Evaluation of a Floating Point Acoustic Echo Canceller Implementation

Examensarbete utfört i Datorteknik
av

Anders Dahlberg

LiTH-ISY-EX--07/4020--SE
Linköping 2007
Evaluation of a Floating Point Acoustic Echo Canceller Implementation

Examensarbete utfört i Datorteknik
vid Linköpings tekniska högskola
av

Anders Dahlberg

LiTH-ISY-EX--07/4020--SE

Handledare:

Johan Eilert
Mikael Rudberg

Examinator:

Dake Liu

Linköping 2007-05-24
Sammanfattning

This master thesis consists of implementation and evaluation of an AEC, Acoustic Echo Canceller, algorithm in a floating-point architecture. The most important question this thesis will try to answer is to determine benefits or drawbacks of using a floating-point architecture, relative a fixed-point architecture, to do AEC. In a telephony system there is two common forms of echo, line echo and acoustic echo. Acoustic echo is introduced by sound emanating from a loudspeaker, e.g. in a handsfree or speakerphone, being picked up by a microphone and then sent back to the source. The problem with this feedback is that the far-end speaker will hear one, or multiple, time-delayed version(s) of her own speech. This time-delayed version of speech is usually perceived as both confusing and annoying unless removed by the use of AEC. In this master thesis the performance of a floating-point version of a normalized least-mean-square AEC algorithm was evaluated in an environment designed and implemented to approximate live telephony calls. An instruction-set simulator and assembler available at the initiation of this master thesis were extended to enable; zero-overhead loops, modular addressing, post-increment of registers and register-write forwarding. With these improvements a bit-true assembly version was implemented capable of real-time AEC requiring 15 million instructions per second. A solution using as few as eight mantissa bits, in an external format used when storing data in memory, was found to have an insignificant effect on the selected AEC implementation’s performance. Due to the relatively low memory requirement of the selected AEC algorithm, the use of a small external format has a minor effect on the required memory size. In total this indicates that the possible reduction of the memory requirement and related energy consumption, does not justify the added complexity and energy consumption of using a floating-point architecture for the selected algorithm. Use of a floating-point format can still be advantageous in speech-related signal processing when the introduced time delay by a subband, or a similar frequency domain, solution is unacceptable. Speech algorithms that have high memory use and small introduced delay requirements are a good candidate for a floating-point digital signal processor architecture.
Abstract

This master thesis consists of implementation and evaluation of an AEC, Acoustic Echo Canceller, algorithm in a floating-point architecture. The most important question this thesis will try to answer is to determine benefits or drawbacks of using a floating-point architecture, relative a fixed-point architecture, to do AEC. In a telephony system there is two common forms of echo, line echo and acoustic echo. Acoustic echo is introduced by sound emanating from a loudspeaker, e.g. in a hands-free or speakerphone, being picked up by a microphone and then sent back to the source. The problem with this feedback is that the far-end speaker will hear one, or multiple, time-delayed version(s) of her own speech. This time-delayed version of speech is usually perceived as both confusing and annoying unless removed by the use of AEC. In this master thesis the performance of a floating-point version of a normalized least-mean-square AEC algorithm was evaluated in an environment designed and implemented to approximate live telephony calls. An instruction-set simulator and assembler available at the initiation of this master thesis were extended to enable; zero-overhead loops, modular addressing, post-increment of registers and register-write forwarding. With these improvements a bit-true assembly version was implemented capable of real-time AEC requiring 15 million instructions per second. A solution using as few as eight mantissa bits, in an external format used when storing data in memory, was found to have an insignificant effect on the selected AEC implementation’s performance. Due to the relatively low memory requirement of the selected AEC algorithm, the use of a small external format has a minor effect on the required memory size. In total this indicates that the possible reduction of the memory requirement and related energy consumption, does not justify the added complexity and energy consumption of using a floating-point architecture for the selected algorithm. Use of a floating-point format can still be advantageous in speech-related signal processing when the introduced time delay by a subband, or a similar frequency domain, solution is unacceptable. Speech algorithms that have high memory use and small introduced delay requirements are a good candidate for a floating-point digital signal processor architecture.
Acknowledgments

While working on this thesis I have received a lot of important help and support, therefore I would like to take this opportunity to express my deepest gratitude to those concerned. First of all, to my supervisor Mikael Rudberg for invaluable discussions and suggestions that have guided me through my work. My examiner Dake Liu and supervisor Johan Eilert without whom this thesis would not have been possible. To all the people at Infineon’s Linköping office. To Andre Adrian for use of his AEC implementation. Finally, a big thanks to my family and friends for always supporting me.
Chapter 1: Introduction ................................................................. 1
  1.1 Outline ......................................................................................... 1
  1.2 Objectives ................................................................................... 1
    1.2.1 Linköping University .......................................................... 2
    1.2.2 Infineon Technologies AG.................................................... 2
  1.3 Scope .......................................................................................... 2
  1.4 Acceptance Levels ...................................................................... 3
    1.4.1 Level 1 ................................................................................ 3
    1.4.2 Level 2 ................................................................................ 3
    1.4.3 Level 3 ................................................................................ 3

Chapter 2: Theory ........................................................................... 5
  2.1 Architecture ................................................................................ 5
  2.2 Digital Signal Processing .......................................................... 6
    2.2.1 Multiply and Accumulate ..................................................... 7
  2.3 Fixed and Floating Point Format .............................................. 7
    2.3.1 Dynamic Range .................................................................. 7
    2.3.2 Precision and Quantization ............................................... 8
    2.3.3 Floating Point Arithmetic .................................................. 8
  2.4 Instruction Set ........................................................................... 8
    2.4.1 RISC and CISC ................................................................ 9
  2.5 Floating Point Architecture .................................................... 10
    2.5.1 Benefits and Drawbacks .................................................... 10
  2.6 Original Floating Point Architecture ....................................... 10
    2.6.1 Architecture and Pipeline Stages ....................................... 11
    2.6.2 Bit Representation ........................................................... 12
    2.6.3 Registers ......................................................................... 13
    2.6.4 Memory .......................................................................... 13
    2.6.5 Instruction Set ................................................................ 13
  2.7 Human Auditory System ......................................................... 13
    2.7.1 Sound Perception ............................................................. 14
    2.7.2 Voiced and Unvoiced Speech .......................................... 14
  2.8 Digital Audio ............................................................................. 14
    2.8.1 Pulse Code Modulation .................................................... 15
    2.8.2 Dynamic Range and Sound to Noise Ratio ..................... 15
    2.8.3 Quantization Noise .......................................................... 15
  2.9 Echo .......................................................................................... 16
    2.9.1 Cancelling ........................................................................ 16
    2.9.2 Normalized Least Mean Square ...................................... 18
    2.9.3 Filtered and Leaky Least Mean Square ......................... 18
    2.9.4 Block and Subband AEC .................................................. 19
    2.9.5 Double Talk Detection .................................................... 19
Chapter 3: Method and Tools ........................................... 21
  3.1 Sequence of Work Description ........................................................21
  3.2 Tools ...........................................................................................22
    3.2.1 Matlab .............................................................................22
    3.2.2 GNU Compiler Collection .....................................................23
    3.2.3 Instruction Set Simulator ....................................................23
    3.2.4 Eclipse and Java ................................................................23

Chapter 4: Architecture Improvements ........................... 25
  4.1 Registers .....................................................................................26
  4.2 Pipeline .......................................................................................26
  4.3 MAC Support ................................................................................27
  4.4 Addressing Modes .........................................................................27
  4.5 Zero Overhead Loop Support ..........................................................27
  4.6 Branch Support ............................................................................28
  4.7 Instruction Encoding .................................................................28

Chapter 5: Implementation ............................................. 29
  5.1 Matlab ........................................................................................29
  5.2 Four Versions ...............................................................................30
  5.3 Double Talk Detection ...................................................................31
  5.4 Prewhitening ................................................................................31
  5.5 Assembly Implementation .............................................................31

Chapter 6: Quantization .................................................. 33
  6.1 Echo Path ....................................................................................33
  6.2 AEC Performance .........................................................................34
    6.2.1 Accumulator Mantissa Bits ..................................................35
    6.2.2 Internal and External Mantissa Bits ......................................35
    6.2.3 External Mantissa Bits ........................................................36
  6.3 Bit Length Dependent Sound Quality Evaluation .................................36
  6.4 Quantization Noise ........................................................................37

Chapter 7: Evaluation of Results and Future Work .......... 41
  7.1 Results ........................................................................................41
    7.1.1 Floating Point C++ AEC Implementation ...............................41
    7.1.2 Bit True Assembly AEC Implementation .................................41
    7.1.3 Architecture Improvements ..................................................41
    7.1.4 Live Test Implementation ....................................................41
    7.1.5 Quantization .....................................................................42
  7.2 Conclusions .................................................................................42
  7.3 Future Work ...............................................................................42
Appendix A: Instruction Set Reference
Appendix B: Instruction Encoding
Appendix C: Data Path
Appendix D: Code
Appendix E: Original AEC Implementation
Chapter 1: Introduction

This chapter contains information about questions this thesis will try to answer, where the work was performed, a section detailing aspects that is not covered and a section specifying levels of acceptance.

1.1 Outline
Background information necessary to understand in order to fully appreciate the rest of the document is found in “Chapter 2: Theory” on page 5. The practical part of the thesis is covered in “Chapter 3: Method and Tools” on page 21. A basic sequence of work is found there and information about tools used. In “Chapter 4: Architecture Improvements” on page 25, “Chapter 5: Implementation” on page 29 and “Chapter 6: Quantization” on page 33, a description of the work to provide answers to the questions in section “1.2 Objectives” is presented. In the final chapter, “Chapter 7: Evaluation of Results and Future Work” on page 41, the thesis work is evaluated and there is also a section detailing future work. Important sections of the document include:

- The boundaries of this thesis are available in section “1.2 Objectives”, section “1.3 Scope” and section “7.3 Future Work” on page 42.
- Summary of the results is available in section “7.1 Results” on page 41, information about future work in section “7.3 Future Work” on page 42.
- A list of used abbreviations is available in “Abbreviations” on page 45.
- A collection of used sources is available in “Bibliography” on page 43.

1.2 Objectives
This document is the master thesis of Anders Dahlberg, written as part of my education at Linköpings Tekniska Högskola. The thesis work was performed during winter and spring 2006 and 2007 respectively. Dake Liu was in charge of examination, Johan Eilert, LiTH, and Mikael Rudberg, Infineon Technologies AG, were supervisors. The master thesis consists of an evaluation of digital signal processing in the form of an acoustic echo-cancellation algorithm, using a floating-point architecture. Questions that the thesis tries to give answers to include:
• How do floating-point specific quantization errors affect the AEC algorithm?
• What are the differences in perceived sound quality of a floating-point format relative a comparable fixed-point format?
• Which extensions or modifications of a given floating-point architecture\(^1\) are needed to achieve adequate performance\(^2\) of the chosen AEC algorithm\(^3\)?
• When using a floating-point architecture, are there any benefits or drawbacks compared to when using a fixed-point architecture and if so, what are they?

1.2.1 Linköping University

Founded in 1975, Linköping university today have 3 500 employees and 27 000 students. This master thesis belong to the department of electrical engineering and specifically the computer engineering section. \[1\]

1.2.2 Infineon Technologies AG

Infineon was founded in 1999 as a spin off from its former parent-company Siemens. During the fiscal year of 2006, Infineon employed approximately 42 000 throughout the world and achieved sales of 7.9 billion euros. Infineon’s main focus is semiconductor products and system solutions that target energy efficiency, mobility and security. Infineon is divided in two large segments, where “automotive, industrial and multimarket” is one, that as the name implies focus on developing technology suited for the automotive industry and industrial production in general. A summary of product categories include; sensors, microcontrollers, power integrated circuits, transceivers, wireless chipsets and plastic optical fibers. Communication solutions, focus on research and development of semiconductor products enabling high-speed data transmission for cellular, wireless and wired communications. Example of products include integrated circuits enabling one, or multiple, of the following technologies; Bluetooth, GPS, cellular base stations, DECT, xDSL (e.g. ADSL, VDSL) and WLAN. This thesis was performed in Infineon’s Linköping office, primarily devoted to concept engineering of DECT-based integrated circuits, i.e. belonging to communication solutions. \[2\],\[3\]

1.3 Scope

This section contains information about areas this thesis will not cover. Obviously, only areas related to the thesis that could have been evaluated are mentioned. This thesis will not cover:

• Economical aspects of choosing a floating-point architecture instead of fixed point.
• Detailed information regarding performance and qualities of different AEC algorithms. In this thesis an AEC algorithm was chosen initially as a reference and it is used as base-performance metric to which different floating-point formats can be compared.
• Optimizations of assembly code other than what is necessary to evaluate benefits and drawbacks of different hardware solutions.

---

1. See section “2.6 Original Floating Point Architecture” on page 10.
2. Adequate performance; a real time implementation on relatively moderate hardware should be possible. I.e. the implementation of the algorithm in assembly should not require more than approximately 50 MIPS.
3. See section “2.9.2 Normalized Least Mean Square” on page 18.
1.4 Acceptance Levels

To be able to more easily plan and carry out the thesis work, three levels of acceptance were defined. The levels are increasingly more complex, where each level adds practical and or additional theoretical results. The acceptance levels will also work as milestones and unless special requirements are found, the practical and most theoretical parts of an acceptance level should be completed before work is begun on practical parts of the following level.

1.4.1 Level 1

Basic functionality and the most important parts of the theory should be covered here. This is the lowest level available. Theoretical and practical aspects that should be completed before this level is accepted follow below in bullet form:

- Theoretical background to instruction sets, floating-point arithmetic, quantization noise, the original architecture\(^1\) and echo cancellation.
- Functional assembly code that can be run with the instruction-set simulator achieving basic AEC functionality.
- Future theoretical, basic, hardware modifications. Justified by theory and results from experiments. If no improvements to the hardware can be found, drawbacks of suggested improvements should be detailed.

1.4.2 Level 2

This level is, compared to the first level, more specific concerning theoretical results. More detail in the documentation and better explanations are required. The aspects, in addition to those specified in level 1, follow below in bullet form:

- A more detailed explanation of quantization errors introduced when using a floating-point processor compared to a fixed point.
- Functional assembly code showcasing the effects of architecture improvements.
- Implemented and explained architecture modifications.

1.4.3 Level 3

The final level contains more functional aspects as the bulk of the thesis theory should be covered in the previous two levels. Theory in this level is limited to explanation of the technologies used in implementation of the extended functionality. Aspects not covered in earlier levels are listed below in bullet form:

- Explanation of technologies introduced and used in this level.
- Modifications of hardware to improve quality and or performance of the functional parts.

---

1. See “2.6 Original Floating Point Architecture” on page 10.
Chapter 2: Theory

This chapter contains information the reader should be familiar with to be able to understand the following chapters. A reader experienced with AEC, DSP and embedded computing can give this part of the document a cursory glance or directly skip forward to “Chapter 3: Method and Tools” on page 21.

2.1 Architecture

Following below as figure ”2-1 High level view of new architecture” is the final design of the architecture that was developed as a part of this master thesis. The following sections contain information relevant to the various components, and reason for use, of the architecture below.

