Linköping Studies in Science and Technology

Thesis No. 1093

Hardware for Speech and Audio Coding

Mikael Olausson

INSTITUTE OF TECHNOLOGY
LINKÖPING UNIVERSITY

LiU-TEK-LIC-2004:22
Department of Electrical Engineering
Linköpings universitet, SE-581 83 Linköping, Sweden
Linköping 2004

Linköping Studies in Science and Technology

Thesis No. 1093

Hardware for Speech and Audio Coding

Mikael Olausson

INSTITUTE OF TECHNOLOGY
LINKÖPING UNIVERSITY

LiU-TEK-LIC-2004:22
Department of Electrical Engineering
Linköpings universitet, SE-581 83 Linköping, Sweden
Linköping 2004

ISBN 91-7373-953-7
ISSN 0280-7971

Abstract

While the Micro Processors (MPUs) as a general purpose CPU are converging (into Intel Pentium), the DSP processors are diverging. In 1995, approximately 50% of the DSP processors on the market were general purpose processors, but last year only 15% were general purpose DSP processors on the market. The reason general purpose DSP processors fall short to the application specific DSP processors is that most users want to achieve highest performance under minimized power consumption and minimized silicon costs. Therefore, a DSP processor must be an Application Specific Instruction set Processor (ASIP) for a group of domain specific applications.

An essential feature of the ASIP is its functional acceleration on instruction level, which gives the specific instruction set architecture for a group of applications. Hardware acceleration for digital signal processing in DSP processors is essential to enhance the performance while keeping enough flexibility. In the last 20 years, researchers and DSP semiconductor companies have been working on different kinds of accelerations for digital signal processing. The trade-off between the performance and the flexibility is always an interesting question because all DSP algorithms are "application specific"; the acceleration for audio may not be suitable for the acceleration of baseband signal processing. Even within the same domain, for example speech CODEC (COder/DECoder), the acceleration for communication infrastructure is different from the acceleration for terminals.

Benchmarks are good parameters when evaluating a processor or a computing platform, but for domain specific algorithms, such as audio and speech CODEC, they are not enough. The solution here is to profile the algorithm and from the resulting statistics make the decisions. The statistics also suggest where to start optimizing the implementation of the algorithm. The statistics from the profiling has been used to improve implementations of speech and audio coding algorithms, both in terms of the cycle cost and for memory efficiency, i.e. code and data memory.

In this thesis, we focus on designing memory efficient DSP processors based on instruction level acceleration methods and data type optimization techniques. Four major areas have been attacked in order to speed up execution and reduce

memory requirements. The first one is instruction level acceleration, where consecutive instructions appear frequently and are merged together. By this merge the code memory size is reduced and execution becomes faster. Secondly, complex addressing schemes are solved by acceleration for address calculations, i.e. dedicated hardware are used for address calculations. The third area, data storage and precision, is speeded up by using a reduced floating point scheme. The number of bits is reduced compared to the normal IEEE 754 floating point standard. The result is a lower data memory requirement, yet enough precision for the application; an mp3 decoder. The fourth contribution is a compact way of storing data in a general CPU. By adding two custom instructions, one load and one store, the data memory efficiency can be improved without making the firmware complex. We have tried to make application specific instruction sets and processors and also tried to improve processors based on an available instruction set.

Experiences from this thesis can be used for DSP design for audio and speech applications. They can additionally be used as a reference to a general DSP processor design methodology.

Preface

This thesis presents my research from October 2000 through January 2004. The following four papers are included in this thesis:

- Mikael Olausson and Dake Liu, "Instruction and Hardware Accelerations in G.723.1(6.3/5.3) and G.729", in *Proceedings of the 1st IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, Cairo, Egypt, December, 2001, pp 34-39
- Mikael Olausson and Dake Liu, "Instruction and Hardware Acceleration for MP-MLQ in G.723.1", in *Proceeding of IEEE Workshop on Signal Processing Systems (SIPS'02)*, San Diego, California, USA, October 16-18, 2002, pp 235-239
- Mikael Olausson, Andreas Ehliar, Johan Eilert and Dake Liu, "Reduced Floating Point for MPEG1/2 Layer III Decoding", to be presented at *the International Conference on Acoustics, Speech and Signal Processing (ICASSP'04)*, Montreal, Quebec, Canada, May 17-21, 2004
- Mikael Olausson, Anders Edman and Dake Liu, "Bit Memory Instructions for a General CPU", to be presented at to *The 4th IEEE International Workshop on System-on-Chip for Real-Time Applications(IWSOC'04)* Banff, Alberta, Canada, July 19-21, 2004

The following two papers related to my research are not included in the thesis:

- Eric Tell, Mikael Olausson and Dake Liu, "A General DSP Processor at the Cost of 23k Gates and 1/2 a Man-Year Design Time", in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP'03)*, Hong Kong, April 6-10, 2003, pp 657-660 Volume II
- Mikael Olausson and Dake Liu, "The ADSP-21535 Blackfin and Speech Coding", in *Proceedings of the Swedish System-on-Chip Conference (SSoCC)*, Eskilstuna, Sweden, April 2003

Acknowledgment

First of all, I want thank my supervisor Professor Dake Liu for guidance, inspiration, and for giving me the opportunity to become a part time Ph.D. student.
I would also like to thank Professor Christer Svensson for making it possible to start my research.

I want to thank Dr. Anders Edman and my fellow Ph.D. students Andreas Ehliar, Johan Eilert, and Eric Tell for co-authoring papers with me.

Further, I want to thank my fellow Ph.D. students Dr. Tomas Henriksson and Dr. Ulf Nordqvist, fellow Ph.D. students Sumant Sathe, Eric Tell, and Daniel Wiklund, and Research Engineer Anders Nilsson for all the sport and outdoor activities such as: bicycle trips, orienteering, ski trips, and bandy both on and off ice.

Thank you to all the members of the divisions of Electronic Devices and Computer Engineering at Linköping University who have contributed to a nice working environment.

Thanks to Sectra Communications AB for giving me the chance to still work for them as a part time employee while doing my research at the university.

A special thanks to my girlfriend Anna Larsson who has read and corrected my English within the papers and supported me in every way.

The thesis work was sponsored by the STRINGENT of Swedish Foundation of Strategic Research (SSF) and the Center for Industrial Information Technology at the Linköping Institute of Technology (CENIIT).

Contents

Abstract iii

Preface v

Acknowledgment vii

Abbreviations xiii

I Introduction 1

1 Introduction 3
- 1.1 Background . . . 3

2 Benchmarking 5
- 2.1 Cycle Cost . . . 6
  - 2.1.1 Application Profiling . . . 6
- 2.2 Memory Cost . . . 8
  - 2.2.1 Code Cost . . . 8
  - 2.2.2 Data Cost . . . 9
- 2.3 References . . . 10

II Coding Strategies 11

3 Speech Coding 13
- 3.1 Introduction . . . 14
- 3.2 Speech Coding Techniques . . . 14
  - 3.2.1 Vocoding . . . 15
  - 3.2.2 Multi Pulse Excitation . . . 16
  - 3.2.3 Multiband Excitation . . . 18
- 3.3 Complexity Aspects . . . 19

3.3.1 Coding delay 20
3.4 Hardware Acceleration Opportunities 20
3.5 References 21

4 Audio Coding **23**
4.1 Introduction 23
4.2 Description of Perceptual Coding 23
4.3 Coding Standard 25
4.4 Hardware Acceleration Opportunities 26
4.5 References 27

III Implementation 29

5 Hardware **31**
5.1 Difference between Speech and Audio CODEC 31
5.2 References 32

6 Architectures **33**
6.1 Architectures for Speech and Audio Coding 33
6.2 Programmable 33
6.2.1 General and Embedded DSP 34
6.3 FPGA 34
6.4 ASIC 34
6.5 References 35

7 Research Methodology and Achievements **37**
7.1 Profile the Algorithm 37
7.1.1 Stimuli to the Algorithm 38
7.2 Acceleration on Instruction Level 38
7.3 Acceleration for Address Calculation 40
7.4 Function Level Acceleration 41
7.5 Data Format and Precision 41
7.6 Compact Data Storing 44
7.7 Advantages and Disadvantages of Hardware Acceleration 47
7.8 References 48

IV Papers 49

8 Paper 1 **51**
8.1 Introduction 52

8.2 General description of G.723.1 and G.729 53
8.3 Statistics of basic operations 53
8.3.1 Description of the operands 53
8.3.2 Investigation of the statistics 55
8.4 Assembly instruction and hardware specification for improvements 56
8.4.1 32-bit conditional move with loop index 56
8.4.2 Hardware improvements in G.723.1 6.3 kbit/s 58
8.5 Performance estimations 59
8.6 Conclusion 60
8.7 Acknowledgment 61
8.8 References 61

9 Paper 2 63
9.1 Introduction 64
9.2 General Description of G.723.1 65
9.3 Statistics of Basic Operations 65
9.4 Former Work 66
9.5 Hardware Improvements in G.723.1 6.3 kbit/s 66
9.6 Performance Estimations 69
9.7 Conclusion 71
9.8 Acknowledgements 71
9.9 References 71

10 Paper 3 73
10.1 Introduction 74
10.2 General description of the MP3 format 75
10.3 The work 75
10.4 Motivation 77
10.5 Testing the quality 78
10.6 Savings in power and area 81
10.7 Future work 82
10.8 Conclusion 83
10.9 Acknowledgment 83
10.10References 84

11 Paper 4 85
11.1 Introduction 86
11.2 Related Work 88
11.3 General description of the OR1200 CPU 88
11.4 The work 88
11.4.1 Modifications in OR1200 89

11.4.2 The memory control module 90
11.4.3 Modifications in tools and code generation 94
11.5 Motivation and Applications 94
11.5.1 FIR filter 94
11.5.2 Compression scheme V42bis 96
11.6 Area and timing 97
11.7 Future work 97
11.8 Conclusion 98
11.9 Acknowledgment 98
11.10References 98

Abbreviations

AAC	Advanced Audio Coding
AaS	Analysis and Synthesis
AbS	Analysis by Synthesis
ACELP	Algebraic Code Excited Linear Prediction
ADC	Analog to Digital Converter
ADPCM	Adaptive Differential Pulse Code Modulation
AGU	Address Generation Unit
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction set Processor
ASSP	Application Specific Standard Product
BMC	Bit Memory Controller
BDTI	Berkeley Design Technology Incorporation
CELP	Code Excited Linear Prediction
CD	Compact Disc
CODEC	COder/DECoder
CPU	Central Processing Unit
DSP	Digital Signal Processor
EFR	Enhanced Full Rate
FA	Full Adder
FIR	Finite Impulse Response
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
FR	Full Rate
GSM	Global System for Mobile communications
HR	Half Rate
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IIR	Infinite Impulse Response
ISO	International Organization for Standardization
ITU-T	International Telecommunication Union Telecom

kbps	Kilo Bit Per Second
LMS	Least Mean Square
LP	Linear Prediction
LPC	Linear Prediction Coding
LSP	Linear Spectral Pair
MAC	Multiply and Accumulate
MFLOPS	Million of Floating Point Operations Per Second
MIPS	Million of Instructions Per Second
MMU	Memory Management Unit
MOPS	Million of Operations Per Second
MPEG	Moving Picture Expert Group
MP-MLQ	Multi Pulse Maximum Likelihood Quantization
MSE	Mean Square Error
MPU	Micro Processor
PAC	Perceptual Audio Coder
PCM	Pulse Code Modulation
PDA	Personal Digital Assistant
RISC	Reduced Instruction Set Computer
RMS	Root Mean Square
RPELPC	Regular Pulse Excitation Linear Prediction Coding
RPE-LTP	Regular Pulse Excitation Long Time Prediction
SIMD	Single Instruction Multiple Data
SHNR	Signal to Hearable Noise Ratio
SNR	Signal to Noise Ratio
SQAM	Sound Quality Assessment Material
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VoIP	Voice over Internet Protocol
VSELP	Vector Sum Excitation Linear Prediction

Part I

Introduction

Chapter 1

Introduction

1.1 Background

Coding of audio and speech can be divided up into two main frames; lossless and lossy coding. In lossless, or noiseless coding the original signal is completely reconstructed at the output of the decoder. In the second coding strategy, lossy coding, the original signal is not completely reconstructed at the output of the decoder, but “good enough”. The meaning of “good enough” may differ from time to time. When you listen to music from a lossy coder the quality requirements of the music are not different from listening to the original music. For a speech coder in a narrow band application it can be enough to understand the speaker in the other end without getting the characteristics of the speaker. The main advantageous of lossy coding over lossless is the lower bit rates achieved in lossy coding. Lossy coding can be further divided into waveform and synthesis based coding. The waveform coding tries to reassemble the original signal as close as possible, while the synthesis based coding extract parameters from the signal based on the assumption that the signal for example is speech like. The focus for the thesis is on synthesized based lossy coding, because it gives the more compression compared to waveform coding and lossless coding. The major difference between speech and audio is the frequency range they each work within. Speech is limited to the range 0-4 kHz. This limit comes from the bandwidth of the telephone lines. Audio, on the other hand has no such limitations. The limitation here is the poor ability of the human ear to extract and understand high frequencies. The Nyquist theorem states that the sampling rate has to be at least twice as high as the highest frequency component. If no low pass filtering takes place before the sampling, higher frequency components will be folded into lower frequency components. Typical sampling rates for music systems are 44.1 kHz in compact discs (CDs). This gives us the highest frequency component at 22.050 kHz. The upper limit

for the human hearing system is below 20 kHz and it decreases for elderly people. From the point of human hearing device, this sampling frequency is high enough. The low pass filtering and the sampling is not enough for digital speech. With the above samplings rates and a precision of 16 bits per sample the bit rates are as follows; for speech 128 kbit/s and for audio in a CD media 700 kbit/s. In addition, if one wants stereo the bit rates will double. For audio, especially music, it is popular with surround effects and in that case five channels are needed. The dynamic range with 16 bits might not be enough in certain applications. Instead, 20, 24 or even 32 bits must be used. This adds up to bit rates of a couple of Mbit/s. By using a coding and compression strategy, a lot of storage area and bandwidth can be saved.

Chapter 2

Benchmarking

The performance of an algorithm on a certain architecture is known as benchmarking. The benchmarks usually only measure the number of clock cycles required for a known application, but there are more things within the expression benchmark, for example the code and the data memory cost; an application might trade memory costs for clock cycles and vice versa. Instead of using the cycle cost measurement, Million of Instructions Per Second (MIPS) or Million of Operations Per Second (MOPS) is more common. For floating point implementations the measurement Million of Floating Point Operations Per Second (MFLOPS) is used. These measurements are used widely and in many cases it is difficult to interpret them. For example, a speech coding algorithm works on 20 ms frames. Every 20 ms a new frame is retrieved and the old frame must be processed. The worst case scenario is that the algorithm requires 400,000 clock cycles to finish a frame. A processor must be able to deliver at least 20 MIPS to run the algorithm in real time. In the data sheets for a processor, the manufacturer claims that the processor performance is 40 MIPS. Even if the processor delivers more MIPS than necessary for the algorithm, it might still not work. The reason is that the processor has parallel data paths, each delivering 10 MIPS, see figure 2.1. If there are four of them, they will deliver 40 MIPS. The question for an implementer of the algorithm is: Can one keep the four parallel data paths busy the whole time? If not, then the delivered performance from the processor is decreased significantly. If only one data path is used the whole time and occasionally one of the others, the delivered performance might be only 15 MIPS. This is less then the required 20 MIPS of the application. The said performance of a processor is usually the peak performance given for marketing purposes. In order to really understand the benchmarking numbers, the underlying architecture of the processor must be fully understood.

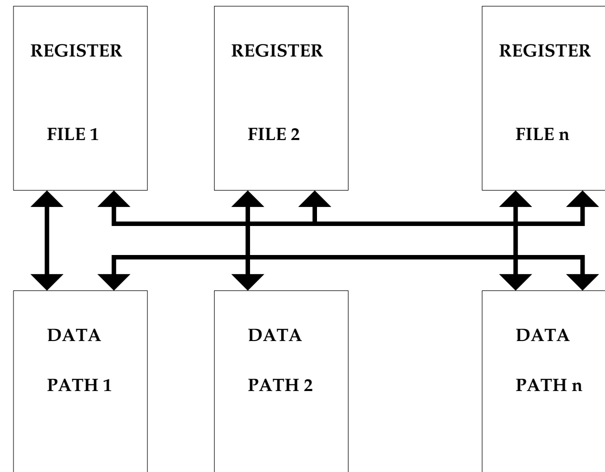

Figure 2.1: Parallel data paths.

2.1 Cycle Cost

The cycle cost is the main parameter for benchmarking [1]. This parameter is dependent on the implementation and the skills of the programmer. An assembly implementation usually performs better than a compilation from any high level language. Customized instructions and dedicated hardware can significantly reduce the number clock cycles needed. Some common benchmarks Berkeley Design Technology Inc. (BDTI) are presented in table 2.1.
It is also important to keep in mind that reducing the number of clock cycles can save power. With a lower cycle cost it might be possible to run another application on the same processor. This in turn can relax or take away another processor. The lower cycle count gives the possibility to reduce the clock frequency of the processor and also the supply voltage. This will reduce power consumption even more. The reduction of the supply voltage, *V*, is more favorable because the power consumption, *P*, is proportional to the square of the supply voltage and only directly proportional to the clock frequency, *f*, see equation 2.1.

$$P \propto V^2 f \tag{2.1}$$

2.1.1 Application Profiling

The main drawback with the benchmarks presented in table 2.1 are that they do not include the kernel operations for speech and audio coding. The benchmarks from the table are important and the speech and audio coding algorithms include

Benchmark	**Description**
Real Block FIR	Finite impulse response filter that operates on a block of real (not complex) data.
Single-Sample FIR	FIR filter that operates on a single sample of real data.
Complex Block FIR	FIR filter that operates on a block of complex data.
LMS Adaptive FIR	Least-mean-square adaptive filter; operates on a single sample of real data.
Two-Biquad IIR	Infinite impulse response filter that operates on a single sample of real data.
Vector Dot Product	Sum of the point-wise multiplication of two vectors.
Vector Add	Point-wise addition of two vectors, producing a third vector.
Vector Maximum	Find the value and location of the maximum value in a vector.
Viterbi Decoder	Decodes a convolutionally encoded bit stream.
Control	A contrived series of control (test, branch, push, pop) and bit manipulation operations.
256-Point FFT	Fast Fourier Transform converts a normal time-domain signal to the frequency domain.
Bit Unpack	Unpacks words of varying length from a continuous bit stream.

Table 2.1: Common benchmarks.

most of them, but they are not enough. For example, the encoding part of speech coding relies heavily on exhaustive search in inner loops. By searching through a lot of combinations, the one that produces the smallest error compared to the original signal is chosen. A kernel instruction here is compare and choose. This kind of operation is not included in the benchmarks. The theory and the hardware solutions are deeper described in chapter 3.2 and 7.2. Also, the address calculations can be complex and are not included in the benchmarks. In the benchmarks are post increment/decrement, modulo addressing, and support for butterfly computation in FFT included. In speech and audio coding are offset calculations with absolute values necessary. The best way to comprise these kernel operations is to profile the application. The application is fed with different kinds of stimuli and from the profiling is statistics coming out. These statistics will then point out the kernel operations. The profiling technique is described in chapter 7.1.

2.2 Memory Cost

A typical memory layout is shown in figure 2.2. The memories closest to the processor are faster, more expensive, and bigger per stored bit than the memories further away. From a performance point of view it would be ideal to place a lot of this memory closest to the processor. Unfortunately, from an economic aspect, this is not possible. The design of the memory layout will definitely influence the overall performance. In the following two subsections the memory cost will be further divided up into code and data memory cost.

2.2.1 Code Cost

The size of the instruction code will affect both the power consumption and the chip area. Every clock cycle is a new instruction fetched from the program memory. The easiest approach for instruction coding is to have the same size on all the instructions. Then, the processor only has to fetch the same amount of new bits from the instruction memory each time. 16 or 32 bits are common instruction set sizes for embedded processors. The big drawback of having the same size of all instructions is the large redundancy in many instructions. For example, some instructions only need a couple of bits for their representation, while others need 16 or even 32. It can be difficult to have instructions where all of them have different lengths. In the DSP Blackfin [2] from Analog Devices the instruction for signal processing tasks and the control tasks are divided into two groups. The group with control flow tasks is only 16 bits wide, while the signal processing related are 32 bits wide. By this separation the code density can be higher. Another approach to separate 16 and 32 bit instructions is to have two instruction sets to the same processor. In the ARM [3] processor two instruction sets exist; one 32 bits wide and one 16 bit wide also known as thumb instructions. By specifying whether the shorter instruction set, the thumb instruction set, is used or not, the processor will fetch either 16 or 32 bits instructions words. There are also other ways to save instruction code size. In paper [4] and [5] the code size is reduced by merging instruction and also adding new ones. A 32 bits conditional move is merged together with a conditional loop index move into one instruction. This kind of operation is very helpful in speech coding algorithm based on Analysis-by-Synthesis (AbS). In addition, extra hardware can be added to the address generation unit to speed up address calculations. This new customized address instruction saves code memory by taking away the need of ordinary instructions for the address calculations. The floating point approach to mp3 decoding in [6] will also save instruction code. While the dynamic range of the variables is increased compared to a fixed point implementation, the need for proper scaling within the code is avoided.

