MASTER'S THESIS

DSP Algorithms on Vector Processors
A framework for testing vectorized algorithms

by
Gert Vrebos

Supervisors:
Kees van Berkel
Rudolf Mak
Eindhoven, April 2004
Abstract

Digital signal processing has become a part of everyday's life as almost every audio or video appliance sold today contains DSP technology. GSM, CD player and wireless networking are only a few of the most well know examples. Quality requirements for these digital devices are constantly increasing and consequently data throughput demands are rising likewise. The current generation of scalar DSP processor doesn't seem to be able to keep up the pace. One solution to this problem is the vector processor architecture. Vector processors are able to increase throughput, and thus performance, without the need for higher clock rates and more advanced VLSI techniques. They achieve this by manipulating blocks of data instead of single values. There is however one pitfall: most DSP algorithms need to be redesigned in order to run efficiently on a vector processor. This paper introduces an abstract vector processor and a framework for testing vectorized algorithms and it studies the redesign and evaluation of some very common DSP algorithms. As a practical example a high quality technique for image resizing is demonstrated and compared with a commercial application.
Contents

Contents

1 Introduction .................................................. 1
  1.1 Performance improvements .............................. 1
     1.1.1 Increase the number of operations per instruction  1
     1.1.2 Increase the number of instructions per clock cycle 2
  1.2 Performance analysis and benchmarking .............. 2
     1.2.1 MIPS .................................................. 3
     1.2.2 Application Implementation ......................... 3
     1.2.3 Algorithm Kernel Benchmarking ...................... 3
  1.3 Research motivation ...................................... 3
  1.4 Short overview ........................................... 4

I A Simple Vector Processor .................................. 5

2 The Simple Vector Processor ................................ 7
  2.1 Introduction .............................................. 7
  2.2 The SVP Model ........................................... 8
     2.2.1 Computational units ................................. 8
     2.2.2 I/O units ............................................ 9
  2.3 The SVP functionality ................................... 11
     2.3.1 The computational capabilities ...................... 11
     2.3.2 The I/O capabilities ................................ 13
  2.4 Conclusion ................................................ 14

3 Simple Vector Processor Language ......................... 15
  3.1 Introduction .............................................. 15
  3.2 Choosing a language model ............................... 15
  3.3 Basic Language Features ................................ 17
     3.3.1 Data structures .................................... 17
     3.3.2 Expressions ........................................ 19
     3.3.3 Statements .......................................... 21
     3.3.4 load and store statements ......................... 21
  3.4 Additional language features ........................... 22
     3.4.1 Indexing the elements of a vector ................. 22
     3.4.2 Array constructions ................................ 23
     3.4.3 Predefined constants ............................... 24
     3.4.4 Vector constructor ................................ 24


3.5 EBNF Notation ................................................. 25
3.6 Conclusion .................................................. 27

4 The SVP Interpreter ........................................... 29
  4.1 Introduction ............................................. 29
  4.2 Library layout ........................................... 29
  4.3 Start using the library ................................... 30
    4.3.1 Initializing the library ............................... 30
    4.3.2 Stopping the library - finalization ................... 31
  4.4 First example ........................................... 31
  4.5 Using the data types ..................................... 32
  4.6 Additional Features ..................................... 32
    4.6.1 I/O Facilities ....................................... 32
    4.6.2 Global constants .................................... 33
    4.6.3 Debugging ........................................... 33
  4.7 Statistics ................................................ 34
    4.7.1 Usage ............................................... 34
    4.7.2 Custom Implementation: Design ....................... 35
    4.7.3 Custom Implementation: Usage ......................... 37
    4.7.4 Custom Implementation: Processor Schedules .......... 37
    4.7.5 Default implementation: BasicProcessorStatistics ........ 39
    4.7.6 Known Issues ....................................... 43
  4.8 Conclusion ................................................ 44

5 Programming in SVPL ........................................ 45
  5.1 The first and last output values ....................... 45
  5.2 Unsupported language constructs ....................... 45
  5.3 Block-average filter .................................... 47
    5.3.1 Initial Implementation ............................... 48
    5.3.2 Optimization ........................................ 50
    5.3.3 Evaluation ........................................... 52

II Study of algorithms ........................................ 55

6 The Scaler ................................................... 59
  6.1 Definitions ............................................. 59
  6.2 The input stream $A$ ..................................... 59
  6.3 Downsample filter ....................................... 59
    6.3.1 Definition of the downsample filter $DN$ ............ 60
    6.3.2 Vectorization ....................................... 60
  6.4 Upsample filter ......................................... 61
    6.4.1 Definition of the upsampled stream $UP$ .............. 61
    6.4.2 Remark ............................................... 61
    6.4.3 Vectorization ....................................... 62
  6.5 A FIR Filter ............................................. 63
    6.5.1 FIR filter formula .................................. 63
    6.5.2 Example ............................................... 64
    6.5.3 Vectorization ....................................... 64
  6.6 Filter combinations ..................................... 67
CONTENTS

6.7 Upsample & FIR filter: Functional approach ........................................... 67
   6.7.1 Upsample and FIR filter combined .................................................. 67
   6.7.2 Eliminating the UP filter ............................................................... 68
   6.7.3 Matrix form ................................................................................... 71
   6.7.4 Grouping outputs according to upsample factor ............................... 72
   6.7.5 Grouping outputs according to vector size ....................................... 73
   6.7.6 $\tilde{C}_{g,s}$ .................................................................................... 75
   6.7.7 $A_{g,s}$ ........................................................................................... 75
   6.7.8 The effect of the limited vector size ............................................... 76
6.8 Upsample & FIR filter: Implementation .................................................... 77
   6.8.1 Initialization ................................................................................... 77
   6.8.2 Solution 1 - Constraint $((N_{d,\nu k}) + (P_{d,\nu k})) \leq P$ .............. 79
   6.8.3 Solution 2 - Eliminating the constraint ......................................... 81
6.9 The FIR & downsample filter ................................................................. 83
   6.9.1 Functional Approach ..................................................................... 83
   6.9.2 Vectorization .................................................................................. 85
6.10 The Three Filters Combined ................................................................. 86
   6.10.1 Functional Approach .................................................................. 87
6.11 Conclusion ............................................................................................ 89

7 The IIR Filter ............................................................................................ 91
   7.1 Introduction ....................................................................................... 91
   7.1.1 The order of an IIR Filter .............................................................. 92
   7.1.2 A metric for comparison ............................................................... 93
   7.2 ARMA-based vector implementations ................................................. 93
   7.2.1 Technique 1: Full unfolding .......................................................... 93
   7.2.2 Technique 2: Partial unfolding ......................................................... 96
   7.2.3 Comparison .................................................................................. 98
   7.2.4 Determining the matrix constants ................................................. 99
   7.3 Block State vector implementation ...................................................... 102
   7.4 Comparison ...................................................................................... 104
   7.5 Conclusion ....................................................................................... 107

8 Summary .................................................................................................. 109

References .................................................................................................. 113

Appendices ................................................................................................. 117

A Practical UP-FIR-DN example: Image Interpolation ............................. 117
   A.1 Introduction ...................................................................................... 117
   A.2 The Filter algorithm ......................................................................... 117
   A.2.1 Selecting an appropriate algorithm ............................................... 118
   A.2.2 How to use the Lanczos2 filter? .................................................. 121
   A.3 The results ...................................................................................... 122
   A.3.1 Resampled images ....................................................................... 123
   A.3.2 Calculating the Difference .......................................................... 124
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>SVPL Class Diagram</td>
<td>127</td>
</tr>
<tr>
<td>C</td>
<td>Header Files</td>
<td>129</td>
</tr>
<tr>
<td>C.1</td>
<td>ProcessorStatistics.h</td>
<td>129</td>
</tr>
<tr>
<td></td>
<td>List Of Figures</td>
<td>130</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Today, digital signal processing (DSP) components are being implemented in an ever-growing range of high-end as well as consumer products. While the use of DSP increases, so do quality requirements and therefore the performance requirements as well. In fact, performance requirements grow faster than the clock speed of DSPs, which is limited by hardware technology.

1.1 Performance improvements

Because of this ever-rising need for speed, other ways of boosting performance of DSP processors must be found. As the number of clock cycles per time unit is limited, engineers are looking for ways to do more work in a single clock cycle.

This goal can only be achieved by increasing the processor’s parallelism and there are two common ways to do so: increasing the number of operations per instruction or increasing the number of instructions per clock cycle [Eyr01], [Bur04], [EB00]. We will now explore both solutions in more detail:

1.1.1 Increase the number of operations per instruction

There are two ways to increase the number of operations per instruction:

Additional execution units

By extending the DSP processor with additional execution units like multipliers, adders or specialized coprocessors more operations can be done in one instruction. Of course the instruction set must be extended to make use of these units and possibly the width of instruction words needs to be increased in order to support this extended instruction set. The width of data buses may also be increased in order to supply the processor with enough data to support the additional units.

Add SIMD capabilities

SIMD is an acronym for Single Instruction, Multiple Data. It refers to an architecture that efficiently handles large quantities of data in parallel such as in a
vector processor or array processor. Operations are performed on blocks of data and therefore each instruction performs lots of work. This means that the technique is most effective on algorithms that process large blocks of data. Not all algorithms can easily be rewritten for implementation on a SIMD architecture.

1.1.2 Increase the number of instructions per clock cycle

Use superscalar techniques

A superscalar architecture has specialized hardware that determines at runtime which instructions will be executed in parallel, based on availability of resources and data dependencies. The advantage of this technique is that because of the small and simple instructions, it rather easy to build compilers for high-level languages. The programmer or compiler does not have to determine which instructions will be performed in parallel. A major drawback of the superscalar approach is that the execution time of a certain piece of code cannot be predicted. The execution time of the same piece of code can even differ, depending on the previous instructions.

Use VLIW techniques

The VLIW or Very Long Instruction Word technique is also known as static superscalar or compile time scheduling. It moves the burden of scheduling instructions to the compiler or the assembler programmer. With this technique single processor instructions contain multiple independent operations. Because of sensitivity to price, size and power usage of DSP processors, instructions words are often kept as small as possible. This results in very complex 'packed' instructions that are highly constrained. There are for instance restrictions on which registers to use and which instructions to execute in parallel because there are not enough bits to encode all possible combinations. The complexity of the instructions makes it very difficult to write efficient compilers for VLIW DSP processors, which means that for most VLIW DSPs, programs are written directly in assembler.

1.2 Performance analysis and benchmarking

As opposed to general-purpose processors (GPPs), DSP processors are designed with a very specific objective: the manipulation of digital signals. Therefore DSP processors always have very specialized and complicated instruction sets that make them very efficient at executing mathematical operations on these signals. It also means that each DSP has its own, unique instruction set, tailored to its architecture. This makes it very difficult to compare different DSPs with regard to performance.

In the past, different techniques have been used to compare DSP processors ([EB98a] [EB98b] [Inc00]):
1.2.1 MIPS

The most basic metric is MIPS or Millions of Instructions Per Second. This is very easy to measure, but also very inaccurate. A DSP processor with a low MIPS ratio can actually be faster than one with a higher MIPS ratio because it does more work in a single instruction. Thus performance does not only depend on the number of instructions that can be done in a certain amount of time, but also on the work that is performed by each instruction, which is fully architecture dependent.

1.2.2 Application Implementation

A better way to measure performance is to implement a balanced set of common algorithms on each DSP processor under consideration and measure their total execution time. It is however very costly and time-consuming not only to implement such a set on every processor but also to make sure that the implementation is optimized for each processor. It is therefore not only a test of the DSP, but of the entire system and even of the programmer's skills.

1.2.3 Algorithm Kernel Benchmarking

A good trade-off between the simple MIPS and the complicated application implementation benchmark is the algorithm kernel benchmarking strategy. Most processing time of a DSP application is concentrated in DSP algorithms and most processing time of a DSP algorithm is concentrated in only a small part, the algorithm kernel. A selected set of such kernels is often used as a benchmark. These kernels are small enough to be implemented and optimized for every DSP processor under test. The FIR filter and IIR filter kernels are two examples of such representative kernels. A well-known example of a DSP kernel benchmark is the BDTI Benchmark™ Suite, a set of 11 DSP algorithm kernels and implementation and measuring rules.

1.3 Research motivation

As we have seen in the previous section, one of the techniques of increasing data throughput without increasing the clock rate is SIMD. We have also indicated the importance of DSP kernel implementations for benchmarking. In this document we will study some Digital Signal Processing (DSP) algorithm kernels and more specifically the implementation of these on a (SIMD) vector processor.

The real challenge is scalability, we want to design algorithms in such a way that they benefit from arbitrary large amounts of parallelism (read: arbitrary vector length). If we can implement an algorithm in such manner, it can be run at arbitrary high data rates by designing a vector processor with sufficiently large vector size without the need to use or possibly invent a faster hardware technology.

Of course, it is not always easy to find such implementations for all algorithms and for some algorithms there simply is no solution. First of all, algorithms can
be of a recursive nature by which certain previous output values must be known before new output values can be calculated. And secondly, algorithm implementations never solely consist of statements calculating output values. There is almost always overhead of processor control statements that need sequential execution and it is this sequential execution that limits the possible speedup of any algorithm. Both indicate the same problem of sequential execution restricting the speedup of parallel execution and this problem is known as Amdahl’s Law\(^1\).

Most modern DSPs use techniques to minimize the control overhead like zero-overhead looping and specialized addressing, but even these resources are limited. This still makes it necessary to optimize not only the computing code of an algorithm, but also the control code.

### 1.4 Short overview

Before we start the study of algorithms and their implementations, we define a machine model and a high-level programming language that is based upon this model. We will also create a programming environment to support this new language.

After the model, the Simple Vector Processor (SVP), and the programming language, the Simple Vector Processor Language (SVPL), have been defined, they will be used in the study of DSP algorithms. The machine model is created in order to perform accurate complexity analysis, but the language will define the actual Simple Vector Processor. SVPL will be used for the implementation of the algorithm and the SVP architecture for performance analysis purposes. We develop a C++ library to support this vector processor language. Each algorithm is first described in an informal way where a mathematical derivation is given. From the mathematical description, a SVP algorithm is derived, implemented and finally run in the SVP library, a library that enables programming in SVP and generates interesting processor statistics. These statistics will then be used to study complexity, processor usage and performance of the algorithm.

As a case study, the use of the FIR filter implementation for digital image resizing is evaluated in Appendix A.

---

\(^1\)Amdahl’s law [Wik04]: Amdahl’s law, named after computer architect Gene Amdahl, states that if \( F \) is the fraction of a calculation that is sequential, and \( 1 - F \) is the fraction that can be parallelized, then the maximum speedup that can be achieved by using \( P \) processors is \( 1/(F + (1 - F)/P) \). In the limit, as \( P \) tends to infinity, the maximum speedup tends to \( 1/F \). In practice, price/performance ratio falls rapidly as \( P \) is increased once \((1 - F)/P\) is small compared to \( F \).
Part I

A Simple Vector Processor
Chapter 2

The Simple Vector Processor

2.1 Introduction

A vector processor consists of the same elements as a standard processor, but it also contains a vector-processing unit. This unit provides high-level operations that work on complete vectors of numbers. Such a vector is a linear array of numbers with a fixed length. This introduces a much bigger physical processor since most functional units are replicated several times. Such a vector processor belongs to the SIMD class of processors.

A vector processor also has great computing advantages. Where it would take a normal processor 32 computing cycles to execute 32 additions, it would only take 1 cycle for a 32-element vector processor to perform the same amount of work! This advantage is especially interesting when manipulating lots of numbers, as is the case in digital signal processing or DSP.

For example, the average data rate or bit rate of an original DVD MPEG-2 stream is 4 to 5 million bits per second (Mbit/s). A typical high-end DVD player's video DSP executes more than 6 billion operations per second when decoding such a data stream for real-time display.\(^1\)

As today's fastest, and therefore most expensive, commercially available PC processors only peak at 6000 MIPS, while consuming a lot of power (up to 60 W) and emitting a lot of heat, it is clear that other solutions must be used in a home-cinema appliance in order to keep prices reasonable. That is where the vector technology can be used.

Extensive information on vector processors can be found in [HP90].

\(^1\)From [Wik04] and other sources: The DVD standard (which includes a restricted MPEG-2 standard) allows a very broad range of bit rates with a maximum of 9.8 Mbit/s. Using your own DVD writer, you could easily create a DVD at one tenth of this data rate, but quality would get very poor. The previously mentioned rate of 4 to 5 million is however a very typical figure for manufactured DVDs.
2.2 The SVP Model

We will now describe a basic vector processor architecture, which we will use as a model in the rest of this document: the SVP or Simple Vector Processor architecture. The architecture as we will see in this chapter, has grown gradually during the work on the SVPL and on the algorithms. However, to make things not too complicated, we will simply look at the specifications of the processor as a given fact.

It is important to understand that it is not the description of the SVP architecture that defines our processor; it will only be a guideline in designing the SVPL or Simple Vector Processor Language. The SVP can be seen as a detailed computational model, a basic implementation of the generic SIMD model. The SVPL language will finally define the processor for which we will write algorithms. In this way, we don't need to create a complete specification of the SVP and we can simply rely on the SVPL definition, which we will specify in much greater detail.

The purpose of the SVP model is twofold. In the first place it is used to define the SVPL and secondly, the model will allow us to perform precise complexity analysis. We will not only be able to give the order of an implemented algorithm, but also the order constants.

We will first create an outline of the Simple Vector Processor by describing its computational units, the input/output units and the data those units can handle. Remember that the SVP is a theoretical model and that it is not designed for implementation in hardware. Most design decisions where made in favor of simplicity. A simple overview of the SVP can be found in figure 2.1.

2.2.1 Computational units

The SVP is capable of working with integer values as well as with values of some scalar type. The actual implementation of this type is left undefined here for the sake of generality. This scalar type can possibly be implemented as a floating-point or fixed-point type or as an other mathematical type. Nevertheless, leaving this type undefined gives us enough detail to work with the SVP throughout this document.

A vector is an array of such Scalar values and the length of the vectors is fixed but is also left undefined. In the SVP emulating library, the user, depending on his needs, can define the vector size.

The SVP has one Scalar unit to handle single Scalar values and one Vector unit that handles entire Vectors of Scalar values at once.

The actual speed of the processor is also not specified. The speed of programs running on the processor will be expressed in cycles. A cycle is the smallest time unit on the processor and it is exactly the time necessary to execute a single instruction. All instructions, scalar as well as vector operations, mathematical
operations as well as memory accesses are defined to take exactly one cycle to complete.

2.2.2 I/O units

We do not only need to describe the computational capabilities of the processor, but also its facilities to communicate with the 'outside world', i.e. the adjacent memory system.

DSP algorithms typical take a (conceptually infinite) stream of data for input and they produce another (also conceptually infinite) data stream, based on current and previous values of the input stream and/or previous output values. In contrast to most DSP processors, we keep the input memory (IM) and output memory (OM) strictly separated. This keeps the model close to the idea of a filter being placed on a data stream. The input memory (IM) and is read-only, the output memory (OM) and is read- and writable.

A set of processor registers is available to store a limited number of integer, scalar and vector values. The number of registers is not defined and for simplicity we suppose that there are enough to hold a reasonable amount of local variables.

The SVP also has a larger intermediate memory to store temporary values. The limited number of processor registers will not be enough in some situations
to hold temporary values. It might be essential for an algorithm to calculate an arbitrary number of intermediate vector or scalar values, depending on algorithm parameters. It cannot be guaranteed that there will be enough registers to store all those values. Since the input memory is read-only and the output memory can only be used to store output values, an extra memory is necessary to store those intermediate values.

In hardware, the RAM would possibly be implemented as a combination of some finite number of local processor registers and a much bigger random access memory with some smart caching techniques, but since we are only defining a theoretical model, we leave this open and will work with the abstraction of one simple Random Access Memory (RAM). This means that every access to this memory will take a vector cycle and that optimizations must be written explicitly. Those optimizations might include:

- reading values from RAM a few cycles earlier than necessary during memory cycles that would otherwise be left unused,

- keeping values that need to be accessed a few times consecutively in a local register for a short period of time before writing it back the correct location in RAM,

- not writing the most frequently accessed values to RAM at all, when there are sufficient registers available.

All three memories (IM, OM and RAM) are address-oriented: values can be loaded from and stored into these memories by supplying a memory address. The memories start at address 0 and each scalar value (also the ones included in a vector value) is individually addressable. IM, OM and RAM can only store scalar and vector values.

Consequently:

- successive scalar locations in memory have successive addresses.

- successive vectors in memory have address values that differ with the vector size.

- vectors can start at arbitrary locations in these memories, i.e. not only at addresses that are a multiple of the vector size.

- the individual elements of a vector can be addressed and accessed as single scalar values.

This ensures the greatest possible freedom in handling vectors and scalars in memory so that algorithms will be restricted by memory limitations as little as possible.
2.3 The SVP functionality

In this section we will take a more detailed look at the capabilities we defined above. We will see which functionality is provided by either computational or I/O units.

2.3.1 The computational capabilities

The processor has three separate computational units: the integer, scalar and vector unit. Whereas the scalar unit only calculates with scalar values, the vector unit can do more or less the same computations on a complete vector of scalar values or on a combination of a vector and a scalar value in exactly the same time (we will speak of one clock cycle).

The integer unit is used for control and address computations. It can also handle boolean type.

The scalar unit can perform all basic mathematical operations, namely:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(+, -, *, /)</td>
<td>Scalar</td>
<td>Scalar</td>
<td>Scalar</td>
</tr>
<tr>
<td>Comparison:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(&lt;, &gt;, ==, !=, &lt;=, &gt;=)</td>
<td>Scalar</td>
<td>Scalar</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

The vector unit also supports these basic operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(+, -, *, /)</td>
<td>Vector</td>
<td>Vector</td>
<td>Vector</td>
</tr>
<tr>
<td>Comparison:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(&lt;, &gt;, ==, !=, &lt;=, &gt;=)</td>
<td>Vector</td>
<td>Vector</td>
<td>Vector</td>
</tr>
</tbody>
</table>

All these vector operators execute pointwise: when two Vectors A and B are added, the n-th value of resulting vector C is the result of adding the n-th values from A and B. Comparison in the vector unit results in a vector that only contains 0's and 1's. 1 corresponds with true and 0 with false.