![Figure 2-1: High level view of new architecture](image_url)
2.2 Digital Signal Processing

Digital signal processing is a form of data processing, where the data is a discrete-time\(^1\) stream of quantized values, e.g. zeros and ones, as opposed to analog signal processing where the data is a continuous stream of analog values. A good example of where digital signal processing is used today is data compression, where audio and video needs to be compressed to be transmitted or stored and later decompressed to be heard or seen. Examples of products are quick and small digital cameras, mobile phones and set-top boxes. Compared with analog signal processing, e.g. analog filters, amplifiers, some advantages of using digital signal processing instead are; flexibility - an existing software algorithm can be updated to a new version while still being able to run on the old hardware, reproducibility - as digital signal processing works with quantized and discrete-time values that can be reproduced easily compared with analog components that usually introduce subtle variations in the continuous data stream. A related advantage is that, given a digital signal processing implementation of an algorithm, i.e. discrete-time with quantized values, validation and even verification is relatively easy. This thanks to software programming tools, e.g. MATLAB, C/C+++, Java, which enables developers to evaluate the DSP algorithm on their personal computer, given sufficient processing power and a realistic\(^2\) simulator. A DSP architecture can be a regular personal computer, as an example; a person watching a movie or using internet telephony to talk to a friend with the aid of their computer is able to claim that it is a “DSP computer”\(^3\). Of course, one can then argue that a personal computer is very expensive and has, relative its energy consumption, very poor performance. Given the previous example, it is not hard to come up with a list of properties that a good\(^4\) DSP architecture should provide:

- Enable efficient implementation of common algorithms\(^5\).
- Low energy consumption.
- Low cost.
- Flexible.
- Software development tools, e.g. assemblers, compilers and simulators.
- Hardware testing tools.

---

1. Discrete-time here refers to that values are available, i.e. valid, at discrete time events, e.g. at a clock flank.
2. Obviously a realistic simulator for an analog signal-processing algorithm also enables easy verification and testing, the difference being that it is easier to develop a digital simulator than a comparably realistic analog simulator.
3. A personal computer without DSP capability, is either analog or not very useful at all.
4. Here good implies that the DSP architecture can be used by real-time embedded applications.
5. E.g. FFT, Viterbi decoder.
2.2.1 Multiply and Accumulate
A feature that in practice is required by a DSP architecture satisfying the requirements stated above is a MAC, i.e. multiply and accumulate, unit. A MAC instruction performs two operations, multiplication and addition, in a single instruction and commonly in a single clock cycle\(^1\). As most digital signal processing tasks contains at least one filtering operation, the use of a MAC can drastically improve the performance. This is due to that filters, either in FIR or IIR format, in a digital signal processor usually is implemented as a series of additions of the current sample, i.e. input, value multiplied with a filter coefficient. When the number of filter coefficients grow large, or the sampling rate\(^2\) high, these multiplications and additions can become a serious bottleneck of the algorithm. [4],[2],[6]

2.3 Fixed and Floating Point Format
An important aspect when implementing an algorithm on a DSP architecture is the choice of digital representation of the numerals. The fundamental difference is if the values are represented as integer fixed-point values or as fixed or floating-point fractional values. As fractional values can not be represented when using an integer fixed-point representation; values are instead scaled. E.g. instead of an initial range of 0.0 to 1.0 a 16-bit fixed-point integer range of 0 to 65536 is used. In the floating-point format there is a mantissa part, giving the value’s precision, a radix and an exponent part, giving the value’s scale, e.g. radix value to the power of the exponent value\(^3\), see equation 2-1 below. I.e. compared to a floating-point format, a fixed-point format only use the mantissa part. In the following parts of this document, unless otherwise specified, fixed-point refers to the integer fixed-point format and floating point to a sign and magnitude\(^4\), radix-2 floating-point format. [7]

\[
d = m \cdot r^e \\
\]

\(d = \text{value in decimal format}\) \(m = \text{Mantissa}\) \(r = \text{Radix}\) \(e = \text{Exponent}\)

2.3.1 Dynamic Range
A large benefit of using floating-point formatted values compared to fixed point is the increased dynamic range. Dynamic range, see section “2.8.2 Dynamic Range and Sound to Noise Ratio” on page 15, is the relationship, usually represented as a decibel scale, between the smallest, excluding zero, and largest positive number that can be represented. To give an idea of the difference between a fixed point and a floating-point representation; the dynamic range of a 16-bits signed-integer fixed-point format is approximately 90 dB, while a floating-point format with as few as 5 or 6 exponent bits have a dynamic range of over 190 dB or 380 dB respectively\(^5\). [7]

1. The final result may take longer to appear, but the MAC instructions can be continuously executed once every clock cycle.
2. Sampling rate is explained in section “2.8 Digital Audio” on page 14.
3. A more detailed description of fixed and floating-point formats is available in, e.g., [7], chapters 1 and 8.
4. An intuitive way of representing negative numbers with a binary stream is to use the decimal-number format’s concept of a “sign digit”. In sign and magnitude, the most significant bit represents the sign of the number. By convention a 1 represents a negative number and 0 a positive.
5. The dB values for floating point here assumes that denormals are not allowed, i.e. the mantissa is always normalized.
2.3.2 Precision and Quantization

Even though the dynamic range of the floating-point format is a lot larger than that of common fixed-point formats, the choice of format to use is not easy. The dynamic range of the floating-point format does not come without a price attached; for the same number of total bits used, compared with the fixed-point format, a trade-off between dynamic range and precision is made. The bits used to represent the exponent do not directly contribute to the precision of the result, only the scale. This implies that for a given number of bits, the floating-point format will have less\(^1\) precision than a fixed-point format. Even worse is the situation if the domain, where a floating-point format is to be used, is dominated by relatively large values\(^2\). The reason being that the quantization, or round off, error of the floating-point format depends on the magnitude of the numeral that is to be represented. The loss of precision is scaled by the radix to the power of the exponent value\(^3\). [7]

2.3.3 Floating Point Arithmetic

In embedded computing it is common with processors that does not, natively, support floating-point arithmetic operations. The reason for this is that the hardware necessary to implement floating-point arithmetic is usually more complex than its fixed-point counterpart. Compared to fixed-point addition, or subtraction, floating-point addition is a bit more complicated. When performing floating-point addition of two values the two mantissas must first be aligned according to each value’s magnitude, in other words shifted left or right the amount specified in their exponent. When the mantissas are aligned, the values can be added using the same procedure as for a similar fixed-point addition. In contrast to the fixed-point case, floating-point addition is not complete after the addition is performed. Floating-point values use a normalized mantissa, in practice the normalization is performed by shifting the mantissa left or right and modifying the exponent value accordingly. The final step is rounding to ensure that the result fits in the bits available. Analogous to the case with addition and subtraction, floating-point multiplication differs compared to its fixed-point counterparts. The part that differs is that multiplication of two floating-point values is performed first by addition or subtraction of their exponents. Following alignment, the mantissas can be multiplied using the same procedure as for fixed-point values. The resulting value is then normalized and rounding is performed. [5],[7]

2.4 Instruction Set

The caption of this section, “Instruction set”, in the case of a DSP architecture refers to the set of assembly instructions available. A common acronym is ISA, instruction-set architecture. The “architecture” part implies that a hardware architecture usually can be defined based on the instructions available. The implication is established as no matter how complex and advanced a given architecture’s hardware is, if the available instruction set only contains a few select primitive and shallow instructions, there is usually no gain from complex hardware, e.g. if all of the instructions combined only use a small subset of the complex hardware. Instructions can be divided in four distinct groups; arithmetic instructions, logic instructions, data instructions and branch, i.e. program-flow control,

---

1. Equal precision implies that the number of exponent bits is zero, i.e. the floating point format can be considered to be a fixed point format.
2. Here it is assumed the values are not larger than what can be represented in the fixed-point format.
3. More on this in section “2.8.3 Quantization Noise” on page 15.
4. An instruction using, e.g., two different addressing modes is here considered as two unique instructions.
instructions. In table ”2-1 Instruction groups” below, each group is presented including a few example instructions that belong to respective group. [6]

<table>
<thead>
<tr>
<th>Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>The instructions contained in this group are concerned with arithmetic operations on fixed or floating-point coded binary data. A few examples are addition, subtraction and multiplication.</td>
</tr>
<tr>
<td>Logic</td>
<td>Logic instructions are similar to arithmetic instructions, but instead of operating on fixed or floating-point data they operate on the individual bits. Included in this group are instructions such as and, exclusive or and bitwise negation.</td>
</tr>
<tr>
<td>Data</td>
<td>Different from the preceding two groups, the data instructions do not usually perform any calculations, instead they focus on loading and storing data; be it individual bits, fixed or floating-point formatted. The most common instructions in this group are load from memory and store in memory.</td>
</tr>
<tr>
<td>Branch</td>
<td>The branch group of instructions contain instructions that change the flow of code by allowing execution to jump from one point in a program to another. Instructions range from jump to a specified program point to more complex instructions that jumps conditionally, i.e. depending on a value. Examples of this latter type of branch instructions include branch if value not equals zero and branch if value equals zero.</td>
</tr>
</tbody>
</table>

Table 2-1: Instruction groups

**2.4.1 RISC and CISC**

An architecture is usually classified as being either a RISC or a CISC. RISC is an acronym for reduced instruction-set computer. As the name implies, architectures that only contains a few, and simple instructions with limited support of different addressing modes, belong to this category. RISC architectures is based on the assumption that the most common instructions should be made to execute as fast and efficient as possible, and the less common or more complex instructions should be implemented by the user, or a compiler, in assembly code as a series of the more primitive instructions provided. The reason for having few instructions and reduced availability of addressing modes is that RISC's rely on pipelining to achieve high performance. Pipelining is the use of parallel functional units, where each unit is specialized on performing a limited part of an instruction, fetch and decode instruction, read from register file et cetera. The use of pipelining usually enables a RISC to execute instructions in a single or two clock cycles. CISC, or complex instruction-set computer, is the opposite of RISC, instead of focusing on providing a few primitive instructions, there is a broad spectra of instructions provided, ranging from primitive to complex and several addressing modes available for each instruction. [6], [8]
2.5 Floating Point Architecture

An important difference between a floating-point architecture and a more regular fixed-point architecture is how the execution of arithmetic instructions is implemented. As the floating-point arithmetic instructions require two extra operations, mantissa alignment and normalization, compared to their fixed-point counterparts, the corresponding hardware is accordingly more complex. Usually this extra hardware complexity manifests itself as both larger chip size and higher energy consumption. Compared to a fixed-point architecture there is not a large difference in how logic instructions are implemented or designed. This can be understood by realizing that when using a logic, i.e. bitwise, operator, the most important aspect is whether or not a bit is zero or one, this implies that the values representation, be it fixed or floating point, is irrelevant. In most cases, floating-point architectures include some support, e.g. instructions, for fixed-point values. This support is provided as fixed-point values are conveniently used, e.g., as counters and more importantly, to support memory addressing.

2.5.1 Benefits and Drawbacks

A large drawback of using a floating-point architecture is, as mentioned above, the increased complexity of hardware required to implement floating-point arithmetic instructions. Floating-point architectures are accordingly not very well suited for real-time embedded-computing projects where; production volumes are large, e.g. millions of devices, development time is long, the factors low cost and energy consumption are very important. Benefits of using a floating-point architecture can still be found though, some examples are when:

- An algorithms input signals and or internal coefficients have a large dynamic range and are highly time varying or uncorrelated, i.e. relatively stochastic, thus increasing the problem of scaling required by a, i.e. with limited dynamic range, fixed-point solution.
- Development time is required to be short due to high development cost, e.g. the benefit of not manually having to scale values internally intuitively implies that development time is reduced.
- The use of a floating-point architecture can reduce the data and or code memory-size requirements, reducing the overall chip size, and possibly energy consumption, if the increase in chip size to support floating-point arithmetic is less than the decrease in memory related chip size and energy consumption.

[6],[9]

2.6 Original Floating Point Architecture

As a convenience to the reader, an overview of the architecture used as a base for this master thesis is given below as figure "2-2 Original architecture". After the architecture and pipeline overview, more detailed information regarding bit representations, registers, memories and instruction set follows in subsequent sections.

---

1. See section “2.3.3 Floating Point Arithmetic” on page 8.
2. The information provided in this section, including sub-sections, is found in its original form in [9].
2.6.1 Architecture and Pipeline Stages

As seen in the figure above there are some notable properties in this architecture. The fixed and floating-point arithmetic and logic units are separate, i.e. their implementation do not share common functionality. The program, data and constant memories are separate and the fixed-point ALU has access to a direct sequential bit view of the data memory. The floating-point ALU has access to two memory values concurrently; one value from data memory and one from the read-only constant memory. The architecture above was designed to be heavily pipelined. The reason for a large number of pipeline stages was the assumption that if the architecture can be run at a high maximum clock frequency, core voltage can be scaled down, yielding lower power consumption, and clock frequency, while still maintaining a high efficiency. A general disadvantage of using this relatively large amount of pipeline stages is that to achieve maximum performance, instructions need to be scheduled in a non-trivial manner. The execution pipeline and its different stages is available in figure ”2-3 Original pipeline”[9].

1. For more information about the data path of fixed and floating-point in the original architecture, see [9].
2.6.2 Bit Representation

The internal and external representation of binary values in the original architecture are available in the three following tables. Internal represents the format used in registers and buses, external the format used when storing data in memory.