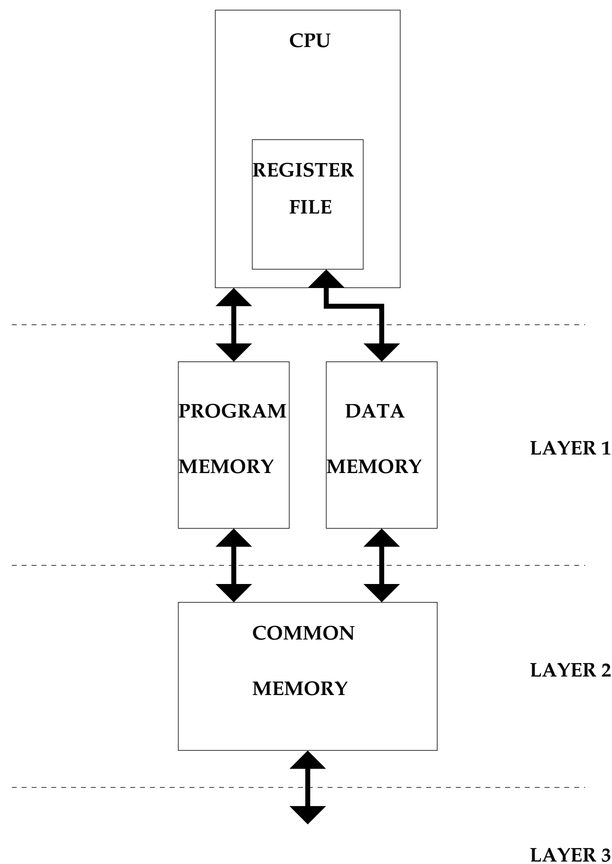

Figure 2.2: Memory hierarchy.

2.2.2 Data Cost

The memory requirements of an embedded system are getting larger and larger. Most of the chip area is dedicated to memory, mainly data memory. Therefore, an efficient use of the data memory can give substantial savings. For example, temporary buffers should be reused. Most of the applications are known before run time, i.e. the memory allocation can be decided at compile time. By careful partitioning of the tables and the buffers needed, fragmentation of the memory can be avoided. The sizes of the variables used are also of importance. Use the smallest variable size as often as possible. Larger variable sizes consume more memory and can also make the computations more cumbersome, for example 32-bit operations in a 16-bit processor. The data memory size can sometimes be traded for cycle cost, i.e. by using more data memory, the number of clock cycles for an application can be reduced. An obvious example is the choice between storing data in a table or calculate them whenever needed. If the whole table is too

large to fit into the data memory, parts of the table can still be stored there. In the case of a sine or a cosine table only the first quarter of the table is necessary to be stored in memory. The remaining three quarter of the table can easily be calculated from the first quarter. Too further reduce the number of entries in the table, one can use interpolation between two consecutive entries to get a more accurate value somewhere in between. In paper [6] the floating point format is changed to just fit the requirements. The sizes of the variables stored in memory are only 16 bits instead of 32, which is stated in single floating point representation [7]. A more aggressive approach is to introduce load and store instructions that work on bits in the memory [8]. Here, the variable sizes can be defined directly within the instruction. The memory as seen from the processor is not byte oriented, rather it is similar to a long bit stream.

2.3 References

[1] Berkeley Design Technology, Inc.(BDTI) *http://www.bdti.com*.

[2] Analog Devices *http://www.analog.com*.

[3] ARM *http://www.arm.com*.

[4] M. Olausson and D. Liu, "Instruction and Hardware Accelerations in G.723.1(6.3/5.3) and G.729," in *The 1st IEEE International Symposium on Signal Processing and Information Technology*, pp. 34–39, 2001.

[5] M. Olausson and D. Liu, "Instruction and Hardware Accelerations for MP-MLQ in G.723.1," in *IEEE Workshop on Signal Processing Systems*, pp. 235–239, 2002.

[6] J. E. M. Olausson, A. Ehliar and D. Liu, "Reduced Floating Point for MPEG1/2 Layer III Decoding," in *International Conference on Acoustics, Speech and Signal Processing*, 2004.

[7] *IEEE Standard 754 for Binary Floating-Point Arithmetic*, 1985.

[8] M. Olausson and A. Edman, "Bit memory Instructions for a General CPU," in *International Workshop on System-on-Chip for Real-Time Applications*, 2004.

Part II

Coding Strategies

Chapter 3

Speech Coding

There are two different places where the encoding/decoding takes place; the terminal and the base station. They will both do the same thing, encoding and decoding, but the requirements on them are different. While a terminal usually only has to serve one channel, a base station has to serve many channels. The power consumption for a terminal might be crucial because many of them are battery powered. A long battery lifetime is definitely a good sales argument. In a base station the power consumption is also important, the reason for this being is the heat generation from the chips; the more heat they generate, the more cooling devices have to be applied and cooling devices cost money. In this chapter the most popular speech coding techniques will be reviewed. For a more detailed understanding and more thoroughly examination of speech coders, the references [1] and [2] is recommended.

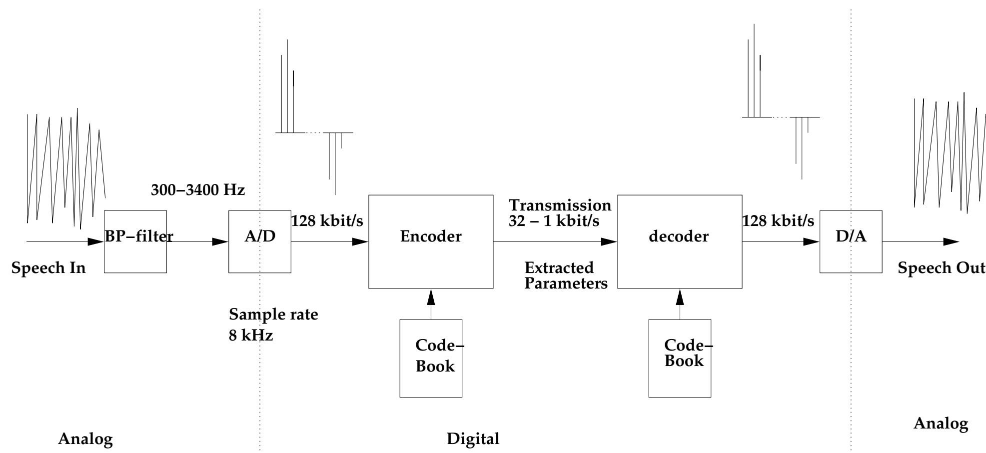

Figure 3.1: Block diagram of a simplified speech coding system.

3.1 Introduction

An overview of a speech coding system can be seen in figure 3.1. The first thing involved in digital speech is sampling and amplitude quantization. For speech the sampling is almost always 8 kHz, i.e. a little more than twice the bandwidth of the speech in an analog telephone network. Before the sampling takes place, the input analog signal is band pass filtered to 300 - 3400 Hz. The amplitude has to be quantized into a binary representation. The representation of the amplitude can be either linear or logarithmic. In a linear representation the step size is always the same between two consecutive samples. In order to get a reasonable small quantization error 16 bits is often used. The 16 bit is also suitable for memory storage. Sampling and representing the amplitude by quantization is called Pulse Code Modulation (PCM). With 8 kHz sampling rate and 16 bits for the quantization, a total bit rate 128 kbit/s is obtained. This is for raw speech and can be reduced to 64 kbit/s by a-law/u-law compression [3]. This is a non-linear quantization where the step size between consecutive samples increases with increasing sample amplitude. This 64 kbit/s PCM is usually used as reference compression technique for comparing lower bit rate speech coders.

Previously, the speech coding has been done through direct quantization of the speech samples. A more efficient way to represent speech is through parametric techniques. Instead of only looking at individual samples, one looks at sequences of samples at the same time. The Adaptive Differential PCM algorithm [4], former [5], works at 4 bit rates, namely 16, 24, 32 and 40 kbit/s. The reduction in bit rate is achieved through adaptive prediction and adaptive quantizers to exploit the redundancies in the speech signal. Some explanation to the expressions used in the text can be found in figure 3.2

3.2 Speech Coding Techniques

Speech coding can be divided into three different groups based on the techniques used. The simplest is the waveform coding. The waveform technique does not necessary apply to speech signals, but any signal can be modeled by this method. The concept is to reassemble the origin wave as exact as possible. The technique is simple concerning memory requirement and MIPS consumption. It works well down to 16 kbit/s.
Another approach to speech coding is to extract parameters that are significant to speech. These parameters are then transmitted and the speech is reconstructed at the receiver side. This technique is called voice coding or simply just vocoding.

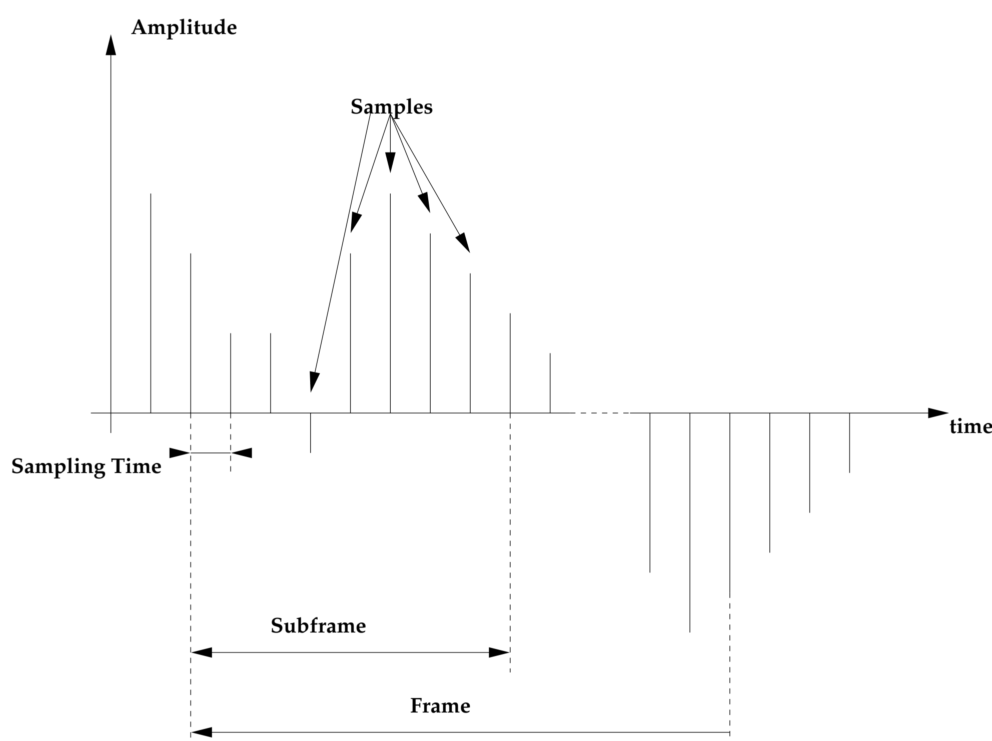

Figure 3.2: Declaration of speech coding attributes.

By using this technique, we can reach bit rates below 16 kbit/s. The remainder of this chapter will be dedicated to different approaches of vocoders.
A combination of both waveform and vocoding exists in the hybrid coder. Here, the advantages of both techniques are combined.

3.2.1 Vocoding

A simple model for synthetic speech generation is shown in figure 3.3. The oral system of human being is modeled by the vocal tract filter. This is a time varying digital filter. A Speech like signal is not stationary, but over shorter periods, 5-20 ms, it can be assumed to be quasi-stationary. The filter coefficients are then updated every 5-20 ms and an excitation signal feeds the filter. The source of the excitation signal is different for periods of voiced speech and unvoiced speech. For voiced speech, the filter is fed by a periodic impulse train of pulses and for unvoiced speech, a random noise generator is used. Most of the algorithm that relies on this model uses a tenth order filter. The method used to calculate the

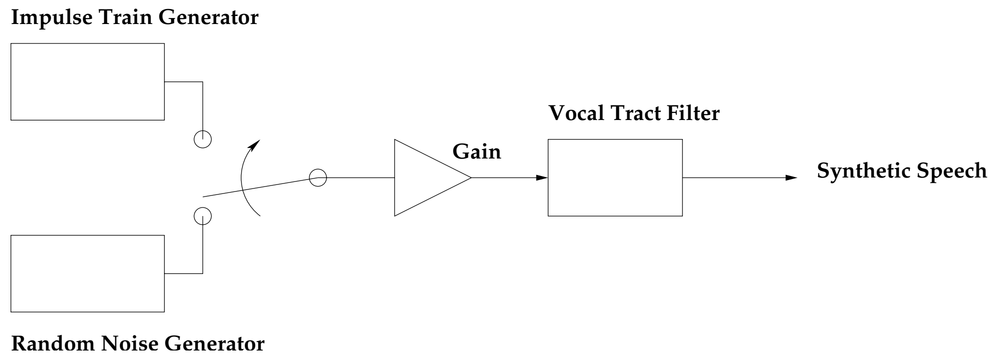

Figure 3.3: Block diagram of a simplified speech synthesis.

coefficients of the filter is known as Linear Prediction Coding (LPC) analysis. The quality and the bit rate of a speech coder are heavily dependent on how the excitation signal is generated and coded. This filter is also known as a short term prediction filter. In the filter the short-term correlation between speech samples is modeled.
An improvement in the model shown in figure 3.3 is the introduction of long term correlation of voiced speech. This periodicity is also known as pitch or pitch period. The pitch is incorporated in the excitation signal to reflect the long term periodicity of speech. The pitch is usually calculated in two steps. First, a rough estimate of the pitch is calculated using cross correlation. Secondly, the pitch is refined by using a closed loop method. In the closed loop method, the speech signal is reconstructed and compared to the original signal. By varying the pitch value around the calculated value from the cross correlation, a more accurate value can be obtained.
From the long-term prediction we know the periodicity of the speech signal. The impulse train of pulses for voiced speech is left to calculate. A couple of different methods exist to determine the positions and their corresponding amplitude.

3.2.2 Multi Pulse Excitation

A straight forward way to determine the pulses in the excitation is to add one pulse at a time. Together with the pitch prediction and the vocal tract filter a speech signal is reconstructed. This reconstructed signal is then compared to the original input signal. By changing the amplitude and the position of this pulse, new speech signals are reconstructed. Then, the signal with the smallest error is chosen. The smallest error contribution is usually based on a Mean-Square-Error (MSE) method. The contribution from this pulse is subtracted from the original

speech signal and a new pulse is determined in the same way. This method of searching for the best pulse positions by speech reconstruction is called Analysis-by-Synthesis (AbS). The drawback of this approach is the large freedom on the amplitude and where to place the pulses. For most algorithms the frame size is used, i.e. the number of pulse position we are looking at, around 40 - 60 pulses. To avoid an extensive search time, restrictions on the pulse positions must be taken and the amplitudes tested must be restricted. In the high rate 6.3 kbit/s of the speech coder G.723.1 [6] a multi pulse scheme is used. By restricting the pulses to be either on even or odd positions, the complexity is reduced. The number of pulses for the excitation is also limited to 5 for odd frames and 6 for even. First, an overall gain for the whole frame is calculated. Then, the amplitudes for pulses within the frame are then chosen around this value.
The GSM Full Rate (FR) speech coder [7] at 13 kbit/s uses a method called Regular-Pulse-Excitation Long-Time-Prediction (RPE-LTP). All the pulses in the excitation signal must be equally spaced and their positions are completely specified by the first pulse.

Codebook excitation

Instead of generating the excitation yourself, the excitation vector can be fetch from a codebook. This codebook does not have to be transmitted, rather it is available at both the transmitter and the receiver. In order to reach good speech quality, the codebook has to be trained and filled with a diversity of different vectors. The larger the codebook, the more vectors can be stored and the potential for better output speech is increased. Unfortunately, a larger codebook consumes more memory and the search for the best vector gets more time consuming. The 4.8 kbit/s U.S. federal standard 1016 [8] also known as Code Excitation Linear Prediction (CELP), uses a codebook for the excitation.

One can reduce the complexity in the vector search by splitting the codebook into smaller ones. This technique is called Vector Sum Excitation Linear Prediction (VSELP) [9]. The codebooks are searched sequentially and then tailored together. One can find this type of speech coders in the Japanese digital cellular standard and in the GSM half rate (HR) coder.

A mixture between the multi pulse excitation and the codebook approach is the Algebraic Code Excitation Linear Prediction (ACELP). For each frame, a couple of pulses are chosen to be non-zero, usually 4 of them. The pulse position for each pulse is read from a codebook. There is one codebook with pulse positions for each pulse and there are no overlapping pulse positions. The freedom on where to place the pulses is limited. The complexity of the codebook is further reduced

by excluding the possibility of mixing even and odd pulse positions. Either all the pulses at are odd positions or at even positions. This technique has been successful in several standards, for example in the lower bit rate 5.3 kbit/s of the G.723.1 [6] speech coder. Also the newer standard G.729 [10] with the bit rates 6.4/8.0/11.8 kbit/s has adopted this technique. A description of these standards can be found in [11]. The Enhanced Full Rate (EFR) of the GSM standard and the Adaptive Multi Rate speech coders for the 3G standard uses this approach.

3.2.3 Multiband Excitation

In the linear prediction coders presented above the voice/unvoiced decisions are made on the whole frame. The frame is the time where a speech signal is treated as a quasistationary signal. In a multiband excitation (MBE) coder the voicing decision is improved and a frame can be declared voiced and unvoiced at the same time. The input speech signal is transformed into the frequency plane, see figure 3.4. After the transformation the spectrum is divided into sub-bands. The sub-bands are then declared voiced or unvoiced independently of each other. This allows the excitation signal for a particular frame to be a mixture of periodic (voiced) and random-like (unvoiced) pulses. This added degree of freedom in the modeling of the excitation signal allows the multiband excitation speech to generate higher quality speech than in the linear prediction model presented above. In addition, the multiband excitation model is more robust to background noise.

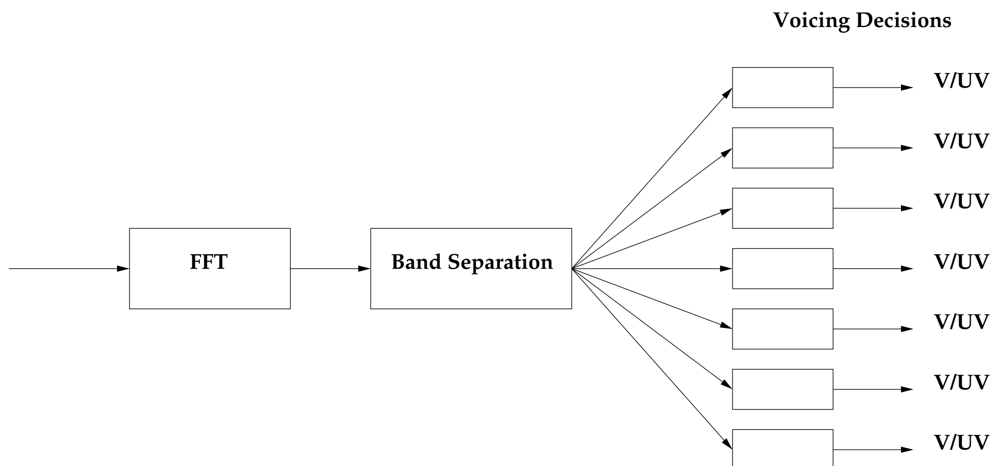

Figure 3.4: Block diagram of the voice/unvoice decision in a multiband excitation coder.

An improved version of the MBE, IMBE, is used in the International Mobile Satellite INMARSAT-M [12]. This speech coder is called IMBE and works at

6.4 kbit/s. A further improved version of the IMBE coder is the Advanced MBE, AMBE, from Digital Voice Systems [13]. This one works down to around 2 kbit/s.

3.3 Complexity Aspects

The ratio in complexity between a coder and decoder is around 1/10. A decoder is much simpler because the model parameters for the speech do not need the computinal intensive extraction, only the synthesis. For a real time system the decoding is hardly an issue. The main focus is the encoding process. From analysis of standard speech coders [14] and [15], the most used operation is multiplication and Multiply-and-Accumulate (MAC). This is the reason why Digital Signal Processors (DSP) are so favorable when implementing speech coding algorithms. Some coders use transforms to move between the time and the frequency domain. Also, for this operation a DSP is suitable, since they usually incorporate hardware for bit-reverse addressing used in Fast Fourier Transforms (FFT). The multiband excitation coders described below are good examples of coders using FFT. The extraction of speech coding parameters can be further divided into two different methods. The first is used for finding LPC coefficients and the pitch and these values are calculated from the input speech signal. The method used is called Analysis-and-Synthesis (AaS). The second method used is for the excitation signal generation. Here, we search through all possible pulse positions and choose the best one. The speech signal is reconstructed in an encoder and compared to the input reference signal. The error signal between the reconstructed and the original signal is then minimized. This method is called Analysis-by-Synthesis (AbS). The time spent searching for the best extraction parameters is usually over 50 % of the total time spent in the encoder. To speed up the encoder process, this excitation parameter search is a place to start. The search procedure usually involves several nested loops and an improvement in the most inner loop can give significant speedups.

Some of the parameters extracted in a speech coding model can not be transmitted directly. For example gain factors and the coefficients for the LPC filter, see figure 3.3, would need too much bandwidth if they transmitted directly. The main approach to reduce the number of bits required is to quantize the values. Two different methods exit; scalar and vector quantization. In scalar quantization the value in a quantization table that is closest to the value under quantization is chosen. The index to the quantization table is then transmitted. This requires the quantization table to be available at both the encoder and the decoder. If many tables or large tables are used, the memory requirements increase. Vector quantization is more efficient if more than one value has to be quantized at the same

time. The ten coefficient of the LPC filter are quantized using vector quantization. These ten coefficients are usually split into smaller blocks, 3-5 coefficient in each. These blocks are then vector quantized. A vector quantization table is searched through and the best entry is chosen due to some error minimization method, like Least-Mean-Square (LMS).

3.3.1 Coding delay

The quality can usually be improved and the number of bits required for the coding can be reduced by looking over a larger range of samples. The drawback of this method is the increased coding delay introduced. The definition of one-way delay is basically the time from when an input sample arrives at the encoder until this sample appears at the output of the decoder. The delay for the transmission of the coded bits is not included in this delay. The delay for a typical speech coding algorithm described above is usually 2-4 frames. The frame lengths differ from 10-30 ms depending on the algorithm used. In addition to the frame length, which is the buffering time for samples, speech coders might use an additional look ahead of 25-50 % of the frame length. The total algorithmic delay is then both the frame length plus the look ahead length. If the delay gets too long, the speakers will be annoyed by this. The limit for the total delay is around 300 ms [11]. This number is reduced if the speakers are in the same room or can hear each other, except from the sound coming through the phone. This case occurs only when speech coding implementations are tried out. There exists a standardized speech coder with a low algorithmic delay, the G.728 [16]. This speech coder has a frame length of only 0.625 ms instead of the usual 10-30 ms.

