed in pairs of odd and even elements. This is due to the hardware limitation as the multiplication function takes two 16-bit elements and produces a 32-bit result, as shown in figure 4-5 [4-8]. Odd/even addressing is determined by the maximum vector length and is explained in more detail in chapter 5.

![Multiply Even Elements](image)

![Multiply Odd Elements](image)

Figure 4-5: Multiplying Even and Odd Elements
Figure 4-6 depicts another segment of code taken from the G.723.1 speech codec during the algebraic codebook search. The speech codec is exploring the optimum positions for the pulses used to construct the codebook vector within four nested loops.

```c
1) #ifdef ORIGINAL
2)   for(i=ip0, j=0; i<SubFrLen; i++, j++)
3)       y[i] = add(y[i], h[j]);
4) #else
5) {
6)   // Set Pointers
7)   Word16 *from1 = y +ip0;
8)   Word16 *from2 = h;
9)   Word16 *to = y+ip0;
10) int index;
11)  ldvlen_r(VLMAX);
12)  // Modulus Part
13)  for(index=0 ; index<(SubFrLen-ip0)/VLMAX; index++)
14)  {
15)   // Load Vector Registers
16)    vldw(1, from1);
17)    vldw(2, from2);
18)  
19)   // Add
20)    vitu_add_r(3, 1, 2);
21)  
22)   // Store Result
23)    vstw(3, to);
24)  
25)   // Increment Pointers
26)    from1 += VLMAX;
27)    from2 += VLMAX;
28)    to += VLMAX;
29)  }
30)  
31)  // Remainder Part
32)    if((SubFrLen-ip0)%VLMAX)
33)    {
34)    
35)    
36)
```
At first glance, the code looks like it is not a data-parallel loop due to data dependencies with two different incrementing variables. However, closer inspection shows that one of the loop variables has an offset, \( ip0 \). Custom instructions can replace this loop by adding the offset to the address of the variable \( y \). The way this loop is replaced with the custom instructions is similar to that discussed in the previous example. The main difference here is that this loop does not use the 32-bit accumulator as 16-bit calculations are all that is needed. Notice in lines 7 and 9, the offset has been applied to the pointers. The modulus part is similar to before with the exception on line 26 and 48. This instruction stores the value in the register back to the address indicated by the pointer.
The section of code in figure 4-7 is taken from the harmonic noise filter coefficients calculation. Lines 17-25 form part of the original speech codec, however they do not appear in data-parallel loops and cannot be replaced by vector instructions. Utilising the two lower lanes of the vector datapath, scalar assembly instructions can also be introduced into the speech coder algorithms. Lines 2-6 load the values needed for the calculations in the registers and lines 8-13 execute the DSP functions. The final instruction stores the original value back to the variable address. Note that the scalar instructions do not store the result for Acc1. The variable is not used after this point and does not need to be stored back saving on the number of instructions executed.

Similar scalar instructions have been designed and functionally verified in C before being added into the SimpleScalar ISA. These instructions can be enabled by defining the token METHOD2 in the compile header file. With the addition of the custom scalar instructions, the original code has been optimised, taking advantage of the vector coprocessor design. These scalar instructions were introduced to the segments of code that perform DSP operations outside loops and iterate the loops that cannot utilise data-parallelism due to data dependences.

4.6 Implementation of the Custom Instructions

Once an instruction had been identified, a C macro was created to represent the instruction functional behaviour. This enabled quick verification to validate the custom instructions at a functional behavioural level giving an instruction-accurate model of the coprocessor before introducing the instructions into the SimpleScalar toolset. It is important that the code is re-
written in such a manner that the end result is the same with or without the modified code so that the output of the modified code line can be directly compared to that of the original. This allows engineers to quickly verify the functional behaviour and identify any problems before implementing the custom instruction into the SimpleScalar ISA.

The custom extended instructions definitions are found in the file `x86_visa.h` header file located in the source directory of the ITU codecs and are implemented as C-macros. Each instruction macro contains two parts as illustrated in figure 4-8 with the multiply and accumulate even (vmace) instruction. During compile time, the user can select which mode the workload is executed in. The file `compile.h` contains a switch which enable lines 2-25 to run in Linux X86 mode or can enable lines 28-40 to run on sim-vector mode.

```c
1) #ifdef X86
2) #define vmace(vacc,vrs1,vrs2) \ 
3) {{
4) extern vstateT vstate;
5) extern Flag Overflow;
6) int index;
7) stats_start;
8) if ((vacc <0) || (vacc > VACCUMULATORS))
9) printf("ERROR: Attempt to reference non-existent \ 
10) VACC[%d]\n",vacc);
11) else if ((vrs1 >= VECTOR_REGS) || (vrs2 >= VECTOR_REGS))
12) printf("ERROR: Attempt to reference non-existent VRF[%d] or \ 
13) VRF[%d]\n",vrs1,vrs2);
14) else \ 
15) {{
16) for (index = 0; index < vstate.VLEN ; index +=2)\ 
17) {{
18) vstate.VACC[vacc][index/2]=\ my_L_mac(index/2,\ 
19) vstate.VACC[vacc][index/2], \ 
20) vstate.VRF[vrs1][index], \ 
21) vstate.VRF[vrs2][index]);\ 
22) vstate.OVERFLOW|=vstate.V32[index];\ 
23) }}
24) }}\ 
25) }};\ 
26) #else\ 
27) #ifndef SS
```
Architectural Results

28) // simplescalar
29) #define vmace(vacc,vrs1,vrs2) \
30) {
31)       asm volatile (".word 0x00010000");
32)       asm volatile (".word \
33)            6 << 29  \* EXT_OPCODE */\
34)            2 << 25 \* CATEGORY */\
35)            1 << 20 \* OPCODE */\
36)            "#vacc"<< 15 \* VRD = VACC */\
37)            "#vrs1"<< 10 \* VRS1 = 1st vector operand */\
38)            "#vrs2"<< 5 \* VRS2 = 2nd vector operand */\
39)            "  "); \* RS2 = HOST REG */ \
40)       });
41) #endif
42) #endif

Figure 4-8: X86_visa.h code example

Lines 1-26 from figure 4-8 were used to test the functionality of the instructions under Linux X86 to verify that they do not have any side effect on the coded bit stream. This enables easy verification of the added scalar and vector instructions. Lines 16-23 of figure 4-8 show the code from the multiply and accumulate even instruction. This calls the _mLL_mac function for each even vector element, performing the multiply and accumulate calculation. This segment of code specifies a data-parallel instruction which will be included in the vector coprocessor.

Once the functional behaviour of the C Macro has been verified it can be introduced into the SimpleScalar toolset. The second part of the code on lines 27-41 in figure 4-8 is the opcode definition of the SimpleScalar vector instruction and is encoded into a second 32-bit word using the assembly volatile statement. The 12 MSB of the opcode defines the instructions. This is split into 3 different parts containing the opcode, bits 20-24, the category, bits 25-28 and bits 29-31 the extended opcode. The lower bits of the instruction opcode contain the register details and immediate data if needed.
Sim-vector decodes the instruction in the vector.def file of the toolset. This file is divided into 3 levels of switch statements. The first level being the category switch, the second level the opcode switch and the final level the extended opcode switch.

1) case 6:\n2) {\n3) /* VMACE */\n4) extern vstateT vstate;\n5) int index;\n6) for (index = 0; index < vstate.VLEN ; index +=2)\n7) {\n8) vstate.VACC[RD_ADDR][index/2] = \n9) my_L_mac(index/2,vstate.VACC[RD_ADDR][index/2], \n10) vstate.VRF[RS1_ADDR][index], \n11) vstate.VRF[RS2_ADDR][index]);\n12) }\n13) break; \n14) }\n
Figure 4-9: SimpleScalar vector.def file

Figure 4-9 shows the C-description of the multiply and accumulate even (vmace) instruction and is similar to the code in figure 4-8. The exception to this is the way the registers are decoded. Register details have been decoded from the opcode instruction earlier. Lines 6-12 execute the vector instruction. This specifies the state transformation performed by that opcode in an iterative way, as each iteration of the loop is effectively a vector datapath lane. When the SimpleScalar toolset is executing the workload, only the extended vector instruction is added to the instruction count, as the execution of the code in figure 4-9 has been fused into a single instruction.
4.6.1 SimpleScalar extended Vector Instructions

Table 4-5 shows the 43 proposed vector instructions that have been added to the SimpleScalar ISA. The instructions have been encoded into a 32 bit word address space. The vector coprocessor has a 16-bit register file containing VLMAX elements as per the programmer’s model in section 4.4.

An address or immediate value included within instructions are transferred from the main CPU registers to the coprocessor. Therefore, in the table, the instruction opcode requires the “Host Reg” number. The instruction macros use register 10 to transfer addresses and immediate values from the main CPU to the coprocessor.