### Fixed-point format

<table>
<thead>
<tr>
<th>22</th>
<th>...</th>
<th>16</th>
<th>15</th>
<th>...</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty / ignored</td>
<td>value</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2-2: fixed-point format

### Internal floating-point format

<table>
<thead>
<tr>
<th>22</th>
<th>21</th>
<th>...</th>
<th>16</th>
<th>15</th>
<th>...</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign bit</td>
<td>exponent ([-32, 31])</td>
<td>Two’s complement</td>
<td>mantissa ([0, 65535])</td>
<td>unsigned</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2-3: internal floating-point format

### External floating-point format

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>...</th>
<th>10</th>
<th>9</th>
<th>...</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign bit</td>
<td>exponent ([-16, 15])</td>
<td>Two’s complement</td>
<td>mantissa ([0, 1023])</td>
<td>unsigned</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2-4: external floating-point format

For the internal and external floating-point formats, converting a number to decimal is calculated as 

\((-1)^{\text{sign}}, 2^{\text{exponent} - 11} \cdot \left(1 + \frac{\text{mantissa}}{65536}\right)\) and 

\((-1)^{\text{sign}}, 2^{\text{exponent} - 11} \cdot \left(1 + \frac{\text{mantissa}}{1024}\right)\) respectively. Except when exponent equals -32, internal, and -16, external, as this signifies that the number represents zero. [9]
2.6.3 Registers
The original architecture includes 16 general-purpose registers that are 23 bits wide. Fixed-point instructions in general only use the 16 least-significant bits of the registers, fixed-point register-file read and write operations does not modify or use the remaining bits. Floating-point instructions uses all the available bits. There is also 16 special-purpose registers available, in the original architecture the special-purpose registers were used to; accelerate access to bits in memory, enable a circular buffer, provide auto-incrementing memory pointers\(^1\) and enable communication with peripherals.\(^9\)

2.6.4 Memory
There is three different memories available in the original architecture: program, data and constant memory. The program memory can contain up to 64k words, where each word is 24 bits wide. As only the fetch-instruction unit can use the program memory, it is not easy to use this memory to store tables of constants for use in user programs. The data memory is also 64k words, but only 16 bits wide. The reason for the choice of 64k words was, given the implementation, that the CPU only could reference that amount of memory. The third and final memory is the constant memory, as this memory is read-only and accessed by the use of an auto-incrementing 16-bit pointer, the memory size is bound by the maximum address this pointer can represent.\(^9\)

2.6.5 Instruction Set
The original instruction set was by design limited and RISC like. Shift instructions are not included and there is only branch instructions that compare values with zero. Instead, some task specific instructions are included\(^2\).\(^9\)

2.7 Human Auditory System
An ear consists of three basic building blocks; the outer ear, the middle ear and the inner ear. The outer ear amplifies, or attenuates, sound waves according to their spatial direction and frequency. The middle ear is responsible for avoiding impedance mismatch, i.e. reflections, when sound waves from the outer ear are transformed to waves in the liquid-filled inner ear. The inner ear analyzes the sound and transmit the result, e.g. frequency information, by way of neural signals to the brain’s auditory cortex. A fully functional human ear is able to detect sounds in an approximate frequency-range of 20 Hz to 20 kHz and has an effective dynamic-range of 100-120 dB SPL.\(^3\). In practice the best results, i.e. sensitivity, is achieved for sounds in the frequency region of 2 kHz to 5 kHz. The previous text was only a cursory glance of our perception of sound, the entire process is quite complex and extensive research has been, and is, performed with the purpose of furthering our understanding of how the human auditory system works.\(^4\). In the following section a few effects, selected as relevant for this master thesis, related to how we perceive sound are explained.

---

1. Not implemented in the original instruction-set simulator.
2. For more details on the original instruction set, see “Appendix A: Instruction Set Reference”.
3. 0-dB SPL is a reference pressure level, in air usually defined as \(20 \mu\text{Pa}\). This level is the approximate threshold of hearing in the most sensitive frequency region. See, e.g., section “2.8.2 Dynamic Range and Sound to Noise Ratio” on page 15 and \([19]\).
4. See, e.g., \([19]\).
2.7.1 Sound Perception

Masking is an effect whereby a given sound become difficult to distinguish due to the presence of another simultaneous sound. An example might be trying to have a conversation on a sidewalk during high traffic, the sounds of the traffic, i.e. noise, masks the sounds relating to the conversation. In the frequency domain, a masking sound affect both lower frequencies and higher frequencies\(^1\), although the masking effect is a lot stronger for higher frequencies compared to lower. An example of this is that low frequency sound emitted from, e.g., a computer fan can mask higher frequency sounds, e.g. high-frequency parts of speech, while the same high-frequency speech parts are unlikely to mask the noise emitted by the computer fan. When the masking sound does not occur simultaneously with the original sound, i.e. in the time domain, the effect is called either forward or backward masking. Forward masking occurs when a preceding sound masks a later sound to some extent, backward masking is, perhaps surprisingly, the same effect but reversed, i.e. a later sound masks an earlier sound. The explanation to how a later version can mask an earlier introduced sound is related to the physiology of our ear. If the masking sound is stronger than the earlier sound, parts of the masker can modify the current sound due to the ear’s inherent processing time and time taken for the neural ear-to-brain communication. Even though backward masking is theoretically possible, most effects of temporal, i.e. forward or backward, masking are in practice related to forward masking[19].

2.7.2 Voiced and Unvoiced Speech

An important distinction to make when dealing with speech is whether a sound segment is voiced or unvoiced\(^2\). Voiced is a result of vocal sounds, i.e. periodic, while unvoiced sounds are related to consonant sounds, i.e. aperiodic or noise like. [16]

2.8 Digital Audio

As the name implies, digital audio is audio stored or represented in a digital format. Digital audio is characterized by sampling-rate, bit-depth and number of channels, where the sampling-rate specify the bandwidth that can be represented and bit depth determines the available dynamic range\(^3\). Bandwidth is a range of frequencies, larger bandwidth implies that more frequencies are available. A large bandwidth thus gives the ability to represent both low-frequency sounds, i.e. bass, and high-frequency sounds, i.e. discant. Digital audio is also usually defined by the number of channels used, if there is only one channel, the audio format is mono, two channels stereo, three and more channels the audio is usually referred to as surround sound. Bit rate is an aggregated description that specify the number of bits that needs to be processed per second to playback, in real time, a digital audio file encoded with a given; bit depth, sampling rate and number of channels. A higher bit rate enables better audio fidelity, i.e. how well the digital audio corresponds with its source, and perceived audio quality. Where analog audio, e.g. vinyl or compact-cassette records, is limited by physical aspects, digital audio have similar limitations. Increasing sampling rate, resolution or the number of channels, also increases the number of bits required to represent a given length of sound. This increase in size can be inconvenient if the data is to be transmitted or stored. To reduce size-related problems for

---

1. Higher and lower here refers to frequencies higher or lower than the individual frequencies the masking sound is composed of.
2. Obviously it can also be “silent”, i.e. the sound segment is not speech at all. E.g. a natural pause in a conversation.
3. See section “2.8.2 Dynamic Range and Sound to Noise Ratio”.

transmission and storage, a lot of different schemes to compress digital sound have been developed. These compression schemes can be divided in two distinct categories, lossy and lossless. Lossless compression allows the identical source to be reconstructed from a compressed version, while lossy increase achievable compression by, more or less, ignoring reconstruction. [19], [20]

2.8.1 Pulse Code Modulation

Audio signals can be digitally encoded in multiple different forms, one of the most common and widespread is the use of pulse-code modulation. This form of modulation, i.e. information encoding, use a conceptually simple technique to encode audio. The analog sound signal is sampled, e.g. by the use of an analog-to-digital converter, and the sampled value is converted to a binary number. In this way the analog sound is converted to a series of binary coded values. A common format used to store pulse-code modulated data, in the technology field of telephony, is the 8 kHz 16-bit fixed-point single-channel PCM format. In this format each binary-code value is represented as a 16-bit fixed-point value and the original analog sound signal is sampled at 8 kHz. Lossy or lossless compression schemes can be used to reduce the size of PCM-coded data, in this master thesis the PCM format is used without compression. [20]

2.8.2 Dynamic Range and Sound to Noise Ratio

Dynamic range is defined as the difference, usually in decibel, between the maximum and minimum absolute value that can be represented. Sound to noise ratio is similarly defined as the difference in decibel between signal value and background noise level, e.g. ambient sounds or quantization noise. If a fixed-point format is used to represent a signal and the quantization effects is random and uniform, equation 2-2 gives the DR and SNR of a signal represented with n bits.

\[
DR = SNR = 20 \cdot \log_{10}(2^n) = 6.02 \cdot n \\
\text{Equation 2-2: Calculation of fixed-point formats DR and SNR}
\]

Similarly if a floating-point format is used and denormal values are not used, equation 2-3 gives the DR and SNR of a signal represented with e exponent bits and m mantissa bits. [7]

\[
DR = 6.02 \cdot 2^e \\
SNR = 6.02 \cdot m \\
\text{Equation 2-3: Calculation of floating-point formats DR and SNR}
\]

2.8.3 Quantization Noise

When trying to represent a numeral with a finite-precision fixed or floating-point value it is often found that the numeral can not be represented exactly. The error introduced when approximating a given numeral with a fixed or floating-point value, i.e. with less resolution available, is often referred to as quantization noise. A common occurrence of quantization noise is when a value is scaled down to use less resolution, e.g. to avoid overflow due to limited dynamic range. If a floating-point format is used, the mantissa is normalized and consequently each arithmetic instruction can require rounding and truncation to be performed, thereby introducing quantization noise. [5]

1. See section “2.8.3 Quantization Noise”.
2. Value that have the minimum exponent available and larger than zero, but not large enough to allow the mantissa to be normalized, thus the name “denormal”. See, e.g., [7].
3. I.e. to use the entire resolution available. Denormal values are not considered here.
2.9 Echo

Two quotations from a dictionary regarding echo:

- “A repetition of sound produced by the reflection of sound waves from a wall, mountain, or other obstructing surface.”
- “A sound heard again near its source after being reflected.”[8]

In a telephony system there is two common forms of echo, line echo and acoustic echo. Line echo, also known as network echo, is introduced by impedance mismatch, i.e. signal reflections, when a hybrid converts a local two-wire call to a long(er)-distance four-wire call. Acoustic echo is introduced by sound emanating from a loudspeaker, e.g. a handsfree or speakerphone, being picked up by a microphone and then sent back to the source. The problem with this\(^1\) feedback is that the far-end speaker will hear a, or multiple, time-delayed version(s) of her own speech. This time-delayed version of speech, i.e. echo, is usually perceived as both confusing and annoying. When there is many reflections not separated enough from their source to be perceived as distinct echoes, these reflections are said to add reverberation to a sound. Reverberation is usually experienced in cathedrals and other similar large structures designed for acoustic performance. [6],[18]

2.9.1 Cancelling

As mentioned in the previous section there is two common forms of echo; line and acoustic echo. The cancellation of the two different types of echo have separate design and implementation constraints. In this master thesis a normalized least-mean-square algorithm, see section “2.9.2 Normalized Least Mean Square” on page 18, was initially chosen\(^2\). Because of this choice, this and following sections will mostly concern LMS-based acoustic-echo cancellers performed on signals with an approximate quality of DECT telephony, see section “2.8 Digital Audio” on page 14, using a single loudspeaker and microphone. Acoustic echo is received when there is an acoustic connection between the loudspeaker used to represent the far-end speaker and the microphone used by the near-end speaker. In order to avoid echo, both near-end and far-end needs to be echo cancelled. An overview of an LMS-based acoustic echo canceller is available in figure “2-4 Overview of a LMS acoustic-echo canceller” and an initial mathematical description is given in equation ”2-4 Echo-path description and echo cancellation”\(^3\).

---

1. Small echo delays are usually no problem, increasingly larger delays will is both annoying and confusing.
2. Desired by a supervisor.
3. Notice that the variables in the equations are named to be consistent with adaptive filter theory. Consequently the somewhat confusing terminology where, e.g., the error signal \(e\) is in practice a desired signal, i.e. near-end speech with echo from far end removed.
The goal of AEC algorithm's is to minimize the echo feedback, in practice this is usually performed by minimizing the mean-square error of the difference between the real echo-path system and its estimate. As seen in equation 2-5, result formed by substitution of equation 2-4 into left hand side, the simplification that the near-end speech is uncorrelated with, e.g. orthogonal to, both near-end ambient noise and far-end echo is used.

\[ d(n) = h^T(n)x(n) + v(n) + w(n) \]
\[ x = \text{Far-end speech} \]
\[ h = \text{Echo-path system} \]
\[ v + w = \text{Near-end speech and ambient noise} \]
\[ e(n) = d(n) - \hat{h}^T(n)x(n) \]
\[ d = \text{Near-end speech, far-end echo and noise} \]
\[ \hat{h} = \text{Estimate of echo-path system} \]
\[ e = \text{Estimated error} \]

Equation 2-4: Echo-path description and echo cancellation

The goal of AEC algorithm's is to minimize the echo feedback, in practice this is usually performed by minimizing the mean-square error of the difference between the real echo-path system and its estimate. As seen in equation 2-5, result formed by substitution of equation 2-4 into left hand side, the simplification that the near-end speech is uncorrelated with, e.g. orthogonal to, both near-end ambient noise and far-end echo is used.

\[
\begin{align*}
E\{|h^T x - \hat{h}^T x|^2\} &= E\{|v(n) - w(n) + e(n)|^2\} \\
&= E\{|v(n)|^2\} + E\{|w(n)|^2\} + E\{|e(n)|^2\}
\end{align*}
\]

Equation 2-5: Mean square error

With the above mentioned simplification, the only term in the equation above that depends on the echo-path system is the last one. This is quite practical as \( e \) is a quantity that can be easily measured compared to both \( v \) and \( w \). In order to be able to judge performance of different echo-cancellation algorithms, a couple of different measurements have been defined. One of the most commonly used is Echo Return Loss Enhancement, ERLE:

\[ ERLE = 10 \log_{10} \left( \frac{BA\{d(n)^2\}}{BA\{e(n)^2\}} \right) \]

Equation 2-6: Echo-return-loss enhancement

Obviously this method is only a guideline to whether an algorithm performs well or not, a large ERLE does not imply that the algorithm achieves good perceived sound quality. The opposite implication, i.e. that an algorithm that achieves good perceived sound quality have large ERLE, is more often the case. ERLE is also highly dependent on ERL; echo return loss, the decrease in dB of the far-end speech due to inherent attenuation in the echo path. [10],[11],[12]

1. See section “2.9.3 Filtered and Leaky Least Mean Square” on page 18 regarding this assumption.
2. Here: removing echo and keeping near-end speech and ambient noise intact
2.9.2 Normalized Least Mean Square

NLMS is an acronym for normalized least mean square and the algorithm is an extension of the least-mean-square algorithm. These two algorithms are online, or stochastic, gradient-descent based algorithms; gradient descent as they try to find a local minimum of a function - here trying to minimize echo, online as an approximative gradient is used instead of the true gradient.

\[
\hat{h}(n + 1) = \hat{h}(n) + \mu \hat{x}(n)^T x(n)
\]

\[
\hat{h}(n + 1) = \text{Updated system estimate} \quad \mu = \text{Step-size scalar}
\]

Equation 2-7: New estimate of echo-path system

The step-size constant, in equation "2-8 New step-size scalar", determines both convergence speed and misadjustment, i.e. steady-state error. A large step-size constant implies quick convergence time and a large misadjustment, while a small step size implies the opposite. Both the use of a very large, or a very small, step-size constant also increase the risk of divergence of the algorithm. The fastest convergence time of a least-mean-square based algorithm is achieved when the step-size value \( \mu \) is set to a value close to twice the reciprocal of the insignal's autocorrelation-matrix' largest eigenvalue.