3.4 Hardware Acceleration Opportunities

Even though speech coders are used in both terminals and in the infrastructure communication gateways, the hardware acceleration possibilities suggested here are most suitable for gateways. The reason is that in a terminal the computing power is usually enough. For a gateway on the other hand, a faster and more power efficient implementation is always of interest. A faster implementation means more channels or less hardware if no more channels are preferred. A more power efficient implementation can reduce the need of cooling devices and cooling devices cost money. From the chapter 3.3 it is clear that any success in hardware acceleration should come from attacking the inner loop search. Most of the time spent in the algorithm is used to searching the excitation parameters. The search is done by testing all the possibilities and the closest fit is chosen. A well

known approach to speed up execution is to do more in parallel. With more than one execution unit and instructions to support parallel execution, this method can increase the performance significantly. The problem with speech coding algorithm is that there is no inherent instruction parallelism. Most of the consecutive instructions are dependent on the results from the previous ones. The use of instruction level parallelism is limited here. On the other hand, on a higher level parallelism is more suitable. For every new iteration in the search for the excitation parameters, the operations from the last iteration are executed again. The only difference between the iterations is the new input data. The main disadvantage is that almost the whole data path has to be duplicated together with the register file. This is costly in terms of hardware. The program code can be the same for both data paths; a typical Single Instruction Multiple Data (SIMD) processor. A less aggressive way is to use an accelerator technique. The innermost kernel of the search can be handed over to dedicated hardware. The processor can continue with the next iteration or simply wait for the result to return back from the accelerator.

So far all the focus has been on the calculations for speeding up the algorithm. Another area, almost as important as the computations itself, is the data feeding. Without suitable hardware for the data fetching and the address calculations the speedup in the actual calculation is less beneficial. In order to feed the processor with data continuously, parallelization of data fetch and arithmetic operations are necessary. Unfortunately, this is not enough, since some parts of a speech coding algorithm uses complex address schemes. This is especially true for the inner loops of the excitation parameter search. A substantial speedup can be achieved by adding more features to the AGU, for example absolute value calculations. This is described more in chapter 7.

3.5 References

[1] A. Kondoz, *Digital Speech*. John Wiley and Sons Ltd, 1994.

[2] A. S. Spanias, "Speech Coding: A Tutorial Review," in *Proceedings of the IEEE, volume 82, Issue 10*, pp. 1541–1582, 1994.

[3] *ITU-T Recommendation G.711, Pulse code modulation (PCM) of voice frequencies*, 1996.

[4] *ITU-T Recommendation G.726, 40, 32, 24, 16 kbit/s adaptive differential pulse code modulation (ADPCM)*, 1990.

[5] *ITU-T Recommendation G.721, 32 kbit/s adaptive differential pulse code modulation (ADPCM)*, 1996.

[6] *ITU-T Recommendation G.723.1, Dual Rate Speech Coder for Multimedia Communications Transmitting at 5.3 and 6.3 kbit/s*, 1988.

[7] P. V. et al, "Speech Codec for the European Mobile Radio System," in *International Conference on Acoustics, Speech and Signal Processing*, pp. 227 –, 1988.

[8] *Federal Standard 1016, Telecommunications: Analog to digital conversion of radio voice by 4800 bit/s code excited linear prediction(CELP), National Communication System-Office of Technology and standards*, 1991.

[9] I. Gerson and M. Jasiuk, "Vector Sum Excited Linear Prediction(VSELP) Speech Coding at 8 kbit/s," in *International Conference on Acoustics, Speech and Signal Processing(ICASSP'90)*, pp. 461–464, 1990.

[10] *ITU-T Recommendation G.729, Coding of Speech at 8 kbit/s Using Conjugate-Structure Algebraic-Code-Excited-Linear-Prediction (CS-ACELP)*, 1996.

[11] R. Cox, "Three New Speech Coders from the ITU Cover a Range of Applications," in *Communications Magazine, IEEE*, pp. 40–47, 1997.

[12] *DVSI. INMARSAT Voice Codec USA*, 1991.

[13] Digital Voice System, Inc *http://www.dvsinc.com*.

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

[15] M. Olausson and D. Liu, "Instruction and Hardware Accelerations for MP-MLQ in G.723.1," in *IEEE Workshop on Signal Processing Systems*, pp. 235–239, 2002.

[16] *ITU-T Recommendation G.728, Coding of speech at 16 kbit/s using low-delay code excited linear prediction*, 1992.

Chapter 4

Audio Coding

4.1 Introduction

The main differences between audio and speech coding are the frequency range and the spread in audio characteristics. Speech is mostly limited to sampling rates at 8 kHz and the sound is supposed to be human like speech. For audio, there are no such limitations on the sound and the sampling frequencies are higher, for example 32, 44.1, or 48 kHz. This makes it more complex to use. Similar to the speech coding case, the encoder and the decoder has asymmetric complexity. The encoding is much more computing extensive.
In speech coding both the encoder and the decoder are specified and there are test vectors associated with the reference code. For audio coding only the bit stream from the encoder to the decoder is specified. There is no need for exact implementations of the encoder and decoder.
While speech coding usually requires both an encoder and a decoder, it is usually enough to use a decoder for playback in audio coding. For example, in mp3 music, the music is compressed once on a device where processing time and power is not an issue. The focus for audio coding is concentrated on an effective implementation of the decoder; the decoder is decoupled from the psychoacoustic model. This makes it possible to modify and improve the encoder without changing the decoder.

4.2 Description of Perceptual Coding

The basic idea behind perceptual audio coding is to extract only the information the human ear can hear. The rest of the information does not need to be transmitted. It is also possible to introduce undesirable sounds, i.e. noise, that was not previously there without hearing a difference. A psychoacoustic model tries

to include this fact. In figure 4.1, there are two parts where the psychoacoustic model is involved.

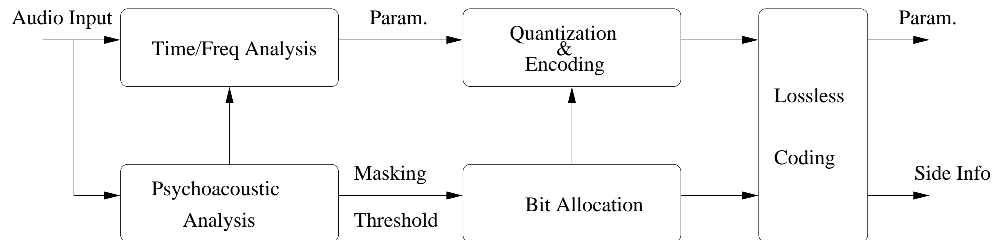

Figure 4.1: An overview of a generic perceptual audio encoder.

First of all in the time/frequency analysis, from which the hearable information is extracted. Frequencies components close to stronger component are masked away. This is known as the masking effect. From the time/frequency analysis only the frequency components are left from the masking. Frequency component masking is shown at the top in figure 4.2. Everything below the dotted line can not be heard by the human ear. The spread of the masking in the frequency domain is increased for increasing masker frequency, i.e. frequencies components can be further apart in higher frequencies than in lower and still be masked away. This is known as the critical bandwidth. From the bottom of the figure the non simultaneous and the simultaneous masking effects of the human are shown. In addition, a frequency component has a spread in the time domain; both from the duration of the masker, but also from the fact that the masking effect does not disappear instantaneously. The latter is known as non simultaneous masking and is further divided up into pre- and post masking. While the duration of the pre masking is for a couple of milliseconds, the post masking can persist up to 100 ms after the masker is removed.

In addition, the masking effect of close frequency components, the absolute hearing of the human ear is also interesting. The total hearing spectrum for the human ranges from 0-20 kHz, with a sensibility peak between 3 and 4 kHz. The sensibility of the ear then decreases when the frequency moves away from this peak. The effects of the absolute hearing are also incorporated in the psychoacoustic model. There is a distinction between a tone masker and a noise masker. A tone in the frequency spectrum looks like a peek, while the noise has a flatter spectrum. For a tone masker to make noise inhearable a higher signal-to-mask ratio is needed than the other way around. The signal-to-mask ratio is the difference in signal intensity between the masker and masked signal. The content of this is that noise is harder to mask away than a tone.
The masking effect is also used in the quantization & encoding box of figure 4.1.

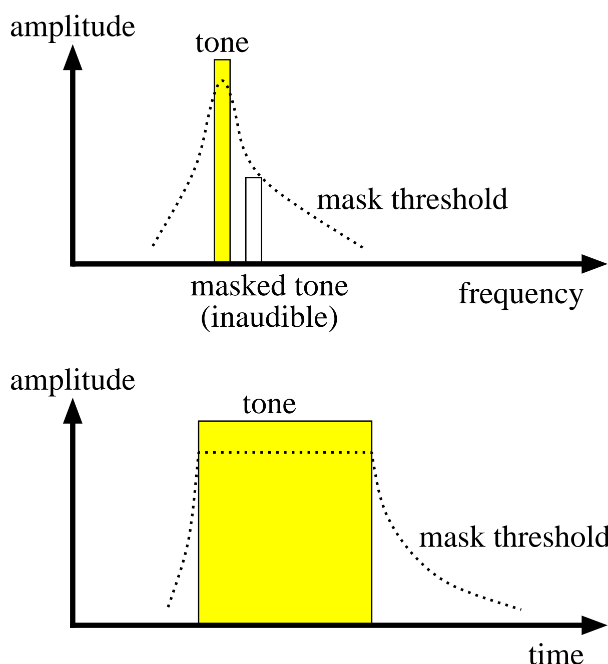

Figure 4.2: Frequency (top) and temporal (bottom) masking effects.

The parameters from the time/frequency analysis must be quantized before transmitted in order to further reduce bit rate. The quantization itself will introduce noise, but as long as this noise is below the masking level of the quantized frequency parameter, it will not be heard. If the noise level reaches above the masker level, the solution is to assign more bits from the bit allocation module. These new bits will lower the introduced noise. For low bit rates the process of assigning bits for the quantization of frequency components can become cumbersome. It is possible to run out of bits and the noise can then not be lowered. The solution is to use a bit reservoir, where unused bits from former encoded frames are used. If no bit reservoir exists or the bits are not enough, then the solution is to simply accept the introduced noise.

4.3 Coding Standard

A summary of different audio coding algorithms are collected in table 4.1. The MPEG 1/2 layer III is more known as mp3 from internet. ATRAC is the algorithm used in minidiscs from Sony.

Algorithm	**Sample Rates (kHz)**	**Channels**	**Bit Rates (kbps)**
APT-X100[1]	44.1	1	176.4
ATRAC[2]	44.1	2	256/ch
Lucent[3]	44.1	1 - 5.1	128/stereo
Dolby AC-2[4]	44.1	2	256/ch
Dolby AC-3[5]	44.1	1 - 5.1	32 - 384
MPEG-1[6]	32, 44.1, 48	1, 2	32 - 448
MPEG-2[7]	32, 44.1, 48	1 - 5.1	32 - 640
LSF	16, 22, 24		
MPEG-2[8]		1 - 96	8 - 64/ch
MPEG-4[9]		1 -	0.2 - 64/ch

Table 4.1: Audio coding standards.

4.4 Hardware Acceleration Opportunities

Even though an audio CODEC contains both an encoder and a decoder, the focus will be on the decoder in this chapter. The reason is the same as mentioned in chapter 4.1; the encoding usually has no real time requirements and it is only done once. The decoding on the other hand will be performed over and over again. Take the example of an mp3 player. The encoding is done once in a desktop machine without any timing requirements. The performance and the power consumption are of less importance. The playback is then done in a battery powered handheld device over and over again. The power consumption, performance and real time requirements become a bigger issue. This chapter will point out possible hardware acceleration opportunities.

The decoding in an audio CODEC is simply running an encoder backwards, see figure 4.1. A psychoacoustic model is not needed in the case of a decoder, neither the bit allocation scheme. There are three major steps in the decoder. The first is reading and interpreting the incoming bit stream. Secondly, all the samples from the bit stream are dequantized and scaled properly, and finally the samples from different subbands are decomposed and transformed to the time domain.

The reading and interpretation of the incoming bit stream contains a lot of bit manipulation. For example, the scaling factors, samples, and pointers to tables are not byte aligned, rather they are assigned only as many bits as they each need. This part of the decoding is preferable done in hardware or in a hardware

accelerator. In a normal CPU bit manipulating operations are complex and time consuming.

In the dequantization step the scaling of the samples can involve difficult mathematics. In MPEG 1/2 layer III calculations are of the form x^y. The fastest way to do these calculations is to use a table lookup, but the table can consume a lot of memory. The second alternative is to calculate the value through an iterative algorithm. The many computational steps and sometimes even the non deterministic time it takes can make this solution too performance hungry.

The last step, transformation from the frequency to the time domain involves some kind of transformation. The choice of the transform can have great influence on the performance. Hardware support for reversed bit addressing together with enough register for intermediate results, can speed up the calculations.

The memory is an issue for both the code and the data storage. The code can be reduced by easy firmware. Fixed point implementations involve scaling of the data to keep as high precision as possible without having overflow situations. This will add extra code to the memory and also slow down the performance. For a floating point implementation the need for scaling is already incorporated in the data format; the separation between the mantissa and the exponent. The standard floating point formats require too many bits to be of any interest. The single floating point format is 32 bits wide and that is not suitable for a small, power efficient solution. In paper [10] a shorter floating point format is used to implement an mp3 decoder. The firmware can be simple due to the floating point format and still keep the data sizes reasonably small for an efficient memory usage.

4.5 References

[1] F. Wylie, "Predictive or perceptual coding...apt-X and apt-Q," in *Proceeding of 100th Conv. Aud. Eng. Soc. May*, 1996.

[2] T. Yoshida, "The rewritable MiniDisc system," in *Proceedings of the IEEE, volume 82, Oct*, pp. 1492–1500, 1994.

[3] S. D. D. Shinha, J.D. Johnson and S. Quackenbush, "The perceptual audio coder(PAC)," in *in The Digital Signal Processing Handbook, V. Madisetti and D. Williams, Eds. Boca Raton, Fl: CRC Press*, pp. 42.1–42.18, 1998.

[4] L. Fielder and G. Davidsson, "AC-2: A family of low complexity transform-based music coders," in *in Proceeding of 10th AES Int. Conf. Sept*, 1991.

[5] M. Davis, “The AC-3 multichannel coder,” in *in Proceeding of 95th Conv. Aud. Eng. Soc., Oct*, 1993.

[6] *ISO/IEC-11172-3, Information technology - Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to About 1.5 Mbit/s - Part 3: Audio*, 1992.

[7] *ISO/IEC-13818-3, Information technology - Generic coding of moving pictures and associated audio - Part 3: Audio*, 1998.

[8] *Information technology – Generic coding of moving pictures and associated audio information – Part 7: Advanced Audio Coding (AAC)*, 1992.

[9] *Information technology – Generic coding of moving pictures and associated audio information – Part 3: Advanced Audio Coding (AAC)*, 1992.

[10] J. E. M. Olausson, A. Ehliar and D. Liu, “Reduced Floating Point for MPEG1/2 Layer III Decoding,” in *International Conference on Acoustics, Speech and Signal Processing*, 2004.

Part III

Implementation

Chapter 5

Hardware

5.1 Difference between Speech and Audio CODEC

Speech and audio coding processors both deal with sounds and try to compress them as much as possible without too much quality degradation. There exist certain differences between these two. The first difference is the application area. While speech CODECs are designed especially for speech like sounds, their ability to reproduce other types of sound is limited. For an audio CODEC on the other hand the whole sound spectrum that is hearable by the human ear is covered. This fact also influences the sampling frequency, which for a speech CODEC is fixed to 8 kHz. Speech CODECs with other sampling frequencies exist, but they are very rare. For an audio CODEC the sampling frequency and the corresponding frequency range is much more diverted; 32, 44.1 and 48 kHz are common in the MPEG-1 or MPEG-2 audio standard [1], [2]. Another major difference is the encoder/decoder usage. For a speech CODEC both the encoder and the decoder is mandatory in an implementation, for example a cellular phone. The user wants to be able to both encode and decode speech-like sounds. In the audio CODEC case only the decoder is necessary. The reason is that the user is only interested in playing the compressed audio. Encoding is done once and the encoded audio can be decoded many times. The best example is an mp3 player. The music in this case is compressed in a machine with high performance and there are no real time requirements involved. The real time requirement is only applicable to the decoding. In the speech CODEC example the real time requirement is applicable for both the encoding and the decoding. The separation of the encoder/decoder for an audio CODEC also influences the testing requirements. For a speech CODEC both the output from the encoder and the decoder are specified through carefully designed test vectors. In order to be called a speech CODEC for a certain speech coding algorithm one must have a bit exact implementation according to the test

vectors. For an audio CODEC the requirements are more relaxed. Here, it is only the bit stream out from the encoder that matters. The output from the decoder is not specified with bit exact test vectors, but there are compliance tests. These tests will ensure the output quality to a certain limit of the decoder. The best way to test an audio decoder is to use subjective tests where people really listen to the output from the decoder and compare it to the output from a reference decoder. The quality of the sound from an audio CODEC is higher than from a speech CODEC. The audio CODECs are used in entertainment devices, while speech CODECs only deal with speech. From a computing point of view the audio CODECs are more time consuming; the decoder itself needs more performance than an encoder and decoder of a speech CODEC. The comparison is tough to make, because of the bit rates and the differencing quality. The bit rates for an audio CODEC are usually around 10 or more times higher. While almost all audio CODECs work in the frequency plane, transformation from the time to the frequency domain is required. For a speech CODEC this is not always the case. A summary o the differences between audio and speech coding can be found in table 5.1.

Subject	**Speech CODEC**	**Audio CODEC**
Sampling frequency	8 kHz	for example 32, 44.1, 48 kHz
Testing	test vectors	subjective listening tests
Real time req.	Both encoder and decoder	Only for decoder
Type of sounds	Only speech like	Most sounds

Table 5.1: Differences between audio and speech CODECs.

5.2 References

[1] *ISO/IEC-11172-3, Information technology - Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to About 1.5 Mbit/s - Part 3: Audio*, 1992.

[2] *ISO/IEC-13818-3, Information technology - Generic coding of moving pictures and associated audio - Part 3: Audio*, 1998.

Chapter 6

Architectures

6.1 Architectures for Speech and Audio Coding

There are two different places where the encoding/decoding takes place. One is the terminal and the other is in the base station. They will both do the same thing; encoding and decoding. The requirements on them are though different. While a terminal usually only have to serve only one channel, a base station has to serve multiple. The power consumption for a terminal might be crucial because many of them are battery powered. A long battery lifetime is definitely a powerful market argument. In a base station the power consumption is also important, but the reason why is different. It is not the power consumption itself that are important, rather the heat generation from the chips. The more heat they generate, the more cooling devices have to be applied and, in turn cooling devices cost money. In the subsiding section, three different approaches of implementation will be evaluated; a fully programmable, an FPGA, and an ASIC.

6.2 Programmable

The advantage with a fully programmable solution is the ability of late changes and late updates. The power consumption is low, but the lack of application specific instruction might give cumbersome solutions. For audio and speech coding a DSP is favorable over a microprocessor. The DSP has specialized instruction for signal processing operations: multiply-and-accumulate, zero overhead looping, and address generator units. While both speech and audio coding are heavy users of these instructions, a DSP is the wisest choice. The biggest vendor of DSPs are Texas Instruments [1] with their C54xx and C55xx series for terminals and C6xxx series for base stations. Analog Devices [2] is also a big vendor of DSPs. Their ADSP 218x family has a very user friendly assembler and is suitable for

terminals. The ADSP2153x, also known as Blackfin, is their latest member in the DSP family. The architecture is very interesting because it is both a DSP and a micro processor within the same product. There are of course other manufactures of DSPs. For example the jointly developed DSP core StarCore from Motorola and Lucent [3].

6.2.1 General and Embedded DSP

The general-purpose programmable DSP market is the best known and is dominated by four companies: Agere, Analog Devices, Motorola SPS and Texas Instruments. The embedded DSP market, on the other hand, is served by over 100 chip vendors providing DSP technology in the form of ASSPs, ASICs, FPGAs, RISC/DSP combos, DSP-enhanced MPUs, DSP-enhanced RISC cores and even DSP state machines. These embedded DSP markets are dominated by companies like Qualcomm, Broadcom, Infineon and Conexant, and many less known companies.

6.3 FPGA

A popular approach nowadays is the reconfigurable architecture. The power consumption is higher and the performance is lower than for an ASIC implementation. One very important factor is the flexibility. Late changes in the construction are now possible. The algorithm used in the communication systems evolves over time and this leaves the designer with great flexibility. The possibility to move parts of an algorithm between hardware and software is also appealing. For terminals where the power consumption needs to be small, a reconfigurable device is not a choice, at least not for the moment. Today, an FPGA is not only pure logic. You can find cores with predefined blocks like processors and multipliers. This makes it much easier to make constructions with them. The big vendors of FPGAs are Xilinx [4] and Altera [5].

6.4 ASIC

For a long time, the only choice for a base station was an ASIC implementation. The reason was that the programmable and the reconfigurable architectures did not give the required performance. The major problem with an ASIC is the impossibility to make changes. If the algorithms update or if a bug is discovered in the system, it is both expensive and difficult to change them. The design time

for an ASIC solution is longer, mainly due to the long construction, testing and manufacturing time of the ASIC itself.

6.5 References

[1] Texas Instrument *http://www.ti.com*.

[2] Analog Devices *http://www.analog.com*.

[3] Starcore *http://www.starcore-dsp.com*.

[4] Xilinx *http://www.xilinx.com*.