The vector unit also supplies some additional features that are standard operations in almost all present vector processors:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift Left</td>
<td>Vector</td>
<td>Scalar</td>
<td>-</td>
<td>Vector</td>
</tr>
<tr>
<td>Shift Right</td>
<td>Vector</td>
<td>Scalar</td>
<td>-</td>
<td>Vector</td>
</tr>
<tr>
<td>Shuffle</td>
<td>Vector</td>
<td>Vector</td>
<td>-</td>
<td>Vector</td>
</tr>
<tr>
<td>MAC</td>
<td>Vector</td>
<td>Vector</td>
<td>Vector</td>
<td>Vector</td>
</tr>
</tbody>
</table>

We'll explain these new operations and illustrate them with an example of vector size 4.
Shift Left
All elements in the vector ‘Operand 1’ are shifted to the left by one location. Hence the leftmost value is discarded. The value of ‘Operand 2’ is inserted into the rightmost location.

\[ \langle a_0, a_1, \ldots, a_{n-1}, b \rangle \longrightarrow \langle a_1, a_2, \ldots, a_{n-1}, b \rangle \]

Example:
\[ \langle 5, 6, 3, 7 \rangle, 9 \longrightarrow \langle 6, 3, 7, 9 \rangle \]

Shift Right
All elements in the vector ‘Operand 1’ are shifted to the right by location. Hence the rightmost value is discarded. The value of ‘Operand 2’ is inserted into the leftmost location.

\[ \langle a_0, a_1, \ldots, a_{n-1}, b \rangle \longrightarrow \langle b, a_0, a_1, \ldots, a_{n-2} \rangle \]

Example:
\[ \langle 5, 6, 3, 7 \rangle, 9 \longrightarrow \langle 9, 5, 6, 3 \rangle \]

Shuffle
All elements in vector ‘Operand 1’ are put in the ‘result’ vector on the locations as specified in ‘Operand 2’.

\[ \langle a_0, a_1, \ldots, a_{n-1} \rangle, \langle b_0, b_1, \ldots, b_{n-1} \rangle \longrightarrow \langle a_0, a_1, \ldots, a_{n-1} \rangle \]

\[ (\forall x : 0 \leq x < n : 0 \leq b_x < n) \]

Two examples:
\[ \langle 5, 6, 3, 7 \rangle, \langle 1, 0, 3, 2 \rangle \longrightarrow \langle 6, 5, 7, 3 \rangle \]
\[ \langle 5, 6, 3, 7 \rangle, \langle 1, 0, 1, 1 \rangle \longrightarrow \langle 6, 5, 6, 6 \rangle \]

Multiply Accumulate (MAC)
Underlying all of the processing, the DSPs are optimized for executing mathematical algorithms that are simple combinations of multiply and addition. These two operations are typically combined into a single unit referred to as a multiply and accumulate (MAC) operator. Optimizing the MAC on a DSP’s internal architecture is the key to getting the greatest performance from a particular DSP.

‘Operand 1’ is multiplied by ‘Operand 2’ and after that, this result is added to ‘Operand 3’ and stored in ‘Operand 3’.

\[ \langle a_0, a_1, \ldots, a_{n-1} \rangle, \langle b_0, b_1, \ldots, b_{n-1} \rangle, \langle c_0, c_1, \ldots, c_{n-1} \rangle \]
\[ \longrightarrow \langle (a_0 \times b_0) + c_0, (a_1 \times b_1) + c_1, \ldots, (a_{n-1} \times b_{n-1}) + c_{n-1} \rangle \]
2.3 The SVP functionality

Example:
\(< 2, 3, 4, 2 >, < 2, 1, 2, 3 >, < 1, 4, 1, 3 > \rightarrow < 5, 7, 9, 9 >\)

Execution time
Both shift operations and the shuffle operations can be performed together with another vector operation, as some kind of preprocessing operation. This means that they do not consume a processor cycle.

The MAC operator however, takes one cycle to execute. Still, this makes it the most interesting operation of the SVP, since it performs the largest amount of work in one cycle: both a vector multiplication and addition.

2.3.2 The I/O capabilities
These are the operations to load and store vectors and scalars from and into the 3 different memories.

It is possible to:
- Load from IM
- Load from and store into OM
- Load from and store into RAM

All three memories can handle both scalar and vector values. Input memory cannot be written to since input values are written into the input memory by an external actor (another program, an ADC\(^2\) or maybe even a person). The intermediate RAM was introduced to avoid the need to write to the IM.

Finally, there is one special function that can be applied when loading a vector from IM or loading/storing a vector from/into OM or RAM and it is called loading/storing with vector stride. This means that a vector is loaded/stored from/into memory with a certain distance between two consecutive values. The length of the interleaving interval is referred to as the vector stride. This function is especially convenient when performing matrix operations. For instance when a matrix is stored in memory row after row, the stride option makes it easy to load a single column from that matrix.

The following table summarizes all possible combinations:

<table>
<thead>
<tr>
<th></th>
<th>IM</th>
<th>OM</th>
<th>RAM</th>
<th></th>
<th>IM</th>
<th>OM</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>without stride</td>
<td>with stride</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Scalar</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Store Scalar</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Load Vector</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Store Vector</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

\(^2\)ADC = Analog to Digital Converter, samples an analog input signal at a fixed rate and converts these samples to digital values.
2.4 Conclusion

In this chapter we have described the architecture of the Simple Vector Processor that we will use in the subsequent chapters. Note that we did not provide an exhaustive description, nor did we define a formal assembly language. We will define a high-level programming language (Simple Vector Processor Language or SVPL) including a formal description in the next chapter and we will be using that high-level language in the rest of this document.

The SVP architecture is a theoretical design that was developed for simplicity, both in the processor and in the implemented algorithms. Therefore, the SVP architecture differs from the commonly used Neumann and Harvard processor architectures that are much closer related to actual hardware implementation. On the other hand, SVP is closely related to the concept of a programmable stream filter.

In the next section the SVP architecture will only be used as a guideline and motivation for the development of SVPL. Afterwards, SVPL will define the actual virtual machine and SVP will only be used as one of the guidelines for complexity analysis.
Chapter 3

Simple Vector Processor Language

3.1 Introduction

In this chapter we develop a programming language, specifically designed for use with the Simple Vector Processor as defined in the previous chapter. The language will be called Simple Vector Programming Language or SVPL. We will use this programming language to implement algorithms and study their performance. In chapter 4 we will explain the design and use of an interpreter for SVPL that will allow us to test the implementations of those algorithms and that will generate processor statistics for comparison with manual complexity analysis.

We need to select a strategy in defining the language syntax. We can either create a language from scratch and create a compiler or interpreter for this language, or we can base the new vector-oriented language on an existing language and reuse existing tools. Clearly, the second strategy is best if we can find a suitable language.

3.2 Choosing a language model

There are lots of good programming languages, all with their own advantages and disadvantages. Therefore it would not be a good idea to create a new programming language, we should rather choose one that can easily be extended with the SVP functionality. Besides, we want to focus on the design and implementation of the algorithms, so the development of the SVPL interpreter should take as little time as possible.

If we can create a library that implements all SVPL programming language constructs, that problem is solved. Moreover, when we create a library that implements the SVP functionality, it’s an easy task to add software modules that support the gathering of processor usage statistics. This implementation of SVP functionality and statistics, introduces the need for an Object Oriented
(OO) language. For reasons of code portability we should only choose from popular OO languages that have been implemented on a wide range of hardware and software platforms.

The above-mentioned criteria narrowed the choice down to the two most widely used OO-languages: Java and C++.

By using C++ as a model for the language, it will be easy to create a C++ library implementation of the programming language constructs. Java is much more restricted in OO-constructs as it does not, for instance, support operator overloading. Operator overloading makes it easy to create processor specific data types that behave like we want them to, while the library user can still use these types in the same way he uses build-in types. This is why we should definitely choose C++ over Java.

C++ is clearly the most interesting choice to use as a model. It is one of the most widely adopted programming languages. There are compilers for almost all operating systems, and most computer engineers are familiar with the C++ language. Finally, programmers that are not familiar with C++, but that do know Java should have no problem in learning SVPL because the Java syntax is very similar to the C++ syntax, especially when limited to the SVPL features.

If we can put some constraints on the use of C++ by only allowing certain statements and constructs, and by enforcing the use of the library in predefined ways, the vector programming language we are about to design will be easy to implement using an existing C++ compiler or interpreter.

Now that we have selected a model for the vector programming language, we can start defining that language. We will start by creating a limited number of basic language features and afterwards we will look at some language constructs we need to add, in order to fully support the SVP architecture.

It is very important to understand that although we will base SVPL on the C++ language, not a single feature from that language belongs to SVPL unless it has explicitly been named in the rest of this section. If you would create a program that contains other constructs than the one named hereafter, your program will not be valid and the correctness of the results cannot be guaranteed!

A final, preliminary note is that the language we are about to design should be a bare programming language. We will add as little features as possible, but without loss of expressivity. By ensuring that, we will not loose ourselves in complicated programming issues. We should focus on the algorithms, not on the actual SVPL implementation. Therefore we will, for example, not allow the programmer to use any OO constructs other than the ones implemented in and imposed by the SVPL library.
3.3 Basic Language Features

We will need two kinds of basic features: data structures and the operations that can be used with these structures and secondly the flow control constructs.

3.3.1 Data structures

We first determine the data types that will be supported by the library. Apart from types that are used to hold data manipulated by the processor (input and output data), we will need data types for program and data control.

Scalar

The basic data type is called Scalar and obviously represents a single scalar value. We define the class Scalar:

```cpp
class Scalar {
    ...
}
```

This is the type of all input and output values.

Vector

Of course we also need the vector type. If we wouldn’t introduce this type, we would design another scalar programming language, and all vectorization would have to be done by the compiler, which is exactly what we are trying to avoid. This vector type will simply be called the Vector type. The Vector type exists of a fixed number (= the vector size) of Scalar values:

```cpp
class Vector {
    ...
    private:
        static Integer fVectorSize;
        std::vector<Scalar> fContent;
}
```

Integer and Address

We still need data type support for memory addressing and control purposes.

As we have seen in the previous chapter, the Scalar type is not suitable for those tasks. Recall that we cannot rely on the Scalar type to act exactly like an integer value since it can have a floating-point or a fixed-point implementation (we left this undefined). Addressing the memory with a value of 1,5 for example would not make any sense and the result would be undefined. As is very common with processors, addresses are implemented as natural numbers. The processor’s integer unit can thus support both Integer and Address types.
There are several good reasons to introduce two separate types for memory addressing and control purposes: the 'Address' and 'Integer' types:

- The most important reason to separate addresses from the generic integers is that it enables the library to generate more detailed statistical information on how processor cycles are consumed. It also makes it easy to implement additional processor optimizations that are aimed at either program control (e.g. zero-overhead looping) or memory addressing (e.g. separate addressing unit).

- Address values should point to a valid location in memory, say positive values and within memory boundaries. Since we will program in SVP using an interpreter, run-time boundary checks can be implemented on the Address type to reduce the possibility of erroneous programs.

- Operations on Address values can be more restricted than those on the Integer type as not all operations on Addresses make sense (e.g. multiplication of two addresses). This can be done by prohibiting access to certain operators.

- We cannot use the standard build-in int type from C++. There are too much operators defined on that type that we do not want to support and, more importantly, as we will see in section 4.7, the SVPL library implements a system to calculate processor usage statistics. These statistics cannot be calculated on the standard int type. Still, it is very important to include the Integer and Address operations in the statistics since they are used for processor control, which is where the overhead of Amdahl's Law is situated (see chapter 1).

Boolean Values

As the presence of boolean variables is not essential, an explicit boolean type will not be included in the SVPL definition. Nevertheless, boolean values will occur in SVPL programs as the result of the evaluation of expressions in conditional branches and loops.

Additional notes on data types

We defined four different data types to be supported by the SVP library: Scalar, Vector, Integer and Address. No other data types, not even the native C++ data types, are allowed in SVP programs. As explained above, the use of such types would render the generated statistics incorrect and thus useless.

Still, it is not possible to avoid every C++ int and int operation since every integer literal will be of the standard int type. If two of those literals are operated on, that operator is also a int operator (e.g. \( s = 5 + 7 \); with s an Integer). Those operators cannot be included by the statistics but luckily that is no real issue. Even the simplest compiler would optimize this code by executing the operator at compile-time and replacing it by the resulting statement (\( s = 12 \)).
All data types default to the corresponding zero value. This means 0 for Address, Integer and Scalar types, and a series of zeros for the Vector type.

Variables and Processor Registers

Every modern C++ compiler tries to assign local variables to processor registers since those registers are the fastest memory locations. In most processors, there is only a very limited number of local registers and therefore, when all registers are in use, the compiler must add instructions to write variables to a slower memory before it can free up a register. This kind of memory management introduces delays and is hidden from the programmer.

As is the case with most processors, the SVP has only a limited number of processor registers and therefore the RAM memory was introduced in section 2.2.2. When programming in SVPL, you may however declare a reasonable number of local variables. Since the number of registers is not defined, we will assume that all the declared variables can be implemented in vector registers.

On the other hand, SVPL requires you to program all dynamic memory management explicitly. There is no way to define array variables or to dynamically allocate variables, the number of used registers is limited and known at compile-time. Whenever a number of intermediate values needs to be calculated and that number of values depends on algorithm parameters, these values should be stored in RAM. Since there is no way to assign memory (new is not supported), you are left no choice but to use the RAM. Using the RAM automatically introduces memory overhead which than can be included in statistical processor reports.

If, for some reason, the number of registers is defined, it is easy to stick to this predefined number and thus to write correct programs. You should declare all available registers as variables at the beginning of your program (e.g. 32 scalar registers s1 to s32 and 16 vector registers v1 to v16) and from that point on only use those variables. You will automatically need to write variables to RAM whenever you run short on registers.

3.3.2 Expressions

Operations that do not modify the object they were called upon, are implemented as const member functions and are thus classified as expressions. On the other hand, all operations that do modify the caller object are classified as statements. In this section we deal with the expressions.

The SVP programming language will support all basic unary and binary operators on Vector and Scalar types in all three combinations (Vector-Vector, Scalar-Scalar and Vector-Scalar) as far as they are defined on the SVP. Since we will be programming pure math based algorithms this no luxury.
Simple Vector Processor Language

Below we list the supported operators, we give the definition of one operator of each category for both Vector and Scalar type.

- **Unary operators: **\(-,++,--\) (only Scalar type)

  ```
  class Scalar {
      Scalar& operator++ () ; // prefix
      Scalar& operator++ (int); // postfix
  }
  ```

- **Binary operators: **\(+,-,*,/\)

  ```
  class Scalar {
      Scalar& operator+ (const Scalar&) const;
  }
  class Vector {
      Vector& operator+ (const Vector&) const;
      Vector& operator+ (const Scalar&) const;
  }
  ```

- **Relational operators: **\(<,>,==,!=,>=,<=\)

  ```
  class Scalar {
      bool& operator< (const Scalar&) const;
  }
  class Vector {
      Vector& operator< (const Vector&) const;
      Vector& operator< (const Scalar&) const;
  }
  ```

Not all desired functionality can be expressed with these operators. In addition we need operators for vector specific operations. Their functionality was explained in detail in the previous chapter.

`'ShiftLeft' and 'ShiftRight'` Takes one argument: a single scalar.

`'Shuffle'` Takes one argument: a vector with shuffle positions, functionality as expected.

These operators are implemented as member functions of the Vector type, exactly like the other operators:

```
class Vector {
    Vector shiftleft (const Scalar&) const;
    Vector shiftright (const Scalar&) const;
    Vector shuffle (const Vector&) const;
}
```

As we mentioned above, the three functions are const functions. E.g., if you want to shift the values of vector and keep the result in that variable, you should not use
3.3 Basic Language Features

```
v1.shifleft(s1);
```

as it does not modify the v1 object, instead you need to use

```
v1 = v1.shifleft(s1);
```

By implementing these operators as expressions, we keep the SVPL as close as possible to the SVP model. As seen in chapter 2 the shift functions can be seen as a preprocessor to the Vector unit, not as a stand-alone operation. That is why we do not implement them as a statement but as an operator that leaves it's object unchanged.

3.3.3 Statements

We need at least a basic set of statements, not only for our programming language to be sufficiently expressive, but also to create programs with at least a minimum amount of convenience.

This basic set consists of:

- The assignment ('=')
- A simple loop construct ('while')
- The conditional if {...} else {...} construct

In this section we need some vector specific statements as well:

- One math related statement: 'MAC'

```cpp
class Vector {
    void MAC(const Vector& factor1, const Vector &factor2);
}
```

- Some statements to load vectors and scalars from and to memory.

Unlike the shift and shuffle operators, the MAC operator changes the object it called upon and thus it is classified as a statement.

3.3.4 load and store statements

Recall the three different memories: IM, OM and RAM. Here we define the operations for reading from and writing into these memories.

To understand the definitions below, you need to know that these three memories are an implementation of the IOMemory abstract class.
These are the different load and store operations:

- load and store for Vector and Scalar:

```cpp
class Vector {
    void load(IOMemory &mem, const Address &addr);
    void store(IOMemory &mem, const Address &addr) const;
}
class Scalar {
    void load(IOMemory &mem, const Address &addr);
    void store(IOMemory &mem, const Address &addr) const;
}
```

- loadWithStride and storeWithStride for Vector objects:

```cpp
class Vector {
    void loadWithStride(IOMemory &mem, const Address &addr,
                         const Integer stride);
    void storeWithStride(IOMemory &mem, const Address &addr,
                         const Integer stride) const;
}
```

Remember that there is no possibility to load or store Integers or Addresses from/in memory since all three memories are Scalar based.

### 3.4 Additional language features

So far we only made a coarse sketch of the programming language. We selected a basic set of constructs from the model language (being C++) and added the obvious vector functionality as present in the SVPL. At this stage, this sketch will not stand when we use it to implement algorithms. Showing the problems that might arise using some simple examples will indicate the missing features of our language. We will then solve these problems with the best solution available.

#### 3.4.1 Indexing the elements of a vector

We will often need the ability to access a single element from a Vector variable, e.g. when performing matrix operations. We also want to specify vector constants in code.

*We assume vector size 4 in the following code examples.*

```cpp
Vector v1, v2;
Scalar s;

v1 = {1, 3.9, 0, s}; // This notation is not valid, but
// raises the question:
// How do we hard code a vector ?

for (Integer k=0; k < 4; ++k) {
    s = s + v1[k];
}
```
3.4 Additional language features

The indexing operator [ ] on Vectors seems indispensable and easy to implement. A vector of constants can then be hard coded by a sequence of assignments on a indexed vector element. This is not a very aesthetic solution but since hard coded vector constants will not occur very often, it is not necessary to introduce the increased complexity of another representation.

We will thus not be using the \{ , , , \} notation as used in the above source code example. This is the corrected implementation:

```c
Vector v1, v2;
Scalar s;

v1[0] = 1;
v1[1] = 3.9;
v1[2] = 0;
v1[3] = 8;

for (Integer k=0; k < 4; ++k) {
    s = s + v1[k];
}
```

3.4.2 Array constructions

We did not create any array concept. In particular, arrays of Vectors can currently not be used. We will however need to implement an algorithm similar to the example below in the UP-FIR filter implementation in section 6.7:

```c
// These vectors will be created on the fly,
// n depends on the input.
Vector v[1] ... Vector v[n]

// These vectors need to be used a loop:
Integer i = 0;

while (i < n) {
    // use v[i];
    ++i;
}
```

Simply permitting arrays of vectors might seem the most suitable solution at first sight, but there is in fact another solution, without creating additional language features, by using the RAM memory:

```c
// All values of the 'virtual' vectors q[1] - q[n]
// are sequentially stored in RAM, starting at
// this base address:
Address v_basepointer = 32;
Vector v;

// The vectors are fetched from RAM when needed
Integer i = 0;
```
while (i < n) {
    v.load (RAM, basepointer + VectorSize * i);
    // use v;
    ++i,
}

This program will behave exactly the same way as the first one. When using the array construction, the compiler will actually need some kind of memory to store the Vector arrays. We will use the RAM memory to solve this kind of problems, regardless of the possible added complexity to the program code. This additional complexity can easily be illustrated:

Example:
Addressing the 4th vector in a dynamic sized collection of vectors:

1. Using the array system:
   myVectorCollection[4]
2. Using the third memory:
   v.load(RAM, baseaddress + 3 * vectorsize)

This last example also reveals the fact that Vector arrays conceal some computing complexity to the programmer, which makes it harder to write efficient programs. We can now, for example, explicitly cache some frequently used Vector elements in local variables (registers) whereas the array system should rely on the memory manager to do this.

3.4.3 Predefined constants
In many algorithms we need the vector size. Since this is a fixed value within a single program, we will make it a global variable that will be accessible as 'VectorSize' within the context of a SVP program:

namespace SVP {
    Integer VectorSize;
}

3.4.4 Vector constructor

Vector v1, v2;
Scalar s;

v2.MAC(v1, s);

The MAC statement in the last line is invalid. The MAC statement needs two Vectors as parameters. There are two different solutions for this problem. First of all we can simply make the MAC operator support a Scalar value as parameter, but this would not be faithful to the SVP architecture. A better solution is to support a Vector constructor that takes one Scalar value and creates a vector that is completely filled with this value:
3.5 EBNF Notation of the Simple Vector Processor Language

This is the Extended BNF notation of SVPL. The non-terminal symbols Identifier, IntegerLiteral and FloatLiteral are not specified below. They inherit their definition from C++ and are omitted here. Their specification introduces a lot of additional less relevant non-terminals and the concepts are well known to every C++ programmer. [ANS96]

Program = StatementSequence;

StatementSequence ::= StatementSequence, ';', Declaration
| StatementSequence, ';', Statement
| Declaration
| Statement
;

Declaration ::= "Vector", Identifier
| "Scalar", Identifier
| "Integer", Identifier
| "Address", Identifier
;

Statement ::= ';
| ScalarIdentifier, '=', ScalarExpression
| VectorIdentifier, '=', VectorExpression
| "while", '(', Expression, ')', '{', StatementSequence, '}'
| "if", '(', Expression, ')', StatementSequence
| "else", '{', StatementSequence, '}'
| ScalarIdentifier, "load",
| '(', Memory, '(', MemoryExpression, ')
| ScalarIdentifier, "store",
| '(', Memory, '(', MemoryExpression, ')
| VectorIdentifier, "load",
| '(', Memory, '(', AddressExpression, ')
| VectorIdentifier, "store",
| '(', Memory, '(', AddressExpression, ')
| VectorIdentifier, "load",
| '(', Memory, '(', AddressExpression, ')
| IntegerExpression, ')
| VectorIdentifier, "store",
|
Simple Vector Processor Language

()` Memory, ', AddressExpression ', IntegerExpression, ')
| VectorIdentifier, ".MAC",
| (`', VectorExpression ', VectorExpression, ')')

BooleanExpression ::= `(', BooleanExpression, ')'
| ScalarExpression, RelationalOperator, ScalarExpression
| IntegerExpression, RelationalOperator, IntegerExpression
| '!', BooleanExpression

VectorExpression ::= `(', VectorExpression, ')
| VectorExpression, BinaryOperator, VectorExpression
| ScalarExpression, BinaryOperator, VectorExpression
| VectorExpression, BinaryOperator, ScalarExpression
| VectorIdentifier, "shiftleft", (`', ScalarExpression, ')
| VectorIdentifier, "shiftright", (`', ScalarExpression, ')
| VectorIdentifier, "shuffle", (`', VectorExpression, ')
| [Sign] VectorIdentifier
| "Vector", (`', ScalarExpression, ')

ScalarExpression ::= `(', ScalarExpression, ')
| Identifier
| FloatLiteral
| IntegerLiteral
| ScalarExpression, BinaryOperator, ScalarExpression
| [Sign] Identifier

IntegerExpression ::= `(', IntegerExpression, ')
| IntegerLiteral
| IntegerExpression, BinaryOperator, IntegerExpression
| [Sign] Identifier

AddressExpression ::= `(', AddressExpression, ')
| IntegerExpression
| AddressExpression, TermOperator, AddressExpression
| [Sign] Identifier

BinaryOperator ::= TermOperator | FactorOperator;
3.6 Conclusion

TermOperator ::= 
    '+', | '-' ;

FactorOperator ::= 
    '*', | '/', | '%' ;

RelationalOperator ::= 
    '==', | '!=', | '>>', | '<', | '>=', | '<=' ;

Sign ::= 
    '+', | '-' ;

Memory ::= 
    "TM" | "OM" | "RAM" ;

3.6 Conclusion

In this chapter we have defined a new programming language with vector support: SVPL or Simple Vector Processor Language. It is based on the well-known C++ programming language and was kept simple and close to the SVP architecture. We only selected the most indispensable features from C++ to be a part of SVPL and added vector constructs and capabilities in order to support the Simple Vector Processor architecture.