4.6.2 SimpleScalar extended Scalar Instructions

Table 4-6 shows the custom Scalar Instructions introduced in the SimpleScalar ISA. There are 32 instructions in total. They perform similar calculations to that of the vector instructions and in the proposed microprocessor silicon implementation, make use of the lower vector lanes. The scalar side of the coprocessor has a 32-bit register file.
4.7 Architecture Exploration Results

As previously described, the custom instructions were first introduced using C Macros. These were validated by comparing the output bit streams of the modified and original codeline to ensure compliance with the ITU-T workloads. The custom instructions were then introduced into the SimpleScalar toolset. Once complete, the overall optimisation benefit could be explored using the toolset in order to compare the dynamic instruction count. Simulations were run for all ITU-T workloads and for vector lengths of up to 128 16-bit elements. The compile switch `#define SS` was enabled in order to execute the custom instructions in SimpleScalar ISA.

The results shown are split into two groups. The first shows the performance results for enabling only vector ISA in section 4.7.1. The second group in 4.7.2 shows the full performance optimisation by using the compile switch `#define METHOD2` enabling the scalar instructions.

4.7.1 Vector Profiling Results

Figure 4-10 and figure 4-11 show the normalised results for the G.723.1 Encoder and G.729A Encoder respectively with only the vector instructions introduced.

![Graph](image.png)

Figure 4-10: G.723.1 Encoder results with Vector Instructions only
It is clear from the results that there is significant improvement in the dynamic instruction count when the custom scalar and vector ISA are used. The majority of improvements are achieved over shorter vector lengths owing to the removal of loop overheads needed in scalar processors as more work is carried out in the vector coprocessor. This occurs between the vector lengths of 2 to 16 corresponding to data widths of 32 to 256 bits wide. The dynamic instruction count levels off with a reduction of around 60% with a vector length of 16 and shows no further improvement for larger vector lengths. Minima can be seen at vector lengths of 32, 64 and 128. This occurs when the vector processor datapaths are optimally used due to the maximum vector length being equal to the speech subframe boundaries. At other vector lengths a slight overhead is needed to determine the number of vector lanes for the operation, i.e. in figure 4-4, the minima occurs when all operations are executed in the modulus part of the assembly code so there is no overhead executing the remainder part. Since there is little improvement at these points they can be ignored.

Similar results were obtained for the Decoders of the G.723.1 and G.729A speech codecs and can be seen in Appendix A - G.723.1 Results and Appendix B - G.729A Results.

4.7.2 Vector and Scalar Profiling Results

Figure 4-12 and figure 4-13 show the normalised improvement of the two speech codecs when the Vector and Scalar instructions were applied.
Considerable improvement can be seen with the introduction of the scalar instructions. There is now a further 25% decrease in the dynamic instruction count taking the overall reduction of the speech codecs with the vector and scalar instructions to 85% with no significant improvement after a vector length of 16 - 256 bit wide configuration.
It is clear that with the introduction of the custom vector and scalar instructions, via architectural exploration of the G.723.1 and G.729A speech codecs, the coprocessor can improve the instruction bandwidth. Since there is little improvement over the vector length of 16, this allows the vector microprocessor design to be restricted to a practical implementation. Vector lengths above this do not give a further performance increase and hence there is little point in considering designs above this length due to large silicon overheads because of large datapaths within the coprocessor and the need for long cache refills, which will be explored in the next chapter when the Load/Store Unit (LSU) is designed.

### 4.8 Function Level Architecture Results

This section explores some of the architecture level results with regards to the functionality of the G.723.1 speech codec, as described in chapter 2.4.2. These results can be used to see the functions of the speech codec where the extended scalar and vector ISA has made improvements or has no effect to the total number of instructions executed. Table 4-7 shows the top ten most compute-intensive functions for the G.723.1 encoder at 6.3 kbits/s as originally illustrated in table 4-3 but with two extra columns.

**Table 4-7: Profiling functions for G.723.1 encoder at 6.3 kbits/s with coprocessor instructions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Number of Calls</th>
<th>Instruction Count Before</th>
<th>Instruction Count After</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find_Best</td>
<td>4408</td>
<td>467584854</td>
<td>1370009644</td>
<td>65.87%</td>
</tr>
<tr>
<td>Find_Acbk</td>
<td>2772</td>
<td>746775844</td>
<td>915225959</td>
<td>18.41%</td>
</tr>
<tr>
<td>Estim_Pitch</td>
<td>1728</td>
<td>33534409</td>
<td>430602013</td>
<td>92.21%</td>
</tr>
<tr>
<td>Lsp_Svq</td>
<td>926</td>
<td>99670992</td>
<td>141876220</td>
<td>29.75%</td>
</tr>
<tr>
<td>Comp_Lpc</td>
<td>864</td>
<td>15303168</td>
<td>126386784</td>
<td>87.89%</td>
</tr>
<tr>
<td>Upd_Ring</td>
<td>3456</td>
<td>70194816</td>
<td>98506368</td>
<td>28.74%</td>
</tr>
<tr>
<td>Sub_Ring</td>
<td>2772</td>
<td>58760856</td>
<td>78871716</td>
<td>25.50%</td>
</tr>
<tr>
<td>Comp_Ir</td>
<td>2772</td>
<td>59057460</td>
<td>78048432</td>
<td>24.33%</td>
</tr>
<tr>
<td>Error_Wght</td>
<td>864</td>
<td>61003584</td>
<td>66604896</td>
<td>8.41%</td>
</tr>
<tr>
<td>Decod_Acbk</td>
<td>6228</td>
<td>20032204</td>
<td>31267516</td>
<td>35.93%</td>
</tr>
</tbody>
</table>

One column showing the dynamic instruction count when the coprocessor instructions are enabled and the other showing the overall percentage improvement it has on that function. For the Find_Best function there is a 66% improvement in instruction count however there is only an 18% improvement in the Find_Acbk function. This is due to the codebook computation having nested data parallel loops. It is only possible to introduce coprocessor
instructions in the inner-most loop due to the adjacent data not being independent in the lower loops. Therefore there is only a small improvement in this function.

```
1)    for ( i = ClPitchOrd-2 ; i >= 0 ; i -- )
2)    {
3)        FltBuf[i][0] = mult_r( RezBuf[i], (Word16) 0x2000 ) ;
4)        for ( j = 1 ; j < SubPrLen ; j ++ ) {
5)            Acc0 = L_deposit_h( FltBuf[i+1][j-1] ) ;
6)            Acc0 = L_mac( Acc0, RezBuf[i], ImpResp[j] ) ;
7)            FltBuf[i][j] = round( Acc0 ) ;
8)        }
9)    }
```

**Figure 4-14: Find_Aebk code loops**

Example code from the Find_Aebk function in the G.723.1 speech codec is depicted in figure 4-14 and shows two loops on line 1 and on line 4. Lines 5-7 are in a nested loop and will be executed over and over again, increasing the instruction count. Problems also arise when the custom vector instructions can not be added to the inner loop. This is due to the nature of the Filter Buffer on line 5. The calculation requires the next element and the previous element in the 2-dimensiononal buffer array. Vector processors require data to be adjacent and as a result this loop can not be accelerated with the custom vector instructions. There are similar nested loops within the codebook search and thus the dynamic instruction count could not be improved. However it is possible to introduce scalar instructions and utilise the lower lanes of the vector architecture to optimise the code. These scalar instructions do not have the similar impact to the overall reduction in dynamic instructions as the vector instructions have.

Figure 4-15 shows the architecture level normalised results with regards to the algebraic codebook search when the vector and scalar accelerator is applied.
Notice that the graph in figure 4-15 only shows results for two workloads: Mixed Rate and 5.3 kbits/s. This is because the algebraic codebook search is only done for lower bit rates. These results show an improvement of around 60% in the dynamic instruction count of the algebraic codebook search. The majority of the improvement happens between the vector lengths of 2 to 32. The dynamic instruction count also decreases at a vector length of 64. With the introduction of the accelerator, the codebook search has been improved by reducing the number of total instructions executed.

Figure 4-16: G.723.1 Encoder - Estimate Pitch
Figure 4-17: G.723.1 Encoder - Harmonic Noise Filter

Figure 4-16 and figure 4-17 show the normalised results for the architectural simulation during the pitch estimation and the harmonic noise filter sections of the speech codec respectively. The results show a significant improvement of over 90% reduction in the total number of instructions executed. The different workloads for pitch estimation and harmonic noise filter functions follow the same curve because the workloads use these functions equally.

Figure 2-19 shows the processing flow of the G.723.1 encoder and section 2.4.2.7 describes where the pitch estimation and the harmonic noise filter are applied. It is clear that there is significant reduction in instruction count within the functions of the codec; similar results can be seen within in other sections of the G.723.1 and the G.729A too.

4.9 Conclusion

This chapter described the process used to identify, implement and explore the benefits of adding custom instructions to the two speech coders, G.729A and G.723.1. These instructions were identified by profiling the coders using the SimpleScalar toolset to identify "hot spots" (places where the speech coder implementation spends most of its time executing instructions) and by visual observations. Basic DSP operations were mainly found in data parallel loops and were targeted by adding custom SIMD style and scalar instruction, outlined in sections 4.6.1 and 4.6.2. They initially took the form of C Macros to verify the functionality and the program execution order using Linux X86 by comparing the bitstream output from the modified and unmodified code.
Once the custom instructions functionality was verified, they were introduced into the SimpleScalar toolset. The speech coder's workloads were then profiled over a range of vector lengths to see the enhancement the custom instructions have made. The results indicate a significant improvement in the number of instructions executed, showing a reduction in the execution of the dynamic instruction count of around 60% for both speech coders when the vector instructions were introduced and a further 25% reduction in dynamic instruction count when the scalar instructions were applied. The significant points of interest in the microprocessor implementation lie in the area between 64-bit (vector length of 4) and 256-bit (vector length of 16) wide configurations.