This is approximated in the normalized least-mean-square algorithm by the use of a scaling factor that approximate the power of the insignal, see the scaling factor in equation 2-8. If a larger step-size value is used, the algorithm is not guaranteed to converge. The increased risk of divergence due to a large step-size value suggests that use of a very small value instead, would lead to guaranteed convergence at a cost of slower convergence time. This is not the case when a practical implementation is to be developed, as finite-precision effects become important. The use of a small step-size value coupled with limited precision leads to problems with excitation of the system coefficients. For more on how to deal with these problems, see the following section. [10],[11],[12]

\[
\mu = \frac{\mu_0}{x^T x + \varepsilon}
\]

\( \mu_0 = \text{Step-size constant, usually } 0 < \mu_0 < 2 \)

\( \varepsilon = \text{Small value included to avoid division by zero} \)

\( x^T x = \text{Scaling factor} \)

Equation 2-8: New step-size scalar

2.9.3 Filtered and Leaky Least Mean Square

The LMS-based algorithms discussed earlier have a large drawback when it comes to acoustic echo cancelling. The original LMS-algorithms are based on the premise that the input values are uncorrelated, i.e. that the input’s autocorrelation-matrix’ eigenvalues have a consistent size. The drawback being that speech is highly autocorrelated, this is usually referred to as an eigenvalue-spread problem\(^1\). Filtered least mean square improves the adaption speed of the least-mean-square algorithm, including the normalized version, by prewhitening of the insignal and error signal. Prewhiteness is in practice decorrelation, i.e. it reduce the eigenvalue spread\(^2\). [6],[10] The leaky least-mean-square algorithm include a leakage factor in the coefficient update, i.e. equation 2-7, forcing coefficients without continuous excitation to slowly revert back to zero. This forced zero reversion is used to avoid instability, or similarly overflow, due to poor excitation due to either a small insignal or use of limited precision. Use of a leakage factor can be shown to have the same effect as the use of dither, i.e. white noise, on the insignal. The use of dither or leakage factor is also related to the insignal’s autocorrela-

---

1. For a more in-detail analysis of eigenvalue spread, see, e.g., appendix E in [10].
2. More information regarding prewhitening is available in, e.g., [14].
tion-matrix’ eigenvalues, as the difference between the largest and smallest eigenvalue, i.e. eigenvalue-spread problem, is proportional to the difference between the largest and smallest value of the signal’s power spectral density. With this relation, an important observation is that the addition of white noise flattens the power spectral density, in effect reducing the eigenvalue-spread problem. [10],[11],[12],[13],[19]

2.9.4 Block and Subband AEC
The LMS-based algorithms do the AEC in the time-domain, i.e. sample by sample. A benefit of this is that the introduced time delay is kept at a minimum. Since the cancelling is performed in the time domain, the entire frequency domain is processed, this is the reason it is categorized as fullband processing. When time delay is not critical, block processing is a viable alternative that reduce the number of necessary computations relative an ordinary LMS-based algorithm. The reduction in number of computations stems from that adaption, e.g. coefficient update, is only performed once for a block of samples, usually in the frequency domain. As, usually, a block of samples is collected before a block of output samples is computed, this introduce a delay of a proportional size. Taking block processing a step further is the use of subband processing, where the processing is performed in the frequency domain with the addition that the frequency domain is divided in subbands. Each subband, i.e. a range of frequencies, is then adapted independently. This enables the ability to use different amount of processing power depending on each subband’s importance. [16]

2.9.5 Double Talk Detection
One of the most important parts of any least-mean-square algorithm is the double-talk detector. This detector is needed to be able to switch off, or at least slow down, adaption when near end is speaking. The reason being that the algorithm should not adapt to near-end speech, in order to avoid cancelling the desired speech signal instead of the echo and reverberations from far end. The complex problem of detecting speech, while at the same time avoiding false positives from e.g. background music, have led to the availability of several algorithms to choose from1. A commonly used algorithm, due to its implementation simplicity, is the Geigel double-talk detector. The Geigel double-talk detector compare the near-end signal with a threshold. If the near-end signal level is larger than the threshold, double talk is declared, see equation 2-9 below. A large drawback of the Geigel double-talk detector is its reliance on an accurate threshold level. The threshold is used to compensate for the ERL of the echo path. Due to the dynamic echo path2 experienced in an acoustic echo canceller implementation, determining a threshold level is a complex task. [15], [16]

\[
\frac{|d(n)|}{\max \{ |x(n-1)|, \ldots, |x(n-N)| \}} > T
\]

Equation 2-9: Geigel double-talk detector

1. For a good comparison of a selection of double-talk detectors, see [15].
2. Speaker and or loudspeaker and microphone moving, a door opened, all drastically modify the echo path. See section “6.1 Echo Path” on page 33.
Chapter 3: Method and Tools

This chapter contains information about how the practical part of the thesis was performed. This include sections describing the work method, tools and platforms used.

3.1 Sequence of Work Description
To enable a good quality of work and increasing productivity, a detailed sequence of work is very helpful. In figure ”3-1 Overview of thesis work” on page 22, a flowchart of how the work was performed is presented. As seen in this figure the thesis can be divided into five stages, where the first was a pre-study phase, here time was spent on acquiring an understanding of the problem domain. In the three following stages the initial reference code of the AEC algorithm and the architecture, i.e. assembler and instructions-set simulator, were gradually transformed to finally become an assembly-code version of relatively high performance running on the improved architecture. The fifth stage consisted of development of an online real-time test environment, enabling real-time AEC filtering of live “phone-calls”.

1. The sixth stage of presentation and related documentation is not included in the figure. See also section “5.2 Four Versions” on page 30.
2. In the current setup a “phone-call” is communication between two, network connected, computers where each have a microphone and loudspeaker.
3.2 Tools
To be able to complete this thesis a selection of tools were used throughout the thesis work.

3.2.1 Matlab
Without Matlab [21] this thesis work would probably have taken a lot longer and required a lot more work. Functionality such as the ability to easily and quickly be able to calculate, plot and in other ways work with data was really helpful. Wave to and from PCM conversion, ERLE calculations and PSD plots, auto and cross-correlation, vector manipulations are just some examples of tasks that Matlab was used to perform. Matlab was also used to enable comparisons between the AEC algorithm's original C++ implementation and the various new floating point and architecture versions.
3.2.2 GNU Compiler Collection

Another tool that was used extensively throughout the project was the GNU compiler collection, or as it is more often referred to, GCC [22]. As both the instruction-set simulator, see following section, and the original AEC code was implemented in C and C++, GNU’s easy to use C and C++ development kit was very helpful.

3.2.3 Instruction Set Simulator

A floating-point architecture instruction-set simulator was available at the start of this thesis, see [9]. The instruction-set simulator was written in C and had a straightforward implementation that enabled easy modification of the available instructions and the internal and external floating-point format. The simulator, with a few modifications, also gave a good opportunity to test modifications to other parts of the architecture, such as memory access and instruction availability. [9]

3.2.4 Eclipse and Java

In the final part of the thesis, the development of a live acoustic echo canceller test environment, Eclipse [23] was used extensively to develop the Java [24] high-level architecture including helper classes to read-from and write-to microphone and loudspeaker respectively. This architecture also provides a framework that enables network communication between two computers, simulating a telephony call.
Chapter 4: Architecture Improvements

This chapter details the modifications of the original architecture, see section “2.6 Original Floating Point Architecture” on page 10.

Seen above, in figure ”4-1 High level view of the new architecture”, is an overview of the new architecture. The architecture is of Harvard type, buses without specified size are 16 bits wide\(^1\). The load-store like and limited instruction set provided implies that the architecture should be categorized as a RISC. The instructions set, available in “Appendix A: Instruction Set Reference” - instruction encoding in “Appendix B: Instruction Encoding”, contain a limited amount of branch instructions and,

---

1. Notice that in this thesis the number of bits used in the various parts of the architecture, including bus size, was varied during analysis of quantization effects.
with a few select exceptions mentioned below, operate on immediate values or registers. The instruction set is not orthogonal as, with the same exceptions as above, e.g. only instructions with a specific suffix enable modulo addressing. Compared to the previous version, figure ”2-2 Original architecture” on page 11, the new architecture enables two concurrent reads from or writes to data memory. Not seen in the figure above is that the new architecture supports; zero-overhead looping, modulo addressing - with optional variable step-size pre or post increment or decrement - for a select set of instructions, floating-point MAC operations - described in section “4.3 MAC Support” below. Indicated by use of a spotted frame is that the current architecture lacks both coefficient memory and accelerated sequential bit access compared to the original architecture. A few noteworthy shortcomings compared to a competitive DSP architecture are that it currently; lacks support for interrupts and user-program load and unload - i.e. the architecture is offline programmable only, provides no caches - i.e. limited by memory-access time, has no explicit support for efficient Viterbi decoding.1

4.1 Registers

Compared to the original architecture, the current architecture provides 4 accumulator registers with 16 bits mantissa and 7 bits exponent. These accumulator registers are used by the MAC instructions, fmul, fdiv2, fadd, fld and fst.

4.2 Pipeline

As seen in figure ”4-2 Six stage pipeline” below, fixed-point instructions require 4 pipeline steps to complete, while floating-point instructions require 6 steps. With the aid of register-write forwarding, fixed-point, and logic floating-point, instructions have zero overhead3, while floating-point instructions require 2 instructions delay. The original pipeline is found in figure ”2-3 Original pipeline” on page 12.

---

1. See theory section “2.5 Floating Point Architecture” on page 10 and section “2.6 Original Floating Point Architecture” on page 10.
2. Notice that fdiv is implemented as exponent subtraction, i.e. actual division is currently not implemented.
3. Zero overhead here refers to that no delay elements needs to be inserted between consecutive instructions.
4.3 MAC Support
In order to support efficient implementation of, e.g., digital filters a multiply and accumulate\(^1\) unit is in practice required to exist in a DSP architecture. Provided by the architecture is two single-instruction\(^2\) multiply and accumulate instructions; \texttt{fmac} and \texttt{fmaci}. The first operates on two memory values and the latter on one memory value and one immediate value. The result is stored in an accumulator register.

4.4 Addressing Modes
To support circular buffers, used to avoid excessive copying of data in memory, modulo addressing is the prevailing solution. Implementing circular buffers with indexes and branch statements is generally somewhat cumbersome and can in worst case dramatically decrease maximum performance, an example of the latter is provided in section “4.5 Zero Overhead Loop Support” below. This architecture not only supports modulo addressing, but also enables\(^3\) bit-reversed addressing. Bit-reversed addressing is predominantly used to support high-performance implementations of FFT algorithms. Both modulo and bit-reversed addressing is implemented by the use of special registers; one to decide the register that should be affected, step size and if the addressing mode should be bit reversed or modulo and two that specify start and end address respectively. Both addressing modes are only supported on a few select instructions; MAC-instructions and instructions with a “\texttt{m}” suffix. Another feature of the architecture is that load and store operations with a “\texttt{p}” suffix automatically post increments their address-register value. An example of the performance impact of register-address increment and modular addressing is available in section “Modular Addressing” in appendix D.

4.5 Zero Overhead Loop Support
Most use cases of a DSP require tight loops, where tight here refers to when overall performance can not tolerate the overhead incurred by branch instructions and counter modification. Zero-overhead loops enables efficient execution of, e.g., a single-instruction loop that comprise a MAC operation, optionally with modulo addressing, applied to large vectors of data. Currently the architecture’s support for zero-overhead looping is limited to either single-instruction loops, \texttt{loop}, or block loops, \texttt{blk}. There is currently no support for nesting of zero-overhead loops. Both loop constructs are limited to 65535 repetitions\(^4\) and the block loop supports large blocks of instructions, as the current implementation loops from the address following the \texttt{blk} instruction to a user specified end address. The introduction of zero-overhead loop support reduced the real-time MIPS requirement of this thesis implementation of an AEC algorithm, see section “5.5 Assembly Implementation” on page 31 and section “Zero Overhead Loop” in appendix D.

---

1. See section “2.2.1 Multiply and Accumulate” on page 7.
2. Single instruction here refers to that despite the individual MAC operations will still take 6 pipeline stages to complete, they can be scheduled with zero overhead.
3. I.e., currently not implemented.
4. Two to the power of the number of fixed-point bits available, currently 16.
4.6 Branch Support
Support for branching is somewhat limited in the current architecture, as there is only 4 conditional branch operations available; beqz, bnez, bibc and bibs. All four branches if a condition is true to a user specified label. Beqz and bnez branch if a register value is zero or not respectively. The latter two branches if bits specified in a bit mask, in a special register, are either all zero or all one in a specified register. Other than conditional branches, the architecture has support for; jumping to a specified address, calling a subroutine and later returning to the previous point of execution. There is currently no support for context switching, i.e. store and restore register values when calling or returning from a subroutine.

4.7 Instruction Encoding
While the architecture is designed to be a combined fixed and floating-point architecture, currently the main focus is floating-point instructions. This is easily recognized as only very basic arithmetic and logic operations have fixed-point versions of their corresponding floating-point instructions. A design decisions specify that floating-point instructions are encoded with a leading “11” in their bit encoding, i.e. the rest of the instruction code match that of their corresponding, if existing, fixed-point instruction. Most notable is that arithmetic and logic operations have their four most significant bits encoded “1111” and “0111” for floating and fixed-point instructions respectively. This encoding provides the opportunity for the hardware to easily recognize modules and units needed to perform operations on a given instruction.
Chapter 5: Implementation

This chapter contains information about the various implementations of the AEC algorithm developed in the course of this thesis.

5.1 Matlab

Matlab scripts were developed and incrementally improved. These scripts were used as an alternative solution to test the AEC functionality until the live test environment was completed. The list and figure 5-1 below details how the scripts were used.

- The original C++-code was tested with the help of Matlab. Sound files were created and stored as Wave-encoded files. Matlab scripts\(^1\) were created to facilitate the testing procedure, the scripts only required two Wave-encoded sound files as parameters. One representing the far-end, the other the near-end speech, and a parameter giving the directory where the C++, later “hybrid-C++” and assembly, code to run was located.
- The scripts transformed the speech files into a format required by the AEC implementations, adding the far-end speech as echo to the near end and saved it to the directory containing the code that should be tested.
- After this first step, the script then executed a Makefile in a specified directory. The Makefile contained the required operations and settings to run the code. The final result being an echo-cancelled raw PCM-encoded file.
- This file was then read by the script, transforming it back to a valid, now echo-cancelled, Wave-encoded sound file.

---

1. See “Appendix D: Code”.

---
5.2 Four Versions

The previous section gave some information about the general testing procedure, this section gives more detail regarding the different parts that were tested. The practical part of the thesis was divided into five distinct parts. The first part was implementing the AEC algorithm in C++ code, basically this step was already completed, see “Appendix E: Original AEC Implementation”. The next step in this first part was to develop a test suite in Matlab to facilitate the testing and evaluation procedure. The fact that all scripts were created and run from within Matlab simplified the task of evaluating different settings and results. The second part was the transformation of the original C++ code to a floating-point hybrid. This hybrid used arithmetic instructions and the floating-point data format defined in the instruction-set simulator and assembler provided by [9], see section “2.6 Original Floating Point Architecture” on page 10. The C++ code was transformed by changing arithmetic operations, addition, subtraction, multiplication et cetera, to equivalent floating-point operations implemented in the simulator. The third part was implementation of assembly code the DSP architecture could understand. This step was quite complex and involved study of the provided architecture and various modifications, the original architecture is available in figure ”2-2 Original architecture” on page 11 and the modified architecture in figure ”2-1 High level view of new architecture” on page 5. Modification and extension of the provided architecture, i.e. the instruction-set simulator and assembler, and implementation of an optimized version of the assembly code was the fourth practical part of this thesis. The final part consisted of; development of a live test environment, an evaluation of how quantization affect the selected AEC algorithm and the difference between fixed and floating-point quantization effects on speech and sound in general.
5.3 Double Talk Detection

In the current implementation a Geigel double-talk detector\(^1\) is used. The double-talk detector is working adequately for the echo path environment used in the experiments, the threshold value has been tuned to be able to detect double talk. To simplify the evaluation, only single talk is used in the experiments in “Chapter 6: Quantization” on page 33.