The exact syntax of this new programming language was specified using the EBNF formal description language. In the next chapters we will demonstrate that this SVPL specification is sufficient for implementation of a broad series of DSP algorithms.

Before we can start programming in SVPL we still have to develop a library that supports this definition and adds additional features to make debugging and evaluating SVPL programs easier.
Chapter 4

The SVP Interpreter

4.1 Introduction

In the previous chapter we discussed all functionality the SVP language needs to contain. In this chapter we take a look at the library that supports this language in a C++ environment.

Since this document's purpose is to discuss DSP algorithms, we will not discuss all implementation details of the SVPL library. First we take a look at the library's structure.

To make it easier to design, debug and evaluate SVPL programs, the SVPL library provides additional functionality that is not a part of the SVP language definition. Here is a brief summary of this functionality:

- An easy way to make partial or complete memory dumps.
- An output stream for debugging purposes.
- Processor statistics, implemented by the user or supplied by the default implementation.
- Extensive tracing information when using the default processor statistics implementation.

4.2 Library layout

To get an overall view of the SVPL library a simplified class diagram is included in figure 4.1. For the complete class diagram, please refer to appendix B.

When you take a look at the class diagram, you will easily recognize the three different memories, the four data types, their common base classes and interfaces and the way all these interact.

The other objects will be treated in sections 4.6 and 4.7.
4.3 Start using the library

The SVP library is a collection of C++ headers and code. There is only one public interface, the SVP.h file and it is the only one you should include in SVP enabled C++ code.

```cpp
#include "SVP.h"
```

All constructs included in the library are contained within the SVP namespace. So you could either enable this namespace using:

```cpp
using namespace SVP;
```

or you could prefix all SVP objects and commands by 'SVP::'. The latter is however not recommended as the resulting programs will not comply with the SVPL syntax.

4.3.1 Initializing the library

Before you can use the objects and types included in the SVP library, you need to initialize this library. This is done by the following statement:

```cpp
SVP::initSVP("inputfile.txt", "output.txt", 16);
```

The first two parameters define the files that contain the input and output streams. The third parameter sets the size of the vectors. This vector size is also stored in SVP::VectorSize, a globally accessible variable of the Integer type (see below) and cannot be changed during a simulation.
4.4 First example

During initialization the input memory is read from file and the random access and output memories are created as well as the debug and processor statistics objects that we will discuss further on in this chapter.

As we will see in a few pages, the initialization can take one extra parameter for advanced customization of the library.

The input and output file have a very simple formatting. All input values need to be of SCALAR_TYPE, in the standard implementation this is the C++ double type. There should be no empty lines, not even the last one. An example:

0.32  
158.5 0  
10  
53.54  
0.3654

All output files generated by the SVPL library, can immediately be used as input for another SVPL program.

4.3.2 Stopping the library - finalization

When you want to stop using the SVP library, you should call:

SVP::finalizeSVP();

This destroys all library objects and the output file is stored and closed. Note that you should never use any SVP command after calling this function. This will lead to unexpected behavior and might even cause your program to crash. Note that for the sake of performance and readability of the code, there are no checks done to make sure the library is not misused in this manner.

4.4 First example

Now that we have seen how to include and initialize the library, we can take a look at this very simple example to get a better understanding.

/ *  
* A very simple Vector-level pass-through filter:  
* input goes straight to output.  
* /

#include "SVP.h"

int main() {

SVP::initSVP("test.txt", "output.txt", 8);  
// From this point on only library constructs
IM.dump();

Vector v;
Address a = 0;

while (a < IM.size()) {
    v.load(IM, a);
    v.store(OM, a);
    a = a + VectorSize;
}

OM.dump();

// No more library constructs allowed
SVP::finalizeSVP();

4.5 Using the data types

The SVP namespace contains a lot of classes, but the provided data types are the only ones you can instantiate, namely these data types: Scalar, Vector, Integer, and Address. They all have been discussed in the previous chapter and it should be clear what their purpose is.

If you want to create a SVPL valid program, you cannot use any other than the above data types (this includes the standard C++ types like bool or int). These data types are not supported by the SVP and as for the SVP library, it does not support proper statistical information gathering and conversion from or to the above, approved types. This means that you will be able to create programs using a mixture of SVPL types and standard C++ types, but that the statistical information (see further on) will certainly be incorrect.

4.6 Additional Features

In this section we will show all objects that are made available through the SVPL library and illustrate their purpose together with an example of their use.

4.6.1 I/O Facilities

Included in the SVP namespace are the three memories: the IM, OM and RAM objects. We did discuss them in the previous chapters. On top of the functionality we have seen, those objects also provide a dump() member function that sends the entire memory or only a specific range of it to the standard output in a formatted way. This is mainly useful for debugging purposes and to avoid that you always need to open the output file in order to see the results:

IM.dump();
4.6 Additional Features

might result in something like this:

```
---- Output Memory Dump [vecsize=4 | memsize=15] ----

[#0]  0.75  1   1.5  2.3
[#4]  2   1.25  0.5  3
[#8]  0.75  1   1.5  1.7
[#12] 2   1.25  1.25
```

The memory content is formatted in rows with 'vectorsize' elements. The address of the first element of every row is indicated between square brackets.

It is also possible to supply a range of memory locations instead of dumping the entire memory.

```
void dump(Integer start, Integer length);
```

4.6.2 Global constants

There is only one global constant: SVP::VectorSize. It can be used in the algorithms you design. You will find out that almost every algorithm you design will make use of this variable.

4.6.3 Debugging

The SVPL library also supplies an instance of a special output stream: SVP::debug. This stream can be used in the same way you would use the cout stream from the standard library (in <iostream.h>):

```
Address a;
Vector v;
a = 54;
v.load(IM, a);

dump << "Vector read at position " << a << " : " << v << endl;
```

Result could be something like:

```
Vector read at position #54 : [1.2, 5.31, 2, 8, 4.5]
```

Of course the debug stream has more options than cout. In fact, debug is a combination of cout and an ofstream. These are the four options that can be set on the debug stream:

```
void enable(bool) With this method you can turn the gathering of debug information on or off.
void setFile(std::string) allows you to select a filename to which all debugging information should be written.
voidToFile(bool) enables or disables the writing of debug info to file. A filename must be set with the setFile method before the call to this method, or nothing will be written to file at all.
```
void toScreen(bool) enables and disables sending debug info to the standard output (cout).

The debug stream was originally developed to be used in the statistical unit that we will look into in the next section, but as an extension, you can use it anywhere in your SVP program.

4.7 Statistics

The statistical unit has been added to offer an easy and transparent way to collect usage information of the Simple Vector Processor and its individual units. The collected absolute measurements as well as computed aggregate values are of major importance when studying the real performance of implemented algorithms. While it is rather easy to calculate theoretical performance of an algorithm, the actual performance is much more difficult to compute.

Unlike the parts of the SVPL library that we have already discussed, the statistical unit is a rather experimental module, but it is also the most interesting part. As there are lots of applications for SVP algorithms and lots of reasons to study them, the desired statistics may vary widely, depending on the application. While the user is not allowed to add operations, data types or any other form of functionality to the library, he can create his own statistical unit by extending the supplied default unit or by creating one from scratch.

The heart of the statistical unit is a very limited interface called Processor-Statistics and the user is obliged to implement this interface when creating his own statistical unit.

Of course there is no need to implement your own statistical unit. A default implementation is supplied so that you don’t necessarily need to dig into the details of the vector processor. You should note that the interface does not define which statistics can be generated, only the way the library supplies operational signals.

4.7.1 Usage

The SVP library exports two functions to toggle the gathering of statistical information. It might be interesting to enable statistics only on a specific part of your algorithm in order to optimize that part separately:

bool enableStats();
bool disableStats();

To display the statistics generated by the ProcessorStatistics implementation, you simply call printStats():

void printStats();
This function is also called by `finalizeSVP()`, so you won’t need to call `printStats()` if you only want to look at the final statistics.

4.7.2 Custom Implementation: Design

In this subsection we will describe in detail how the end user can create his own statistical unit by creating an implementation of the `ProcessorStatistics` interface. In the next subsection we will give a detailed overview of the capabilities of the default `ProcessorStatistics` implementation: `BasicProcessorStatistics`.

There are three different pure virtual functions that the interface requires to be implemented:

- `virtual void print() = 0`
- `virtual void reset() = 0`
- `virtual void signal(...)`

Both the `print()` and `reset()` functions serve for obvious purposes and are completely implementation dependent. As they don’t require any parameters, we will not discuss them any further.

The actual heart of the statistical unit is the `signal` function. You should never call this, or any other function of the statistical unit directly, including `print()` and `reset()`. They are currently not hidden to the user because of the development state of the library. Of course they can be completely protected by defining a lot of friend dependencies.

The `signal` function is called by every imaginable operation on one of the predefined types (Scalar, Vector, Integer, and Address). Together with this call some flags indicating the nature of the operation performed are passed on to the statistical unit as well as the memory addresses of all variables accessed by that operation and an indication of whether they are being written to or not:

```cpp
virtual void signal(const int flags,
const ProcessorDataType* op1 = 0,
const bool writeOp1 = false,
const ProcessorDataType* op2 = 0,
const bool writeOp2 = false,
const ProcessorDataType* op3 = 0,
const bool writeOp3 = false
) = 0;
```

The enumeration `FlagType` contains the different signals that can be send by the SVP library. The signals are stored as an integer, all with a different bit set to 1. This makes it possible to combine as much signals as necessary by simple bitwise addition.
enum FlagType
{
    // These define the object(s) of operation
    NONE = 0x00,
    SCALAR = 0x01,
    VECTOR = 0x02,
    ADDRESS = 0x04,
    INTEGER = 0x08,

    // These define the nature of the operation
    MATH = 0x10, // +, -, *, /,
            // <, >, ==, !=, <=, =>
    MEMORY = 0x20,
    VECTORSHUFFLE = 0x40, // Shuffle
    VECTORSHIFT = 0x80,  // Shift
    VECTORMODIFIER = 0xC0,  // Shuffle OR shift
                   // (0xC0 = 0x40 | 0x80 )
    ASSIGNMENT = 0x100
};

Example

A typical call of the signal function is illustrated below with this addition operator of the Vector type.

Vector Vector::operator+ (const Scalar &s) const
{
    Vector tmp;

    fStats->signal(
        ProcessorStatistics::VECTOR
        | ProcessorStatistics::MATH,
        this, false,
        (ProcessorDataType*)&s, false,
        &tmp, true
    );

    bool oldStatState = fStats->enable(false);

    for (int i = 0; i < fVectorSize.fContent; i++) {
        tmp.fContent[i] = (this->fContent[i] + s);
    }

    fStats->enable(oldStatState);
    return tmp;
}

Clearly, both operands (this and s) are not being written to. Only the new resulting variable tmp is written to. You can also notice that using fStats->enable it is avoided that internal operations, in this case Scalar additions, are included in the statistics when this is not desirable.
As a final note, you should know that variables of the ProcessorDataType type are distinguished by an internal ID, not by memory location. The latter would require a rather complicated garbage collector, which does certainly not fall within the scope of this document.

You can find some additional information on the interface in the ProcessorStatistics.h header in appendix C.1.

4.7.3 Custom Implementation: Usage

When you initialize the SVP library, a BasicProcessorStatistics object is automatically created. If you implemented your own statistical unit, you can replace this default object by your own instantiation by passing a reference as fourth parameter to the SVP::initSVP function:

```c
SVP::initSVP("inputfile.txt", "output.txt", 3,
              new PersonalProcessorStatistics() );
```

4.7.4 Custom Implementation: Processor Schedules

While the kind of statistics you want to gather with your custom statistical unit is in no way defined, it is very likely that you want to calculate the number of processor cycles used by your SVP programs.

Most modern processors manage to execute at least some instructions in parallel. When programming in a sequential programming language (like C++, Java or our SVPL), this kind of parallelism is either managed by the compiler or by an intelligent scheduler in the processor. Since the SVPL library, at its level, can be seen as an interpreter, you will have to implement your own scheduling algorithm in order take advantage of this parallelism.

Since we did not define the kind of parallelism a SVP might support, you are totally free in designing your own scheduling algorithm. The default IBasicProcessorStatistics implements such a scheduler and the exact algorithm thereof is explained below in subsection 4.7.5.

For now you should remember that the scheduling algorithm can have a great impact on the actual performance of your algorithm and that specific code optimizations might be needed depending on that algorithm. We will illustrate the impact of the scheduler by looking at the following excerpt of a SVPL program with indicated line numbers:

(0) a = 7;
(1) s.load(IM, a);
(2) t = s + 1;
(3) p = q + r;
(4) q = m + u;
For the two different scheduling algorithms below we suppose that the SVP can use all its units (as shown in the component diagram 4.2) exactly once in a cycle. The scheduler included in BasicProcessorStatistics makes the same assumptions, and therefore these are explained in more detail in subsection 4.7.5.

![SVP Layout Diagram]

Figure 4.2: SVP Layout

This simple scheduler, as implemented by BasicProcessorStatistics, could create the following schedule:

```c
// cycle 0
a = 7;

// cycle 1: new cycle because the 'a' read in line 1 is being written to in line 0
s.load(IM,a);

// cycle 2: new cycle because the 's' read in line 2 is being written to in line 1
t = s + 1;

// cycle 3: new cycle because the scalar math unit is already used in line 2
p = q + r;

// cycle 4: new cycle because the scalar math unit is already used in line 3
q = m + u;
```

Total cycles: 5

A more intelligent scheduler that is looking more than one instruction ahead could create the following schedule by reordering statements to make optimal
use of all the processor units. This reordering of statements is only possible as long as no so-called 'data-hazards'\(^1\) occur.

```
// cycle 0
a = 7;
p = q + r;

// cycle 1
s.load(IM,a);
q = m + u;

// cycle 2
t = s + 1;
```

Total cycles: 3

It is not hard to see that while the second schedule only takes 60% of the time the first schedule needs, both execute exactly the same algorithm and yield the exact same result. Of course this example piece of code was selected to have few dependent instructions, but still it illustrates how the scheduler algorithms can have dramatic impact on processor and algorithm efficiency.

If you don't have an intelligent scheduler as mentioned above, you can of course still get the same efficiency. In that case, it is however the programmers task to reorder the statements in his program for them to be scheduled in the most optimal way by the simple scheduler. It is such a basic scheduler that is implemented in the default ProcessorStatistics implementation that we will discuss in detail in the next subsection.

### 4.7.5 Default implementation: BasicProcessorStatistics

The SVP library provides one implementation of the ProcessorStatistics interface, simply called BasicProcessorStatistics. This implementation provides a lot of useful statistics for analyzing and optimizing algorithms with the SVP library. These statistics are displayed on standard output when you call the `SVP::finalizeSVP()` function.

**Generated statistics**

The BasicProcessorStatistics is a basic implementation of a superscalar processor scheduler. It keeps track of all operations that are executed on the SVP and the variables that are accessed by these operations. The scheduler tries to fit as much operations as possible into a single processor cycle, while maintaining program correctness. This includes tracking the availability of processor components and data dependencies.

\(^1\)A data hazard a term from processor pipelining and should in this context be interpreted as follows: A data hazard occurs whenever there is a dependence between instructions, and reordering them would change the order of access to an operand. More information on data hazards can also be found in [HP90].
As we already indicated in the previous section, the default scheduler implementation looks only one instruction ahead and is therefore not capable of reordering instructions automatically. It is the programmers responsibility to do so in order to create algorithms that can be optimally scheduled.

The SVP processor is divided into different independent units as illustrated in figure 4.2.

This is a summary of the rules on which the scheduler is based:

- Processor Unit Rules:
  - All below items can be executed in one processor cycle:
    - One scalar OR one vector can be loaded OR stored from/to memory.
    - One Integer or Address calculation can be executed.
    - One Scalar calculation can be executed.
    - One Vector calculation can be executed.
    - The Vector calculation can be preceded by a Vector shift and/or shuffle operation.

- Data dependency rules:
  Sometimes two operations can be done in one cycle and sometimes it even takes two cycles to execute a single operation. To understand the rules below, you must know that every cycle is divided into three consecutive stages: the ‘data read where the operands of the operation are read, the ‘execute where the actual operation is performed, and the ‘data write where the result is stored.

  - Variables can be read by multiple operations in one cycle simultaneously (= concurrent read).
  - A variable can be written to only once in a cycle (= exclusive write).
  - If in the SVPL program a variable is read in one statement and written to in a subsequent operation, this can be done in one cycle, since the read is executed at the start of the cycle (data read) and the write is only performed at the end (data write).
  - The other way around, if a variable is written to in the SVPL program, a new cycle must be started before a subsequent operation can read the updated value. Otherwise the old value of that variable would be used instead of the new, updated one.

Whenever a newly signaled operation cannot be fit into the current cycle, that cycle is flushed and a new cycle commences. Flushed cycles are divided into three categories: the ones that make use of the vector unit, the ones that do not make use of the vector unit and only perform Integer or Scalar math and the rest, cycles that do use the vector unit, but contain memory access. With these three categories all necessary statistics can be calculated. An example of such statistics as provided by BasicProcessorStatistics is illustrated below:
### Basic Processor Stats

<table>
<thead>
<tr>
<th></th>
<th>VECTOR</th>
<th>SCALAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>26100</td>
<td>0</td>
</tr>
<tr>
<td>Math</td>
<td>19800</td>
<td>0</td>
</tr>
<tr>
<td>Modify</td>
<td>39600</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>88500</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>INTEGER</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>53100</td>
<td>53101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>SCHEDULER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total signals</td>
<td>213301</td>
</tr>
<tr>
<td>Total operations</td>
<td>122401  (100 %)</td>
</tr>
<tr>
<td>Total cycles</td>
<td>78301</td>
</tr>
<tr>
<td>Total unit usage</td>
<td>39 %</td>
</tr>
<tr>
<td>Vector unit usage: 25%</td>
<td>(58500 lost cycles)</td>
</tr>
<tr>
<td>(excl. Int ops): 43%</td>
<td>(26100 lost cycles)</td>
</tr>
</tbody>
</table>

The upper half of the output breaks down all operations (note that this is not the same as cycles) into four different categories, based on the type of values that is operated on. For Vector and Scalar operations, some sub categories are displayed as well.

The lower half provides us with some more interesting information. It shows how many times the signal function has been called, the number of operations that have been executed and the number of cycles it took to fit in all these operations. Total unit usage gives an indication of the amount of parallelism that could be introduced. For this metric the use of these four processor units is considered: Vector, Scalar and Integer math unit and the memory unit. The full 100% usage is only achieved when all four units are in use during every cycle. 25% unit usage is the worst-case scenario, since only one unit is used per cycle. Less usage is of course impossible since a cycle cannot exist without at least one unit being used.

Finally it also shows how much use is made of the vector mathematical unit, both including and excluding Integer and Address operations. It makes sense to make this last distinction since Integer and Address operations are mostly control logic operations and they do not directly contribute to the calculation of output values. Between brackets the number of cycles that make no use of the vector math unit is indicated, again including and excluding Integer and Address operations.

How all these statistics can help us creating efficient SVP algorithms is illustrated in the next chapter, *Programming in SVPL*.

**Tracing capabilities**

Whenever the debug stream is enabled, the statistical unit provides a lot of useful tracing information on the standard output or to file, depending on the
debug stream settings. This information can be used to optimize your algorithms or to track down bugs without the need for stuffing your code with your own debug information.

Note that the tracing information is provided by the implementation of the ProcessorStatistics interface. The default BasicProcessorStatistics class is designed to provide a lot of useful information, but if you want to design your own ProcessorStatistics implementation you will need to provide this information yourself.

This is a very short example of the debug output provided by BasicProcessorStatistics: 7

SVP:Debugmode ON
SVP:Debugmode FILE SET debug.txt
SVP:Debugmode FILE ON
SVP:Signal: [........] ADDR MATH
SVP:Signal: [.........] VEC ASSIGN
SVP:Signal: [.........] VEC MEM
SVP:Signal: [M......] ADDR MATH -> PURGE NORESOURCE LOST INTEGER
SVP:Signal: [.........] ADDR ASSIGN
SVP:Signal: [.........] VEC MATH
SVP:Signal: [.........V....] SCAL MEM -> PURGE UNFREE
SVP:Signal: [M........] VEC SHIFT -> PURGE UNFREE LOST
SVP:Signal: [.........i...] VEC ASSIGN
SVP:Signal: [.........i...] VEC SHIFT -> PURGE NORESOURCE LOST
SVP:Signal: [.........i...] VEC MATH
SVP:Signal: [.....Vi...] ADDR MATH
SVP:Signal: [....V...i] SCAL MEM -> PURGE UNFREE
SVP:Signal: [M........] VEC SHIFT -> PURGE UNFREE LOST
SVP:Debugmode FILE OFF
SVP:Debugmode OFF

The lines starting with SVP:Debugmode speak for them selves. We will only take a closer look at lines the starting with SVP:Signal. For every signal that is received by the statistical unit, one single line is printed. The first part [........] indicates the usage of the processor units before the new signal is processed. When the processor units are fully used, this indicator would look like [MISViu]. A letter means that the unit is used, a point that it is currently unused:

M - The memory unit
I - The Integer/Address math unit
S - The Scalar math unit
V - The Vector math unit
i - The Vector Shift logic
u - The Vector Shuffle logic
4.7 Statistics

Subsequently, the nature of the new signal is indicated: the type that is operated on (VEC, SCAL, INT, ADDR) and the type of operation (MEM, MATH, SHIFT, SHUFFLE, ASSIGN). Note that ASSIGN is a valid signal, but that it is not counted as an operation. The assignment signal is only to indicate destination variables, so that they are included in the write access list of the current cycle.