These results show the potential benefit of applying custom instructions and having associated coprocessor vector functional units. From this, the microarchitecture can be designed and attached to a generic RISC CPU. This is explained in more detail in the next chapter.

### 4.10 References

[4-2] ITU Recommendation G.723.1, 'Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s', 3/96  
http://www.simplescalar.com  
5 VECTOR PROCESSOR

5.1 Introduction

This section presents the overall microarchitecture of the system, including the Leon3 RISC CPU, the design of the vector coprocessor and the associated infrastructure. The following sections present a brief background into Load/Store Units (LSU) including previous research and descriptions on caches. Section 5.6 presents a detailed explanation of the LSU designed for the vector coprocessor. The final two sections present the clock cycle and power results respectively. Test vectors were obtained from the architectural exploration simulation and were applied to the LSU. To obtain the power results, the VHDL model was synthesised using Synopsys Design Compiler [5-1], placed and routed using Cadence SoC Encounter [5-2]. Finally, the statistical power results are obtained using Synopsys again.

5.2 The Leon3 System

The Leon3 processor core [5-3] is a synthesizable VHDL model of a 32-bit RISC processor which is compliant to the SPARC (Scalable Processor ARCHitecture) V8 architecture [5-4]. The processor is highly configurable allowing designers to select and customise blocks to optimize the performance, power consumption, area and cost of the processor for a specific application. The processor core is interfaced using the Advanced Microcontroller Bus Architecture (AMBA) -2.0 AMBA High-performance Bus (AHB) [5-5]. The processor and associated IP are provided in full source code under the General Public License (GPL) open-source licences. This allows ease for customising the whole system. The Leon3 system is shown in figure 5-1 and also shows the configurable vector coprocessor connected to the Leon3’s Integer Unit via a custom port. The vector coprocessor is also a master on the AHB which allows communication between the coprocessor’s load/store unit and data stored in memory.
The Leon3 system has a 7-stage pipeline with a Harvard Architecture where the processor has a separate instruction and data bus and cache. The integer pipeline of the processor is shown in figure 5-2 with the vector coprocessor custom ports.
The Leon3 processor issues one instruction which flows through the 7-stage pipeline:

- **Fetch** — Instructions are fetched from the instruction cache and are committed at the end of this stage.
- **Decode** — The instructions are decoded here and the Branch target addresses are also calculated.
- **Register Access** — Operands are read from the register files or from a bypass.
- **Execute** — All operations are performed here, ALU, Shift etc. If there are any Load/Store accesses for the Leon3, addresses are generated.
- **Memory** — The data cache is accessed during this stage. Here data is either loaded from or stored to external memory.
- **Exception** — Any traps and interrupts are evaluated during this stage, prior to committing processor state.
- **Write** — The results calculated in the execute stage or cache operations are written back to the register file.

If there is a coprocessor instruction, it is identified during the Decode stage of the Leon3 processor. This is pipelined via a coprocessor interface to the vector decode stage (VDEC). The instruction is further decoded giving the control signals later on in the pipeline. The addresses for the scalar and vector registers are extracted from the instruction and the register files are accessed. The next stage of the vector pipeline is Vector Register Access (VREG) where the operands are produced by the register files. Any move data instruction from the Leon3’s register files are inserted into the vector processor pipeline during this stage. The operands are further processed in case a bypass is required or an intermediate value is necessary. The final operands are pipelined into the input registers of the two-deep vector execution pipeline or to the LSU. During the first stage of the vector execution pipeline (VDP1), the arithmetical, shift and miscellaneous operations take place as well as the multiplication of the multiply-add and multiply-sub instructions. The second stage of the vector execution pipeline (VDP2) performs the addition/subtraction part of the multiply-add/multiply-sub instructions. It also performs saturation or reduction. The result from the LSU is also returned during this stage. The result is multiplexed and returned to the register files. In the case where the vector coprocessor is executing a scalar instruction, the scalar operands are pipelined down the same vector datapath. This mechanism reuses the lower two vector lanes (32-bits) of the existing vector datapath instead of implementing a dedicated scalar datapath, resulting in reduced silicon area and control logic overhead and at the same time, decreasing the verification effort.

### 5.3 Coprocessor Instructions

This section presents the custom instructions introduced into the ISA. In total there are 80 custom instructions. Each instruction has been given a unique opcode. The opcodes have been encoded into 7 bits, giving $2^7 = 128$ possible instructions. 6 bits would only give 64 instruction and therefore not enough. The remaining lower bits have been left for the operands. Although each instruction has been given a sequential unique opcode due to time requirements, more care could have been taken on choosing which instruction is allocated to which opcode. Undoubtedly, if this design was continue, further instructions could be identified in the future. Reserving opcode space at an early stage for future use can be beneficial.
5.3.1 Vector Instructions

Table 5-1 shows the proposed forty-three vector instructions that will be implemented on the coprocessor replacing the codec’s DSP functions. The table also shows the proposed opcode extension.

Bits 31 down to 22 of the instruction opcode are reserved as stated in the SPARC user manual [5-6] for extending the ISA and are executed as “unimplemented instructions”. Including scalar instructions, the coprocessor has 80 different instructions it can implement. Therefore 7 bits \(2^7=128\) instructions in the address field are required for the coprocessor opcode. Bits 21 down to 15 contain this information.

The first instruction, `Idvlen_r`, loads the vector length register with a value up to the maximum vector length. The next four instructions are load/store instructions. These instructions Load/Store data in the vector registers from the address in the scalar register. Instructions six to forty-three contain the different DSP calculations that the speech codecs compute. For opcode encoding in the address field, bits 14 down to 10 contain the destination information, operand one in bits 9 down to 5 and operand two in bits 4 down to 0.

5.3.2 Scalar Instructions

Table 5-2 shows the proposed scalar instructions implemented on the coprocessor. As with the vector instructions, bits 31 down to 22 are reserved and are used to implement the coprocessor instructions. Bits 21 down to 15 contain the opcode whereas bits 14 down to 0 contain the operand information.

The first four scalar instructions are load/store instructions and are similar to their vector counterparts. These instructions load/store data from a memory address given by the scalar register. Instructions five to thirty-two are the DSP calculations, similar to the vector instructions. The final four instructions created were not explored in section 4.6.1. These instructions move data between the coprocessor scalar/vector register file and the RISC CPU register file. For a vector move instruction, a vector element needs to be supplied in the operand field. These were not included in the original instruction architecture as they were built into the instruction macros by using the main CPU registers. The final instruction is “No Operation” i.e. the coprocessor has no work to do.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Operands</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>000000</td>
<td>w</td>
<td>w</td>
<td>佥</td>
</tr>
<tr>
<td>4</td>
<td>000000</td>
<td>w</td>
<td>w</td>
<td>佥</td>
</tr>
<tr>
<td>5</td>
<td>000000</td>
<td>w</td>
<td>w</td>
<td>佥</td>
</tr>
<tr>
<td>6</td>
<td>000000</td>
<td>w</td>
<td>w</td>
<td>佥</td>
</tr>
<tr>
<td>7</td>
<td>000000</td>
<td>w</td>
<td>w</td>
<td>佥</td>
</tr>
<tr>
<td>8</td>
<td>000000</td>
<td>w</td>
<td>w</td>
<td>佥</td>
</tr>
<tr>
<td>9</td>
<td>000000</td>
<td>w</td>
<td>w</td>
<td>佥</td>
</tr>
<tr>
<td>10</td>
<td>000000</td>
<td>w</td>
<td>w</td>
<td>佥</td>
</tr>
<tr>
<td>11</td>
<td>000000</td>
<td>w</td>
<td>w</td>
<td>佥</td>
</tr>
<tr>
<td>12</td>
<td>000000</td>
<td>w</td>
<td>w</td>
<td>佥</td>
</tr>
<tr>
<td>13</td>
<td>000000</td>
<td>w</td>
<td>w</td>
<td>佥</td>
</tr>
<tr>
<td>14</td>
<td>000000</td>
<td>w</td>
<td>w</td>
<td>佥</td>
</tr>
<tr>
<td>15</td>
<td>000000</td>
<td>w</td>
<td>w</td>
<td>佥</td>
</tr>
<tr>
<td>16</td>
<td>000000</td>
<td>w</td>
<td>w</td>
<td>佥</td>
</tr>
<tr>
<td>17</td>
<td>000000</td>
<td>w</td>
<td>w</td>
<td>佥</td>
</tr>
<tr>
<td>18</td>
<td>000000</td>
<td>w</td>
<td>w</td>
<td>佥</td>
</tr>
<tr>
<td>19</td>
<td>000000</td>
<td>w</td>
<td>w</td>
<td>佥</td>
</tr>
<tr>
<td>20</td>
<td>000000</td>
<td>w</td>
<td>w</td>
<td>佥</td>
</tr>
<tr>
<td>21</td>
<td>000000</td>
<td>w</td>
<td>w</td>
<td>佥</td>
</tr>
<tr>
<td>Instruction</td>
<td>Opcode</td>
<td>Operands</td>
<td>Instruction Format</td>
<td>Comment</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>----------</td>
<td>-------------------</td>
<td>---------</td>
</tr>
<tr>
<td>1 m2ahix</td>
<td>0100111</td>
<td>ssr1</td>
<td>m2ahix</td>
<td>Load 16 bit scalar value from address</td>
</tr>
<tr>
<td>2 m2ahix</td>
<td>0101100</td>
<td>ssr1</td>
<td>m2ahix</td>
<td>Load 16 bit scalar value from address</td>
</tr>
<tr>
<td>3 m2ahix</td>
<td>0101101</td>
<td>ssr2</td>
<td>m2ahix</td>
<td>Store 16 bit scalar value to address</td>
</tr>
<tr>
<td>4 m2ahix</td>
<td>0101110</td>
<td>ssr2</td>
<td>m2ahix</td>
<td>Store 16 bit scalar value to address</td>
</tr>
<tr>
<td>5 m2ahix</td>
<td>0101111</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>6 m2ahix</td>
<td>0101000</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>7 m2ahix</td>
<td>0101001</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>8 m2ahix</td>
<td>0101010</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>9 m2ahix</td>
<td>0101011</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>10 m2ahix</td>
<td>0101000</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>11 m2ahix</td>
<td>0101001</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>12 m2ahix</td>
<td>0101010</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>13 m2ahix</td>
<td>0101011</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>14 m2ahix</td>
<td>0101100</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>15 m2ahix</td>
<td>0101101</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>16 m2ahix</td>
<td>0101110</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>17 m2ahix</td>
<td>0101111</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>18 m2ahix</td>
<td>0110000</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>19 m2ahix</td>
<td>0110001</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>20 m2ahix</td>
<td>0110010</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>21 m2ahix</td>
<td>0110011</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>22 m2ahix</td>
<td>0110100</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>23 m2ahix</td>
<td>0110101</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>24 m2ahix</td>
<td>0110110</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>25 m2ahix</td>
<td>0110111</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>26 m2ahix</td>
<td>0111000</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>27 m2ahix</td>
<td>0111001</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>28 m2ahix</td>
<td>0111010</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>29 m2ahix</td>
<td>0111011</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>30 m2ahix</td>
<td>0111100</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>31 m2ahix</td>
<td>0111101</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>32 m2ahix</td>
<td>0111110</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
<tr>
<td>33 m2ahix</td>
<td>0111111</td>
<td>ssr1, ssr2</td>
<td>m2ahix</td>
<td>Long add scalar negative</td>
</tr>
</tbody>
</table>
5.4 Load/Store Units Background

