A Domain Specific DSP Processor

Eric Tell

Reg nr: LiTH-ISY-EX-3209

Supervisor: Mikael Olausson
Examiner: Dake Liu

Linköping 2001
Abstract

This thesis describes the design of a domain specific DSP processor.

The thesis is divided into two parts. The first part gives some theoretical background, describes the different steps of the design process (both for DSP processor design in general and for this project) and motivates the design decisions made for this processor.

The second part is a nearly complete design specification.

The intended use of the processor is as a platform for hardware acceleration units. Support for this has however not yet been implemented.
Contents

I Design of a Domain Specific DSP Processor  5

1 Introduction  6
  1.1 Purpose of this processor  6
  1.2 Reading guidelines  6

2 DSP vs. General Purpose Processors  8
  2.1 Architecture  8
  2.2 The MAC Unit  8
  2.3 Saturation Arithmetic  9
  2.4 Special Addressing Modes  9
    2.4.1 Modulo Addressing  10
    2.4.2 Bit-Reversed Addressing  10
  2.5 Hardware Looping  11
  2.6 Different Types of DSP Processors  11

3 The Design Flow  12
  3.1 Requirement Analysis  12
  3.2 Instruction Set Design and Architecture Planning  12
  3.3 Behavioral Model  13
  3.4 Benchmarking  13
  3.5 Architecture Design  14
  3.6 RTL Implementation  14
  3.7 Verification  14

4 Instruction Set Analysis  16
  4.1 Choosing The Instruction Set  16
  4.2 This Processor  17
11 Verification
11.1 The Verification Strategy .......................... 40
11.2 Verification for This Project ...................... 41
   11.2.1 Block Level Verification .................... 41
   11.2.2 Instruction Level Verification ............... 41
   11.2.3 Random Testing ................................ 42
   11.2.4 Application Level Verification ............... 42

12 Conclusions and Future Improvements 44
12.1 Results and Conclusions .......................... 44
12.2 Alternative Solutions ............................. 44
12.3 Future Improvements .............................. 45
   12.3.1 Interrupts .................................. 45
   12.3.2 I/O Ports ................................... 46
   12.3.3 Additional Instructions ...................... 46
   12.3.4 Hardware Accelerator and Multiprocessor Support 46

II Design Specification 47
13 Introduction 48
   13.1 Processor Features .............................. 48
   13.2 Outline of This Part of the Thesis .............. 49

14 Data Path 50
   14.1 Architecture Overview ........................... 50
   14.2 Arithmetic Unit .................................. 51
   14.3 Shift Unit ....................................... 52
   14.4 Logic Unit ....................................... 52
   14.5 MAC Unit ........................................ 53
      14.5.1 Rounding .................................. 54
      14.5.2 Saturation Unit ............................. 54
   14.6 Register File .................................... 54
      14.6.1 The Accumulator Registers ................ 57
      14.6.2 The Control Register ....................... 57
   14.7 Addressing ..................................... 60
      14.7.1 Addressing Modes ......................... 60
      14.7.2 Modulo Addressing ......................... 61
Part I

Design of a Domain Specific DSP Processor
Chapter 1

Introduction

DSP is an abbreviation for Digital Signal Processing. Accordingly, a DSP processor is a processor that is designed specifically for signal processing tasks.

The purpose of the project described in this thesis, was to design a 16-bit fixed point DSP Processor.

The Project includes all steps from instruction set analysis, machine code design and architecture planning, via a C++ behavioral model to register transfer level implementation using VHDL.

1.1 Purpose of this processor

The processor is intended to be used as a platform for hardware accelerators. That is, it should be possible to easily connect application specific hardware units to the processor core. For this reason the instruction set of the processor is quite simple and instead space for adding hardware accelerator instructions has been reserved.

Furthermore the processor is intended to be used in a system of up to four similar processors that should be able to share memory.

Due to the limited time available for this project however, the actual hardware for supporting hardware accelerators and multiple processors has not yet been implemented (although it is supported by the instruction set).

1.2 Reading guidelines

The thesis is divided into two parts. The first part gives some background to DSP processor design and describes the design flow and the design decisions made in
this project. The second part is a more or less complete design specification for
the processor.

The first part has the following contents:

**Chapter 2** describes the special features of DSP processors that separates them
from general purpose processors.

**Chapter 3** gives an overview of the design process.

**Chapters 4 to 11** goes into details on different parts of the design.

**Chapter 12** has some conclusions and proposals for future improvements.

Many of chapters 4 to 11 have a corresponding section in part two of the report
and the reader may want to look ahead to see the actual implementation of some
part of the processor, before continuing to the next chapter.
Chapter 2

DSP vs. General Purpose Processors

This chapter describes some important differences between DSP processors and general purpose processors. It also explains some concepts that are used later in the report.

2.1 Architecture

The most important difference between the architecture of DSP processors and general purpose processors is probably the possibilities for multiple memory accesses in one clock cycle. Generally a DSP processor has separate program and data memories. This allows the processor to fetch an instruction, while simultaneously fetching operands or storing results for a previous instruction. Often it is also possible to fetch multiple data from memory in one clock cycle by using multiple busses and multi port memories or multiple independent data memories.

2.2 The MAC Unit

The single most typical feature of DSP processors is the dedicated hardware for multiply-and-accumulate or MAC operations. The MAC operation is used for calculating a sum of products - two operands are multiplied and the product is added (or subtracted) to a cumulative sum. The MAC operation is very common in DSP applications and is used for example for vector products, digital filters, correlation and Fourier transforms.

Usually the operands in the addition/subtraction has more bits than the output of the multiplication. The extra bits are called guard bits. The guard bits makes it
possible to accumulate a number of values without the risk of overflow. If $n$ guard bits are used $2^n$ values can be accumulated without the possibility of overflow. Most DSP processors have four or eight guard bits.

The MAC hardware usually also supports saturation (see 2.3 below) and rounding, to get a result of the native data width\(^1\).

**Example:** If the native data width is $n$, then the result of the multiplication will have $2n$ bits. So with $m$ guard bits the result of the MAC operation will have $2n + m$ bits. This value can be saturated to a $2n$-bit value and then rounded to get a value of the native data width $n$, that can be stored in memory or used in other kinds of operations.

### 2.3 Saturation Arithmetic

Normally if the result of an arithmetic operation in a hardware unit is outside the data range the result will “wrap around”. For example if one is added to the highest possible number, the result will be the lowest possible number. Saturation on the other hand means that if the real result is larger than what can be represented with the available number of bits, the output will be the highest possible value and if the result is lower than the lowest value that can be represented, the result will be the lowest possible value. The difference is illustrated in figure 2.1.

Using saturation arithmetic reduces distortion due to overflow and may also prevent parasitic oscillations in recursive algorithms [5].

Saturation arithmetic is basically always supported for the MAC operations in DSP processors and sometimes also for other operations (like addition and subtraction).

### 2.4 Special Addressing Modes

Addressing modes of DSP processors are chosen to fit the applications. The most common memory addressing mode is register indirect addressing with post-increment, which is used to execute repetitive operations on data stored sequentially in memory. Two other special addressing methods common in DSP processors are described below.

---

\(^1\)The native data width is the width of data memory and most busses
2.4.1 Modulo Addressing

Modulo addressing (or circular addressing) means that when a memory pointer (used for example for post-increment addressing) reaches the end of a specified memory area, it automatically starts over from the beginning. This can be used for example for implementing circular data buffers.

2.4.2 Bit-Reversed Addressing

The bit-reversed addressing mode is used specifically for implementing the fast Fourier transform (FFT) algorithm. The problem with the FFT algorithm is that it either takes its input or leaves its output in a scrambled order, so at some point the order of the data has to be rearranged.

The most common form of the FFT requires the data to be taken in bit-reversed order. The term bit-reversed comes from the fact that the ordering matches the output one would get from a binary counter if the bits where taken in reversed order (that is the least significant bit first). This is illustrated below

<table>
<thead>
<tr>
<th>Normal order</th>
<th>Bit-reversed order</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 = 0</td>
<td>000 = 0</td>
</tr>
<tr>
<td>001 = 1</td>
<td>100 = 4</td>
</tr>
<tr>
<td>010 = 2</td>
<td>010 = 2</td>
</tr>
<tr>
<td>011 = 3</td>
<td>110 = 6</td>
</tr>
<tr>
<td>100 = 4</td>
<td>001 = 1</td>
</tr>
<tr>
<td>101 = 5</td>
<td>101 = 5</td>
</tr>
</tbody>
</table>
2.5 Hardware Looping

Since many DSP algorithms are based on repetitive computations most DSP processors provide hardware support for efficient looping. Usually there is a loop or repeat instruction, that allows loops to be implemented without spending any extra clock cycles for testing and updating the loop counter, or for jumping back to the start of the loop.

2.6 Different Types of DSP Processors

DSP Processors can be divided into three categories: A general purpose DSP processor usually has a large instruction set can be used in almost any DSP application. A domain specific DSP processor is made for a special category of applications, for example audio processing. Finally an application specific DSP processor is developed for one single application only.

Because the FFT algorithm is so common, many DSP processors have hardware support for bit-reversed addressing.
Chapter 3

The Design Flow

This chapter gives an overview of the design flow for a DSP Processor. Figure 3.1 illustrates the flow and a short description of every step follows.

3.1 Requirement Analysis

Before the actual design starts you have to know what to design. In the requirement analysis step it is specified what the processor should be able to do and demands on performance are carefully analyzed. For an application specific DSP processor the requirements are generally determined by the system in which the processor will be used. In this project no requirement analysis was made, since it is not known exactly what the processor will be used for.

3.2 Instruction Set Design and Architecture Planning

In the instruction set design step, it is decided which instructions should be available in the processor.

The format of the instruction word is also decided and a plan for the top level architecture is made. It is necessary to do this in conjunction with the instruction set design to make sure it is possible to actually implement all instructions in hardware.

These activities are described in chapters 4, 5 and 6.
3.3 Behavioral Model

When a suggestion for an instruction set is ready, a behavioral model of the processor is written in some high level language, for instance C. The behavioral model, or instruction set simulator, is a program that simulates the behavior of the processor on instruction level. It is used in the benchmarking step and also allows software engineers to test software before the actual processor exists.

Chapter 7 says more about the instruction set simulator.

3.4 Benchmarking

Benchmarking is used to verify that the instruction set offers sufficient performance to fulfill the requirements set up during requirement analysis. If it does
not, the instruction set has to be modified. Typically performance is increased by moving tasks from software to hardware. After a few iterations, hopefully a working instruction set can be released. After this is done the software engineers can start their work concurrently with the hardware development.

Chapter 8 describes benchmarking further.

3.5 Architecture Design

This step is basically just top-down design of the whole processor architecture, ending at register-transfer level (RTL; basically registers, buses, multiplexers and primitive arithmetic units).

3.6 RTL Implementation

Application and domain specific DSP processors are typically implemented using a hardware description language (HDL), usually VHDL or Verilog. The HDL implementation can be simulated with simulation tools and other tools can be used to synthesize hardware.

Full custom design (“drawing transistors on silicon by hand”) could also be used for timing-critical parts.

Chapter 10 says a little more about architecture design and RTL implementation.

3.7 Verification

Verification is a very large and very important part of the design process. Although it is the final step before the implementation is released it is important to have a good verification strategy from the beginning and to keep it in mind during every step of the flow.

The verification can be divided into functional verification, where for example the logical correctness of HDL code is verified, and physical verification, which means verifying for example timing constraints. To perform physical verification obviously the HDL code (or at least parts of it) has to be synthesized first.

In this project no synthesis and thus no physical verification has been made yet.
If errors are found during verification one, has to go back to the RTL implementa-
tion or architecture design to make corrections. When the verification result is sa-
atisfactory (it is impossible to test everything) the RTL implementation is released.

The functional verification process is described more thoroughly in chapter 11.
Chapter 4

Instruction Set Analysis

4.1 Choosing The Instruction Set

The instruction set is the interface between hardware and software. To design the instruction set you need to know both what jobs should be run on the processor and what parts of the jobs should be done in hardware and software respectively. The second is however not an easy question. Implementing a certain function in hardware, instead of as a subroutine, of course increases hardware complexity, but saves program memory and probably increases performance for executing that particular function. The right choice naturally varies from one application to another depending on how often the function is used and what the requirements are on performance, memory and so on.

Some points to consider for designing a DSP processor with good performance might be [1]:

1. Make the instruction set simple
2. Avoid having instructions of different length
3. All normal instructions should be executed in one clock cycle
4. Normal instructions should only use operands from registers.
5. Use a multiple bus architecture that allows multiple memory accesses in one clock cycle.
6. Use dedicated multiply and accumulate (MAC) hardware.
7. Provide support for fast hardware looping.

8. Provide hardware support for modulo and bit-reversed addressing.

All these recommendations have been followed in this project.

4.2 This Processor

For this processor there was no particular demands on performance nor any special type of jobs that should be run. Therefore the instruction set has been kept quite simple. However some instructions typical for DSP processors have been included. These are:

MAC and multiply instructions.
Hardware looping instructions.
32-bit shift and add instructions

Most of the addressing modes common in DSP processors are also supported. Furthermore the instruction set allows memory accesses in parallel with computational operations (execute one operation and simultaneously load operands for the next one from memory). This can improve the performance for many algorithms (for example convolution based algorithms like FFT and FIR/IIR-filters) most significantly.

See chapter 16 and appendix A for a complete description of the instruction set.
Chapter 5

Machine Code Design

This chapter discusses how to choose the instruction encoding and explains the choice of instruction word for this processor.

5.1 Orthogonality

One way to measure how “good” the instruction set of a processor is, is the concept of orthogonality. On instruction set level, orthogonality refers to the completeness and consistency of the instruction set and to which degree different addressing modes are uniformly available with different operations [2]. For example a processor that has an add function but not a subtract function, or where the subtract function supports different addressing modes than the add function would be considered nonorthogonal.

On machine code level orthogonality relies on the principle of dividing the instructions into different groups of instructions that works similarly. The machine code can then be multiplexed, except for the multiplex control field of the binary code that chooses which group the instruction belongs to. This significantly simplifies the instruction decoding, since most control signals can be decoded from only a small number of the instruction word bits.

Figure 5.1 shows an example of an orthogonal instruction word.

Another way of increasing orthogonality and simplifying decoding is to divide the instruction word into subfields that as far as possible always have the same function. For example the bits selecting the instruction and those selecting operands should be separated in the instruction word and source/destination register should always be decided by the same bits.
Figure 5.1: An instruction word and its subfields. An instruction set where all instructions used this format would be considered highly orthogonal.