5.4 Prewhitening

The original C++ code\(^2\), use a form of prewhitening\(^3\) where the far-end speech and estimated-error signal is highpass filtered before adaption, i.e. coefficient update, is performed. Prewhitening is disabled in the experiments performed in “Chapter 6: Quantization” on page 33.

5.5 Assembly Implementation

With the above mentioned architecture an assembly implementation was developed with a real-time requirement, in a worst-case scenario, of 15 MIPS when using a dedicated hardware instruction enabling efficient update of coefficient values. Without this extra support for coefficient update the MIPS requirement increases drastically to 35 or 47 MIPS depending on implementation. The former requires a modified MAC-instruction that accumulates to an ordinary, i.e. non-accumulator, register. The large increase in required MIPS is explained by the increase of instructions involved in the coefficient update; the dedicated instruction requires two\(^4\), the modified MAC instruction require seven\(^5\) and multiplication followed by addition require ten\(^6\) instructions. The coefficient update is performed 8000 times per second on 512 coefficients, i.e. \(8000 \cdot 512 \cdot n \approx 4 \text{ MIPS} \cdot n\), where \(n\) is two, seven or ten. An observation based on the above, is that the MIPS requirement of the selected AEC algorithm running on this architecture without the use of hardware accelerated loops and modular addressing becomes relatively large, i.e. more than 100 MIPS. Another observation is that this large dependence on efficient coefficient modification explains the popularity of, e.g., block-update LMS-based and subband AEC algorithms\(^7\). The entire assembly implementation, including prewhitening filters and double-talk detector, use 1485 bytes of program memory and 1609 words of data memory. Here word equals external size, i.e. the data-memory requirement depends on the external format. Using six bits exponent with eight bits mantissa or 16 bits mantissa use 2816 bytes or 4425 bytes respectively.

---

1. See section “2.9.5 Double Talk Detection” on page 19.
2. See “Appendix E: Original AEC Implementation”.
3. See section “2.9.3 Filtered and Leaky Least Mean Square” on page 18.
4. Only two combined memory reads or writes per cycle. The dedicated instruction require three, two reads and one write, this requires a single \texttt{nop} instruction following every dedicated instruction.
5. One load, one modified MAC, one store and 4 \texttt{nop} instructions. See section “4.2 Pipeline”.
6. One load, one multiplication, one addition, one store and 6 \texttt{nop} instructions. See section “4.2 Pipeline”.
7. See section “2.9.4 Block and Subband AEC” on page 19.
Chapter 6: Quantization

This chapter contains an evaluation of the effects of quantization noise on the chosen AEC algorithm. In the sections below, a six bit exponent is used in floating-point format if not otherwise specified.

6.1 Echo Path

Figure 6-1 below show an approximative echo path of the room where the following sections experiments were performed. Care was taken to avoid large modifications of the echo path between the measurements, e.g. loudspeaker and microphone placement was stationary and doors remained closed. The echo path was estimated by the use of correlation of an original maximum-length sequence\(^1\) with a recording of its playback. The microphone was placed behind the loudspeakers in all experiments, i.e. there was no line of sight between microphone and loudspeaker.

![Figure 6-1: Echo-path impulse and frequency response](image)

\(^1\) See section “Generation of Maximum Length Sequence” in appendix D and, e.g., [16].
6.2 AEC Performance

All ERLE plots below are smoothed according to equation "6-1 Lowpass filter".

\[ x_n[i] = \alpha \cdot x_n[i-1] + (1 - \alpha) \cdot x[i^2] \]
\[ \alpha = 0.98, \quad x = [e, y] \]

**Equation 6-1: Lowpass filter**

Seen below, in figure "6-2 ERLE with 16 internal, external and accumulator mantissa bits”, is the base white-noise input\(^1\) performance metric that the following internal and external mantissa-bits settings are compared with. As seen in the figure an ERLE of approximately 20 dB is achieved. With an ERL of 14 dB\(^2\), the far-end echo is attenuated almost 35 dB\(^3\) without the use of non-linear processing\(^4\). The relevance of the attenuation’s magnitude, as far as this thesis is concerned, is not of significant importance. This magnitude is only a performance metric that the attenuation achievable when using less internal and or external mantissa bits\(^5\) is compared with.

---

1. As seen in the input signal spectrum, the input signal is not pure white noise. The low frequency high-energy parts are present as the insignal is recorded ambient-noise with white noise added synthetically.
2. This might seem like a high value for ERL, but is explained by loudspeaker placement and poor microphone sensitivity. To improve realism of the experiments, the loudspeakers were placed to not directly face the microphone.
3. Notice the use of a white-noise input signal, i.e. speech is a lot more difficult to cancel.
4. NLP is usually implemented as a multiplication with a relatively small constant to reduce the output when the near-end speaker is silent.
5. Only the number of mantissa bits used is being varied, as only these limit the precision, given that no underflow occurs during execution of the algorithm. See section “2.3 Fixed and Floating Point Format” on page 7.
6.2.1 Accumulator Mantissa Bits

Seen below in figure "6-3 Achievable ERLE, variable accumulator mantissa bits" is the maximum achievable ERLE with the white-noise insignal. The left and right figure indicates the effects of the number of accumulator bits when internal and external formats are using 16 and 12 mantissa bits respectively. As evident by the graphs below, the number of accumulator mantissa bits have a lower bound of 15 bits.

![Figure 6-3: Achievable ERLE, variable accumulator mantissa bits](image)

6.2.2 Internal and External Mantissa Bits

Opposite to the previous sections description of how the AEC algorithm was affected by variation in the number of accumulator, with constant internal and external, mantissa bits this section instead describes the behavior of using a fixed number of accumulator mantissa bits, here 16. As evident in figure "6-4 Achievable ERLE, variable internal and external mantissa bits", with less than eleven or twelve mantissa bits, there is a relatively steep linear decrease of performance.

![Figure 6-4: Achievable ERLE, variable internal and external mantissa bits](image)
6.2.3 External Mantissa Bits

Similar to the previous section, the number of accumulator mantissa bits was kept constant in this experiment. Instead of decreasing both internal and external number of mantissa bits, here only the external format was modified. With 16 internal and accumulator mantissa bits, as few as 8 external mantissa bits maintains an almost identical performance compared to the base metric with the selected white-noise input signal, see figure "6-5 Achievable ERLE, variable external mantissa bits".

Figure 6-5: Achievable ERLE, variable external mantissa bits

6.3 Bit Length Dependent Sound Quality Evaluation

To test how the floating-point format relates to fixed point in the area of sound quality, a simple and obvious test is to actually listen to sound files encoded by the different formats. To use listening tests scientifically the test would of course need to be a lot more specific and detailed, not to mention performed by a diverse and statistically large enough sample of test persons. Though if listening only are used to get a coarse overview of the relative performance of different bit depths of fixed versus floating point, a simple test case with only a very small amount of test persons was deemed satisfactory. The listening test was set up to compare floating and fixed-point versions of 8 kHz single-channel PCM-encoded sound files. The modification was quite straightforward, for fixed point the sound files were truncated to a limited number of significant bits, 5-10 bits\(^1\). In the floating-point case, only the number of mantissa bits were limited, 4-6 bits, the number exponent bits were kept large enough to represent PCM-encoded files without saturation, 5 bits, i.e. able to represent integer values in range -65536 to 65535. In the informal and approximative audio-quality test evaluation, the floating-point version with as few as 5 mantissa bits performed equal or better to the fixed-point version using 8-10 bits.

\(^1\) Sign bit not counted in neither fixed nor floating-point format.
6.4 Quantization Noise

In the previous section a comparison of floating and fixed-point format was performed by a practical listening test. In this section the difference in quantization-related errors is instead analyzed by use of sound-to-noise ratio and power-spectral-density figures. Shown in the first two figures below, figure 6-6 and figure 6-7, is the quantization noise of a sinusoid signal and a short speech segment.

![Figure 6-6: Sinusoid quantization noise](image1)

The figures above show the effect explained in the theory, that the floating-point format’s quantization noise is dependent on the values size. Another observation is that for a periodic signal the sound-to-noise ratio of fixed and floating-point format is approximately equal given an equal amount of precision related bits. For the scaled speech version the SNR of five floating-point mantissa bits is approximately equal to seven fixed-point bits. More silent or low volume parts, with a few high

![Figure 6-7: Speech quantization noise](image2)
volume peaks to require a relatively high dynamic range, implies that the floating-point format's SNR become increasingly better compared to the SNR of a fixed-point version. Seen below in figure 6-8 and figure 6-9, is another view of how the quantization noise of a floating-point format is colored, i.e. related to the signal amplitude, and the quantization noise of a fixed-point format is relatively uncorrelated with the insignal. Also visible in the power spectral density of speech is that the main part of energy is present in the low frequency range of approximately less than 1 kHz. The flat PSD of the fixed-point quantization noise, visible in both figures, resembles the effect of an increased noise floor\(^1\). This increase of the noise floor can be seen as an effect where small amplitude parts of a signal, e.g. unvoiced or silent parts of speech, is partly or entirely masked\(^2\) while high amplitude sections, e.g. voiced, are less affected. The floating-point format have the opposite effect where small amplitudes yields low quantization noise and consequently are less affected, i.e. not masked, and high amplitudes have high quantization noise and are more affected. With the earlier mentioned quantization effects, the trade off is between the fixed-point format's low effect on voiced parts of speech and floating-point format's low effect on unvoiced parts. Another interesting aspect is that the floating-point format's quantization noise have its largest effect on high amplitude signals, while at the same time these same high-amplitude signals are most likely to mask sounds in higher, and to some extent lower, frequencies. As stated earlier, the highest amplitude parts of speech is voiced, or periodic, and as seen in figure 6-9, a periodic signal have a large part of its quantization-noise energy in its harmonics. A conclusion is that the floating-point format in theory has an advantage for unvoiced parts of speech compared to a fixed-point format, while the floating-point format's disadvantage for voiced parts is likely to be reduced by the effects of masking, i.e. the quantization noise in the harmonics are partly or entirely masked. It is important to remember that even though the resolution of a floating-point format is higher than for a comparable fixed-point format for low-amplitude parts of a signal, in practice these low-amplitude parts are relatively unlikely to be affected by quantization effects in a fixed-point architecture. E.g. low-amplitude values are less likely to overflow the number of significant bits after a multiplication, i.e. truncation or rounding is not necessary and consequently no quantization noise is introduced.

---

1. In practice this scale of sound would imply a very loud speech segment.
2. An important notice is that fixed-point quantization noise is related to the signal level, compared to floating-point quantization noise the relation is relatively weak. See, e.g., [20].

A description of masking, voiced and unvoiced speech is available in, e.g., section “2.7.1 Sound Perception” on page 14, section “2.7.2 Voiced and Unvoiced Speech”, and [16].
Figure 6-8: Power spectral density of original speech signal and quantization noise

Figure 6-9: Power spectral density of original sinusoid signal and quantization noise
Chapter 7: Evaluation of Results and Future Work

This chapter contains a summary of the work performed in this thesis, an evaluation of the results that have been achieved and a section detailing future work.

7.1 Results
This section contains a summary of the thesis achievements.

7.1.1 Floating Point C++ AEC Implementation
The C++ AEC version, with floating-point format used for representation of values and in operations, was completed without issues. This version was later used as a reference implementation to ensure that the implementation in assembly was bit-true, i.e. identical.

7.1.2 Bit True Assembly AEC Implementation
The bit-true assembly implementation including highpass filters and double-talk detector is capable of real-time AEC requiring 15 MIPS. The MIPS requirement was achieved in a worst-case scenario, i.e. far-end single talk.

7.1.3 Architecture Improvements
The improved architecture, i.e. the instruction-set simulator and assembler, enables zero-overhead loops, modular addressing, register increment or decrement on load and store and register-write forwarding. With these improvements the assembly AEC implementation mentioned above had its real-time MIPS requirement reduced by more than 80%. The instruction-set encoding was modified to have a consistent encoding with regard to floating and fixed-point arithmetic and logic instructions.

7.1.4 Live Test Implementation
A live demo and testing environment was developed, i.e. a form of “IP-phone”-solution. This test implementation features ability to connect, over TCP/IP, two computers with a microphone and loudspeaker each and test the AEC functionality live.
7.1.5 Quantization
The effect of mantissa-size dependent quantization noise on an AEC algorithm was analyzed. A solution with as few as 8 external mantissa bits was found to have a negligible effect on the selected AEC implementation’s performance. The effect of quantization noise on both fixed and floating-point format was investigated. For high-amplitude sounds and particularly voiced speech, even though the fixed-point format has very low quantization noise for these signals relative the floating-point format's large quantization noise, masking reduce the audibility of quantization noise in the presence of high amplitude sounds. For low-amplitude sounds and unvoiced speech, the floating-point format has better resolution relative a comparable fixed-point format.

7.2 Conclusions
Floating-point architectures relative comparable fixed-point architectures in general require a large area due to more complex implementation of arithmetic instructions. The complex implementation and the increased area usually result in increased energy consumption. To justify this increase in energy consumption, a relatively large reduction of required memory size and thereby memory-related energy consumption is required. The selected algorithm and its original implementation use a small amount of memory, less than 6 kB. Due to this low memory requirement of the algorithm, the use of a small external format has a minor effect on required memory size. In total this indicate that the reduction of the memory requirement for the selected algorithm does not justify the added complexity and energy consumption of using a floating-point architecture. The floating-point format was deemed to perform good in a LMS-based AEC due to; large dynamic range, good performance and resolution for unvoiced speech segments while not suffering from a large noticeable effect of quantization noise from high-amplitude voiced segments thanks to masking\textsuperscript{1}. The benefit of using a floating-point format is not as evident in, e.g., subband processing, where each subband can be scaled separately to use the entire dynamic range and resolution of a fixed-point format. Use of a floating-point format can still be advantageous in speech-related signal processing when, e.g., a subband’s, or similar frequency domain solution, introduction of time-delay is unacceptable. Speech algorithms that have high memory use and small introduced delay requirements are a good candidate for a floating-point digital signal processor architecture. In total the thesis have fulfilled all of the acceptance levels in section “1.4 Acceptance Levels” on page 3.

7.3 Future Work
Continue to study how masking affects the floating-point format’s quantization noise, especially for high-amplitude speech segments. An evaluation of an algorithm that use a lot of data memory would be an interesting choice for a continued comparison of fixed and floating-point formats. The instruction-set simulator and the architecture can be further improved, e.g. implement division, bit-reversed addressing and hardware interrupts. The implementation of the NLMS algorithm can be improved to support features that are required in a practical and commercial AEC. This include better support for detecting, and fast adaption following, echo-path changes and ability to cope with, e.g., call-waiting signals. As this thesis test procedure used physically separate microphone and loudspeaker units, the effect of using a floating-point format on “near echo” has not been investigated. “Near echo” is here defined as echo from an echo-path in the plastic, or similar, surrounding and connecting microphone and loudspeaker.

\textsuperscript{1} Masking effects on voiced speech segments require further study, see section “7.3 Future Work”.