Advances in Very Large Scale Integration (VLSI) design have created a 'gap' between CPU speed versus memory speed. Processors often have small faster on-chip memories, called a caches which are explained in more detail in the next section, where a LSU is needed to obtain data from a slower off-chip DRAM for the on-chip memory. CPUs can issue hundreds of instructions in the time it can take to fetch a piece of data into on-chip memory. This low memory bandwidth accompanied with high latency can cause bottlenecks. Instructions can not be completed until the load is complete. Designers often use different techniques to hide this latency by designing complicated memory systems which incorporate expensive logic. These include enlarging the cache and hence lowering the cache miss rate by decreasing the number of accesses to the memory system, or using a non-blocking cache which is able to return data while the cache is under a miss [5-7]. However this second option leads to increased complexity. Different techniques include speculative loads [5-8], pre-execute loads [5-9], out-of-order architectures [5-10], multithreading [5-11], multi-level caches [5-12] and victim caches [5-13].

There has been a considerable amount of research into memory systems exploring different cache configurations and LSU architectures to reduce the latency or tolerate the slow speed of memories. [5-14] reports bandwidth limitations in memory systems and that memory will soon reside in with the processor silicon die as the penalty to communicate with off-chip memory will become too expensive. This has been investigated in [5-15] which reports a dramatic improvement on the transfer of data as they can replace a whole cache-line at a time. They also present a Variable Line-Size Cache architecture where the size of the cache line can change according to the characteristics of the application program. This architecture design provides the effect of pre-fetching and is good for programs with spatial locality references. However, the performance of the system can worsen if there are frequent cache misses due to space limitations of the cache. In [5-16] they explore different cache configurations and architectures to find the best simulated processor performance for a range of benchmarks, including multiple ports and on-chip DRAM; however, the resulting configuration is too dear to implement.

A victim cache is a small cache which holds data that has been evicted from registers. [5-17] presents a small victim cache for vector register file. This study has shown the performance improvement of vector processors when reducing the number of accesses to the memory system. Vector processors are capable of tolerating memory latency as they initiate a memory transfer over a large number of operations [5-18]. With a scalar miss, the LSU tends to access
memory once or twice. A vector miss accesses memory a few times; this is because a vector instruction is effectively a collection of scalar instructions.

On-chip memory can be expensive and to obtain the high bandwidth required for vector machines a decoupling technique can be employed [5-19]. A scalar unit runs ahead and queues instructions to be executed on the vector units. A vector refill unit pre-executes any memory instructions. In [5-20] the authors report limitations in vector machines and design a scalable vector processor using the decoupling method, accessing off-chip memory directly without the need for a cache with a small performance loss.

5.5 An Overview of a Cache

A cache is a small but fast area of memory in close proximity to the processor that contains the most recently accessed pieces of data/instructions. A cache is needed as the time it takes for data to be retrieved from memory is very slow compared to the time it takes a processor to execute an instruction, which is typically every clock cycle. It can take many clock cycles to access memory therefore stall cycles can occur while the processor waits for the required data/instructions to be retrieved. Programs tend to reuse data and instructions they have used recently and have to access memory frequently. Caches are a way of hiding this memory latency as they are faster; decreasing the time it takes to move data to the processor. However, cache memories are several times more expensive than Dynamic RAM (DRAM) and can only hold small subsets of the main memory. If a processor accesses a small part in memory, the cache memory will load the region that the data is located in, allowing the processor to access that memory region faster.

5.5.1 Cache Terms

- **Cache Hit** – The cache contains the required piece of data.
- **Cache Miss** – The cache does not contain the requested piece of data.

The cache is a copy of the main memory. It is important that the cache always reflects what is in main memory. To maintain cache coherency, the following terms are used:

- **Snoop** – The cache memory monitors the address bus. This allows the cache to see transactions that are accessing memory that are also contained in the cache.
- **Snarf** – The cache memory takes the data on the data bus and updates itself. The cache is said to have “snarfed the data”. This is expensive but helps maintain cache coherency. This is mainly used in multiprocessor designs where data in memory can be modified by other processors.
- **Dirty Data** – Data that is modified in the cache and that is not written back.
• Stale Data – Data that is modified in main memory but not the cache.

5.5.2 Cache System

The most common cache system is the "Write Through" design [5-7] as shown in figure 5-3.

There are three main units in a cache system:

• Data RAM – The memory block that contains the stored data. The size of the Data RAM determines the size of the cache.

• TAG RAM – A small piece of memory that contains the addresses of the data that is stored in the Data RAM.

• Cache Controller – This is responsible for checking to see if the memory access is a cache hit or miss, updating the TAG RAM and SRAM, implementing the write policies and performing snoops and snarfs.

![Figure 5-3: Write Through Cache](image)

The cache unit sits between the CPU and the main memory. For the CPU to access the memory, it must go through the cache unit. When the CPU starts a read memory access, the cache checks to see if the data is located in the cache memory. If there is a cache hit, the data is returned without accessing the slower memory via the system bus. If there is a cache miss, the cache unit passes the address onto the system bus. The main memory responds with the required data and the cache unit passes it back to the main CPU as well as updating the data in cache's memory so next time the CPU requests this piece of data it will be a cache hit.

There are two different write policies a cache design can have:

• Write-Back – The cache acts as a buffer. When the CPU starts a write access, the cache will accept the data on behalf of the main memory so the processor can carry on its tasks. The cache unit will write the data back to main memory whenever the
data block can be scheduled on the system bus. If there is also a cache hit, the cache unit updates the cache memory.

- Write-Through – The main CPU writes through the cache unit and will have to wait to gain access to the system bus and for the memory transaction to complete. The cache also updates its memory if there is cache hit. This write policy is less complex than the write-back policy.

5.5.3 Cache Architecture

Figure 5-4 shows two different cache architectures. Diagram A depicts a parallel cache configuration where the TAG is checked in parallel to the data being retrieved. The data is retrieved in one clock cycle. In diagram B, the design is in a serial or a cascade configuration where the address is checked in one cycle and data is retrieved in the following cycle.

Both architectures have advantages and disadvantages. The parallel configuration design is simpler than the serial configuration, and data is retrieved in just one cycle. The serial configuration however checks the TAG in one cycle and hence knows exactly where the data is stored in the Data RAM. This can then power up only the parts of the Data RAM that are required, lowering the power consumption of the cache. Therefore, what is lost in performance is gained in saving power.

5.5.4 Cache Structure

For a cache to work, the main memory is divided into equal sections called Cache Pages. The size of the page is determined by the size of the cache memory and its scheme as discussed in the next section.
Cache pages are broken down further into cache lines, as shown in figure 5-5, and are determined by the processor (16-bit, 32-bit etc.) and the scheme.