The disadvantage of a highly orthogonal instruction set is that it needs a longer instruction word. Since all instructions don’t use all subfields, better orthogonality means more redundancy in the encoding. Longer instruction word means larger bus and memory widths, which increases the system cost. Obviously a tradeoff has to be made. For example it is quite common to have restrictions on what registers can be used as source/destination for different instructions. It is also common to use control bits, that partly determines the behavior of different instructions.

This processor uses control bits to control modulo and bit-reversed addressing, to enable or disable saturation for arithmetic and shift units and to choose between integer and fractional mode for multiplication (see 14.6.2).

5.2 The Instruction Word of This Processor

For this processor a 32-bit instruction word has been chosen. This is a quite long instruction word considering the limited number of instructions and addressing modes. If this had been a commercial product, probably a shorter instruction word length would have been chosen. All available instructions could certainly have been implemented using a 29 or 30-bit instruction word. With a little more use of implied addressing and some restrictions on operands, most of it could even have been possible with a 24-bit instruction set.

However the 32-bit instruction word gives some advantages that would not have been possible with a 24-bit instruction word, for example very good orthog-
onality and lots of space for new instructions and future improvements. Some instructions, particularly those using 16-bit immediate data, could hardly have been implemented at all using a 24-bit instruction set.

Furthermore since this processor is partly for demonstration purposes the simplicity provided by a highly orthogonal instruction set was even more preferable. The instruction word is further described in section 16.1.
Chapter 6

Top Level Architecture

This chapter describes the design of the top level architecture - Computational units, busses, memories and registers.

6.1 Mapping the Instruction Set to Hardware

The principle of the architecture planning is to map one instruction at the time into hardware until all instructions are executable.

Example: For the add function we need some kind of register file where the operands are stored. Then, since we want the addition to be executed in one clock cycle we need two operand buses from the register file to some arithmetic unit that performs the operation. Finally we need some result bus back to the register file. Next we look at the move to memory and load from memory instructions. Let us assume both address and data is in the same register file as the operands for the add function. For the move to memory instruction we need an address bus and a data bus from the register file to the memory. For the load instruction we should be able to use the same address bus, but we add another data bus from memory to the register file.

The process continues like this until we are sure all instructions can be executed. The next step is to try to find buses that can be multiplexed, in this case for example the data bus from register file to memory could probably be the same as one of the operand buses to the arithmetic unit. The resulting architecture for this processor can be found in figure 14.1.

Note: The processor will be implemented using CMOS-technology and hence output buses from memories and different computational units cannot easily be
connected together - tri-state buffers are not used.

### 6.2 The Register File

One of the first things to decide in the design process is the organization of registers. The main questions are how many registers are needed and if each register should be used only for a specific purpose or if they should be general purpose registers.

More registers means easier programming and possibly fewer memory accesses but more hardware, more instruction word bits for addressing and higher power consumption.

Special purpose registers makes it possible to use fewer bits for addressing. General purpose register on the other hand increases flexibility since a data value in a certain register can be used for any (or almost any) operation.

A quite common compromise is to have special purpose registers for addressing and special accumulator registers for MAC operations. One reason to do it like this is that these registers often do not have the same width as the general purpose registers (at least the accumulator registers certainly do not).

In this processor there are 32 general purpose registers that can all be used for arithmetic, logic and shift operations. Eight of them can also be used as address registers. Two of the address registers supports modulo addressing and one supports bit-reversed addressing. Another eight registers are used for other addressing purposes (like step size for post-incremental addressing or to specify modulo addressing areas).

The processor has two 40-bit accumulator registers for MAC-operations (The MAC unit uses a 32-bit multiplier and 8 guard-bits for the accumulator). Each of these uses three of the general purpose registers - One is used for the lower 16 bits, one for the higher 16 bits and half of the third register is used for the eight guard bits.

Using only half of the third register feels a bit awkward sometimes - there will hardly be any use for the other eight bits of that register except maybe for some sort of flag bits. It would have felt more natural to use all 16 bits as guard bits, but that would have meant using a 48-bit accumulator which is much more than anyone would have use for. Maybe it would have been better to extend the “high” register with an extra eight bits (these would have been inaccessible to the programmer but that really doesn’t matter because the result is always saturated before it is used for anything else than MAC-unit operations anyway).
6.3 Concurrent Design of Instruction Set and Architecture

In practice instruction set design, machine code design and architecture planning are to a large extent done concurrently. When you decide to add an instruction to the instruction set you also have to consider how it could be implemented in hardware and if there is “enough space” for it in the instruction word. Otherwise you will surely run into trouble at the later steps.

Furthermore as the architecture and machine code “evolves” you can often find new instructions that can be implemented with very little extra cost (in terms of hardware, instruction word length or loss of orthogonality).

So the development is in fact more of an iterative process - instruction set and architecture are built up concurrently step by step.

The architecture of this processor is described in chapter 14.
Chapter 7

Instruction Set Simulator

This chapter describes the instruction set simulator (ISS), what it is, why there is one and how it works.

7.1 What?

The Instruction set simulator is just what the name says - a program that simulates the function of all the instructions of the processor.

The ISS simply loads a binary file generated by the assembler, transforms it back to assembly language instructions and runs it instruction by instruction, generating the exact same result as the actual processor would have. It also has features for debugging, saving simulation results to file and more.

7.2 Why?

The ISS is very important in the design flow and is used to some extent in almost every step.

7.2.1 The Assembler

Since the ISS does the inverse transformation of the assembler it can be used to verify the function of the assembler - If the output assembly program of the ISS is the same as the input to the assembler there is a good probability that the function of the assembler is correct.
7.2.2 A Behavioral Model

The ISS is used to verify the behavior of the processor, that is to verify that it really does exactly what it is intended to do, that it can really run all the kinds of applications it is supposed to and, last but not least important, that it can do it with sufficient performance (measured in number of useful instructions per clock cycle or something similar; See also chapter 8). For these reasons a bit-true and cycle-true ISS is needed, in other words it has to both produce exactly the right results on instruction level and keep track of exactly how many clock cycles will be used. (It is not as simple as just one instruction per clock cycle, especially with a more complex pipeline)

7.2.3 Verification

Maybe the most important use of the ISS is for verification of the hardware. Beginning at the Instruction level basically all verification of the hardware is done by comparing the test results from the hardware with those generated by the ISS behavioral model.

7.2.4 Concurrent Engineering

Another very important reason for having a good ISS early in the design process is the possibilities for concurrent development of hardware and software. As soon as the ISS is ready, software engineers can start developing application software although the actual hardware does not exist. This is absolutely necessary to achieve the short time to market that is needed today.

7.3 How?

This section describes the ISS developed for this project.

7.3.1 Features

Apart from disassembling and running the program, either the whole program or one instruction at the time, and showing the contents of registers, the ISS has the following features:
Modifying Registers and Memory

Contents of general purpose registers, program counter and memory can be altered manually.

Breakpoints

Breakpoints can be entered causing execution of the program to halt at a specified line of code.

Load/Save Memory to File

The contents of data and tap memory\(^1\) can be loaded from or saved to file. This is useful for example for importing input data or filter coefficients generated by matlab or for exporting execution results to other programs or comparing simulation results.

Tracking Memory and Register Use

To simplify debugging the simulator keeps a record of which registers and memory positions have been loaded with values, either by the program or manually by the user. If the program uses a register or memory position with an undefined value a warning message is displayed.

Script files

All functions available within the simulator can be executed from a script file, that can be run either from within the simulator or automatically at startup.

Batch Mode

The simulator has a special batch mode for use in for example shell scripts. In batch mode the simulator automatically starts, loads a program, runs a script file and quits. (The script would typically load input data from file, run the program and save the output to another file.)

\(^1\) the term tap comes from digital filtering: a filter is divided into taps, each consisting of a MAC operation where data is multiplied with a coefficient, so a tap memory is typically a data memory holding (filter-) coefficients
7.3.2 Implementation

The ISS was implemented using C++. The code is divided into different files so that everything that is dependent on the processor architecture is separated from things related only to how the simulator works. Figure 7.1 and 7.2 shows flow charts of the most important functions of the simulator, namely loading and running a program and executing an instruction.
Figure 7.1: Loading and executing a program in the ISS.
Figure 7.2: Executing an instruction in the ISS.
Chapter 8

Benchmarking

A benchmark is some absolute measure of the performance of a processor.

Benchmarks are basically used for two tasks: To compare the performance of different processors and to verify that a processor fulfills the necessary requirements.

In the DSP processor design flow, requirement verification is the important part. Benchmarking is first used after the instruction set design, to verify that the instruction set fulfills the performance requirements that were found during requirement analysis.

Benchmarks for comparing different processors are important for marketing purposes, or if you want to buy a commercial DSP processor for a system, instead of designing one of your own. However it is not easy to find a benchmark that is both relevant for the application where the processor will be used and gives a fair comparison between different processors.

8.1 MIPS and MACS

Traditionally it has been common to measure performance in MIPS or Million Instructions Per Second. This is a very simple metric, but it is often misleading, especially for DSP processors. The reason is that the actual amount of useful work performed by an instruction, varies a lot between different processors.

Because the multiply-and-accumulate operation is so common in DSP algorithms the performance of DSP processors are often given in MACS (multiply-accumulates per second). This is however also an unreliable measure, because most applications use many operations other than MACs and many processors
can also perform other operations in parallel with MAC operations.

8.2 Application Benchmarking

Benchmarks using a complete application or suit of applications, are more suitable than MIPS and MACS for comparing different processor families. Furthermore it also makes it possible to measure for example memory use and power consumption.

Application benchmarking values are often given as the number of MHz needed to perform a certain task.

**Example:** Let us say that a processor has a benchmark of 20 MHz for real-time speech encoding and 2 MHz for decoding. If we want to perform both tasks simultaneously on the processor we add the two numbers together (and add a little more, maybe 10%, for control code) to get an estimate of what clock frequency is necessary.

One problem with application benchmarking is that the applications are often written in a higher level language, like C, and therefore the benchmark is a measure of the compiler as well as of the processor. Many low cost DSP processors have quite inefficient compilers and the performance critical parts of the software is typically coded in assembly language.

But even if the applications are coded in assembly language, it is difficult to achieve an optimal or even near-optimal implementation, so the benchmark becomes partly a measure of the skill of the programmer. It is also very time consuming to develop complete applications for multiple processors.

8.3 Algorithm Kernel Benchmarking

A compromise between the oversimplified MIPS and MACS benchmark and the complicated application benchmarking is algorithm kernel benchmarking. The idea is to benchmark the algorithms that are the building blocks of most DSP processing systems. These are quite simple algorithms to implement and you can usually be sure you have the optimal implementation.

To evaluate a processor for a specific application, a weighted sum of the benchmarks from kernel algorithms used in the application is calculated.

As an example of kernel algorithms, table 8.1 [3] lists the algorithms used in the BDTI Benchmarks (BDTI - Berkeley Design Technology, Inc, is a com-
pany that, among other things, publishes impartial technical evaluations of DSP processors).

8.4 Tools for Benchmarking

Generally an instruction set simulator is used for benchmarking. For this of course a cycle-true ISS is needed. If benchmarks for things like power consumption are wanted other methods have to be used, for example emulator hardware.

8.5 Benchmarks for This Processor

Due to lack of time no real benchmarking has been done for this project. However the FIR filter mentioned in 11.2.4 is a typical kernel algorithm. The implementation used executes a FIR-filter with \( T \) taps and \( N \) samples in \( N(T + 7) + 12 \) clock cycles which seems to be quite normal (some DSP processors on the market are better, some are worse). With better I/O instructions this value would improve further.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real Block FIR</td>
<td>Finite impulse response filter that operates on a block of real (not complex) data.</td>
<td>Speech processing (e.g. G.728 speech encoding).</td>
</tr>
<tr>
<td>Complex Block FIR</td>
<td>FIR filter that operates on a block of complex data.</td>
<td>Modern channel equalization.</td>
</tr>
<tr>
<td>Real Single-Sample FIR</td>
<td>FIR filter that operates on a single sample of real data.</td>
<td>Speech processing, general filtering.</td>
</tr>
<tr>
<td>LMS Adaptive FIR</td>
<td>Least-mean-square adaptive filter; operates on a single sample of real data.</td>
<td>Channel equalization, servo control, linear predictive coding.</td>
</tr>
<tr>
<td>IIR</td>
<td>Infinite impulse response filter that operates on a single sample of data.</td>
<td>Audio processing, general filtering.</td>
</tr>
<tr>
<td>Vector Dot Product</td>
<td>Sum of the pointwise multiplication of two vectors.</td>
<td>Convolution, correlation, matrix multiplication, multi-dimensional signal processing.</td>
</tr>
<tr>
<td>Vector Add</td>
<td>Pointwise addition of two vectors, producing a third vector.</td>
<td>Graphics, combining audio signals or images.</td>
</tr>
<tr>
<td>Vector Maximum</td>
<td>Finding the value and location of the maximum value in a vector.</td>
<td>Error control coding, algorithms using block floating-point.</td>
</tr>
<tr>
<td>Viterbi Decoder</td>
<td>Decode a block of bits that has been convolutionally encoded.</td>
<td>Error control coding.</td>
</tr>
<tr>
<td>Control</td>
<td>A sequence of control operations (test, branch, push, pop and bit manipulation).</td>
<td>Virtually all DSP applications include some control code.</td>
</tr>
<tr>
<td>256-Point In-Place FFT</td>
<td>Fast Fourier Transform converts a time-domain signal to the frequency domain.</td>
<td>Radar, sonar, MPEG audio compression, spectral analysis.</td>
</tr>
<tr>
<td>Bit Unpack</td>
<td>Unpacks variable length data from a bit stream.</td>
<td>Audio decompression, protocol handling.</td>
</tr>
</tbody>
</table>

Table 8.1: Kernel algorithms in the BDTI benchmarks.
Chapter 9

Pipeline and Control Path

The control path\(^1\) of a processor has three necessary parts. The first is the program memory or control memory, where all the instructions of the program are stored. The second is the program flow controller that generates the program counter (PC) address, that points out the next instruction to be fetched from program memory. Finally the instruction decoder decodes the control signals (both to control path and data path) from the instruction word.

Usually there is also a PC stack for saving return addresses for subroutine calls, hardware for supporting hardware looping, interrupt handling and many other things (though many of these might be considered to be part of the program flow controller).

This processor has a PC stack for subroutine calls, a loop stack for supporting nested hardware loops, a repeat register for simple repeating of one instruction and a pipeline controller whose purpose is described in 9.1 below. Interrupt handling is not yet implemented.

9.1 The Pipeline

The execution of an instruction in a processor includes several steps. First the instruction is fetched from program memory, then control signals are decoded from the instruction. Next, operands may be fetched from memory or registers, an operation could be performed by some computational unit and finally the result is saved somewhere. The principle of pipelining is to divide this process into several

\(^1\)A processor is divided into the data path where all computations are made, and the control path which generates the control signals to the data path
pipeline steps and execute all steps in parallel. This could mean for example that in the same clock cycle as one instruction is fetched from memory, another instruction is decoded, and yet another is executed by a computational unit. In this way the performance of the processor is increased.