Finally, there is an optional part that is only included when the new signal cannot be fit into the current cycle. PURGE indicates that the current cycle will be purged and that this new signal will be included in a new cycle. NORESOURCE and UNFREE indicate whether the cycle was purged because of a lack of processor resource (unit already used) or because of a dependency between values of two or more operations. LOST indicates that the purged cycle is counted as a lost cycle\(^2\) and INTEGER indicates that the cycle was only dedicated to Integer or Address mathematics, which is supposed to be part of the control logic of the executing algorithm.

All this information should be sufficient to debug most SVPL programs, but of course you can still write your own information to the debug stream.

Note on Custom ProcessorStatistics Implementation

When you create your own implementation of the ProcessorStatistics interface, you should remember that the above tracing information is provided by the default BasisProcessorStatistics implementation and that if you will need to implement this kind of functionality yourself by writing to the debug stream if you require such.

4.7.6 Known Issues

Unaccounted operations

As already indicated in chapter 3, it is impossible make sure that every SVPL program only includes Integer operations and thereby that not a single C++ int and int operation is included. Every integer literal will be of the standard int type, and if two of those literals are operated on, that operator is also a int operator (e.g. \(s = 5 + 7\) with \(s\) an Integer). Those operators cannot be included into the statistics but luckily that is no real issue. Even the simplest compiler would optimize this code by executing the operator at compile-time and replacing it by the result \((s = 12)\).

Invalid code

A lot of effort has been put into making it very hard to write illegal SVPL programs and enforcing the programmer to use only valid SVP objects and constructs. One of the tricks to do so was making the SVP library classes as incompatible with standard C++ types as possible. This is however not possible for 100% because of the tight connection between statements like \(\text{while}\) and \(\text{if}\) and the native C++ data type bool.

\(^{2}\)Remember that a cycle is called a loss when the Vector math unit was left unused.
Because of the way we implemented the SVP library, using C++ as a foundation, it is impossible to enforce the exact SVPL syntax. It is therefore possible that a programmer writes illegal code without him being aware of this. As mentioned earlier it cannot be guaranteed that statistics from the statistical unit (both default and custom implementation) are correct.

There is no solution to this problem other then trying hard to write valid SVPL programs. When you make sure that you don’t use any other than the valid data types and statements, chances are rather small that you create an illegal program (although it is not impossible).

4.8 Conclusion

The definition of the Simple Vector Processor Language alone in chapter 3 was not sufficient to start programming in SVPL. In this chapter we developed a C++ library that makes it possible to actual create and run SVPL programs as it acts as an SVPL interpreter. Moreover, it facilitates the creation process and the performance analysis of such programs because of its additional features.

The tracing and memory dump capabilities as well as the advanced statistical unit are perfectly suited to debug, analyze and enhance SVPL programs. In the next chapter, we explore this process in detail by means of a simple example.
Chapter 5

Programming in SVPL

In this chapter, we will cover a few topics related to the writing of programs in SVPL: some basic rules you will need to take into account and some 'best practices' you are encouraged to adopt. Finally we will take an in-depth look at a simple SVPL program, its output and statistics and how to use those in order to work towards an optimized implementation.

5.1 The first and last output values

A very important rule which will always apply, is that algorithms can have a certain, limited 'startup' and 'shutdown' period in which the output values do not need to be correct, this is often called the startup and shutdown noise.

In most case we will not try to avoid this startup and shutdown noise because might lead to a considerable overhead in the programming logic and the number of noise values can almost always be neglected. Moreover, it is common practice in describing DSP algorithms.

In practice, this usually implies that during startup, the algorithms may presume some preceding zeros in front of the actual input stream. Another option is simply to let the first value from the input stream coincide with the first value needed by the algorithm. It depends on the algorithm which solutions produces the least bogus output values and both solutions are permitted. The same applies to the last calculations of an algorithm (shutdown noise) where one can simply stop the calculations when the end of input is reached or one can append enough zeros to finish the last calculation.

5.2 Unsupported language constructs

Here you find a short list of language constructs that are not supported by the SVPL library. While they may compile correctly, the generated SVP statistics will not be correct. These constructions are not in the EBNF notation of SVPL, but are mentioned here because one could easily believe that they are permitted and thereby lead to very common mistakes.
Initialization with a value

You cannot or at least should not initialize a variable with a value, instead you should use a separate assignment in case you want to give a variable a default value other than 0:

```c
// Incorrect
Integer i = 5;
Address a = i * 9;
```

```c
// Correct
Integer k; k = 6;
Address a; a = i * 9;
```

While statement (2) will not even compile, statement (1) will, but is also incorrect use of SVPL. (1) and (2) need are executed with constructors `Integer(long)` and `Address(long)` respectively. For several not so obvious issues, internal to the SVP library, these constructors cannot be accounted by the statistical unit (no signals are raised).

A simple initialization with a constant like in (1) will not often raise a huge problem, but this depends on the implementation of the statistical unit. At least the default `BasicProcessorStatistics` will not be fooled. It gets worse when a variable is initialized with a mathematical expression like in (2). Not only will the assignment not be counted, it will also be impossible to detect the correct data dependencies. This might lead to invalid processor schedules.

In order to avoid these problems, the constructors for `Address` and `Scalar` are simply not accessible, but the `Integer(long)` constructor could not be hidden from the user. This yields that statement (2) will give a compiler error, but that (1) will not, although it is also incorrect.

absence of the for loop

As you might have noticed, the specification of SVPL does not include a construct similar to the `for` loop. There are several good reason not allow such a construct and these are explained in this section.

First of all since initialization like `Integer i = 5` are not allowed, the `for` loop would also need to comply with this rule. Therefore, instead of writing:

```c
for (Integer i = 0; i < c; i++)
```

you would need to write

```c
Integer i;
for (i = 0; i < c; i++)
```

which makes code much less readable and the `for` loop much less convenient in use.
5.3 Block-average filter

Secondly, permitting the use of for loops will encourage the programmer to use loop variables that are not necessary and thereby lead to inefficient code. It almost never a good idea to use dedicated loop variables since they often conceal a lot of overhead, like in the following example.

```java
Address base_address = 543;

for (Integer i = 0; i < 1000; i++) {
    s.load(IM, base_address + i);

    //... use s
}
```

This is of course a correct piece of code, but it can be written much more efficiently. In the above code, two additions are performed in every loop iteration, while only one would be sufficient:

```java
Address base_address = 543;

Address tmp; tmp = base_address;
Address max; max = base_address + 1000;

while (tmp < max) {
    s.load(IM, tmp);
    tmp++;

    //... use s
}
```

As can be seen in the code above, almost half of the additions could be avoided at the expense of introducing two extra variables. You could even omit the `tmp` variable if the original value of `base_address` is not longer needed after the loop:

```java
Address base_address = 543;
Address max; max = base_address + 1000;

while (base_address < max) {
    s.load(IM, base_address);
    base_address++;

    //... use s
}
```

5.3 Block-average filter

The goal of this section is to learn how to create a SVPL program from a mathematical definition and how to use the statistics and tracing information provided by the default statistical unit to optimize that program. Therefore we will study a simple example algorithm.
The algorithm we will look at is the block-average filter. Such a filter takes
the average of every \( n \) values from an input stream and writes this average to
the output stream. Consequently the data rate of the output stream is only \( \frac{1}{n} \)
of the input streams data rate. The parameter \( n \) is called the filter window.

\[
B_{avg}(n, A)(i) = \frac{1}{n} \left( \sum_{j: 0 \leq j < n} A((i \times n) - j) \right) \tag{5.1}
\]

5.3.1 Initial Implementation

In order to keep the example simple, we will not implement the generic block-
average filter, but only the specific filter with window size 2. A full vector of
output values \( \overline{B}(i) \), can then be defined as:

\[
\overline{B}(i) = \\
\begin{bmatrix}
    Average_2(A)(i) \\
    \ldots \\
    Average_2(A)(i + P - 1)
\end{bmatrix}
\]

\[
= \\
\begin{bmatrix}
    \frac{1}{2} \left( \sum_{j: 0 \leq j < 2} A((i \times 2) - j) \right) \\
    \ldots \\
    \frac{1}{2} \left( \sum_{j: 0 \leq j < 2} A((i + P - 1) \times 2) - j) \right)
\end{bmatrix}
\tag{5.2}
\]

A first, naive implementation of the above could be:

```cpp
SVP::initSVP("input.txt", "output.txt", 4);
IM.dump();

Vector v1, v2, v3;
Address a1, a2;

while (a1 < IM.size()) {
    v1.loadWithStride(IM, a1, 2);
    v2.loadWithStride(IM, a1 + 1, 2);
    v3 = (v1 + v2) / 2;
    v3.store(OM, a2);
    a1 = a1 + (VectorSize * 2);
    a2 = a2 + VectorSize;
}

OM.dump();
SVP::finalizeSVP();
```

*Note: From now on we will remove the library initialization and finalization commands as well as memory dump and debug statements from the program listings in this document in order to avoid distraction from the more relevant code.*
5.3 Block-average filter

We will use a very short test input file, containing the following numbers, all separated by a linefeed: 5, 9, 6, 4, 8, 5, 1 and 0. With the chosen vector size (4) only one run of the main loop will occur. A run of our first program when fed with this input data will generate the following output:

---- Input Memory Dump [vecsize=4 | memsize=8 | range=0-7] ----

[#0]  
5     9     6     4

[#4] 
8     5     1     0

SVP: Debugmode ON
SVP: Signal: [.....] ADDR MATH
SVP: Signal: [.I....] VEC MEM
SVP: Signal: [MI.....] ADDR MATH --> PURGE NORESOURCE LOST INTEGER
SVP: Signal: [.I....] VEC MEM --> PURGE UNFREE LOST
SVP: Signal: [M.....] VEC MATH --> PURGE UNFREE LOST
SVP: Signal: [.V....] VEC MATH --> PURGE NORESOURCE
SVP: Signal: [.V....] VEC ASSIGN
SVP: Signal: [.V....] VEC MEM --> PURGE UNFREE
SVP: Signal: [M.....] INT MATH
SVP: Signal: [MI.....] ADDR MATH --> PURGE NORESOURCE LOST INTEGER
SVP: Signal: [.I....] ADDR ASSIGN
SVP: Signal: [.I....] ADDR MATH --> PURGE NORESOURCE LOST INTEGER
SVP: Signal: [.I....] ADDR ASSIGN
SVP: Signal: [.I....] ADDR MATH --> PURGE NORESOURCE LOST INTEGER

---- Output Memory Dump [vecsize=4 | memsize=4 | range=0-3] ----

[#0]  
7     5     6.5   0.5

---- Basic Processor Stats ----

VECTOR
Memory ops: 3
Math ops: 2
Modify ops: 0
Total ops: 5

SCALAR
Memory ops: 0
Math ops: 0

INTEGER
Total ops: 1

ADDRESS
Total ops: 7

SCHEDULER
Total signals: 14
Total operations: 11 (100 %)
Total cycles: 9
Total unit usage: 31 %
Vector unit usage: 22% (6 lost cycles)
(excl. Int ops): 40% (2 lost cycles)

This was clearly not an optimal solution. Both from the schedules shown in the tracing information and from the aggregate statistics we can deduct that
we should at least try to rewrite our program for faster execution. Currently, the total unit usage is only 32%, a mere 7% higher than the absolute minimum.

Execution of the algorithm teaches us that the current algorithm can handle an average only one input sample per cycle, while using a vector size of 4! Of course, when increasing the vector size, this metric will improve, but still we should be able to improve this metric without change the vector size.

### 5.3.2 Optimization

Now, let’s try to optimize the algorithm step by step. It is not a good idea to try optimizing an SVP algorithm based on statistics that were generated with only a very small set of input data. The initialization and finalization instructions would weight on the final results in a very unbalanced way. We will take the same input file of 128 numbers in the following examples.

These are the statistics of the initial program, when applied on the 128-number input file. We will compare all future optimizations with these unoptimized statistics.

--- Basic Processor Stats ---

<table>
<thead>
<tr>
<th>VECToR</th>
<th>SCALAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory ops: 48</td>
<td>Memory ops: 0</td>
</tr>
<tr>
<td>Math ops: 32</td>
<td>Math ops: 0</td>
</tr>
<tr>
<td>Modify ops: 0</td>
<td></td>
</tr>
<tr>
<td>Total ops: 80</td>
<td>Total ops: 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INTEGER</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total ops: 16</td>
<td>Total ops: 97</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCHEDULER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total signals : 209</td>
</tr>
<tr>
<td>Total operations : 161</td>
</tr>
<tr>
<td>Total cycles : 129</td>
</tr>
<tr>
<td>Total unit usage : 31%</td>
</tr>
<tr>
<td>Vector unit usage: 24%</td>
</tr>
<tr>
<td>(excl. Int ops): 49%</td>
</tr>
</tbody>
</table>

One should always avoid loop independent instructions inside a loop. These are instructions that evaluate to the same result, independent of the state of the loop and they can easily be calculated outside that loop. (This is of course not specific to vector programming) While the instruction might look like a small effort, its weight on final performance might be larger than expected because it is executed in every loop iteration. An example of such a loop independent instruction is `VectorSize * 2`. This is what our program looks like when the statement is put out of the loop:

```c
Vector v1, v2, v3;
Address a1, a2;
Integer DblVectorSize; DblVectorSize = VectorSize * 2;
```
while (a1 < IM.size()) {
    v1.loadWithStride(IM, a1, 2);
    v2.loadWithStride(IM, a1 + 1, 2);
    v3 = (v1 + v2) / 2;
    a1 = a1 + (Db1VectorSize);
    v3.store(IM, a2);
    a2 = a2 + VectorSize;
}

And executing this program with our 128-samples test data generates the following statistics:

```
---- Basic Processor Stats ----

VECTOR                          SCALAR
Memory ops: 48                  Memory ops: 0
Math ops: 32                     Math ops: 0
Modify ops: 0                    Total ops: 0
Total ops: 80                    Total ops: 0

INTEGER                          ADDRESS
Total ops: 2                     Total ops: 97

SCHEDULER
Total signals : 195              (100 %)
Total operations: 146            (65 lost cycles)
Total cycles : 98                (32 lost cycles)
Total unit usage : 37 %           (excl. Int ops): 49%
Vector unit usage: 32%           (65 lost cycles)

We eliminated 31 cycles out of 129 (24%) by simply bringing that instruction out of the loop!

From the tracing info we get that a lot of cycles are lost because of a used unit, while only one unit is used in these cycles. If we reorder the statements so that we avoid using the same unit back to back, parallelism can be introduced and again cycles will be saved:

Vector v1, v2, v3;
Address a1, a2, a3;
Integer DblVectorSize; DblVectorSize = VectorSize * 2;

v1.loadWithStride(IM, a1, 2);
    a3 = a1 + 1;
    a1 = a1 + DblVectorSize;
    v2.loadWithStride(IM, a3, 2);
    v3 = (v1 + v2);
    v3 = v3 / 2;
```
while (a1 < IM.size()) {
    v1.loadWithStride(IM, a1, 2);
    a3 = a1 + 1;
    v3.store(OM, a2);
    v2.loadWithStride(IM, a3, 2);
    a1 = a1 + DblVectorSize;
    v3 = (v1 + v2);
    a2 = a2 + VectorSize;
    v3 = v3 / 2;
}

v3.store(OM, a2);

As you can see, we reordered the statements inside the while loop. The v3.store(OM, a2) statement was moved up, because a3 = a1 + 1; and v2.loadWithStride(IM, a3, 2); could not be executed simultaneously (UNFREE). As a consequence, we had to unfold one step of the while loop and add a final v3.store(OM, a2) after the loop. Although this introduces additional program code, the execution time is once again highly reduced:

----- Basic Processor Stats -----

<table>
<thead>
<tr>
<th>VECTOR</th>
<th>SCALAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory ops: 48</td>
<td>Memory ops: 0</td>
</tr>
<tr>
<td>Math ops: 32</td>
<td>Math ops: 0</td>
</tr>
<tr>
<td>Modify ops: 0</td>
<td>Modify ops: 0</td>
</tr>
<tr>
<td>Total ops: 80</td>
<td>Total ops: 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INTEGER</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total ops: 2</td>
<td>Total ops: 110</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCHEDULER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total signals : 224</td>
</tr>
<tr>
<td>Total operations : 144 (100 %)</td>
</tr>
<tr>
<td>Total cycles : 66</td>
</tr>
<tr>
<td>Total unit usage : 55 %</td>
</tr>
<tr>
<td>Vector unit usage: 48% (33 lost cycles)</td>
</tr>
<tr>
<td>(excl. Int ops): 49% (32 lost cycles)</td>
</tr>
</tbody>
</table>

Total unit usage is now about 55%, meaning that on average, in every cycle a bit more than two units are used. If we compare this final result with the first statistics, we see that we eliminated only 17 operations (10.5%) while reducing execution time with 63 cycles (49 %, or halved) and increasing unit usage with 24 % (or doubled). As a final measure, the number of cycles per processed input sample is only 0.51.

5.3.3 Evaluation

It is safe to say that the above solution is the most optimal solution. Let’s verify this by taking a look at the generated schedules. For every iteration of the main loop, the following output is generated:
5.3 Block-average filter

SVP:Signal: [....V..] ADDR MATH
SVP:Signal: [..I.V..] VEC MEM
SVP:Signal: [MI.V..] ADDR MATH  \rightarrow PURGE NORESOURCE
SVP:Signal: [..I....] ADDR ASSIGN
SVP:Signal: [..I....] VEC MEM
SVP:Signal: [MI....] VEC MEM  \rightarrow PURGE NORESOURCE LOST
SVP:Signal: [MI....] ADDR MATH
SVP:Signal: [MI....] ADDR ASSIGN
SVP:Signal: [....V..] VEC MATH  \rightarrow PURGE UNFREE LOST
SVP:Signal: [....V..] VEC ASSIGN
SVP:Signal: [....V..] ADDR MATH
SVP:Signal: [..I.V..] ADDR ASSIGN
SVP:Signal: [..I.V..] VEC MATH  \rightarrow PURGE NORESOURCE
SVP:Signal: [....V..] VEC ASSIGN

As you can see, every loop iteration takes 4 cycles and these are the 4 corresponding schedules:

[MI.V..]
[MI....]
[MI....]
[.I.V..]

The Integer math unit is occupied all the time by address calculations, this means that the current instruction could not fit in less than those 4 cycles. The only way to reduce the number of processor cycles per iteration is to remove at least one of those address calculations. This is however impossible: There are 3 different addresses needed: two for the input vectors and one four the output vector. Those 3 addresses need to be calculated for every iteration, either by calculating them independently (like we did above) or by calculating one address based on an other. Both ways require at least 3 address calculations. The 4th address calculation is the loop condition $a1 < IM.size()$, which is indispensable for the program to finish. We can therefore conclude that the minimum amount of cycles per iteration has been reached.

Now that we have seen how statistics and tracing information can be very useful in optimizing SVPL programs, we can move on to the next part of this document: the analysis and optimized implementation of some very common DSP filter algorithms on the Simple Vector Processor.
Part II

Study of algorithms
Introduction

In chapters 6 and 7 we derive vectorized versions of some very common DSP algorithms. For each of these algorithms there already exist a lot of very efficient serial, scalar implementations, for specific DSP chips as well as for generic processors. It will however be a challenge to create efficient implementation for the Simple Vector Processor.

We will briefly discuss the working of those algorithms and then we will rewrite each algorithm in order to make it suitable for vector-based computation. Finally we create an implementation of each algorithm for the SVP, using the SVPL library. Our main goal will always be to optimize the use of processor resources and especially the use of the vector mathematical unit.

The best implementation of each algorithm will be the one that makes the vector math unit calculate an entire vector every cycle and where every single scalar calculation in every vector calculation is useful, not superfluous work like multiplication by 0, and no duplicated work.

In the end, the implementation we create will be executed on the SVPL library and the resulting statistics will be analyzed.

We will not specify the vector size and therefore the algorithm implementation cannot depend on this parameter. In some cases we will, however, indicate what the optimal vector size should be for a given algorithm and how much complexity is added when the vector size differs from this ideal size.

One of the main reasons to use a vector processor, is the ability to process data at a higher rate than the clock rate of the processor. We will therefore only design algorithms that read vectors from the input stream. Algorithms that only read scalar values can never process more than 1 input sample per cycle.

Performance metrics

For each algorithm, we discuss both the theoretical performance and the actual performance achieved by the SVPL implementation.

When discussing the theoretical performance of an algorithm, we only focus on the scalar computations, that is stream data manipulations, and ignore memory, addressing and control overhead. This overhead differs from DSP to DSP and most of it can be avoided by the use of techniques like zero-overhead looping and special addressing techniques and instructions. The overhead is however still taken into account when discussing the SVPL implementation of each algorithm.

Performance of a vectorized algorithm is always compared to the performance of the scalar one as this gives the best indication of performance loss due to vectorization. In the optimal case this loss is zero. The vectorized version of an algorithm then produces a vector of output values in the same time a scalar
algorithm produces a single output value. Speedup of the vectorized algorithm is expressed as the quotient of this ideal performance and the actual performance of the vectorized algorithm. The ideal speedup is thus 100%.

The SVPL implementations of vectorized algorithms are not compared to their scalar counterparts. Instead, performance is given by an absolute metric: the number of Cycles Per output Vector or CPV. The goal is to lower this metric as much as possible and the ideal value is 1, but performance is always limited by data dependencies and SVP resource availability.

We conclude with a few final remarks.

• The $+$ or $\times$ operators used on vectors indicate point-wise addition and multiplication.

• Unless indicated otherwise, a vector size of 32 scalar elements is used in for the generation of statistical information.

• In program listings, we will no longer show the library initialization and finalization statements.
Chapter 6

The Scaler

6.1 Definitions

In this chapter we design a vector implementation of the scaler. This scaler is a very common combination of three filters: the upsample, FIR and downsample filters. This combination is used for scaling digital signals. While all kinds of digital signals can be scaled, the scaler is most often used for resampling digital audio streams and for resizing digital images.

Resampling is a combination of interpolation and decimation with the purpose of changing the sampling rate by a rational factor. The scaler performs this resampling by sending the input stream through an upsample filter followed by a FIR filter and finally a downsample filter.

We first explain how the individual filters work. All three are linear filters that read an input stream and produce an output stream and they can be described as a function between the values of these input and output streams.

6.2 The input stream \( A \)

The input stream \( A \) is defined as an infinite sequence of scalar values:

\[
A(i) \quad (i > 0)
\]  

(6.1)

6.3 Downsample filter

We start with the easiest filter: the downsample filter. A downsample filter takes a continuous stream \( A \) of scalar values as input. The output stream of this filter contains only every \( l \)-th value from its input stream. \( l \) is referred to as the downsample ratio.
6.3.1 Definition of the downsample filter $DN$

Since most filters have one or more parameters that define the actual behavior of those filters, we will define a generic filter function for every filter. This filter function takes as parameters all values that define its output stream, including the input stream. We demonstrate this with the downsample filter.

A downsample filter's output stream is completely defined by its downsample ratio $l$ and its input stream $A$. This means that the generic downsample filter function $DN(l, A)$ defines all possible downsample filters and that it specifies exactly one downsample filter when $A$ and $l$ are defined.

As the output of a filter is a data stream, it can be indexed like the input stream. And as a single output value of a filter depends on that filter and on previous inputs, the behavior of the filter can be described using indices:

$$DN(l, A)(i) = A(li)$$  \hfill (6.2)

By using this notation, it is easy to define more specific downsample filters, e.g. the Odd and Even filters, which obviously only let every second value pass through.

$$Even(A)(i) = DN(2, A)(i) = A(2i)$$
$$Odd(A)(i) = DN(2, A)(i) = A(2i + 1)$$  \hfill (6.3)

6.3.2 Vectorization

In order to create an efficient implementation of the $DN$ filter, we want the algorithm to create a vector $\overline{DN}$ of $P$ output values at a time.

Notation: $P$ is the vector size

$$\overline{DN}(i) = \begin{bmatrix}
DN(l, A)(i) \\
\ldots \\
DN(l, A)(i + P - 1)
\end{bmatrix} = \begin{bmatrix}
A(il) \\
\ldots \\
A((i + P - 1)l)
\end{bmatrix}$$  \hfill (6.4)

Theoretical Performance

The performance analysis of the $DN$ filter is very simple. There are no data calculations performed, only input and output. These can be done one vector at a time, instead of one scalar at a time with the scalar implementation. Theoretical speedup is thus 100%.

SVPL Implementation

The SVPL implementation of the downsample filter is illustrated below. The algorithm uses loadWithStride to read every $l$-th value from the input stream. Considering the simplicity of the code, no further explanation is provided.
Remember from this chapter's introduction that from now on we will often omit the library initialization and finalization in order to avoid distraction from the more relevant code.

Address a1, a2;
Vector v1;
Integer l, m;
l = 5; // downsample coefficient
m = 1 * VectorSize;

while (a1 < IM.size()) {
    v1.loadWithStride(IM, a1, l);
    a1 = a1 + m;
    v1.store(OM, a2);
    a2 = a2 + VectorSize;
}

This implementation is optimal, it has a total unit usage of 41%. It cannot be improved because the Integer unit is used 100%. The CPV is 3, which, for a vector size of 32, means that the number of cycles per output sample is only 0.096. Yet this is 3 times slower than the potential theoretical speedup!

6.4 Upsample filter

6.4.1 Definition of the upsampled stream UP

Analogous to the downsample filter, the upsample filter is defined by the input stream and the upsample value:

\[ UP(k, A)(i) = \begin{cases} 
    A(i \text{ div } k) & i \text{ mod } k = 0 \\
    0 & i \text{ mod } k \neq 0
\end{cases} \quad (0 \leq i) \quad (6.5) \]

or written otherwise:

\[ UP(k, A)(qk + r) = \begin{cases} 
    A(q) & r = 0 \\
    0 & 0 < r < k
\end{cases} \quad (0 \leq q) \quad (6.6) \]

6.4.2 Remark

Note that the upsample filter generates a data stream from which the original input stream can still be deduced, we call such a filter a lossless filter. It would only take the appropriate downsample filter to reconstruct that original input stream.

On the other hand, the downsample filter simply drops a lot values from the input stream that cannot be recovered by applying another filter like the upsample filter. We call this kind of filter lossy.
6.4.3 Vectorization

Theoretical Performance

As with the downsample filter, the speedup of the vectorized upsample filter is 100%. No data calculations are done and input and output can be done by vectors instead of scalars.

SVPL implementation

This is the SVPL implementation of the upsample filter. Basically, the algorithm reads consecutive values from the input stream, stores them at the correct, interleaved locations in the output stream and fills the gaps between two input values with zeros. (One could argue that this 'gap filling' is not necessary if memory is initialized with zero values. This would obviously yield a much higher efficiency.)

Address a1, a2;
Vector v1, zero;

Integer k; k = 5; // upsample coefficient
Integer l; l = k - 1;
Address d;

zero = Vector(0);

while (a1 < IM.size()) {
    a2 = a1 * k;
    v1.load(IM, a1);
    d = a2 + l;
    // Store values from input
    v1.storeWithStride(OM, a2, k);
    // Store intermediate zeros
    while (a2 < d) {
        a2++;
        zero.storeWithStride(OM, a2, k);
    }
    a1 = a1 + VectorSize;
}

Total unit usage of this implementation is 36% and CPV is 2,604 at upsample value 5. This result is a little better than with the downsample filter because the most inner loop of the upsample program only uses 2 cycles where the loop of the downsample program always needs 3 cycles to complete. This also means that CPV decreases slightly with increasing l and converges to 2:
6.5 A FIR Filter

The FIR filter is a linear combination of a fixed, finite number of samples from the input signal. It moves a window of length \( N \) over the input stream. This window is shifted by one element for every output value. The filter multiplies the input values in the current window element-wise with a block of \( N \) constants, the filter pattern, and it adds all these results together to create a single output value in the output stream. The FIR filter is illustrated in figure 6.1.

The FIR filter pattern is also often referred to as the filter kernel. A filter with \( N \) values in its kernel is called a \( N \)-tabs filter.

The order of any non-recursive filter is defined as the largest number of previous input values required to compute the current output. In other words, the order of a FIR filter equals \( N - 1 \). [WB02]

### 6.5.1 FIR filter formula

The FIR filter is defined by the input stream and the filter kernel \( c \) of length \( N \).

\[
FIR(c, N, A)(i) = \left( \sum_{j : \ 0 \leq j < N} A(i-j) \times c(j) \right) \quad (6.7)
\]

Note that for the first few values \( i < j \) the result is undefined, because the input stream \( A(i) \) is undefined for \( i < 0 \). This is an example of the so-called startup noise that we have discussed in section 5.1.
6.5.2 Example

The simplest nontrivial FIR filter is this running average filter:

\[ R_{\text{avg}}(n) = \frac{1}{2} A(n) + \frac{1}{2} A(n - 1) \]  \hfill (6.8)

As illustrated by this example, the block-averaging filter that we saw as an example in section 5.3, can also be implemented as a combination of a FIR filter and a downsample filter:

\[
B_{\text{avg}}(N, A)(i) = \\
\frac{1}{N} \left\langle \sum j : 0 \leq j < N : A(i \times N) - j \right\rangle = \\
\left\langle \sum j : 0 \leq j < N : \frac{1}{N} A((i \times N) - j) \right\rangle = \\
\{DN(I, A)(i) = A(i \times l)\} = \\
DN(N, \left\langle \sum j : 0 \leq j < N : \frac{1}{N} A((i - j) \right\rangle) = \\
\{\text{Define } c \text{ as } c[i] = \frac{1}{N} \text{ with } 0 \leq i < N\} = \\
DN(N, \text{FIR}(c, N, A)) = \\
\{R_{\text{avg}}(N, A) = \text{FIR}(c, N, A) \text{ with } c(j) = \frac{1}{N}; (0 \leq j < N)\} = \\
DN(N, R_{\text{avg}}(N, A)) \]  \hfill (6.9)

6.5.3 Vectorization

Again, we want to calculate an entire vector \( \vec{B} \) of outputs at once:

\[
\vec{B}(i) = \begin{bmatrix}
\text{FIR}(c, N, A)(i) \\
\cdots \\
\text{FIR}(c, N, A)(i + P - 1)
\end{bmatrix} \hfill (6.10)
\]

From formula 6.7 we derive:

\[
\vec{B}(i) = \left\langle \sum j : 0 \leq j < N : \bar{A}(i - j) \times c(j) \right\rangle \]  \hfill (6.11)

Theoretical Performance

Since this is the same as the scalar version, except for the vector variables instead of scalar values, speedup is again 100%.
SVPL Implementation

This formula consists of a sum of products with one factor a vector value and the other constant scalar value. We recognize the SVP’s MAC operator. To implement the algorithm on the SVP, we will need a loop over \( j \) from 0 to \( N \).

In every iteration we calculate one \( \bar{A}(i-j) \times c(j) \) and add this to the result. Since there is no operation to calculate a vector with a constant, we will create vectors \( c(j) \) containing the value \( c(j) \) at every position, for every \( j \). It is most efficient to do this in advance, i.e., in the initialization and since the number of registers is limited, we write these vectors to RAM.

```c
Vector v1, v2, fir;
Address a1, a2;
Integer c1, tabs;

/* Initialization */
// Write the FIR filter to RAM

tabs = 3;

a1 = 0;
fir = Vector(0.25);
a1 = a1 + VectorSize;
fir = Vector(0.50);
fir.store(RAM, a1);
a1 = a1 + VectorSize;
fir = Vector(0.25);
fir.store(RAM, a1);
```

The vector \( \bar{A}(i-j) \) always contains \( P-1 \) values from the vector \( \bar{A}(i-j-1) \) of the previous iteration, shifted by one location, and one new value \( A(i) \) from the input stream. This can be implemented by reading the \( A(i) \) vector once before the inner loop and by reading only the scalar \( A(i-j) \) in every iteration. That scalar will then be shifted into \( \bar{A}(i-j) \).

Taking into account that the addresses of our memories do not extend below 0, this translates into the following implementation:

```c
/* Fir Filter main loop */
// During every iteration, one output vector is produced.

a1 = 0;

while (a1 < IM.size()) {
    Vector v2;
    Address a3;

    // Load vector of input values and
    // MAC with the first filter tab
    v1.load(IM, a1);
```
a1 = a1 + VectorSize;
fir.load(RAM, a3);
c1 = 1;
v2.MAC(v1, fir);

while (c1 < tabs) {
   // For each subsequent filter tab, read
   // a new input value, shift accordingly
   // and MAC.
   Scalar s1;

   a3 = a3 + VectorSize;
   s1.load(IM, a1);
   ++a1;
   fir.load(RAM, a3);
   v1 = v1.shiftright(s1);
   ++c1;
   v2.MAC(v1, fir);
}

// Store the vector with results
v2.store(IM, a2);

a2 = a2 + VectorSize;
a1 = a2;
}

This solution has a total unit usage of 45% and CPV is 11.48 when N = 3, a
25% speedup, or is almost 4 times slower than the ideal performance (CPV = N).

When the number of filter tabs is known beforehand, it is more efficient to
roll out the above formula and to hardcode it like this 3-tab FIR filter:

Vector v1, v2, fir1, fir2, fir3;
Scalar s1, s2;
Address a1, a2, a3;

fir1 = Vector(0.25);
fir2 = Vector(0.50);
fir3 = Vector(0.25);

while (a1 < IM.size()) {
   v2 = Vector(0);

   v1.load(IM, a1);
   a1 = a1 + VectorSize;
   v2.MAC(v1, fir1);
s1.load(IM, a1);
v2.MAC(v1.shiftleft(s1), fir2);
a3 = a1 + i; // calculating in advance improves performance
v1 = v1.shiftleft(s1);

s2.load(IM, a3);
v2.MAC(v1.shiftleft(s2), fir3);
v1 = v1.shiftleft(s2);

v2.store(OM, a2);

a2 = a2 + VectorSize;
}

Total unit usage is less than with the generic solution (only 39%), but the CPV is much lower: only 7.04 instead of the 11.84 above. An improvement of over 40% and only 2.4 times slower than ideal!

6.6 Filter combinations

We have now defined the three filters separately. In order to combine those filters, we could of course simply concatenate them, but that would not be very efficient. The FIR filter will perform a lot of multiplications with zero since the UP filter added those. Moreover, the DN filter will disregard a lot of outputs from the FIR filter, so it would also be a waste of resources to calculate all these values. Therefore we will be looking for an optimized combined implementation.

We will start by analyzing and optimizing the upsample - FIR-filter combination in section 6.7 and 6.8. After that we will add the downsample filter and see if we can optimize even more in section 6.9.

6.7 Upsample & FIR filter: Functional approach

To make sure that our final algorithm is correct, we will create a mathematical derivation, like we did before. In the next section we will translate this mathematical result into program code.

6.7.1 Upsample and FIR filter combined

\[
FIR(UP(k, A), c, N)(i) = \left< \sum_{j : 0 \leq j < N} UP(k, A)(i - j) \times c(j) \right>
\]  

(6.12)

To make the formulas in this section less complicated, we will from this point on refer to this stream as \( B \):

\[
B(i) = FIR(UP(k, A), c, N)(i)
\]  

(6.13)

After applying the upsample filter to the input stream, only every \( k \)-th value is non-zero. Therefore it might be convenient to express the indices \( i \) in terms of \( k \):
\[ i = qk + r \quad (0 \leq r < k) \quad (6.14) \]

and this yields:

\[
FIR(UP(k, A), c, N)(qk + r) \\
= (\sum j : 0 \leq j < N : UP(k, A)(qk + r - j) \times c(j))
\]

The combination of upsampling and FIR filter is also illustrated in figure 6.2:

Figure 6.2: Upsample + Fir Filter

6.7.2 Eliminating the UP filter

From the definition of the UP filter, we know that most of the data in its output stream consists of zero values \((k - 1 \text{ out of } k \text{ values})\). This implies that if we calculate the values of \(B\) using the above definition (6.13), most multiplications will be multiplications by zero and these are of course a waste of time. We will now try to rewrite the definition of the \(B\) stream in terms of the original \(A\) stream and thus get rid of the \(UP\) filter in the definition.
Suppose \( 0 \leq q \) and \( 0 \leq r < k \)

\[
B(qk + r)
= \quad \text{FIR}(UP(A, k), c, N)(qk + r)
= \quad \text{Specification of } B
\]
\[
\left\langle \sum j : 0 \leq j < N : UP(A, k)(qk + r - j) \times c(j) \right\rangle
= \quad \{(r - j) \mod k \neq 0 : UP(A, k)(qk + r - j) = 0\}
\]
\[
\left\langle \sum j : 0 \leq j < N \land (r - j) \mod k = 0 : UP(A, k)(qk + r - j) \times c(j) \right\rangle
= \quad \{\text{dummy change} \quad j = sk + t\}
\]
\[
\left\langle \sum s, t : 0 \leq sk + t < N \land 0 \leq t < k \land (r - t) \mod k = 0 : \quad UP(A, k)((q - s)k + (r - t)) \times c(sk + t) \right\rangle
= \quad \{0 \leq r, t < k \land (r - t) \mod k = 0 : r = t; \ \text{hence eliminate } t\}
\]
\[
\left\langle \sum s : 0 \leq sk + r < N : UP(A, k)((q - s)k) \times c(sk + r) \right\rangle
= \quad \{UP(A, k)((q - s)k) = A(q - s)\}
\]
\[
\left\langle \sum s : 0 \leq sk + r < N : A(q - s) \times c(sk + r) \right\rangle
= \quad \{\text{arithmetic}\}
\]
\[
\left\langle \sum s : 0 \leq s < \frac{N - r}{k} : A(q - s) \times c(sk + r) \right\rangle
\]
(6.15)

This result is exactly what we wanted to achieve: We rewrote the values of the output stream in function of the input stream and every predictable multiplication with zero is eliminated.

Because of this elimination, the number of multiplications necessary for calculating a single output value does no longer equal \( N \), but it is much less and in some cases it might fluctuate, depending on \( N \) and the current index \( i (= qk + r) \).

This means that when we want to calculate successive output values, the number of multiplications needed inf the formula

\[
\left\langle \sum s : 0 \leq s < \frac{N - r}{k} : A(q - s) \times c(sk + r) \right\rangle
\]

is not necessary constant.

We derive:
\[
\frac{N - r}{k} = \frac{\{N = k(N \div k) + N \mod k\}}{k(N \div k) + N \mod k - r} \\
\frac{k}{k} = \{Arithmetic\} \\
N \div k + \frac{N \mod k - r}{k} \\
\Rightarrow \{Arithmetic\} \\
N \div k + \frac{N \mod k - r}{k}
\]

Now because \(0 \leq s < \frac{N - r}{k}\), the range of \(s\) is extended by one when the second term from the last formula in 6.16 is greater than zero:

\[
\frac{N \mod k - r}{k} > 0 \\
\Leftrightarrow \\
N \mod k > r
\]

When \(N \mod k = 0\) (Remember: \(N \mod k = r\)), the number of multiplications in the formula above is always \(N \div k\). But when \(N \mod k \neq 0\), the number of multiplications is \(N \div k\) when \(N \mod k > r\) and \(N \div k + 1\) when \(N \mod k \geq r\).

As we will see in the next subsection, implementation of the FIR filter is much easier when the number of multiplications for each output value remains constant for a fixed \(N\) and \(k\). This is easy to obtain by adding enough zeros to the FIR filter pattern. The maximum number of zeros introduced is \(k - 1\).

After adding these zeros, we can simplify the formula from 6.15:

\[
\left\{ \sum_{s: 0 \leq s < \frac{N - r}{k}} : A(q - s) \times c(sk + r) \right\} \\
= \{\text{Extend } c \text{ with zeros}\} \\
\left\{ \sum_{s: 0 \leq s < u} : A(q - s) \times c(sk + r) \right\}
\]

with \(u\) defined as:

\[
u = \begin{cases} 
N \div k, & N \mod k = 0 \\
N \div k + 1, & N \mod k \neq 0
\end{cases}
\]

Of course the addition of zeros introduces some overhead, because multiplications with zero are introduced. It is however unlikely that an implementation that does not insert those zeros can be more efficient because of the additional complexity that algorithm would suffer from.
6.7 Upsample & FIR filter: Functional approach

The actual number of zeros that is added, is \( k - N \mod k \) for each block of \( k \) outputs. And consequently, a loss of efficiency of \( \frac{k - N \mod k}{(N \div k + 1)k} \) percent is introduced by multiplications with those zeros.

### 6.7.3 Matrix form

In the previous subsection, we have rewritten the definition of the output stream \( B \) in terms of the original input stream \( A \) and thereby eliminated (almost) all multiplications by zero.

Our next goal is to calculate formula (6.18) as efficient as possible using vector operations. It should be clear that this formula is perfectly suitable for implementation using the MAC operation. We will now investigate what is the most efficient way to implement this formula on a vector processor.

As can be seen in formula 6.18, the computation of consecutive output values shows a recurring pattern every \( k \) outputs: every \( k \)-th output value uses the same elements from the pattern vector. Since we want to calculate the output values in blocks of \( P \), we will look at the calculation of \( Pk \) values.

We will only calculate outputs in blocks of \( Pk \) (vector size \( \times \) upsample factor). According to the notation in 6.14, we define the first value of an arbitrary block as \( qPk \) (\( q \geq 0 \)).

This is the matrix representation:

\[
\begin{bmatrix}
B(qPk) \\
B(qPk + 1) \\
\vdots \\
B(qPk + (P - 1)k + k - 1)
\end{bmatrix}
= \begin{bmatrix}
C_{0,0} & C_{0,u-1} & 0 & 0 & \cdots \\
C_{k-1,0} & C_{k-1,u-1} & 0 & 0 & \cdots \\
0 & C_{0,0} & C_{0,u-1} & 0 & \cdots \\
0 & C_{k-1,0} & C_{k-1,u-1} & 0 & \cdots \\
A(qP - u + 1) & & & & \\
\vdots & & & & \\
A(qP) & & & & \\
A(qP + P - 1)
\end{bmatrix}
\]

(6.20)

The matrix that we use to multiply with values from \( A \) is a fixed matrix that can be calculated in advance. Inside this matrix we also recognize a recurring matrix \( C \), based on values from the FIR filter pattern \( c \). This matrix with values from the FIR filter kernel \( c \) counts \( k \) rows and \( u \) columns.
From formula 6.18 we can derive that the values of the recurring matrix

$$
C = \begin{bmatrix}
C_{0,0} & \ldots & C_{0,u-1} \\
& \ldots & \\
C_{k-1,0} & \ldots & C_{k-1,u-1}
\end{bmatrix}
$$

(6.21)

are given by

$$
C_{r,s} = c((u - 1 - s) \times k) + r)
$$

(6.22)

As we indicated above, when $N \mod k \neq 0$, a number of zeros has to be introduced into the filter kernel. This is a good time to show where these end up, for example with $N = 7$ and $k = 3$ the $C$-matrix looks like this (a $3 \times 3$ matrix):

$$
\begin{bmatrix}
c(6) & c(3) & c(0) \\
c(7) & c(4) & c(1) \\
c(8) & c(5) & c(2)
\end{bmatrix}
$$

(6.23)

c(7) and c(8) do not exist because the number of tabs in the filter is 7. Therefore the filter was extended with $k - 1 (= 2)$ zeros.

A certain repetitive structure can be found in the calculation of any arbitrary FIR filter. Our next step is to define this pattern in terms of the vector size $P$ in order to create an efficient vectorized version of the FIR filter. In the following two subsections we will see how we can group the output values in blocks of $P$.

### 6.7.4 Grouping outputs according to upsample factor

Obviously, we can write the matrix multiplication from formula 6.20 as a sequential set of formulas, grouped in $P$ blocks of $k$ formulas each.
\[ B(qP_k) = C_{0,0}.A(qP-u+1) + \ldots + C_{0,u-1}.A(qP) \]

\[ \ldots \]

\[ B(qP_k+(k-1)) = C_{k-1,0}.A(qP-u+1) + \ldots + C_{k-1,u-1}.A(qP) \]

\[ \ldots \]

\[ B(qP_k+ek) = C_{0,0}.A(qP-u+2) + \ldots + C_{0,u-1}.A(qP+1) \]

\[ \ldots \]

\[ B(qP_k+ek+(k-1)) = C_{k-1,0}.A(qP-u+2) + \ldots + C_{k-1,u-1}.A(qP+1) \]

\[ \ldots \]

\[ B(qP_k+(P-1)k) = C_{0,0}.A(qP-u+P) + \ldots + C_{0,u-1}.A(qP+P-1) \]

\[ \ldots \]

\[ B(qP_k+(P-1)k+(k-1)) = C_{k-1,0}.A(qP-u+P) + \ldots + C_{k-1,u-1}.A(qP+P-1) \]

(6.24)

Let us count the blocks in 6.24 by \( e \) \((0 \leq e < P)\) and the number of the values in such a block by \( f \) \((0 \leq f < k)\). An arbitrary output value in a \( Pk \) block starting at \( qP_k \) is then defined as:

\[ B(qP_k+ek+f) = \sum_{s=0}^{u-1} C_{f,s}.A(qP+u+1+e+s) \]

(6.25)

There are some interesting properties to mention here. Within a block of length \( k \), the input values needed to calculate the output values are the same because \( A(qP+u+1+e+s) \) does not depend on \( f \).

Secondly, the input values necessary for an arbitrary output value are all successive. If you consider all values from \( A \) that are vertically aligned in formula 6.24 \((= \text{fixed } s)\), leaving out the doubles, you get a window on the input stream of length \( k \). For adjacent columns \((s \text{ increased by } 1)\), this window is only shifted by one location.

We will be able to exploit these properties later on when working towards the implementation.

### 6.7.5 Grouping outputs according to vector size

The above \( P \) blocks of length \( k \) can also be grouped as \( k \) blocks of length \( P \). Since the blocks have length \( P \), they can also be written in vector notation.
We count the blocks of length \( P \) as \( g \) (0 \( \leq g < k \)) and the values inside a block by \( h \) (0 \( \leq h < P \)). This means that \( \vec{B}_{q,g} \) is an arbitrary output vector from the block starting at \( B(qPk) \) and that \( \vec{B}_{q,g}(h) \) is an arbitrary value in that vector.

We can define the relation between the values \( e \), \( f \), \( g \) and \( h \) as follows:

\[
\begin{align*}
g &= (ek + f) \mod P \\
\h &= (ek + f) \mod P \\
e &= (gP + h) \div k \\
f &= (gP + h) \mod k
\end{align*}
\tag{6.26}
\]

\[
\vec{B}_{q,0} = \begin{bmatrix}
B(qPk) \\
\vdots \\
B(qPk + h) \\
\vdots \\
B(qPk + (P - 1))
\end{bmatrix}
\]

\[
\vec{B}_{q,g} = \begin{bmatrix}
B(qPk + gP) \\
\vdots \\
B(qPk + gP + h) \\
\vdots \\
B(qPk + gP + (P - 1))
\end{bmatrix}
\tag{6.27}
\]

\[
\vec{B}_{q,k-1} = \begin{bmatrix}
B(qPk + (k - 1)P) \\
\vdots \\
B(qPk + (k - 1)P + h) \\
\vdots \\
B(qPk + (k - 1)P + (P - 1))
\end{bmatrix}
\]

Now that we grouped the output values in blocks of the vector size, we want to write the calculation of such a block (= an output vector) with vector operations. Let's define \( \vec{C}_{g,s} \) as a vector based on filter values and \( \vec{A}_{q,g,s} \) as a vector based on input stream values.

\[
\vec{B}_{q,0} = \vec{C}_{0,0} \times \vec{A}_{q,0,0} + \vec{C}_{0,1} \times \vec{A}_{q,0,2} + \ldots + \vec{C}_{0,u-1} \times \vec{A}_{q,0,u-1}
\]

\[
\vec{B}_{q,g} = \vec{C}_{g,0} \times \vec{A}_{q,g,0} + \vec{C}_{g,1} \times \vec{A}_{q,g,1} + \ldots + \vec{C}_{g,u-1} \times \vec{A}_{q,g,u-1}
\tag{6.28}
\]

\[
\vec{B}_{q,k-1} = \vec{C}_{(k-1),0} \times \vec{A}_{q,(k-1),0} + \vec{C}_{(k-1),1} \times \vec{A}_{q,(k-1),u-1}
\]

Or in general:

\[
\vec{B}_{q,g} = \sum_{s=0}^{u-1} \vec{C}_{g,s} \times \vec{A}_{q,g,s}
\tag{6.29}
\]
With this definition, we rewrote the FIR filter so that it computes vectors instead of single scalar values. But we still have to define the relation between the vector parameters $\tilde{C}_{g,s}$ and $\tilde{A}_{q,g,s}$ from formula 6.24 on the one hand and the values from the $C$ matrix and the input stream $A$ on the other hand.

These vectors can now easily be defined by translating $e$ and $f$ coordinates to $g$ and $h$ coordinates, using the conversions in formula 6.26.

### $\tilde{C}_{g,s}$

To select the values from the pattern, necessary for the current block $g$ and iteration $s$, we need to convert $C_{f,s}$ from formula 6.25 to the new coordinate system $(g$ and $h)$:

\[
C_{f,s} = C_{gP} \text{ div } k, s
\]  

(6.30)

For a block of length $P$, that gives us:

\[
\tilde{C}_{g,s} = \begin{bmatrix}
C_{((gP) \text{ div } k), s} \\
\vdots \\
C_{((gP+h) \text{ div } k), s} \\
\vdots \\
C_{((gP+(P-1)) \text{ div } k), s}
\end{bmatrix}
\]  

(6.31)

where $C_{r,s}$ is defined as in formula (5.22).

### $\tilde{A}_{q,g,s}$

If we look back at formula 6.25, we see that the selected values from the input stream depend on three variables (for a fixed filter and vector size): $q$, $e$ and $s$. We can again translate $A(qP - u + 1 + e + s)$:

\[
A(qP - u + 1 + e + s) = A(qP - u + 1 + (gP + h) \text{ div } k + s)
\]  

(6.32)

of for an entire vector:

\[
\tilde{A}_{q,g,s} = \begin{bmatrix}
A(qP - u + 1 + (gP) \text{ div } k + s) \\
\vdots \\
A(qP - u + 1 + (gP + h) \text{ div } k + s) \\
\vdots \\
A(qP - u + 1 + (gP + P - 1) \text{ div } k + s)
\end{bmatrix}
\]  

(6.33)

The entire FIR algorithm is now rewritten to be implemented on the SVP, but there are a few things that can make the implementation of this algorithm less complicated.

Currently the calculations of the indices on $A(qP - u + 1 + (gP + h) \text{ div } k + s)$, used in $\tilde{A}_{q,g,s}$ take too much cycles to compute every time. Most of these cycles can however simply be avoided.
The first part, \( qP - u + 1 \), needs only to be calculated once for every block of \( Pk \) outputs. The last part, \(+s\), is used to increase the indices once every cycle over \( s \).

Finally, the remaining parts \((gP + h) \div k\) can be calculated in advance and stored in \( k \) vectors. These vectors can be seen as some kind of pattern used to select values from the input stream, we will indicate these vectors with \( \vec{I}_g \) \((0 \leq g < k)\):

\[
\vec{I}_g = \begin{bmatrix}
(g \times P) \div k \\
\cdots \\
(g \times P + h) \div k \\
\cdots \\
(g \times P + P - 1) \div k
\end{bmatrix}
\]  

(6.34)

And we can write \( \tilde{A}_{q,g,s} \) in function of these pattern vectors \( \vec{I} \):

\[
\tilde{A}_{q,g,s} = qP - u + 1 + \vec{I}_g + s
\]  

(6.35)

We have now formally defined the entire algorithm. There are however a few issues to examine. For instance, we did not check whether all temporary vector results will fit into a single machine vector of size \( P \). This will not be the case, and we will provide some solutions in the next subsection 6.7.8.

**Theoretical Performance**

In formula 6.29 we see that it takes \( u \) MACs to calculate a single output vector. Again, this is the same as in the scalar algorithm (formula 6.18) and thus the speedup is 100% !

**6.7.8 The effect of the limited vector size**

We designed the implementation of the FIR filter with as target system the SVP and its fixed vector length \( P \). Therefore we made sure that all intermediate values as well as the output values are vectors of size \( P \). There is only one vector that will not fit in a \( P \)-size vector and that’s the vector of input values as described in the matrix notation of formula 6.20.

In matrix representation 6.20 we need \( P + N \div k \) values from the input stream \( A \) to calculate \( k \) output vectors (or \( k \times P \) output values). This is always more than will fit in a single vector. Consequently we cannot read all input values that are necessary for a \( Pk \) block at once.

How much values from \( A \) would we need if we agree to read \( A \) values from memory before every calculation of a single output vector \( \vec{B}(i) \) ? Therefore we look back at formula 6.20. If we calculate \( P \) output values, we need at least \( u \) values (this is the width of matrix 6.21). Furthermore we need one additional value from \( A \) for every indentation of that matrix in formula 6.20.
These indentations occur at most \((P - 1) \text{ div } k\) times for \(P\) output values. So in total we need \((u + ((P - 1) \text{ div } k))\) values from \(A\) to calculate the next \(P\) outputs. In many cases, these will fit in a single vector, but still there are some cases that cannot be resolved in this way. In general, we need \((u + ((P - 1) \text{ div } k) - 1) \text{ div } P + 1\) vectors to store all necessary values.

Clearly, this might have an impact on performance. Later on in this document we will try to measure this impact.

### 6.8 Upsample & FIR filter: Implementation

We will now implement two versions of the vectorized FIR filter algorithm. In the first version, we assume \((u + ((P - 1) \text{ div } k)) \leq P\). The second version will be an adaptation of the first one, making sure that also the cases with \((u + ((P - 1) \text{ div } k)) > P\) are supported.

#### 6.8.1 Initialization

Common to both solutions presented below is the initialization section of the algorithm. In this section the \(C\) and \(I\) vectors are created. Those vectors can be calculated as soon as we now the actual FIR kernel.

In order to keep the example simple, we only present the initialization of the case where \(N \leq P\). It is easy to adapt this code to support arbitrary kernel sizes. The initialisation here is only provided as an implementation example and is not optimized. As the initialisation is only performed once, at filter startup, optimizing this part of the code is less important.

First we need to define the FIR filter by defining and initializing the necessary variables:

```c
Vector filter;

filter[0] = 0.25; filter[1] = 0.50; filter[2] = 0.25;
```

```plaintext
Integer k; k = 2; // upsample coefficient
Integer N; N = 3; // number of tabs in FIR filter
Integer Ndivk; Ndivk = 1;
```

Secondly we state how we will use the RAM memory. The first part of RAM will be used store the \(C\) vectors and the adjacent part to store the \(I\) vectors. For convenient and efficient access to those vectors we calculate the base pointers of above-mentioned memory chunks.

There are \(uk\) \(C\) vectors and \(k\) \(I\) vectors.

```c
// RAM Memory Base Pointers
Address C_BP, I_BP;
```
C_BP = 0;
I_BP = (Ndivk + 1) * k * VectorSize;

Vector C, I;

// Initialization
Integer basefilterindex, filterindex;

Address pointer;

Now all is set to calculate the \( \tilde{C} \) vectors as explained in section 6.7.6.

// 1. Calculate C vectors and store them in RAM
Integer y = 0;
while (y <= Ndivk)
{
    pointer = C_BP + (VectorSize * (Ndivk - y));
    basefilterindex = (y * k);
    filterindex = basefilterindex;

    Integer x = 0;
    while (x < k)
    {
        Integer i = 0;
        while (i < VectorSize)
        {
            if (filterindex == (basefilterindex + k))
                filterindex = basefilterindex;

            if (filterindex >= N) C[i] = 0;
            else C[i] = filter[filterindex];

            filterindex++;
            i++;
        }

        C.store(RAM, pointer);
        pointer = pointer + (VectorSize * (Ndivk + 1));
        x++;
    }
    y++;
}

And the \( \tilde{I} \) vectors can be calculated as well.
// 2. Calculate the I vectors and store them in RAM
pointer = I_BP;

Integer x=0;
while (x < k)
{
    Integer base = (x * VectorSize) / k;

    Integer i = 0
    while (i < VectorSize)
    {
        I[i] = (((x * VectorSize) + i) / k) - base;
        i++;
    }
    I.store(RAM, pointer);
    pointer = pointer + VectorSize;
    x++;
}

6.8.2 Solution 1 - Constraint \(((N_{divk}) + (P_{divk})) \leq P\)

Now that we have performed all preliminary work, it is time to implement the actual algorithm using the predefined vectors from above.

Implementation

Integer tmp, tmp2;
Vector A, B;

A_P, A_Psave = 0;
O_P = 0;

while (A_P < IM.size())
{
    C_P = C_BP;
    C.load(RAM, C_P);

    Integer b = 0;
    while (b < k)
    {
        tmp = b * VectorSize;

        B = Vector(0);
        I.load(RAM, I_BP + tmp);
        A_P = A_Psave + (tmp / k);
tmp2 = 0;
A.load(IM, A_P);

while (tmp2 <= Ndivk) {
    B.MAC(A.shuffle(I), C);
    A = A.shiftdown(0);
    C_P = C_P + VectorSize;
    C.load(RAM, C_P);
    tmp2++;
}

B.store(0M, 0_P);
0_P = 0_P + VectorSize;
b++;
}

A_P = A_Psave + VectorSize;
A_Psave = A_P;

Performance
If we run the program with vector size 32, a 20-tab FIR filter and upsample ratio 2 for 1600 input values, these are the generated statistics.

---- Basic Processor Stats ----

<table>
<thead>
<tr>
<th>VECTOR</th>
<th>SCALAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory ops: 1450</td>
<td>Memory ops: 0</td>
</tr>
<tr>
<td>Math ops: 1100</td>
<td>Math ops: 0</td>
</tr>
<tr>
<td>Modify ops: 2200</td>
<td></td>
</tr>
<tr>
<td>Total ops: 4750</td>
<td>Total ops: 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INTEGER</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total ops: 2950</td>
<td>Total ops: 2951</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCHEDULER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total signals : 11851</td>
</tr>
<tr>
<td>Total operations : 6801</td>
</tr>
<tr>
<td>Total cycles : 4351</td>
</tr>
<tr>
<td>Total unit usage : 39 %</td>
</tr>
<tr>
<td>Vector unit usage : 25%</td>
</tr>
<tr>
<td>(excl. Int ops): 43%</td>
</tr>
</tbody>
</table>

We did not include the initialization in these statistics, because it only needs to run once and the results of that initialization can even be hard-coded when the algorithm is implemented with a fixed kernel.

Since 3200 output samples were generated, thus the CPV is 43.51 when upsample ratio 2 is used. When we keep all conditions the same, but change the
6.8 Upsample & FIR filter: Implementation

upsample ratio to 10, CPV is only 18.302. This stands to reason: the higher the upsample ratio, the higher the number of multiplications with zero per output sample, the lower the number of actual multiplications to perform per output sample.

Still, this is roughly 4 to 9 times worse than the ideal of $u$ cycles.

6.8.3 Solution 2 - Eliminating the constraint

If we want to eliminate the constraint $(u + (P - 1) \div k) \leq P$, we need to implement the solution that was given in section 6.7.8.

Implementation

In order to make our implementation support those cases, we need to change and extend two pieces of code in the above implementation.

First we want to change the line

\[ A\_load(IM, A\_P); \]

so that it is capable of reading all values necessary for the calculation of the current output vector.

One could wrongly suppose that it's most optimal to read all these values at once into multiple variables and select all values we need into a new variable before executing the MAC in the most inner loop. However, it takes more than one cycle to select values from multiple variables into a single variable. A possible improvement on the SVP architecture would be to support this kind of 'shuffle from multiple variables'.

Still, it is more efficient to read all necessary values of $A$, just before executing the MAC. So we move the load statement into the most inner loop and we can discard the shift operation.

```c
Integer b = 0;
while (b < k)
{
    tmp = b * VectorSize;
    B = Vector(0);
    I.load(RAM, I_BP + tmp);
    A_P = A_Psave + (tmp / k);
    tmp2 = A_P + Ndivk;
```
while (A_P <= tmp2) {
    A.load(IM, A_P);
    B.MAC(A.shuffle(I), C);
    C_P = C_P + VectorSize;
    C.load(RAM, C_P);
    A_P++;
}

B.store(OM, O_P);
O_P 4= O_P + VectorSize;
b++;
}

Performance

Let us take a look at the statistics generated by this slightly modified program:

---- Basic Processor Stats ----

<table>
<thead>
<tr>
<th>VECTOR</th>
<th>SCALAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory ops: 2450</td>
<td>Memory ops: 0</td>
</tr>
<tr>
<td>Math ops: 1100</td>
<td>Math ops: 0</td>
</tr>
<tr>
<td>Modify ops: 1100</td>
<td></td>
</tr>
<tr>
<td>Total ops: 4650</td>
<td>Total ops: 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INTEGER</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total ops: 550</td>
<td>Total ops: 5451</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCHEDULER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total signals : 10751</td>
</tr>
<tr>
<td>Total operations : 7901</td>
</tr>
<tr>
<td>Total cycles : 4351</td>
</tr>
<tr>
<td>Total unit usage : 45 %</td>
</tr>
<tr>
<td>Vector unit usage: 25%</td>
</tr>
<tr>
<td>(excl. Int ops): 46%</td>
</tr>
</tbody>
</table>

Surprisingly, the second, more complete algorithm actually performs LESS operations. Nevertheless, the total number of processor cycles remains the same with our basic scheduler.

What happened here? While the number of vector modifiers has reduced substantially, the number of vector memory operations increased with almost 70%. On most processors, memory operations take at least a few processor cycles to complete and that is why one could easily expect the performance to decrease. We did however define that every operation on a SVP takes the same amount of time to complete, namely one processor cycle.
We can conclude that an algorithm that looks worse at first sight, might actually not be so bad at all since its performance depends partly on the processor scheduler.

6.9 The FIR & downsampling filter

As a final step before combining the three filters, we will create a vector implementation for the concatenation of FIR and downsampling filter. For simplicity, we will denote this combination of filters as the FIR-DN filter.

By adding the downsampling filter to the FIR filter, a lot of output values from the FIR filter are no longer necessary in the final output stream and therefore they should not be computed at all. Our implementation's main target is to avoid any calculation that is only used for such an unused output value from the FIR filter.

6.9.1 Functional Approach

Let us write down the specification for the FIR-DN combination and write it as a function of the original input stream:

\[
DN(l, FIR(c, N, A))(i) = \{ \text{Specification of FIR} \}
\]

\[
\left\langle \sum_{j : 0 < j < N : DN(l, A)(i - j) \times c(j)} \right\rangle
\]

\[
\left\{ \text{Equation 6.2} \right\}
\]

\[
\left\langle \sum_{j : 0 < j < N : A((i \times l) - j) \times c(j)} \right\rangle
\]

This specification of the FIR-DN filter is much like that of the standalone FIR filter. The only difference lies in the way values from the input stream are selected to create an output value. While the FIR filter had a window of length \( N \) sliding over the input stream at exactly one position per output value, the window of the FIR-DN filter is shifted by \( l \) positions for consequent output values.

This observation is also illustrated in figure 6.3.

If we eliminate the image of the intermediate data stream, the simplified sketch looks like in figures 6.4 and 6.5. Figure 6.4 shows us the case in which the downsampling value \( k \) is smaller than the FIR filter length \( N \). All input values are used, some (or all) of them more than once.

Figure 6.5 illustrates the case where \( l \) is bigger than \( N \): \( l - N \) out of \( l \) values from the input stream not used for any output value.

In either case, the proposed specification of the FIR-DN filter is optimized. No useless calculations are performed nor are they performed more than once.
Figure 6.3: Fir + Downsample Filter

Figure 6.4: Fir + Downsample Filter: $l < N$

Figure 6.5: Fir + Downsample Filter: $l \geq N$
6.9.2 Vectorization

Rewriting formula 6.36 to calculate an entire output vector $\bar{B}$ at once is obvious:

$$\bar{B} = \left\langle \sum_{j: 0 < j < N} \overline{A((i \times l) - j)} \times c(j) \right\rangle \quad (6.37)$$

Theoretical Performance

Not surprisingly, the theoretical speedup is again 100%.

SVPL Implementation

We will now convert the formulas and reasoning above into a working SVPL program. If we look back at formula 6.7 we notice that the only difference with formula 6.36 is the multiplication of indices of the input $A$ with the constant factor $l$. This makes it very easy to create a SVPL implementation: we only need to make a change to the way the original implementation of the FIR filter reads the input values. As the input window moves in larger steps than with the FIR filter, we cannot use the trick of reading only one scalar and shifting the $A$ vector. For every iteration, completely different values will be necessary.

Obviously, this results in:

```svpl
Vector v1, v2, fir;
Address a1, a2;
Integer c1, tabs, l;

/* Initialization */
// Write the FIR filter to RAM

l = 5; // Downsample value

tabs = 3;

a1 = 0;
fir = Vector(0.10);
fir.store(RAM, a1);
a1 = a1 + VectorSize;
fir = Vector(0.50);
fir.store(RAM, a1);
a1 = a1 + VectorSize;
fir = Vector(0.40);
fir.store(RAM, a1);

/* FIR Filter main loop */
// During every iteration, one output vector is produced.

a1 = 0;
```
enableStats();

while (a1 < IM.size()) {
    Vector v2;
    Address a3, a4;

    // Load vector of input values and
    // MAC with the first filter tab
    v1.loadWithStride(IM, a1, 1);
    a4 = a1 + 1;

    fir.load(RAM, a3);
    c1 = 1;
    v2.MAC(v1, fir);

    while (c1 < tabs) {
        // For each subsequent filter tab, read
        // a new input value, shift accordingly
        // and MAC.
        Scalar s1;
        a3 = a3 + VectorSize;

        v1.loadWithStride(IM, a4, 1);
        fir.load(RAM, a3);
        ++c1;
        v2.MAC(v1, fir);
        ++a4;
    }

    // Store the vector with results
    a1 = a1 + VL;
    v2.store(IM, a2);

    a2 = a2 + VectorSize;
}

When we look at the schedules that are created by this solution, we see that the integer math unit is occupied in every schedule. This means that unless any integer calculations in the solutions above can be avoided, this is the optimal SVP solution, independent of other optimizations.

Total unit usage is 44% and the average CPV is 12.8. The obtained performance is almost the same as the performance of the standalone FIR filter.

6.10 The Three Filters Combined

In this final section we combine the three different filters to create the Scaler as illustrated in figure 6.6.
6.10 The Three Filters Combined

Figure 6.6: The Scaler

6.10.1 Functional Approach

\[ DN(l,FIR(c,N,UP(kA)))(i) = \{ \text{Specification of } DN \} \]
\[ FIR(c,N,UP(k,A))(il) = \{ il = qk + r \text{where} 0 \leq q \text{ and} 0 \leq r < k \} \]
\[ FIR(c,N,UP(k,A))(qk + r) = \{ \text{Formula 6.15} \} \]
\[ \left\langle \sum s : 0 \leq s < u : A(q - s) \times c(sk + r) \right\rangle \]

This result is the same as with the combination of UP and FIR filter, the only difference is that \( sk + r \) is always a multiple of \( l \). This condition will have its impact on how efficient we can calculate an entire vector of output values.

For brevity, we will denote \( DN(l,FIR(c,N,UP(kA))) \) as \( D \).

The scaler generates a stream that contains only selected values from the combination of UP and FIR. Therefore we must avoid to calculate all other outputs in the first place. When we look back at formula 6.20 we see that this can easily be achieved by crossing out entire lines in that matrix multiplication.

To find a correct implementation, we can follow the same system that we used to develop the UP-FIR filter implementation. If we look back at the formulas in 6.24, only every \( l \)-th line is necessary to compute the result of the UP-FIR-DN filter. If we remove all other lines, we form group of \( P \) remaining lines to be calculated in a single cycle.
\[ D(qPk) = B(qPkl) \]
\[ \ldots \]
\[ D(qPk + (P - 1)) = B(qPkl + (P - 1)l) \]
\[ \ldots \]
\[ D(qPk + (k - 1)P) = B(qPkl + (k - 1)Pl) \]
\[ \ldots \]
\[ D(qPk + (k - 1)P + (P - 1)) = B(qPkl + (k - 1)Pl + (P - 1)l) \]

Again, we took \( Pk \) output values together and from the \( P \)-size groups we can once again determine \( \widetilde{A} \) and \( \widetilde{C} \) vectors. After \( Pk \) output values, those vectors will be repeated as before. We count the blocks with variable \( g(0 \leq i < k) \) and inside a block with \( h(0 \leq j < P) \).

\[ \widetilde{D}_{g,s} = \sum_{s=0}^{u-1} \widetilde{C}_{g,s} \times \widetilde{A}_{q,g,s} \]  \hspace{1cm} (6.40)

And the relation between \( a, f, g \) and \( h \) can now be defined as:

\[
\begin{align*}
g &= (ek + f) \text{ div } P(l(ek + f) \text{ mod } l = 0) \\
h &= (ek + f) \text{ div } l \text{ mod } P((ek + f) \text{ mod } l = 0) \\
e &= (gPl + hl) \text{ div } k \\
f &= (gPl + hl) \text{ mod } k
\end{align*}
\]  \hspace{1cm} (6.41)

These conversion rules can be used to define the \( \widetilde{A} \) and \( \widetilde{C} \) vectors:

\[
\widetilde{C}_{q,s} = \begin{bmatrix}
C((gPl) \text{ div } k),s \\
\ldots \\
C((gPl+(P-1)l) \text{ div } k),s
\end{bmatrix}
\]  \hspace{1cm} (6.42)

These vector can be calculated in advance, like we did in the UP-FIR case. Now we need the define the \( \widetilde{A} \) vectors:

\[ A(qP - u + 1 + e + s) = A(qP - u + 1 + ((gP + h)l) \text{ div } k + s) \]  \hspace{1cm} (6.43)

therefore:

\[
\widetilde{A}_{q,g,s} = \begin{bmatrix}
A(qP - u + 1 + (gPl) \text{ div } k + s) \\
\ldots \\
A(qP - u + 1 + ((gP + h)l) \text{ div } k + s) \\
\ldots \\
A(qP - u + 1 + ((gP + P - 1)l) \text{ div } k + s)
\end{bmatrix}
\]  \hspace{1cm} (6.44)
which leads to these indices:

\[
\tilde{I}_g = \begin{bmatrix}
  gP_l \div k \\
  \ldots \\
  ((gP + h)l) \div k \\
  \ldots \\
  ((gP + P - 1)l) \div k
\end{bmatrix}
\] (6.45)

Up till now, we could use the exact same algorithm for the UP-FIR-DN filter as for the UP-FIR filter. Nevertheless, these indices on \( A \) raise a problem. The maximum consequent range of values from \( A \) from which of values for \( \tilde{A} \) are selected is no longer \( 1 + (P - 1) \div k \), but \( 1 + ((P - 1) \div k) \). This also implies that consequent values in \( \tilde{A} \) are not necessarily consequent values from the input stream (whenever \( l > k \)).

When \( l > k \), there can be a varying number of samples between two necessary input values. This number is bigger than zero for at least one \( \tilde{A} \) (only when \( l > k \)). This makes it impossible to compare a \( \tilde{A} \) with a single shuffle statement. We will not create an implementation that overcomes this issue.

Conclusion: when \( k \geq l \), the UP-FIR-DN filter has exactly the same performance as the FIR-DN filter. We can use the same algorithm, only the predefined vectors \( A \) and \( \tilde{C} \) need to be calculated in a slightly different way. We did already study this solution in the chapter on the UP-FIR filter.

When \( l > k \), the maximum achieved performance will drop in proportion to \( \frac{1}{k} \) because it takes \( \left\lceil \frac{1}{k} \right\rceil \) vector load statements to read all necessary input values from the range with length \( 1 + ((P - 1) \div k) \). Additionally, the same amount of shuffle statements is needed to put together the \( \tilde{A} \) vector.

The most important consequence from this result is that the implementation we have seen is particularly useful for interpolation (creating a data stream with an higher sample rate than the original) and not for decimation (the opposite).

### 6.11 Conclusion

For every algorithm in this chapter we found a vectorized version that guarantees, at least under certain conditions, a 100% speedup. This means that under these conditions arbitrary amounts of parallelism can be achieved and thus, at expense of additional hardware costs, arbitrary speeds are possible.

From the implementation of these algorithms on the SVP processor, it became clear that although the algorithms indicate a possible 100% speedup, this speedup cannot be guaranteed on the simple SVP architecture. A lot of cycles or processing time is lost due to control, addressing and memory operations. The more the parallelism is increased, the more distinct these problems become: they are the bottleneck of the SVP's speed.
Luckily, in modern DSP design a lot of effort is put in techniques to overcome these bottlenecks: zero-overhead looping, specialized addressing modes, integrated addressing instructions, more and wider memory busses, etc.

The SVPL library makes it possible to implement such additional features with ease. For most features, one should simply extend the statistical unit, or create one from scratch, so that statistics are generated for the improved vector processor. Because of its transparent architecture, it’s even possible to extend the Simple Vector Processor Language to support the new features. In this second case, you are not obligated to create your own statistical unit implementation as long as the default one suits your needs. We explored these expansion possibilities in detail in chapter 4.
Chapter 7

The IIR Filter

7.1 Introduction

In this chapter we study various vector implementations of the IIR or Infinite Impulse Response filter and compare their efficiency.

An IIR filter is represented as a difference equation where the output at a given instant is based on the current and previous input values as well as on previous output values. The general difference equation that represents an IIR filter is:

\[
IIR(a, b, N, M, A)(i) = \left\langle \sum_{n : 0 \leq n < N} A(i - n) \times a(n) \right\rangle + \left\langle \sum_{m : 0 < m < M} IIR(a, b, N, M, A)(i - m) \times b(m - 1) \right\rangle
\]

(7.1)

\(a\) and \(b\) are arrays of filter constants, containing \(N\) and \(M\) values respectively. This notation is often called the Auto-Regressive Moving Average (ARMA) representation. [Roc03]

The impulse response of a FIR filter is limited in time but the impulse response of an IIR filter is, in general, infinite. This explains why it is possible to obtain the same filter characteristics with IIR filters as with FIR filters while using much less taps. As an example, [RKHD81] concludes that a sixth-order filter is approximately equivalent to a 41-tap FIR filter.

Although \(a\), \(b\), \(N\), \(M\) and \(A\) together define a specific IIR filter, we will, from now on, denote \(IIR(a, b, N, M, A)\) as \(B\) in order to keep the formulas simple. Formula 7.1 then becomes:
\[ B(i) = \left( \sum_{n:0 \leq n < N} A(i-n) \times a(n) \right) + \left( \sum_{m:0 < m < M} B(i-m) \times b(m-1) \right) \]  
(7.2)

A graphical representation of an IIR filter is shown in figure 7.1

![Diagram of IIR Filter](image)

Figure 7.1: IIR Filter

As an example, the simplest nontrivial IIR filter has coefficients \( a_1 = -\frac{1}{2} \) and \( b_0 = \frac{1}{2} \) ([Roc03]):

\[ B(i) = \frac{1}{2} B(i-1) + \frac{1}{2} A(i) \]  
(7.3)

### 7.1.1 The order of an IIR Filter

The order of a FIR filter was defined earlier as the number of previous inputs used to generate an arbitrary output value. This definition is appropriate for non-recursive (FIR) filters, which use only the current and previous inputs to compute the current output. In the case of recursive filters, the definition can be extended as follows:

*The order of an IIR filter is the maximum of the number of previous input and the number of previous output values required to compute the current output.*

Thus, the order of IIR\((a, b, N, M, A)\) is given by

\[
\begin{align*}
M - 1, & \text{ if } N \leq M \\
N - 1, & \text{ if } N > M
\end{align*}
\]  
(7.4)

Clearly, this definition also applies to FIR filters and to all digital filters in general.

Finally, a filter depending only on the current input is called zero-order filter. Zero-order IIR filters are non-existent because such a filter would include no previous output and therefore be a non-recursive (FIR) filter.
7.2 ARMA-based vector implementations

In practice recursive filters usually require the same number of previous inputs and outputs and that is the type of IIR filter we will examine further later on in this chapter.

There are three different techniques that we will study. First we will create two ARMA-based implementations. These implementations are derived from the Auto-Regressive Moving Average representation of formula 7.2. Secondly we will study a block-state implementation based on a technique that has been developed by Hui-Hung Lu, Edward Ashford Lee and David G. Messerschmitt in their paper 'Fast Recursive Filtering with Multiple Slow Processing Elements' ([LLM85]).

7.1.2 A metric for comparison

Since we want to compare different algorithms that calculate the IIR filter, we need a metric. In order to keep our complexity measurements comparable to the ones found in the mentioned paper [LLM85], we will count the number of multiplications per output sample. This is the same metric we used in the previous chapter on FIR filters, when studying the theoretical performance of vectorized algorithms.

As an example, a FIR filter requires $N$ scalar multiplications to output a single value or $N \times L$ scalar multiplications to output a vector of $L$ output values. Calculating a single output of an IIR filter using to formula 7.2 requires $N + M - 1$ multiplications.

7.2 ARMA-based vector implementations

For this technique, we rewrite the ARMA definition of an IIR filter in such a manner that it is suitable for parallel computation on a vector processor. Because the general case results in complicated formulas, we first illustrate this by means of a simple example.

This is the example that we study in detail: a simple 2nd order IIR filter with an equal number of previous input and output values.

$$B(i) = \frac{1}{4} A(i) + \frac{1}{4} A(i - 1) + \frac{1}{2} B(i - 1)$$ (7.5)

In the two following subsections, we rewrite this example, using two different techniques, so that we can calculate 3 outputs simultaneously.

7.2.1 Technique 1: Full unfolding

Formula 7.5 needs to be rewritten until all simultaneously calculated terms are substituted. Since we need three values simultaneously, all indices of $B$ on the right-hand side of the formula must be at least 3 less than the one on the left-hand side.
\[ B(i) = \{ \text{Example 7.5} \} \]
\[
\frac{1}{4} A(i) + \frac{1}{4} A(i - 1) + \frac{1}{2} B(i - 1)
\]
\[
\frac{1}{4} A(i) + \frac{3}{8} A(i - 1) + \frac{1}{8} A(i - 2) + \frac{1}{2} B(i - 2)
\]
\[
\frac{1}{4} B(i - 2) = \frac{1}{16} A(i - 2) + \frac{1}{16} A(i - 3) + \frac{1}{8} B(i - 3)
\]
\[
\frac{1}{4} A(i) + \frac{3}{8} A(i - 1) + \frac{3}{16} A(i - 2) + \frac{1}{16} A(i - 3) + \frac{1}{8} B(i - 3)
\]

It is clear that the final definition of the given IIR filter is only based on information that is available before the calculation of three new output values: some input values and an output value at time instance \( i - 3 \).

With this information we can easily create the matrix form which calculates the three desired outputs at once by substituting \( i \) with \( i - 1 \) and \( i - 2 \):

\[
\begin{bmatrix}
B(i) \\
B(i - 1) \\
B(i - 2)
\end{bmatrix} = \begin{bmatrix}
\frac{1}{4} & \frac{3}{8} & \frac{3}{16} & \frac{1}{16} & 0 & 0 \\
0 & \frac{1}{2} & \frac{3}{8} & \frac{1}{16} & 0 & 0 \\
0 & 0 & \frac{1}{4} & \frac{3}{8} & \frac{1}{16} & \frac{1}{16}
\end{bmatrix} \begin{bmatrix}
A(i) \\
A(i - 1) \\
A(i - 2) \\
A(i - 3) \\
A(i - 4) \\
A(i - 5)
\end{bmatrix} + \begin{bmatrix}
\frac{1}{8} & 0 & 0 \\
0 & \frac{1}{8} & 0 \\
0 & 0 & \frac{1}{8}
\end{bmatrix} \begin{bmatrix}
B(i - 3) \\
B(i - 4) \\
B(i - 5)
\end{bmatrix}
\]

\[
(7.7)
\]

**Theoretical Performance**

As we have seen in the chapter on FIR filters, the multiplication of a vector with a banded matrix takes only as much multiplications as the width of the band. Therefore, in the general case, the number of multiplications necessary to calculate an output vector is defined as \( P + N - 1 \) for the non-recursive part and \( M \) for the recursive part or \( P + N + M - 1 \) in total.

**SVPL Implementation**

Below you find an implementation of this algorithm in SVPL. It is very similar to the implementation of the FIR filter algorithm. Initialization is not shown. \( A_{\text{base}} \) and \( B_{\text{base}} \) refer to the first memory locations where to find the coefficients of the recursive and non-recursive parts in RAM.
7.2 ARMA-based vector implementations

```c
a1 = 0;

while (a1 < IM.size()) {
    i1 = 0;
    a3 = A_base;
    result = Vector(0);

    in.load(IM, a1);
    a1 = a1 + VectorSize;

    coef.load(RAM, a3);

    // Nonrecursive part
    while (i1 < Atabs) {
        result.MAC(in, coef);

        Scalar s1;
        a3 = a3 + VectorSize;
        coef.load(RAM, a3);

        s1.load(IM, a1 + i1);
        in = in.shiftright(s1);
        i1++;
    }

    // Recursive part
    i1 = 0;
    a3 = B_base;

    in.load(OM, a2 - VectorSize);

    coef.load(RAM, a3);

    while (i1 < Btabs) {
        result.MAC(in, coef);

        Scalar s1;
        a3 = a3 + VectorSize;
        coef.load(RAM, a3);

        s1.load(OM, a2 + i1);
        v1 = v1.shiftright(s1);
        i1++;
    }

    result.store(OM, a2);
    a2 = a2 + VectorSize;
}
```
Resulting statistics are shown later in subsection 7.2.3.

7.2.2 Technique 2: Partial unfolding

By substituting the $i$ in formula 7.6 in order to calculate the formulas for $i - 1$ and $i - 2$, we need a lot of previous inputs and outputs. A lot of them can be eliminated by rewriting the formulas for $B(i)$, $B(i - 1)$ and $B(i - 2)$ separately and by only rewriting the necessary terms.

We keep the result for $B(i)$ from formula 7.6. Rewriting $B(i - 1)$ gives us:

$$B(i - 1) = \{ B(i) = \frac{1}{4} A(i) + \frac{1}{4} A(i - 1) + \frac{1}{2} B(i - 1) \}$$

$$\frac{1}{4} A(i - 1) + \frac{1}{4} A(i - 2) + \frac{1}{2} B(i - 2)$$

$$= \{ \text{Formula 7.9} \}$$

$$\frac{1}{4} A(i - 1) + \frac{3}{8} A(i - 2) + \frac{1}{8} A(i - 3) + \frac{1}{4} B(i - 3)$$

Note that this formula only contains 4 terms instead of 5. Rewriting $B(i)$ results in the following formula, which only contains 3 terms:

$$B(i - 2) = \{ B(i) = \frac{1}{4} A(i) + \frac{1}{4} A(i - 1) + \frac{1}{2} B(i - 1) \}$$

$$\frac{1}{4} A(i - 2) + \frac{1}{4} A(i - 3) + \frac{1}{2} B(i - 3)$$

Using the above results, we can create another matrix representation for the calculation of $B(i)$, $B(i - 1)$ and $B(i - 2)$.

$$\begin{bmatrix} B(i) \\ B(i - 1) \\ B(i - 2) \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & \frac{3}{15} & \frac{1}{18} \\ 0 & \frac{8}{15} & \frac{8}{18} \\ 0 & 0 & \frac{8}{4} \end{bmatrix} \begin{bmatrix} A(i) \\ A(i - 1) \\ A(i - 2) \end{bmatrix} + \begin{bmatrix} \frac{1}{2} \\ \frac{8}{3} \\ \frac{8}{2} \end{bmatrix} B(i - 3)$$

Theoretical Performance

We now no longer have banded matrixes, but the matrixes are smaller. As there is no easy and efficient way to multiply a vector with a triangular matrix, the multiplication is done completely, including a lot of multiplications with zero. This yields $P + N - 1$ vector multiplications for the non-recursive part and $M$ multiplications for the recursive part and a total of $(P + N + M - 1)$ multiplications, which is the same as in solution 1.
SVPL Implementation

```c
a1 = 0;

while (a1 < IM.size()) {
    i1 = 0;
    a3 = A_base;
    result = Vector(0);
    in.load(IM, a1);
    coef.load(RAM, a3);
    // Non-recursive part
    while (i1 < Atabs) {
        result.MAC(Vector(in), coef);
        a3 = a3 + VectorSize;
        coef.load(RAM, a3);
        ++i1;
        in.load(IM, a1 + i1);
    }
    // Recursive part
    i1 = 0;
    a3 = B_base;
    in.load(OM, a2);
    coef.load(RAM, a3);
    while (i1 < Btabs) {
        result.MAC(in, coef);
        a2 = a2 + VectorSize;
        coef.load(RAM, a2);
        i1++;
        in.load(OM, a2 + i1);
    }
    result.store(OM, a2);
    a1 = a1 + VectorSize;
}
```
While both techniques have the same mathematical complexity, there are at least two important differences:

Firstly, the number of previous input and output values necessary is different. Full unfolding uses more previously calculated values that the partial unfolding technique. Secondly, the resulting matrices from both techniques have a different structure. The matrix of full unfolding has the same values in every row of the band and is therefore less complex than the matrix of partial unfolding. In that matrix there is no such recognizable pattern.

These differences might have their impact on the numeric stability of both solutions, but this will not be explored in detail. More importantly, performance will probably not be affected by the differences shown above. Reading one scalar value takes the same amount of time as reading a full vector of values and secondly, of course the complexity of vectors has no impact on the speed of operations performed on them.

### 7.2.3 Comparison

The implementations of both unfolding techniques were executed for an IIR filter with order 3 and an input stream of 128 values. This is the result for the full unfolding technique:

<table>
<thead>
<tr>
<th></th>
<th>VECTOR</th>
<th>SCALAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory ops</td>
<td>176</td>
<td>96</td>
</tr>
<tr>
<td>Math ops</td>
<td>96</td>
<td>0</td>
</tr>
<tr>
<td>Modify ops</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>Total ops</td>
<td>388</td>
<td>96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>INTEGER</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total ops</td>
<td>256</td>
<td>404</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>SCHEDULER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total signals</td>
<td>1236</td>
</tr>
<tr>
<td>Total operations</td>
<td>834 (100 %)</td>
</tr>
<tr>
<td>Total cycles</td>
<td>563</td>
</tr>
<tr>
<td>Total unit usage</td>
<td>37 %</td>
</tr>
<tr>
<td>Vector unit usage: 17% (excl. Int ops): 23% (466 lost cycles)</td>
<td></td>
</tr>
<tr>
<td>(305 lost cycles)</td>
<td></td>
</tr>
</tbody>
</table>

And these are the statistics for partial unfolding:

<table>
<thead>
<tr>
<th></th>
<th>VECTOR</th>
<th>SCALAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory ops</td>
<td>144</td>
<td>128</td>
</tr>
<tr>
<td>Math ops</td>
<td>96</td>
<td>0</td>
</tr>
<tr>
<td>Modify ops</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Total ops</td>
<td>240</td>
<td>128</td>
</tr>
</tbody>
</table>
7.2 ARMA-based vector implementations

<table>
<thead>
<tr>
<th>INTEGER</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total ops: 256</td>
<td>Total ops: 404</td>
</tr>
</tbody>
</table>

SCHEDULER

<table>
<thead>
<tr>
<th>Total signals</th>
<th>1044</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total operations</td>
<td>834</td>
</tr>
<tr>
<td>(100 %)</td>
<td></td>
</tr>
<tr>
<td>Total cycles</td>
<td>463</td>
</tr>
<tr>
<td>Total unit usage</td>
<td>43 %</td>
</tr>
<tr>
<td>Vector unit usage: 19%</td>
<td>386 lost cycles</td>
</tr>
<tr>
<td>(excl. Int ops): 27%</td>
<td>257 lost cycles</td>
</tr>
</tbody>
</table>

Like we already expected, there are no big differences in performance between both algorithms. The total number of operations is even exactly the same. It must be said however that it might still be possible to increase performance for both or only one of the algorithms by applying extra optimizations. For the algorithms as shown above, the second one is 14% faster than the first one as it uses 80 cycles less thanks to a 6% higher unit usage.

It is now time to draw some early conclusions. It seems like we rewrote the IIR filter into a form that is perfectly suitable for implementation on a vector processor. Both solutions can easily be implemented on the SVP.

There is however one major drawback to the ARMA approach that we already can deduce. The width of the matrices (and therefore the number of multiplications needed per output vector) is directly related to the number of simultaneous outputs (vector length). This means that this solution cannot be used to create efficient vector implementations for arbitrary amounts of parallelism. Nevertheless, let us not immediately write off this implementation. It might still be the most optimal solution for specific degrees of parallelism. We will investigate this matter further at the end of this chapter where we will compare all techniques.

7.2.4 Determining the matrix constants

Apparently, the efficiency of an ARMA solutions does not depend on the actual values of the matrix constants. Moreover, it is difficult to present a closed formula for these constants. Therefore, in this section an easy way to calculate these constants is given. The technique is not formally proven, but rather illustrated by use of a representative example.

Consider the formula:
\[ B(i) = \langle \sum j : 0 \leq j < N : c_j \times A(i - j) \rangle + \langle \sum p : 0 < p < 4 : d_p \times B(i - p) \rangle \]
\[ = \langle \sum j : 0 \leq j < N : c_j \times A(i - j) \rangle + d_1 B(i - 1) + d_2 B(i - 1) + d_3 B(i - 1) \]
(7.11)

We will now show how this formula should by unfolded and how the matrix constants are to be calculated on a processor with vector size 4.

Because of this vector size, we need to calculate \( B(i) \), \( B(i - 1) \), \( B(i - 2) \) and \( B(i - 3) \) simultaneously. For the calculation of \( B(i) \), this means that we need to rewrite the formula until all occurrences of \( B(i - 1) \), \( B(i - 2) \) and \( B(i - 3) \) are eliminated. We do this in a few steps:

\[ B(i) = \langle \sum j : 0 \leq j < N : c_j \times A(i - j) \rangle + d_1 B(i - 1) + d_2 B(i - 1) + d_3 B(i - 1) \]
\[ = \langle \sum j : 0 \leq j < N : c_j \times A(i - j) \rangle \\
+ d_1 \left[ \langle \sum j : 0 \leq j < N : c_j \times A(i - j - 1) \rangle + d_1 B(i - 2) + d_2 B(i - 3) \right] \\
+ d_2 \left[ \langle \sum p : 2 \leq p < 4 : d_p \times B(i - p - 1) \rangle \right] + d_3 \left[ \langle \sum j : 0 \leq j < N : c_j \times A(i - j - 2) \rangle + d_1 B(i - 3) \right] \\
+ d_2 \left[ \langle \sum p : 1 \leq p < 4 : d_p \times B(i - p - 2) \rangle \right] + d_3 \left[ \langle \sum j : 0 \leq j < N : c_j \times A(i - j - 3) \rangle \right] + \langle \sum p : 0 < p < 4 : d_p \times B(i - p - 3) \rangle \]
(7.12)
\[ \langle \sum_{j:0 \leq j < N} c_j \times A(i-j) \rangle \]
\[ = \begin{align*}
+ d_1 & \left[ \langle \sum_{j:0 \leq j < N} c_j \times A(i-j-1) \rangle \\
+ d_2 & \left[ \langle \sum_{j:0 \leq j < N} c_j \times A(i-j-2) \rangle \\
+ d_3 & \left[ \langle \sum_{j:0 \leq j < N} c_j \times A(i-j-3) \rangle \\
+ d_4 & \left[ \langle \sum_{p:0 < p < 4} d_p \times B(i-p-1) \rangle \\
+ d_5 & \left[ \langle \sum_{p:1 \leq p < 4} d_p \times B(i-p-2) \rangle \\
+ d_6 & \left[ \langle \sum_{p:2 \leq p < 4} d_p \times B(i-p-3) \rangle \\
+ d_7 & \left[ \langle \sum_{p:3} d_p \times B(i-p-4) \rangle \\
\end{align*} \] (7.13)

So far we have recursively rewritten the definition of \( B(i) \) in order to remove all occurrences of \( B(i-1) \), \( B(i-2) \) and \( B(i-3) \). We want to define for each values of \( A \) and \( B \) which constant value it is multiplied with. Therefore we group all recurring formulas together:

\[ = \begin{align*}
d_1 & \langle \sum_{j:0 \leq j < N} c_j \times A(i-j) \rangle \\
+ (d_1^2 + d_2) & \langle \sum_{j:0 \leq j < N} c_j \times A(i-j-1) \rangle \\
+ (d_1^3 + d_1 d_2 + d_2 d_1 + d_3) & \langle \sum_{j:0 \leq j < N} c_j \times A(i-j-2) \rangle \\
+ d_1 & \langle \sum_{p:2 \leq p < 4} d_p \times B(i-p-1) \rangle \\
+ (d_1^2 + d_2) & \langle \sum_{p:1 \leq p < 4} d_p \times B(i-p-2) \rangle \\
+ (d_1^3 + d_1 d_2 + d_2 d_1 + d_3) & \langle \sum_{p:0 < p < 4} d_p \times B(i-p-3) \rangle \\
\end{align*} \] (7.14)

This result can also be written as:

\[ B(i) = \langle \sum_{x:0 \leq x < 4} \left( \langle \sum_{a,b,c:0 \leq a,b,c < x} a + b + c = x : d_a d_b d_c \rangle \right) \]
\[ + \langle \sum_{j:0 \leq j < N} c_j \times A(i-j-x) \rangle \]
\[ + \langle \sum_{p:4-x-1 \leq p < 4} d_p \times B(i-j-x) \rangle \] (7.15)

And from this formula the necessary constants can easily be computed.
The calculations above give a good idea of how the matrices can be calculated. Since the general case gets even more complicated, it is not included in this document. The last important remark is that you only need to unfold from $B(i - 1)$ to $B(i - P + 1)$ or in general, from $B(i - 1)$ to $B(i - (P|M) + 1)$.

### 7.3 Block State vector implementation

This section shortly describes the method introduced in [LLM85].

The technique has been developed to implement a high rate IIR filter on a VLSI chip with multiple slow processing elements. It is shown that an IIR filter can be implemented using large amounts of concurrency. This means that an IIR filter can be created to output exact one vector at a time. Of course we still need to examine whether this algorithm is suitable for implementation on a vector processor.

In this section we will define $O$ the order of the IIR filter and take $P$, the vector size as the number of concurrent outputs.

Any digital filter, non-recursive or recursive can be rewritten to a state variable form. A state variable form is composed of two equations: the output equation and the state update equation:

$$\begin{align*}
x(n + 1) &= Ax(n) + bu(n) \\
y(n) &= cTx(n) + du(n)
\end{align*} \tag{7.16}$$

It turns out that the above formula can be rewritten for an unlimited number of concurrent outputs:

$$\begin{align*}
y^{(P)}(kP) &= Cx(kP) + Du^{(P)}(kP) \\
x((k + 1)P) &= A^Lx(kP) + Bu^{(P)}(kP)
\end{align*} \tag{7.17}$$

With $A$ a lower triangular $O \times O$ matrix, $B$ full $O \times P$, $C$ full $P^*O$ and $D$ lower triangular $P \times P$. It is shown that these equation exist and can be computed. Finally as a metric for complexity, the number of minimum required multiplications is:

$$\frac{O(O + 1)}{2} + 2PO + \frac{P(P + 1)}{2} \tag{7.18}$$

Too bad there is no generic efficient way to calculate the triangular matrix products on a vector processor. This means that the final number of multiplication will rather be:

$$O(O + 1) + 2PO + P(P + 1) \tag{7.19}$$
7.3 Block State vector implementation

As a sixth-order IIR filter is approximately equivalent to a 41-tap FIR filter as shown by Rabiner et al., both complexities can be displayed in a graph for fixed $O$ and varying $P$. When we do so, we found out that there is an error in the Fig. 12 in [12] on page 1126. Figure 7.2 shows us what the correct graph should look like.

![Graph showing comparison between IIR and FIR complexities]

Figure 7.2: Plot of the number of multiplications per output sample as function of the vector size $P$ for a sixth-order IIR filter and a 41-order FIR filter. This plot shows that above $P = 51$ the FIR filter is less complex than the IIR filter.

The above complexities are however too optimistic since it is not possible to implement the block-state version of the IIR filter with the same efficiency as we did previously with the FIR filter. Nevertheless the graph indicates that we need a fairly large amount of concurrency before the FIR-filter implementation gets more efficient than the IIR filter.

The major drawback of this algorithm is that it is impossible to implement it on a vector processor with no loss of efficiency. First of all, the multiplication of triangular matrices yields a lot of multiplications with zero that cannot easily be avoided. And secondly, the calculation of the term $A^T \bar{x}(kP)$ of the state update function is difficult to implement efficiently on the vector processor since $O$ is independent of the vector size.

These remarks are to be taken into consideration when you read the next section where we compare the block state solution with the previous solutions.
7.4 Comparison

We will only look at one specific type of IIR filters. This will make it easier to compare the different solutions as the number of variables is kept limited. We will only look at those filters for which \( N = M = O + 1 \). This gives following formulas for the number of multiplications per output value:

Both ARMA solutions:

\[
\#_{\text{ARMA}}(P, N, M) = P + N + M - 1 = P + 2N - 1 \tag{7.20}
\]

Block-state solution:

\[
\#_{\text{BS}}(O, P) = \frac{O(O + 1)}{2P} + \frac{2PO}{P} + \frac{P(P + 1)}{2P} + \left[\frac{O + 1}{2}\right] \tag{7.21}
\]

This is the same graph as above (sixth-order filter), with the function of the ARMA solutions added. It shows the number of multiplications needed for FIR and both IIR solutions in function of the vector size. At the intersection points, solutions of intersecting functions are equally suitable.

![Graph showing comparison between ARMA, Block-State, and FIR solutions](image)

Figure 7.3: The number of multiplications per output sample \( (M) \) necessary to calculate a 6th IIR filter of order for a different amounts of concurrency (vector size \( L \)). Results are shown for the FIR, ARMA and block-state solutions.

As I did not find any other comparisons of IIR and FIR filters with regard to the number or tabs needed, the number of multiplications of a comparable FIR function were not added to the next graphs.
In figure 7.4 the results of a 15th order IIR filter are shown.

![Graph showing number of multiplications per output sample (M) necessary to calculate a 15-th order IIR filter of order for a different amounts of concurrency (vector size L). Results are shown for ARMA and block-state solutions.]

As you can see, for increasing filter order, the advantage of the block state solution over the ARMA solutions diminishes.

The points of intersection from the graphs above are the critical points where most efficient solution changes from one to another. Therefore, figure 7.5 show the most interesting graph, as it shows those points of intersection as a function of the filter order (on the horizontal axis). It shows for a certain filter at which vector size (vertical axis) the block state solution is faster than the ARMA solutions.

Although these graphs speak in favor of the block-state solution, it is very important to note that while the ARMA solutions are tailored for use on a vector processor, the block-state implementation is not! The matrix multiplications that occur in the block-state solution have different dimensions. These are the worst-case scenarios:

\[ O < P: \text{ We can only calculate matrix multiplications in blocks of } P \text{ efficiently, so even if } O \text{ is smaller than } P, \text{ this is the result:} \]

\[
\text{Number of multiplications per output value} = \frac{PP}{P} + \frac{PP}{P} + \frac{PP}{P} = 4P \quad (7.22)
\]
Figure 7.5: This graph shows the for each IIR filter order (N) which vector size (L) is at least needed for the block-state solution to be better than the ARMA solution. Or the other way around: up till which vector size (L) should the ARMA solution be chosen over the block-state solution for a given IIR filter order (N).

This result is clearly worse than that of the ARMA solutions. When $O > P$, things get less worse, depending on $O$ and $P$ on the possible optimizations that can be applied with regard to the calculation of triangular matrices.
7.5 Conclusion

In this chapter we explored to different approaches to vectorize the IIR filter algorithm. We compared two solutions that are based on the ARMA representation of the IIR filter with a well known and often-cited technique based on the block-state representation of the filter.

Although the block-state solution seemed a very promising solution, its results are much worse than expected. In contrast to both ARMA solutions, the block-state solution was developed for use in hardware circuits and not for use on a vector processor. While every part of the ARMA technique is designed for efficient computation by the vector processor, the block-state technique is not.

Not only did we explore the qualitative differences of both solutions, we also defined the quantitative difference and discovered the conditions that make each of the techniques the most optimal one.
Chapter 8

Summary

In this document we indicated the ever growing need for faster DSP chips and the emerging problem of hardware not able to keep up. The use of vector processor techniques was shown to be a viable solution because it can combine both SIMD and superscalar processor techniques.

In order to study the vectorization of a few of the most common DSP algorithm kernels, we developed a simple vector processor architecture and designed a matching programming language, SVPL of Simple Vector Processing Language. As this language alone was not enough for studying the algorithms, we also designed a C++ library that does not only support this new language but also provides an extendible set of tools for debugging and optimization.

For all combinations of the first set of algorithms, the upsample, downsample and FIR filters, we designed vectorized versions that can deliver a speedup of 100% when compared to their scalar counterparts.

For the second algorithm, the IIR filter, we did not find such an optimal solution. Instead we compared an existing technique, based on the block-state representation of an IIR filter, with two new techniques that are based on the ARMA representation. We found that it depends on the filter parameters, which of the techniques is the most optimal.

Finally, we found that the SVP model and the default instruction scheduler in the SVPL library are open for many improvements. These support more efficient handling of control and addressing overhead and can be found in today's more advanced DSP chips. Luckily, these improvements can easily be applied, thanks to the open design of the SVPL library.
Bibliography


Appendices
Appendix A

Practical UP-FIR-DN example: Image Interpolation

A.1 Introduction

In this section we will see a practical example of the UP-FIR implementation. Adding the DN filter is oblivious and therefore we will only interpolate with integer factors in this example. As mentioned before, the UP-FIR(DN) is most often used in resampling audio and image data. Of course the latter is most appealing for demonstration in this document and therefore we will search FIR filter parameters to resample a test image. Simply out of interest, we will also briefly examine the errors generated by this transformation.

We will interpret image data as a sampled data stream in time. An image passes through the filter, line per line, and every line from the left to the right. This will clearly introduce errors at the edges of the image because certain edge values will be computed from values at the end of a line as well as from the beginning of the next line. It is not the purpose of this paper to explain techniques to eliminate them.

Another problem that arises from handling the image data as a one-dimensional stream is that we can only interpolate the image in one direction. Luckily the interpolation filters we will use are separable, which means that both dimensions can be dealt with separately. To get a resampled image with correct aspect ratio, the image will be fed into the filter in one direction and the output thereof will be rotated by 90 and fed again to the same filter to resample the other dimension. This process is illustrated in figure A.1:

A.2 The Filter Algorithm

Since our UP-FIR-DN filter always performs upsampling as first part of the resampling process, we need an interpolation filter to compute the FIR tabs
from. There is an enormous variety in interpolation filters, most of them with only slightly different characteristics.

A.2.1 Selecting an appropriate algorithm

Filters are evaluated on their ability to retain detail in the passband (sharpness is valued more than blurriness) and to eliminate aliasing in the stopband (smoothness is valued more than jaggedness). Since the perfect finite filter does not exist, you always need to choose between sharpness and smoothness.

Every interpolation filter is a low-pass filter, this implies that only low-frequency signals are allowed through the output. An ideal low-pass filter with a cutoff frequency of 50 Hz is demonstrated in figure A.2:

As indicated in [Tur90] and [OS75] the ideal low-pass filter is the sinc function. It can be shown that it exactly reproduces the unique continuous signal that passes through the pixel corners with the correct pixel values and is bandwidth limited to the Nyquist limit for the pixel samples.

If the above frequency response is inverse Fourier transformed, the result will be the impulse response of the filter in the time domain. The result hereof is a sinc function centered at the origin: (figure A.3)
Figure A.2: An ideal low-pass filter

Figure A.3: The sinc function
Two problems arise when using this filter: in the first place the filter is infinite so that we need to truncate it and secondly this filter is non-causal which means that future inputs are needed to calculate the current value and this automatically introduces a delay. Of course this small delay is only of importance when working in real-time situations, but nevertheless it something to take into account when implementing the filter.

The problem with sinc is that it requires a large kernel width for good results, and so takes a long time to calculate. So we multiply it by another function that goes to zero outside a small window. A good choice of this function still gives pretty good results but is much quicker to calculate with. There are many different truncated or windowed sinc functions. The sinc-approximating kernel that produces best results, depends heavily on the application.

The lanczos-windowed sinc function is found most suitable for 2D-graphics by [Tur90]. In [MNPV99] and [MN01] that discuss tests of different kernels for medical image interpolation, the Lanczos2 function is also found to be one of the best kernels available.

This is the Lanczos2 windowed sinc function:

\[
\text{Lanczos2}(x) = \begin{cases} 
\frac{\sin(\pi x)}{\pi x} \cdot \frac{\sin(\pi |x|/2)}{\pi |x|/2}, & |x| < 2 \\
0, & |x| \geq 2
\end{cases}
\]  

The lanczos2 function is also graphically illustrated in figure A.4.

![Figure A.4: The Lanczos2 function](image)

There is also a lanczos3 function that we will not discuss, but you can remember that it uses a window of width 6 instead of 4, which makes it slightly more correct at the expense of much higher computational complexity. Lanczos2 offers the best trade-off between accuracy and computational cost.
A.2 The Filter Algorithm

The frequency response of a sampled filter is always rather different from the frequency response of the original function, in this case the sinc function. The following graph illustrates the frequency response of the lanczos2 filter and the original sinc function. (Remember that the frequency response of the sinc function drops from 1 to 0 at 0.5):

![Frequency Response Graph](image)

Figure A.5: Frequency Response of Lanczos2 and Sinc

A.2.2 How to use the Lanczos2 filter?

Now to use this function, we need to sample it, depending on the upsample factor. Since the truncated sinc function Lanczos3 is only 4 samples wide ([-2,2]), we can only sample it at 4 points, except when sampling it with phase 0, then there are 5 samples: 0,0,1,0,0:

![Sampled Lanczos2 Function](image)

Figure A.6: The Lanczos2 function sampled with phase 0

An interpolation kernel should always return the input values when applied to those values with a phase shift and indeed, if we use the fir filter 0,0,1,0,0 on an input stream, we get a new stream with the same values. But since we are generating an upsampled stream, we also need to calculate the intermediate values. To do so, we sample the lanczos2 function with different phase.
Suppose we want to upsample an input stream by factor 4, then we need to sample the lanczos2 function with phases 0, 0.25, 0.75, 1. This results in the following filter when we limit the precision of the results and adjust the filter afterwards. The first and the last sampled value are always zero so we don’t include them in our filter.

\{
-0.018, -0.073, -0.084, 0, 0.233, 0.573, 0.869, 1, \\
0.869, 0.573, 0.233, 0, -0.084, -0.073, -0.018
\}

The sampled lanczos2 function is illustrated in figure A.7.

![Sampled lanczos2 function](image)

Figure A.7: Sampled lanczos3 function at 4 different phases.

The adjustment mentioned above is to make sure that all values that are 3 samples apart (in general: \((factor - 1)\) apart) exactly add up to 1 by changing them slightly but still keeping the symmetry. This is necessary because otherwise interpolated values lose a bit of information.

### A.3 The results

Using the SVPL implementation of the UP-FIR-DN filter, we can apply this filter to an example image. This is the source image:

![Source image](image)

The small image is the original, but we also scaled the original by a factor 5 (simple pixel resize), in order to make it easier to compare the source image with the resampled images below. It will also become clear that interpolation can really increase visual quality of an image.
A.3 The results

Figure A.8: The original image: real size (left) and scaled (right)

A.3.1 Resampled images

This is the resulting image from a factor 5 resize using the above implementation. Border errors are not shown, since this is only a small part of a much larger image:

Figure A.9: Image resized using Lanczos2

And this is the same image, resized to 500% by Paint Shop Pro 8 (PSP) using the 'Smart Resize' option. It should be clear that visual quality is much less than with our Lanczos2 implementation. PSP produces an image that appears to have what can best be described as JPEG artifacts, small square blocks all over the images.
A.3.2 Calculating the Difference

Although it is clear that the lanczos2 method provides better visual quality, I planned to make a quantitative comparison of the lanczos2 and PSP resize methods. The best technique to measure the accurateness of a resampling algorithm is to compare the original with a resampled one. To do so, you should upscale a source image by a non-integer factor (in this case 3/2) and then downscaling it by the same factor. Both source and resulting image will have the same size can be compared on a pixel-by-pixel basis and the average pixel deviation can be calculated.

The comparison between lanczos2 and PSP could not take place because it turned that Paint Shop Pro cuts away a few, but undefined, number of pixels from the borders of an image every time it resizes it. This is probably done to avoid the border issues we discussed earlier. So what happens exactly? Suppose you have a 100 x 100 pixel image and want to resize it to 150 x 150, PSP resizes this image to a slightly higher resolution and then crops it to the expected resolution. As I couldn't find out the exact way PSP handles this, it was impossible to anticipate on the situation, for instance by entering a slightly smaller destination resolution.

Nevertheless, I applied the comparison technique to the lanczos2 algorithm. In figure A.3.2 the source image (from which the detail we used earlier was taken), the resulting image after upscaling by factor 3/2 and downscaling by 2/3 and the resulting difference image are shown. The difference image was inverted to be more photocopy-friendly.

As you can see, most errors are situated at strong color transitions and the difference image looks much like the result of a edge detection algorithm. Since all errors are situated in those regions, the human eye can hardly notice them as the errors are still relatively small compared to the actual color transitions. Reason for the errors is a combination of a shift that occurs because of the non-
A.3 The results

Figure A.11: From left to right: original, resampled and difference image

Integer scaling factors and the errors introduced by the windowed sinc kernel (recall that the frequency response of the lanczoe2 function is not perfect).

It is also possible to express the error in numerically. Remember that we are working with grayscale images and that pixel values (indicating lightness) range from 0 (black) to 255 (white). Results were gathered by a small homemade statistical program.

<table>
<thead>
<tr>
<th>Image Size (pixels)</th>
<th>= 120 x 240</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total pixels</td>
<td>= 28800</td>
</tr>
</tbody>
</table>

- Original Image -
  Total lightness     = 2.9614e+06
  Average pixel lightness = 102.827

- Resampled Image -
  Total lightness     = 2.92918e+06
  Average pixel lightness = 101.708

Difference in average pixel lightness = 1,119 (0.4 %)

- Difference image -
  Total pixel error   = 138368
  Average pixel error = 4.80444 (1.8 %)
As a final remark, we can conclude from the above statistics that the resampled image was darkened by negligible 0.4% and that the average pixel error stays well within reasonable boundaries (1.8%).
Appendix B

SVPL Class Diagram
Appendix C

Header Files

C.1 ProcessorStatistics.h

#ifndef INC_PROCSTATS
#define INC_PROCSTATS

#include <string>
#include <iostream.h>
#include "Debug.h"
#include "ProcessorDataType.h"

using namespace std;

/**
 * ProcessorStatistics
 *
 * Abstract class for calculating SVP statistics.
 * print, reset and signal are pure virtual functions
 * and need to be implemented by the derived class.
 */
namespace SVP {

class ProcessorStatistics {

public:
    ProcessorStatistics(DebugStream &d) : debug(d) {fEnabled = false;}
~ProcessorStatistics() {};

    enum FlagType
    {
        // These define the object(s) of operation
        NONE = 0x00,
        SCALAR = 0x01,
        VECTOR = 0x02,
        ADDRESS = 0x04,
    };
}
INTEGER = 0x08,

// These define the nature of the operation
MATH = 0x10,    // +, -, *, /, <, >, ==, !=, <=, >=
MEMORY = 0x20,
VECTORSHUFFLE = 0x40,    // Shuffle
VECTORSHIFT = 0x80,     // Shift
VECTORMODIFIER = 0xC0,   // Shuffle OR shift (0xC0 = 0x40 | 0x80)
ASSIGNMENT = 0x100

);

virtual void print() = 0;
virtual void reset() = 0;
virtual void signal( const int flags,
    const ProcessorDataType* op1 = 0,
    const bool writeOp1 = false,
    const ProcessorDataType* op2 = 0,
    const bool writeOp2 = false,
    const ProcessorDataType* op3 = 0,
    const bool writeOp3 = false
) = 0;

/* to temporary disable counting
   (e.g. for system classes while doing uncounted math
    or for SVFL program debugging purposes)
   returns previous state */
bool enable(bool b) {bool prev = fEnabled;
    fEnabled = b; return prev;};

protected:
    DebugStream &debug;
    bool fEnabled;

private:
    ProcessorStatistics();
};
}

#endif /* INC_PROCSTATS */
List of Figures

2.1 SVP Architecture: processor and I/O units .......................... 9
4.1 Simplified class diagram of the SVP library .......................... 30
4.2 SVP Layout .................................................................... 38
6.1 Fir Filter ......................................................................... 63
6.2 Upsample + Fir Filter ......................................................... 68
6.3 Fir + Downsample Filter ...................................................... 84
6.4 Fir + Downsample Filter: l < N ................................................ 84
6.5 Fir + Downsample Filter: l \geq N ............................................. 84
6.6 The Scaler ....................................................................... 87
7.1 IIR Filter ......................................................................... 92
7.2 Multiplications per output sample of 6th order filter ................ 103
7.3 Multiplications per output sample of 6th order filter ................. 104
7.4 Multiplications per output sample of 15th order filter ................ 105
7.5 Turning points for block-state and ARMA solution ..................... 106
A.1 Interpolating an image in two dimensions .............................. 118
A.2 An ideal low-pass filter ....................................................... 119
A.3 The sinc function ............................................................... 119
A.4 The Lanczos2 function ......................................................... 120
A.5 Frequency Response of Lanczos2 and Sinc ............................. 121
A.6 The Lanczos2 function sampled with phase 0 ......................... 121
A.7 Sampled lanczos3 function at 4 different phases ....................... 122
A.8 The original image: real size (left) and scaled (right) ............... 123
A.9 Image resized using Lanczos2 .............................................. 123
A.10 Image resized using Paint Shop Pro’s ‘Smart Resize’ ............... 124
A.11 From left to right: original, resampled and difference image ........ 125
B.1 SVPL Class Diagram .......................................................... 128