### 5.5.5 Direct Mapped and Set Associative Schemes

Figure 5-6 shows a Direct Mapped cache which is also called a one-way set associative cache. The main memory is split into cache pages and cache lines as before. A direct mapped scheme can only store a specific line in memory to the corresponding line within the cache memory. E.g. Line 0 of any page in figure 5-6 can be stored in Line 0 of the cache memory. So if Line 0 of Page 0 is currently stored in cache memory and the processor requires data from Line 0 of Page 1 then the cache system will need to replace the data currently stored with the data that is needed. Note that this does not affect the data stored on the Line 1, which could be data from Page 3.
A set associative cache is similar to a direct mapped cache scheme. This scheme works by splitting up the SRAM into equal parts called cache ways. The cache page size is equal to the size of cache way. Figure 5-7 shows a two-way set associative cache scheme. Each way is treated as a direct mapped cache i.e. Line 0 of any page could be located in Line 0 of Way 0 or Way 1. This means two lines of memory can now be stored in the cache at any time reducing the number of times the lines have to be replaced. The performance of a set associative scheme is higher than a direct mapped scheme because it is more flexible [5-21].

5.5.6 Cache Replacement Policies

For a set associative cache, a replacement policy decides which cache way to replace on a cache refill. There are four commonly used replacement policies:

- First in, First Out (FIFO) – This policy is also known as Round Robin. In this policy, the line that was first written into is the first to be replaced i.e. for a cache with a 4 ways and concentrating on Line 0, the cache would first replace the line in Way 0, then Way 1, Way 2 and then Way 3. Because Way 0 was the first to be written, the cache system would then replace this way next. This is the simplest cache policy to implement.

- Least Recently Used (LRU) – As the name suggests, the cache system replaces the line which has not been accessed recently. This policy will then update the line with current data. This is a complex cache replacement policy as the cache system has to track which line has been accessed last for every cache line and way.

- Least Frequently Used (LFU) – This replacement policy tracks the number of times each cache line has been accessed and will replace the line which has not been accessed the most. Again, this is a complex replacement policy due to the system tracking the frequency of accesses for each line for each way.

- Random – The cache system will randomly replace the cache line.
A study into modelling different replacement policies has been conducted by the authors in [5-22]. They propose an analytical model of a cache which predicts the cache miss rates from different configuration for a number of applications.

5.6 Load/Store Unit Microarchitecture

Figure 5-8 shows the outline of the LSU microarchitecture for the load store unit.

![Microarchitecture outline for the Load/Store Unit](image)

The LSU contains a vector cache and a Finite State Machine (FSM). The LSU is a serial configuration where the TAG part of the address (shown in figure 5-9) is checked in one cycle and the data is retrieved in the following cycle. The FSM handles the communication with the SRAM via the AHB and either loads data into a buffer 32 bits at a time before committing to the vector cache or stores the vector data coming in by splitting it up into 32 bit chunks. The LSU also supports unaligned accesses as this substantially simplifies the programming model of the SoC vector processor.

The LSU’s configuration can be altered in the coprocessor’s configuration file by setting the following variables:

- The number of set-associative ways.
- The size of the vector cache (in bytes).
For a direct-mapped cache the number of set-associative ways is set to one, the minimum possible limit. There is also a minimum limit for the size of the vector cache. To support unaligned accesses there has to be a minimum of 2 lines. E.g. For a maximum vector length of 8, the minimum size of the cache has to be 2kB with 2 lines, \((8 \times 16) \times 8 \times 2 = 2048\) bytes.

### 5.6.1 LSU Addressing

Figure 5-9 shows the addressing table for the LSU.

<table>
<thead>
<tr>
<th>TAG</th>
<th>Line Index</th>
<th>Odd/Even</th>
<th>Shift</th>
</tr>
</thead>
</table>

- \(DC\_WIDTH = VLMAX \times 16\)
- \(DC\_LINES = DC\_WAYSIZE / DC\_WIDTH \times 8\)
- \(A = \log_{2}(DC\_WIDTH / 8)\)
- \(B = A + 1\)

**Figure 5-9: Load/Store Addressing**

This information is used to work out where the data is located in the vector cache. Bits 31 down to \(C\) contain the TAG information. This is a unique combination of bits at a specific line index, given by bits \(C\) down to \(B\).

**Figure 5-10: Checking TAG**
B down to A is only one bit wide and tells the data section which RAMs the data is located, i.e. either in the Data Even or the Data Odd RAMs in figure 5-8. The vector cache supports unaligned accesses with bits A down to 0 being the shift amount.

The TAG RAMs in figure 5-10 are powered up if the address is valid. The line index is presented to all the ways and the TAGs that are stored in the RAMs are retrieved. These are then checked against the address to see if any of the TAGs match. If a TAG matches then there is a read “hit”, the way is encoded and passed onto the next stage of the LSU. If there is a read “miss” then the FSM is initiated to get the data via the AHB. There is also an “Even/Odd Valid” register which is only one bit wide. When high, this means that the TAG and data in the RAMs are valid and can be used. The TAG RAMs are dual port access types, where two different addresses can be presented to retrieve two TAGs in one clock cycle. This enables the LSU to retrieve the TAG for the next line index and is used for unaligned accesses. This is covered in more detailed in the next section.

### 5.6.2 Unaligned Accesses

Unaligned accesses only happen when bits A down to 0 are not equal to 0. This means that the data has been shifted in the vector cache.

![Figure 5-11: Even Unaligned Access](image)

Figure 5-11 shows the positions of each vector in the data RAMs for different even unaligned accesses with a vector length of 4 (64 bits). The data RAMs have a bit granularity of 8 bits and for each different shift amount, the data is moved to the right by 8 bits. As the shift amount is increased, more of the even vector data is stored in the Odd cache RAMs.
Odd unaligned accesses in figure 5-12 work in a similar manner to the even unaligned accesses. When the data wraps into the even cache RAM, the data is stored on the next line up (Line Index + 1). Therefore the LSU needs to check the TAG on the next line up to make sure that the data is still valid, hence the TAG RAMs are dual port accesses. This is only done during odd unaligned accesses.

### 5.6.3 Finite State Machine

The Finite State Machine is in charge of the communication between the LSU and the AHB. The input/output interface can be seen in figure 5-13 with the interface for the LSU on the left hand side of the diagram and the interface for the AHB on the right.

![Finite State Machine Interface](image)

**Figure 5-13: FSM interface**

### 5.6.4 AHB Transactions

The LSU sits on the AHB as a master as well as the main CPU as shown in figure 5-1. This enables the vector coprocessor to access the same off-chip memory that the main CPU has. This section describes typical read and write transactions between the LSU and the SRAM via the AHB.
Figure 5-14: AHB Read Transactions

Figure 5-14 shows a typical read transaction from the LSU to the SRAM via the AHB for a vector length of 4 beats (128 bit transaction). Before the rising clock edge of cycle 1, the FSM asserts the AHB Request high in order to gain access to the AHB. After a few cycles (rising clock edge of cycle 2 in this case), the AHB Arbiter will grant access to the AHB by asserting the HGrant signal high. The FSM then sets the required information about the transaction; the address of the data being received, it is a read, burst type and the size. The SRAM controller will see this information on the rising clock edge of cycle 3 and start getting the data from memory. Once the data is ready, the HReady signal is asserted high before the rising clock edge of cycle 5. The first data to be retrieved is 32 bits and contains the first two vectors, since one vector is only 16 bits wide. Before the rising clock edge in cycle 4, the FSM can increment the address to retrieve the next two vectors, but must keep the address stable till the rising clock edge of cycle 5 when the next block of data is retrieved. Data is retrieved in a similar manner to the first two vectors, once the HReady signal is asserted the FSM can read the data from the AHB bus at the rising clock edge of cycle 7. At this point, the transaction is no longer non-sequential as the address has been incremented to the next place in the memory and is quicker to access as there are no set-up times. This completes the even cache fill.

The following four vectors retrieved in the next two transactions are for the odd section of the cache. The address is sampled on the rising clock edge of cycle 7. The AHB request signal can now be driven low indicating that the next address presented will be the last. In cycle 9 the penultimate data is retried, the last address is read and the HGrant signal is now low.
indicating that the FSM can no longer drive the address bus. The last data segment is retrieved in cycle 11.

Figure 5-15: AHB Write Transactions

Figure 5-15 shows two different write transactions between the FSM and the SRAM. The first appears in cycles 3-5, the second during cycles 8-11. There are two 32 bit burst writes (4 vectors) followed by one 16 bit write transaction. This type of timing graph would be seen when there is an odd number of vectors, in this case a vector length of 5 (80 bits transmitted). For even vectors only the first half of the timing diagram would be seen.