DSP processors usually use three or four pipeline steps, but other solutions also exist. A longer pipeline allows the processor to execute faster, but programming usually becomes a bit more complicated and branching effects (see 9.2) and similar complications have greater impact.

This processor has a variable pipeline depth. Most instructions are executed in three steps (fetch, decode and execute) but due to the long critical path of the multiplication unit, the execution part of the multiply and mac instructions\(^2\) are pipelined into two, steps giving a total of four pipeline steps for these instructions.

This might sound like a complicated solution, but as it turned out it could be handled with little extra hardware. Conflicts can occur when a four step instruction is followed by a three step instruction that uses some of the same resources in the third step as the four step instruction in its last step, but this is handled without greater difficulties: The pipeline control unit monitors what kind of instruction is currently executing, what the next instruction is and what resources these instructions use. It will then halt the pipeline for one clock cycle (by inserting a nop instruction) when this is needed to avoid conflicts. An example of this is shown in figure 9.1. In most cases it is possible to avoid these extra clock cycles by rearranging the program code (so that a four step instruction is never followed directly by a three step instruction that uses the same resources.)

Note: The organization of the register file allows one MAC unit instruction and one other instruction to write to it in the same clock cycle, as long as they don’t use exactly the same register.

### 9.2 Jumps and Branches

Instructions that changes the value of the program counter causes some problems in a pipelined processor: When the instruction reaches the execution step of the pipeline, the following instruction/s are already in the pipeline. This is usually handled in one of two ways. Either the pipeline is flushed, that is the instructions in the pipeline steps preceding the execution step, are “thrown away” and replaced

\(^2\)From here on mac in small letters refers to the multiply-and-accumulate instruction, while MAC in capital letters refers to the MAC computational unit. Other instructions than mac (for example multiplication) are executed on the MAC unit.
Figure 9.1: A mac instruction (four pipeline steps) is followed by two add instructions (three pipeline steps). In a) There are no problems. In b) an operand of the first add is part of the result from the mac instruction, so a nop is inserted by the processor to avoid error.

with nop operations. This means that every jump consumes one extra clock cycle for every pipeline step before the execution step. The other solution is to use delayed jumps. This means that the instructions that are already in the pipeline are also executed. To the programmer it looks as if the jump is delayed by a number of (typically two) instructions. This tends to make the program a bit more difficult to follow and the possibility of having two jump instructions immediately following each other has to be handled somehow.

This processor uses delayed jumps (for both conditional and unconditional jumps, subroutine calls and return from subroutine instructions). Furthermore an instruction that may cause a jump must always be followed by two non-jump instructions.

### 9.3 Hardware Looping

This processor has two instructions for hardware looping: the simple ‘repeat’ instruction that just repeats one instruction a number of times and the more complex ‘loop’ instruction that repeats two or more instructions and also allows nested loops. The reason for having two different instruction, is that the pipeline makes it difficult to handle very short loops (one or two instructions) in the same way as longer loops. Many processors that use only one loop instruction, have special restrictions for short loops (for example they may have to be repeated at least some minimum number of times).

The two instructions uses completely different hardware. The hardware for the ‘repeat’ instruction is basically just a counter, counting down for as long as an instruction is repeated. The ‘loop’ instruction is based on a loop stack, where start and end addresses as well as loop counter values for up to four nested loops are
stored.

See section 15.4.2 for further information on hardware looping.
The complete control path is described in chapter 15.
Section 16.3 discusses restrictions to the use of some instructions due to pipeline complications.
Chapter 10

RTL Implementation

10.1 Micro Architecture

When the top level Architecture is completed, the next step is to describe every block on Register-Transfer level. This means making circuit diagrams consisting of components like registers, multiplexers and arithmetical primitives (for example adders). The principle for doing this is similar to that of designing the top level architecture in that operations are mapped into the hardware of the design unit in question one at the time, while trying to multiplex the hardware as far as possible.

The control signals to all multiplexers are named and a table describing which control signals are used for every instruction is created. (This table is in fact practically a truth table for the function of the instruction decoder.)

10.2 VHDL Implementation

The final step is to translate the whole processor into synthesizeable hardware description language code. The tools used for this was Renoir from Mentor Graphics and the hardware description language VHDL.

Renoir can generate VHDL or Verilog code from block diagrams, truth tables, state machines and flowcharts. It also has an interface to the simulation tool Modelsim (from the same company), that was used for all verification of the VHDL code, and many other features of which a few (like version management) were used.

Mostly the block diagram entry method was used - basically (hierarchical) block diagrams are created and the blocks at the lowest level are described in
VHDL code.

Synthesizeable VHDL code was generated for the whole processor core except the memories, for which simple behavioral models were used.
Chapter 11

Verification

Verification is a major part of the hardware design work. It could be up to 80% of the design time for a complex system. Deciding the verification strategy early allows early development of the verification environment (test benches and so on), which improves concurrent engineering possibilities. The verification flow has a major influence on the whole design flow.

11.1 The Verification Strategy

A good verification strategy might be to focus on achieving a very high test coverage at block level and then focus on interconnections between blocks and corner cases on the higher levels [1].

For clarity here follows some common verification related terminology:

**Compliance Testing**
Verifying that the design or part of the design follows its specification.

**Corner Testing**
Trying to find and test the most complex scenarios that are most likely to cause errors.
**Random Testing**

Since it is usually impossible to find all corner cases, it can be useful to use a setup that generates and tests random test vectors. This often generates strange unanticipated corner cases.

**Path Coverage**

Path coverage is a measure of how many of all possible interconnections between different components are tested. Normally a path coverage of 100% is required.

**Branch Coverage**

This is a measure of how many of all possible combinations of multiplexer inputs are tested. Usually a branch coverage of 100% is needed at least at the lowest block level.

### 11.2 Verification for This Project

Due to the limited time for this project the verification has not been as extensive as it would have been in a “real” project.

Below follows a discussion about what verification has been done and what would have been done if there had been more time.

#### 11.2.1 Block Level Verification

The verification performed on block level is mainly compliance testing. However the test vectors have been chosen to at least reach full branch coverage and full path coverage. No corner or random testing was done at block level.

#### 11.2.2 Instruction Level Verification

Most of the effort on instruction level verification went into corner testing of computational instructions. However still a lot more effort could have been put into finding corner cases if there had been time. There was actually some bugs related to corners, that were missed here but turned up during the following random testing.
Program flow instructions were also tested rather extensively on instruction level. The exception is the 'loop' instruction which has a lot of strange special cases that might cause problems, these where not all tested to the extent they should have been, however most of them were tested quite thoroughly during the block level testing of the program flow controller and PC-, loop- and repeat-stacks.

11.2.3 Random Testing

Testing was performed with random data, but not with random instructions. In other words an assembler program was written that loads data from memory, executes different operations on this data and writes it back to memory. For every execution new random input memory data and new random values for control flags where generated. The program tested every mode of every computational instruction, however program flow instructions where not tested.

In the last session, the random testing was run approximately 220 000 times without finding any errors. That is every computational instruction was run with approximately 220 000 different combinations of input data and control bit settings. Although this is only a small fraction of all possible input data, the result implies that the possibilities of finding additional errors within reasonable time are quite small.

It would have been possible to also generate random instructions (just generate random 32-bit words and throw away all that are not valid instruction words) and this is usually done “in reality”, but it was considered to be a bit too time consuming for this project.

11.2.4 Application Level Verification

Application level verification means running the sort of applications the processor is intended to run “in reality”. This is to prove that the processor really can do what it is intended to do in practice.

The application tested on this processor was a 30th order FIR-filter. This application tests both the repeat and loop instructions, modulo addressing and the parallel computation and memory access possibilities. It is also an example of the type of convolution based algorithms that are very common in digital signal processing. The program can be found in appendix B.

The filter used was a low-pass filter. The input and output can be seen in figure 11.1.
The program turned out to work very well. The difference to the result of a Matlab implementation, using 64-bit floating point representation, was in the same order as the precision possible with 16-bit fractional numbers ($2^{-15}$ or approximately three units in the fifth decimal).
Chapter 12

Conclusions and Future Improvements

This chapter summarizes results and conclusions from the project and presents ideas for changes and future improvements of the processor. Many of these things have already been mentioned in the previous chapters.

12.1 Results and Conclusions

On the whole the processor works well. It is fairly uncomplicated to program (at least with a somewhat more advanced assembler software than what was written for this project) and it has quite good performance for convolution based algorithms. Performance for other algorithms has not been investigated due to lack of time and limited knowledge in the area of DSP applications. The verification was also rather limited, but everything that has been tested works.

12.2 Alternative Solutions

This section summarizes some things that might have been implemented in other ways.

The Accumulator Registers

As mentioned before the way the guard bits uses half of a general purpose register feels a bit strange and maybe it would have been better to have the guards bits in
a separate register.

Choice of Source Accumulator Register

The way it works now, instructions using an accumulator register as source (MAC, 32-bit add and 32-bit shift), must use the same accumulator register both as source and destination. The reason for this lies only in the instruction word (in other words it is not because of the data path architecture) and it would have been quite easy to allow both source and destination accumulator registers to be specified in the instruction. The price for this would have been that the source register for both multiplication operands, would have been restricted to use only half of the 32 general purpose registers (the second operand is already restricted to use only registers 16 to 31).

Among other things this would have made it possible to execute operations on an accumulator register value without losing the old value and to copy the value of one accumulator register to the other.

Shorter Instructions

As mentioned before there is very much “space left” in the instruction word and some bits are almost not used at all. Even with half of the instruction space saved for accelerator instructions, the instruction word could easily have been made at least two bits shorter. However if standardized memories would be used and the instruction word length therefore should be the traditional “multiple of eight”, 24 bits would be the next smaller step and that would hardly have been achievable without further limitations to the instruction set.

12.3 Future Improvements

Here are some examples of things that has not been implemented at all yet.

12.3.1 Interrupts

Although not implemented for this processor yet, interrupt handling is necessary to efficiently communicate with other hardware. Basically all DSP processors handle interrupts, however the way in which it is done is often a bit simpler (and quicker) than for general purpose processors.
Specifically for this processor, the support for hardware accelerators would probably include some sort of interrupt.

12.3.2 I/O Ports

The processor supports no I/O yet (except maybe memory mapped) and some sort of port interface should be added.

12.3.3 Additional Instructions

As previously stated, this processor is intended as a platform for hardware accelerators. This means that “application specific instructions” should be added and therefore the “base” instruction set is quite simple. However some more general instructions could be added. For example many DSP processors have instructions to support division and square root calculations - operations which are quite complicated to do without hardware support. Also simpler instructions like minimum and maximum value calculations could be added.

12.3.4 Hardware Accelerator and Multiprocessor Support

As mentioned before, this processor is intended to eventually be used in a system together with four other similar processors that should be able to share memory. Although the instruction set supports this, the necessary hardware is not yet implemented.

The situation is similar for the hardware accelerator support.
Part II

Design Specification
Chapter 13

Introduction

This second part of the thesis describes the architecture and instruction set of the processor. It is not a complete specification, but should at least be enough for the user of the processor.

13.1 Processor Features

The processor uses 32-bit instructions. The Instruction set is highly orthogonal, but the number of instructions is not so large (about 60). There is a lot of “unused space” for future additions. Particularly the processor is intended as a platform for hardware accelerator units and there is room reserved for this in the “instruction space”.

The processor has a 16-bit native data width and uses fixed-point number representation.

It has a Multiply-and-accumulate unit consisting of a 32-bit multiplier, a 40-bit accumulator (in other words 8 guard bits are used) and a 32-bit barrel shifter for scaling and other purposes.

The processor supports some parallelism, as it is possible (under certain circumstances) to do up to two memory access operations and one computational operation every clock cycle. Among other things, this makes it possible to execute convolution based algorithms, with one multiply-and-accumulate operation per clock cycle.

Other features include support for zero overhead hardware looping and modulo and bit-reversed addressing.
13.2 Outline of This Part of the Thesis

This part of the thesis has the following chapters:

**Chapter 14** Describes the architecture of the data path, the computational units, registers and addressing.

**Chapter 15** Gives an overview of how the control part of the processor architecture works.

**Chapter 16** Describes the instruction word and its subfields and lists the machine code of all instructions. This chapter also contains information on some restrictions that applies to the use of some instructions (mainly program flow instructions)
14.1 Architecture Overview

The computational units of the processor are the following:
Arithmetic unit for addition, subtraction and other common arithmetic operations
Logic unit for bitwise ‘and’, ‘or’ ‘xor’ and ‘not’ operations.
Shift unit for arithmetic shift, logic shift and rotation operations.
MAC unit for multiplication and multiply-accumulate operations. The MAC can also perform 32-bit arithmetic shift and 32-bit addition/subtraction.

The data path architecture can be seen in figure 14.1.

Operands are always taken from the 32x16 bit register file. All of the 32 registers can be used as general purpose registers for common arithmetic, shift and logic operations, but most of them also have other functions. Particularly six of these 32 registers are also used as 40-bit accumulator registers for MAC operations. Some of the registers are also used as address registers or for other address generation purposes.

Data exchange between computational units, register file and memories are facilitated by the following busses:
Two 16-bit data busses DA and DB that provides operands for computational units and data to memories.
Two 16-bit result busses RA and RB for sending result from computations and data from memory to the register file.
One 40-bit accumulator register bus from register file to the MAC unit and one 40-bit bus from the MAC unit back to the register file.
At the most one 40-bit word and two 16-bit words can be written to the register file in one clock cycle.

The architecture also has two address busses $\text{AA}$ and $\text{AB}$ so two memories can be addressed simultaneously.

### 14.2 Arithmetic Unit

The arithmetic unit performs 16-bit addition, subtraction, absolute value and average value computations in one clock cycle. The first operand comes either from
DA or is immediate data from the instruction word. The second operand (if there is one) is always from DB. Addition and subtraction is done with or without saturation depending on the saturation mode control bit.

The Arithmetic unit can be seen in figure 14.2

Figure 14.2: The arithmetic unit.

14.3 Shift Unit

The shift unit performs 16-bit logic and arithmetic shift operations and rotation, with or without intermediate carry, in one clock cycle. All operations are specified as left shift operations. Right shift is accomplished by specifying a negative number of steps. The value to be shifted is always provided on DB. The number of steps is either given by the five least significant bits of DA or by a 5-bit immediate data value in the instruction word.

Figure 14.3 shows the shift unit.

14.4 Logic Unit

The logic unit performs bitwise ‘and’, ‘or’, ‘xor’ and ‘not’ operations between 16-bit words. For ‘and’, ‘or’, and ‘xor’ operations the first operand is either on DA
or immediate data from the instruction word and the second operand is on DB. The single operand for the 'not' operation is always on DA.

Figure 14.4 shows the logic unit.

14.5 MAC Unit

The MAC unit performs multiplication with or without rounding, and multiply-and-accumulate operations in two clock cycles. It also performs 32-bit shift, 32-bit addition/subtraction and round operations in one clock cycle. All operations can be executed with or without saturation.

The multiplication uses integer or fractional number representation depending on the fractional mode control bit (see 14.6.2).

The MAC unit consists of the following parts:
A **32-bit multiplier** multiplying two 16-bit operands from DA and DB into a 32-bit result. Both operands can be taken as signed or unsigned values independently. The result of the multiplication is stored in an internal pipeline register in the MAC unit.

A **40-bit adder** where the registered result of the multiplication, sign extended with eight guard bits to a total of 40 bits, can be added to or subtracted from one of the 40-bit accumulator registers. A 16- or 32-bit value from DA, or DA concatenated with DB, can also be added or subtracted directly to an accumulator register. The adder also facilitates rounding (see below).

A **32-bit barrel shifter** which enables the value from the accumulator to be arithmetically shifted before reaching the adder. The number of steps to shift is either the six least significant bits of DA or 6-bit immediate data from the instruction word (positive value for left shift and negative for right shift).

The MAC unit is shown in figure 14.5

### 14.5.1 Rounding

Rounding is executed by adding 1 to the 17:th bit position (i.e. bit 16) of the 40-bit value, if the 16 least significant bits are larger than \(h7FFFF\). This means that the 24 most significant bits (16 bits plus 8 guard bits) of the result is the rounded value and the 16 least significant bits are unaffected. (The equivalent operation using decimal numbers would be to add one if the decimal part was greater than or equal to 0.5 and then truncate the decimals)

### 14.5.2 Saturation Unit

All MAC unit operations can be performed with or without saturation. If saturation is enabled, the result will be saturated to the smallest or largest possible 32-bit values (\(hF80000000\) and \(h007FFFF\) respectively), whenever the result from the adder is smaller or larger than these values.

### 14.6 Register File

As mentioned before, the processor has a register file consisting of 32 16-bit registers. All registers are listed in table 14.1 All registers can be used as general purpose registers for holding operands and results for computational operations.
Figure 14.5: The MAC unit
<table>
<thead>
<tr>
<th>Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRP0/ARP0</td>
<td></td>
</tr>
<tr>
<td>GRP1/ARP1</td>
<td></td>
</tr>
<tr>
<td>GRP2/ARP2</td>
<td></td>
</tr>
<tr>
<td>GRP3/ARP3</td>
<td></td>
</tr>
<tr>
<td>GRP4/ARP4</td>
<td></td>
</tr>
<tr>
<td>GRP5/ARP5</td>
<td></td>
</tr>
<tr>
<td>GRP6/ARP6</td>
<td></td>
</tr>
<tr>
<td>GRP7/ARP7/LOOP</td>
<td></td>
</tr>
<tr>
<td>GRP8/STEP0</td>
<td></td>
</tr>
<tr>
<td>GRP9/STEP1</td>
<td></td>
</tr>
<tr>
<td>GRP10/STEP2</td>
<td></td>
</tr>
<tr>
<td>GRP11/STEP3</td>
<td></td>
</tr>
<tr>
<td>GRP12/STEP4/TOP0</td>
<td></td>
</tr>
<tr>
<td>GRP13/STEP5/BOTTOM0</td>
<td></td>
</tr>
<tr>
<td>GRP14/STEP6/TOP1</td>
<td></td>
</tr>
<tr>
<td>GRP15/STEP7/BOTTOM1</td>
<td></td>
</tr>
<tr>
<td>GRP16/CONTROL</td>
<td></td>
</tr>
<tr>
<td>GRP17</td>
<td></td>
</tr>
<tr>
<td>GRP18</td>
<td></td>
</tr>
<tr>
<td>GRP19</td>
<td></td>
</tr>
<tr>
<td>GRP20</td>
<td></td>
</tr>
<tr>
<td>GRP21</td>
<td></td>
</tr>
<tr>
<td>GRP22</td>
<td></td>
</tr>
<tr>
<td>GRP23</td>
<td></td>
</tr>
<tr>
<td>GRP24</td>
<td></td>
</tr>
<tr>
<td>GRP25</td>
<td></td>
</tr>
<tr>
<td>GRP26/ACC0-low</td>
<td></td>
</tr>
<tr>
<td>GRP27/ACC0-high</td>
<td></td>
</tr>
<tr>
<td>GRP28/ACC0-guard</td>
<td></td>
</tr>
<tr>
<td>GRP29/ACC1-low</td>
<td></td>
</tr>
<tr>
<td>GRP30/ACC1-high</td>
<td></td>
</tr>
<tr>
<td>GRP31/ACC1-guard</td>
<td></td>
</tr>
</tbody>
</table>

Table 14.1: The register file.
GRP0 - GRP7 (ARP0 - ARP7) can also be used as address registers for addressing data and tap memory.

GRP7 (LOOP) is also used to hold the loop counter value during hardware loops.

GRP8 - GRP15 (STEP0 - STEP7) holds step lengths for updating the addressing registers during post increment addressing.

GRP11 - GRP 15 Also holds top (TOP0/TOP1) and bottom (BOTTOM0/BOTTOM1) registers for modulo addressing.

GRP16 (CONTROL) holds the control bits and GRP26 - GRP31 (ACC0/ACC1) the accumulator registers.

The architecture of the register file can be seen in figure 14.6.

14.6.1 The Accumulator Registers

The Register file has two 40-bit accumulator registers, ACC0 and ACC1, for storing results of MAC unit operations. ACC0 consists of GRP26 (holding the 16 least significant bits of the 40-bit accumulator register) GRP27 (bit 16 to 31 of the accumulator register) and the 8 least significant bits of GRP28 (guard bits of the accumulator register). In the same way ACC1 consists of GRP29 (low bits) GRP30(high bits) and GRP31 (guard bits).

The data on the 40-bit bus to the MAC is always either ACC0 or ACC1 and the data on the 40-bit bus from the MAC can only be written to either ACC0 or ACC1.

14.6.2 The Control Register

The control register holds the following control bits (the five least significant bits of the register):

```
| BR | M0 | M1 | S | F |
```

The BR (Bit Reverse) Control Bit

When the BR control bit is set the address from ARP0 is bit-reversed. See 14.7.3 below for more information on bit-reversed addressing.
Figure 14.6: The register file. inc0, inc1 etc is new values for post increment addressing, that is \( inc_x = ARP_x + STEP_x \). For inc0 and inc1 modulo updating is also applied if modulo addressing is enabled, see figure 14.7. The block marked modulo is used for offset modulo addressing. It is similar to the blocks calculating inc0 and inc1. The block marked BR is for bitreversal.
Figure 14.7: Modulo updating of address registers. A circuit like this generates the signals inc0 and inc1 in figure 14.6. The multiplexers controlling the calculation of the delta value are controlled by the sign of the step value. If STEP is positive then \( \text{delta} = \text{ARP} + \text{STEP} - \text{BOTTOM} - 1 \) and if STEP is negative \( \text{delta} = \text{TOP} - \text{ARP} - \text{STEP} - 1 \). If \( \text{delta} < 0 \) then \( \text{ARP} + \text{STEP} \) is still in the modulo addressing area and \( \text{inc} = \text{ARP} + \text{STEP} \). Otherwise \( \text{inc} = \text{TOP} + \text{delta} \) for \( \text{STEP} > 0 \) and \( \text{BOTTOM} - \text{delta} \) for \( \text{STEP} < 0 \). If modulo addressing is not enabled \( \text{inc} \) is always \( \text{ARP} + \text{STEP} \).

**The MO and M1 (Modulo Addressing) Control Bits**

M0 and M1 enables modulo addressing for the address registers ARP0 and ARP1 respectively. See 14.7.2 below for more information on modulo addressing.

**The S (Saturation Mode) Control Bit**

This control bit enables saturation mode for the arithmetic and shift units. Note that saturation in the MAC unit is *not* affected by this control bit.

**The F (Fractional Mode) Control Bit**

When this control bit is enabled data words represent fractional numbers in the range \([-1, 1]\) instead of integers. Only multiplication operations are affected by this.
14.7 Addressing

The processor should be able to address up to 64 kWord program memory (PM), 64 kWord tap memory (TM), 4x64 kWord data memory (DM0 - DM3, each belonging to a different processor) and 64 kWord third memory (3M). However, at this point no 3M and only one DM have been implemented. When everything has been implemented, all references to TM in this thesis should be replaced by “TM or 3M” and almost all references to DM should be replaced by “DM0 - DM3”.

14.7.1 Addressing Modes

Here follows descriptions of all addressing modes.

Register Direct

The data is taken from a register, GRP0-GRP32, pointed out by the instruction.  
**Example:** add GRP5 GRP6

Register Indirect With Post Increment

The data is taken from a memory address pointed out by an address register, ARP0-ARP8, given in the instruction. The address register is then updated by adding the value in the corresponding STEP register.  
**Example:** loadm ARP1++ GRP5

Immediate Address

The data is taken from a memory address from the instruction.  
**Example:** loaddm #hFF03 GRP4

Immediate Data

The data is taken from the instruction.  
**Example:** addi #34 GRP5

Register Indirect With Offset

The data is taken from an address given by an address register plus an offset from the instruction.
**Example:** loadm ARP0 #4 GRP3

**Note:** Register indirect addressing without post increment can be achieved either by using offset addressing with zero offset or by using Register indirect addressing with post increment with the step length set to zero.

### 14.7.2 Modulo Addressing

Address registers ARP0 and ARP1 supports modulo addressing when the flags M0/M1 are enabled in the control register. The top and bottom registers TOP0/TOP1 and BOTTOM0/BOTTOM1 are implied in the operation.

Modulo addressing mode affects both post increment and offset addressing.

### 14.7.3 Bit-Reversed Addressing

If the flag BR is set the address from address register ARP0 is bit-reversed (regardless of which of the addressing methods in 14.7.1 is used), that is the least significant bit of the register becomes the most significant bit of the address etc.

**Example:** For executing a 32 point FFT, first ARP0 could be loaded with b0000000000000000 and STEP0 with b0000100000000000. Then when memory is accessed using post increment addressing the sequence of addresses used would be:

- b0000000000000000 = 0
- b0000000000010000 = 16
- b0000000000001000 = 8
- b0000000000001100 = 24
- b0000000000000100 = 4
  :
- b0000000000011111 = 31

### 14.8 The Status Register

The status register has the following status flags:

- **N** - Negative Value
- **Z** - Equal zero Value
- **C** - Carry/borrow bit
O - Overflow has occurred

The status flags are used to generate conditions for branch instructions. The following branch conditions are available:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greater than</td>
<td>N=0 and Z=0</td>
</tr>
<tr>
<td>Greater or equal</td>
<td>N=0</td>
</tr>
<tr>
<td>Equal</td>
<td>Z=1</td>
</tr>
<tr>
<td>Less or equal</td>
<td>N=1 or Z=1</td>
</tr>
<tr>
<td>Less than</td>
<td>N=1</td>
</tr>
<tr>
<td>Not equal</td>
<td>Z=0</td>
</tr>
<tr>
<td>Carry</td>
<td>C=1</td>
</tr>
<tr>
<td>Not carry</td>
<td>C=0</td>
</tr>
<tr>
<td>Overflow</td>
<td>O=1</td>
</tr>
<tr>
<td>Not overflow</td>
<td>O=0</td>
</tr>
</tbody>
</table>

See appendix A for information on which instructions affects which flags. Note however especially that the borrow bit is inverted, that is C is set when unsigned addition generates carry and when unsigned subtraction does not generate borrow.
Chapter 15

Control Path

The control path of the processor consists of the following blocks:
The Program Memory (PM) contains the program.
The Instruction decoder (ID) decodes the control signals from the instruction word.
The Program Counter (PC) Generates the address for the program memory.
The Program flow controller (PFC) controls the program counter. It also contains the PC stack, the loop stack and the repeat register.
The Pipeline Controller monitors the pipeline and halts the execution when a conflict appears.
The Branch controller keeps track of jump conditions.

The control path is shown in figure 15.1

15.1 The Pipeline

The processor uses a variable length pipeline with three or four steps. Instructions that incorporates a multiplication operation (‘mult’, ’mac’) are executed in four steps and all other instructions in three steps. The reason for this is that the MAC unit is pipelined in two steps.

The first pipeline step is the instruction fetch. Here the instruction is fetched from program memory and loaded into the instruction register.

In the second step, instruction decoding, the control signals are decoded from the instruction word by the instruction decoder and stored in control registers.

The third step differs slightly between three- and four step instructions. For
three step instructions operands are fetched, the operation is performed and the result is written back to the register file. For four step instructions operands are fetched, multiplication is executed and the product is written to the pipeline register in the MAC unit.

In the fourth step of a four step instruction the value of an accumulator register is fetched (if it is a mac instruction) the 40-bit addition/subtraction is executed and the result is written back to the accumulator register.

The variable pipeline depth causes problems in some cases. These are handled by the pipeline controller which is described in 15.5.

Appendix C has timing diagrams for the pipeline for some different program flow cases.

15.2 Instruction Decoder

The instruction decoder decodes all control signals to the data path and some to the control path. It also manages the variable pipeline depth. For this purpose the control signals are stored in three registers:

The $ctrl$ register stores all control signals that are always only used in the third pipeline step.
The \textit{ctrl2} register stores signals that can be used either in step three or step four. The \textit{pipe4} register is used \textit{during the third step to store control signals that will be used in the fourth step}.

A special control signal in the \textit{ctrl} register controls whether the signals in \textit{ctrl2} is for the third or fourth step.

\section*{15.3 Program Counter}

The program counter produces the address for the program memory. During normal execution, the PC is increased by one every clock cycle, but due to program flow instructions and pipeline complications this can change. The next address can be loaded either from the register file or from an immediate address in the instruction. It can also keep its old value to perform the ‘repeat’ instruction or when the pipeline has to be halted and it can be be loaded with the loop start value from loop stack. Finally it can be loaded with the top value of the PC stack in case of a subroutine return and is set to zero on reset.

The program counter is 16 bits wide, which means 64 kWords of program memory can be addressed.

\section*{15.4 Program Flow Controller}

This block controls the updating of the PC. It also monitors and controls hardware looping (loop stack and repeat register) and subroutine calls (the PC stack).

The program flow controller is the most complicated and tricky part of the whole processor and the design details will not be presented in this thesis. However the information below, regarding the PC stack and hardware looping, should be enough for understanding how to program the processor.

\subsection*{15.4.1 Subroutine Calls - The PC Stack}

When a subroutine call is made (the ‘call’ instruction is executed) the program counter is loaded with the starting address of the subroutine and the old program counter address is pushed to the PC stack. When the program returns from the subroutine (the ‘rts’ instruction is executed) the PC stack is popped and the top value is loaded to the program counter.

The PC stack depth is four, so up to four nested subroutine calls are possible.
15.4.2 Hardware Looping

The processor has two instructions for zero overhead hardware looping: The simple 'repeat' instruction that repeats the following instruction a number of times and the more complex 'loop' instruction that repeats any number of instructions larger than one.

Repeat

The 'repeat' instruction is facilitated by the repeat register. During normal execution this register holds the value one, every instruction is executed once and the program counter is increased by one every clock cycle. However as soon as the value of the repeat register is not one, the PC, the instruction register and the control registers are no longer updated. Instead the repeat register is decreased by one every clock cycle until its value is one again. The 'repeat' instruction simply loads a value (larger than one) into the repeat register thereby causing the next instruction to be repeated the specified number of times.

Loop

The 'loop' instruction works in a completely different way than the 'repeat' instruction.

Before the 'loop' instruction is executed the number of repetitions must be loaded into the loop register (GRP7). The code section to be looped starts with the instruction following the 'loop' instruction and ends at an absolute address specified in the instruction. When the 'loop' instruction is executed, these two program addresses and the value of the loop register are pushed to the loop stack.

When the PC reaches the address equal to the loop end address on the top of the loop stack, the PC is set to the corresponding loop start address and the corresponding loop counter value is decreased by one. The loop counter value is then copied back to the loop register in the register file. In that way the current loop counter value is accessible from the program. If the loop counter value is one when the PC reaches the loop end, the loop stack is popped.

Since the loop stack depth is four, up to four nested loops are possible.

Note: due to pipeline complications some restrictions applies to how other program flow instructions can be used with the 'loop' and 'repeat' instructions. This information can be found in section 16.3
15.5 Pipeline Controller

The pipeline controller monitors the pipeline. By looking at what type of instruction is currently executing (if it is a three or four step instruction and which accumulator register it uses) and which is the next instruction to be executed, it determines if the pipeline has to be halted for one clock cycle, before the next instruction is executed.

Halting the pipeline means in practice that the value of PC and instruction registers are kept and control signals of a ‘nop’ operation is loaded to the control registers.

Halting the pipeline is necessary if a four step instruction is followed by a three step instruction and one of the following is true:
1. The source register of the three step instruction is in the accumulator register used by the four step instruction.
2. The three step instruction is dependent on status flags generated by the four step instruction.
3. The three step instruction uses the MAC unit (but not the multiplier, because then it would not be a three step instruction).

15.6 Branch Controller

The branch controller is a very small block whose only purpose is to keep track of status flags and branch conditions and tell the PFC when to branch.
Chapter 16

Instruction Set

16.1 The Instruction Word

The processor uses an orthogonal, 32-bit instruction set. The instruction word is divided into between four and eight subfields in one of the following ways:

<table>
<thead>
<tr>
<th></th>
<th>Mux</th>
<th>Op</th>
<th>Mem</th>
<th>Addr1</th>
<th>SReg</th>
<th>Addr2</th>
<th>DReg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Addr2</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Addr1</td>
<td>S/DReg</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Addr1</td>
<td>Address/Constant</td>
<td>DReg</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Addr1</td>
<td>Address/Constant</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Addr1</td>
<td>Condition</td>
<td>Prog Addr</td>
</tr>
</tbody>
</table>
### Name  Bit  Description
Mux  31:27  Multiplexer switching between different instruction groups.
Op  26:22  Operation code choosing the actual operation.
Mem  21  1 for memory write operations, 0 for read.
   20  1 if TM or 3M is used.
   19  0 for TM, 1 for 3M.
   18  1 if DM is used.
   17:16  Selects DM memory bank 0-3.
Addr1  15:13  Address register for DM or 3M/TM addressing.
SReg  12:8  Source register.
Addr2  7:5  Address register for DM addressing when two parallel memory operations are made.
DReg  4:0  Destination register.
A  4  Accumulator register, 0 for ACC0, 1 for ACC1
Address/
Constant  20:5  Immediate address or constant value.
Offset  7:0  Immediate offset address value.
Prog Addr  15:0  Immediate program address.

**Note:** As long as 3M and DM1-DM3 are not implemented, bits 16, 17 and 19 will always be zero.

### 16.2 Parallel Memory Instructions

Under certain circumstances the instruction set allows memory operations to be executed in parallel with other operations.

In all cases where a memory load is executed in parallel with a computational operation, \( SReg \) and sometimes also \( DReg \) are used both as source for the computational operation and as destination for the memory operation.

In the assembly code, parallel instructions are always separated by a “,” with space on both(!) sides. The different possibilities are described below.

#### 16.2.1 Move to Memory:

Two moves from register to memory, both using register indirect addressing with post increment, can be executed in parallel. The first move must use tm and the second one must use dm.
Example: move2tm ARPx++ GRPx , move2dm ARPy++ GRPy

16.2.2 MAC Operation and Load

Up to two load instructions, using register indirect addressing with post increment, can be executed in parallel with any MAC unit operation (mpy[u][s], mac[sub][u][s], rnd, sat, clracr, addacr, subacr, add32 or sub32).

If one load is parallel with a MAC operation, the destination register for the load must be the same as the first operand register for the MAC operation, if it has any operand registers (for example ‘rnd’ has no operands so any register can be loaded).

If two loads are parallel with a MAC operation the first load must use tm and its destination must be the same as the first operand of the MAC operation. The second load must use dm and its destination must be the same as the second operand register of the MAC operation (if it has two operands). The destination for the second load must also always be one of GRP16 - GRP31.

It is also possible to execute the two load operations in parallel without the MAC operation. In that case any register can be used.

Example 1: mac GRPx GRPy ACCz , loadtm ARPx++ GRPx , loaddm ARPy++ GRPy (where GRPy must be one of GRP16 - GRP31)

Example 2: loadtm ARPx++ GRPx , loaddm ARPy++ GRPy (GRPy can be any register)

16.2.3 Arithmetic, Logic, Shift or Move Operation and Load

One load instruction, using register indirect addressing with post increment, can be executed in parallel with any arithmetic, logic or shift operation using registered operands (or in other words: all arithmetic, logic and shift operations that do not use immediate data) or with the ‘move’ instruction. The destination for the load must be the same as the first/source operand of the other operation.

Example: add GRPx GRPy , loaddm ARP++ GRP

16.3 Instruction Set Restrictions

Due to pipeline complications there are some restrictions to when some instructions can be used, these are listed below.
16.3.1 Branch and Jump Instructions

This processor uses delayed jump, branch, subroutine call and subroutine return instructions. In other words the two instructions following the jump/branch etc is always executed whether the jump is taken or not. If one of these two instructions was also a jump instruction, or for example a loop instruction, that would cause complications, therefore 'jmp', 'bra', 'call' and 'rts' instructions must always be followed by two instructions that are not program flow instructions.

16.3.2 Hardware Loops

The 'repeat' Instruction

The only restriction that applies to the 'repeat' instruction is that the repeated instruction may not be a program flow instruction.

The 'loop' Instruction

The more complex 'loop' instruction has the following restrictions:
1. The loop must consist of at least two instructions (otherwise use 'repeat').
2. The two last instructions of the loop must not be program flow instructions.
3. Two nested loops may not end at the same address.
4. No more than four nested loops are allowed.

16.3.3 Modulo Addressing

The implementation of modulo addressing does not allow the address to “wrap around” the modulo addressing area more than once. This results in the following restrictions:
1. When using modulo addressing and post increment, the step size should not be larger than the modulo addressing area (that is \( \text{STEP}_x < \text{BOTTOM}_x - \text{TOP}_x \) should hold).
2. When using modulo addressing with offset addressing the offset should not be larger than the modulo addressing area.
16.4 Instruction Encoding

This section describes the machine code of every instruction. The letters representing different subfields in the table have the following meanings:

A = Address register
C = Constant data, address or offset
c = Condition
D = Destination Register
M = Memory use
P = Program address register
r = Round accumulator
S = Source Register
s = Saturate accumulator
Y = Accumulator register
X = Occupied
- = Don’t care

<table>
<thead>
<tr>
<th>Mux</th>
<th>OpCode</th>
<th>Memory</th>
<th>Addrl</th>
<th>SReg</th>
<th>Addr2</th>
<th>DReg</th>
<th>Instruction</th>
</tr>
</thead>
</table>

Data move instructions:

00000 00000 0 0-0– — — — — — nop
00000 00000 0 0-0– — — — — — loadm ARPx++ GRPx , loaddm ARPy++ GRPy
00000 00001 0 MMMMM AAA DDDDD AAA DDDDD move2tm ARPx++ GRPx , move2dm ARPy++ GRPy
00000 00001 0 0-1MM AAA DDDDD CCC CCCCC loadtm (ARPx + #offset) GRPx
00000 00010 1 0-1MM AAA DDDDD CCC CCCCC loadtm (ARPx + #offset) GRPx
00000 00010 1 0-1MM AAA DDDDD CCC CCCCC move2dm (ARPx + #offset) GRPx
00000 00001 1 1M0– AAA DDDDD CCC CCCCC move2tm (ARPx + #offset) GRPx
00000 01011 0 CCCCC CCC CCCCC CCC DDDDD loadmci GRPx #addr
00000 01011 1 CCCCC CCC CCCCC CCC DDDDD movedmi GRPx #addr

ALU instructions, arithmetic unit:

00010 00000 0 0-0– — — — — — abs GRPx GRPy
00010 00000 0 0-1MM AAA SSSSS — DDDDD add GRPx GRPy
00010 00000 0 1M0– AAA SSSSS — DDDDD avg GRPx GRPy
00010 00010 0 MMMMM AAA SSSSS — DDDDD addc GRPx GRPy
00010 00010 0 CCCCC CCC CCCCC CCC DDDDD addc GRPx GRPy
00010 00010 0 CCCCC CCC CCCCC CCC DDDDD comp GRPx GRPy
00010 00010 0 MMMMM AAA SSSSS — DDDDD subc GRPx GRPy
00010 00010 0 MMMMM AAA SSSSS — DDDDD sub GRPx GRPy
00010 00100 0 MMMMM AAA SSSSS — DDDDD loadm ARPx++ GRPx
00010 00100 0 0-0– — SSSSS — DDDDD move GRPx GRPy
00010 00100 0 1M0– AAA SSSSS — DDDDD loadm ARPx++ GRPx
00010 01010 0 0-0– — SSSSS — DDDDD move GRPx GRPy
00010 01010 0 1M0– AAA SSSSS — DDDDD move GRPx GRPy

72
0010 010110 00– SSSS DDDD movc32 GRPx GRPy
0010 011000 CCCC CCC CCC CGGG load #data GRPy

ALU instructions, logic unit:
00100 000000 MMMMM AAA SSSS DDDD and GRPx GRPy
00100 000001 CCC CCC CCC CCC andi GRPx GRPy
00100 000110 MMMMM AAA SSSS DDDD or GRPx GRPy
00100 001110 CCC CCC CCC CCC ori GRPx GRPy
00100 010000 MMMMM AAA SSSS DDDD xor GRPx GRPy
00100 010100 CCC CCC CCC CCC xor GRPx GRPy
00100 010110 MMMMM AAA SSSS DDDD not GRPx GRPy

ALU instructions, shift unit:
00110 000000 MMMMM AAA SSSS DDDD asl GRPx GRPy
00110 000001 —— —— CC CCC DDDD asli GRPx GRPy
00110 000100 MMMMM AAA SSSS DDDD lsl GRPx GRPy
00110 000101 —— —— CC CCC DDDD lsli GRPx GRPy
00110 001000 MMMMM AAA SSSS DDDD rsl GRPx GRPy
00110 001001 —— —— CC CCC DDDD rsli GRPx GRPy
00110 001100 MMMMM AAA SSSS DDDD rslc GRPx GRPy
00110 001101 —— —— CC CCC DDDD rslci GRPx GRPy

MAC instructions:
01000 000000 MMMMM AAA SSSS AAA YSSSS mpy GRPx GRPy ACCx
01010 000000 MMMMM AAA SSSS AAA YSSSS mpy GRPx GRPy ACCx SAT
01001 000000 MMMMM AAA SSSS AAA YSSSS mpy GRPx GRPy ACCx RNDb
0101r 000100 MMMMM AAA SSSS AAA YSSSS mpyu GRPx GRPy ACCx [SAT/RND]
0101r 000110 MMMMM AAA SSSS AAA YSSSS mpyu GRPx GRPy ACCx [SAT]
0100r 001000 MMMMM AAA SSSS AAA YSSSS mpyu GRPx GRPy ACCx [SAT]
0100r 001010 MMMMM AAA SSSS AAA YSSSS mpyu GRPx GRPy ACCx [SAT]
0100r 001100 MMMMM AAA SSSS AAA YSSSS mpyu GRPx GRPy ACCx [SAT]
0100r 001110 MMMMM AAA SSSS AAA YSSSS mpyu GRPx GRPy ACCx [SAT]
0100r 010000 MMMMM AAA SSSS AAA YSSSS mpyu GRPx GRPy ACCx [SAT]
0100r 010010 MMMMM AAA SSSS AAA YSSSS mpyu GRPx GRPy ACCx [SAT]
0100r 010100 MMMMM AAA SSSS AAA YSSSS mpyu GRPx GRPy ACCx [SAT]
0100r 010110 MMMMM AAA SSSS AAA YSSSS mpyu GRPx GRPy ACCx [SAT]
0100r 011000 MMMMM AAA SSSS AAA YSSSS mpyu GRPx GRPy ACCx [SAT]
0100r 011010 MMMMM AAA SSSS AAA YSSSS mpyu GRPx GRPy ACCx [SAT]
0100r 100000 MMMMM AAA SSSS AAA YDDDD addacc GRPx ACCx [SAT/RND]
0100r 100010 MMMMM AAA SSSS AAA YDDDD addacc GRPx ACCx [SAT]
0100r 100100 MMMMM AAA SSSS AAA YDDDD addacc GRPx ACCx [SAT]
0100r 100110 MMMMM AAA SSSS AAA YDDDD addacc GRPx ACCx [SAT]
0100r 101000 MMMMM AAA SSSS AAA YDDDD addacc GRPx ACCx [SAT]
0100r 101010 MMMMM AAA SSSS AAA YDDDD addacc GRPx ACCx [SAT]
0100r 101100 MMMMM AAA SSSS AAA YDDDD addacc GRPx ACCx [SAT]
0100r 101110 MMMMM AAA SSSS AAA YDDDD addacc GRPx ACCx [SAT]
0100r 110000 MMMMM AAA SSSS AAA YDDDD addacc GRPx ACCx [SAT]
0100r 110010 MMMMM AAA SSSS AAA YDDDD addacc GRPx ACCx [SAT]
0100r 110100 MMMMM AAA SSSS AAA YDDDD addacc GRPx ACCx [SAT]
0100r 110110 MMMMM AAA SSSS AAA YDDDD addacc GRPx ACCx [SAT]
0100r 111000 MMMMM AAA SSSS AAA YDDDD addacc GRPx ACCx [SAT]
0100r 111010 MMMMM AAA SSSS AAA YDDDD addacc GRPx ACCx [SAT]
0100r 111100 MMMMM AAA SSSS AAA YDDDD addacc GRPx ACCx [SAT]
0100r 111110 MMMMM AAA SSSS AAA YDDDD addacc GRPx ACCx [SAT]
0100r 100000 MMMMM AAA SSSS AAA YDDDD add32 GRPx ACCx [SAT]
0100r 100010 MMMMM AAA SSSS AAA YDDDD add32 GRPx ACCx [SAT]
0100r 100100 MMMMM AAA SSSS AAA YDDDD add32 GRPx ACCx [SAT]
0100r 100110 MMMMM AAA SSSS AAA YDDDD add32 GRPx ACCx [SAT]
0100r 101000 MMMMM AAA SSSS AAA YDDDD add32 GRPx ACCx [SAT]
0100r 101010 MMMMM AAA SSSS AAA YDDDD add32 GRPx ACCx [SAT]
0100r 101100 MMMMM AAA SSSS AAA YDDDD add32 GRPx ACCx [SAT]
0100r 101110 MMMMM AAA SSSS AAA YDDDD add32 GRPx ACCx [SAT]

Program flow instructions:
01100 000000 cccc —— —— PPPPPP bracond GRPx
01100 000001 cccc —— CCC CCC CCC CCCbracond #addr
01100 000100 cccc —— CCC CCC CCC CCC jmp #addr
01100 000110 cccc —— CCC CCC CCC CCC jmp #addr
01100 001000 cccc —— —— —— P PPPPPP call GRPx
01100 001010 cccc —— CCC CCC CCC CCC call #addr
01100 001110 —— CCC CCC CCC CCC loop #addr
01100 010010 —— CCC CCC CCC CCC repeat #data
01100 010100 —— —— —— rts

Accelerator instructions:
1XXXXXXXX XXXXX XXX XXXXX XXX XXXXX Accelerator instructions.
Bibliography

[1] Dake Liu, Design an embedded digital signal processor, LiTH.


Appendix A

Instruction set summary

This chapter has complete descriptions of all instructions of the processor. This includes:

Type of instruction - Instruction group, short description.
Syntax - What the assembly code looks like, addressing modes.
Operands - What data the instruction can use.
Execution - What the instruction does (“mathematically”)
Description - Description of the behaviour of the instruction, and which registers and statusflags are affected.
Example - A short exemple of use of the instruction.
**abs**

**Type of instruction**
Arithmetic instruction - absolute value

**Syntax**
Register direct: abs GRPx, GRPy

**Operands**
- GRPx: GRP0 - GRP31
- GRPy: GRP0 - GRP31

**Execution**
- \( |GRPx| \rightarrow GRPy \)

**Description**
The absolute value of register GRPx is stored in register GRPy. The flags are not updated.

**Example**
- abs GRPx, GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPx</td>
<td>hFF12</td>
<td>hFF12</td>
</tr>
<tr>
<td>GRPy</td>
<td>h0020</td>
<td>h00EE</td>
</tr>
</tbody>
</table>
add

Type of instruction
Arithmetic instruction - addition.

Syntax
Register direct without carry: \texttt{add GRPx, GRPy}

Operands
\texttt{GRPx: GRP0 - GRP31, GRPy: GRP0 - GRP31}

Execution
\texttt{\texttt{GRPx} + \texttt{GRPy} \rightarrow \texttt{GRPy}}

Description
The values in register \texttt{GRPx} and \texttt{GRPy} are added and the result is stored in register \texttt{GRPy}. The flags N, Z, C and O are updated. C is set when unsigned addition generates carry. O is set when signed addition generates overflow.

Example
\texttt{add GRPx, GRPy}

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0010</td>
<td>b0000</td>
</tr>
<tr>
<td>GRPx</td>
<td>h0012</td>
<td>h0012</td>
</tr>
<tr>
<td>GRPy</td>
<td>h0020</td>
<td>h0032</td>
</tr>
</tbody>
</table>
addc

**Type of instruction**
Arithmetic instruction - addition with carry in.

**Syntax**
Register direct with carry: addc GRPx, GRPy

**Operands**
GRPx: GRP0 - GRP31
GRPy: GRP0 - GRP31

**Execution**
\[ GRPx + GRPy + C \rightarrow GRPy \]

**Description**
The values in register GRPx and GRPy and the value of the flag C are added and the result is stored in register GRPy. The flags N, Z, C and O are updated. C is set when unsigned addition generates carry. O is set when signed addition generates overflow.

**Example**
addc GRPx, GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0010</td>
<td>b0000</td>
</tr>
<tr>
<td>GRPx</td>
<td>h0012</td>
<td>h0012</td>
</tr>
<tr>
<td>GRPy</td>
<td>h0020</td>
<td>h0033</td>
</tr>
</tbody>
</table>
addi

Type of instruction
Arithmetic instruction - addition with immediate data

Syntax
Immediate data without carry: addi #Data GRPy

Operands
h8000 ≤ Data ≤ h7FFF
GRPp: GRP0 - GRP31

Execution
Data + GRPpy → GRPpy

Description
The value in register GRPpy and the Data value are added. The result is stored in register GRPpy. The flags N, Z, C and O are updated. C is set when unsigned addition generates carry. O is set when signed addition generates overflow.

Example
addi #h1234 GRPp

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPp</td>
<td>h0020</td>
<td>h1254</td>
</tr>
</tbody>
</table>
and

Type of instruction
Logic instruction - bitwise and.

Syntax
Register direct: and GRPx, GRPy

Operands
GRPx: GRP0 - GRP31
GRPy: GRP0 - GRP31

Execution
\[ GRPx \land GRPy \rightarrow GRPy \]

Description
Bitwise and between the values in register GRPx and GRPy. The result is stored in register GRPy. The flags N and Z are updated.

Example
and GRPx GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b1010</td>
<td>b0010</td>
</tr>
<tr>
<td>GRPx</td>
<td>h0012</td>
<td>h0012</td>
</tr>
<tr>
<td>GRPy</td>
<td>h8010</td>
<td>h0010</td>
</tr>
</tbody>
</table>
andi

Type of instruction
Logic instruction - bitwise and with immediate data

Syntax
Immediate data: \( \text{andi} \#\text{Data GRPy} \)

Operands
\( h8000 \leq Data \leq h7FFF \)
GRPy: GRP0 - GRP31

Execution
\( Data \& GRPy \rightarrow GRPy \)

Description
Bitwise and between the value in register GRPy and the Data. The result is stored in register GRPy. The flags N and Z are updated.

Example
\( \text{andi} \#\text{hFF GRPy} \)

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>h1000</td>
<td>h0000</td>
</tr>
<tr>
<td>GRPy</td>
<td>h8020</td>
<td>h0020</td>
</tr>
</tbody>
</table>
**asl**

**Type of instruction**
Shift instruction - arithmetic shift

**Syntax**
Register direct: asl GRPx, GRPy

**Operands**
GRPx: GRP0 - GRP31
GRPy: GRP0 - GRP31

**Execution**
GRPy >> (GRPₓ & h001F) → GRPy

**Description**
If the value in GRPx is positive the value in register GRPy is shifted GRPx steps to the left. If the value in GRPx is negative the value in GRPy is arithmetically shifted -GRPₓ steps to the right. The result is stored in register GRPy. The flags N, Z, C and O are updated. O is set if overflow occurs on a left shift. C is the last bit shifted out on a right shift.

**Example**
asl GRPx GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0001</td>
<td>b0010</td>
</tr>
<tr>
<td>GRPx</td>
<td>h0013</td>
<td>h0013</td>
</tr>
<tr>
<td>GRPy</td>
<td>h9F22</td>
<td>hFFFC</td>
</tr>
</tbody>
</table>
Type of instruction
Shift instruction - arithmetic shift with immediate data.

Syntax
Immediate data: asli #Step, GRPy

Operands
-15 ≤ Step ≤ 15
GRPy: GRP0 - GRP31

Execution
GRPy >> Step → GRPy

Description
If the value Step is positive the value in register GRPy is shifted GRPx steps to the left. If the value in GRPx is negative the value in GRPy is arithmeticaly shifted -GRPx steps to the right. The result is stored in register GRPy. The flags N, Z, C and O are updated. O is set if overflow occurs on a left shift. C is the last bit shifted out on a right shift.

Example
asli #-4 GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b1000</td>
<td>b1000</td>
</tr>
<tr>
<td>GRPy</td>
<td>hff22</td>
<td>hfff2</td>
</tr>
</tbody>
</table>
avg

Type of instruction
Arithmetic instruction - average value.

Syntax
Register direct: avg GRPx, GRPy

Operands
GRPx: GRP0 - GRP31
GRPy: GRP0 - GRP31

Execution
\[
\frac{GRPx + GRPy}{2} \rightarrow GRPy
\]

Description
The average value of the value in register GRPx and in register GRPy is stored in register GRPy. The flags N and Z are updated.

Example
avg GRPx GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0000</td>
<td>b0000</td>
</tr>
<tr>
<td>GRPx</td>
<td>h0023</td>
<td>h0023</td>
</tr>
<tr>
<td>GRPy</td>
<td>h0020</td>
<td>h0021</td>
</tr>
</tbody>
</table>
bra{cond}

**Type of instruction**
Program flow instruction - conditional jump.

**Syntax**
- Register direct: \( \text{bra}\{\text{cond}\} \text{ GRPx} \)
- Immediate PC address: \( \text{bra}\{\text{cond}\} \#\text{Addr} \)

**Operands**
- \( 0000 \leq \text{Addr} \leq FFFF \)
- GRPx: GRP0 - GRP31

**Execution**
- \( \text{if } \{\text{cond}\} \text{ is TRUE} \)
  - \( \text{GRP}_x \rightarrow \text{PC} \)
  - \( \text{or} \)
  - \( \text{Addr} \rightarrow \text{PC} \)
- \( \text{else} \)
  - \( \text{PC} + 1 \rightarrow \text{PC} \)

**Description**
A conditional branch jump. Either register or constant based.
The jump is delayed two cycles, that is the two instructions following the branch
instruction are executed either the branch is taken or not. None of the two fol-
lowing instructions may be bra, call, rts, loop or repeat instructions. Bra may not
be used as a repeat instruction or as one of the two last instructions in a hardware
loop. No flags are updated.

<table>
<thead>
<tr>
<th>{cond}</th>
<th>Relation</th>
<th>Flag status</th>
</tr>
</thead>
<tbody>
<tr>
<td>gt</td>
<td>( \text{GRP}_x &gt; \text{GRP}_y ) ( Z=0 ) and ( N=0 )</td>
<td></td>
</tr>
<tr>
<td>ge</td>
<td>( \text{GRP}_x \geq \text{GRP}_y ) ( N=0 )</td>
<td></td>
</tr>
<tr>
<td>lt</td>
<td>( \text{GRP}_x &lt; \text{GRP}_y ) ( N=1 )</td>
<td></td>
</tr>
<tr>
<td>le</td>
<td>( \text{GRP}_x \leq \text{GRP}_y ) ( Z=1 ) or ( N=0 )</td>
<td></td>
</tr>
<tr>
<td>eq</td>
<td>( \text{GRP}_x = \text{GRP}_y ) ( Z=1 )</td>
<td></td>
</tr>
<tr>
<td>ne</td>
<td>( \text{GRP}_x \neq \text{GRP}_y ) ( Z=0 )</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>( carry )</td>
<td>( C=1 )</td>
</tr>
</tbody>
</table>
nc  not carry  C=0
o  overflow  O=1
no not overflow  O=0

Example
bragt #h30

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0000</td>
<td>b0000</td>
</tr>
<tr>
<td>PC</td>
<td>h0100</td>
<td>h0030</td>
</tr>
</tbody>
</table>

braeq GRPx

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0000</td>
<td>b0000</td>
</tr>
<tr>
<td>GRPx</td>
<td>h0010</td>
<td>h0010</td>
</tr>
<tr>
<td>PC</td>
<td>h0012</td>
<td>h0013</td>
</tr>
</tbody>
</table>
call

Type of instruction
Program flow instruction - subroutine jump

Syntax
Immediate PC adress: call #Addr
Register direct: call GRPx

Operands
\( h0000 \leq Addr \leq hFFE \)

Execution
\( PC \rightarrow PC_{stack} \)
\( Addr \rightarrow PC \)

Description
A subroutine jump

The jump is delayed two cycles, that is the two instructions following the call instruction are executed before the jump is taken. None of the two following instructions may be bra, call, rts, loop or repeat instructions. Call may not be used as a repeat instruction or as one of the two last instructions in a hardware loop. No flags are updated.

Example
call #h3000

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>h0012</td>
<td>h3000</td>
</tr>
</tbody>
</table>
clracc

**Type of instruction**
MAC instruction - clear accumulator

**Syntax**
Register direct: clracc ACCx

**Operands**

\[ ACC_x : ACC0, ACC1 \]

**Execution**

\[ 0 \rightarrow ACC_x \]

**Description**
Clear accumulator register.
No flags are updated.

**Example**
clracc ACC0

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC0</td>
<td>hxxxxxxx</td>
<td>h0000000000</td>
</tr>
</tbody>
</table>
comp

Type of instruction
Arithmetic instruction - compare two values.

Syntax
Register direct:          comp GRPx GRPy

Operands
GRPx: GRP0 - GRP31
GRPpy: GRP0 - GRP31

Execution

\[ GRPy - GRPx \rightarrow None \]

Description
The value in register GRPx is subtracted from the value in register GRPpy, but the result is not stored. The flags are updated. C is set when unsigned subtraction does not generate borrow. O is set when signed subtraction generates overflow.

Example
comp GRPx GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0000</td>
<td>b0100</td>
</tr>
<tr>
<td>GRPx</td>
<td>hFF12</td>
<td>hFF12</td>
</tr>
<tr>
<td>GRPpy</td>
<td>hFF12</td>
<td>hFF12</td>
</tr>
</tbody>
</table>
jmp

Type of instruction
Program flow instruction - jump

Syntax
Register direct: jmp GRPx
Immediate PC adress: jmp #Addr

Operands
h0000 ≤ Addr ≤ hFFFF

Execution
Addr → PC

Description
Unconditional jump.
The jump is delayed two cycles, that is the two instructions following the jmp
instruction are executed before the jump is taken. None of the two following in-
structions may be bra, call, rts, loop or repeat instructions. Jmp may not be used
as a repeat instruction or as one of the two last instructions in a hardware loop.
No flags are updated.

Example
jmp #h3000

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>h0012</td>
<td>h3000</td>
</tr>
</tbody>
</table>
**load**

**Type of instruction**
Data move instruction. Load register with immediate data

**Syntax**
Immediate data: load #Const GRPy

**Operands**
\[ h_{8000} \leq Const \leq h_{7FFF} \]
GRP\text{y}: GRP0 - GRP31

**Execution**
\[ Const \rightarrow GRPy \]

**Description**
The constant, Const, is loaded into the register GRPy. No flags are updated.

**Example**
loadi #h2034 GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPy</td>
<td>h1210</td>
<td>h2034</td>
</tr>
</tbody>
</table>
loadtm, load3m, loaddm, loaddm0, loaddm1, loaddm2, loaddm3

**Type of instruction**
Memory instruction - Load register from memory

**Syntax**
Register indirect with postincrement: \( \text{loadXmX ARPx}++ \) GRPy
Register indirect with offset address: \( \text{loadXmX ARPx} \#\text{Offset} \) GRPy

**Operands**
\( h0 \leq \text{Offset} \leq hFF \)
ARPx: ARP0 - ARP8
GRPy: GRP0 - GRP31

**Execution**
\( DM0(\#\text{Addr}) \rightarrow \text{GRPy} \)

**Description**
The value stored at the address ARPx or ARPx + Offset in the specified memory is copied to register GRPy. loaddm is equivalent with loaddm0. The memory bits decides which memory is used.
If addressing with postincrement is used ARPx is increased with the value in the STEPx register.
The flags N and Z are updated.

**Example**
loaddm ARPx++, GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARPx</td>
<td>h0200</td>
<td>h0201</td>
</tr>
<tr>
<td>GRPy</td>
<td>hFF12</td>
<td>h1234</td>
</tr>
<tr>
<td>DM(h0200)</td>
<td>h1234</td>
<td>h1234</td>
</tr>
</tbody>
</table>

loaddm ARPx h2, GRPy
<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARPx Register</td>
<td>h0200</td>
<td>h0200</td>
</tr>
<tr>
<td>GRPy</td>
<td>hFF12</td>
<td>h2222</td>
</tr>
<tr>
<td>DM(h0200)</td>
<td>h0000</td>
<td>h0000</td>
</tr>
<tr>
<td>DM(h0201)</td>
<td>h1111</td>
<td>h1111</td>
</tr>
<tr>
<td>DM(h0202)</td>
<td>h2222</td>
<td>h2222</td>
</tr>
<tr>
<td>DM(h0203)</td>
<td>h3333</td>
<td>h3333</td>
</tr>
<tr>
<td>DM(h0204)</td>
<td>h4444</td>
<td>h4444</td>
</tr>
</tbody>
</table>
loaddmi

Type of instruction
Memory instruction - Load register, immediate address

Syntax
Immediate address: loadmi #Addr GRPx

Operands
$h0 \leq Offset \leq hFF$
ARPx: ARP0 - ARP8
GRP y: GRP0 - GRP15

Execution
$DM0(#Addr) \rightarrow GRPx$

Description
The value stored at the address #Addr in dm0 is copied to the register GRPx. The flags are not updated. Note that there is no equivalent function for any other memories than dm0.

Example
loaddmi #hFF00 GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPy</td>
<td>hFF12</td>
<td>h1234</td>
</tr>
<tr>
<td>DM(hFF00)</td>
<td>h1234</td>
<td>h1234</td>
</tr>
</tbody>
</table>
**loop**

**Type of instruction**
Program flow instruction - hardware loop

Immediate PC adress: loop #Addr

**Operands**
$h0000 \leq Addr \leq hFFFF$
LOOP register

**Execution**
$PC + 1 \rightarrow Loopstartstack$
$Addr \rightarrow Loopendstack$
$LOOPregister \rightarrow Loopcountertop$

**Description**
The instructions between the loop instruction and the PC adress Addr (including that address) is repeated a number of times specified by the value in the LOOP register.
Up to four nested loops are possible, however two loops may never end at the same adress. The loop must have at least two instructions (otherwise repeat is used) and the last two instructions in a loop must not be jmp, bra, call or repeat.

**Example**
loadi #30 LOOP
loop #2000

*Instructions from program addresses PC+1 to 2000 will be looped 30 times*
**l_shl**

**Type of instruction**
MAC instruction - 32-bit shift

**Syntax**
Register direct: \textit{l_shl GRPx ACCy}

**Operands**
GRPx: GRP0 - GRP31
ACCy: ACC0, ACC1

**Execution**
\[ GRPy <<= (GRPx \& \text{h}3F) \rightarrow GRPy \]

**Description**
If the value in GRPx is positive the value in accumulator GRPy is shifted GRPx steps to the left. If the value in GRPx is negative the value in ACCy is arithmetically shifted -GRP x steps to the right. The result is stored in register ACCy. The flags N, Z, C and O are updated. O is set if overflow occurs on a left shift. C is the last bit shifted out on a right shift.

**Example**
\textit{l_shl GRPx ACCy}

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPx</td>
<td>h0008</td>
<td>h0008</td>
</tr>
<tr>
<td>ACCy</td>
<td>hxx0000FF22</td>
<td>hxx00FF2200</td>
</tr>
</tbody>
</table>
**l_shli**

**Type of instruction**
MAC instruction. 32-bit shift with immediate data

**Syntax**
Immediate data: \[ l_{shli} \#\text{Steps} \text{ ACCy} \]

**Operands**
\[-32 \leq \text{Steps} \leq 31 \]
\[ \text{ACCy} : \text{ACC0, ACC1} \]

**Execution**
\[ \text{GRP}y \ll \text{Steps} \rightarrow \text{GRP}y \]

**Description**
If the value Steps is positive the value in accumulator GRPy is shifted Steps steps to the left. If Steps is negative the value in ACCy is arithmetically shifted \(-\text{Steps}\) steps to the right. The result is stored in register ACCy. The flags N, Z, C and O are updated. O is set if overflow occurs on a left shift. C is the last bit shifted out on a right shift.

**Example**
\[ l_{shl} \#12 \text{ ACCy} \]

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCy</td>
<td>hxx0000FF22</td>
<td>hxx0FF22000</td>
</tr>
</tbody>
</table>
**lsl**

**Type of instruction**
Shift instruction - logical shift.

**Syntax**
Register direct: lsl GRPx, GRPy

**Operands**
- GRPx: GRP0 - GRP31
- GRPy: GRP0 - GRP31

**Execution**
- \(GRPy \leftarrow (GRP_x \& h1F) \rightarrow GRPy\)

**Description**
If the value in GRPx is positive the value in register GRPy is shifted GRPx steps to the left. If the value in GRPx is negative the value in GRPy is logically shifted \(-GRPx\) steps to the right. The result is stored in register GRPy. The flags N, Z, C and O are updated. O is set if overflow occurs on a left shift. C is the last bit shifted out on a right shift.

**Example**
lsl GRPx GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b1000</td>
<td>b0010</td>
</tr>
<tr>
<td>GRPx</td>
<td>hFFFE</td>
<td>hFFFE</td>
</tr>
<tr>
<td>GRPy</td>
<td>hFF22</td>
<td>h3FC8</td>
</tr>
</tbody>
</table>
**Isli**

**Type of instruction**
Shift instruction - logical shift with immediate data.

**Syntax**
Register direct: Isli #Step, GRPy

**Operands**
$-15 \leq \text{Step} \leq 15$
GRPy: GRP0 - GRP31

**Execution**
$\text{GRPy} \gg \text{Step} \rightarrow \text{GRPy}$

**Description**
If the value Step is positive the value in register GRPy is shifted Step steps to the left. If the value in GRPx is negative the value in GRPy is logically shifted -Step steps to the right. The result is stored in register GRPy. The flags N, Z, C and O are updated. O is set if overflow occurs on a left shift. C is the last bit shifted out on a right shift.

**Example**
Isli #4 GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0000</td>
<td>b1000</td>
</tr>
<tr>
<td>GRPy</td>
<td>hFF22</td>
<td>hF220</td>
</tr>
</tbody>
</table>
**mac, macu, macus, macs, macsub, macsubu, macsubus, macsubsu**

**Type of instruction**
Mac instruction - multiply and accumulate

**Syntax**
Register direct: mac[sub][u/su/us] GRPx GRPy ACCz [SAT]

**Operands**
GRPx: GRP0 - GRP31
GRPy: GRP0 - GRP31
ACCz: ACC0, ACC1

**Execution**
\[ ACC_z + GRP_x \times GRP_y \rightarrow ACC_z \]
\[ ACC_z - GRP_x \times GRP_y \rightarrow ACC_z \]

**Description**
The value of register GRPx is multiplied by the value of register GRPy and the product is added to (mac, macu, macus, macsu) or subtracted from (macsub, macsubu, macsubus, macsubsu) the accumulator ACCz. mac and macsub executes a signed multiplication and macu and macsubu an unsigned multiplication. macus/macsu and macsubus/macsubsu considers the first or the second operand to be unsigned respectively. If SAT is added the accumulator will be saturated after the accumulation. The status flags N, Z and O are updated.

**Example**
mac GRPx GRPy ACC0

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPx</td>
<td>h0002</td>
<td>h0002</td>
</tr>
<tr>
<td>GRPy</td>
<td>h0003</td>
<td>h0003</td>
</tr>
<tr>
<td>ACC0</td>
<td>h0000001000</td>
<td>h0000001006</td>
</tr>
</tbody>
</table>
**move**

**Type of instruction**
Data move instruction - move between registers

**Syntax**
Register direct: `move GRPx GRPy`

**Operands**
GRPx: GRP0 - GRP31  
GRPy: GRP0 - GRP31

**Execution**
`GRPx → GRPy`

**Description**
The value of register GRPx is copied to register GRPy. No flags are updated.

**Example**
`move GRPx GRPy`

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPx</td>
<td>hFF12</td>
<td>hFF12</td>
</tr>
<tr>
<td>GRPy</td>
<td>h1010</td>
<td>hFF12</td>
</tr>
</tbody>
</table>
move2tm, move23m, move2dm, move2dm#

**Type of instruction**
Memory instruction - write to memory

**Syntax**
Register indirect with postincrement: \( \text{move2}\{\text{tm/3m/dm}[x]\} \) ARPx++ GRPx
Register indirect with offset address: \( \text{move2}\{\text{tm/3m/dm}[x]\} \) ARPx #Offset GRPx

**Operands**
\( 0 \leq \text{Offset} \leq \text{FFFF} \)
ARPx: ARP0 - ARP7
GRPx: GRP0 - GRP31

**Execution**
\[
\begin{align*}
\text{GRP}_x & \rightarrow \text{DM}(\text{ARP}_x) \\
\text{GRP}_x & \rightarrow \text{DM} (\text{ARP}_x + \text{Offset})
\end{align*}
\]

**Description**
The value of register GRPx is copied to the specified memory adress. No flags are updated.

**Example**
move2dm ARPx++ GRPx

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPx</td>
<td>hff12</td>
<td>hff12</td>
</tr>
<tr>
<td>ARPx</td>
<td>h0200</td>
<td>h0201</td>
</tr>
<tr>
<td>DM(h0200)</td>
<td>h1234</td>
<td>hff12</td>
</tr>
</tbody>
</table>

move2dm ARPx #h2 GRPx

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPx</td>
<td>hFF12</td>
<td>hFF12</td>
</tr>
<tr>
<td>ARPx</td>
<td>h0200</td>
<td>h0200</td>
</tr>
<tr>
<td>DM(h0200)</td>
<td>h1234</td>
<td>h1234</td>
</tr>
<tr>
<td>-----------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>DM(h0201)</td>
<td>h1234</td>
<td>h1234</td>
</tr>
<tr>
<td>DM(h0202)</td>
<td>h1234</td>
<td>hFF12</td>
</tr>
<tr>
<td>DM(h0203)</td>
<td>h1234</td>
<td>h1234</td>
</tr>
<tr>
<td>DM(h0204)</td>
<td>h1234</td>
<td>h1234</td>
</tr>
</tbody>
</table>
**movedmi**

**Type of instruction**
Memory instruction - write to memory, immediate address

**Syntax**
Adress direct: movedmi #Addr GRPx

**Operands**
$h0 \leq \text{Offset} \leq hFF$
ARPx: ARP0 - ARP8
GRPz: GRP0 - GRP15

**Execution**
$GRPx \rightarrow DM0(Addr)$

**Description**
The value stored in register GRPx is copied to the address Addr in dm0. The flags are not updated. Note that there is no equivalent function for any other memories than dm0.

**Example**
movedmi #hFF00 GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPy</td>
<td>hFF12</td>
<td>hFF12</td>
</tr>
<tr>
<td>DM(hFF00)</td>
<td>h1234</td>
<td>hFF12</td>
</tr>
</tbody>
</table>
**mpy, mpyu, mpyus, mpysu**

**Type of instruction**
Mac instruction - multiplication

**Syntax**
Register direct:  
mpy[u/su/us] GRPx GRPy ACCz [SAT/RND]

**Operands**
GRPx: GRP0 - GRP31
GRPy: GRP0 - GRP31
ACCz: ACC0, ACC1

**Execution**

\[
GRPx \times GRPy \rightarrow ACCz
\]

**Description**
The value of register GRPx is multiplied by the value of register GRPy and the product is placed in the accumulator ACCz. mpy executes a signed multiplication and mpyu an unsigned multiplication. mpyus/mpysu considers the first or the second operand to be unsigned respectively. If SAT is added the result will be saturated and if RND is added the result will be rounded. The status flags N and Z are updated.

**Example**

mpy GRPx GRPy ACC0

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPx</td>
<td>h0002</td>
<td>h0002</td>
</tr>
<tr>
<td>GRPy</td>
<td>h0003</td>
<td>h0003</td>
</tr>
<tr>
<td>ACC0</td>
<td>h1000</td>
<td>h0006</td>
</tr>
</tbody>
</table>
**neg**

**Type of instruction**
Arithmetic instruction - negate value.

**Syntax**
Register direct: neg GRPx GRPy

**Operands**
GRPx: GRP0 - GRP31
GRPy: GRP0 - GRP31

**Execution**
\[-GRP_x \rightarrow GRPy\]

**Description**
The value in register GRPx is negated and stored in register GRPy. The flags are not updated.

**Example**
neg GRPx GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPx</td>
<td>h0012</td>
<td>h0012</td>
</tr>
<tr>
<td>GRPy</td>
<td>h0020</td>
<td>hFFEE</td>
</tr>
</tbody>
</table>
**nop**

**Type of instruction**
Program flow instruction - no operation

**Syntax**
No operands: `nop`

**Operands**

**Execution**
\[ PC + 1 \rightarrow PC \]

**Description**
This instruction only affects the PC and is used to create execution delays.

**Example**
nop

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>h8020</td>
<td>h8021</td>
</tr>
</tbody>
</table>
not

Type of instruction
Logic instruction - invert register.

Syntax
Register direct: not GRPx GRPy

Operands
GRPx: GRP0 - GRP31
GRPy: GRP0 - GRP31

Execution
\( \text{inv}(GRPx) \rightarrow GRPy \)

Description
The value in register GRPx is inverted bitwise and stored in register GRPy. The flags are not updated.

Example
not GRPx GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPx</td>
<td>h0012</td>
<td>h0012</td>
</tr>
<tr>
<td>GRPy</td>
<td>h0020</td>
<td>hFFED</td>
</tr>
</tbody>
</table>
**Or**

**Type of instruction**
Logic instruction - bitwise or.

**Syntax**
Register direct: or GRPx GRPy

**Operands**
GRPx: GRP0 - GRP31
GRPy: GRP0 - GRP31

**Execution**

\[
GRPx | GRPy \rightarrow GRPy
\]

**Description**
Bitwise or between the values in register GRPx and GRPy. The result is stored in register GRPy. The flags N and Z are updated.

**Example**
or GRPx GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0000</td>
<td>b1000</td>
</tr>
<tr>
<td>GRPx</td>
<td>h0012</td>
<td>h0012</td>
</tr>
<tr>
<td>GRPy</td>
<td>h8020</td>
<td>h8032</td>
</tr>
</tbody>
</table>
**ori**

**Type of instruction**
Logic instruction - bitwise or with immediate data.

**Syntax**
Immediate data: ori #Data GRPy

**Operands**
\[ h_{8000} \leq Data \leq h_{7FFF} \]
GRPy: GRP0 - GRP31

**Execution**
\[ Data \mid GRPy \rightarrow GRPy \]

**Description**
Bitwise or between the value in register GRPy and the value Data. The result is stored in register GRPy. The flags N and Z are updated.

**Example**
ori #h1111 GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b1000</td>
<td>b1000</td>
</tr>
<tr>
<td>GRPy</td>
<td>h8020</td>
<td>h9131</td>
</tr>
</tbody>
</table>
repeat

**Type of instruction**
Program flow instruction - repeat instruction

**Syntax**
Immediate data: repeat #Data

**Operands**
$0 \leq Addr \leq 255$

**Execution**
$Data \rightarrow RepeatReg$

**Description**
The instruction following the repeat instruction is repeated Data number of times before the PC is incremented again. The flags are not updated.

**Example**
repeat #20

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeat Reg</td>
<td>[h0000]</td>
<td>[h0014]</td>
</tr>
</tbody>
</table>
round

Type of instruction
MAC instruction - round.

Syntax
Register direct: round ACCx [SAT]

Operands
ACCx : ACC0, ACC1

Execution
if ACCx low > h8000 then ACCX + h8000 → ACCx

Description
Rounds the accumulator register
If bit 15 of ACCx is a ’1’ h8000 is added to ACCx
If SAT is added the result will be saturated after rounding.
The flags N, Z and O are updated.

Example
rnd ACC0

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC0</td>
<td>h0000109011</td>
<td>h0000119011</td>
</tr>
</tbody>
</table>

**rsl**

**Type of instruction**
Shift instruction - rotational shift.

**Syntax**
Register direct: rsl GRPx GRPy

**Operands**
GRPx: GRP0 - GRP31
GRPpy: GRP0 - GRP31

**Execution**
\[ GRPy \leftarrow GRPx \ll (GRP_x \& h1F) \rightarrow GRPy \]

**Description**
The value in register GRPy is rotated GRPx steps to the left without carry between msb and lsb. The result is stored in register GRPy. The flags N and Z are updated. Negative value in GRPx results in right rotation.

**Example**
rsl GRPx GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0000</td>
<td>b1000</td>
</tr>
<tr>
<td>GRPx</td>
<td>h0002</td>
<td>h0002</td>
</tr>
<tr>
<td>GRPy</td>
<td>h2222</td>
<td>h8888</td>
</tr>
</tbody>
</table>
rsli

Type of instruction
Shift instruction - Rotational shift with immediate data.

Syntax
Immediate data: rsli #Step GRPy

Operands
−15 ≤ Step ≤ 15
GRP: GRP0 - GRP31

Execution
GRP ← Step → GRP

Description
The value in register GRPy is rotated, Step, steps to the left without carry between
msb and lsb. Negative Step gives rotation to the right. The result is stored in reg-
ister GRPy. Theflags N and Z are updated.

Example
rsli #4, GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b1000</td>
<td>b0000</td>
</tr>
<tr>
<td>GRPy</td>
<td>hF222</td>
<td>h222F</td>
</tr>
</tbody>
</table>
rslc

**Type of instruction**
Shift instruction - Rotation with intermediate carry.

**Syntax**
Register direct: `rslc GRPx GRPy`

**Operands**
- **GRPx**: GRP0 - GRP31
- **GRPy**: GRP0 - GRP31

**Execution**


**Description**
The value in register GRPy is rotated GRPx steps to the left with carry storage between msb and lsb. Negative value in GRPx gives rotation to the right. The result is stored in register GRPy. The flags N, Z, and C are updated.

**Example**

`rslc GRPx, GRPy`

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0010</td>
<td>b0010</td>
</tr>
<tr>
<td>GRPx</td>
<td>h0003</td>
<td>h0003</td>
</tr>
<tr>
<td>GRPy</td>
<td>h2222</td>
<td>h1114</td>
</tr>
</tbody>
</table>
rslici

Type of instruction
Shift instruction - Rotation with intermediate carry and immediate data.

Syntax
Immediate data: rslici #Step GRPy

Operands
$-15 \leq Step \leq 15$
GRPy: GRP0 - GRP31

Execution
$GRPy \ll Step \rightarrow GRPy$

Description
The value in register GRPy is rotated, Step, steps to the left with carry storage between msb and lsb. The result is stored in register GRPy. The flags N, Z and C are updated.

Example
rslici #4 GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b1000</td>
<td>h0010</td>
</tr>
<tr>
<td>GRPy</td>
<td>hF222</td>
<td>h2227</td>
</tr>
</tbody>
</table>
**rts**

**Type of instruction**
Program flow instruction - return from subroutine.

**Syntax**
No operands: rts

**Operands**

**Execution**
\[ \text{PC} \leftarrow \text{stack} \rightarrow \text{PC} \]

**Description**
This instruction jumps back from the subroutine and restores the PC value.

The jump is delayed two cycles, that is the two instructions following the rts instruction are executed before the jump is taken. None of the two following instructions may be bra, call, rts, loop or repeat instructions. rts may not be used as a repeat instruction or as one of the two last instructions in a hardware loop.

No flags are updated.

**Example**

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC stack top</td>
<td>h0008</td>
<td>hxxxx</td>
</tr>
<tr>
<td>PC</td>
<td>h0200</td>
<td>h0008</td>
</tr>
</tbody>
</table>
**sat**

**Type of instruction**
MAC instruction - saturate.

**Syntax**
register direct: \sat \ACCx

**Operands**
\ACCx : \ACC0, \ACC1

**Execution**
\sat(\ACCx) \rightarrow \ACCx

**Description**
Saturate accumulator register
If the value of \ACCx cannot be represented with 32 bits, \ACCx will be set too h00007FFFFFFFF or hFFFF80000000 depending on the sign of \ACCx. Otherwise the value will be kept.
Flag O is set if \ACCx was larger than 32-bits.

**Example**
sat \ACC0

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ex1: \ACC0</td>
<td>h03xxxxxxx</td>
<td>h007FFFFFFFF</td>
</tr>
<tr>
<td>Ex2: \ACC0</td>
<td>hF3xxxxxxx</td>
<td>hFF80000000</td>
</tr>
</tbody>
</table>
**sub**

**Type of instruction**
Arithmetic instruction - subtraction.

**Syntax**
Register direct without carry: \( \text{sub GRPx GRPy} \)

**Operands**
- GRPx: GRP0 - GRP31
- GRPy: GRP0 - GRP31

**Execution**
\[ GRPy = GRPx \rightarrow GRPy \]

**Description**
The value in register GRPx is subtracted from the value in register GRPy and the result is stored in register GRPy. The flags N, Z, C and O are updated. C is set when unsigned subtraction does not generate borrow. O is set when signed subtraction generates overflow.

**Example**
\( \text{sub GRPx GRPy} \)

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0000</td>
<td>b0010</td>
</tr>
<tr>
<td>GRPx</td>
<td>h0012</td>
<td>h0012</td>
</tr>
<tr>
<td>GRPy</td>
<td>h0020</td>
<td>h000e</td>
</tr>
</tbody>
</table>
**subc**

**Type of instruction**
Arithmetic instruction - subtraction with carry.

**Syntax**
Register direct with carry: \( \text{subc~GRPx GRPy} \)

**Operands**
GPx: GRP0 - GRP31
GRPy: GRP0 - GRP31

**Execution**
\[ GRPy \rightarrow GPx - 1 + C \]

**Description**
The value in register GPx is subtracted from the value in register GRPy. If C is not set (for example if the previous instruction was as subtraction that generated borrow) one more is subtracted. The result is stored in register GRPy. The flags N, Z, C and O are updated. C is set if borrow does not occur.

**Example**
\( \text{subc~GRPx GRPy} \)

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b0000</td>
<td>b0010</td>
</tr>
<tr>
<td>GPx</td>
<td>h0012</td>
<td>h0012</td>
</tr>
<tr>
<td>GRPy</td>
<td>h0020</td>
<td>h000d</td>
</tr>
</tbody>
</table>
subi

Type of instruction
Arithmetic instruction - subtraction with immediate data

Syntax
Immediate data without carry: subi #Data, GRPy

Operands
\( h0 \leq Data \leq hFFFF \)
GRPy: GRP0 - GRP31

Execution
\( GRPy - Data \rightarrow GRPy \)

Description
The value Data is subtracted from the value in register GRPy and the result is stored in register GRPy. The flags N, Z, C and O are updated. C is set when unsigned subtraction does not generate borrow. O is set when signed subtraction generates overflow.

Example
subi #h5 GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPy</td>
<td>h0020</td>
<td>h001b</td>
</tr>
</tbody>
</table>
**XOR**

**Type of instruction**
Logic instruction - bitwise xor.

**Syntax**
Register direct: xor GRPx GRPy

**Operands**
GRPx: GRP0 - GRP31  
GRPy: GRP0 - GRP31

**Execution**

\[
GRPx \ xor \ GRPy \rightarrow GRPy
\]

**Description**
Bitwise xor between the values in register GRPx and GRPy. The result is stored in register GRPy. The flags N and Z are updated.

**Example**
xor GRPx GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>b1000</td>
<td>b0000</td>
</tr>
<tr>
<td>GRPx</td>
<td>h8000</td>
<td>h8000</td>
</tr>
<tr>
<td>GRPy</td>
<td>h8012</td>
<td>h0012</td>
</tr>
</tbody>
</table>
**xori**

**Type of instruction**
Logic instruction - bitwise xor with immediate data.

**Syntax**
Immediate data  
\( \text{xori } \#\text{Data GRPy} \)

**Operands**
h8000 ≤ Data ≤ h7FFF  
GRPy: GRP0 - GRP31

**Execution**
Data xor GRPy → GRPy

**Description**
Bitwise xor between the value in register GRPy and the value Data. The result is stored in register GRPy. The flags N and Z are updated.

**Example**
xori #h1111 GRPy

<table>
<thead>
<tr>
<th>Register/Memory</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Reg</td>
<td>h0000</td>
<td>h0000</td>
</tr>
<tr>
<td>GRPy</td>
<td>h1234</td>
<td>h0325</td>
</tr>
</tbody>
</table>
Appendix B

Assembly code for FIR-filter

This is the assembly code for the FIR-filter program that was used for verification. The lack of I/O instructions makes is a bit awkward.

* FIR-filter
  * input is dm(0:m)
  * output is dm(2000:2000+m)
  * tap coefficients tm(0:n)
  * ARP0 Tap
  * ARP1 Input
  * ARP2 First sample
  * ARP3 Output

load #9 CONTACT * fractional mode and modulo addressing for ARP0
load #0 ARP0 ** Initialize modulo addressing **
load #0 TOP0
load #31 BOTTOM0 * = number of taps
load #1 STEP0

load #1 STEP1
load #0 ARP2 * input start address
load #1 STEP2
load #2000 ARP3 * output start address
load #1 STEP3

load #1189 LOOP * = number of samples-number of taps = 1220-31
loop #19
move ARP2 ARP1 ** LOOP START **
ciracc ACC0
loaddm ARP0++ GRP20 , loaddm ARP1++ GRP21
repeat #31 * = number of taps
mac GRP20 GRP21 ACC0 , loadtm ARP0++ GRP20 , loadtm ARP1++ GRP21
addi #1 ARP2
rnd ACC0 SAT * rounding and saturation
move2dm ARP3++ GRP27 ** LOOP END ** save output
Appendix C

Pipeline Timing Analysis

In order to find potential pipeline conflicts many special program flow cases where studied in detail and pipeline timing diagrams where made. Here, a few simple cases are shown to illustrate how delayed jumps and hardware loops work.

0: braeq #10
1: instr1
2: instr2
3: instr3
10: instr10

Figure C.1: Delayed branch. The two instructions following a jump or branch are always executed, whether the jump is taken or not.
Figure C.2: The repeat instruction. When the repeat instruction is executed, its argument is copied to the repeat register. As long as the value in the repeat register is greater than one, the PC, the instruction register and the control registers are not updated.

Figure C.3: The loop instruction. Before executing the loop instruction the number of loops has to be loaded to the LOOP register. When the loop instruction is executed, loop start, loop end and number of loops are pushed to the loop stack. When PC is equal to the loop end value, the loop start value is copied to the PC. There is a two cycle delay before the loop counter value is copied back to the LOOP register. In that way LOOP is updated the same cycle as the first instruction in the loop is executed.
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

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

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

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

In English

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

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

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

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

© Eric Tell
### Title

**En domänspecifik DSP-processor**

A Domain Specific DSP Processor

### Author

Eric Tell

### Abstract

This thesis describes the design of a domain specific DSP processor.

The thesis is divided into two parts. The first part gives some theoretical background, describes the different steps of the design process (both for DSP processors in general and for this project) and motivates the design decisions made for this processor.

The second part is a nearly complete design specification.

The intended use of the processor is as a platform for hardware acceleration units. Support for this has however not yet been implemented.

### Keywords

DSP processor design, CPU design