Bibliography


ISBN: Not Available

ISBN: 0470841370

ISBN: 1558607986

ISBN: Not Available


ISBN: 0130901261

ISBN: 0471492876

ISBN: 0824705637


ISBN: 3540000518


[18] International engineering consortium, “Echo cancellation”, IEC, web resource: 

ISBN: 0070047774

ISBN: 0672226340


[25] Maximum length sequences, web resource:  

[26], A. Adrian, “Voice over Internet Acoustic Echo Cancellation”, web resource:  
http://home.arcor.de/andreadrian/echo_cancel/draft-aec-03.txt, April 2007
Abbreviations

e.g. Exempli gratia, for example
i.e. Id est, that is
LiTH Linköpings tekniska högskola
AEC Acoustic echo cancellation
GPS Global positioning system
DECT Digital enhanced cordless telecommunications
ADSL Asymmetric digital subscriber line
VDSL Very high speed digital subscriber line
WLAN Wireless local area network
MIPS Million instructions per second
DSP Digital signal processing
DSP Digital signal processor
FFT Fast fourier transform
MAC Multiply accumulate
FIR Finite impulse response
IIR Infinite impulse response
dB Decibel
ISA Instruction set architecture
RISC Reduced instruction set computer
CISC Complex instruction set computer
PCM Pulse code modulation
SPL Sound pressure level
DR Dynamic range
SNR Sound to noise ratio
LMS Least mean square
NLMS Normalized least mean square
ERLE Echo return loss enhancement
ERL Echo return loss
NLP Non linear processing
TCP/IP Transmission control protocol and internet protocol respectively
Appendix A: Instruction Set Reference

This appendix contains an instruction-set reference. Most operations are equivalent to those used in the original architecture, see [9].

A.1 Notation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rA, rB</td>
<td>Source registers.</td>
</tr>
<tr>
<td>rD</td>
<td>Destination register.</td>
</tr>
<tr>
<td>sr[n]</td>
<td>Special register, named index n.</td>
</tr>
<tr>
<td>immX</td>
<td>Immediate data, X bits are specified inline in the assembly.</td>
</tr>
<tr>
<td>signextend(immX)</td>
<td>Extend immX to 16 bits, fill the most-significant bits with immX's sign bit.</td>
</tr>
<tr>
<td>zeropad(rX)</td>
<td>Extend register rX to 22 bits, fill the most-significant bits with zero.</td>
</tr>
<tr>
<td>PC</td>
<td>Program counter.</td>
</tr>
<tr>
<td>push(X)</td>
<td>The value X is pushed to the stack.</td>
</tr>
<tr>
<td>fp(rX)</td>
<td>Indicates that the register rX is expected to contain a floating-point value. Both low and high parts of the register are affected.</td>
</tr>
<tr>
<td>AR</td>
<td>Accumulator register.</td>
</tr>
<tr>
<td>loadfp(rX)</td>
<td>Loads an external floating-point value from memory address rX and converts the value to the internal floating-point format.</td>
</tr>
<tr>
<td>storefp(rX)</td>
<td>Stores an external floating-point value at memory address rX.</td>
</tr>
<tr>
<td>modulo(rX)</td>
<td>Value is modified according to modulo settings in the modulo-related special registers. See addm for an example.</td>
</tr>
<tr>
<td>store(X)</td>
<td>Stores a 16-bit value in memory at address X.</td>
</tr>
<tr>
<td>load(X)</td>
<td>Loads a 16-bit value from memory at address X.</td>
</tr>
</tbody>
</table>

Table A-1: Notation used to describe instructions

A.2 Instructions

abort

Abort simulation

The instruction-set simulator will abort simulation when this instruction is encountered.
add  
\[ rA + rB \rightarrow rD \]
16-bit fixed-point addition. Only low part of registers is affected.

addm  
\[ rA + rB \rightarrow rD \]
\[ \text{if } rD = \text{sr}[\text{MODULO}_1/2] \]
\[ \text{if } rD > \text{sr}[\text{MODULO}_\text{END}_1/2] \]
\[ \text{sr}[\text{MODULO}_\text{START}_1/2] \rightarrow rD \]
\[ \text{else if } rD < \text{sr}[\text{MODULO}_\text{START}_1/2] \]
\[ \text{sr}[\text{MODULO}_\text{END}_1/2] \rightarrow rD \]
16-bit fixed-point addition, with modulo: \[ \text{MODULO}_\text{START}_1/2 \leq rD \leq \text{MODULO}_\text{END}_1/2 \]
Only low part of registers is affected.

addm  
\[ \text{signextend} (\text{imm}8) + rB \rightarrow rD \]
\[ \text{if } rD = \text{sr}[\text{MODULO}_1/2] \]
\[ \text{if } rD > \text{sr}[\text{MODULO}_\text{END}_1/2] \]
\[ \text{sr}[\text{MODULO}_\text{START}_1/2] \rightarrow rD \]
\[ \text{else if } rD < \text{sr}[\text{MODULO}_\text{START}_1/2] \]
\[ \text{sr}[\text{MODULO}_\text{END}_1/2] \rightarrow rD \]
16-bit fixed-point addition, with modulo: \[ \text{MODULO}_\text{START}_1/2 \leq rD \leq \text{MODULO}_\text{END}_1/2 \]
Immediate data. Only low part of registers is affected.

add  
\[ \text{signextend} (\text{imm}10) + rB \rightarrow rD \]
16-bit fixed-point addition, immediate data. Only low part of registers is affected.

and  
\[ \text{zeropad}(rA) \& \text{zeropad}(rB) \rightarrow rD \]
16-bit bitwise and. Only low part of registers is affected.
and #imm8, rB, rD
zeropad(imm8) & zeropad(rB) --> rD

16-bit bitwise and, immediate data. Only low part of registers is affected.

beqz rA, imm16

if rA = 0
  imm16 --> PC

Conditional jump to address specified by imm16. Jump is taken if rA is zero. There is one delay slot, i.e. one instruction after the jump is executed unconditionally. The results of having a branch instruction, including ret, in the delay slot is undefined.

bibc rA, imm16

if rA & sr[BRANCH] = 0
  imm16 --> PC

Conditional jump to address specified by imm16. Jump is taken if all bits specified, in the BRANCH special register, in rA are zero. There is one delay slot, i.e. one instruction after the jump is executed unconditionally. The results of having a branch instruction, including ret, in the delay slot is undefined.

bibc rA, imm16

if rA & sr[BRANCH] = sr[BRANCH]
  imm16 --> PC

Conditional jump to address specified by imm16. Jump is taken if all bits specified, in the BRANCH special register, in rA are one. There is one delay slot, i.e. one instruction after the jump is executed unconditionally. The results of having a branch instruction, including ret, in the delay slot is undefined.

blkl rA, imm16

rA --> sr[LOOP_COUNTER]
imm16 --> sr[LOOP_END]
PC + 1 --> sr[LOOP_START]

Block loop, repeats the instructions between the next instruction and the address specified in imm16, rA times.
bnez rA, imm16
    if rA != 0
      imm16 --> PC

Conditional jump to address specified by imm16. Jump is taken if rA is not zero. There is one delay slot, i.e. one instruction after the jump is executed unconditionally. The results of having a branch instruction, including ret, in the delay slot is undefined.

bra imm16
    imm16 + PC --> PC

Unconditional jump to address specified by imm16. There is one delay slot, i.e. one instruction after the jump is executed unconditionally. The results of having a branch instruction, including ret, in the delay slot is undefined.

call rA
    push(PC + 1)
    rA --> PC

Unconditional jump to subroutine at address specified by rA. There is one delay slot, i.e. one instruction after the jump is executed unconditionally. The results of having a branch instruction, including ret, in the delay slot is undefined.

call imm16
    push(PC + 1)
    imm16 --> PC

Unconditional jump to subroutine at address specified by imm16. There is one delay slot, i.e. one instruction after the jump is executed unconditionally. The results of having a branch instruction, including ret, in the delay slot is undefined.

cp rA, rD
    rA --> rD

Copies register rA to register rD. Only low part of registers is affected.
\textbf{cp} \quad rA, srD

\hspace{1cm} rA \rightarrow srD

Copies register \textit{rA} to special register \textit{srD}. Only low part of registers is affected.

\textbf{cp} \quad srA, rD

\hspace{1cm} srA \rightarrow rD

Copies special register \textit{srA} to register \textit{rD}. Only low part of registers is affected.

\textbf{fadd} \quad rA, rB, rD

\hspace{1cm} \textit{fp(rA)} + \textit{fp(rB)} \rightarrow \textit{fp(rD)}

Floating-point addition.

\textbf{fand} \quad rA, rB, rD

\hspace{1cm} \textit{fp(rA)} \& \textit{fp(rB)} \rightarrow \textit{fp(rD)}

Bitwise and.

\textbf{fbtflyh} \quad [rA], rB, [rD]

\hspace{1cm} \text{modulo}(rA) \\
\hspace{2cm} \text{modulo}(rD) \\
\hspace{2cm} \text{loadfp}(rA) \times \text{fp}(rB) + \text{loadfp}(rD) \rightarrow \text{storefp}(rD) \\
\hspace{2cm} rA + 1 \rightarrow rA \\
\hspace{2cm} rD + 1 \rightarrow rD

Half butterfly operation. Loads two external floating-point values from address \textit{rA} and \textit{rD}, expands into internal floating-point values. Multiplies the value loaded from \textit{rA} with \textit{rB}, adds the result to the value loaded from \textit{rD} and finally stores in \textit{rD}. With modulo addressing for registers \textit{rA} and \textit{rD}. Registers \textit{rA} and \textit{rD} is post incremented.

\textbf{fcp} \quad rA, rD

\hspace{1cm} \textit{fp(rA)} \rightarrow \textit{fp(rD)}

Floating-point copy of register \textit{rA} to \textit{rD}. 
fdivl rA, AR, rD

fp(rA) / AR --> fp(rD)
Currently implemented as an exponent subtraction. I.e. exponent of the accumulator register AR is subtracted from the value in register rA. The modified exponent and register rA's original mantissa is stored in register rD.

fexp rA, rD

rA --> fp(rD)
Expands an external floating-point value in rA and stores it as a floating-point value in register rD.

fld [rA], rD

loadfp(rA) --> fp(rD)
Loads an external floating-point value from address rA, expands it to an internal floating-point value and stores it in register rD.

fld #imm16, rD

loadfp(imm16) --> fp(rD)
Loads an external floating-point value from address imm16, expands it to an internal floating-point value and stores it in register rD.

fldm [rA], rD

loadfp(rA) --> fp(rD)
modulo(rA)

Loads an external floating point value from address rA, expands it to an internal floating-point value and stores it in register rD. Register rA is post-modified according to modulo settings in relevant special registers.
fldp [rA], rD

\[
\text{loadfp}(rA) \rightarrow \text{fp}(rD) \\
rA + 1 \rightarrow rA
\]

Loads an external floating point value from address \( rA \), expands it to an internal floating-point value and stores it in register \( rD \). Register \( rA \) is post incremented.

fslr rA, rB, rD

\[
\text{fp}(rB) \gg rA \rightarrow \text{fp}(rD)
\]

Logical shift right. \( rB \) is shifted right \( rA \) bits.

fslr #imm8, rB, rD

\[
\text{fp}(rB) \gg \text{imm8} \rightarrow \text{fp}(rD)
\]

Logical shift right. \( rB \) is shifted right \( \text{imm8} \) bits.

fmac [rA], [rB], AR

\[
\text{modulo}(rA) \\
\text{modulo}(rB) \\
\text{loadfp}(rA) \ast \text{loadfp}(rB) + \text{AR} \rightarrow \text{AR} \\
rA + 1 \rightarrow rA \\
rB + 1 \rightarrow rB
\]

Loads two external floating-point values from address \( rA \) and \( rB \), expands into internal floating-point values. Multiplies the value loaded from \( rA \) with the value loaded from \( rB \), adds the result to the accumulator register \( \text{AR} \). With modulo addressing for registers \( rA \) and \( rB \). Registers \( rA \) and \( rB \) is post incremented.

fmaci [rA], rB, AR

\[
\text{modulo}(rA) \\
\text{loadfp}(rA) \ast \text{fp}(rB) + \text{AR} \rightarrow \text{AR} \\
rA + 1 \rightarrow rA
\]

Loads an external floating-point values from address \( rA \), expands into internal floating-point value. Multiplies the value loaded from \( rA \) with the value in register \( rB \), adds the result to the accumulator register \( \text{AR} \). With modulo addressing for register \( rA \). Register \( rA \) is post incremented.
fmul rA, rB, rD
  fp(rA) * fp(rB) --> fp(rD)
Floating-point multiplication.

for rA, rB, rD
  fp(rA) | fp(rB) --> fp(rD)
Floating-point bitwise or.

fpack rA, rD
  fp(rA) --> rD
Packs an internal floating-point value from rA and stores it as an external floating-point value in register rD.

frol rA, rB, rD
  fp(rB) rol rA --> fp(rD)
Floating-point value rotated left. Register rB is rotated left rA bits.

frol #imm8, rB, rD
  fp(rB) rol imm8 --> fp(rD)
Floating-point value rotated left. Register rB is rotated left imm8 bits.

fst [rD], rA
  fp(rA) --> store(rD)
Takes an internal floating-point value from register rA, packs it to an external floating-point value and stores it in memory at address rD.
fst  #imm16, rA
     fp(rA) --> store(imm16)
Takes an internal floating-point value from register rA, packs it to an external floating-point value and stores it in memory at address imm16.

fstm [rD], rA
     fp(rA) --> store(rD)
     rD + 1 --> rD
     modulo(rD)
Takes an internal floating-point value from register rA, packs it to an external floating-point value and stores it in memory at address rD. With modulo addressing for register rD. Register rD is post-incremented.

fstp [rD], rA
     fp(rA) --> store(rD)
     rD + 1 --> rD
Takes an internal floating-point value from register rA, packs it to an external floating-point value and stores it in memory at address rD. Register rD is post incremented.

fsub rA, rB, rD
     fp(rA) - fp(rB) --> fp(rD)
Floating-point subtraction.

fxor rA, rB, rD
     fp(rA) xor fp(rB) --> fp(rD)
Bitwise exclusive or.

htol rA, rD
     hi(rA) --> rD
Copies high bits from register rA to register rD.
jmp rA
    rA --> PC
Unconditional jump to address specified by rA. There is one delay slot, i.e. one instruction after the jump is executed unconditionally. The results of having a branch instruction, including \texttt{ret}, in the delay slot is undefined.

ld [imm16],rD
    load(imm16) --> rD
Loads a 16-bit fixed-point value from address \texttt{imm16} and stores it in register rD.

ld [rA],rD
    load(rA) --> rD
Loads a 16-bit fixed-point value from address rA and stores it in register rD.

loop #imm16
    PC + 1 --> sr[\texttt{LOOP\_END}]
    PC + 1 --> sr[\texttt{LOOP\_START}]
    imm16 --> sr[\texttt{LOOP\_COUNT}]
Repeats the following instruction \texttt{imm16} times.

loop rA
    PC + 1 --> sr[\texttt{LOOP\_END}]
    PC + 1 --> sr[\texttt{LOOP\_START}]
    rA --> sr[\texttt{LOOP\_COUNT}]
Repeats the following instruction rA times.

lsr #imm8,rB,rD
    rB >> imm8 --> rD
Fixed-point value logical shift right. rB is shifted right \texttt{imm8} bits. Only low part of registers is affected.
lsr        rA,rB,rD
  rB >> rA --> rD
Fixed-point value logical shift right. rB is shifted right rA bits. Only low part of registers is affected.

ltoh       rA,rD
  (rA & 0xFF) --> hi(rD)
8 low bits from register rA copied to high bits of register rD.

nop
No operation.

or         rA,rB,rD
  rA | rB --> rD
Fixed-point bitwise or. Only low part of registers is affected.

ret
  pop --> PC
Return from subroutine. There is one delay slot, i.e. one instruction after the jump is executed unconditionally. The results of having a branch instruction, including ret, in the delay slot is undefined.

rol        #imm8,rB,rD
  rB rol imm8 --> rD
Fixed-point value rotated left. Register rB is rotated left imm8 bits. Only low part of registers is affected.
rol rA,rB,rD

rB rol rA --> rD

Fixed-point value rotated left. Register rB is rotated left rA bits. Only low part of registers is affected.

set #imm16,rD

imm16 --> rD

Register rD is set to value imm16. Only low part of register is affected.

seth #imm8,rD

imm8 --> hi(rD)

High bits of register rD is set to value imm8.

setsr #imm16,srD

imm16 --> srD

Special register rD is set to value imm16.

sub #imm10,rB,rD

imm10 - rB --> rD

Fixed-point subtraction. Only low part of registers is affected.

sub rA,rB,rD

rA - rB --> rD

Fixed-point subtraction. Only low part of registers is affected.

st [imm16],rA

rA --> store(imm16)

Stores the fixed-point value from register rA in memory at address imm16.
st [rD],rA
    rA --> store(rD)
Stores the fixed-point value from register rA in memory at address rD.

xor rA,rB,rD
    rA xor rB --> rD
Fixed-point bitwise exclusive or. Only low part of registers is affected.
## Appendix B: Instruction Encoding

| 00000111111111111111111111111111 | abort          |
| 00000000000000000000000000000000 | nop            |
| 00000001011111111111111111111111 | ret            |
| 00000000100111111111111111111111 | rA jmp rA      |
| 00000000111111111111111111111111 | rA call rA     |
| 00000010000000000000000000000000 | imm16 bra imm16|
| 00000010000000000000000000000000 | imm16 call imm16|
| 00000010000000000000000000000000 | imm16 loop imm16|
| 00001000000000000000000000000000 | rT loop rT     |
| 00100000000000000000000000000000 | blkl rS, imm16 |
| 00110000000000000000000000000000 | ld [imm16], rD |
| 00100111111111111111111111111111 | st [imm16], rS|
| 00001111111111111111111111111111 | set imm16, rD  |
| 00000000000000000000000000000000 | imm16 setsr imm16, rD |
| 00110000000000000000000000000000 | rT add rT, rS, rD|
| 00110000000000000000000000000000 | rT sub rT, rS, rD|
| 00110000000000000000000000000000 | rT adc rT, rS, rD|
| 00110000000000000000000000000000 | rT adm rT, rS, rD|
| 00000000000000000000000000000000 | rT and rT, rS, rD|
| 00000000000000000000000000000000 | rT or rT, rS, rD|
| 00000000000000000000000000000000 | rT xor rT, rS, rD|
| 00000000000000000000000000000000 | rT rol rT, rS, rD|
| 00000000000000000000000000000000 | rT lsr rT, rS, rD|
| 00000000000000000000000000000000 | imm8 seth imm8, rD |
| 00000000000000000000000000000000 | rT ld [rT], rD |
| 00000000000000000000000000000000 | rT st [rT], rS |
| 00000000000000000000000000000000 | rT ldp [rT], rD |
| 00000000000000000000000000000000 | rT stp [rT], rS |
| 00000000000000000000000000000000 | rT ldm [rT], rD |
| 00000000000000000000000000000000 | rT stm [rT], rS |
| 00000000000000000000000000000000 | rT cp rT, rD  |
| 00000000000000000000000000000000 | rT cp rT, srD |
| 00000000000000000000000000000000 | srT cp srT, rD |

<p>| Table B-1: Instruction encoding |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1</td>
<td>rD</td>
<td>. . .</td>
<td>0 0 1 1 1 0 1 1</td>
<td>rT</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>rD</td>
<td>. . .</td>
<td>0 0 1 1 1 0 0 0</td>
<td>rT</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>rD</td>
<td>rS</td>
<td>0 1</td>
<td>imm10</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>rD</td>
<td>rS</td>
<td>1 0</td>
<td>imm10</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>rD</td>
<td>rS</td>
<td>1 1 0 0</td>
<td>imm8</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>rD</td>
<td>rS</td>
<td>1 1 0 1</td>
<td>imm8</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>rD</td>
<td>rS</td>
<td>1 1 1 0</td>
<td>imm8</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>rD</td>
<td>rS</td>
<td>1 1 1 1</td>
<td>imm8</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>rS</td>
<td>imm16</td>
<td>beqz</td>
<td>rS, imm16</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>rS</td>
<td>imm16</td>
<td>bnez</td>
<td>rS, imm16</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>rS</td>
<td>imm16</td>
<td>biec</td>
<td>imm16</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>rS</td>
<td>imm16</td>
<td>bibs</td>
<td>imm16</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>. . .</td>
<td>rY</td>
<td>0 0 0 0 0 0 0 0</td>
<td>rX</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>. . .</td>
<td>rS</td>
<td>0 0 0 0 0 0 0 1</td>
<td>rX</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>rD</td>
<td>. . .</td>
<td>0 0 0 0 1 0 0 0</td>
<td>rT</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>rD</td>
<td>. . .</td>
<td>0 0 0 0 1 0 0 1</td>
<td>rT</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>rD</td>
<td>imm16</td>
<td>fadd</td>
<td>[imm16], rD</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>rS</td>
<td>imm16</td>
<td>fsub</td>
<td>[imm16], rS</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>rD</td>
<td>rS</td>
<td>0 0 0 0 0 0 0 0</td>
<td>rT</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>rD</td>
<td>rS</td>
<td>0 0 0 0 0 0 0 1</td>
<td>rT</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>rD</td>
<td>rS</td>
<td>0 0 0 0 1 0 0 0</td>
<td>rT</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>rD</td>
<td>. . .</td>
<td>0 0 0 0 1 0 1 0</td>
<td>rT</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>rD</td>
<td>rS</td>
<td>0 0 0 1 0 0 0 0</td>
<td>rT</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>rD</td>
<td>rS</td>
<td>0 0 1 0 0 0 1 0</td>
<td>rT</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>rD</td>
<td>rS</td>
<td>0 0 1 0 0 1 0 1</td>
<td>rT</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>rD</td>
<td>rS</td>
<td>0 0 1 1 1 0 0 0</td>
<td>rT</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>rD</td>
<td>rS</td>
<td>0 0 1 1 1 0 1 0</td>
<td>rT</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>rD</td>
<td>. . .</td>
<td>0 0 1 1 0 0 0 0</td>
<td>rT</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>. . .</td>
<td>rS</td>
<td>0 0 1 1 0 0 0 1</td>
<td>rT</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>rD</td>
<td>. . .</td>
<td>0 0 1 1 0 0 0 1</td>
<td>rT</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>rD</td>
<td>. . .</td>
<td>0 0 1 1 0 0 1 1</td>
<td>rT</td>
</tr>
</tbody>
</table>

Table B-1: Instruction encoding
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
<th>Table B-1: Instruction encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>rD</td>
<td>1 1 1 1</td>
<td>rD . . . 0 0 1 1 0 1 0 0 rT fldm [rT], rD</td>
</tr>
<tr>
<td>rS</td>
<td>1 1 1 1</td>
<td>rS 0 0 1 1 0 1 0 1 rT fstm [rT], rS</td>
</tr>
<tr>
<td>rD</td>
<td>1 1 1 1</td>
<td>rD . . . 0 0 1 1 1 0 0 0 rT fcp rT, rD</td>
</tr>
<tr>
<td>rA</td>
<td>1 1 1 1</td>
<td>rA . . . 0 0 1 1 1 0 0 1 rT fcp rT, rA</td>
</tr>
<tr>
<td>rD</td>
<td>1 1 1 1</td>
<td>rD . . . 0 0 1 1 1 0 1 0 rA fcp rA, rD</td>
</tr>
<tr>
<td>rD</td>
<td>1 1 1 1</td>
<td>rD . . . 0 0 1 1 1 1 0 0 rA fcp rA, rD</td>
</tr>
<tr>
<td>rS</td>
<td>1 1 1 1</td>
<td>rS 0 0 1 1 1 1 0 1 rA fst rA, rS</td>
</tr>
<tr>
<td>rD</td>
<td>1 1 1 1</td>
<td>rD rS 1 1 0 0 imm8 frol imm8, rS, rD</td>
</tr>
<tr>
<td>rD</td>
<td>1 1 1 1</td>
<td>rD rS 1 1 0 1 imm8 flsr imm8, rS, rD</td>
</tr>
</tbody>
</table>

Table B-1: Instruction encoding


Appendix C: Data Path

Figure C-1: Schematic of fixed-point data path

1. Notice that in this thesis only the instruction-set simulator and assembler have been modified. I.e. the schematic above is a description of how the instruction-set simulator implementation could be realized in hardware.
Figure C-2: Schematic of floating-point data path

---

1. See previous footnote.
Appendix D: Code

D.1 Matlab

D.1.1 Generation of Maximum Length Sequence

\[ N = 15; \]
\[ M = 2^{\text{\texttt{N}}-1}; \]
\[ \text{m} = \text{zeros}(1,\text{\texttt{N}}); \]
\[ \text{m}(1) = 1; \text{m}(3) = 1; \text{m}(4) = 1; \]
\[ \text{m}(8) = 1; \text{m}(13) = 1; \text{m}(15) = 1; \]
\[ \text{regout} = \text{zeros}(1,\text{\texttt{M}}); \]
\[ \text{for} \ i = 1:\text{\texttt{M}} \]
\[ \quad \text{buf} = \text{xor}(\text{m}(14),\text{m}(15)); \]
\[ \quad \text{m}(2:\text{\texttt{N}}) = \text{m}(1:\text{\texttt{N}}-1); \]
\[ \quad \text{m}(1) = \text{buf}; \]
\[ \quad \text{regout}(i) = \text{m}(15); \]
\[ \text{end} \]
\[ \text{comp} = \sim \text{regout}; \text{sequence} = \text{regout} - \text{comp}; \]
For an explanation of the code above, please see [25].

D.2 Assembly

dotp:

\text{set } \#\text{0x0000}, r6; \quad ; r6 := "sum" = 0.0
\text{seth } \#\text{0x0040}, r6 \quad ; 16 \text{ mantissa, } 6 \text{ exponent bits}
\text{fst } ar0, r6 \quad ; ar0 := 0.0
\text{setsr } \#\text{0x0013}, sr0; \quad ; sr0 := (r3 + 1) \% 512 below
\text{setsr } \#\text{aec_w}, sr1; \quad ; sr1 := \text{begin w[0]}
\text{setsr } \#\text{aec_w +512}, sr2; \quad ; sr2 := \text{end address "w[512]"}
\text{setsr } \#\text{0x0014}, sr3; \quad ; sr3 := (r4 + 1) \% 512 below
\text{setsr } \#\text{aec_x}, sr4; \quad ; sr4 := \text{begin x[0]}
\text{setsr } \#\text{aec_x +512}, sr5; \quad ; sr5 := \text{end "x[512]"}
\text{set } \#\text{aec_w}, r3; \quad ; r3 := \text{begin w[0]}
\text{set } \#\text{aec_x}, r4; \quad ; r4 := \text{begin x[0]}
\text{ld } [\text{aec_index}], r5; \quad ; r5 := \text{"start index of x"}
\text{add } r4, r5, r4; \quad ; r4 := \text{"start at x[start index]"}
\text{nop}
\text{loop } \#\text{512} \quad ; 512 \text{ iterations}
\text{fmac } [r3],[r4], ar0 \quad \text{; Avoid register-write collision}
\text{nop}
\text{nop}
\text{fld } ar0, r6 \quad ; r6 := "sum"
\text{setsr } \#0, sr0; \quad ; \text{Reset special registers}
setsr #0,sr1
setsr #0,sr2
setsr #0,sr3
setsr #0,sr4
setsr #0,sr5
ret
nop

538 instructions required to compute dot product of two 512-elements vectors.

D.2.1 Modular Addressing

dotp:
set    #0x0000,r6; ; r6 := "sum" = 0.0
seth   #0x0040,r6 ; 16 mantissa, 6 exponent bits
fst    ar0,r6 ; ar0 := 0.0
set    #aec_w,r3; ; r3 := begin w[0]
set    #aec_x,r4; ; r4 := begin x[0]
ld    [aec_index],r5; ; r5 := "start index of x"
add    r4,r5,r4; ; r4 := "start at x[start index]"
nop
blkl   #512,dotp_loop_end ; Instructions to (incl.) "end label" rep.
fmac    [r3],[r4],ar0
nop
nop ; Avoid register-write collision
inc    r3
inc    r4
sub    #512,r4,r7 ; r7 := "zero if wrap needed" = 512 - r4
bnez   r7, dotp_loop_end
nop
set    #0,r4 ; Wrap

dotp_loop_end:
nop
fld    ar0,r6 ; r6 := "sum"
ret
nop

4621 instructions required to compute dot product of two 512-elements vectors.

D.2.2 Zero Overhead Loop

dotp:
set    #0x0000,r6; ; r6 := "sum" = 0.0
seth   #0x0040,r6 ; 16 mantissa, 6 exponent bits
fst    ar0,r6 ; ar0 := 0.0
setsr #0x0013,sr0; ; sr0 := (r3 + 1) % 512 below
setsr #aec_w,sr1; ; sr1 := begin w[0]
```assembly
setsr #aec_w +512,sr2; ; sr2 := end address "w[512]"
setsr #0x0014,sr3; ; sr3 := (r4 + 1) % 512 below
setsr #aec_x,sr4; ; sr4 := begin x[0]
ssetsr #aec_x +512,sr5; ; sr5 := end "x[512]"
set #aec_w,r3 ; r3 := begin w[0]
set #aec_x,r4 ; r4 := begin x[0]
ld [aec_index],r5; ; r5 := "start index of x"
add r4,r5,r4; ; r4 := “start at x[start index]”
set #512,r7 ; r7 := “loop counter” = 512
nop
dotp_loop:
  fmac [r3],[r4],ar0
  nop
  nop ; Avoid register-write collision
dec r7
bnez r7,dotp_loop ; Repeat, if less than 512 repetitions
nop
fld ar0,r6 ; r6 := “sum”
setsr #0,sr0; ; Clear special registers
setsr #0,sr1
setsr #0,sr2
setsr #0,sr3
setsr #0,sr4
setsr #0,sr5
ret
nop
```

2587 instructions required to compute dot product of two 512-elements vectors. The least-mean-square algorithm implemented in this master thesis contains two loops where large vectors are processed, the introduction of zero-overhead loops gave more than a 2X performance improvement, i.e. more than halved the real-time MIPS requirement.
Appendix E: Original AEC Implementation

The base of this thesis AEC implementation is the following C++ NLMS AEC implementation developed by Andre Adrian. Permission to use his implementation granted by Andre Adrian. [26]

/******************** A.1 APPENDIX aec.h *****************/

/* aec.h
*
* Copyright (C) DFS Deutsche Flugsicherung (2004). All Rights Reserved.
*
* Acoustic Echo Cancellation NLMS-pw algorithm
*
* Version 1.3 filter created with www.dsptutor.freeuk.com
*/

#ifndef _AEC_H                  /* include only once */
#include <math.h>

// use double if your CPU does software-emulation of float
typedef float REAL;

/* dB Values */
const REAL M0dB = 1.0f;
const REAL M3dB = 0.71f;
const REAL M6dB = 0.50f;
const REAL M9dB = 0.35f;
const REAL M12dB = 0.25f;
const REAL M18dB = 0.125f;
const REAL M24dB = 0.063f;

/* dB values for 16bit PCM */
/* MxdB_PCM = 32767 * 10 ^ (x / 20) */
const REAL M10dB_PCM = 10362.0f;
const REAL M20dB_PCM = 3277.0f;
const REAL M25dB_PCM = 1843.0f;
const REAL M30dB_PCM = 1026.0f;
const REAL M35dB_PCM = 583.0f;
const REAL M40dB_PCM = 328.0f;
const REAL M45dB_PCM = 184.0f;
const REAL M50dB_PCM = 104.0f;
const REAL M55dB_PCM = 58.0f;
const REAL M60dB_PCM = 33.0f;
const REAL M65dB_PCM = 18.0f;
const REAL M70dB_PCM = 10.0f;
const REAL M75dB_PCM = 6.0f;
const REAL M80dB_PCM = 3.0f;
const REAL M85dB_PCM = 2.0f;
const REAL M90dB_PCM = 1.0f;

const REAL MAXPCM = 32767.0f;

/* Design constants (Change to fine tune the algorithms */

/* The following values are for hardware AEC and studio quality */
* microphone */

/* maximum NLMS filter length in taps. A longer filter length gives
* better Echo Cancellation, but slower convergence speed and
* needs more CPU power (Order of NLMS is linear) */
#define NLMS_LEN (80*8)

/* convergence speed. Range: >0 to <1 (0.2 to 0.7). Larger values give
* more AEC in lower frequencies, but less AEC in higher frequencies. */
const REAL Stepsize = 0.7f;

/* minimum energy in xf. Range: M70dB_PCM to M50dB_PCM. Should be equal
* to microphone ambient Noise level */
const REAL Min_xf = M75dB_PCM;

/* Double Talk Detector Speaker/Microphone Threshold. Range <=1
* Large value (M0dB) is good for Single-Talk Echo cancellation,
* small value (M12dB) is good for Double-Talk AEC */
const REAL GeigelThreshold = M6dB;

/* Double Talk Detector hangover in taps. Not relevant for Single-Talk
* AEC */
const int Thold = 30 * 8;

/* for Non Linear Processor. Range >0 to 1. Large value (M0dB) is good
* for Double-Talk, small value (M12dB) is good for Single-Talk */
const REAL NLPAttenuation = M12dB;

/* Below this line there are no more design constants */

/* Exponential Smoothing or IIR Infinite Impulse Response Filter */
class IIR_HP {
    REAL x;

public:
    IIR_HP() { x = 0.0f; }

    REAL highpass(REAL in) {
        const REAL a0 = 0.01f; /* controls Transfer Frequency */
        /* Highpass = Signal - Lowpass. Lowpass = Exponential Smoothing */
        x += a0 * (in - x);
        return in - x;
    }
};

/* 13 taps FIR Finite Impulse Response filter
* Coefficients calculated with
*/
class FIR_HP13 {
    REAL z[14];

public:
FIR_HP13() { memset(this, 0, sizeof(FIR_HP13)); }

REAL highpass(REAL in) {
    const REAL a[14] = {
        // Kaiser Window FIR Filter, Filter type: High pass
        // Passband: 300.0 - 4000.0 Hz, Order: 12
        // Transition band: 100.0 Hz, Stopband attenuation: 10.0 dB
        -0.043183226f, -0.046636667f, -0.049576525f, -0.051936015f,
        -0.053661242f, -0.054712527f, 0.82598513f, -0.054712527f,
        -0.053661242f, -0.051936015f, -0.049576525f, -0.046636667f,
        -0.043183226f, 0.0f
    };
    memmove(z+1, z, 13*sizeof(REAL));
    z[0] = in;
    REAL sum0 = 0.0, sum1 = 0.0;
    int j;
    for (j = 0; j < 14; j+= 2) {
        // optimize: partial loop unrolling
        sum0 += a[j] * z[j];
        sum1 += a[j+1] * z[j+1];
    }
    return sum0+sum1;
}

class IIR1 {
    REAL x, y;

public:
    IIR1() { memset(this, 0, sizeof(IIR1)); }
    REAL highpass(REAL in) {
        // Chebyshev IIR filter, Filter type: HP
        // Passband: 3700 - 4000.0 Hz
        // Passband ripple: 1.5 dB, Order: 1
        const REAL a0 = 0.105831884f;
        const REAL a1 = -0.105831884f;
        const REAL b1 = 0.78833646f;
        REAL out = a0 * in + a1 * x + b1 * y;
        x = in;
        y = out;
        return out;
    }
};

/* Recursive two pole IIR Infinite Impulse Response filter
 * Coefficients calculated with
 */
class IIR2 {

REAL x[2], y[2];

public:

IIR2() { memset(this, 0, sizeof(IIR2)); };

REAL highpass(REAL in) {
    // Butterworth IIR filter, Filter type: HP
    // Passband: 2000 - 4000.0 Hz, Order: 2
    const REAL a[] = { 0.29289323f, -0.58578646f, 0.29289323f, 1.3107072E-16f, 0.17157288f };   
    REAL out =
    a[0] * in +
    a[1] * x[0] +
    a[2] * x[1] -
    b[0] * y[0] -
    b[1] * y[1];

    x[1] = x[0];
    x[0] = in;
    y[1] = y[0];
    y[0] = out;
    return out;
}

// Extention in taps to reduce mem copies
#define NLMS_EXT (10*8)
// block size in taps to optimize DTD calculation
#define DTD_LEN 16

class AEC {
    // Time domain Filters
    IIR_HP hp00, hp1;                   // DC-level remove Highpass)
    FIR_HP13 hp0;                       // 300Hz cut-off Highpass
    IIR1 Fx, Fe;                       // pre-whitening Highpass for x, e

    // Geigel DTD (Double Talk Detector)
    REAL max_max_x;                     // max(|x[0]|, .. |x[L-1]|)
    int hangover;
    // optimize: less calculations for max()
    REAL max_x[NLMS_LEN / DTD_LEN];
    int dtdCnt;
    int dtdNdx;

    // NLMS-pw
    REAL x[NLMS_LEN + NLMS_EXT];        // tap delayed loudspeaker signal
    REAL xf[NLMS_LEN + NLMS_EXT];       // pre-whitening tap delayed signal
    REAL w[NLMS_LEN];                   // tap weights
    int j;                             // optimize: less memory copies
    int lastupdate;                    // optimize: iterative dotp(x,x)
    double dotp_xf_xf;                // double to avoid loss of precision
    double Min_dotp_xf_xf;
    REAL s0avg;

public:
    AEC();

/* Geigel Double-Talk Detector */
* * in d: microphone sample (PCM as REALing point value)
* in x: loudspeaker sample (PCM as REALing point value)
* return: 0 for no talking, 1 for talking
*/
    int dtd(REAL d, REAL x);

/* Normalized Least Mean Square Algorithm pre-whitening (NLMS-pw) */
* The LMS algorithm was developed by Bernard Widrow
* * in mic: microphone sample (PCM as REALing point value)
* in spk: loudspeaker sample (PCM as REALing point value)
* in update: 0 for convolve only, 1 for convolve and update
* return: echo cancelled microphone sample
*/
    REAL nlms_pw(REAL mic, REAL spk, int update);

/* Acoustic Echo Cancellation and Suppression of one sample */
* in  d: microphone signal with echo
* in  x: loudspeaker signal
* return: echo cancelled microphone signal
*/
    int AEC::doAEC(int d, int x);

float AEC::getambient() {
    return s0avg;
};

void AEC::setambient(float Min_xf) {
    dotp_xf_xf = Min_dotp_xf_xf = NLMS_LEN * Min_xf * Min_xf;
};

#define _AEC_H
#endif
# A.2 APPENDIX aec.cpp

/* aec.cpp *

* Copyright (C) DFS Deutsche Flugsicherung (2004). All Rights Reserved. *
* *
* Acoustic Echo Cancellation NLMS-pw algorithm *
* *
* Version 1.3 filter created with www.dsptutor.freeuk.com *
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "aec.h"

/* Vector Dot Product */
REAL dotp(REAL a[], REAL b[]) {
    REAL sum0 = 0.0, sum1 = 0.0;
    int j;

    for (j = 0; j < NLMS_LEN; j+= 2) {
        // optimize: partial loop unrolling
        sum0 += a[j] * b[j];
        sum1 += a[j+1] * b[j+1];
    }
    return sum0+sum1;
}

AEC::AEC() {
    max_max_x = 0.0f;
    hangover = 0;
    memset(max_x, 0, sizeof(max_x));
    dtdCnt = dtdNdx = 0;

    memset(x, 0, sizeof(x));
    memset(xf, 0, sizeof(xf));
    memset(w, 0, sizeof(w));
    j = NLMS_EXT;
    lastupdate = 0;
    s0avg = M80dB_PCM;
    setambient(Min_xf);
}

REAL AEC::nlms_pw(REAL mic, REAL spk, int update) {
    REAL d = mic;       // desired signal
    x[j] = spk;
    xf[j] = Fx.highpass(spk);   // pre-whitening of x
// calculate error value
// (mic signal - estimated mic signal from spk signal)
REAL e = d - dotp(w, x + j);
REAL ef = Fe.highpass(e);  // pre-whitening of e
// optimize: iterative dotp(xf, xf)
dotp_xf_xf += (xf[j]*xf[j] - xf[j+NLMS_LEN-1]*xf[j+NLMS_LEN-1]);
if (update) {
    // calculate variable step size
    REAL mikro_ef = Stepsize * ef / dotp_xf_xf;

    // update tap weights (filter learning)
    int i;
    for (i = 0; i < NLMS_LEN; i += 2) {
        // optimize: partial loop unrolling
        w[i] += mikro_ef*xf[i+j];
        w[i+1] += mikro_ef*xf[i+j+1];
    }
}
if (--j < 0) {
    // optimize: decrease number of memory copies
    j = NLMS_EXT;
    memmove(x+j+1, x, (NLMS_LEN-1)*sizeof(REAL));
    memmove(xf+j+1, xf, (NLMS_LEN-1)*sizeof(REAL));
}
return e;
}

int AEC::dtd(REAL d, REAL x)
{
    // optimized implementation of max(|x[0]|, |x[1]|, .., |x[L-1]|):
    // calculate max of block (DTD_LEN values)
    x = fabsf(x);
    if (x > max_x[0]) {
        max_x[0] = x;
        if (x > max_max_x) {
            max_max_x = x;
        }
    }
    if (++dtdCnt >= DTD_LEN) {
        dtdCnt = 0;
        // calculate max of max
        max_max_x = 0.0f;
        for (int i = 0; i < NLMS_LEN/DTD_LEN; ++i) {
            if (max_x[i] > max_max_x) {
                max_max_x = max_x[i];
            }
        }
        // rotate Ndx
        if (++dtdNdx >= NLMS_LEN/DTD_LEN) dtdNdx = 0;
        max_x[dtdNdx] = 0.0f;
    }
}
// The Geigel DTD algorithm with Hangover timer Thold
if (fabsf(d) >= GeigelThreshold * max_max_x) {
    hangover = Thold;
}

if (hangover) --hangover;
return (hangover > 0);
}

int AEC::doAEC(int d, int x)
{
    REAL s0 = (REAL)d;
    REAL s1 = (REAL)x;

    // Mic Highpass Filter - to remove DC
    s0 = hp00.highpass(s0);

    // Mic Highpass Filter - telephone users are used to 300Hz cut-off
    s0 = hp0.highpass(s0);

    // ambient mic level estimation
    s0avg += 1e-4f*(fabsf(s0) - s0avg);

    // Spk Highpass Filter - to remove DC
    s1 = hp1.highpass(s1);

    // Double Talk Detector
    int update = !dtd(s0, s1);

    // Acoustic Echo Cancellation
    s0 = nlms_pw(s0, s1, update);

    // Acoustic Echo Suppression
    if (update) {
        // Non Linear Processor (NLP): attenuate low volumes
        s0 *= NLPAttenuation;
    }

    // Saturation
    if (s0 > MAXPCM) {
        return (int)MAXPCM;
    } else if (s0 < -MAXPCM) {
        return (int)-MAXPCM;
    } else {
        return (int)roundf(s0);
    }
}
/* aec_test.cpp
 *
 * Copyright (C) DFS Deutsche Flugsicherung (2004). All Rights Reserved.
 * Test stub for Acoustic Echo Cancellation NLMS-pw algorithm
 * Author: Andre Adrian, DFS Deutsche Flugsicherung
 * <Andre.Adrian@dfs.de>
 *
 * compile
 * c++ -O2 -o aec_test aec_test.cpp aec.cpp -lm
 *
 * Version 1.3 set/get ambient in dB
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "aec.h"

#define TAPS (80*8)

typedef signed short MONO;

typedef struct {
  signed short l;
  signed short r;
} STEREO;

float dB2q(float dB)
{
  /* Dezibel to Ratio */
  return powf(10.0f, dB / 20.0f);
}

float q2dB(float q)
{
  /* Ratio to Dezibel */
  return 20.0f * log10f(q);
}

/* Read a raw audio file (8KHz sample frequency, 16bit PCM, stereo)
 * from stdin, echo cancel it and write it to stdout
 *
 * int main(int argc, char *argv[])
 {
   STEREO inbuf[TAPS], outbuf[TAPS];
   fprintf(stderr, "usage: aec_test [ambient in dB] <in.raw >out.raw\n");
   AEC aec;
if (argc &ge; 2) {
    aec.setambient(MAXPCM*dB2q(atof(argv[1])));
}

int taps;
while (taps = fread(inbuf, sizeof(STEREO), TAPS, stdin)) {
    int i;
    for (i = 0; i &lt; taps; ++i) {
        int s0 = inbuf[i].l;    /* left channel microphone */
        int s1 = inbuf[i].r;    /* right channel speaker */
        /* and do NLMS */
        s0 = aec.doAEC(s0, s1);

        /* copy back */
        outbuf[i].l = 0;         /* left channel silence */
        outbuf[i].r = s0;        /* right channel echo cancelled mic */
    }

    fwrite(outbuf, sizeof(STEREO), taps, stdout);
}

float ambient = aec.getambient();
float ambientdB = q2dB(ambient / 32767.0f);
fprintf(stderr, "Ambient = %2.0f dB\n", ambientdB);
fflush(NULL);
return 0;
}
På svenska
Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English
The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Anders Dahlberg, Linköping 2007-05-24