The FSM requests permission to gain access to the AHB before the rising clock edge of cycle 1. The AHB Arbiter grants ownership of the data bus by asserting the HGrant signal high for the rising clock edge of cycle 2. After a few cycles the HReady signal is asserted high for the clock edge of cycle 5 telling the FSM that the data is committed and that the next piece of data can be presented on the following rising clock edge. During cycle 3 the HRequest signal is asserted low again signifying that next piece of data at the presented address will be the last and that this is the end of the burst transaction. The ready signal is asserted high for the rising edge of cycle 8 indicating that the data has been committed. It is possible that the HReady signal can take a number of cycles before being asserted high and the FSM must make sure that the Data on the bus remains stable.
The size of the transaction can not be changed when a burst transaction is initiated on the AHB. This is why the FSM asserts the HRequest signal low, changes the size of the transaction, along with the address and data then asserts the request signal back high again. The FSM puts the vector in the 16 LSB of the AHB data bus. As soon as the AHB master grants the FSM access to the bus, the request signal is asserted low to indicate that the transaction is only for one cycle.

5.6.5 FSM States

There are 10 states in the FSM. The interaction between states can be seen in figure 5-16.

![Figure 5-16: FSM State Diagram](image)

The states are:

- **Reset** – This state resets all variables and its registers into a known state.
- **Run** – This is the main state for the FSM. It stays in this state until the TAG checking process finds a read miss; which will cause the FSM to change state to Read Miss Grant and sets the HRequest signal high along with the size of the transaction. If there is a write (hit or miss) then the FSM will change state to Write Grant or Write Half Grant depending on the size of the vector, put the first 32-bit data on the AHB data out bus, the first address and set the HRequest signal high along with the size of the transaction. The FSM will only change state to Write Half Grant if the vector length is 1 (16 bits).
- **Read Miss Grant** – This state waits for the HGrant signal to be set before the FSM starts to receive data.
- **Write Grant** – During this state the FSM will split up the vectors into a 32-bit buffer to be written to memory via the AHB. This state then waits for the HGrant signal to be set high before changing state to **Write Data**.

- **Write Half Grant** – This is a similar state as Write Grant but does not split up data to be sent as only the 16 LSB need to be sent. After the HGrant signal has been set the FSM changes state to **Write Half Data**.

- **Fill Buffer** – Every time that the HReady signal is set, the data on the AHB bus is captured and put into a buffer. The HRequest signal is set low after the last but one piece of data is read from memory into the buffer. Once the buffer is full, the FSM changes state to **Replace Cache**.

- **Write Data** – The data in the buffer is put on the AHB Data-out bus 2 vectors at a time after the HReady signal is set. When the penultimate piece of data is to be written, the HRequest signal is set low to indicate the end of the transaction. If there are an odd number of vectors, the HRequest signal is set high again and changes state to **Write Half Grant** otherwise the state will be changed to **Run**.

- **Write Half Data** – As soon as this state is entered the HRequest signal is set low. The data to be written has been placed on the data bus by the previous state and therefore does not need to be set. This state will then wait for the HReady signal to be set and then change state to **Run**.

- **Replace Cache** – During this state, the data retrieved is placed into cache vector buffers which are a map of the line to be replaced. This is presented to the data cache and the replace signal is asserted high for one clock cycle to indicate to the cache to write the line. The replace signal is also an input to the TAG RAMs which indicates the process to update the TAGs and set the valid register. After one clock cycle, the FSM changes state to **Idle**.

- **Idle** – This state is only for one cycle. This gives a chance for the LSU to re-check the TAGs to make sure that the correct data has been retrieved. It is possible for an unaligned access for two lines to have completely missed where two cache lines will need to be retrieved and hence the TAGs need to be re-checked. The next state is **Run**.

### 5.6.6 Power Conservation

Power conservation has been taken into account when designing the LSU. Power is saved by only enabling certain parts of the data RAMs and enabling the TAG RAMs when needed. Due to the data RAMs having an 8 bit granularity, it is possible to only enable certain parts of the RAMs, thus saving power.
In figure 5-17, data has been shifted to the right and is held in 8 bit cells C-J. When data is retrieved from the RAMs, only these cells are enabled, while the other cells, A-B and K-P, are not. This will save power due to the RAMs not needing to retrieve data that is stored.

5.6.7 The Vector Data Cache

The vector data cache is configurable where the user can select the number of set-associative cache ways and the size of each. Calculations are then done during design elaboration on how many lines the cache can have. When there is a cache miss, the whole line is replaced using Round Robin i.e. for a 3-way set associative cache, it will fill Way 0 first, then Way 1, Way 2 and then replaces the data in Way 0.

When there is a Read Hit, data needs to be retrieved from the cache. Using the line index from the address and the way information encoded in the previous cycle, the bytes in the data cache are powered up and the data is retrieved from the cache. This is then placed into a shift buffer which is two times the size of VLMAX. Where the data is placed into the shift buffer depends on whether it is in an even or odd access, this is determined from the address. If it is in an even access, figure 5-18, the even cache is put into the MSB, bits 128-64, and the odd cache is
put into the LSB, bits 63-0. If it is an odd access, figure 5-19, then the opposite is done. As already explained, for odd unaligned access, the data is wrapped over onto the next line in the even cache. However, it is possible for the data not to be in the same way as the odd cache and hence it is feasible to get a partial hit either for the data in the odd cache or the even cache. This is calculated when the TAGs are checked and the data that is missing is retrieved by the FSM.

Once the data is in the shift buffer, it is passed through a barrel shifter and the final resulting vectors can be found in the MSB.

![Diagram of data cache and shift buffer](image)

**Figure 5-19: Odd Vector Cache Sort**

### 5.7 Cycle Results

Load and Store test vectors and Synchronous Dynamic RAM (SDRAM) contents were obtained from the ITU-T speech codecs during execution on the architecture simulator and were then applied to the LSU via the testbench shown in figure 5-20.
The test vectors were read from a file by the testbench and then applied to the LSU on every rising clock edge. If the LSU asserted the hold signal, the testbench would not issue the next test vector until the hold signal asserted low again. The AHB and off-chip memory were based upon the Leon SoC. The LSU result was collected by the testbench and then compared with the expected results. Any discrepancies were then recorded in an error report. The testbench also recorded the number of clock cycles it took to complete the test. This was then used to calculate the Clocks-per-Instruction (CPI) and the encoder results for the G.723.1 can be seen in figure 5-21 – figure 5-23 for three different workloads, 5.3kbps (figure 5-21), 6.3 kbps (figure 5-22) and at the mixed rate (figure 5-23). The encoder is the most compute-intensive part of the speech codecs and similar results can be seen in the decoder and the G.729A speech codec.

For a direct-mapped scheme (1-way set associative cache) the CPI performance is poor, especially as the vector length is increased. For a vector length of 128 (2048 bits), the CPI is off the scale with an average performance of 80 CPI across all workloads. This is due to the size of the cache not being large enough to store the required data for the custom instructions; cache misses are more frequent and the LSU is spending more time replacing data. As the vector length is increased, the cache can not hold the vectors due to its size i.e. a 1 Kbyte cache can not have a vector of length 128.

For a 2-way set associative cache, the CPI performance improves compared to the direct-mapped scheme particularly in the larger vector lengths. There is now more data that can be
stored and the LSU is spending less time accessing the slower off-chip memory. The CPI also improves as the cache size is increased.

Figure 5-21: G.723.1 Encoder at 5.3 kbps Clocks-per-Instruction Results

Figure 5-22: G.723.1 Encoder at 6.3 kbps Clocks-per-Instruction Results
Across all workloads, the optimum CPI performance is for a vector length of 8x16, 4-way set-associative cache and a cache size of 32 Kbytes per way with an average of 2 CPI.

For a small cache it takes more time to replace the required data as it is written back to memory and the new data is replaced. Therefore processing speed is reduced due to this bottleneck in the cache. This will have an effect on reconstructed sound for the end user as this will cause a delay or worst still garbled noise. Although replacing parts of the speech coders with custom instructions has shown improvement in the overall instruction count, the speed of processing is not necessarily increased. A cache is designed to reduce the perceived memory latency of the off-chip memory. If the cache is too small then the memory latency is not hidden which affects the processing speed as the main CPU will have to wait for the required data. If the cache is too big, then the parts of the cache are wasted so space of the ASIC design is too big. Alternatively, the memory latency of the off-chip memory can be too long and a larger and smarter cache is needed. These include changing the write policy and replacement policy, designing a victim cache (a smaller cache to store the recently discarded...
data), pre-empt the next data from the off-chip memory, retrieve the required data first before getting the rest of the cache line, increase the data bus width. These methods have not been investigated in this research and could be explored further to investigate their potential to increase the performance of a vector processor.

5.8 Power and Area Results

The LSU has been designed in RTL VHDL, and follows a strict, technology-independent coding style which ensures that the design can be targeted towards either an FPGA or ASIC technology with minimal changes. This is done by use of a switch in the coprocessor configuration file.

For ASIC implementations, the design was read into Synopsys Design Compiler and was synthesized for a high-performance 0.13µm standard-cell library. The optimized netlist was then imported into Cadence SoC Encounter where floor planning, including macro placement, power grid design and clock tree synthesis took place. This was followed by global and detail routing and 2.5D extraction using the Encounter timing engine. The output from the Encounter environment included gate-level netlists, standard-delay-format (sdf) and standard parasitic extraction format (spdf) files which were read again into Design Compiler for statistical power analysis.

Figure 5-24: Area results for 16 Kbyte cache at 250MHz

Figure 5-24 depicts area results from SoC Encounter for a 16Kbyte cache at a frequency of 250MHz. As the number of ways is incremented the area increases proportionally for each
vector length and is due to adding an extra RAM block for each way. The complexity of the LSU is also increased as the vector length is incremented and can be seen in the area results by the area doubling in size.