[5] Altera *http://www.altera.com*.

Chapter 7

Research Methodology and Achievements

This chapter describes the research methodology and the current achievements. After the presentation of the methodology, a couple of problems are stated. These problems are then discussed and finally presented in the four papers presented in the next section.

7.1 Profile the Algorithm

The best way to evaluate a new algorithm is to do profiling. The profiling will answer the following question: What kind of instructions is used and how often? In profiling, make a list of all the instructions needed for the algorithm and count every occasion of them when running the algorithm. The algorithm has to be run with different input stimuli to get the variations of the algorithm, see figure 7.1. First, all the arithmetic, logic, and shift instructions must be incorporated. Examples of instructions are, addition, subtraction, multiplication, shift, and, or etc. These are easy to identify within an algorithm. Secondly, a weight is assigned to every identified instruction. For example, a multiplication might take more than one clock cycle to execute and division usually takes one clock cycle per bit in the quotient. All the operations are assumed to work on variables stored in the register file. Unfortunately, this is not the case, since most of the variable first has to be fetched from the data memory and the result sometimes has to be stored back into the data memory again. On the other hand, it is too pessimistic to assume that all variables have to be fetched from the data memory and all the results have to be stored back into the data memory after each usage. Variables used very often and temporary variables are stored in the register file. In addition, arithmetic instructions occasionally have the possibility to do data fetches at the

same time as the operation is executed. While Multiply-And-Accumulate (MAC) is the most common operation in signal processing applications, concurrent dual data fetch and multiplication together with accumulation is possible. Another difficult operation is the branch operation. The calculation of the condition itself is not complicated, rather it is the jump instruction that causes problems. The deep pipeline of the processors of today make it difficult to fill the delay slots with useful instructions; the deeper the pipeline, the more delay slots are introduced. The cycle cost and the code cost for a conditional jump can be difficult to estimate. Another approach to filling out the delay slots with useful data is a delayed branch. While the conditional statement is executed, the pipeline is stalled and no new instructions are fetched. The advantage of this is that it is not necessary to fill out the delay slots with no-operation (nop) instructions and code memory can be saved. A more aggressive approach is a branch predictor. Based on static or dynamic statistics, the outcome of the conditional statement is predicted before it is calculated. If the prediction turns out to be correct, the execution continues as usual, but if the prediction is wrong, the pipeline is flushed and the execution starts over from the correct position.

7.1.1 Stimuli to the Algorithm

For the speech coding algorithms profiled, G723.1 and G.729, in [1] and [2], the input stimuli is taken from the test vectors to each speech coder. The test vectors are designed to cover most of the paths within the algorithm, i.e. all the branches should be tested. These test vectors do not guarantee the worst case performance of the algorithm, which usually is rather difficult to extract. In addition to theses test vectors, normal speech should also be incorporated in the statistics. We used only one speech file in English, more would have been preferable and more languages as well.

7.2 Acceleration on Instruction Level

When the whole algorithm is profiled, it is time to identify the time consuming parts. (In paper [1] such a profiling table can be found in the end of the paper.) The major parts of the computation are dedicated to multiplication and accumulation. If the operation already is single cycled, a performance increase can be achieved by duplicating the multiplication data path. In order to feed the multipliers with data, the bandwidth to the memory has to be extended as well. Merging instructions is a technique used when two or more instructions are executed after each other very often. For example, an addition might often be followed by a right shift. When this occurs, it is advantageous to merge these two instructions into

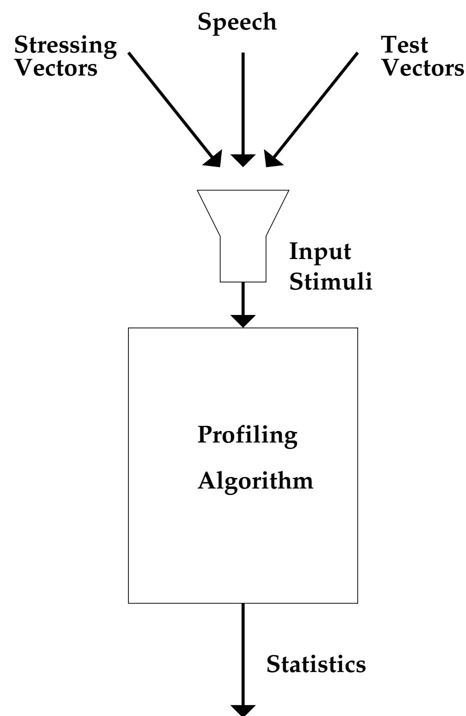

Figure 7.1: The algorithm under test is fed with various kinds of input stimuli.

an add-shift instruction. Very little new hardware is needed because addition and shift is already present as single instructions. A new data path from the output of the adder to the input of the shifter, together with extended decoding are the only new hardware needed, see figure 7.2.

Take into account that when merging instructions may not increase the critical path through the processor. If the critical path is increased, the overall clock frequency for the processor is decreased. In total this can decrease the overall performance of the algorithm. In [1], a more complicated instruction merge is described. While many speech coding algorithms are based on the Analysis-by-Synthesis concept, see chapter 3.2, a lot of the computations involve a comparison statement in the end. When the computations are completed the result is compared to the “best value” so far. The “best value” refers to smallest error compared to a reference signal. If the new result is better than the former “best value”, both the new result and the loop index must be stored. Instead of using an approach with an if-statement, everything is merged into one instruction. The savings from this merge is substantial while the comparison statement is located in the inner part of several loops. This comparison statement will be entered up to 10000 times.

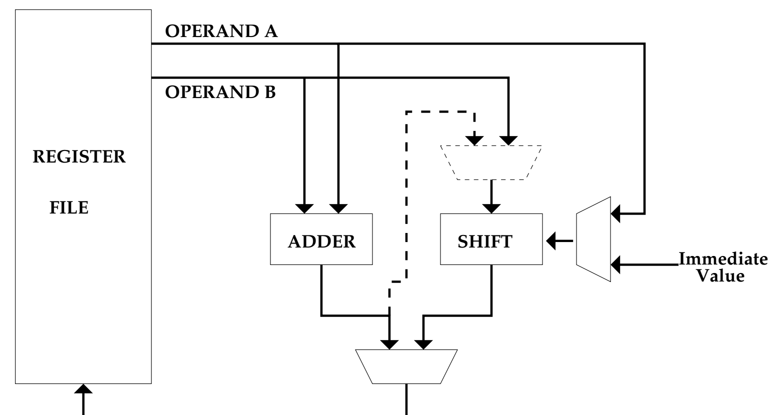

Figure 7.2: The dashed line represents the added hardware for merging the add and the shift instruction.

7.3 Acceleration for Address Calculation

Addressing operands in the data memory can be done by using a register in the general purpose register file. If consecutive operands have to be fetched, the register must first be updated to contain the correct address value. With an Address Generator Unit (AGU), this can be done simultaneously as the operand is fetched from the memory. The update can either be incrementing or decrementing the address by a specified value without utilizing clock cycles from the CPU. To further improve accesses to the data memory, modulo addressing is popular. When the address register value reaches a specified value it wraps around and a start value is loaded into the register. These address specific operations are not applied to the whole register file, rather they are applied to a specific part of the register file called address registers, see figure 7.3.

An even more dedicated address scheme in [1] and [2], include address pointer, absolute calculations, and segment based address calculation. This address scheme is also found in the inner loop of the algorithm and the savings are substantial. In [2] the address calculation concept is further improved. From the address calculation a data dependent control signal is wired to the multiplier. Based on this control signal, the multiplier can decide whether the multiplication should take the input operand as multiplicand or the zero value. The zero value is used for the case that the multiplication is illegal due to operand fetch outside their buffer range. This extra control signal to the multiplier removes a lot of unnecessary multiplications and makes the program code easier.

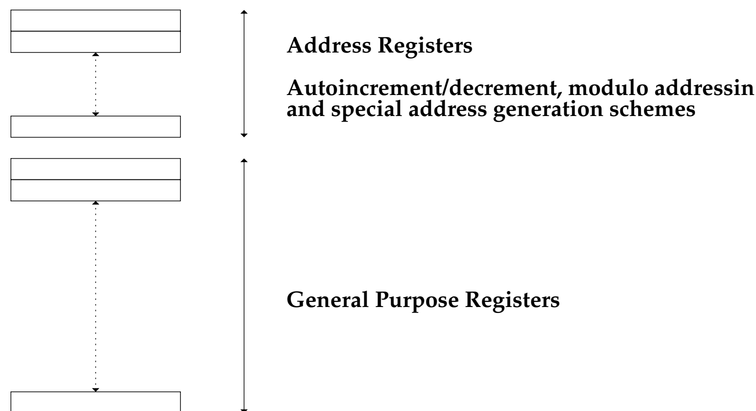

Figure 7.3: The classification of the registers in the register file. The address registers are usually a subset of the general purpose registers.

7.4 Function Level Acceleration

Acceleration can be done on a larger scale than instruction level. By identifying tasks that is either time consuming or too complex for the main processor, extra hardware can be added for this specific task, see figure 7.4. When the processor reaches the hardware accelerated task in the execution, it handles over the needed data to the hardware accelerator. While the hardware accelerator is executing the processor can either wait for it to finish in idle mode or continue executing. From a performance point of view it is better if the main processor can continue its execution. This may be difficult in reality, while the inherent parallelism of the algorithm is low. The main processor then has to wait for the result from the hardware accelerator before it can continue executing. By using the hardware accelerator technique, a substantial speedup can be achieved. The number of hardware accelerators is not limited to one, instead more hardware accelerators can speed up the execution even more. More than one can be executing at the same time. The main drawback with the hardware accelerators is the more complicated firmware.

7.5 Data Format and Precision

The format on the data will influence the overall performance and the memory requirements. For most applications where power is an issue the fixed point format is dominant. The data widths of the fixed point formats are usually a power of 2, for example 8, 16, or 32 bits. Within the fixed point format the binary point is implicit, i.e. you have to keep track of the binary point by your self. In figure

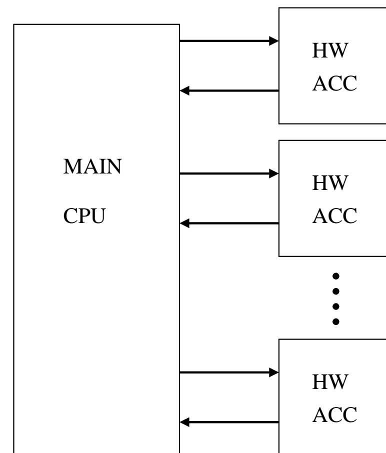

Figure 7.4: The main processor and the hardware accelerator block.

7.5 is the fixed point number 8 bits wide. The binary point is located between bit 5 and bit 6. Everything to the left of the binary point is integer and to the right is fractional. This example is also known as a 2.6 format, i.e. 2 bits for the integer part and 6 bits for the fractional part. In a signal processing system there is always a trade off between dynamic range and precision. The dynamic range sets the limit on the maximum numbers, both positive and negative, that can be represented. The precision on the other hand gives the smallest step size between two consecutive samples. For a fixed number of bits the dynamic range will be traded for a higher precision and vice versa. The drawback is that small numbers in comparison to the highest possible representable, will have very few bits for the precision. This can partly be solved by intelligent scaling. Unfortunately, the scaling is time consuming and it makes the firmware more complex. The contradiction with dynamic range and precision is separated in the floating point format, see figure 7.5. In the floating point format the exponent and the mantissa are separated from each other. The number of bits available for the precision is constant and due to the normalization when all the arithmetic operations are completed, the maximum precision is always achieved. The normalization shifts the bits in the mantissa to always having a decimal value between 1 and 2. All the bits in the mantissa are fractional, but there is an implicit one within the mantissa. The traditional way for implementing speech and audio coding algorithms is using fixed point arithmetic. In [3], an implementation of an mp3 decoder is presented. Instead of using fixed point arithmetic for the calculations, reduced floating point

arithmetic is used. The intended audience for the mp3 decoder is not fastidious listeners, rather an ordinary music consumer with descent requirements on the quality.

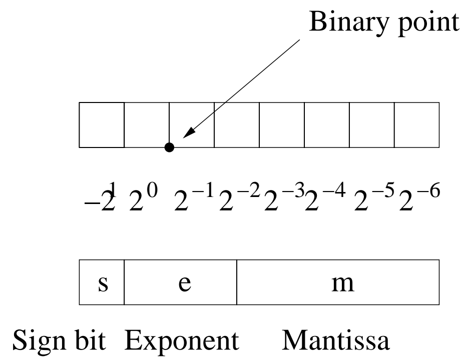

Figure 7.5: Fixed and floating point arithmetic representation.

The basic idea behind this approach was that the precision could be limited but the dynamic range still needs to be sufficient high. The solution to this problem was a floating point approach. The IEEE floating point format for single precision uses 32 bits [4]. These are too many bits, because there already exists fixed point implementation that requires only 20-24 bits. The reference code for the MPEG 2 layer III decoder was modified for the new floating point format. The mantissa and the exponent could be set independently of each other. Later on, this code was extended to admit different sizes on the internal and external data representation. The external representation applies to data stored in memory and the internal representation is the data within the data path. As a first measurement of the quality, the MPEG compliance test was used [5]. To be called a fully compliant audio decoder, the Root Mean Square (RMS) level of the difference signal between the reference decoder and the decoder under test should be less than $2^{-15}/\sqrt{12}$ for a sine sweep signal 20Hz - 10 kHz with an amplitude of -20 dB relative to full scale. In addition, the difference shall have a maximum absolute value of no more than 2^{-14} relative to full scale. To be referred to as a limited accuracy decoder, the RMS level of the difference signal between the reference decoder and the decoder under test should be less than $2^{-11}/\sqrt{12}$ for a sine sweep signal 20Hz - 10 kHz with an amplitude of -20 dB relative to full scale. There are no requirements on the maximum absolute difference. The first intention of having the same data format both for the internal and for the external representation turned out not to fulfill the compliance testing. If the internal representation was changed, it was possible to reach the limited accuracy level, see figure 7.6. In addition to the compliance

test, we also listened to the decoded output.

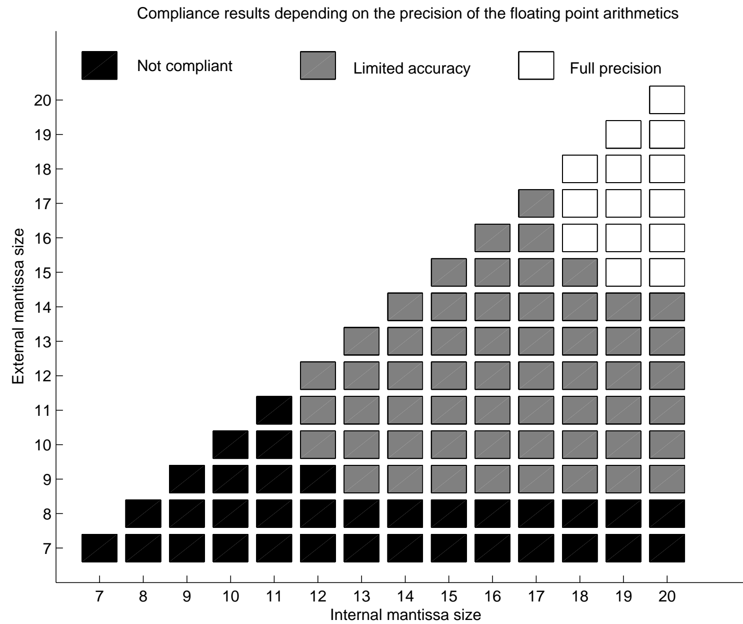

Figure 7.6: Compliance test for different sizes of the internal and external mantissa.

As a final result, the implementation uses only 16 bits in total for the external representation of variables. This makes it suitable for a 16-bit word oriented memory. In order to enhance the quality and reach the level for limited accuracy, the variables use 20 bits in the data path. As long as the variables are within the processor the bigger format is used, but as soon as they are moved out to memory they are rounded off to 16 bits.

7.6 Compact Data Storing

The size of a variable is usually assigned 8, 16, or 32 bits. Variables that differ from these formats are rounded up to nearest power of 2, i.e. 8, 16, or 32. The reason is the load and store instructions of embedded processors that only support these formats. In addition, a high level language like C, only has support for these formats. There are situations where a more flexible view of the variables is desirable. Many compression algorithms, for example the data compression algorithm V.42bis [6] widely used in modems, uses variable data widths that are not a power of 2. The basic concept behind the data compression algorithm is

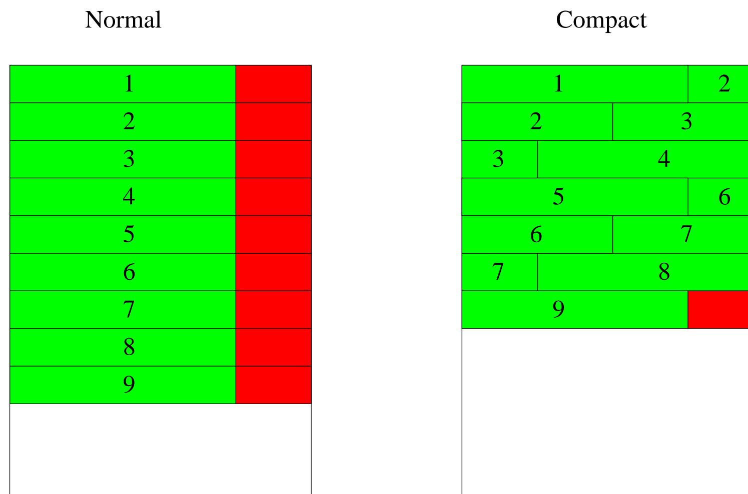

Figure 7.7: The data memory with and without compact stored data. The dark boxes mark wasted memory.

to store sequences of bytes as codewords. From figure 7.7 one can see that there are two alternatives to store data not aligned to a power of 2. To the left is the memory consuming way of rounding up variables to the nearest power of 2. This makes it is easier for the programmer, but parts of the memory will be filled with useless data. The other way is the compact way, shown on the right in the figure. The memory utilization is 100%, but this comes to an expense of a more complicated firmware. On the left example in figure 7.7 one memory address equals one variable. For the compact storing the right variables can span over two memory locations. Furthermore, the variable needs masking to get rid of adjacent variables. In the fourth paper [7] a solution to this problem is presented. To a general 32-bit CPU [8] two custom instructions are added. These two instructions make it possible to load and store variables of arbitrary bit widths. The upper limit is 32 bits, the same as the width of the data path. The modifications in the processor itself are small. All of the instructions available before the modifications begun are still present, but three new signals out from the processor is added, see the bottom of figure 7.8.

These signals go to an extra hardware block, from now on called Bit Memory Controller (BMC). The BMC takes care of all the loading and storing. Data to and from the processor comes directly from a 32 bit register. The alignment and masking is done within the BMC. When data reaches the processor it is interpreted as a 32 bit variable and the ordinary data path can be used. The data stored in the memory can be interpreted in three different ways when it is loaded into the processors register file. The first one is as a fractional number and the data is then

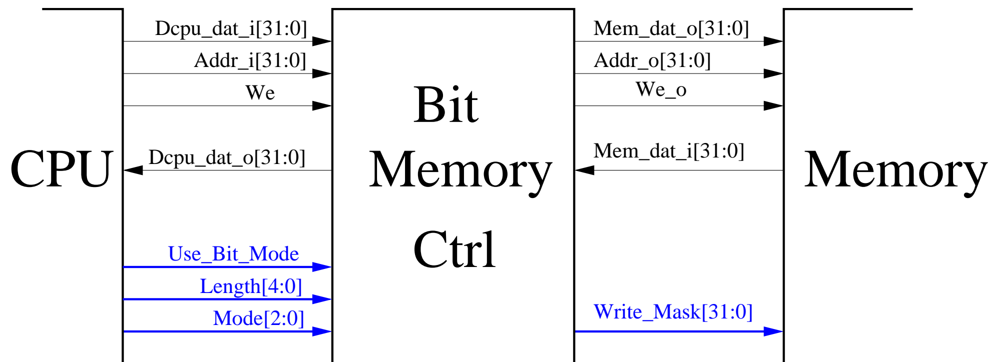

Figure 7.8: The memory controller with the three new signals, Use_Bit_Mode, Length[4:0], and Mode[2:0], coming from the processor.

loaded into the upper part of the register. The least significant bits are set to zero. Secondly, as unsigned integer number, in which the data is loaded into the lower part of the register and the most significant bits are set to zero. The last possibility is as a signed integer number, where the data is loaded into the lower part of the memory, but the most significant bits are sign extended. Figure 7.9 shows these three possibilities.

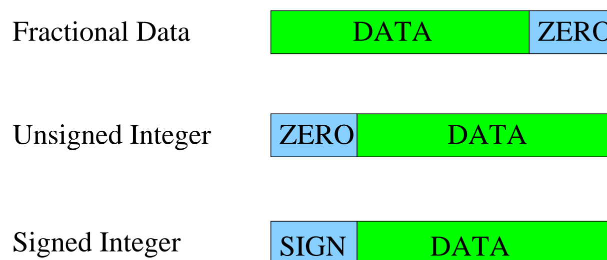

Figure 7.9: The three different loading modes for data loaded into a register.

Within the ordinary load and store instructions one can specify address register, source/destination register, and 16 bit address offset. For the custom load and store instructions 8 bits from the address offset are discarded. Five of these bits are for length indication of the variable and three bits are for the variable interpretation mode, fractional/integer and signed/unsigned mode. Figure 7.10 shows

the principle for a load and custom load instruction. The same idea applies to the custom store instruction.

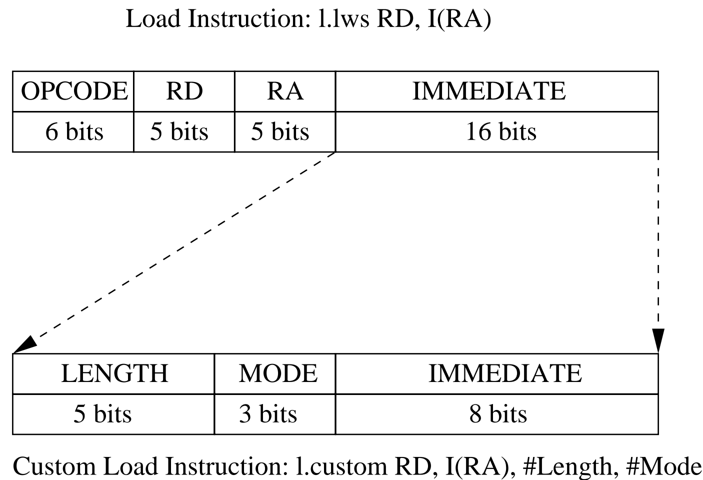

Figure 7.10: The instruction encoding for the custom load instruction (bottom) compared to the ordinary load instruction (top).

7.7 Advantages and Disadvantages of Hardware Acceleration

From the sections above it is clear that the acceleration techniques described are important. Instruction level acceleration both decreases the code size and lowers the execution times. Address calculation acceleration have the same effect as instruction level acceleration on both the code size and the execution times. Speeding up the execution and still keep the flexibility is favorable goal. The major problem with the instruction level acceleration and the address calculation acceleration techniques are the more complicated firmware. The need for programming parts of the algorithm in assembler increases. It is difficult for the compilers to keep up with all the new instructions and even worse with the address calculation hardware. The introduction of these acceleration techniques also requires the simulators to update. The simulator for the processor core must be expandable for all the new hardware. The same thing applies to the rest of the tool set, assembler, linker. From a user point of view the greater diversity of the processors, i.e. Application Specific Instruction Set Processors, can make processor changes difficult. When a processor is not supported any more or the processing power

of the utilized processor is not enough, a new processor much be chosen. If a lot of accelerating instructions are used the move to the new processor requires that a major part of the code is rewritten. The backward compability can be hard to retain when a lot new requirements are coming up from both new and evolving algorithms.

7.8 References

[1] M. Olausson and D. Liu, "Instruction and Hardware Accelerations in G.723.1(6.3/5.3) and G.729," in *The 1st IEEE International Symposium on Signal Processing and Information Technology*, pp. 34–39, 2001.

[2] M. Olausson and D. Liu, "Instruction and Hardware Accelerations for MP-MLQ in G.723.1," in *IEEE Workshop on Signal Processing Systems*, pp. 235–239, 2002.

[3] J. E. M. Olausson, A. Ehliar and D. Liu, "Reduced Floating Point for MPEG1/2 Layer III Decoding," in *International Conference on Acoustics, Speech and Signal Processing*, 2004.

[4] *IEEE Standard 754 for Binary Floating-Point Arithmetic*, 1985.

[5] *ISO/IEC-11172-4, Information technology - Generic coding of moving pictures and associated audio - Part 4: Compliance testing*, 1998.

[6] *ITU-T Recommendation V.42bis, Data compression procedures for data circuit-terminating equipment (DCE) using error correction procedures*, 1990.

[7] M. Olausson and A. Edman, "Bit memory Instructions for a General CPU," in *International Workshop on System-on-Chip for Real-Time Applications*, 2004.

[8] Open RISC 1000 *http://www.opencores.com/projects/or1k/Home*.

Part IV

Papers

Chapter 8

Paper 1

Instruction and Hardware Accelerations in G.723.1 (6.3/5.3) and G.729

Mikel Olausson and Dake Liu

Department of Electrical Engineering

Linköping university,

SE-581 83 Linköping, Sweden

{mikol,dake}@isy.liu.se

Proceedings of the 1st IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)

Cairo, Egypt, December, 2001

Instruction and Hardware Accelerations in G.723.1 (6.3/5.3) and G.729

Mikael Olausson and Dake Liu
Department of Electrical Engineering
Linköping university,
SE-581 83 Linköping, Sweden

Abstract

This paper makes accelerations on instruction level based on the three speech coding algorithms G.723.1, 6.3 kbit/s and 5.3 kbit/s and G.729 8 kbit/s with hardware implementation. All these three algorithms are proposed by the H.323 standard together with G.711 64 kbit/s and G.728 16 kbit/s. The work has been done by thoroughly examining the fixed point source code from ITU, International Telecommunication Unions [1], [2]. Three hardware structures are proposed to increase the performance.

8.1 Introduction

The market for voice over Internet protocol, also called VoIP, has increased over the years. Voice has been a natural choice of communicating for a long long time and will continue to be so. The H.323 standard contains four different speech coders with different complexity and bit rates. The first one is G.711, which is mandatory and uses A/u-law compression at 64 kbit/s. Another coder is the G.728 Adaptive differential PCM (ADPCM) at 16 kbit/s. The last two are more interesting if we are dealing with bandwidth limited transmission channels. These are G.723.1 and G.729. While the first one have two different bit rates specified, 6.3 and 5.3 kbit/s, the last have three different, 6.4/8.0/11.8 kbit/s. These two both have parts that are common, but also parts that differ a lot. From a market point of view it is of highest interest to make the implementations of these algorithms as efficient as possible. A couple of factors may influence the choice of algorithm. For example some users want to squeeze as many channels as possible on a limited transmission channel. Then their choice is as low bit rate as possible if the speech quality is good enough. Others might use them in battery powered applications and their aim is low power consumption by reduced complexity with reduced speech quality as a tradeoff. Others might aim for high speech quality with limited bit rate. This paper will point out some factors that will influence the choice of speech codec from hardware and complexity point of view. The examination

is done from a behavior approach where we are not bound to certain hardware manufacture. The quality or the robustness will not be treated.

8.2 General description of G.723.1 and G.729

The first difference between the G.723.1 and the G.729 is the frame size. While the G.723.1 is based on 30 ms(240 samples), the G.729 is based on 10 ms(80 samples) frames. The delay of the algorithms are 37.5 ms and 15 ms respectively. The fixed codebook extraction is the most exhaustive part of the whole algorithm. In these two codecs there exist two different approaches. One which are used in both the lower bit rate of G.723.1 and in G.729 and is Algebraic-Code-Excited-Linear-Prediction, (ACELP). This ACELP places at most 4 non-zero pulses within a subframe. A subframe is 5 ms long in G.729 and 7.5 ms in G.723.1. The positions are determined from the codebook. The second approach is to use Multi-Pulse Maximum Likelihood Quantization (MP-MLQ). This one is used in the higher bit rate of G.723.1. In this case you have more opportunities to place the pulses more individually and not based on an algebraic codebook.

8.3 Statistics of basic operations

All the arithmetic and logic functions like add, sub, shift, multiply and so on are implemented with a standard C library. This makes it simply to do statistics over how many times different functions are used. Additional to this, the C code has been thoroughly examined and all the branch, loop and move functions have also been identified and classified. All these statistics over basic operations, branch, loop and move instructions give a good hint on where to find improvements on instruction and architecture level. The statistics are presented in table 8.4 in the end of the paper. The table corresponds of three columns of numbers. The statistics for each speech coder with both the average and maximum number of times each operations occurs in a frame. The stimuli used for the speech coders are the test vectors included with the C code from ITU [1], [2]. All the statistics are calculated from the encoders only, while the encoding part is the most time consuming.

8.3.1 Description of the operands

We can see that the most used function is the multiply-and-accumulate, L_MAC. In table 8.4 we have not made any distinction between accumulation with addition

Operation	**Explanation**
ABS_S	16 bit absolute value.
MULT	16-bit multiplication with 16-bit result.
L_MULT	16-bit multiplication.
L_MAC	16-bit multiplication and accumulation.
MULT_R	16-bit multiplication and rounding.
L_SHL	32-bit left shift.
L_SHR	32-bit right shift.
L_ABS	32-bit absolute value.
DIV_S	16 by 16 bit division.
L_MLS	32 by 16-bit multiplication.
DIV_32	32 by 16-bit division.

Table 8.1: Explanation of some of the operations in table 8.4.

or subtraction. This is not significant from hardware point of view, while most DSP's incorporate both this functions. Here comes some explanation to the table.

The extension _A stands for multiplication with equal input to both operand x and y, for example when performing autocorrelation. Extension _I stands for integer multiplication, i.e. multiplication without proceeding left shift. All the other multiplications are fractional. The basic 16-bit operations like, addition, subtraction, shift, round, negate and so on are left out from the table of statistics. They are almost always present in a DSP.
The second part of the table deals with branch instructions. Except from the total number of them, some special cases has been sorted out. First of all is a distinction between the comparison statements made. We distinguish between 16-bit and 32-bit as indicated by the second part of the word in table 8.4, MOVE_16 and MOVE_32. The last part of the word, _COND, _CONDA and _COND_I, stands for special cases of branch instructions. _COND means conditional move, _CONDA is conditional move with absolute value of the operand before the comparison statement is executed and the last one, _COND_I, means conditional move of both operand and loop counter. Absolute value calculation of the operand before branch comparison is optional in this operation. The operation TOTAL BRANCH in table 8.4 is the total number of branch statements found in a frame. The third part of table 8.4 describes the number of loop operations. The last part of 8.4 is the number of data movements within a frame. This includes all data movements from clear, set update and move data. No distinction is made between 32-bit and 16-bit move.

8.3.2 Investigation of the statistics

If we look at the L_MAC operations of table 8.4 more deeply, we will find that around 20% of all MAC operations actually are integer multiplication in the 6.3 kbit/s of G.723.1 and over 33% in the 5.3 kbit/s case. When we look at G.729, there is no need at all for integer multiplication. This means that the multiplier unit must have two modes of operation, fractional and integer in the G.723.1 case. Also from table 8.4, around 2%-8% of the MAC operations are of the type autocorrelation, this means that the same word must be fed into both the x and y operand of the multiplier.

If we look closer into the branch operations we can see that a large amount of them actually just are conditional moves. The branch condition can be both 16- and 32-bit and be using absolute value or not. For 16-bit branch instructions this corresponds to the row MOVE_16_CONDA in table 8.4. The C code for this 16-bit branch instruction together with the move instruction looks like the following:

```
a16 = abs_s(a16);
if ( a16 > b16 )
   b16 = a16;
```

This kind of operation will be merged into one instruction, amax a16, b16. The instruction takes the absolute value of a16 and compares it with b16. The biggest of the two are then stored in b16. An extension to the amax instruction is needed for the lower bit rate of G.723.1. The absolute value of operand a16 must be optional. We will call this new instruction max, max a16, b16, and it compares a16 and b16 and stores the biggest value in b16.

For 32-bit branch instructions it gets more complicated. There are conditional moves with and without absolute value, but they are not so many. Instead, a large amount of the 32-bit branch instructions are of the form named MOVE_32_COND_I in table 8.4. In this case we do not just perform a move instruction if the branch condition is true, we also have to store the loop counter value. While around 40-45% of the branch related instructions of 6.3 kbit/s of G.723.1 are of the type conditional move, we will design a hardware structure that merge this into one instruction. The hardware is shown in figure 8.1. This will also reduce the number of branch jumps needed. The branch instructions can be cumbersome if the pipeline is deep. For the other two speech coders, this conditional move is not so pronounced. Especially not in the case of G.729.

Even though division is not used very often, around 60 times per frame for both long and short division, it will be ineffective and time consuming to implement

this instruction in software. With hardware support, the clock cycles can be at the range as the number of bits required in the quote.

Normalization, or count the leading one's or zero's of a signed number is another important instruction to incorpate in hardware. Doing it in software will be time consuming.

8.4 Assembly instruction and hardware specification for improvements

In this section we will present a couple of hardware architectures to improve the performance of these speech coding algorithms, especially the 6.3 kbit/s implementation of G.723.1.

8.4.1 32-bit conditional move with loop index

As we saw from table 8.4 the 32-bit compare together with conditional move and storage of the loop counter will occur up to 10000 times per frame in the 6.3 kbit/s implementation of G.723.1. This sequence may also include an 32-bit absolute value calculation before the comparison. The pseudo C code looks something like this:

```
for-loop with index i
basic operations
.
.
.
a32 = L_abs(a32); Optional
if ( a32 > b32 )
   b32 = a32;
   store index i;
end of if-statement
end of for-loop
```

These five instructions will be merged into one instruction. A propose of the architecture is shown in figure 8.1. To perform this operation 3-8 clock cycles would have been required, whether the branch expression is true or not and depending on the hardware support for 32-bit operations. Now, with this hardware improvement, it is reduced down to 1 clock cycle. The 32-bit full adder (FA) on

the left in the figure is used for absolute value calculations and is fed from register ACR1. The result from absolute calculation is then compared to the value in register ACR2, the reference register. The biggest of these values can then be stored in either ACR1 or ACR2 by data driven control, MSB, of the result of the compare. We do not need to perform the absolute calculation, instead we can perform the comparison directly between ACR1 and ACR2. In addition to this, a control signal must be sent to the register file if a new loop counter value has to be stored. The path delay from ACR1 via the 32-bit full adder (FA) and the accumulator to

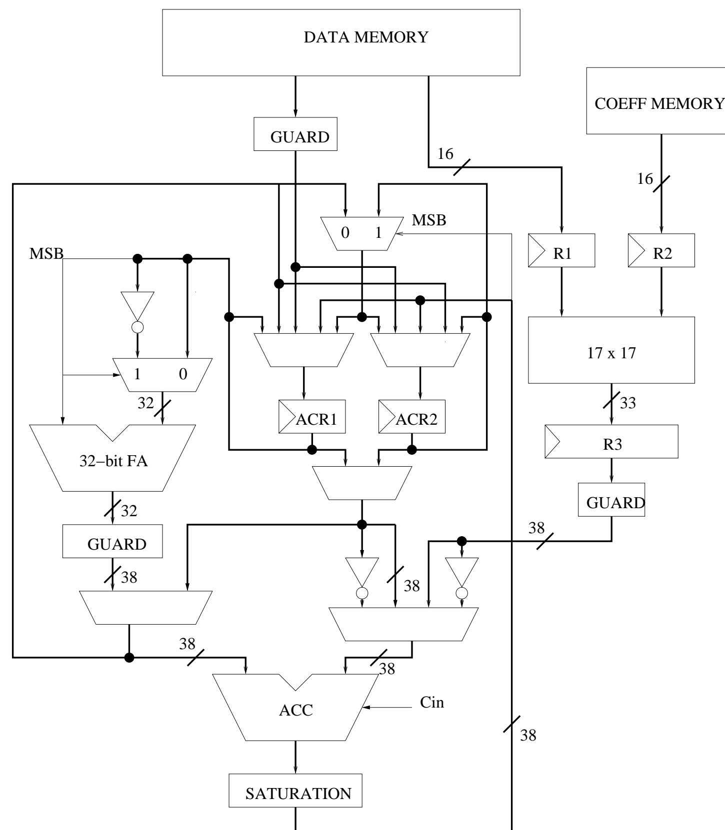

Figure 8.1: A 32-bit conditional move with absolute value and loop counter move.

the reference register (ACR2) is less the path delay through the multiplier. This

extra hardware will not add extra delay to the system.

8.4.2 Hardware improvements in G.723.1 6.3 kbit/s

By examining the C code for the G.723.1 6.3 kbit/s we found one loop where this 32-bit conditional move with loop index and extra hardware could give dramaticly improvements in the performance. The loop is the search for the pulse positions in the fixed codebook excitation. This loop also uses a complex scheme for the pointer update, when fetching data together with the 32-bit compare and absolute value presented in the previous section. The C code for the loop look like the following:

```
for ( l = 0 ; l < 60 ; l += 2 ) {

   if ( OccPos[l] != (Word16) 0 ){
      continue ;
   }

   Acc0 = WrkBlk[l] ;
   Acc0 = L_msu( Acc0, Temp.Pamp[j-1],
      ImrCorr[abs_s((Word16)(l-Temp.Ploc[j-1]))] ) ;
   WrkBlk[l] = Acc0 ;
   Acc0 = L_abs( Acc0 ) ;

   if ( Acc0 > Acc1 ) {
      Acc1 = Acc0 ;
      Temp.Ploc[j] = (Word16) l ;
   }
}
```

This loop will in the worst case be entered 288 times. The last part of this loop, from the L_abs instruction, is covered by our hardware proposal from the previous section. To keep the performance efficient, we have to make the fetch of the variable ImrCorr in one clock cycle. We can not use an ordinary pointer and just post increment after data fetch due to the absolute value. The solution is instead segmentation addressing with offset. The principle of this addressing and the offset calculation is shown in figure 8.2. The value stored in Temp.Ploc[j-1] is constant during the whole loop and will be stored in a register, REG in figure 8.2. To get an efficient calculation of the offset, we have to use a loop counter with

variable step size. In this case the step size needs to be two. A second problem is to make it bit exact with the C implementation above. This problem will rise from the fact that the loop in the C code is increasing the loop index l, while the loop counter in hardware is decreasing it's value. This will give a different result in the stored loop index if two or more calculations on Acc0 will give the same result. The solution can be to implement a loop counter that counts in the same direction as the for loop. A better solution is to keep the hardware of the loop counter and instead change the branch option from '>' to '>='.

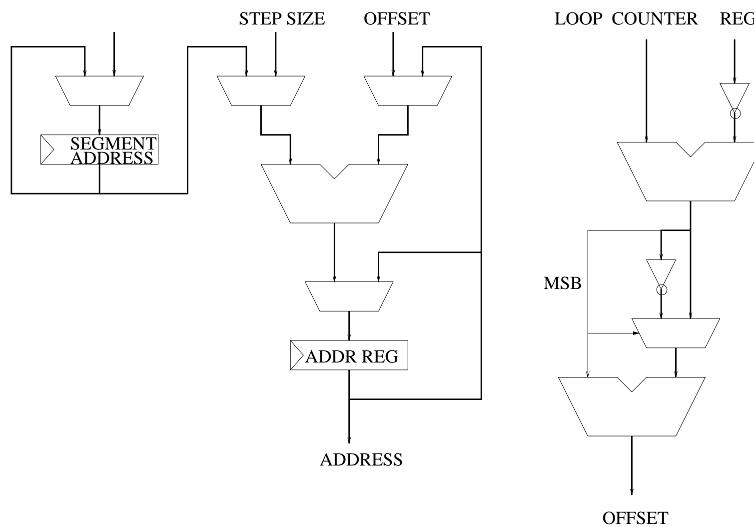

Figure 8.2: Offset calculation with loop counter and absolute value operation.

8.5 Performance estimations

In order to make an estimation on the performance we have weighted the operation by how many clock cycles they consume. This is of course very hardware dependent, but to get a rough estimate it is a good starting point. It is also important when evaluating the improvements. All the operation from table 8.4 are grouped together with operations that consumes the same amount of clock cycles. Even the operations that are left out from the table are included in this performance estimation. The table 8.5 below lists all the operations and their corresponding clock cycle consumption. The branch instructions are weighted after their complexity in the comparison statement. If you compare with zero, then the cycle count is 1. For two 16-bit number the cycle count is 2 and finally, when you compare two 32-bit numbers the cycle count is 3. All the initialization of loops are counted as one clock cycle. When moving data are 16-bit movement treated as 1 clock cycle

Cycles	**Operation**
1	add, sub abs_s, shl, shr, negate, extract_h extract_l, round, l_deposit_h, l_deposit_l norm_s and multiplications
2	l_add, l_sub, l_negate, l_shl, l_shr, norm_l
3	l_abs
18	div_s
20	div_32

Table 8.2: Number of clock cycles per operation

and 32-bit movement treated as two clock cycles. The only exception is when data is set to zero, then are both 16-bit and 32-bit treated as 1 clock cycle. Table 8.5 gives the estimated performance of the three speech coders.

8.6 Conclusion

In this paper we have proposed three hardware architectures and three assembly instruction to improve the performance. We have also seen statistics over three different speech coders in terms of basic operations, add, sub, etc, branch statements, loop and move instructions. In table 8.6 are the estimated savings presented. Most of this work applies to the higher bit rate of G.723.1, 6.3 kbit/s. The number presented, both the cycle count and the performance improvements, are estimated from worst case scenario.

Operation	**G.723.1(6.3)**	**G.723.1(5.3)**	**G.729(8.0)**
Cycle count	6000000	431000	458000
32-bit conditional with loop index	56-90000	12-30000	9-14400
Total saving (%)	9-15	3-7	2-3

Table 8.3: Improvements of the different hardware and assembly proposals in the G.723.1 and G.729. Note that the figures for G.729 is normalized to 30 ms, 3 frames, in order to be comparable with G.723.1.

8.7 Acknowledgment

This work was financially supported by the Center for Industrial Information Technology at Linköping Institute of Technology (CENIIT), Sweden.

8.8 References

[1] *ITU-T Recommendation G.723.1, Dual Rate Speech Coder for Multimedia Communications Transmitting at 5.3 and 6.3 kbit/s*, 1988.

[2] *ITU-T Recommendation G.729, Coding of Speech at 8 kbit/s Using Conjugate-Structure Algebraic-Code-Excited-Linear-Prediction (CS-ACELP)*, 1996.

	G.723 (6.3 kbit/s)	**G.723 (5.3 kbit/s)**	**G.729 (8.0 kbit/s)**
Operation	**Average/Max**	**Average/Max**	**Average/Max**
L_MAC	222373/264810	130872/141129	38734/42384
L_MAC_I	53118/69108	65343/67188	0/0
L_MAC_A	5538/5660	5908/5916	5081/5383
L_MAC_IA	684/720	713/720	0/0
L_MULT	2007/6822	5610/10339	5914/6842
L_MULT_A	119/254	123/254	9/10
MULT	329/7544	2067/7544	6564/7544
I_MULT	0/0	3280/3312	0/0
MULT_R	3308/3478	3436/3478	240/240
ABS_S	6975/8945	1189/1201	21/21
L_ADD	557/682	489/597	596/597
L_SUB	638/841	366/841	832/845
L_SHL	8550/10242	5210/5693	3068/3097
L_SHR	2625/4075	4540/6794	3150/4081
L_ABS	6937/9068	646/652	100/100
NORM_S	6/11	6/11	11/11
NORM_L	378/782	400/790	70/71
L_MLS	291/404	303/404	0/0
MPY_32_16	48/1002	9/1002	972/1002
MPY_32	8/166	1/166	166/166
DIV_S	11/24	15/24	23/24
DIV_32	47/50	49/50	10/10
TOTAL OP	343149/461334	275471/329995	111612/124163
MOVE_16_COND	28/104	2229/2256	84/104
MOVE_16_CONDA	1050/1105	1094/1105	0/0
MOVE_16_COND_I	1/23	91/92	22/23
MOVE_32_COND	4/10	4/10	7/10
MOVE_32_CONDA	337/351	586/591	80/80
MOVE_32_COND_I	8876/10869	3063/3148	508/508
TOTAL BRANCH	18534/28358	12516/20485	4585/5690
TOTAL LOOP	15333/18011	13180/13676	2618/2749
TOTAL MOVE	58633/73370	50360/56163	11592/12335

Table 8.4: Statistics of G.723.1 6.3 kbit/s, 5.3 kbit/s and G729 8.0 kbit/s. Note the different frame sizes between G.723.1 and G729, 30 ms and 10 ms respectively.

Chapter 9

Paper 2

Instruction and Hardware Acceleration for MP-MLQ in G.723.1

Mikel Olausson and Dake Liu

Department of Electrical Engineering

Linköping university,

SE-581 83 Linköping, Sweden

{mikol,dake}@isy.liu.se

Proceedings of the IEEE Workshop on Signal Processing Systems (SIPS)

San Diego, California, USA, October 16-18, 2002

Instruction and Hardware Acceleration for MP-MLQ in G.723.1

Mikael Olausson and Dake Liu
Department of Electrical Engineering
Linköping university,
SE-581 83 Linköping, Sweden

Abstract

This paper describes a significant improvement in complexity for the higher bit rate, 6.3 kbit/s, speech coding algorithm G.723.1. The solution is to reduce the number of multiplication of the most computing extensive part of the algorithm. This part stands for around 50% of the total complexity. This is done by identifying and excluding multiplication with zeros. G.723.1 is one of the proposed speech coders in the H.323 standard. The work has been done by thoroughly examining the fixed point source code from ITU, International Telecommunication Unions [1]. A hardware structure for an application specific instruction set processor (ASIP) is proposed to increase the performance.

9.1 Introduction

The market for voice over Internet protocol, also called VoIP, has increased over the years. Voice has been a natural choice of communicating for a long time and will continue to be so. The H.323 standard contains five different speech coders with different complexities and bit rates. The first one is G.711, which is mandatory and uses A/u-law compression at 64 kbit/s. Another coder is the G.728 Adaptive differential PCM (ADPCM) at 16 kbit/s. The third is G.722 and works on the bit rates of 48/56/64 kbit/s. The last two are more interesting if we are dealing with bandwidth limited transmission channels. These are G.723.1 and G.729. While the first one have two different bit rates specified, 6.3 and 5.3 kbit/s, the last have three different, 6.4/8.0/11.8 kbit/s. These two both have parts that are common, but also parts that differ a lot. From a market point of view it is of highest interest to make the implementations of these algorithms as efficient as possible. A couple of factors may influence the choice of algorithm. For example some users want to squeeze as many channels as possible on a limited transmission channel. Then their choice is as low bit rate as possible if the speech quality is good enough. Others might use them in battery powered applications and their aim is low power consumption by decreased complexity with reduced

speech quality as a trade off. The examination is not done in deep, rather from a behavior approach where we are not bound to a certain hardware manufacture. The quality or the robustness will not be treated.

9.2 General Description of G.723.1

The frame size of G.723.1 is 30 ms(240 samples). In addition to this, the algorithm uses a look ahead of 7.5 ms, this gives a total algorithmic delay of 37.5 ms. The first component of the algorithms is a high pass filter to get rid of undesired low frequency components. The short term analysis is based on 10th order linear prediction (LP). These coefficients are calculated for every subframe, 7.5 ms or 60 samples. The unquantized coefficients are then transferred to Linear Spectral Pairs (LSP) for the last subframe. These LSP are then quantized using a Predictive Split Vector Quantizer(PSVQ). The excitation parameters from both the fixed and the adaptive codebook are determined on subframe basis. The codec then uses an open loop approach to calculate the pitch delay. It is estimated every 15 ms(every second subframe) in the G.723.1. This pitch value is then refined in the closed-loop pitch analysis. The closed-loop analysis is done for every subframe. The gains and the pitch delay are referred to as adaptive codebook parameters. The fixed codebook extraction is the most exhaustive part of the whole algorithm. In this codec there exist two different approaches. One, which is used in the lower bit rate 5.3 kbit/s and is called Algebraic-Code-Excited-Linear-Prediction, (ACELP). This ACELP places at most 4 non-zero pulses within a subframe. The positions are determined from the codebook. The second approach is to use Multi-Pulse Maximum Likelihood Quantization (MP-MLQ). In this case you have more freedom to place the pulses more individually and not based on an algebraic codebook. It is in this part of the algorithm our proposal fits in. Any improvement in this part, will give great effects on the performance, because this part alone stands for around 50% of the total execution time.

9.3 Statistics of Basic Operations

All the arithmetic and logic functions like add, sub, shift, multiply and so on are implemented with a standard C library. This makes it simply to do statistics over how many times different functions are used. Additional to this, the C code has been thoroughly examined and all the branch, loop and move functions have also been identified and classified. All these statistics over basic operations, branch, loop and move instructions give a good hint on where to find improvements on instruction and architecture level. A more detailed description can be found in [2].

9.4 Former Work

In order to handle as many channels as possible within a fixed bandwidth, we want an algorithm with as low output bit rate as possible. The drawback of a low bit rate is that the complexity is increased in order to keep the quality high. In [2] there were three hardware structures proposed. They all contributed to a lower complexity. One of those proposals also hold for other speech coding algorithms like G.729 and also the lower bit rate of G.723.1. The first one merges the instructions 32-bit absolute value, 32-bit comparison and conditional data move of both a 32-bit data and the loop counter into one instruction. This instruction turned out to be really useful for the speech coding, while they all use the analysis-by synthesis approach [3]. It is rather a trial and error procedure than a deriving procedure for finding the best fitting signal out of many alternatives. In addition to this we also presented an address calculation scheme including offset addressing and absolute value calculation. These calculations were incorporated within the address generator unit instead of using the ordinary arithmetic unit. While the first one could give complexity savings from 9-15%, the last two could give savings on another 5% in the higher bit rate of the G.723.1 speech coder.

9.5 Hardware Improvements in G.723.1 6.3 kbit/s

Here we will look at another part of the MP-MLQ. This part includes a lot of multiplications, where it turns out that many of them are just multiplication by zero. Here we can see a great potential for savings. The C code for a step of the loop looks like the following:

```
for ( j = 0 ; j < 60 ; j ++ ){
   OccPos[j] = (Word16) 0 ;
}

for ( j = 0 ; j < Np(5 or 6) ; j ++ ){
   OccPos[Temp.Ploc[j]] = Temp.Pamp[j] ;
}
for ( l = 59 ; l >= 0 ; l – ) {
   Acc0 = (Word32) 0 ;

   for ( j = 0 ; j <= l ; j ++ ){
      Acc0 = L_mac( Acc0, OccPos[j], Imr[l-j] ) ;
   }
```

```
    Acc0 = L_shl( Acc0, (Word16) 2 ) ;
    OccPos[l] = extract_h( Acc0 ) ;
}
```

This nested loop will be entered 64 times in the worst case and the total number of multiplication-and-accumulations will then be:

$$64 * \sum_{l=0}^{59} \sum_{j=0}^{j=l} 1 = 64 * \frac{59 * 60}{2} = 113280 \tag{9.1}$$

Out of these 113280 MAC-operations are only 21120 actually multiplication with operand OccPos[j] not zero. That means that over 90000 multiplications are wasted. Before the loop starts all the entries to variable OccPos are cleared. We only have 5 or 6 array indices, Temp.Ploc[i] as positions, which are between 0 and 59. We also know their corresponding values, Temp.Pamp[i] as amplitudes, which are non zero. There are six elements for even subframes and five for the odd ones out of 60 entries in OccPos, which are not zero. So, instead of forcing the inner loop to multiply over all OccPos values, even the zero valued ones, we just loop over the non-zero values. The new proposed C code for this nested loop will look like the following:

```
for ( l = 59 ; l >= 0 ; l – ) {
    Acc0 = (Word32) 0 ;
    for ( j = 0 ; j < Np(5 or 6) ; j ++ ){
        if ( (l-Temp.Ploc[j]) >= 0 )
            Acc0 = L_mac( Acc0,
                Temp.Pamp[j], Imr[l-Temp.Ploc[j]] ) ;
    }
    Acc0 = L_shl(Acc0, 2);
    OccPos2[l] = extract_h(Acc0);
    }
```

This nested loop will also be entered 64 times in the worst case and the total number of multiplications has decreased to 64*60*5.5=21120. The number of multiplications has actually decreased even more, because some of the multiplication will not occur due to the branch operation. By implementing this in software

on a DSP, the performance estimations can become tricky. We have introduced a conditional branch within the loop. While it is such a short loop, the penalty from pipeline stalls might be high. Some DSP's include operations, which can do conditional operations. A good solution here would have been a conditional multiply-and-accumulate.

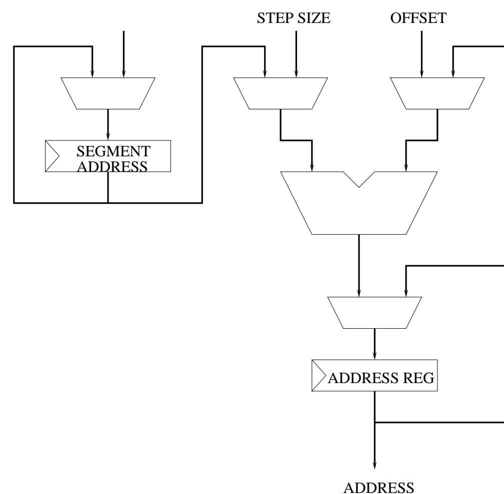

Figure 9.1: Address generator with offset addressing and segment addressing.

To do this even better we will use the hardware architecture of figure 9.1, segmentation addressing. This one was originally presented in [2]. The offset calculation is rather simple, while it is a subtraction between the loop counter and the stored value of variable Temp.Ploc[j] in a register. The offset value is then added to the start address of the variable Imr. This modification is not enough, the calculated address offset, l-Temp.Ploc[i], can point outside the Imr buffer and we will perform an illegal multiplication. This can only occur when the result from the subtraction is negative. The solution is to use the msb bit from the subtraction between the loop counter and the Temp.Ploc[j] in figure 9.2. This data dependent control signal will then be propagated to the multiplier. If this bit turns out to be a one, i.e. a negative value of the subtraction result, the multiplier will not take

the fetched operand as input, rather it will take a zero. We call this operation conditional operand. This will introduce an extra multiplexer in the multiplier circuit. The reason for the extra absolute value calculation in the offset calculation circuit is found in [2]. We can address calculations, which require absolute value calculations.

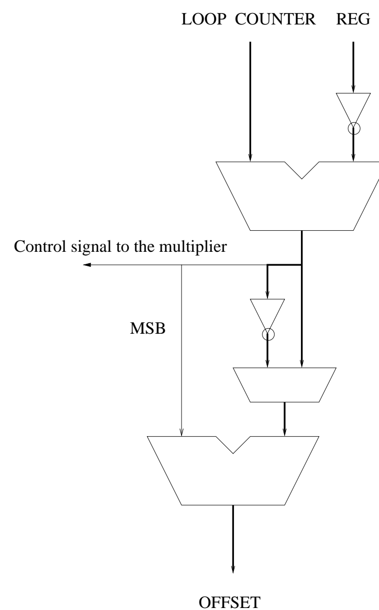

Figure 9.2: Offset calculation with loop counter and absolute value operation.

9.6 Performance Estimations

In order to make an estimation on the performance we have weighted the operation by how many clock cycles they consume. This is of course very hardware dependent, but to get a rough estimate it is a good starting point. It is also important when evaluating the improvements. The table 9.1 below lists all the operations and their corresponding clock cycle consumption.

The branch instructions are weighted after the complexity in the comparison statement. If you compare with zero, then the cycle count is 1. For two 16-bit numbers the cycle count is 2 and finally, when you compare two 32-bit numbers the cycle count is 3. All the initialization of loops are counted as two clock cycles.

Cycles	**Operation**
Arithmetic operations	
1	16-bit operations
2	32-bit operations
3	32-bit absolute value calculations
18	16-bit by 16-bit division
20	32-bit by 16-bit division
Branches	
1	comparison with zero
2	comparison between two 16-bit values
3	comparison between two 32-bit values
Loops	
2	Loop start
Moves	
1	16-bit move
2	32-bit move

Table 9.1: Number of clock cycles per operation.

When moving data are 16-bit movement treated as 1 clock cycle and 32-bit movement treated as two clock cycles. The only exception is when data is set to zero, then both 16-bit and 32-bit are treated as 1 clock cycle. Table 9.2 gives the estimated performance of the four different implementations. To make the estimate even more accurate, we have also introduced memory related issues. We have taken into account that fetching operands from memories take one clock cycle, at least for the first operands of the loop. When you need the next operand of a buffer it is assumed to be in a register already. This fetch has been done during the operation of the operands. As we can see from table 9.2 there is no big difference between the branch implementation and the conditional multiply. This is not true because pipeline issues are hard to calculate. For a 2-stage pipeline, this is true, but for deeper pipelines you have to insert nop operations after the branch instruction. The total complexity of the whole algorithm is around 20 MIPS, without any modifications. This value is calculated by counting the basic operations and multiplying them by their weights from table 9.1.

Operation	Average MIPS	Maximum MIPS	Improve-ment
Original	8.40	11.08	
With branch	7.74	10.64	4-8 %
With conditional multiply	7.41	10.12	8-11 %
With extra HW	5.73	7.80	29-31 %

Table 9.2: Performance estimation for the four different implementations of the fixed codebook excitation for the higher bit rate of the speech coder G.723.1. The improvement in percent compared to the original implementation is stated in the fourth column.

9.7 Conclusion

In this paper we have seen four different implementations of a critical part of the fixed codebook search. By examining the code carefully, we have reduced the number of required multiplication by more than five times. This gives a performance improvement of about 30 % compared to the original implementation. While this number just corresponds to the fixed codebook excitation part and this part stands for approximately 50 % of total execution time, the overall improvement is around 15 %. The modified hardware architecture to improve the performance is also presented. We have calculated the performance improvements by applying weights to all basic operations, add, sub, etc, branch statements, loop and move instructions.

9.8 Acknowledgements

This work was financially supported by the Swedish Foundation for strategic Research (SFF) and the Technical Research and Research Education (TFF).

9.9 References

[1] *ITU-T Recommendation G.723.1, Dual Rate Speech Coder for Multimedia Communications Transmitting at 5.3 and 6.3 kbit/s*, 1988.

[2] M. Olausson and D. Liu. Instruction and Hardware Accelerations in G.723.1(6.3/5.3) and G.729. In *The 1st IEEE International Symposium on Signal Processing and Information Technology*, pages 34–39, 2001.

[3] A.M. Kondoz. *Digital Speech*. John Wiley and Sons Ltd, 1994.

Chapter 10

Paper 3

Reduced Floating Point for MPEG1/2 Layer III Decoding

Mikael Olausson, Andreas Ehliar, Johan Eilert and Dake Liu

Department of Electrical Engineering

Linköping university,

SE-581 83 Linköping, Sweden

{mikol,ehliar,je,dake}@isy.liu.se

Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)

Montreal, Quebec, Canada, May 17-21, 2004

Reduced Floating Point for MPEG1/2 Layer III Decoding

Mikael Olausson, Andreas Ehliar, Johan Eilert and Dake Liu
Department of Electrical Engineering
Linköping university,
SE-581 83 Linköping, Sweden

Abstract

A new approach to decode MPEG1/2-Layer III, mp3, is presented. Instead of converting the algorithm to fixed point we propose a 16-bit floating point implementation. These 16 bits include 1 sign bit and 15 bits of both mantissa and exponent. The dynamic range is increased by using this 16-bit floating point as compared to both 24 and 32-bit fixed point. The 16-bit floating point is also suitable for fast prototyping. Usually new algorithms are developed in 64-bit floating point. Instead of using scaling and double precision as in fixed point implementation we can use this 16-bit floating point easily. In addition this format works well even for memory compiling. The intention of this approach is a fast, simple, low power, and low silicon area implementation for consumer products like cellular phones and PDAs. Both listening tests and tests versus the psychoacoustic model has been completed.

10.1 Introduction

Entertainment in small handheld devices like cellular phones and PDAs are getting more and more popular. One of these extra features is audio playback. MPEG-1/2 layer III, often known as MP3, is an audio coding standard that provides high audio quality at low bit rates [1]. Since a lot of these consumer product are portable, it is important to use low power implementations. The idea is to use small arithmetic units and still achieve high computional dynamic range. The standard for MPEG includes both encoder and decoder, but for the applications discussed here, the only interesting part is the decoder. Usually the decoder is implemented on a 24 or 32-bit fixed point processor. The bit size is chosen to give reasonable quality in the decoded music. When using a standard 16-bit processor, i.e., a DSP, double precision must be used in parts of the computations. Otherwise, a quality degradation can be heard. Here we will present a new approach to a fast, simple, low power, and low silicon area implementation using 16-bit floating point. The target is portable devices without extreme audio quality requirements; a cellular phone

or a PDA. The headphones or the loud speakers are usually of quite poor quality. Therefore there is no need for high demands on the output music.

10.2 General description of the MP3 format

The ISO/IEC 11172-3 and ISO/IEC 13818-3 are coding standards that provide high quality audio at low bit rates. There are three layers associated with the standard, layer I, II and III. They offer both increasing compression ratios, and increasing computing complexity. Layer III is more known as "mp3" based on the file extension it uses. The encoded bitrates ranges from 32 kbit/s up to 320 kbit/s. There are three main sampling frequencies associated with the standard 32, 44.1 and 48 kHz. There are also half frequencies which are just the main frequencies divided by 2. For a more complete description of the standard, see [1].

10.3 The work

The work began with the reference code in C for the MPEG 2 layer III decoder. First the arithmetic instructions were exchanged against functions calls. This made it easier to perform profiling of the code and to elaborate with the precision and the dynamic range. The first approach was to use one format for all calculations within the decoder. While memory sizes usually are limited to byte lengths, we tried to use a floating point format while only using 16 bits. One bit is allocated for the sign bit and the remainder is split between the mantissa and the exponent. This approach turned out to be insufficient for the compliance testing [2]. To be called a fully compliant audio decoder, the rms level of the difference signal between the reference decoder and the decoder under test should be less than $2^{-15}/\sqrt{12}$ for a sine sweep signal 20Hz - 10 kHz with an amplitude of -20 dB relative to full scale. In addition to this, the difference shall have a maximum absolute value of no more than 2^{-14} relative to full scale. To be referred to as a limited accuracy decoder, the rms level of the difference signal between the reference decoder and the decoder under test should be less than $2^{-11}/\sqrt{12}$ for a sine sweep signal 20Hz - 10 kHz with an amplitude of -20 dB relative to full scale. There are no requirements on the maximum absolute difference. We were unable to hear any degradation in quality when we listened to the decoded files. The listening tests are described more in detail in section 10.5. We then decided to increase the internal precision, keeping the external precision to 16 bits. The distinction between internal and external precision lies in where the data is stored. While data is in the datapath the format of the data can be higher, but as soon as it is stored in memory it must be converted to 16 bits. Figure 10.1 shows the result

of the compliance test for different sizes of the internal and external mantissa. To be fully compliant we needed a mantissa of 19 bits internally and 15 bits externally, alternatively 18 bits internally and 16 externally. The IEEE single precision floating point format consists of a 23 bits mantissa with an imlpicit one at the beginning. Our goal of this approach was not to aim for full compliance, instead the intention was a descent quality for low end equipment. Therefor, the limited accuracy test was acceptable. According to figure 10.1 the demands of the mantissa is only 13 bits internally and 9 bits externally, alternatively 12 bits internally and 10 bits externally for limited accuracy. This is a reduction of 6 bits from the fully compliant requirements and half of the mantissa size compared to the IEEE floating point format. The compliance test has been performed on fixed point arithmetic in [3]. The requirements from their test is a 20 bits implementation.

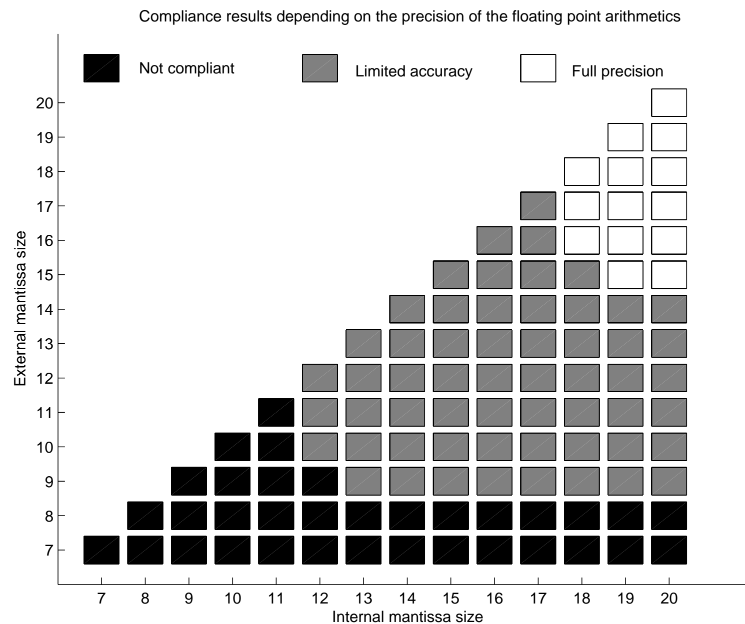

Figure 10.1: Compliance test for different sizes of the internal and external mantissa.

While the mantissa is responsible for the precision, the exponent determines the dynamic range. One of the reasons for using this floating point approach was to avoid the problem of scaling variables that had to be completed in integer representation. Since we did not want variables to overflow, the upper limits of the exponent was set by the dynamic range of the variables. This was done by profiling the code and storing the maximum values of the variables. As a result, we could distinguish a difference in the upper limit for variables in the data path and the ones stored in memory. We needed a higher range for internal variable in

the data path.

10.4 Motivation

By using this reduced floating point representation we can achieve a high dynamic range and a reasonable precision, with fewer bits. We use 6 bits for the exponent in the internal representation. In order to get the same dynamic range using fixed point, it is necessary to use 64 bits. If this amount of bits is not available, double precision would be used instead. This results in a performance degrading. Another approach is to use appropriate scaling in order to reach the required dynamic range. Unfortunately, this process is tricky and time consuming. In addition, the code size would become bigger, the firmware would be more complex, and the debugging harder. However, floating point is a more straight forward implementation. In this floating point approach we can attack the problem of dynamic range and precision independently. If our goal is a high dynamic range we would allocate more bits for the exponent and if there is high demands on the precision we allocate more bits for the mantissa. In the fixed point approach we cannot separate these issues. By increasing the dynamic range the precision would also increase and vice versa. In the case of fixed point representation, an extra bit will increase the dynamic range by $2^{n+1}/2^n = 2$. In the floating point alternative, an extra bit in the exponent will give a $2^{2n}/2^n = 2^n$ increase in the dynamic range. From the calculations above it is clear that the floating point is superior when one wants to have an high dynamic range. For the precision aspect the floating point representation is also favorable. Due to the normalization of all the values, the same number of precision bits will always occur. In a fixed point representation we have to trade precision for dynamic range. By shifting the binary point to the right we can represent larger numbers on the expense of lower precision. The other extreme is when there are too many bits allocated for the precision resulting in an overflow. The IEEE standard[4] specifies a number of floating point formats. The first two are single precision and double precision formats that use a total of 32 and 64 bits respectively for their implementation. The reference decoder for the mp3 format uses the double precision format in their calculations. While our target for this paper is low-end applications, we will use the same concept as for the floating point standard but reduce both the mantissa and the exponent to fit our purposes. To find a balance between the fixed point and the floating point, the block floating point representation can be used. Again, the mantissa must be separated from the exponent, but the exponent stays the same for a whole block of values. As in the fixed point case, we will need to search through the whole block of data to find the right exponent and then scale the values appropriately. As a result both high dynamic range and high precision are possible. The drawback is

when the values within a block differ significantly. The exponent is then chosen from the largest value and the small values will be shifted down and in turn lose most of their precision. The three representations are shown in figure 10.2.

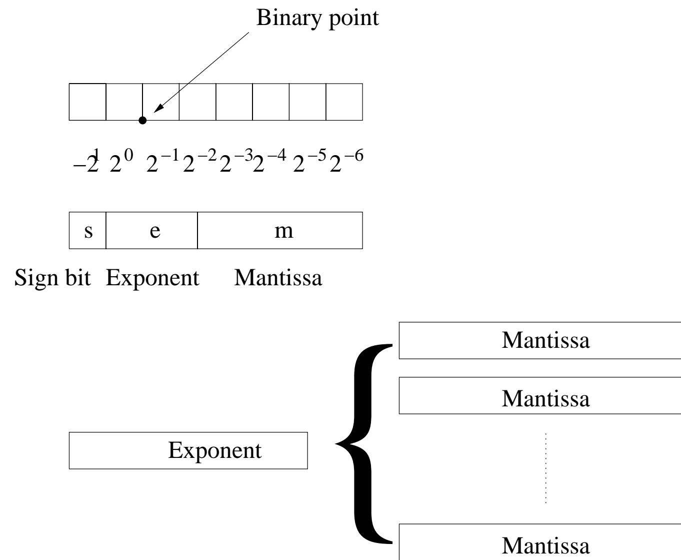

Figure 10.2: Fixed point, floating point and block floating point representation. In fixed point, the binary point is imaginary.

10.5 Testing the quality

In addition to the compliance test described in chapter10.3, we conducted objective measurement tests. The idea was to use the psychoacoustic model from an mp3 encoder to see if the error introduced by thereduced precision was inaudiable. for this purpose, the LAME version 3.92 was used. First, the masking levels from each subband were extracted. The decoder under test was then compared to the reference decoder using double floating point precision. From this, two different measurement could be made; one where we calculated the Signal-to-“hearable” noise ratio (SHNR) and one where we measured the masking flag rate. The SHNR is calculated in the same way as the more traditional Signal-to-Noise ratio (SNR), except that the noise is exchanged with the “hearable noise”. All the noise that is introduced by the decoder under test is compared to the reference decoder that exceeds the masking levels. The masking levels are calculated from the reference decoder output signal.

$$SHNR[dB] = 10 \times log_{10} \frac{\sum P_{dut}[i]}{\sum P_{hr}} \quad (10.1)$$

where $P_{dut}[i]$ is the output samples from the decoder under test squared and

$$P_{hr}[i] = \begin{cases} diff[i] - Mask[i] & diff[i] > Mask[i] \\ 0 & diff[i] <= Mask[i] \end{cases}$$

Mask[i] is the masking levels from the psychoacoustic model. diff[i] is the difference between the output signal from the reference decoder and the decoder under test squared; diff[i] = $(pcm_{ref} - pcm_{dut})^2$ To make it more convenient are all the calculations made on a midchannel. $\frac{(l+r)}{\sqrt{2}}$, where l is the left channel and r is the right channel. The second measurement is from [5]. The flag rate gives the percentage of frames with audible distortion, i.e., frames where the introduced noise is above the masking level. In table 10.1 we used the sine wave file from the compliance test and changed the number of bits for the mantissa and the exponent. The first test was the single precision floating point. The result was very good performance, but we used 32 bits for the representation. In order to be called a full precision decoder we had to use a 19 bit mantissa internally and a 15 bit mantissa externally. See figure 10.1. The degradation from the single floating point is only 7 dB. For limited accuracy the degradation is much bigger, especially in the case hearable noise. Here, a 13 bits internal mantissa and 9 bits externally were used. Further, a test was conducted where the internal and external representation were the same. Namely, 10 bits mantissa, 5 bits exponent and 1 sign bit. This is interesting since it fits into a 16 bit memory and a 16 bit datapath. As a conclusion of this test the difference between the SNR and SHNR decreases as you decrease the precision, i.e., the noise introduced becomes more and more “hearable”.

Precision	**SNR**	**SHNR**	**Flag Rate**
Single float	89.1	115.8	$2.5 * 10^{-7}$
Full precision	82.9	108.6	$6.0 * 10^{-4}$
Limited Accuracy	53.9	60.9	0.16
16-bits	46.2	47.3	0.13

Table 10.1: The result of the objective measurement.

The second test was the sound files from the SQAM disc [6], Sound Quality Assessment Material, a mixture of both male and female speech in English, French and German. It also includes electronic tune in file frer07_1. The remaining sounds are purely instrumental. These sounds are known to be revealing for MPEG coding. As we can see in table 10.2 the SNR remains rather constant. It is approximately 54 dB and similar to the compliance test in table 10.1. It seems as if the noise introduced is constant but the amount of “hearable” noise differ. This

Test file	**SNR**	**SHNR**	**Flag Rate**
bass47_1	54.9 dB	69.0 dB	0.011
frer07_1	54.6	64.3	0.084
gspi35_1	54.4	69.4	0.012
gspi35_2	54.2	72.9	0.0086
harp40_1	54.1	107.0	$3.7 * 10^{-5}$
horn23_2	54.1	64.5	0.18
quar48_1	53.6	72.2	0.047
sopr44_1	54.8	65.0	0.020
spfe49_1	55.0	70.5	0.0050
spff51_1	55.2	70.4	0.0048
spfg53_1	54.3	74.1	0.0024
spme50_1	55.3	71.2	0.0067
spmf52_1	55.2	74.6	0.0034
spmg54_1	54.8	75.4	0.0024
trpt21_2	53.9	73.3	0.0076
vioo10_2	54.8	73.5	0.0013

Table 10.2: The result of the objective measurement.

is one reason why the SNR measure does not suit well for the purpose of audio quality measurement. All the noise introduced will not effect the listening quality.

In figure 10.3 and figure 10.4 the extremes from table 10.2 harp40_1 and horn23_2 can be seen. Harp40_1 has the highest SHNR and the lowest flag ratio, which indicates the best sound quality. The opposite is the file horn23_2, which has one of the lowest SHNR and the highest flag ratio. In figure 10.3 we can see the difference between the reference decoded file and the file decoded with our proposal; 1 sign bit, 9 bits mantissa and 5 exponent bits externally and 13 bits mantissa and 6 bits exponent internally. The difference between the two test files is not substantial. The SNR is very similar, around 54 dB. Finally, the error signal after the masking level is taken into account in figure 10.4. Here we can see that there is a significant difference. The noise introduced from our reduced floating point format is not masked away as well in the test file horn23_2 as in the test file harp40_1. This is not clear from the SNR, but with the measurements SHNR and flag ratio it becomes much clearer.

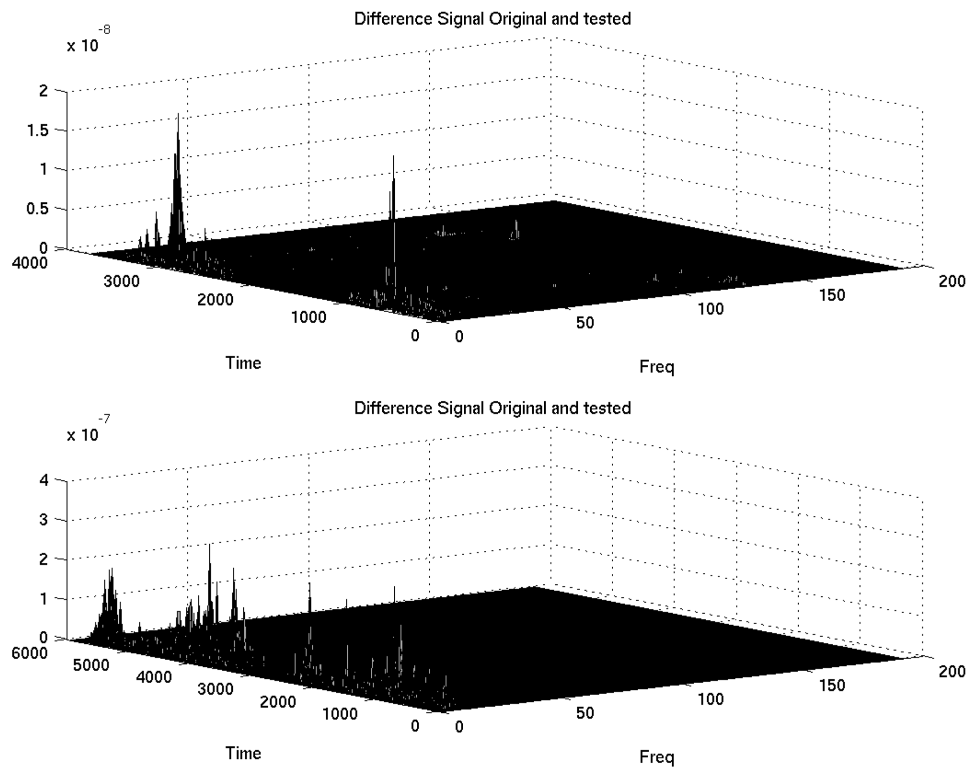

Figure 10.3: The difference between the original and the decoder under test. Above for the test file harp40_1 and below for horn23_2. The frequency axle is the output from the fft and the sampling frequency is 44.1 kHz.

10.6 Savings in power and area

The high dynamic range from this reduced floating point has more advantages as compared to an ordinary fixed point implementation. First, a mantissa of just 13 bits reduces the multiplier. In the comparing fixed point implementation a multiplier of 20-24 bits is needed. Here it is alright with just 13 bits. By using a reduced floating point format, one can also reduce the size of the accumulator register. There is no need for double precision and guard bits. In addition, the smaller size of the variables gives smaller adder units. In fixed point arithmetic you have to keep track of the scaling of the variables, otherwise you will run into precision or overflow problems. Since the scaling takes place within the arithmetic unit, there is no need for an barrel shifter, just a small one for the six bit exponent. The absence of need for dynamic scaling results in a decrease of the amount of code and the programming task becomes easier. The reduced format of the external variables, i.e. the variables that are stored in memory, reduces the

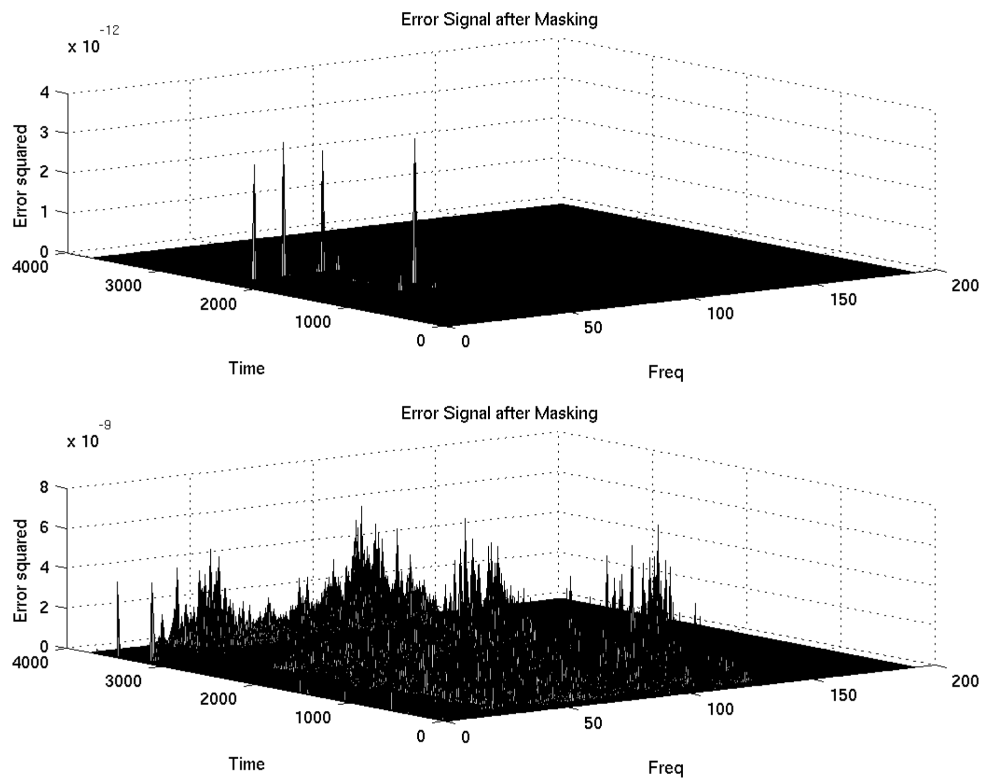

Figure 10.4: The remaining error after the the noise below masking level is taken away. Above for the test file harp40_1 and below for horn23_1.The frequency axle is the output from the fft and the sampling frequency is 44.1 kHz.

size of the data memory. If you customize your own memory, it is enough with 15 bits for external storage. The internal variables within registers are still larger; 20 bits.

On the negative side is the more complex arithmetic unit, where variables need to be shifted before addition and subtraction. There is also a need for post scaling to make sure that the bits in the mantissa are aligned correctly. This hardware is expensive and power consuming. Finally, it might give a deeper pipeline.

10.7 Future work

To prove this concept more accurately it has to be implemented in silicon and conduct additional measurements. The real numbers on silicon sizes, memory issues and implementation difficulties could then be obtained. It might be possible to exchange algorithm from the reference decoder to a more simple without any

Operation	**20-bit reduced floating point**	**20-bit fixed point**
Multiplier	13x13	20x20
Accumulator	20 bits	48 bits
Register	20 bits	20 bits
Memory	15 bits	20 bits

Table 10.3: Comparison between 20-bit reduced floating point and 20-bit fixed point

quality degradation. This new algorithm might suit better for this implementation.

An subjective listening test with proffesional listeners is also preferable. We have made less complicated listening tests ourselves, but we do not know what kind of artifacts to listen for. The aim for this implementation is not high performing audio devices, rather low end products. Consequentially, we might be able to shrink the mantissa and exponent sizes even further.

Another interesting aspect would be to have a reconfigurable architecture. The number of mantissa bits and exponent bits would then be programmable on the fly. In that case you can trade power consumption for audio quality.

10.8 Conclusion

We have proposed a floating point approach to implement a mp3 decoder. Instead of the usual fixed point we have used a floating point implementation with different number of bits for the internal and the external representation. By this approach we can reduce the size of the arithmetic units and still keep good quality sound. The firmware also becomes simpler. There will be no need of scaling of variables, this is done automatically within the arithmetic unit. We have also performed simpler listening tests and done some objective sound quality measurements.

10.9 Acknowledgment

This work was financially supported by the stringent of SSF and the Center for Industrial Information Technology at Linköping Institute of Technology (CENIIT).

10.10 References

[1] *ISO/IEC-13818-3, Information technology - Generic coding of moving pictures and associated audio - Part 3: Audio*, 1998.

[2] *ISO/IEC-11172-4, Information technology - Generic coding of moving pictures and associated audio - Part 4: Compliance testing*, 1998.

[3] In-Cheol Park Yongseok Yi. A Fixed-Point MPEG Audio Processor Operating at Low Frequency. *IEEE Transactions on Circuits and Systems–II: Analog and Digital Signal Processing*, 47:779–786, November 2001.

[4] *IEEE Standard 754 for Binary Floating-Point Arithmetic*, 1985.

[5] *ITU-R 1387-1, Method for objective measurements of perceived audio quality*, 1998.

[6] SQAM Sound Quality Assessment Material. *http://www.tnt.uni-hannover.de/project/mpeg/audio/sqam/*.

Chapter 11

Paper 4

Bit memory instructions for a general CPU

Mikel Olausson, Anders Edman and Dake Liu

Department of Electrical Engineering

Linköping university,

SE-581 83 Linköping, Sweden

{mikol,anded,dake}@isy.liu.se

To be presented at the The 4th IEEE International Workshop on System-on-Chip for Real-Time Applications(IWSOC)
Banff, Alberta, Canada, July 19-21 2004

Bit memory instructions for a general CPU

Mikael Olausson, Anders Edman and Dake Liu
Department of Electrical Engineering
Linköping university,
SE-581 83 Linköping, Sweden

Abstract

Embedded memories in an Application Specific Integrated Circuit(ASIC) consume most of the chip area. Data variables of different widths require more memory than needed because they are rounded up to nearest power of 2, i.e., 6 to 8 bits, 11 to 16 bits, and 25 to 32 bits. This can be avoided by adding two bit oriented load and store instructions. The memories can still be 8, 16 or 32 bits wide, but the loads and stores can have arbitrary variable sizes. The hardware changes within the processor are small and an extra hardware block between the memory and the memory is added.

11.1 Introduction

Typically, we define the variables as either byte, word, or double word. The reason is that these data sizes are supported by load and store instructions within processors. In addition, the high level languages, for example C, do not have support for any other data sizes. The memories on the other hand, are more prepared for a variety of formats. Usually, one can define store sizes down to a bit level. This can be done by using the internal write mask within the memories when the memories are generated. In Artisan memories the resolution of write mask can be programmed down to one bit [1]. Because the hardware for the write mask is already there, it will not add much extra area or delay by using it. This is only suitable for embedded memories closest to the processor. The processor and memory have to be integrated in the same ASIC. For external memories the extra bus for the write mask will be too demanding.

To implement an algorithm as efficient as possible, with regards to memory usage, speed, and power consumption, an ASIC is preferable. With an ASIC, the data path can be adjusted to the required data lengths and memories can have arbitrary and different widths. The instructions and the hardware to implement them can be customized for your special needs. The drawbacks to the use of an ASIC are the long design times and manufacturing times. In addition, the

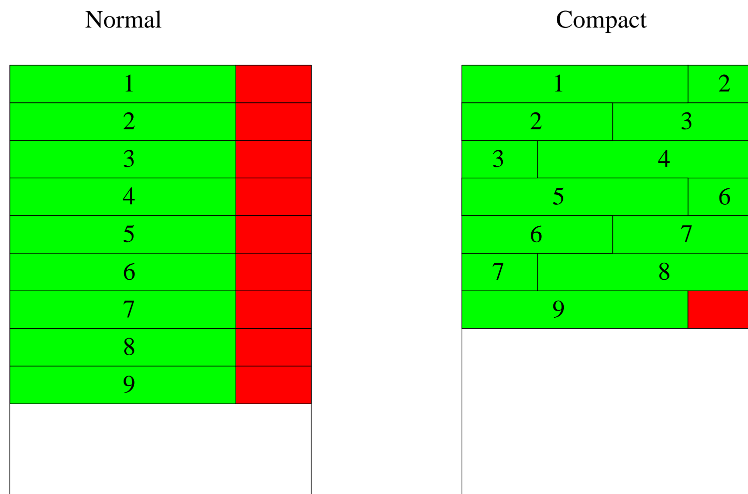

Figure 11.1: The data memory with and without compact stored data. The dark boxes mark wasted memory.

inflexibility once it's is manufactured, make the ASIC extremely risky and costly. A completely different way to implement the algorithm is to use a programmable device, such as a microcontroller or a DSP. Late changes are possible by updating the software. This solution comes to the expense of fixed hardware resources, a fixed data path width, and fixed memory widths. A higher power consumption, more memory, and lower performance are also expected. The fixed memory width can cost a lot of wasted memory, see figure 11.1. For example, the output from an Analog-Digital-Converter(ADC) is not always a power of 2. Instead, the output is padded up to the nearest power of 2. In the case of 13 bits from the ADC, the storage requirement is 16 bits. In turn, every sample one store 3 bits out of 16 is a waste of memory. This is an awkward situation, since the memories usually are the bottlenecks in embedded systems; They consume most of the chip area. The other alternative is to store the variables in a compact way, but unfortunately this will consume a lot of extra instructions and make the code writing cumbersome. These new instructions will increase the size of the program memory. By adding two new instructions, one store and one load, together with extra hardware between the processor and the memory, we can reduce this problem. The basic idea behind the instruction is to define within the instruction itself, the size of the variable to either load or store. As soon as the variable reaches the processor, it will be treated as a variable of native length. The idea is to make changes as small as possible within the existing processor. This will be called bit allocated memories. The memories are still byte or word aligned, but the loading and storing can be bit oriented.

11.2 Related Work

Bit addressing on register level is presented in [2]. This bit addressing is a network processor built on a Reduced Instruction Set Computer(RISC) core. Operations on bits are possible through bit addressing in registers. First, the data word containing the required bits is addressed. In addition to this information, the starting point of the bits and the length of the bits are given. For a logical or an arithmetical operation, three registers are specified; Two for the sources and one for the destination. To every register, the starting bit position is also added. Finally, one bit length is supplied to all registers.

11.3 General description of the OR1200 CPU

The processor chosen for this project is a general 32-bit processor from the OpenRISC project called OR1200 [3]. The OR1200 is an open source project which includes a processor, a C-compiler, a simulator, and also Linux support. The main reason for choosing this processor was the availability of tools and the availability of the source code for the processor. Furthermore, it contains a 32x32 general register file, a multiplier, and a 5-stage pipeline. Since it is a general processor, it lacks most of the instructions found in a DSP. For example, there is no address generation unit(AGU). Instead all the addresses must be incremented/decremented by arithmetic operations on a general register in the register file. There is no support for hardware loops, i.e., you have to dedicate a register for the loop counter and do the update and comparison yourself. In addition, there are separate data and program memories, but the data memory only support one data fetch/store per clock cycle, a Harvard architecture. In order to make the processor even less complicated, we have also taken away both the instruction and data Memory Management Unit, MMU, the interrupt module is discarded, there is no debug unit and the power management unit is no longer available. It is only the processor core left. Within the processor it is possible to add custom instructions. This was a mandatory requirement, because we did not want to change the original behavior of the processor. It was also structured and well written in Verilog.

11.4 The work

The basic idea behind this work was to keep the majority of the processor without changes. All the instructions that were available prior to the modifications are still available. Instead, we added two custom instructions; one for load and one for store. In addition, there are three extra signals out from the processor core.

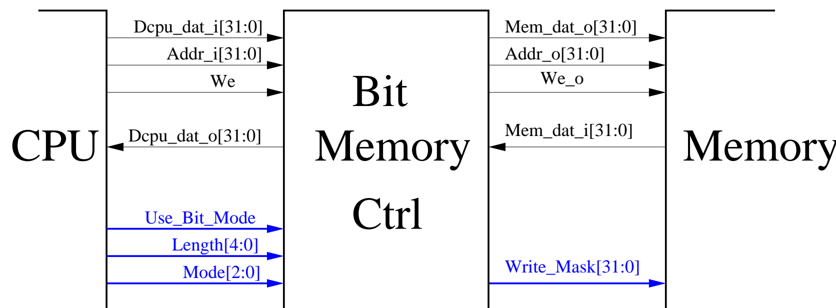

Figure 11.2: The memory controller with the three new signals, Use_Bit_Mode, Length[4:0], and Mode[2:0], coming from the processor.

All of the arithmetic operations that were available before are still available. No new operations are added. I.e., even if the loaded variables are not word oriented, we still perform arithmetic and logic operations on them as if they were word oriented. In this processor, the native length is 32 bits. The new load and store instructions permit variables of various lengths between 1 and 32 bits. These different lengths leave us with a couple alternatives when issuing a load instruction. In the first alternative, the loaded variable is a signed number and the sign extension is then performed before writing it to a register. For an unsigned number zero extension is required. We can also choose to interpret the loaded variable as a fractional number and then store it in the most significant bits in the register. For a store instruction we simply have to know whether the variable is stored in the most significant bits or the least significant bits of the register. All of these issues are taken into account within the new load and store instructions.

11.4.1 Modifications in OR1200

The first intention was to leave the processor intact, without any changes at all, unfortunately that turned out to be impossible and the concept changed to using as few modifications as possible. First of all we had to add the custom load and store instructions to the instruction set. We wanted the syntax of these new instructions to be as similar as possible to the ordinary load and stores. The main difference is that on top of specifying registers and an offset address, we needed to include the length of the variable and the mode. The mode is whether a variable is interpreted as signed or unsigned, and as integer or fractional variable. Figure 11.4 and 11.5 show that the bits for length indication and mode is taken from immediate offset address value. We have reduced it from 16 bits to 8 bits. This implies that the

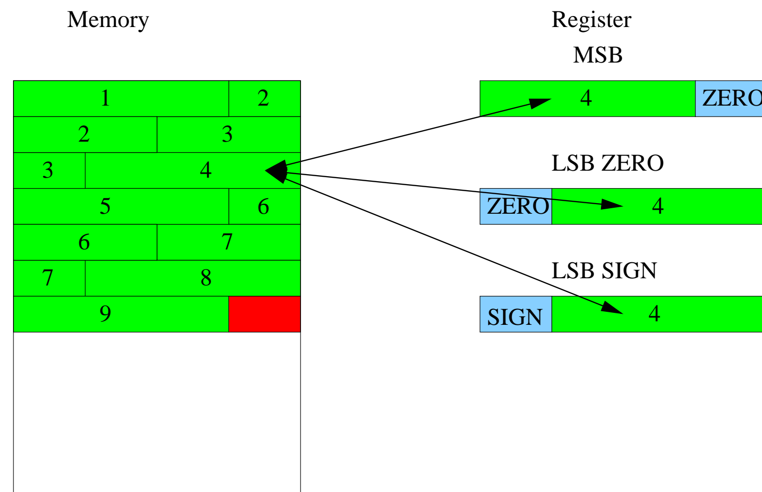

Figure 11.3: Three different modes for loading and two for storing variable of sizes not equal to the native word size.

possible offset addressing range is -128 to 127 bits. For the ordinary store and load instructions the address range is from -32768 to 32767 bytes, the same as before.
From the processor three new signals are added, see the lower part of figure 11.2. Length[4:0] is the length of the variable, 1 to 32 bits. By setting the signal Length[4:0] equal to 0 load or store equals 32 bits. The difference between loading a variable of the size 32 and a word with the ordinary load instruction is that the 32 bits does not need to be word aligned in memory in the former alternative. In the mode signal, Mode[2:0], one bit is allocated for unsigned/signed interpretation, one bit for fractional/integer representation, and one bit is reserved for future applications. These two signals will be generated continuously. For the hardware between the memory and the processor to know whether to load and store variables as words or specified with the length signal, the Use_Bit_Mode signal is generated. If this signal is high, the Length and the Mode signal are used for storing and loading variables to and from the processor.

11.4.2 The memory control module

A majority of the work was to design and build the extra memory hardware between the processor and the memory, see figure 11.2. From here on this hardware will be called the bit memory controller. It needs to be compact for the silicon cost and fast enough not to introduce a new critical path. Further, it should be transparent for the existing load and store instructions. Our assumption is that the

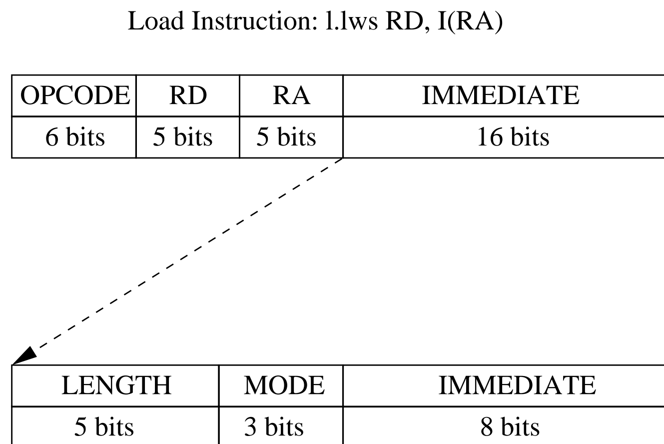

Figure 11.4: The instruction encoding for the custom load instruction(bottom) compared to the ordinary load instruction(top).

memory itself is 32 bits wide. For a load operation of a variable of arbitrary size, we first need to read the whole 32 bit word from the memory. The word is then shifted in order to get the wanted bits in the right position, either as a fractional word or an integer word. In the fractional mode, the bits are shifted to the most significant part of the register and for the integer to the least significant part. The number of shift steps and the direction of the shift are calculated from both the bit address and the length of the variable. After the shifts, the variable is in the right position and we need to mask the unwanted bits away. At the same time we need to sign extend signed variables. Then, the variable is ready for the processor as a 32-bit word. To make this even more complicated, sometimes one wants to read a variable that spans over two memory locations. For example, to load variable noted as 2 in figure 11.3. As a result, we need to stall the processor one extra clock cycle to wait for the second read from the memory. The new data needs to be shifted as well and then be merged together with the first read data. The masking and eventual sign extension is then the same as the masking for just one load. The approach of stalling the processor when reading over memory boundaries can be avoided by using two memories; one for even and one for odd addresses. Then one can load both words in one clock cycle. The drawback of this method is that the estimated area penalty by splitting one memory into two smaller is 20%. The second drawback is the need of a 64-bit shifter compared to a 32-bit in the former approach.Alternatively,a 32-bit rotational shifter can be used.

The store operation is less complicated than the load operation. Here, we do

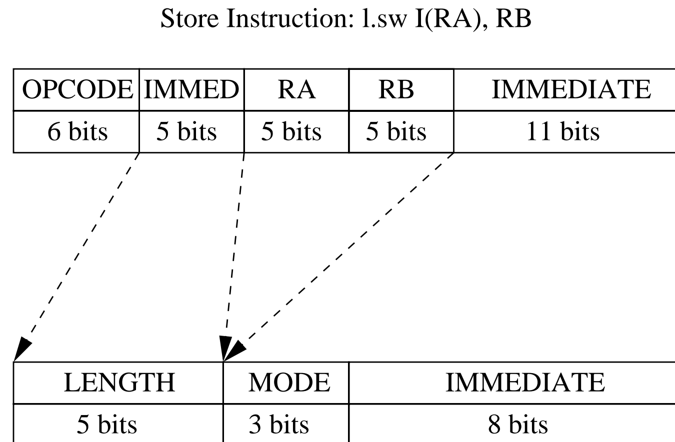

Figure 11.5: The instruction encoding for the custom store instruction(bottom) compared to the ordinary store instruction(top).

not need to mask the data before it is sent to the memory. Instead, the masking takes place within the memory itself. We do need to supply a write mask vector to the memory. This mask is calculated from both the bit address and the length of the variable to be stored. As in the load operation, the first operation of the data coming from the processor is to shift it into the right position. Then, the data is ready to be sent to the memory. A store can span over two memory locations, just like a load. Two consecutive stores have to be done. A new write mask for the second store has to be calculated. This is solved by generating a 64-bit write mask from the beginning. For the first store the lower 32 bits is selected and for the second the upper 32 bits is selected from the 64-bit write mask. The processor will be stalled, while the second store is performed. If the memory is divided up into two smaller memories, then one can complete the stores in one clock cycle. Here a 64-bit shifter or a rotational shifter is needed as well. Instead of using two memories and still be able to avoid the stall of the processor, a write cache can be used. Here, simply a register will be used. The second store will take place while the processor continues to execute. If the next instruction in line to be executed is a load or a store, first then is the processor stalled.

In this design, the load and the store path can share shifter, i.e. it is not necessary to shift both load data and store data in the same clock cycle.

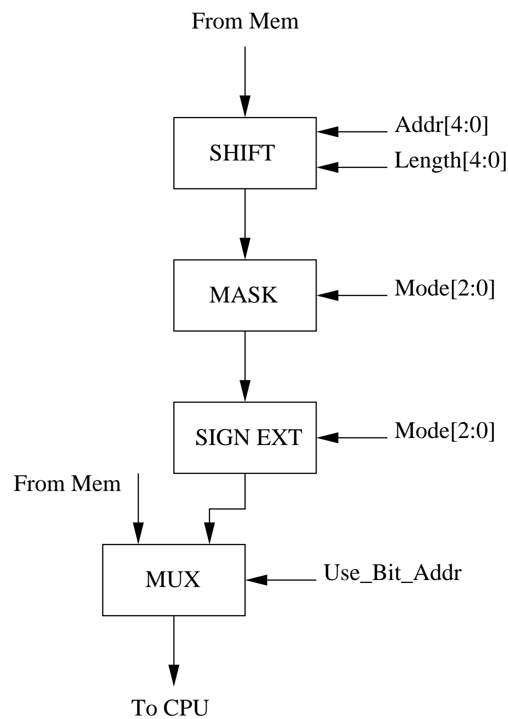

Figure 11.6: The data path for a variable from the memory to the processor. The signal Bit_Use_Mode selects between data coming directly from the memory and data being modified in the memory control module.

Alternative Design Approaches

Instead of using a left-right shifter, a rotational shifter was tried. The idea was to shift data in only one direction. A left shift by 5 can be replaced by rotational right shift of 32-5=27. This solution gives a slower and bigger design. When loading data from memory, we have to mask the data before the rotational shift and then perform sign extension. With the left-right shifter, the mask and sign extension is merged together after the shift takes place.
There are separate maskers for the load and store data in this design. A reasonable advantage could be to use a common masker for both. The choice would then be to use the write masker because it is already needed for writing to the memories. The load path would have to change order of operations. Instead of first performing the shift and then the masking as in figure 11.6, the masking would take place before the shift. This masking can be done with the always generated write mask. After the masking, the shifting is done as usual. Before the data can propagate to the processor, it might have to be sign extended. In the former approach, the

sign extension and the masking were integrated. This design turns out to also be slower and bigger. The main reason is that incoming data from the memory has to wait for the write mask to be generated before it can propagate further into the shifter.

11.4.3 Modifications in tools and code generation

The first intention was to incorporate these new instructions directly in the C-compiler and generate code with these instructions whenever needed. This turned out to be more complicated then expected. Instead, we decided to use inline assembler. For the applications we wrote, we used the C language as much as possible. We used the inline assembler only when the new load and store instructions were needed and when the address pointers for the instructions were updated. In addition, the initialization of the address pointers was modified. All the pointers are in normal mode byte oriented, but for the mode of load and store variables of arbitrary length, the address pointer need to be bit oriented. This modification is done by an additional shift three steps to the left. Most desirable would have been to incorporate this shift in the linking step of the code. This change of the address pointers are written in C. The C-compiler used was the gcc version 3.1. The simulator included with the processor needs modifications to accept the new instructions. We did not make these modifications, instead we simulated all the applications on a lower level in a Verilog simulator, modelsim from Mentor. The reason was to get a larger observability of the hardware and that the applications tested were small enough.

11.5 Motivation and Applications

In the following two chapters two different applications are discussed where one can either save program or data memory by using bit allocated memory. The first one is a simple FIR filter with either coefficients, data, or both in data lengths not equal to byte, word or double word. The second is a compression algorithm widely used in modems. Here, data has to be stored in a compact form to actually work. The savings are the reduced numbers of instructions needed for this compact storing and loading.

11.5.1 FIR filter

To use a simple example: the incoming data from the ADC is not aligned to 8, 16, or 32 bits. Suppose the precision of the ADC is only 13 bits. Instead of padding with 3 zeros to get a 16-bit word, simply store the 13 bits continuously

Bits	DM Savings	Load Penalty	FIR Penalty
1	88 %	0 %	0 %
2	75	0	0
3	62	6	1
4	50	0	0
5	38	12	1
6	25	12	1
7	12	19	2
8	0	0	0
9	44	25	3
10	38	25	3
11	31	31	3
12	25	25	3
13	19	38	4
14	12	38	4
15	6	44	5
16	0	0	0
17	47	50	6
18	44	50	6
19	41	56	6
20	38	50	6
21	34	62	7
22	31	62	7
23	28	69	8
24	25	50	6
25	22	75	8
26	19	75	8
27	16	81	9
28	12	75	8
29	9	88	10
30	6	88	10
31	3	94	10

Table 11.1: Data memory savings and load penalties for different bit sizes of the input data. The data memory savings comes from storing the variables after each other instead of aligning them to nearest upper power of 2.

in the data memory, see figure 11.1. The formula for a FIR filter is like follows: $y[i] = \sum_{k=0}^{N-1} h[k] * x[i-k]$, where N is the number taps in the filter. Even though the number of bits stored in the memory is less than the internal precision of the

processor, all the calculations performed on the data are done with full precision. The savings in the data memory depends mainly on two things. First, the number of bits compared to the native word length, i.e. to store 13 bit data instead of 16 bit gives a saving of three bits per data. Secondly, how much data is stored in total compared to the rest of the data stored. If the rest of the data stored is much bigger, this bit memory scheme will not save much.
From table 11.1 one can see the memory savings by using the bit memory loading scheme. The penalty for two loads when the data is located over the memory boundary is also calculated. First, the number of extra memory accesses is shown, column load penalty, and then the total performance degradation due to these extra memory accesses compared to the overall memory accesses of both program and data, FIR penalty. The processor chosen is not suitable for this kind of calculations. The lack of hardware loop, auto increment, and dual data fetch increases the number of instructions per lap in the FIR loop. To reduce the load penalty when a data is loaded over the memory boundary, a load cache can be included. While data is loaded in a consecutive way, the load cache is a register with the last loaded data from the memory. For random accesses within the data memory , this would not be possible.

11.5.2 Compression scheme V42bis

The V42bis algorithm is a compression scheme used in many modems. Instead of sending data byte by byte, sequences of bytes known as a codeword are sent. For example, in a text document the word “and” is frequently used. Instead of sending it byte by byte, three bytes, it can be sent as one codeword. If the codeword size is 9 bits, 3 codewords are allocated for control information, 256 for the basic characters, and the remaining 253 for codewords. By increasing the codeword size to 10 or 11 bits, the number of codewords are increased to 765 or 1789. If you allocate more bits to the codewords, the memory requirements will also increase. A tree structure is built up on both the coding and the decoding side. For more information on the algorithm see [4].
In comparison with the FIR filter, where it was advantageous to store data in a compact manner, here it is mandatory. The data will be sent over a transmission line, and we do not want any unnecessary data transmitted over this line. If we start to send padding zeros, the gain from the compression algorithm will be lost. The benefit of using bit oriented loading and storing instructions is performance and code density. With bit oriented instructions, only two things are necessary; a load/store of the codeword and an update of the bit address pointer. Without them, instructions like shift, masking, and store with load and write back are necessary. The savings in the code density for both storing and loading codewords to memory is shown in table 11.2.

Mode	**Normal**	**With bit oriented instructions**	**Savings**
Load	40 instr.	9 instr.	77 %
Store	38	9	76

Table 11.2: Instruction saving by using bit oriented load/store instructions in V42bis.

11.6 Area and timing

The area estimations are done with Physically Knowledgeable Synthesis(PKS) from Cadence version v05.12. There is no routing included, the area is only from the standard cells. The first process in a 0.35 from Austria Micro Systems(AMS). The second process is a 0.13 from UMC. The result of the synthesis is presented in table 11.3.

	Timing	**Area**	
Process		**CPU**	**CPU+BMC**
0.35	12 ns	1.65 mm^2	1.80 mm^2
	11	1.64	1.82
	10	1.68	
	9	1.74	
0.13	8 ns	1.09 mm^2	1.15 mm^2
	6	1.09	1.19
	4	1.11	1.18
	3	1.14	

Table 11.3: Area estimations and timing for the CPU standalone and the CPU together with the Bit Memory Controller(BMC).

11.7 Future work

So far we have only tried the concept with bit oriented memory instructions on two applications. There are of course many more where this would be applicable. One example is a linked list, where the pointer to the next element in the list does not need full address range. If the list spans over a maximum of 2 kByte, a byte

is 8 bits, the length of the pointers only need to be 14 bits instead of 32. 11 bits are needed to address 2 kByte and another 3 to make the address bit oriented. The comparison with 32 bits comes from the assumption that we are using a 32-bit processor.
In this paper, the bit memory load is only available from the data memory. Another application might be to load instructions from the program memory bit oriented. All instructions do not need the same amount of bits for their coding. This can compact the program memory.
For the moment the complexity of using bit oriented instruction is a major drawback. In the two applications presented in the paper the parts where these instructions are used is written with inline assembler. A more efficient approach and more user friendly approach is to incorporate the modifications directly in the compiler. This can be done by using PRAGMA and specifying certain variables to be bit oriented and then use the bit oriented load and store instructions for pushing it back and forth in memory.
So far, the simulations are only done within a Verilog/VHDL simulator, modelsim from Mentor. In order to get more accurate results a complete implementation on a chip or a FPGA would be preferable.

11.8 Conclusion

We have proposed an extension of the instruction set for a general CPU. These instructions make it possible to access the memory on bit level, instead of byte level. Two applications have been presented where these new instructions are very suitable and can save either data memory, in the FIR filter, or program memory, in the V42bis compression scheme. The load penalty from reading over the word size boarder can be avoided by splitting one memory into two. The drawback is the increased area of 20 %. For consecutive loads can a load cache, register, be used instead.

11.9 Acknowledgment

This work was financially supported by the stringent of SSF and the Center for Industrial Information Technology at Linköping Institute of Technology (CENIIT).

11.10 References

[1] Artisan memory *http://www.artisan.com/products/memory.html*.

[2] F. E. X. Nie, L. Gazsi and G. Fettweis, "A Network Processor Architecture for High-Speed Communications," in *in IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 548–557, 1999.

[3] Open RISC 1000 *http://www.opencores.com/projects/or1k/Home*.

[4] *ITU-T Recommendation V.42bis, Data compression procedures for data circuit-terminating equipment (DCE) using error correction procedures*, 1990.