![Figure 5-25: Statistical Power Results for 8 Kbytes Cache (High Power)](image)

Power results for a cache size of 8 Kbytes per way for a range of vector lengths can be seen in figure 5-25 (High Power) and figure 5-26 (Low Power) and similarly for a cache size of 16 Kbytes in figure 5-27 (High Power) and figure 5-28 (Low Power).
Figure 5-27: Statistical Power Results for 16 Kbytes Cache (High Power)

Figure 5-28: Statistical Power Results for 16 Kbytes Cache (Low Power)

Figure 5-26 and figure 5-28 depicts the results from using low power optimization techniques where only the bytes that are needed are powered up in the cache. Figure 5-25 and figure 5-27 shows the results from the simulated parallel configuration. Each graph shows that there is a difference in power results from a direct mapped cache scheme to a 4-way set associative. The difference in power consumption also increases as the clock frequency of the LSU is increased.
To simulate a parallel configuration cache, where the TAG is checked in parallel to all the data ways are accessed, it is possible to configure the LSU so all the data from the cache is retrieved. The power results for these are shown in figure 5-26 and figure 5-28. There is a clear power saving between the parallel and serial LSU configuration especially for the 4-way set associative cache design. Comparing figure 5-26 and figure 5-28, there is not much difference in power consumption between the cache sizes.

Figure 5-29: LSU VLSI Layout - VLEN 8, 4-Way, 2 Kbytes Cache Size

Figure 5-29 depicts the VLSI Layout for the LSU for a vector length of 8 (128-bits), 2 kbytes, 4-way set associative cache for a TSMC 0.13μm silicon process.

5.9 Conclusion

The final section of this thesis has introduced the microprocessor architecture of the Load/Store Unit for the Coprocessor. By researching into different cache architectures and replacement policies, a serial vector cache was designed and implemented using VHDL. The performance of the LSU was recorded by monitoring the number of clocks needed per instruction executed. As the cache increased its size and set-associated ways, the performance also increases.

To obtain area and power results for an ASIC implementation, Synopsys Design Compiler and Cadence SoC Encounter were used. The size of the vector cache is dependent on the number of RAM blocks it has. It can be seen from figure 5-29 that the bulk of the area is made up of RAM blocks. The number of RAM blocks are dependent on the number of set associative ways and the vector length. Therefore by increasing the number of ways and
vector length the overall area is also increased. Taking into account the profiling results from
the previous chapter and performance results of the LSU, the best configuration for the LSU
would be having a vector length of 8 using a 4 way set associate cache with size of 16Kbytes.
This would lead to a CPI performance of around 2.4 across all workloads.

Throughout this thesis, the effect of adding custom data-level-parallel instructions to two ITU
speech coding algorithms, the G.729A and the G.723.1 has been studied. Through the results
in architecture exploration seen in chapter 4 over a range of vector lengths, the benefits of
reducing the dynamic instruction count can be seen. The results have shown an improvement
of 85% for both speech codecs especially during open loop pitch estimation and Harmonic
Noise Filtering when the harmonic noise filter coefficients are applied. There is not much
more improvement over a vector data length of 16 (256 bits) due to the size of the frames.
The speech codecs operate on blocks of frames of 240 samples. More often, the speech is
processed on a subframe basis of 60 samples and hence a vast improvement can be seen with
a low vector length. With a vector length of 4, the coprocessor would save 75% of its time
executing code. By doubling the vector length, the coprocessor would only save 88% which is
not a huge incremental improvement.

This thesis has shown the benefits of Single Instruction, Multiple Data processors and their
potential in performance compared to existing solutions. However, in order to achieve the
performance gains by going beyond one operation per clock cycle it is necessary to introduce
the custom instructions into the instruction set architecture to utilise the functional units that
have been designed. This task is labour intensive and time consuming for large applications.

This combined architecture has significant potential in achieving leading area/power/cost per
voice channel compared to existing solutions, while capable of being easily integrated in a
VoIP gateway SoC. This new architecture will potentially lead amongst others to a new
generation of VoIP gateways demonstrating an order of magnitude improvement in voice
channel capacity compared to the current state of the art.

The outcome of this work is a collection of processes for embedded systems utilizing vector
processors for speech coding, C-based simulation models of the proposed architecture as well
as software (vectorized vocoders) and hardware (VHDL implementation) intellectual property
(IP).
5.10 References

6 CONCLUSION

6.1 Project Findings

The aim of this thesis was to improve channel capacity of the VoIP network by targeting voice compression at the gateway of two networks. This was achieved by studying two speech coding algorithms usually found in VoIP applications and investigate their benefit for efficient execution on configurable extensible embedded vector CPU architecture.

The first part of this thesis presented the optimisation of speech algorithms from the C reference codes. The algorithms were profiled over different workloads to identify the computational intensive parts of the codes. This showed that codecs were spending most of their time executing emulated DSP functions e.g. addition, subtraction, multiplication, shifting etc. Upon further investigation, these emulated DSP functions appear in data-parallel loops. Therefore, it was decided that these functions could be replaced by custom instructions to target a configurable data-parallel architecture.

Once an instruction has been identified, a custom C Macro function was created and inserted into both of the speech reference codes in place of data-parallel loops and other non-vectorizable parts of the code. This was in order to quickly verify the functionality of the custom instructions for the ITU workloads against the expected results before implemented on a simulated processor. The custom extended instructions and functional blocks were then introduced into the SimpleScalar toolset which was modified to include the coprocessor’s state in order to evaluate the potential performance improvement of the introduced custom instructions.

The modified SimpleScalar toolset allowed adjusting the vector length configuration of the functional blocks in order to explore the vector architecture performance before implementing it in hardware. These showed a reduction in the dynamic instruction count of around 60% with just the vector instructions and a further 25% when the scalar instructions were applied. Dramatic improvement was seen over the lower vector lengths with the main point of interest being between a vector length of 4-16 (64-bit to 256 bit wide configurations). As the vector lengths increased, the dynamic instruction count levelled off. This is due to the size of each speech frames only being 240 samples. Therefore as the vector length increased, the vector architecture was not being use efficiently as not all the vector lanes were being used.
The next section of the work was the design of the SoC. The Leon3 processor core was chosen as the main RISC processor. The processor is highly configurable allowing designers easily to customise blocks. Leon3 system has a 7-stage pipeline with a Harvard Architecture where the processor has a separate instruction and data bus. The vector coprocessor has been tightly coupled to the main integer pipeline via custom ports. Instructions for the coprocessor are issued to the vector architecture during the decode stage of the Leon3. Data is then written back to the integer pipeline during the memory stage.

The coprocessor design consists of 4 stage pipeline:
- Decode
- Register
- Vector Datapath 1
- Vector Datapath 2

The vector datapath consists of an array of SIMD functional units with each lane being 16-bits wide. The lower two lanes can be fused together in order to execute the custom scalar instructions instead of introducing a custom scalar datapath.

Along side the 2-stage vector datapath is the vector Load/Store Unit. The main contribution of this research was the design and implementation of the LSU in RTL. A cascade TAG/DATA configuration was chosen in order to reduce the power of the coprocessor by only accessing the required data from the cache. The LSU is also attached the AHB bus as a Master on SoC in order to access off-chip data for cache misses. Load/Store test vectors were obtained from the ITU workloads and applied to a testbench in order to verify the design of the LSU and to obtain cycle results for different configurations. The optimum design of the LSU was for a configuration of a 4-way set-associative cache with a size of 32 Kbytes per way for a 128-bit wide configuration showing an average of 2 cycles per instruction for the Leon3 SoC, although this may increase if the latency of the off-chip memory is increases.

Since the design of the LSU is technology independent, it can be retargeted for different silicon technologies. The LSU can take a multiple different configurations (vector length, set associative ways, cache sizes etc) and lead to different area/power/frequency which can be investigated.

The full vector coprocessor, including the datapath and load/store unit was taken through the full front end, logical synthesis, and back end, place and route, flows. The configuration of the
vector coprocessor included a vector length of 16 (256-bit wide configuration), 4-way set associate vector cache with a size of 8Kbytes. The synthesized design target frequency was 200 MHz for TSMC 0.13μm process. The resulting VLSI macrocell can be seen in figure 6-1.

![VLSI Layout for Vector Coprocessor (VLSU and Vector Datapath)](image)

**Figure 6-1: VLSI Layout for Vector Coprocessor (VLSU and Vector Datapath)**

**Table 6-1: VLSI Physical Layout Properties**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width (μm)</td>
<td>1802</td>
</tr>
<tr>
<td>Length (μm)</td>
<td>3491</td>
</tr>
<tr>
<td>Area (mm²)</td>
<td>6.29</td>
</tr>
<tr>
<td>Cells (RAMs)</td>
<td>257308 (22)</td>
</tr>
<tr>
<td>Cell rows</td>
<td>921</td>
</tr>
<tr>
<td>Speed (MHz)</td>
<td>182</td>
</tr>
</tbody>
</table>

The physical characteristics are given in table 6-1. The design includes approximately 257 K gates, 22 RAM macros in 921 standard cells rows. The cell area is 1.8 by 3.5 mm². The
design achieved 182 MHz post-route worst case maximum frequency and clearly indicates that the critical path lies within the Vector Datapath.

The closest SIMD architecture targeting speech processing that is commercially available is the ARC Advanced Sound System [6-1]. This is coprocessor which can be used with many systems like ARM, MIPS etc via a 32-bit shared bus. It has a 128-bit wide configuration and introduces 60 new instructions. It can be synthesised for a frequency of 400 MHz and has an area of 2.54 mm² for a 90nm TSMC process. There was no information available on the memory infrastructure of the SIMD accelerator. Although the specifications given are better than the implemented design in this thesis, they only concentrate on high-fidelity audio products such as in-car entertainment, home theatres, DVD burning, so only support encoding standards such as MP3, Dolby Digital etc.

IBM have been researching a heterogeneous chip multiprocessor which consists of a 64-bit Power Core with 8 128-bit wide configuration coprocessors and is called “Cell” [6-2]. Each coprocessor has 128 registers, 16-bit load/store bus to access local memory which houses a controller to communicate with other coprocessors. The coprocessors are able to handle 2 instructions per cycle, once for computation and the other to access memory. The total area of the Cell architecture is 221mm² using 90nm Silicon-on-Insulator (SOI) technology process. Comparing IBM’s Cell with this project, the load/store bus is smaller so the processing could suffer from long latencies from the system bus, especially with a large number of registers available.

The final coprocessor to compare is the Intel’s MMX accelerator [6-3]. The accelerator introduces 57 new instructions with the processor able to issue instructions per clock cycle, similar to IBM’s cell, one for computation and the other for memory access. The accelerator only has a 64-bit wide configuration and 8 registers, smaller than this project proposed design. It has a 4-way set associative 16Kb instruction/data write-through cache. Intel have also chosen to have a 64-bit wide memory bus which means the bandwidth of accessing external memory is reduced as data is retrieved across a wider bus interface. Coupled with an Intel processor, the accelerator can be used up to a frequency of 230 MHz. Although the configuration is smaller than the processor proposed in this thesis, the processing speed could be comparable due to the wider bus configuration and cache. This would depend on the type of instructions that are implemented on the accelerator.
6.2 Future Work

The vector processor was developed to target speech coding for two standards, G.729A and the G.723.1 where the basic DSPs operations were similar. Since the ISA introduced is based on these basic operations, other ITU speech coding standards could also be implemented into an embedded application such as the G.711, G.726, G.727 and G.728 and incorporate them on a H.323 network. Other VoIP networks could be researched, such as a SIP network, and compare the implementation of speech coding to see if the same basic operations are implemented in the same manor and hence the utilise the proposed ISA and processor design. If they are not similar, the ISA/processor design could be modified slightly to become a generic design for VoIP networks speech accelerator.

Since the vector coprocessor has been designed to connect to any scalar CPU with little modification to its interface it has the potential to attach to different architectures and ASIC designs. Further research could be undertaken to measure the coprocessor performance when coupled with different CPUs.

Another aspect that could be of interest for potential work is the coprocessor architecture. IBM’s Cell and Intel MMX both have dual issuing instructions split into compute/memory instructions. This has the potential of utilising the whole coprocessor at the same time making more efficient without wasting resources i.e. currently the instructions either operate on the datapath or the LSU. If instructions could issued at the same time, both resources would be utilised at the same time making it more efficient. Other work that could be investigated with regards to the coprocessor architecture is the number of registers it incorporates. Currently, the vector processor design has only 16 registers and can be addressed in 4-bits. Interestingly, IBM Cell found that had 7-bits for addressing registers so implemented 128 registers, which is a large number of registers while Intel’s MMX processor only has 8 registers and are addressed using 3-bits. The number of registers effect how often data needs to be loaded and stored to memory. Too few registers and it is possible that data needs to be frequently loaded/stored into registers for execution. If there are too many registers, not all the resources will be used and will be a waste of space.

With regards to the specific design of the VLSU architecture, there are a number of avenues that could be explored. The design of the VLSU in this thesis featured a cascade configuration where the TAGs where checked in the first clock cycle and the data retrieved in the following. This leads to an increase in the power saving but decreased the time it took to produce the data for processing. The effect on a parallel configuration with regards to processing could be
investigated further. With the current datapath design, data would be produced at the end of first stage of the datapath instead ready for processing then instead of second stage to return to the registers. This gives the ability to do some processing on the data before it is returned to the registers e.g. simple addition, shifting etc. Alternatively, the whole vector coprocessor architecture could be modified further to introduce the memory access earlier in the vector pipeline, but would complicate the design further due to pipeline stalls. Other modifications to LSU design could include the addition of a victim cache. When data is evicted from the cache, it would be temporally stored in a victim cache. This would allow the LSU to quickly restore data into the main cache for processing without accessing slower memory. A victim cache would probably not be useful for speech coding for a large cache. Since speech coding operates iteratively on data frames, once it has finished its calculation on that frame, it will move on to the next frame. Therefore, if the cache is small, a victim cache would be more useful. The LSU could also be modified to include different fetching methods. Here, the LSU could determine the next address to be access and pre-fetch the data while idle before it is needed, reducing the need to stall the pipeline. Alternatively, the LSU could fetch the required data first before filling the cache line. Finally, different replacement policies could be investigated, i.e. First In - First Out, Most Frequently Used, Random etc to see which policy best suits speech coding.

As VoIP gains more attraction coupled with the increase of combined multimedia applications, there is a significant amount of parallelism that can be targeted in embedded systems. Therefore by attaching the vector coprocessor to multithreaded architecture could further accelerate these multimedia heavy loaded systems that incorporate speech coding into their design [6-4].

6.3 References

[6-1] ARC Advanced Sound System, www.arc.com
This section shows the results for the G.723.1 ITU speech codec results (encoder and decoder) first with vector instructions enabled followed by vector and scalar instructions enabled.

**G.723.1 Encoder - Vector Coprocessor**

**G.723.1 Decoder - Vector Coprocessor**
Appendix A - G.723.1 Results

G.723.1 Encoder - Vector and Scalar Coprocessor

G.723.1 Decoder - Vector and Scalar Coprocessor
APPENDIX B – G.729A RESULTS

This section shows the results for the G.729A ITU speech codec results (encoder and decoder) first with vector instructions enabled followed by vector and scalar instructions enabled.

G.729A Coder - Vector Coprocessor

[Graph showing dynamic instruction count vs. vector length]
Appendix B – G.729A Results

G.729A Decoder - Vector and Scalar Coprocessor

DYNAMIC INSTRUCTION COUNT

0 0.1 0.2 0.3 0.4 0.5 0.6 0.7

0 16 32 48 64 80 96 112 128

VECTOR LENGTH

Algorithm
Fixed
Lap
Pitch
Tone
Speech
Erasure
Overflow
Parity
This section presents the results from the G.723.1 speech encoder showing the improvement made from a function perspective.
Appendix C – G.723.1 Function Results

G723.1 Calc_Exc_Rand

G723.1 Cod_cmg
Appendix C – G.723.1 Function Results

### G723.1 Comp_Ir

[Graph showing dynamic instruction count vs. vector length for different rates and vector operations]

### G723.1 LPCDiff

[Graph showing dynamic instruction count vs. vector length for different rates and vector operations]
G.723.1 Comp_Vad

G.723.1 Cor_h_x

Vector Length

Vector Length
Appendix C – G.723.1 Function Results

G723.1 Decod_Acbk

G723.1 Durbin
Appendix C – G.723.1 Function Results

G723.1 Estim_Pitch

G723.1 Filt_Pw
Appendix C – G.723.1 Function Results

G723.1 Find_Ack

G723.1 Find_Best

Mix Rate, Vector
Mix Rate, Vector and Scalar
Rate 53, Vector
Rate 63, Vector and Scalar
Rate 63, Vector and Scalar

Mix Rate, Vector
Mix Rate, Vector and Scalar
Rate 53, Vector
Rate 63, Vector
Rate 63, Vector and Scalar

Mix Rate, Vector
Mix Rate, Vector and Scalar
Rate 53, Vector
Rate 63, Vector
Rate 63, Vector and Scalar

0 16 32 48 64 80 96 112 128
Vector Length

0 0.1 0.2 0.3 0.4 0.5 0.6 0.7
Dynamic Interaction Count

0 0.45 0.55 0.65 0.75 0.85 0.95 1.05 1.15
Dynamic Interaction Count
Appendix C – G.723.1 Function Results

**G723.1 Init_Vad**

- Mix Rate, Vector
- Mix Rate, Vector and Scalar
- Rate 53, Vector
- Rate 53, Vector and Scalar
- Rate 63, Vector
- Rate 63, Vector and Scalar

**G723.1 Lsp_Inq**

- Mix Rate, Vector
- Mix Rate, Vector and Scalar
- Rate 53, Vector
- Rate 53, Vector and Scalar
- Rate 63, Vector
- Rate 63, Vector and Scalar
Appendix C – G.723.1 Function Results

G.723.1 Lsp_int

G.723.1 Lsp_Qnt
Appendix C – G.723.1 Function Results

G723.1 Mem_Shift

G723.1 Upd_Ring
The following are the list of publications related to this work:


