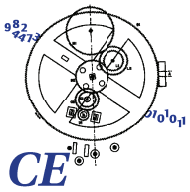


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

CE-MS-2009-11

M.Sc. Thesis

PRAGMA: A Partial-Reconfigurable Audio Platform

Siebe Krijgsman B.Sc.

Abstract

As the area of applications for Field Programmable Gate Arrays, or FPGAs, continues to expand, designers are searching for new methods to enhance the flexibility and efficiency of these devices. A technique called *Dynamic Partial Reconfiguration* is based on a principle of reconfiguring a small region of the FPGA, while the remainder of the device remains operational.

This thesis will investigate the current status of the field of dynamic partial reconfiguration and select the most promising technique for implementation. A proof-of-concept system will be designed and implemented using the selected technique in order to clearly uncover the properties and possibilities of dynamic partial reconfiguration.

The implemented system is an audio processor, capable of manipulating sound through the use of several filters. All filters can be replaced while the system remains functional by performing partial reconfiguration. As such, this system also provides a platform upon which new filters can be designed and tested.

The MOLEN polymorphic processor is a processor architecture that supports the notion of partial reconfiguration. The technique for partial reconfiguration selected in this thesis will be tested for compatibility with the existing implementation of the MOLEN processor.

project website: http://pragma-fpga.googlecode.com

PRAGMA: A Partial-Reconfigurable Audio Platform

Exploring the usability and feasibility of partial reconfiguration

---

Thesis

submitted in partial fulfilment of the
requirements for the degree of

Master of Science

in

Computer Engineering

by

Siebe Krijgsman B.Sc.
born in The Hague, The Netherlands

This work was performed in:

Computer Engineering Group
Department of Microelectronics & Computer Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Delft University of Technology

Delft University of Technology
Department of
Microelectronics & Computer Engineering

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"PRAGMA: A Partial-Reconfigurable Audio Platform"** by **Siebe Krijgsman B.Sc.** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: 02-07-2009

Chairman: _____
dr. K.L.M. Bertels

Advisor: _____
dr.ir. J.S.S.M. Wong

Committee Members: _____
dr.ir. T.G.R.M. van Leuken

_____

Abstract

As the area of applications for Field Programmable Gate Arrays, or FPGAs, continues to expand, designers are searching for new methods to enhance the flexibility and efficiency of these devices. A technique called *Dynamic Partial Reconfiguration* is based on a principle of reconfiguring a small region of the FPGA, while the remainder of the device remains operational.

This thesis will investigate the current status of the field of dynamic partial reconfiguration and select the most promising technique for implementation. A proof-of-concept system will be designed and implemented using the selected technique in order to clearly uncover the properties and possibilities of dynamic partial reconfiguration.

The implemented system is an audio processor, capable of manipulating sound through the use of several filters. All filters can be replaced while the system remains functional by performing partial reconfiguration. As such, this system also provides a platform upon which new filters can be designed and tested.

The MOLEN polymorphic processor is a processor architecture that supports the notion of partial reconfiguration. The technique for partial reconfiguration selected in this thesis will be tested for compatibility with the existing implementation of the MOLEN processor.

project website: http://pragma-fpga.googlecode.com

Acknowledgments

I would like to thank my advisor Stephan Wong for his assistance during this entire project.

My undying gratitude goes out to my girlfriend Minda for her endless mental support, without which I would never have been able to get this far in this education, let alone finish this thesis.

I would also like to use this opportunity to thank Thijs van As, not only for proofreading this thesis and providing invaluable feedback, but also for being my partner in engineering for most of my Bachelors and Masters. Working with you allowed me to rise above myself, always leading to a result that was greater than the sum of our respective parts.

Additional thanks go out to my roommates Steven and Tamar for their technical support and my parents for putting me through all this. Thanks Mom and Dad!

Siebe Krijgsman B.Sc.
Delft, The Netherlands
02-07-2009

Preface

In order to clarify the choice for this project, I would like to start by providing the reader with a little background information on the writer. Since long before starting a Bachelor in electrical engineering I have always had a special interest in music and sound. Being faced with the choice of starting a musical career or an education at the university, I ended up choosing the latter. Throughout my engineering education, I have always had a special interest in the borderline between the physical and the virtual domains. As one might be able to expect, I had a particular interest in combining two of my greatest passions, being engineering and sound.

When dr.ir. Wong proposed a subject incorporating dynamic partial reconfiguration, we decided to integrate a sound-based proof-of-concept, which evolved into a partial reconfigurable sound manipulation engine. Furthermore, I was to investigate the possibilities for incorporating my project into the MOLEN research project [1].

Contents

List of Figures

List of Tables

"Only those who dare to fail greatly can ever achieve greatly."
— *Robert Francis Kennedy*

1 Introduction

Designers are on an everlasting quest to improve the functionality of our computers. One way of realizing this is to extend the flexibility of an implementation without sacrificing on another area of interest such as speed. This thesis will explore a way to do just this: improving the flexibility of a (co-)processor by applying the notion of *Dynamic Partial Reconfiguration*. In order to clearly point out the strengths and weaknesses of this method, both a theoretical background and a proof-of-concept in the form of an audiovisual implementation will be provided.

This chapter is divided into several sections. First, Section 1.1 will discuss the motivation for undertaking this project, followed by the project goals discussed in Section 1.2. Finally, Section 1.3 will discuss the organization of the remainder of this document.

1.1 Project Motivation

Although the notion of partial reconfiguration is not new, there has been a recent influx of interest in this topic. Despite the fact that there is quite a lot of research going on in the field of partial reconfiguration, there are very few actual implementations of large projects to be found. This thesis is intended to contribute to the field by both doing research regarding methods of partial reconfiguration, as well as designing and implementing a large project in order to show the workings of partial reconfiguration.

The goal of this project is to create an audio-manipulation platform using partial reconfiguration. In order to best visualize the effect a system was designed that has both an audio output and a video output. This system is built up out of several partial reconfigurable time domain effects, followed by a frequency transform, several partial reconfigurable frequency domain effects and a transform back to the time domain. Apart from this main path, video data is gathered from several points in this path, feeding information to a video screen.

Aside from these goals, a section of this project will be dedicated to researching the possibility of including this partial reconfigurable design in the MOLEN polymorphic processor. Although the MOLEN architecture was built with a support for partial reconfiguration, there have been no actual implementations using this feature. Therefore, investigating the possibility of implementing such a feature would provide us with some insight into the requirements and restrictions one would encounter when attempting to incorporate partial reconfiguration.

Throughout the project discussions took place with other engineers about partial reconfiguration. In many cases the issue of feasibility came up, being: If

reconfiguring an entire FPGA generally takes only a few hundredths of a second, why would you bother to configure only a part of the system, which takes a few milliseconds, as both are basically unnoticeable? Since this project had seemed challenging and technologically significant, there had been no previous consideration regarding the relevance to the development of FPGAs in general. However, we have had time to research this problem and we would like to present three examples in which case partial reconfiguration is useful, if not indispensable. These situations, however, are by no means the only situations in which partial reconfiguration provides a solution, they are just to illustrate some fields of application.

- For the first example we would like the reader to consider the situation where a partial reconfigurable FPGA serves as a co-processor in a PC. Say that the FPGA would be programmed to handle all video and audio applications with scalable parts, meaning resources could be reassigned while a minimum required core could be kept to sustain the application. In this particular case, one would be watching a streamed video that is being fully handled by the FPGA. Since streaming video is usually handled best by buffering a few seconds, the FPGA would have to keep track of this buffer. If our test case person would now get an VoIP phone call in the middle of his video, the partial reconfigurable core could reassign some resources to the audio section of the FPGA, pausing the video, but sustaining the video buffer. Would we have reconfigured the entire FPGA at this point we would have lost the control over the buffer and thus the stored contents.

- In the second case, consider an FPGA that has been pre-programmed with several (license-protected) cores and shipped to a customer. Although this customer may want to implement several functions in this FPGA using the protected cores, he could also be assigned a user-programmable section of the FPGA that could be programmed freely without damaging of overwriting the protected cores. One might argue that in this situation the manufacturer would use a hard-wired core, however, this would strongly reduce the ease with which the manufacturer could provide updates or redesigns of the protected cores.

- The third example is quite straightforward: Say we have a unit with a static core that uses 90% of available resources on the FPGA. The remaining 10% is occupied by 3 different modules that all occupy 10% and are thus switched in and out one after another. Now say that this particular design is used to compute some numbers, for which the computation would take a few hours. If we would use a full FPGA reprogram every time we needed to switch designunits we would lose all intermediate values, thus making it impossible to ever finish the task.

1.2 Project Goals

The main goal of this project is formulated as follows:

- To design and implement an audio processor making use of the partial reconfiguration technique.

Additionally, several secondary goals are defined:

1. To research partial reconfiguration and select the most feasible implementation to incorporate in this project.
2. To design and implement an audio processor making use of the selected partial reconfiguration technique.
3. The to be designed system is to feature a clear user interface as well as be as autonomous as possible.
4. To investigate the possibility of implementing this project on the available implementation of the MOLEN polymorphic processor.

Chapter 3 will further clarify these goals.

1.3 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 will deal with all the background information required to comprehend the work described in the following chapters, as well as the preliminary research done in the course of this project in the field of partial reconfiguration. Next, all design choices will be motivated in Chapter 3, after which the actual implementation will be discussed in Chapter 4. Although most final test results are hard to present in this thesis, as they are audio signals, a number of significant test results will be shown in Chapter 5. Finally, this thesis will be concluded in Chapter 6.

2 Background

In order to understand the processes used in the system, a basic understanding of several functions is required. This chapter will discuss theory required in order to understand the entire range of techniques used in the project.

We will start with an explanation regarding Reconfigurable Computing, Field-Programmable Gate Arrays (FPGAs) and Dynamic Partial Reconfiguration, which also contains a paragraph dedicated to related work, in Section 2.1. The remainder of this chapter, just like the system itself, is split up into the time domain and the frequency domain. First all necessary information regarding the time domain will be handled in Section 2.2, along with the theory with respect to the modules and effects that were implemented or were attempted to implement, directly followed by a similarly structured section regarding the frequency domain in Section 2.3. The conclusion will summarize what has been discussed in this chapter in Section 2.4.

2.1 Reconfigurable Computing

Before starting on an in-depth explanation of the functionality of FPGAs, the concept and history of reconfigurable computing will be discussed, in order to show in what cases a reconfigurable implementation can aid, extend or replace a static design.

One of the roots of the need for a reconfigurable hardware solution can be found in the trade off between speed and flexibility. A General Purpose Processor can implement any function by breaking it down into the various instructions it supports. Although this is the most flexible solution, it requires many instruction fetches and instruction decodes for most functions, giving it a large overhead. Furthermore, breaking down a complex function into basic operations is in many cases not an efficient approach, reducing the speed of the solution even further. On the other end of the spectrum are Application-Specific Integrated Circuits, or ASICs. As the name reveals these are full hardware circuits designed to execute a single task. While this is generally the fastest solution, it has no flexibility at all. Although several hybrids can be thought up at this time, of which for example Digital Signal Processors are a notable family, these will be left out in order not to stray from our core explanation.

As one may already suspect, reconfigurable computing populates the area in between the two extremes. While a reconfigurable solution is generally static in operation, it can be reconfigured to change the functionality, thus retaining some flexibility with respect to an ASIC, while sacrificing some speed. Although, in

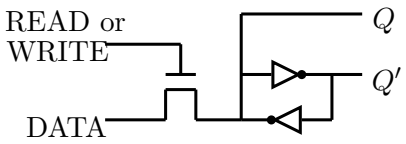


Figure 2.1: An SRAM programming bit

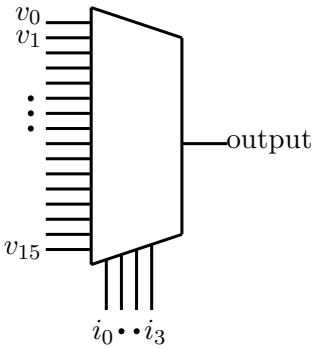


Figure 2.2: A 4-input Lookup Table

contrast to the GPP, where only the instructions are changed, we need to physically change an FPGA to change its behavior, an FPGA can be designed to implement complex functions in a direct and efficient manner. The same implementation on a GPP would have to be constructed out of the supported basic operations and would be (far) less efficient and therefore far slower.

2.1.1 Field-Programmable Gate Arrays

A Field Programmable Gate Array, or FPGA, is a structured grid of standard logic cells that can be programmed to fulfill any desirable function. In modern FPGAs, the programming is performed by SRAM bits, depicted in Figure 2.1. These programming bits consist of 5 transistors –1 enabling switch and two inverters– and are the atomic particle of the FPGA. As FPGAs are reprogrammable and can implement any (very large) function, they are most often used for prototyping purposes. Although they generally lack the speed of an ASIC, FPGAs can be used to test a designed function in minutes rather than days or even weeks.

From a macro design perspective, an FPGA consists of lookup tables, connection blocks (C-blocks) and switching blocks (S-blocks). The lookup tables, or

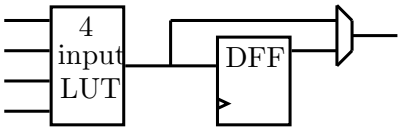


Figure 2.3: A Basic Logic Element (BLE)

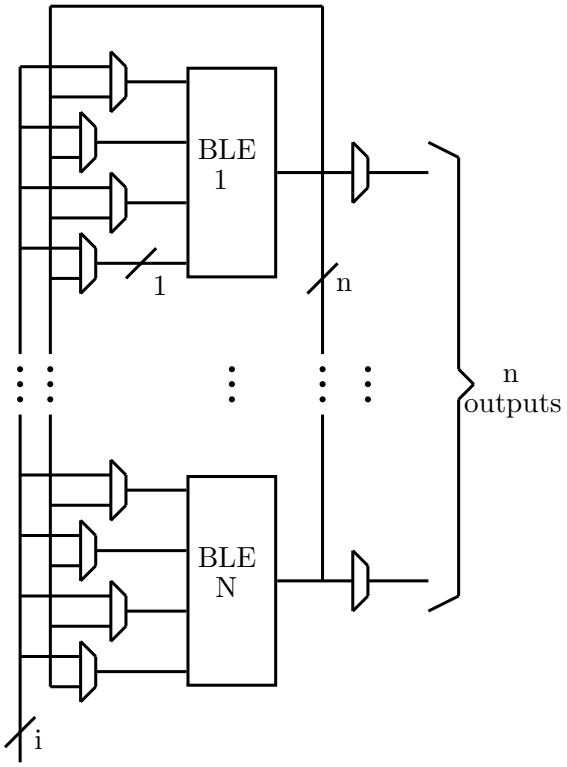


Figure 2.4: A Configurable Logic Block (CLB)

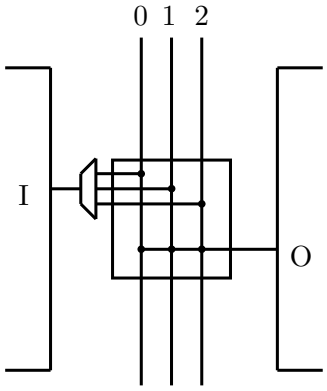


Figure 2.5: A connection (C-)block

LUTs, can be programmed to behave like any binary function and are generally based on 4 inputs and 1 output. A 4 input LUT is shown in Figure 2.2, where the four inputs are used to select one of the 16 saved values. When combined with a memory element, such as a flip-flop, we end up at a Basic Logic Element (BLE), shown in Figure 2.3. In order to facilitate easy routing, we would like to group BLEs together and build an infrastructure of routes around the formed clusters. The BLE clusters, which Xilinx refers to as Configurable Logic Blocks, or CLBs,

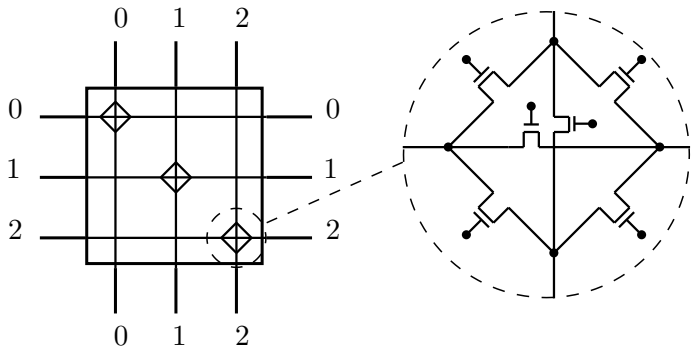


Figure 2.6: A switching (S-)block

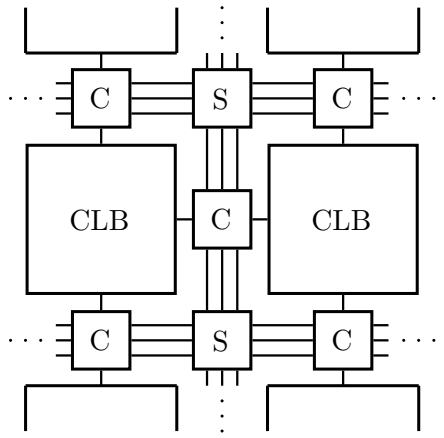


Figure 2.7: The final FPGA layout

with i inputs and n outputs are shown in Figure 2.4. Research shows that small cluster sizes are the most routing efficient [2], and as such most commercial FPGAs use cluster sizes between 4 and 8 BLEs per cluster. When we arrange a number of CLBs in a structure connected by connection and switching blocks, shown in Figures 2.5 and 2.6, respectively, we arrive at the standard FPGA layout depicted in Figure 2.7. This layout is often referred to as an "island-style" layout.

Configuration of the FPGA Configuration of an FPGA is quite straightforward. In order to explain the process, we should first inspect the breakdown of an FPGA, meaning to show how an FPGA is built up. For clarification, the Virtex-II Pro will be used as an example.

The breakdown of the FPGA is presented in Table 2.1. Further explanation regarding the table's contents are at the end of this section. The FPGA is configured using a *bitstream*, which can address any frame in the device uniquely. Table 2.2 shows the 32 bit frame address composition. the 2 BA bits specify the Block Address. Block Address 01 specifies all BRAM columns and BA 10 specifies all BRAM interconnect columns, while everything else (IOB, IOI etc.) is addressed

Table 2.1: FPGA breakdown for the Virtex-II Pro (XC2VP30). Numbers represent Frames Per Column (FPC) and Columns Per Device (CPD)

IOB		IOI		CLB		BRAM		BRAM interconnect		GCLK	
FPC	CPD	FPC	CPD	FPC	CPD	FPC	CPD	FPC	CPD	FPC	CPD
2	4	2	22	46	22	8	64	8	22	1	4

with BA 00. The Major Address (MJA) then specifies a column in the selected block, and the Minor Address (MNA) specifies the frame in the selected column. The byte number is only used by the configuration logic and should always be zeros.

Table 2.2: The Virtex-II Pro Frame Address Composition

					BA		MJA								
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	
MNA								Byte Number							
16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The fact that we can address separate frames within the FPGA is very useful in the perspective of partial reconfiguration. This means that we can address a small number of frames or columns, and program these, while leaving the rest of the configuration untouched. It even means that we can reprogram a part of our device, while the rest of the device continues to run, which is referred to a dynamic partial reconfiguration, which will be discussed in the next section.

For clarification of Table 2.1, a direct citation of the Virtex-II Pro User Guide is provided here [3].

IOB Columns IOB columns configure the voltage standard for the I/Os on the left and right edges of the device. IOBs on the top and bottom edges of the device are configured by the CLB Columns with which they are vertically aligned. There are two IOB columns per device.

IOI Columns IOI columns configure the IOB registers, multiplexers, and 3-state buffers in the IOBs on the left and right edges of the device. IOBs on the top and bottom edges of the device are configured by the CLB columns with which they are vertically aligned. There are two IOI columns per device.

CLB Columns The CLB columns program the configurable logic blocks, routing, and most interconnect. IOBs on the top and bottom edges of the device are also programmed by CLB configuration columns. The number of CLB configuration columns matches the number of physical CLB columns in the device.

BlockRAM Columns BlockRAM configuration columns program only the user memory space BlockRAM. BlockRAMs are accessed by the configuration control logic in the same way that user designs access BlockRAMs: via the address and data pins that are available on the BlockRAM primitive. Consequently, the user design is not able to access BlockRAM while BlockRAM columns are being addressed by the configuration logic. For this reason, active reconfiguration and readback on BlockRAMs should not be attempted. The number of BlockRAM configuration columns matches the number of physical BlockRAM columns in the device.

BlockRAM Interconnect Columns BlockRAM Interconnect columns program all other BlockRAM and multiplier features, including aspect ratios. The number of BlockRAM Interconnect configuration columns matches the number of physical BlockRAM columns in the device.

GCLK Column The global clock column configures most global clock resources, including clock buffers and DCMs. There is one global clock column per device.

2.1.2 Dynamic Partial Reconfiguration

What if we could now extend our flexibility without sacrificing speed? Dynamic Partial Reconfiguration, in this thesis referred to as simply Partial Reconfiguration, involves changing a section of the programmed hardware, while the remainder of the design continues to function. Static Partial Reconfiguration, which reconfigures an FPGA partially when it is off-line, will not be discussed here. The direct effect of Dynamic Partial Reconfiguration is that we are no longer strictly bound to the size of the FPGA we are using. Consider this small example: Say that 90% of an FPGA is filled with necessary hardware and we want to implement two more modules, both requiring 10% of the FPGA. Traditionally, this would be impossible. However, if we do not need both additional modules to function at the same time, we can swap them back and forth. Now the only restriction is that we have to do this while the other 90% of the chip remains active.

Currently, partial reconfiguration is only supported by chips manufactured by Xilinx and Atmel. Since there are Xilinx FPGAs and toolsets readily available, Xilinx has been selected for this project and Atmel solutions will not be discussed in-depth here. Those interested in Atmel partial reconfiguration solutions could consider reading [4] and [5].

Xilinx distinguishes two styles of partial reconfiguration [6]. With *difference-based partial reconfiguration*, partial reconfiguration is achieved by comparing two bitstreams of a full design, being the one currently programmed on the FPGA and the one that contains the full design modified with new logic, removing the overlapping parts, and successively programming the difference onto the FPGA. The other strategy is *module-based partial reconfiguration*, which is used when

several modules are required to communicate with one another. In this case, *bus macros* need to be inserted between the modules.

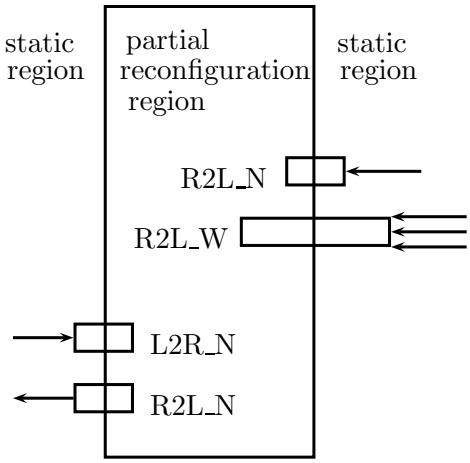


Figure 2.8: The orientation of bus macros in a partial reconfigurable region

These bus macros are placed on the boundary between the static and the dynamic regions of the FPGA, and ensure correct routing when reconfiguration is performed. Bus macros are strictly directional, and it is mandatory for a bus macro to stride across the static-dynamic boundary, placing all the input ports inside the dynamic region and all the output ports inside the static region or vice versa. This setup is shown in Figure 2.8, where L2R indicates a dataflow from left to right and R2L the other way around. The addition N and W stand for *narrow* and *wide* versions of the bus macro. While narrow bus macros, depicted in Figure 2.9, can transport up to 8 bits across the boundary, wide bus macros, presented in Figure 2.10 are intended for grouping signals wider than 8 bits together. Using wide bus macros can be advantageous when the PRR has a limited space at the boundary between the static and partial regions. We can see here that the bus macro itself does not change the signal, it is only used to carry the signals over the PRR boundary. There should not be any connections between the static and dynamic regions that are not routed through a bus macro, *with exception of global clocks*. Another important note here is that the Virtex-II family of Xilinx devices can only reconfigure an entire column of cells, while the Virtex 4 and later devices can dynamically reconfigure in blocks, making reconfiguration even more feasible. This difference is shown in Figure 2.11: While a small partial reconfigurable block requires an entire column to be empty on a Virtex-II, it has a much smaller spacial impact on a Virtex 4.

Related work Since a significant portion of this project and thesis are based on the notion of partial reconfiguration, a summary of related work in this field is presented here.

The earliest relevant reference appeared in 2000 in an application note [7], in

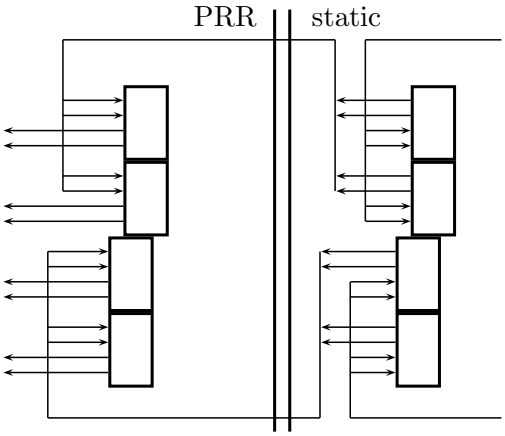


Figure 2.9: The layout of a narrow bus macro

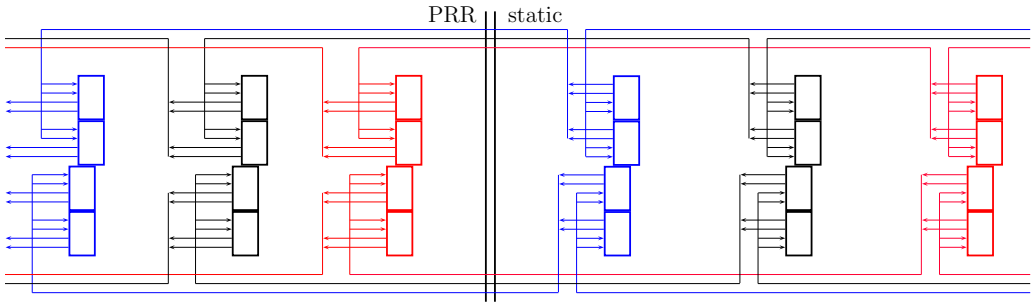


Figure 2.10: The layout of a wide bus macro

which Xilinx introduces the notion of configuring a single column of data. Here we see that they divide the configuration memory into frames, with a width of 1 bit and spanning from the top to the bottom of the device. A frame is the atomic unit of configuration, meaning it is the minimum amount that can be addressed separately. The FPGA is then divided into CLB columns, where each column is

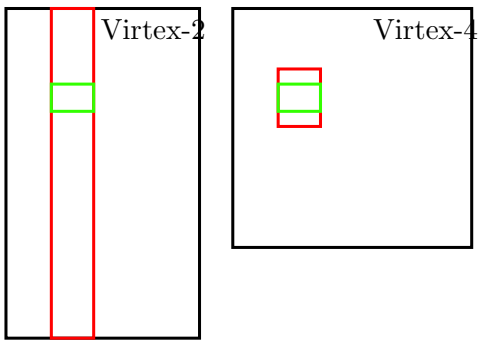


Figure 2.11: Impact of partial reconfiguration. The used module is shown as the small box, the unusable area as the larger

48 frames wide - this only holds for the Virtex-I, the Virtex-II uses 22 frames per column. Finally, and most importantly, this application note states that partial reconfiguration of the FPGA is possible with *and without* shutting down the device.

Secondly, an article by Mesquita et al. [8] gives an overview of the origin of dynamic reconfiguration as well as an insight into the development of the toolchain required for this process. Although the tools and techniques discussed in Mesquita et al. are outdated now, they provide us with an insight of how partial reconfiguration is performed. The main problem of partial reconfiguration identified by Mesquita et al. is that you cannot define the wire positions, resulting in a discrepancy in input and output connections. They chose to define special input/output (IO) buffers, similar to bus macros, and to synthesize the module separately, replacing the unknown modules with dummies with only the mentioned IO buffers in place. Although at this time they did not achieve a fully automated partial reconfiguration design flow, they clearly identified the restrictions of the tools regarding this subject, thereby contributing to the quality of the tools available today.

An article in Xilinx' XCell Journal, titled "PlanAhead Software as a Platform for Partial Reconfiguration", discusses the use of Xilinx' new program PlanAhead for use with partial reconfiguration [9]. This article briefly describes the workings of the Virtex series FPGAs, after which it clearly explains how to incorporate partial reconfiguration in a project through the use of PlanAhead. Alongside this article are several other publications by Xilinx [10, 11, 12] , all of which aided in achieving an understanding of and and implementation of this partial reconfigurable project using Xilinx tools.

In a chapter of the book "Introduction to Reconfigurable Computing" Bobda and Murr describe the entire design flow process from the start to the implementation of a partial reconfigurable project [13]. Although this chapter is actually targeted towards users looking to implement a project using Handel-C [14], which is an extension of the C language specifically designed for FPGA development, this chapter provides a very water-tight approach to designing a partial reconfigurable system. Every single design constraint is mentioned and explained, making this a very good reference for people looking to understand how to use partial reconfiguration and its requirements.

Andres Upegui and Eduardo Sanchez discuss the feasibility of implementing evolving hardware using partial reconfiguration [15]. Their study in mapping chromosomes on an FPGA to study the evolution has led them to start searching for a flexible solution. In their article they discuss three approaches to achieve evolving hardware by using the capabilities of the Virtex family FPGAs. The first approach is coarse-grained, using a modular design structure. Secondly, they apply a more fine-grained implementation, using the difference-based partial reconfiguration, allowing them to modify the contents of a LUT directly. The third method is the direct manipulation of the configuration bitstream, another fine grain angle. Finally, they propose a mix of the three techniques, allowing for different evolution paradigms to be supported.

McDonald's paper on dynamic partial reconfiguration [16] provides a good in-

sight on how to devise a project with partial reconfiguration in general, and specifically through use of the SelectMap/ICAP interface for on-board reconfiguration. This paper suggests an implementation where the MicroBlaze processor reads the configuration data from some off-chip memory, and configures the FPGA through the ICAP interface. As this project strives for autonomy, on-board reconfiguration would be a useful addition to implement.

In a chapter of the book "Field-Programmable Logic and Applications" [17], Xilinx engineers B. Blodget et al. suggest a new approach to enable a device to (partially) reconfigure itself, by using the internal ICAP interface and a PowerPC or microBlaze processor available to the Virtex device. They have devised a toolkit for this very purpose, which they called the Xilinx Partial Reconfiguration Toolkit (XPART). However, for some reason this toolkit was never (yet) made available to the public, but this project may have led to or inspired the partial reconfiguration features in PlanAhead. Although the toolkit was not actually released, this chapter provides us with the necessary knowledge and insights, should we decide to implement such a feature.

Although there has been a lot of research and discussion in the field of partial reconfiguration, there are very few large projects based on this principle. This project aspires to fill a piece of this void.

2.2 The Audio Time Domain

In this section all the behavior of the modules that are considered to be part of the time domain are explained.

2.2.1 Analog to Digital Conversion

Since we require our system to be digital in order to apply a partial reconfiguration strategy, we need to convert our continuous time based signal into a sampled time based signal. In order to do this we first need to apply a *sample-and-hold* strategy on the input so we get a signal that is discrete in time but continuous in magnitude as in Figure 2.13. To do this we need to select a suitable sample frequency. According to the Nyquist-Shannon Condition [18, 19] we need to choose a sample frequency that is at least twice as large as the highest frequency we are using as input, in order to avoid *aliasing*. Aliasing is caused by the spectrum of the input signal that is 'folded' back around the sample frequency. When aliasing occurs, we can no longer unambiguously determine our input signal and the result will be a distorted output signal. In Figure 2.12 we can clearly see the effect of aliasing: we see the desired spectrum and the mirror spectrum that has been folded around the sample frequency. The overlapping sections in Figure 2.12a will cause distortion of the output signal. In Figure 2.12b we can see that using a sample frequency of twice the maximum input frequency will result in a clean, unaliased output signal.

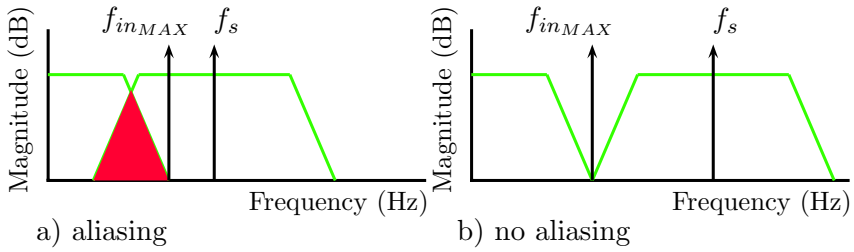


Figure 2.12: The effect of aliasing

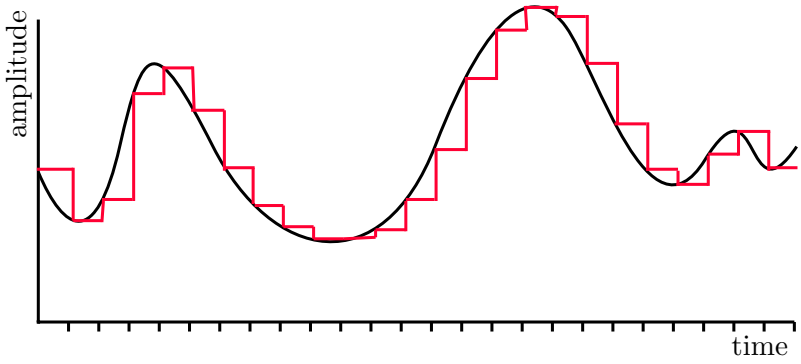


Figure 2.13: The sample-and-hold strategy

After discretizing the magnitude of the obtained time-discrete samples we end up with a set of time-discrete magnitude-discrete samples. In most cases a 16-bit per channel representation is used, although some high-end cases use up to 24 bits of precision in the magnitude.

2.2.2 Time-based Effects [20]

Although all effects used in today's music can be implemented in the time domain, some of them have a strong root in this domain in particular. One class of effects for which this holds are delay effects, which will be discussed first. Another class of effects which can be performed most efficiently in the time domain are volume based effects, as they are based on amplitude modulation. Although we can change the amplitude of a signal in the frequency domain, there are multiple operations we have to perform first, while it is very straightforward in the time domain.

Delay Effects Delay effects are here considered to be all effects that in some form use a time delay to achieve a sound effect. The pure delay/echo, the chorus, the reverb and the repeater, among others, all belong to this class. A short overview of the attributes of these effects will be provided.

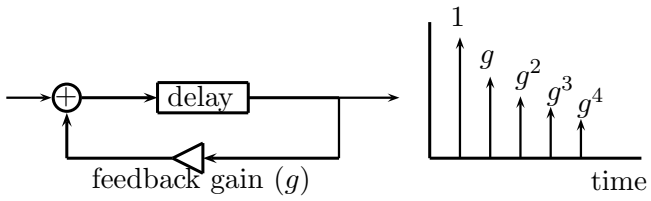


Figure 2.14: The digital comb filter

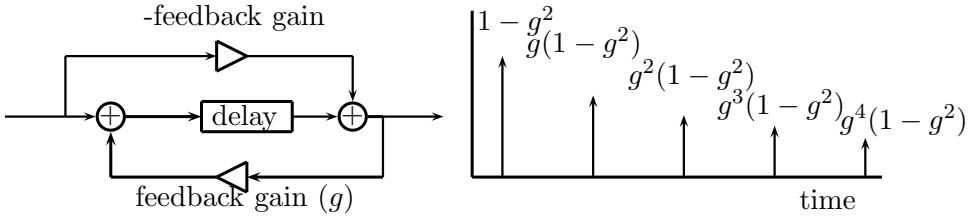


Figure 2.15: The digital allpass filter

Important Filters Before we start on the actual effects, we should take some time to inspect two important filters that many effects are based on. The *Comb filter*, shown in Figure 2.14, can be viewed as the decaying effect of something bouncing back and forth, diminishing its energy with every consecutive step. The *Allpass filter*, shown in Figure 2.15, is similar to the comb filter. However, it has the nice property of passing all frequencies with equal amplitude, although it does introduce a phase lag, which will be explained later on in this chapter.

The Pure Delay A pure delay effect is one the simplest effects, yet very powerful. When applied correctly, a delay can produce a broader sound or spice up other effects. When using the delay with a feedback loop we get an echo effect, shown in Figure 2.16. The attentive reader will have identified the comb filter in the feedback loop. If the feedback gain is smaller than one, the sound that is reheard will diminish over time, creating an echo. In fact, choosing the feedback gain as 0 brings us back to the pure delay. These effects should have a delay time in the range of $[50 : 500]$ ms, depending on the effect you want them to have on the sound.

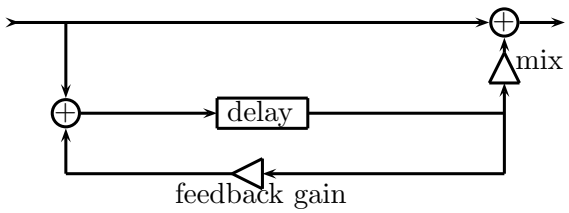


Figure 2.16: The echo effect

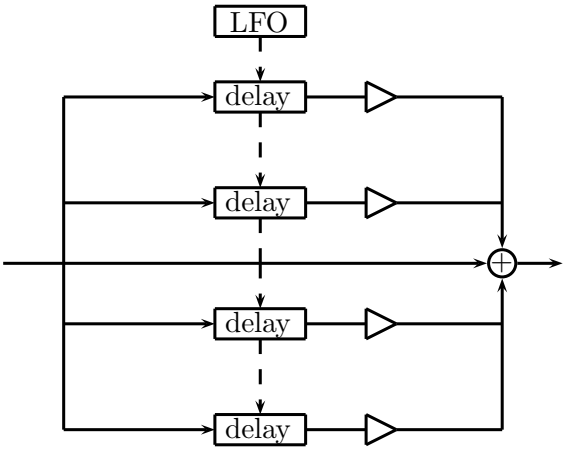


Figure 2.17: The chorus effect

Chorus The chorus effect is achieved by mixing the original input signal with one or more copies of itself, all slightly out of phase and off pitch. This gives the listener the impression that there are several of the same instruments playing and gives a richer more dense sound. The effect is achieved by splitting the input signal and routing every copy through a different delay to achieve different phase. These delays are all individually modulated by a Low Frequency Oscillator (LFO) that runs at a frequency of at most 3 Hz. Figure 2.17 shows this setup. In the figure all delays are driven by the same LFO for simplicity. Modulating the delay with an LFO will change the pitch of the sound. To understand this it is easiest to think of a record playing at the wrong speed. Since the record was recorded at a certain speed, playing it back at a different speed will change the pitch of the sound. Similarly, when we change the length of the delay we effectively read the data faster or slower than intended, modulating the pitch of the sound. If we want to change the sound of the chorus, we can change the amplitude, frequency, or the waveform of the LFO.

Reverberation One of the most widely used effects is the reverberation, or reverb for short. Although most people think of reverb as an effect, it is actually a simulation of an everyday phenomenon. Consider the room shown in Figure 2.18. Although the bulk of the sound travels directly from the source to the receiver, a part of the sound are reflected by the walls, reaching the receiver slightly later and with less power, because of absorption. Because this effect is so embedded in our day-to-day life, we generally have to focus in order to perceive the effect. So why would we apply the effect manually if it is already present? Basically, people tend to listen to music in a space with no or poor reverberation, such as on headphones or in a living room. With the reverb effect applied, one will then still have the same sound properties as though he or she was in -for instance- a concert hall. A rather basic implementation of a reverb effect is depicted in Figure 2.19, known as

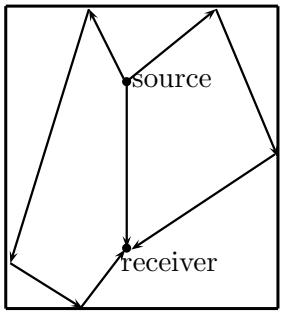


Figure 2.18: A graphical explanation of reverberation

Schroeder's reverberator [21]. The comb filters all have a slightly different delay time, thereby generating several unpredictable, unevenly spaced echoes, similar to a real room. The two all-pass filters increase the pulse density, enhancing the reverb effect.

The reverberator has two main coefficients. First is the feedback gain for every unit, g. This factor should be the same for all comb and all-pass filters. The second is the delay time. The delay time is different for every unit in the reverberator. For the best results, the delay time of the comb filters should be between 30 and 50 ms, and should be relatively prime, meaning they do not share common factors, e.g. $[31; 37; 41; 47]$. The reverberation time, being the time a sounds takes to be reduced to 1/1000 of the original volume, can be calculated based on the two mentioned factors as follows:

$$.001 = g^{\frac{T_{reverb}}{T_{delay}}},\ or\ T_{reverb} = T_{delay} \times \log_g(.001)$$

The Repeater A repeater is a device that is essentially a big controllable delay. This effect is frequently used by solo artists. By enabling the input the device will start "recording", if successively the output is enabled the device will start looping the recorded sound. Additional sounds can be recorded on top of the original recording by enabling the mix. This system is shown in Figure 2.20. It should be clear that since this delay should be able to record several seconds, implementing a repeater in digital hardware requires a very large memory, for example: recording 10 seconds would require $16\ bit/sample \times 48000\ samples/second \times 10\ seconds = 7.68\ Mbit$.

Volume-based Effects Although one might not think of volume as an effect, there are some powerful effect that can be realized by only modifying the signals amplitude.

Volume Control The simplest possible implementation of a volume effect is the direct volume control, similar to a volume knob on any audio-producing

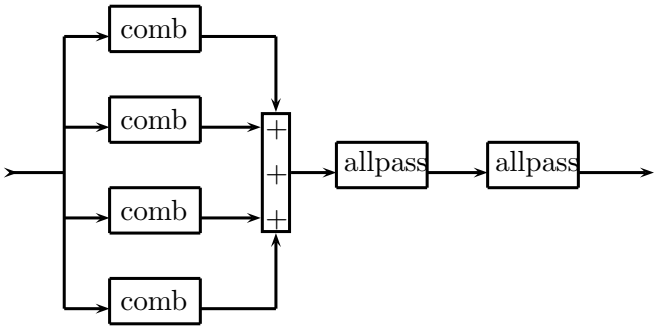


Figure 2.19: The reverb effect

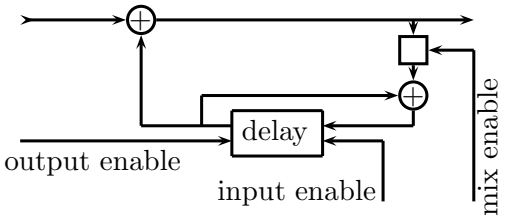


Figure 2.20: The repeater

device. Volume control gives the musician power over the impact the sound has on the listener and therefore the message he or she is trying to convey. Disregarding the fact that most amplified instruments have a volume control on-board, this feature is very important for any musician and is thus featured in this project.

Tremolo The tremolo effect is a more nuanced implementation of volume control. The output of an LFO is added to the amplitude of the signal, creating a slight cyclic variation in the sounds volume, yielding a pulsating sound similar to a church organ. Most electric guitars are equipped with what is called a 'tremolo arm'. This term, however, is incorrect as tremolo refers to a change in amplitude, while a tremolo arm produces a change in pitch, generally referred to as *vibrato*. The frequency of effective tremolo is in the region of $[1 : 20]$ Hz.

Distortion Distortion, shown in Figure 2.21, is in fact a direct limiter on the amplitude of the signal, flattening everything that exceeds the maximum value. Flattening the top of the signal creates a large number of 'unwanted' frequencies that pollute the clean sound of the signal. The lower the threshold, the lower the signal to noise ratio (SNR) will be, and the more distorted the sound will be.

Compression and Expansion When one hears the term compression of sound these days, the first image that we get is that of mp3. This is not the compression that we are discussing here. Compression (and its counterpart expansion) are in fact effects that are identified by their creation of a 'flat' sound. The compression

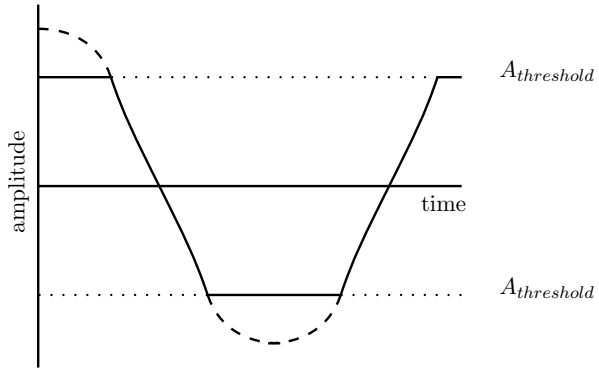


Figure 2.21: The distortion

effect will diminish the volume of any sound that crosses a pre-set threshold, thus reducing the dynamic range. We cannot do this directly, as any signal alternates between its maximum positive and negative value and as such will at some point have a value below the threshold, even if only for a short time. We therefore have to find an average value for the volume, and diminish the sound if it exceeds the threshold for a certain amount of time. These time offsets (both at the start and the end of the effect) are known as the 'attack' and the 'release' time. Figure 2.22 shows how a compressor functions. Expansion is the opposite of compression: Any value over a certain threshold is amplified, thereby increasing the dynamic range. However, in digital hardware we are bound to a certain maximum value for our signal. As such, expansion is more useful when implemented to attenuate values below a certain threshold, creating a similar effect.

Octaver Although this may sound like a frequency based effect, this is an effect that is actually performed very easily in the time domain. What this effect does is take the signal, and mirror all the negative values to their positive counterpart, as shown in Figure 2.23. The fact that this doubles the effective frequency might be hard to grasp at first, and will be easier to understand once one is familiar with the Fourier transform explained in Section 2.3.1. What happens is: if we count the times we encounter the value $1/2$ in the first full period of the signal of Figure 2.23, we would arrive at a frequency of 2, while the value $-1/2$ would also have frequency 2. If we now take the absolute value of the signal, the frequency of the value $1/2$ will have increased to 4, thus doubling the frequency.

Of course this effect has a side effect. By taking away the entire negative part in the signal, we lose a part of the character of the sound, thus creating some distortion. Although one might think of this as a negative effect, if used correctly this can add some excitement and tension to the signal, giving it more appeal than the original.

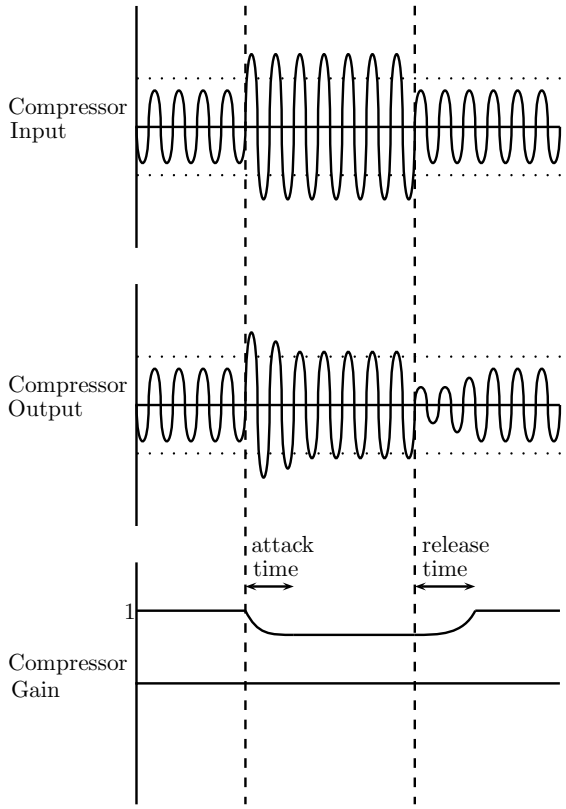


Figure 2.22: The compression effect explained

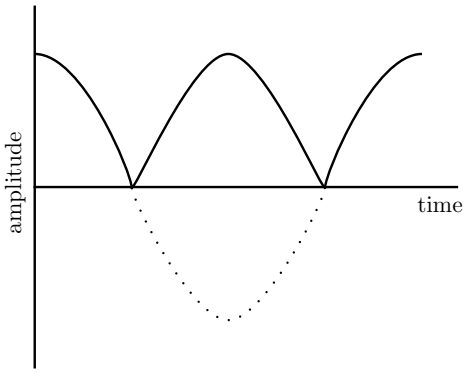


Figure 2.23: The octaver

2.3 The Audio Frequency Domain

This section will handle all explanations that are based around the frequency domain. This includes the transform from the time to the frequency domain and the effects most easily performed in this domain.

2.3.1 The Fourier Transform [22, 23]

The Fourier transform is generally used to break down a signal into its frequency components. This is done by describing any signal as a sum of simple waves. This can be easily seen when we analyse the formula:

$$F(x) = \mathscr{F}\{f(t)\} = \int_{-\infty}^{\infty} f(t)e^{-2\pi ixt}dt \tag{2.1}$$

where

$$e^{inx} = \cos(nx) + i\sin(nx) \tag{2.2}$$

The first thing we have to note is that since $e^{-2\pi ixt}$ is complex valued, $F(x)$ will also be complex valued, meaning any value of $F(x)$ is described in the complex plane as a magnitude and a phase. Now, we can see in the formula that $F(x)$ will only have a high value when $f(t)$ and $e^{-2\pi ixt}$ are oscillating at or close to the same frequency. What we end up with are peaks on the base frequencies of the signal, thus effectively breaking it down into its component frequencies.

One important attribute of the Fourier transform when dealing with purely real input signals is that it is symmetric, more specifically it has an even symmetry in the real output component and an odd symmetry in the imaginary output, i.e.

$$\Re\{F(x)\} = \Re\{F(N-x)\} \tag{2.3}$$

and

$$\Im\{F(x)\} = -\Im\{F(N-x)\} \ \forall \ k \in [0..N/2] \tag{2.4}$$

This basically means that we have the description of the entire input signal contained in the first half of the output.

The Inverse Fourier Transform In order to recreate the original (time domain) input signal, we also need an inverse transform. Luckily, the inverse transform constitutes as little as using the opposite sign in the exponent for the transform:

$$f(t) = \mathscr{F}^{-1}\{F(x)\} = \int_{-\infty}^{\infty} F(x)e^{+2\pi itx}dx \tag{2.5}$$

The Discrete Fourier Transform Now that we have created the frequency spectrum of the signal, we want to start using it. However, the Fourier transform deals with a continuous signal, requiring an infinite amount of computational steps to compute. Since we do not have an infinite amount of time, we want to use a sampled input in order to reduce the necessary computation to an amount we can handle. The effect of this step on the frequency spectrum is that the spectrum will now be divided into small sets of frequencies, commonly referred to as *bins*. This also means that only the exact bin frequencies will only contribute to their own bins. Any other frequencies in between will be "smeared" across its neighbouring bins, although its contribution to its own bin will still be the largest. Coming back

to the theory, this means that we have limited the number of simple component waves, constraining out frequency resolution to ($\frac{f_{sample}}{\#bins}$) Hz/bin, thereby limiting ourselves in the recreation of the original signal.

The Short-Time Discrete Fourier Transform If we are handling a streaming input, i.e. the duration of the input is unknown, we still want to calculate the DFT without waiting for the input to finish. What we then arrive at is known as the Short-Time Discrete Fourier Transform, or STDFT. The STDFT approach constitutes waiting for a predefined number of samples, then calculating the intermediate DFT and then waiting for another set of samples to arrive, and so forth. This, again, means limiting our simple component frequencies even more, up to a resolution of ($\frac{f_{sample}}{transform\ length}$) Hz/bin. Now, the formula for the STDFT looks as follows:

$$X[k] = \sum_{n=0}^{N-1} x(n) e^{-2\pi i (n/N) k} \tag{2.6}$$

It can be easily seen that this algorithm requires N^2 complex multiplications and $N^2 - N$ complex additions. Since experience has taught us that a divide and conquer strategy can yield a complexity of $O(N \log N)$ in these kind of cases, we will attempt to apply this in the next section.

The Inverse Discrete Fourier Transform Just as with the continuous Fourier Transform, the inverse DFT, or IDFT, we need to change the sign of the exponent to convert back to our original signal. However, for normalisation of the summing over N samples, we also need a factor of $1/N$.

$$x(n) = \frac{1}{N} \sum_{n=0}^{N-1} X[k] e^{2\pi i (k/N) n} \tag{2.7}$$

The Fast Fourier Transform In order to analyse and alter the frequency components of the system input, we will use a special form of the Fourier transform, the Fast Fourier Transform, or FFT. The FFT is in essence an implementation of the Short-Time Discrete Fourier Transform described in the previous paragraph. Now, if we apply a divide and conquer strategy to the STDFT we arrive at the FFT algorithm known as the Cooley-Tukey FFT. This algorithm divides the DFT into an odd and an even part, as shown in this formula:

$$X[k] = \sum_{m=0}^{M-1} x(2m) e^{-2\pi i (m/M) k} + \sum_{m=0}^{M-1} x(2m+1) e^{-2\pi i (m/M) k} \tag{2.8}$$

This strategy is then applied recursively until we are left with a set of DFTs of length 2. Because we halve the FFT with every successive step, this algorithm

is called *radix-2* and since we are rearranging the time samples rather than the frequency samples it is furthermore referred to as *decimation-in-time*, or DIT for short. By applying this strategy we are limiting ourselves to an input vector length of only powers of 2, although in practice this is not a hard constraint to meet. Also, as we recursively continue to separate the odd and the even numbers we will end up with a scrambled input such that it hold that:

$$\{0, 1..N-1\} >>$$

$$\{0, 2..N-2,\ 1, 3..N-1\} >>$$

$$\{0, 4..N-4,\ 2, 6..N-2,\ 1, 5..N-3,\ 3, 7..N-1\} >> \textit{etc.}$$

In order to counter this scrambled input we have to pre-shift the inputs to match this structure. To do this we can apply a method known as *bit reverse ordering*. This method takes the index of the input and reverses it bitwise such that

$[b_{N-1} \dots b_1\ b_0] >> [b_0\ b_1 \dots b_{N-1}]$ where b_i denotes the ith index in the bit vector

For instance, if we take assume a 3-bit tree, the value at index 4 would end up at position 1: $4_d = 100_b >> 001_b = 1_d$.

FFT Structure Next, since there is only a limited set of possible powers of e, we can precompute these values, otherwise known as *twiddle factors*. Before continuing there are some important properties to these twiddle factors, commonly denoted as

$$W_N^n = e^{-2\pi i(n/N)} \tag{2.9}$$

Firstly, because of the periodic nature of e it holds that

$$W_N^{n+N/2} = -W_N^n \tag{2.10}$$

The second important attribute is that

$$W_N^0 = 1\ \forall\ N \in \mathbb{N} \tag{2.11}$$

Finally, it should be noted that

$$W_N^n = W_{kN}^{kn}\ \forall\ k \in \mathbb{N} \tag{2.12}$$

Now that we have reduced the problem at hand to a structure of length 2 DFTs, let us take a look at an implementation of one of these basic building blocks of our FFT.

What we can see in Figure 2.24 is that we have reduced the partial problem to a complex multiplication, an addition and a subtraction. This basic building block of our FFT is called a butterfly for visual similarity.

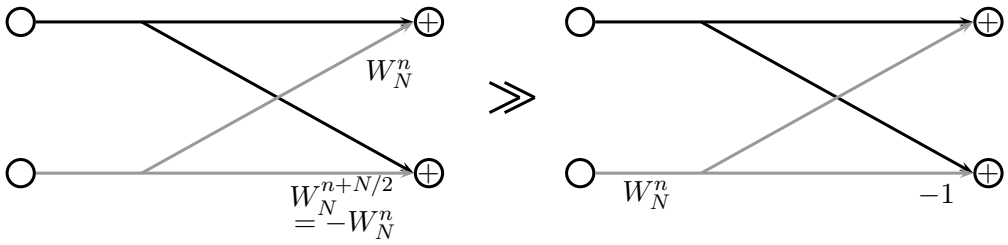


Figure 2.24: The construction of a single butterfly

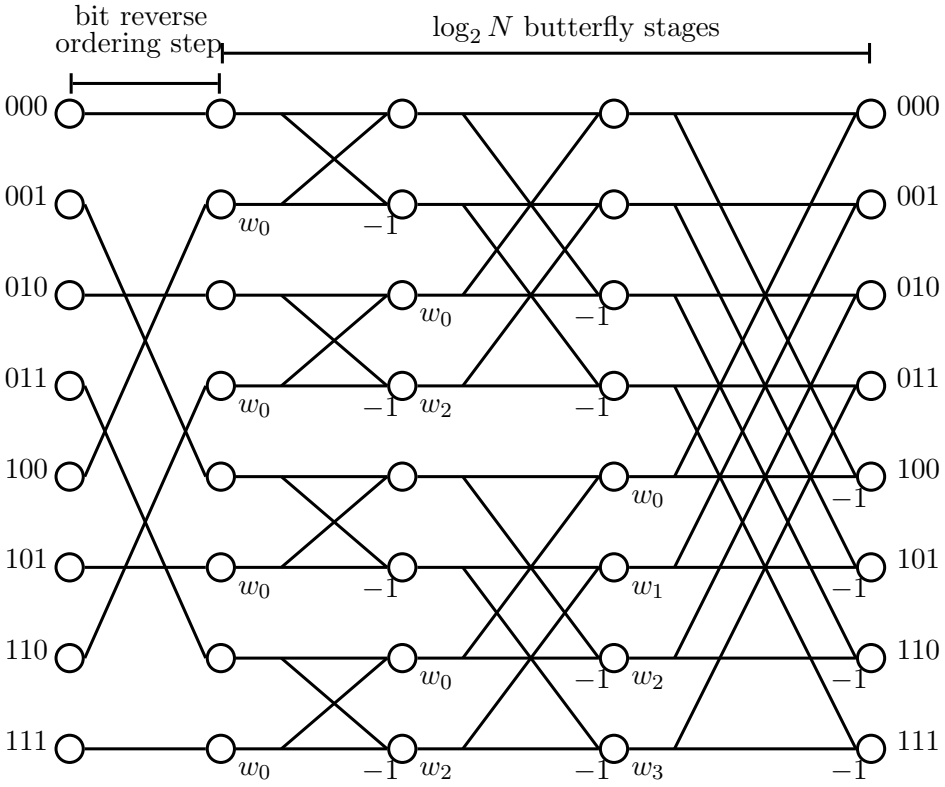


Figure 2.25: The construction of an FFT by stacking butterflies

When we start expanding the tree of butterflies we have to keep in mind to keep simplifying the twiddle factors in order to keep a clean structure. For instance, in a size 4 FFT, the twiddle factors of the second stage will be W_4^0 and W_4^1, while in the size 8 FFT they will be W_8^0 and W_8^2, as in compliance with equation 2.12. When we have expanded the structure to a size 8 FFT, we end up with the structure depicted in Figure 2.25.

The bit-reverse process is added to the structure for clarification. We can now clearly see the final structure of the radix-2 DIT FFT.

Algorithm Complexity Now that we have finished the radix-2 DIT FFT, let us inspect the savings we have accomplished. Remember that for the original STDFT we needed N^2 complex multiplications and $N^2 - N$ complex additions. Now that we have applied the divide and conquer strategy with some minor optimisations we have ended up with $\log_2 N$ stages and $N/2$ butterflies per stage. Combining this with the fact that we need 1 multiplication, 1 addition and 1 subtraction, which in essence is another addition, per butterfly we end up with

$$\frac{N}{2}\log_2 N \quad \text{complex multiplications} \tag{2.13}$$

$$N \log_2 N \quad \text{complex additions} \tag{2.14}$$

We can see that without any further optimisations we have saved a factor $\frac{2N}{log_2 N}$ complex multiplications and a factor $\frac{N-1}{log_2 N}$ complex additions.

Frequency Estimation As mentioned earlier in this chapter, dividing our spectrum up into bins results in a loss of frequency information. While the exact frequency of the bin will result in an exact peak in the spectrum, any frequency in between will result in a "smearing" of the energy over the entire spectrum, as depicted in Figure 2.26.

Since at this point we cannot see all the distinct frequencies anymore, it will be hard to conduct any clean effects on the signal. In order to counter this we can employ frequency estimation. There are several ways to create a system suitable for frequency estimation. We either require overlapping FFTs or we need several FFTs operating at different sample frequencies. Where the former gives us higher resolution locally, being in the region of a specific sample frequency, the latter will help us reduce the smearing effect, on we which we will focus at this time.

Although every non-exact bin frequency will cause smearing across all frequencies, every single one will leave a unique pattern. Since we know these patterns in advance we can use these to our advantage in calculating the frequency that caused it. However, we need at least an overlapping of 75% [23] in order be able to uniquely identify any frequency.

At this point we can apply many useful and nice effects much more efficiently than in the time domain, such as accurate frequency filtering and pitch shifting, which will be discussed in the next section.

2.3.2 Frequency-based Effects [20]

This section will describe all filters that are rooted strongly in the frequency domain.

Filtering When we discuss filters in the audio domain, we discuss one of the most basic, distinguishable and most audible effects we can imagine. In our case, filtering can be employed to remove parts of the signal, allowing us to analyse

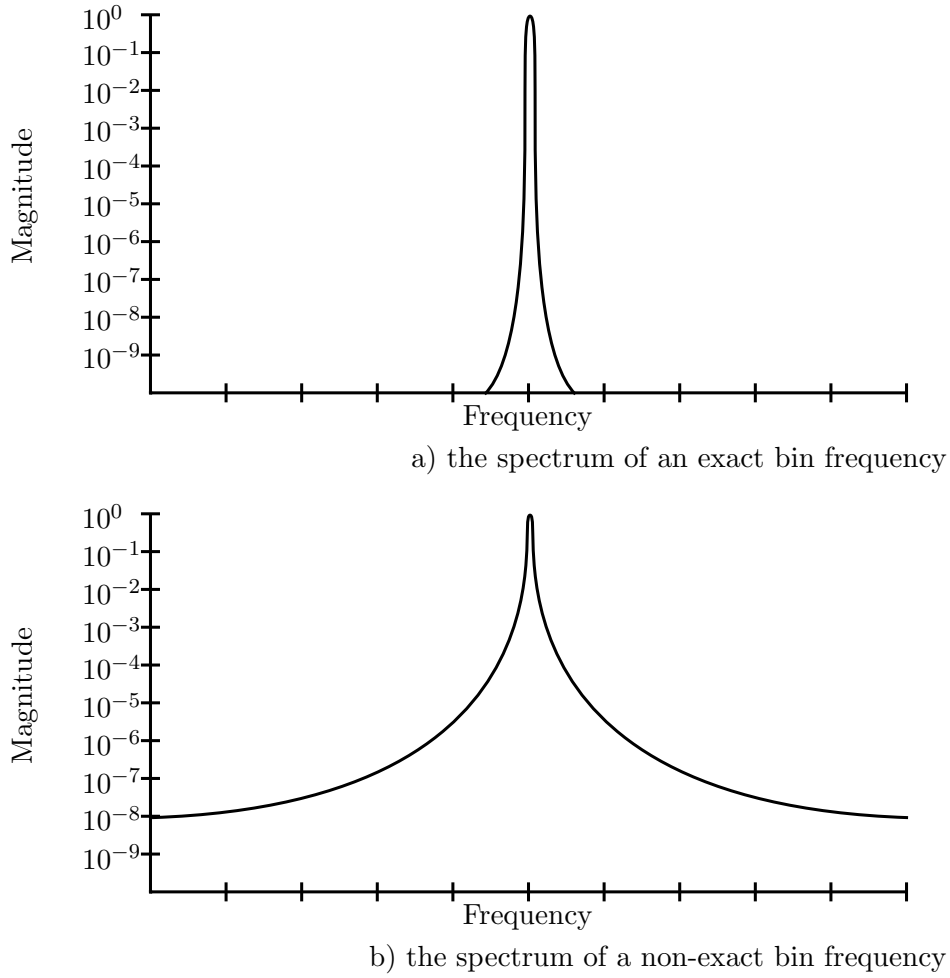


Figure 2.26: The effect of smearing on the spectrum of a pure frequency

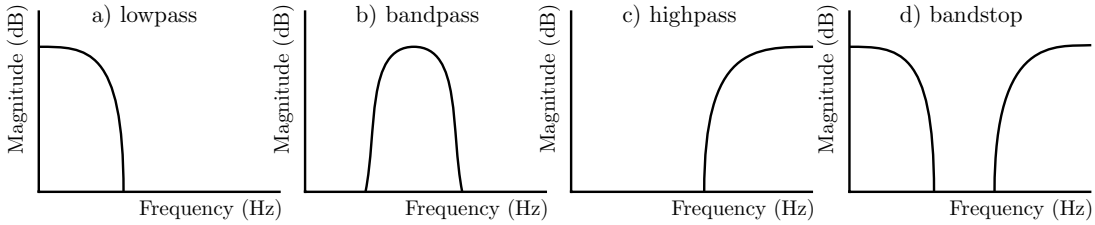


Figure 2.27: The various filters

a certain part of the spectrum. The most basic implementations of a filter are the Low-, Band- and High-pass filter and their blocking counterparts. Figure 2.27 shows these basic setups.

The main advantage of applying these filters in the frequency domain is the complexity of the operations. While you need several multiplications, additions

and divisions, of which the last is quite resource expensive to implement, to achieve filtering in the time domain, a single multiplication yields the same results in the frequency domain. If we devise a strict filter, we can even do without the multiplier by just passing the desired frequency bins and blocking the rest.

The main problem of filtering in the frequency domain is the smearing effect: the blocked bins will also contain a small piece of our desired signal and vice versa. This means that in order to achieve a truthful filter we have to apply frequency estimation.

Vibrato The vibrato effect, as mentioned earlier in this chapter, is the effect of quickly but slightly varying the frequency of the signal, resulting in a minor pulsating sound, similar to a guitar player moving an excited string perpendicular to the neck of the guitar. By moving the string in this fashion, the length of the string –and thus the frequency– is varied. Although this effect has a strong base in the frequency domain, it can be quite easily implemented as a delay effect as well.

The Wah Effect In some special cases, such as the wah effect, pieces of the spectrum are extracted and added to the original to alter the sound. The wah effect is basically a band-pass filter with a tunable centre frequency whose output is added to the original signal. When the centre frequency is varied up and down at a low frequency (around 1 Hz) we create a sound that is similar to a person saying "wah-wah", hence the name.

Phase Shifting As explained earlier, the representation of a signal in the frequency domain consists of a magnitude and a phase. Where a filter essentially changes the magnitude, we can also change the phase of the signal. Although this could be represented quite easily as a delay element in the time domain, we are looking for a delay in the range of 10 ms to achieve our phase shifting, more often referred to as *phasing*. Since generally the sample frequency is 48 kHz, resulting in a sample time of about 20 μs, we require quite a lot of memory to store 500 samples, while we do not require any additional memory in the frequency domain. Although we call this a delay, the human ear can not perceive it as such. Only delays of over about 50 ms will be heard as an echo.

A phaser is in essence an all-pass filter: a filter with a flat magnitude transfer, but with a (non-linear) phase transfer, as depicted in Figure 2.28. After the filter, the result is mixed with the original signal. If we now change the delay of the all-pass filter, effectively moving the phase lag back and forth at a frequency of around 1 Hz, we create a "whooshing" sound similar to the sound that is created by the wind blowing back and forth.

Flanging Another effect that is usually generated with a delay is *flanging*. A flanger is a variable delay element whose output is added to its original. This is

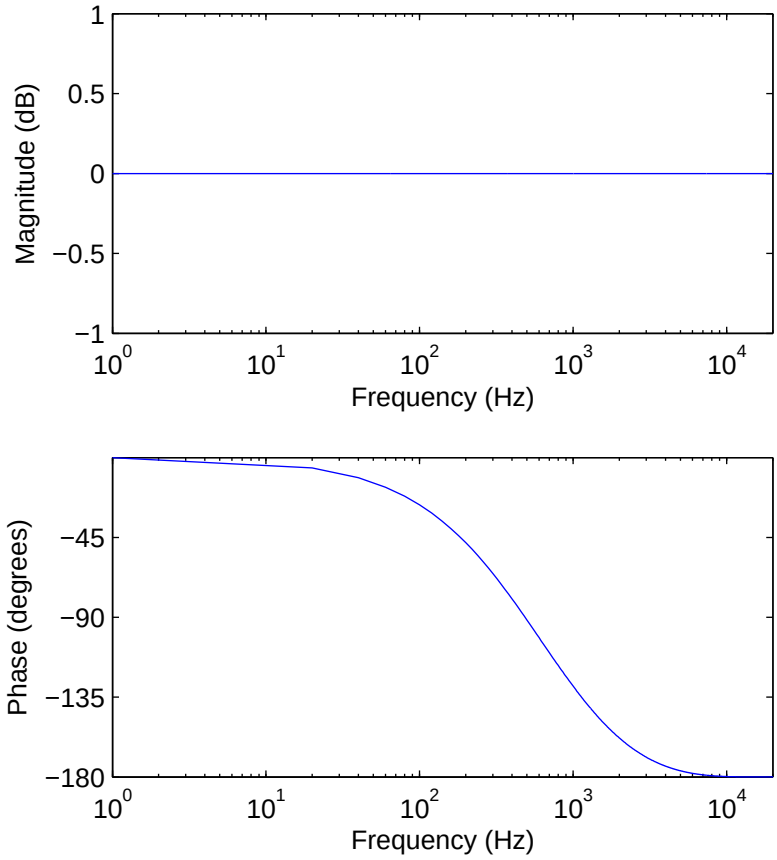


Figure 2.28: The Bode plot of an all-pass filter

the linear case of the all-pass filter depicted in Figure 2.28, for instance, a frequency of 180 Hz would be delayed by -180 degrees and added to its original, cancelling it out, the same holding for any frequency of $180 + 360 \times k$ Hz, where k is an integer. This effect creates "notches" in the signal transfer, as shown in Figure 2.29.

This is called a *comb filter*. As with the phaser, varying the delay causes a "whooshing" sound, although slightly different from its non-linear equivalent. In Figure 2.29 this could be viewed as the "comb" acting like a spring, expanding and retracting in time. Flangers are often used by drummers to broaden their sound, while keeping it fairly straightforward.

2.4 Conclusion

In this chapter, all the information required to understand the work done during this project, and the processes and functions described in this thesis, is contained.

A short history of reconfigurable computing shows us that FPGAs populate a

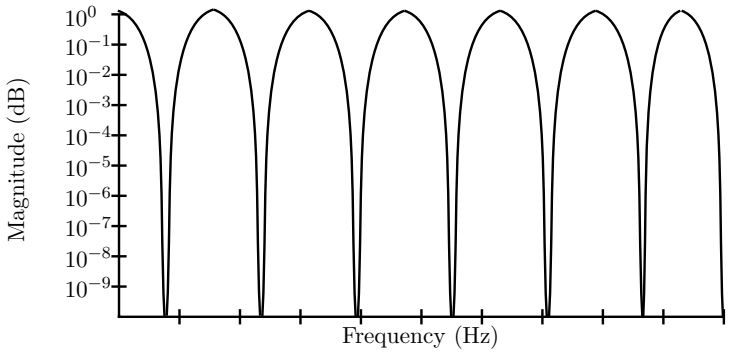


Figure 2.29: The Bode magnitude plot of a comb filter

region in hardware in between the fast but rigid ASIC and the slow but flexible GPP. Rather than excelling at either speed or flexibility only, the FPGA can be designed to fit a large region of this trade-off, sacrificing speed for flexibility or vice versa. Along with the description of how an FPGA works and how it is (re)configured, the notion of partial reconfigurability is introduced, referring to a technique where only a small portion of the FPGA is configured, leaving the remainder intact. Partial reconfiguration can be divided into static and dynamic partial reconfiguration, where the former means the FPGA does not operate while it is partially reconfigured, whereas the with the latter strategy the remainder of the FPGA will continue to function whilst partially reconfiguring.

We continued by discussing several pieces of work directly related to this project. Despite the fact that there are many papers regarding partial reconfiguration, few hold a direct relation to this project. As such, several researchers that have compiled other related work were cited, in order to form a complete picture of this field of work. Furthermore, there have been many publications by Xilinx that are relevant to what this thesis is proposing to do, as this project is implemented using their product, and therefore the majority of the citations have direct tie-ins with this company.

Apart from taking in a sounds signal and sending out a sounds signal, this project incorporates additional input and output. The input is in the form of a keyboard, allowing us to control the processes existing within the FPGA. The additional output is a visual output to a screen, displaying the sounds time or frequency spectrum, enabling us to visually analyse the sound and the difference between the unaltered and the altered sound.

As this project is for the most part built up out of custom made blocks, a large section of this chapter is dedicated to explaining the workings of these modules. The most notable part covers the functionality of the Fourier Transform and its derivatives. The Fourier Transform is used to derive the relative frequencies from a signal. In the case of sound, the extraction of the frequency pattern provides us with a good insight into the frequency composition of the sounds, and enables us to extract certain frequencies from the signal, in order to process them separately.

The remainder of this chapter handles the functionality contained within the effect modules, designed to alter the sound stream running through the system.

3 System Design

This chapter will describe the design step from the project goals to the design to be implemented, as well as the motivation for the design choices made to accomplish this.

This chapter is divided into five sections. First, the translation from design requirements to a system design will be discussed and motivated in Section 3.1. Section 3.2 will elaborate on the choice for the used technology, whereas Section 3.3 will discuss the changes made to the design while the project was in progress, as some unforeseen restrictions were uncovered. Afterwards, The investigation regarding the incorporation of this project in the MOLEN polymorphic processor will be discussed in Section 3.5, after which the chapter will be concluded in Section 3.6.

3.1 The Initial System Design

For the design of the system, a project with a clearly visible relation to partial reconfiguration was required. To best visualize and emphasize the fact that only a part of the system is reconfigured, it was decided to design a system where there is a drastic change of behavior, while the system remains fully operational. A good way to create a clearly different sound (as we had already decided on a audio implementation) is to alter the sound signal in some way using filters and other effects.

In order to best visualize the effect in case the audible effect is not clearly distinguishable, and to add to the user friendliness of the system, we decided to add an output to a screen. Although it might be useful in some cases to visualize a time spectrum graph, most effects are best visualized in the frequency domain. As such, an FFT was required to generate the frequency spectrum. Since we already need an FFT, we might as well incorporate some effects that are easily performed in the frequency domain and transform the signal back to the time domain with an inverse FFT.

At the location of both the time and the frequency domain effects there were to be an unspecified number of cascaded effects, depending on the desire of the user. Figure 3.1 shows the first system concept based on these initial constraints. In this figure, the designated partial reconfigurable units are marked by a double box, as well as the designation PRU (Partial Reconfigurable Unit).

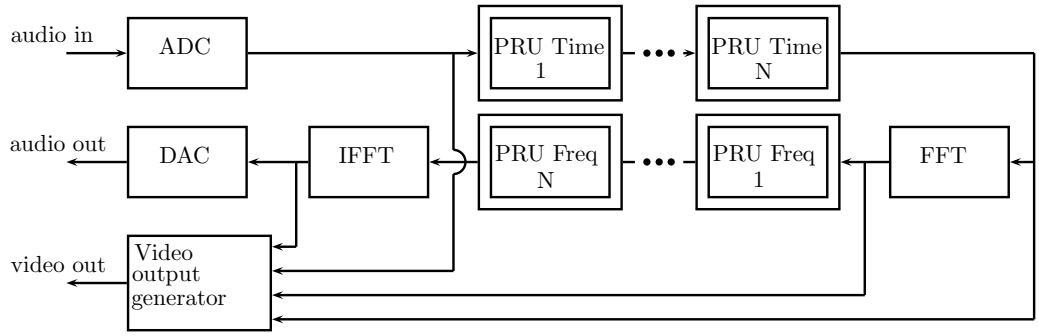


Figure 3.1: The first system concept

3.2 Technology Motivation

Looking at the baseline requirements for this project, we cannot get around the use of an FPGA, as the desired design goals mandate the use of a platform with support for partial reconfiguration. Although there are several FPGA manufacturers, there are only two that currently support partial reconfiguration. Since Xilinx boards are at our disposal at this department of the Delft University of Technology, it would seem quite logical to use one of these boards for this project. If we then look at the hardware best suited for partial reconfiguration, we would end up with a Virtex 4 or Virtex 5, as these FPGAs do not put any constraints on the location of the partial reconfiguration region (PRR). However, since one of the goals of the project is to research integration in the MOLEN project, we are bound to the Virtex-II, as the only existing implementation of the MOLEN was built on this platform. Since the Virtex-II has a built in ADC and DAC, as well as suitable in and output audio connections, it is still very suitable for the purpose of this project.

As previously discussed, Raaijmakers [24] proposes a partial reconfiguration method devoid of any placement constraints. Although this strategy would be ideal for this project, the tools proposed and presented in the mentioned paper do not seem to be available. As such, this project is bound to Xilinx tools that are prepared for partial reconfiguration, despite the fact that it is mostly in an experimental phase. The aforementioned tools, being ISE and PlanAhead both pose some difficulties when working with partial reconfiguration. ISE requires a specific version (being 9.2iPR8) for partial reconfiguration support, and PlanAhead supports partial reconfiguration through a command line setting, but there are still a lot of design flaws –in the worst case complete program crashes– when using this setting. Furthermore, the method for partial reconfiguration that Xilinx uses (explained in Chapter 2), poses several constraints on the design, for instance on the number of usable PRRs. As such, this implementation has taken a slightly different form throughout its lifetime, which will be elaborated on in the next section.

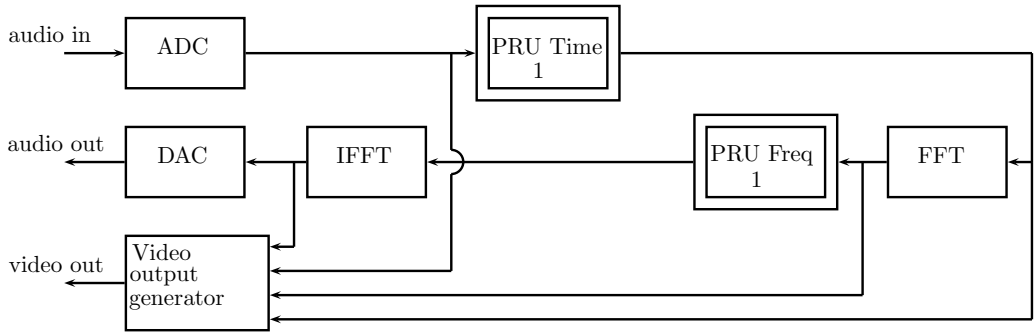


Figure 3.2: The final system concept

3.3 The Final System Design

As this project advanced, it became clear that not all of the features of the initial system could be implemented in the set time frame. For this reason the design was tuned down to the final prototype shown in Figure 3.2.

The reason for this modification is the following: At the time of writing, Xilinx PlanAhead (v10.1.8) supports only a single PRR. Although this may be fixed in future versions, there is currently no way to implement additional modules. However, Figure 3.2 still shows two PRUs, due to the fact that both regions have been modified to fit into a single PRR, making it impossible to reconfigure them separately, but keeping the flexibility of designing filters for both domains.

FFT Length Issues A second unforeseen constraint is in relation to the frequency domain. Unfortunately it turned out that the 128-point FFT running at a 48 kHz sample rate used for this project by definition has a resolution of only $48000/128 = 375$ Hz/bin. This imposes constraints on the display, as the lower frequencies cannot be separated as nicely as one would want. For instance, most equalizers use frequency bands of $[20 : 50]$ Hz for the first, $[50 : 100]$ for the second, $[100 : 200]$ for the third and $[200 : 500]$ for the fourth band. The best our system could do here is $[0 : 375]$ for the first band and $[375 : 750]$ for the second.

Although this first constraint is not very crucial to the project, the second consequence of this low resolution is: For a reasonable frequency estimation we require a much higher resolution. For instance, if we want to be able to uniquely identify any note of the third octave and up, we would need a resolution of $138,59$ Hz (Db3) - $130,81$ Hz (C3) = $7,78$ Hz/bin. This would result in an FFT of length $48000/7,78 = 6169$, and as we can only use power-of-two lengths, we would need a 8192-point FFT. This would not only require far too much memory and space on the FPGA, but also be much too slow to use in this real-time system. For illustration, using Equation (2.13), we arrive at $\frac{8192}{2}\log_2 8192 = 53248$ butterflies, where a 128-point FFT uses only 448 by the same equation.

FFT Length Solution To resolve the issue addressed in the previous paragraph, the system could be redesigned to use several FFTs in parallel. The first would be used with a low sample rate, for instance 1000 Hz, resulting in a resolution of $7,81$ Hz/bin when using a 128-point FFT, getting very close to the desired resolution in that region. A second FFT could then for instance be used with a sample rate of 5 kHz resulting in a resolution of 39 Hz/bin, which would be quite feasible for the frequencies in the fifth octave and up (frequencies over 500 Hz), where the smallest required step is from C5 to Db5 which is $554,37 - 523,25 = 31,12$. Furthermore this strategy would require very accurate filtering of the signal in order to divide it into the rough sets of frequencies needed by the separate FFTs. Granted the time required for this change in the project would be too long, this was not included in this project. If at any time any further work was to be done by a fellow student on this project, this addition is recommended to be the first to be made. Not only will the existing filtering improve in quality by a large amount, but it would open up a new set of filtering possibilities altogether.

Frequency Estimation Finally, the issue of frequency estimation discussed in Section 2.3.1 should be addressed. Many effects discussed in the previous chapter require some sort of accurate frequency knowledge, and therefore can not be implemented on the current system. In addition to the more accurate FFT method described in the previous paragraph, we also require an overlapping FFT structure exceeding the minimal requirement of 75% overlap. Since the accurate FFT method was not implemented, this addition to the system will not be included in this project either. However, anyone conducting further work is advised to also feature this FFT overlap, as again it opens up a entirely new set of filtering possibilities.

3.4 Interface Considerations

To start out, two signals were introduced to the system, being the system clock and the reset. The system clock is assumed to run at 100 MHz, and the reset signal is assumed to be active low, i.e. reset is enabled when 0. Soon to follow was a third signal, being the sample clock, running at 48 kHz.

At the start of this project, it became clear that all modules to be implemented in the partial reconfiguration regions were to have an identically defined interface. So, before starting the implementation process, a PRAGMA standard PRU interface specification was designed, as follows in Table 3.1. Furthermore, the standard dictates that the least significant bit of the control input is an (active high) enable signal for the module. Upon disabling this signal, the module should assume a transparent behavior, i.e. output = input. Additionally, the module should not run when the input clock is disabled, as well as output all zeros when the reset signal is low.

Table 3.1: The PRAGMA time domain PRU standard interface. Changes to the interface made at a later stage are shown in parentheses

	sound data left	sound data right	control data	clock 100 MHz	clock 48 kHz	reset
in	16(24)	16(24)	8	1	−(1)	1
out	16(24)	16(24)	4	–	–	–

During the development of the system, several design requirements arose, altering the standard definition slightly. First of all, the intermediate signal changed from 16 to 24 bits, which is shown in parentheses in the table. Secondly, the frequency PRUs were easier to operate with several extra control signals. As such, the frequency domain PRU standard differs slightly from the time domain version, and is shown in Table 3.2. A second clock was added to the interface, allowing an insight into the sample rate. Also 8 bits were added to the control input. The 8 most significant bits of this control signal (15 through 8) are mandated to contain the current sample number, allowing us the knowledge of which frequency we are currently dealing with.

Table 3.2: The PRAGMA frequency domain PRU standard interface. Changes to the interface made at a later stage are shown in parentheses

	sound data left	sound data right	control data	clock 100 MHz	clock 48 kHz	reset
in	16(24)	16(24)	8(16)	1	−(1)	1
out	16(24)	16(24)	4	–	–	–

3.5 Incorporation in MOLEN

One of the design goals for this project states that the feasibility of incorporating this system into the MOLEN polymorphic processor is to be investigated. At first sight, this should be very well possible, seeing as the MOLEN architecture is capable of partial reconfiguration [1], and would even be able to control the partial reconfiguration by itself. There are, however, several reasons why it is impossible at this stage to combine this project with the existing MOLEN implementation.

The first problem that arises is that the existing MOLEN prototype only has an implementation for Xilinx ISE 6.3*i* and Xilinx ISE 8.1/8.2. As mentioned earlier, partial reconfiguration using PlanAhead currently only supports Xilinx ISE 9.3*i*PR8. Porting the prototype to a newer version of ISE is a daunting task in itself, and is as such beyond the scope of this project.

Other than the fact that the versions of the working environments do not match, there is another fundamental problem withholding the combination of both projects. Even if the MOLEN prototype would support the required version of Xil-

inx ISE, an elaborate redesign of this prototype would be necessary in order to suit it to the requirements that PlanAhead poses on any reconfigurable project. One would for instance have to insert bus macros, although this is likely to be one of the easier tasks. This suggests that the strategy of partial reconfiguration as designed by Xilinx might not be a feasible option for MOLEN, and a different approach, such as the strategy proposed by Raaijmakers [24], would conform better to the approach used in the MOLEN architecture.

3.6 Conclusion

The goal of this project is to create an audio manipulation platform using partial reconfiguration. In order to best visualize the effect a system was designed that has both an audio output and a video output. The initial system is built up out of several partial reconfigurable time domain effects, followed by a frequency transform, several partial reconfigurable frequency domain effects and a transform back to the time domain. Apart from this main path, video data is gathered from several points in this path, feeding information to the VGA output.

Due to several unforeseen constraints, the initial system was reduced to a single partial reconfigurable region, only able to reconfigure a single module partially, leaving the rest fixed. Furthermore, multiple effect previously scheduled for the frequency domain were moved to the time domain or left out completely.

In order to create the possibility for partial reconfiguration, a special interface was defined. At a later stage of the project, this standard was divided into a time domain standard and a separate frequency domain standard.

The possibility for implementing this project in the MOLEN polymorphic processor was investigated. Using the technology this project is based on, however, does not seem to be a feasible approach to accomplish this. First, the implementation of the MOLEN processor would have to be adapted to the required version of Xilinx ISE 9.3*i*PR8, which is a time consuming project. Once this is complete, the MOLEN implementation would have to be redesigned in order to fit the constraints posed by Xilinx PlanAhead.

4 Implementation

This chapter describes the implementation steps takes while creating the modules that constitute this project, as well as the step taken to realize partial reconfiguration. If a module was not designed and implemented by me, this is stated clearly in the respective section, together with a link or reference to the original design.

This chapter is divided up into five sections. First, Section 4.1 handles the particulars regarding the inclusion of partial reconfiguration in this project. Next, Section 4.2 will discuss the time domain, while Section 4.3 will discuss the frequency domain. Section 4.4 contains a description of the implementation of the peripheral modules that are not in the direct path of the sound data, but fulfill a role in controlling or handling the data surrounding the system. Finally, Section 4.5 will review what has been discussed in this chapter.

4.1 Partial Reconfiguration

The basic requirements for partial reconfiguration are as follows. First, the design has to be modular, meaning the design has to be divided up into modules, because we need clear separation between the static core and the reconfigurable core. Since this project is already module based, this requirement has been fulfilled. Secondly, all reconfigurable cores need to have the exact same interface to the static core. This was discussed and handled in Chapter 3, and as such this requirement was met as well. The remaining requirements, such as the global clock path and bus macros, have been addressed earlier, and have all been met as well. What remains now is the actual implementation.

Since an off-the-shelf solution to partial reconfiguration using PlanAhead was chosen for this project, the steps taken to implement this project will not be elaborated on here. For a complete explanation of how this project was designed and generated using PlanAhead and ISE, the reader is referred to Appendix B, which shows all the implementation steps taken to generate the partial reconfigurable bitfiles.

What remains to say here is that the implementation using PlanAhead was successful and that a single partial reconfiguration region is operational within this system, as discussed in Section 3.3.

4.2 The Audio Time Domain

This section describes the implementation phase of the project. During the implementation phase it became clear that several modules would be too large to implement. As such, all modules in this project only use the left channel of the stereo input. The right input is added to the modules as both an input and an output for future use, but it is never used in the current state of the project. This holds for both the time- and the frequency domain.

4.2.1 AD & DA Conversion

The used wrapper for the AC'97 AD and DA converters on the Virtex-II Pro development board has been borrowed from the Xilinx Built-in Demo project [25]. This module configures the converters and outputs the serial data as two 16 bit vectors, as well as takes two 16 bit input vectors, in both cases one vector for the left and one for the right channel. Several small modifications to this core were made in order to be able to incorporate it in this design.

One important function of this core is that it outputs a synchronize signal with the frequency of the sample rate every time a new sample is ready. This signal has a duty cycle of 1/16 and is therefore not suitable to be used as a clock. It can be used, however, be used to construct a clock signal, which is essential to the correct functioning of a large portion of this design.

4.2.2 Sample Clock Generator

Since the synchronization signal from the AC97 wrapper core does not have a suitable duty cycle to be used as a clock, we need a different solution to obtain this clock. The AC97 Bit Clock is available on a system pin, running at 12240 kHz. In order to obtain the 48 kHz clock from this the AC97 wrapper divides it into 255 timeslots, resulting in an exact frequency of 48 kHz. Since the AC97 core only uses this locally and does not actually output this clock, a custom clock generator based on the same principle was created. Xilinx has Digital Clock Managers (DCMs) available to generate accurate clock signals by division or multiplication, but the minimal output frequency is 1 MHz, and therefore these are not feasible in this case. In order to stay synchronized with the actual samples, this core waits for the first issue of the synchronization signal from the AC'97 wrapper core, and starts the counting process from there.

4.2.3 Time Based Effects

This section will relate the implementation phase of the effect modules. Please note that all modules are required to use their least significant control bit as an enable, which will not be repeated in every paragraph. Although implementing the essence most of these effects is straightforward, most of them still took a lot of time to configure just right in order to get the desired sound.

The Delay Effect Implementing a pure delay is quite straightforward. If we model the delay with a FIFO, we require a depth of half the sample rate to use a delay of up to 500 ms, resulting in 24000, or actually the first higher power of two, 32768. We can then use 4 of our control bits to control the delay length. The 4 control bits are then shifted to the left by 11 bits, shifting in 1's with every step, changing the value range from $[(2^0 - 1) : (2^4 - 1)]$ or $[0 : 15]$ with steps of 1 to $[(2^{11} - 1) : (2^{14} - 1)]$ or $[2047 : 32767]$ with steps of 2048. If we map this to the time axis, this is equivalent to a range between 43 ms and 683 ms, with steps of 43 ms. Our theoretical desirable range of $[50 : 500]$ ms falls nicely within our possible range.

Since we cannot change the length of the FIFO if we want to change our delay, we need some control parameters. As the FIFO has a separate read and write enable, the write enable is bound to to high, resulting in the FIFO continuously reading data on the input and writing is into its memory. Subsequently, the read enable on a timer was put on a timer, counting down from the desired delay length. Once this timer reaches zero, both the input and the output work continuously, keeping the data pipeline at the desired length. If any value is changed the FIFO is reset, and the process starts anew.

The remaining 3 bits of our control signal are used to select the decay of the signal. By default, the signal is right shifted by 3 bits and as such 1/8 of the original. The signal is then multiplied by the 3 bit value of the control signal, resulting in a maximum of 7/8 being fed back to the input signal. To induce minimal losses in the signal quality, the initial value is only shifted by 1 if the control value is 4, and only shifted by 2 if the control value is 2. For the best signal quality these values should be used.

Pitch Modulation Now that we have arrived at the pitch modulated effects, we are faced with a challenge: How do we model the variable delay with the LFO explained in the theory? We are unable to change the length of the FIFO at runtime and we can not change the sample frequency. However, we *can* change the sample frequency without our effect module.

Based on this notion, a FIFO with a varying output clock was designed, as depicted in Figure 4.1. If we change the frequency at which the samples are read back from the FIFO, we create an asynchronicity with the rest of the system. This will result in reading some samples multiple times, causing the pitch to lower, while "missing" some other samples, effectively increasing the pitch. Of course, since we write the samples into the FIFO at the sample rate, we need to alternate raising and lowering the pitch in a well-distributed way, keeping the *average* clock speed equal to the sample rate.

Please mind: The sample signal in Figure 4.1 is stated as 1 Hz as an example, as is the 10% frequency and pitch change. This is to clarify the effect, and by no means a restriction. Furthermore, a 10% change in the clock frequency may or may not induce a 10% change in the pitch. The amount the frequency of the

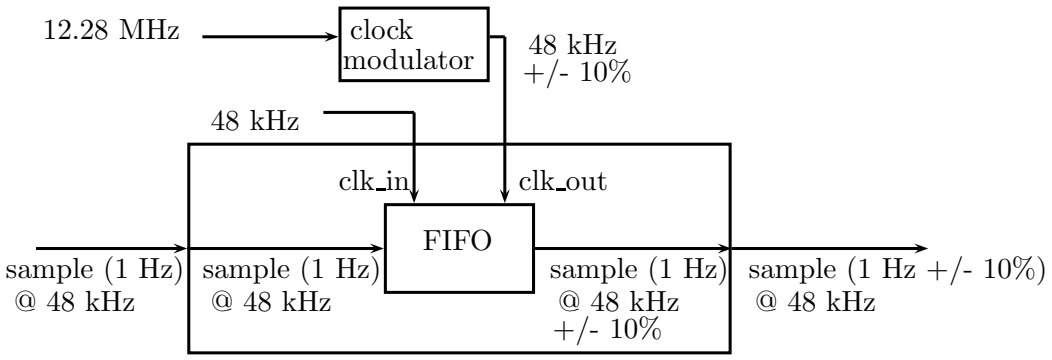


Figure 4.1: The modulation of the sound's pitch by using a FIFO with different clocks

sample is changed varies with the frequency of the sample, and the value of 10% is only used to show that the modulation of the clock frequency has a direct effect on the frequency of the sample.

The clock modulator shown in Figure 4.1 functions in the same way as the sample clock generator does. If we want to achieve the exact sample rate, we need to divide the 12, 28 MHz clock by 256. However, if we for instance add or subtract 10% of this number by adding a integer-valued sine wave with an amplitude of 25 and at a frequency relevant to the effect we are trying to create, we create a 48 kHz *average* clock with an embedded pulse-modulated sine wave. Changing the amplitude, frequency and shape of the pulse-modulated signal will change the sound of the effect.

The Chorus Effect Having implemented a pitch modulating circuit, we can now create a chorus effect. The implemented chorus effect uses 3 pitch shifters, all with a different phase and a different period. According to the template, the least significant bit of the control signal is the enable of our module. The remaining bits are unused, but could be used to control the delay and/or the phase shift of the module, when implemented by the user.

Reverb The reverb effect was implemented exactly as explained in the theory. 4 comb filters were placed in parallel, cascaded with 2 all-pass filters. The 3 most significant bits of the control vector select one of these units, while the least significant bit again functions as an enable. The remaining 4 bits are used to select a value for the selected unit.

Repeater Due to a lack of onboard memory, the repeater was not implemented. In the future one might implement a DDR controller, opening up enough memory to make this effect feasible to implement.

Volume Control The volume controller was implemented using the available multipliers. The most significant bit (bit 7) is used to enable multiplication, and bit 6 to enable division. Bits 5 through 3 are used for the multiplicand (range of 0 up to 7). Table 4.1 shows the results for the various combinations of settings. The division scheme is made in such a way that we do not actually need division; the input is pre-shifted in the same way as with the delay effect. Note that any multiplication could lead to clipping of the output signal if the result is larger than can be contained in a 16 bit signed vector.

Table 4.1: Settings of the volume controller explained

Multiply	Divide	Result
0	0	output = input (Transparent)
0	1	output = 1/8 x multiplicand x input
1	0	output = multiplicand x input
1	1	output = 0 (Mute)

Tremolo The tremolo effect is implemented using a sine lookup table. This implementation uses a relative amplitude, meaning it scales with the amplitude of the actual sound. The ratio between the original signal and the tremolo wave is set at $1 : 3$. The sine lookup table uses 16 bits to represent a value between $[-1 : 1]$ and a table length of 32 containing 1/4 of the complete sine wave, meaning that one full period of the resulting sine wave contains $4 \times 32 = 128$ samples. If we now add the same value from the sine table to our signal for $48000/128 = 375$ consecutive samples and then continue to the next table entry we end up with a 1 Hz tremolo. Control bits 7 through 5 can be used to control 8 frequency settings, presented in table 4.2.

Table 4.2: Control values of the tremolo effect and their corresponding frequencies

Control Value	Step Size	Frequency
000	375	1.000 Hz
001	187	2.005 Hz
010	125	3.000 Hz
011	75	5.000 Hz
100	37	10.135 Hz
101	25	15.000 Hz
110	19	19.737 Hz
111	15	25.000 Hz

Distortion Distortion was implemented in a full range scalable manner. The 4 most significant bits of the control vector are used to select a value v between 0 and 15. We then shift a value of 1 v times, which is used as an upper bound for the

signal. Any amplitude larger (in the absolute sense) than the threshold is changed to the value of the threshold. Note that this means that a control value of 0 results is the worst distortion, while a value of 14 results in the mildest distortion. a shift value of 15 would result in an upper bound of -1 and is therefore bypassed, and used to disable the effect.

Compression and Expansion Compression and expansion are implemented in a similar fashion to distortion. The most significant bit is used to select either effect (0 being compression and 1 being expansion), and bits 6 through 3 are used to select the trigger level. The remaining bits, 2 through 1, are used to select the attenuation factor. The affected input in shifted by 2 bits, and then multiplied by the attenuation factor, obtaining a attenuation factor in the range of $[0 : 3/4]$. The used attack and release time are both set to 1023 samples, resulting in a minimum detection frequency of approximately 50 Hz.

Octaver This is possibly the most straightforward effect. All negative input is multiplied by a factor of -1, resulting in the desired effect. Only one minor bug occurred in the test, because the absolute value function would not substitute the maximum negative value, as it has no positive counterpart in the 2's complement notation. A simple condition was used to correct this problem.

4.3 The Audio Frequency Domain

Keeping in mind the restrictions defined in Chapter 3, the implementation steps undertaken to create the modules that are still possible to implement within the given restrictions will be explained.

4.3.1 The Fourier Transform

The FFT module is one of the largest modules in this design, as well as one of the most timing-critical. Although the FFT has to sample at the set sample rate of 48 kHz, we want its core to operate at 100 MHz in order to reduce the input-output lag to a minimum.

Xilinx FFT Core Implementation To start out, it was decided decided to use one of the Xilinx FFT cores [26]. Despite the fact that the are many parameters to modify this cores behavior, there is no direct way to enable the desired dual clock speed. In order to implement this behavior, a basic wrapper using Xilinx FIFO cores was created. This FIFO would have its input enable connected to the 48 kHz sample clock and its output enable to the 100 MHz system clock. When the 'full' flag of the FIFO would go high, the wrapper would enable the output clock, thus feeding the FFT data at 100 MHz until the FIFO was emptied. At the output, a similar reverse structure was built.

In order to establish an efficient transform length we need to consider the number of cycles we can spend on performing the FFT without creating any, or as little as possible, input/output lag. Given that our system runs at 100 MHz and our ADC samples at 48 kHz, we have 2083 cycles to perform a transform at 100 MHz without missing any data. Given that we need $\frac{N}{2}\log_2 N$ butterflies, we can solve the equation:

$$\frac{N}{2}\log_2 N = 2083$$

which yields $N = 469$. This would imply that we could go as high as the closest lower power of 2, 256. However in this case $N = 128$ was chosen, so we get 2 cycles to perform each butterfly, as well as some extra slack in the control overhead.

After several trials, some problems arose with the core-wrapper construction. First of all, using this structure has an inherent flaw that it uses double the necessary memory, since the data from the FIFO is just read directly into the input memory of the FFT core. Secondly, the FFT core uses an unload pin unless the natural ordered output is requested. Since we do not want to use natural ordering, the core will start unloading its data as soon as it is ready without any control from the outside. This property made it very hard to time the output FIFO correctly, resulting in a very inefficient timing scheme. Finally, The Xilinx core needed to be used in unscaled mode, as we needed as high an accuracy as we could get. As every stage of the FFT uses an addition, each introduces an extra bit in its output, as does the final stage, resulting in $\log_2 128 + 1 = 8$ extra bits for a 128 input FFT, resulting in a 24 bit vector. The reverse transform, in turn, would scale the 24 bit input back to the 16 bit vector we started out with. However, the Xilinx FFT core does not take this reverse transform into account. We are thus required to use a different core for the reverse transform, using a 24 bit input, creating a 32 bit output we would then have to post-scale back to the 16 bit vector we started with in order to feed it back to the system.

Custom FFT Implementation Because of all these problems, it was decided to implement a custom version of the FFT algorithm, resulting in a design able to suit every specific need of this project. For the implementation of the Fast Fourier Transform the radix-2 DIT FFT discussed in the Section 2.3.1 is used. However, in its current form there is no practical way to implement it. Although this would yield a very fast implementation, $\log_2 N$ times the number of cycles required for 1 butterfly, it would also require $2N\log_2 N$ multipliers and $4N\log_2 N$ adders, as we require 4 multipliers and 2 adders for 1 complex multiplication and 2 adders for 1 complex addition. In order to keep the amount of resources a limited as possible, a structure with a single butterfly, 3 memory units, for the input, output and working memory, and some control logic in order to reuse the single butterfly for the entire tree is used. The proposed structure can be found in Figure 4.2. In this way we retain some of the efficient attributes of the FFT, while tuning it for minimal resource usage.

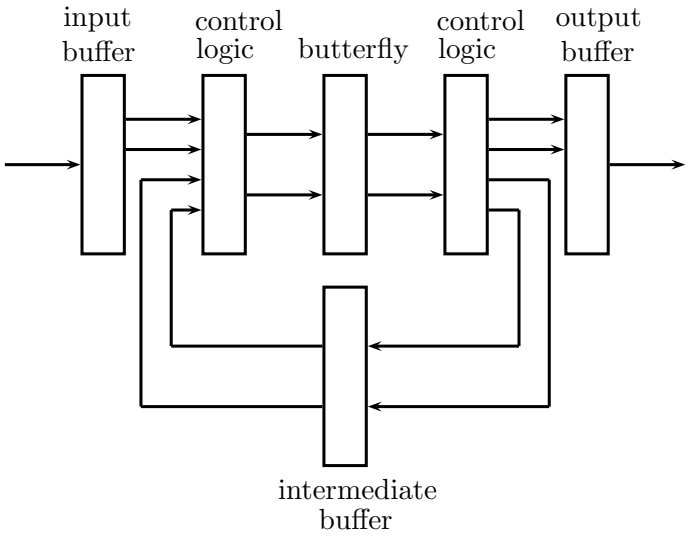


Figure 4.2: The implementation of the radix-2 DIT FFT algorithm

To ensure correct functionality, the design was first implemented in MATLAB. Since MATLAB has a built-in FFT function, testing the custom FFT structure was trivial, and any bugs it contained were eliminated efficiently. Despite the fact that translating the well-formulated FFT to VHDL was a challenge, results similar to the MATLAB implementation were soon achieved. The obtained values from the VHDL model, however, strayed somewhat from the MATLAB model, due to the fact that the VHDL butterfly uses (16 bit) fixed-point values where MATLAB uses double precision floating points. To counter this problem, the MATLAB model was redesigned to scale and truncate its values to simulate the 16 bit fixed point notation.

The first version of the VHDL FFT model used distributed RAM, meaning the memory values were stored in LUTs in the FPGA. Since 8 large memories are used, specified in Table 4.3, this took up a lot of space on the FPGA. The design was therefore redesigned for the use of BRAMs, of which the data sheet stated that it would introduce a single cycle of output lag. Keeping this in mind the FFT was redesigned, but the output results achieved with the previous version were not realized. Eventually, after having spent almost a month on this problem, it turned out that the requested values from the BRAM would arrive *just after* the next clock cycle, resulting in it being used on the *secondnext* clock cycle, destroying the entire functionality of the FFT. Once this problem was uncovered, the FFT was modified to give the BRAM access time another cycle of slack, thereby increasing its input/output lag.

In order to determine the input/output lag of the final FFT module, the system was simulated with Modelsim. Figure 4.3 shows the amount of time used to calculate the 128 point FFT. It can be seen that the calculation in fact takes longer than a single cycle of the 48 kHz clock. When measured one full FFT calculation

Table 4.3: The memory used by the FFT. (i)x(k) represents i bits and length k

	input memory	intermediate memory	output memory	twiddle factors
16x64	0	0	0	2
24x128	2	2	2	0

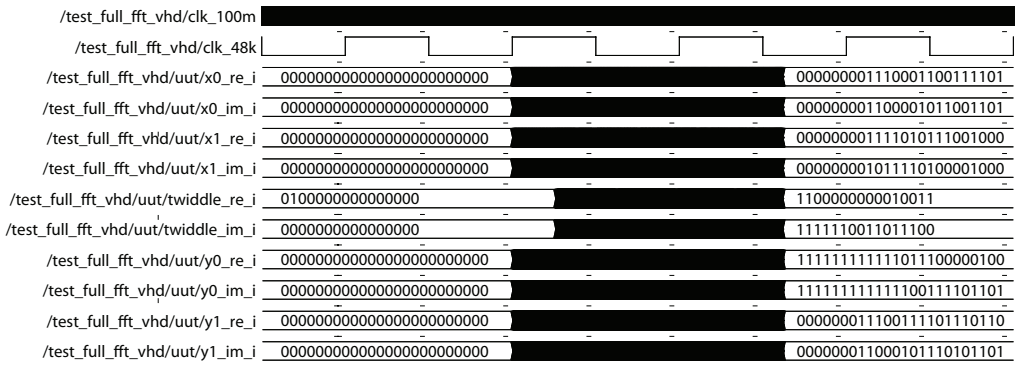


Figure 4.3: The FFT calculation time

takes 32640 ns, which in turn is equivalent to 3264 cycles of our system clock. This is more than our targeted 2083 cycles, so we lose 1 sample every 128 samples. Although this could be countered using a pipelined structure, the loss of one sample is so slight that the improvement in sound would not outweigh the effort needed to change the structure of the FFT.

4.3.2 Frequency-based Effects

As discussed in Section 3.3, the final system poses many constraints on the number of usable effects in the frequency domain. Nevertheless, several functions were implemented that are discussed in this section, although most of them are now implemented in the time domain. As with the time domain effects, implementing these modules took a lot of tweaking the settings before they produced the desired effect.

Filtering Frequency filtering is easy to implement when we have the FFT data readily available. As discussed in Section 3.4, the interface was extended with the sample number. Now that we know what sample, and therefore what frequency, we are operating on, we can, for instance in the case of a LPF, simply discard this data if its frequency is higher than the desired cut-off frequency, and pass through any other data without modifying it. This yields the desired behavior, but due to the smearing effect we also remove a part of the desired frequencies, as well as leave in part of the unwanted signal, resulting in a messy sound. Further contributing to a bad sound quality is the fact that a crisp cut-off is used instead of a gradual

cut-off. The contribution of the latter effect, however, is small for this inaccurate FFT, but will become more and more significant as a more accurate FFT is used.

Vibrato The vibrato effect was implemented in the time domain, for the mentioned reasons, as were the rest of the following effects. The attentive reader may have noticed that the implementation of the chorus effect explained earlier was not a pure chorus effect. As we can not really change the pitch continuously. As such, the chorus effect is now built-up out of 4 vibrato effects. The vibrato effect itself was implemented in a similar way. As yet there are no configurable parameters, as the modulation of the clock is quite a delicate process.

Wah The wah effect has not (yet) been implemented. One would need a specific time domain variable band-pass filter that is too much work to create at this time.

Phasing and Flanging The phaser and flanger were implemented using the knowledge that they are variable all-pass and comb filters, respectively. As with the vibrato, they are not currently tunable, as the clock modulation is a delicate process that is too easy to unbalance.

4.4 Support Modules

This section describes the implementation process regarding the modules that are not directly in the audio input/output datapath.

4.4.1 VGA Controller

In order to realize a clear interface to the device, as well as to visualize the effects of the modules on the sound, a VGA interface is required to output the data on a screen. A VGA interface operates based on several parameters listed in Table 4.4.

The basic operation is simple: at every clock pulse a pixel (R,G,B) is written and the horizontal counter is incremented. If both the horizontal and the vertical counters are within the active region the pixel is written to the screen. Once the horizontal counter is outside the active region the blanking signal is asserted and as a result the RGB values are overwritten with zeros. Once the horizontal front porch has passed the horizontal synchronization signal is asserted and once the horizontal back porch had expired the horizontal counter is reset to zero and the vertical counter is incremented. Once the vertical counter moves out of the active region the signal is once again blanked. Once the vertical front porch has passed the vertical synchronization is asserted, which, after the vertical back porch has passed, resets both the horizontal and the vertical counter, and the process starts over.

Implementing the VGA interface is a straightforward task if all parameters are readily available. Using the parameters from Table 4.5 [27], we can create a

Table 4.4: The VGA parameters explained

R,G,B	The amount of Red, Green or Blue used to define the current pixel. Values range from [0 : 255].
Front Porch	The number of pixels between the end of a line or column and the start of the synchronization.
Active	The number of pixels used to display all the pixel data in the line or column.
Back Porch	The number of pixels between the synchronization signal and the start of a new line or column.
Total	The total number of pixels in the line or column.
Synchronization	The signal used to mark the end of a line or column.
Blanking	The signal used to overwrite any data incoming with blank data (all zeros).

simple VGA output without the use for a very fast clock, by using a low resolution. We can obtain the desired 40 MHz frequency by placing a Digital Clock Manager (DCM) in front of the module, using the parameters in this table.

Table 4.5: The VGA parameters for a resolution of 800x600@60Hz

	Pixel clock	DCM settings		active	front porch	synch	back porch	total
	Mhz	M	D	pixels	pixels	pixels	pixels	pixels
Horizontal	40.00	4	10	800	40	128	88	1056
Vertical	40.00	4	10	600	1	4	23	628

Once the structure described in the previous paragraphs was implemented, the need for the creation of a process to generate pixel data in order to visualize the data arose. To start out, the top right quarter of the screen was selected to function as a character-written section. After that a 9x12 pixel font was defined, which used the respective hexadecimal keyboard codes to refer to the different letters in this font. Lines and columns were then defined, and as such, letters and punctuation can be displayed quite easily on this part of the screen. Through the use of a lookup function we can determine what the line sent to the screen should contain.

The next process draws axes on the screen and divides the incoming data up into frequency bands. These frequency bands are then displayed on the screen, both before the effects and after, in order to visualize the difference the effect has created.

4.4.2 Keyboard Interface

The standard AT keyboard interface is a serial interface, sending over packets of 11 bits shown in Figure 4.4. When in idle state, both the clock and the data line are high. When a key is pressed, the data stream is initiated with a start bit, which

is always a logical 0. After the start bit 8 data bits are transmitted, followed by a parity bit.

The keyboard uses odd parity, signifying that if the total number of ones transmitted is even, the parity bit will be one (the total number of ones *including the parity bit itself* will then be odd, hence the name), and zero otherwise. If the parity bit does not match the calculated parity, the data has an error and should be discarded. If, for some reason, there is more than one transmission error, the parity check scheme might yield a false positive, for example when two errors occur where a 1 is received as a 0 and another 0 is received as a 1. Despite this fact, a 1 bit error check should be enough for the connection of a keyboard. Should many errors occur, one should check the connection or replace the device. After the parity bit, a logical 1 follows as the stop bit.

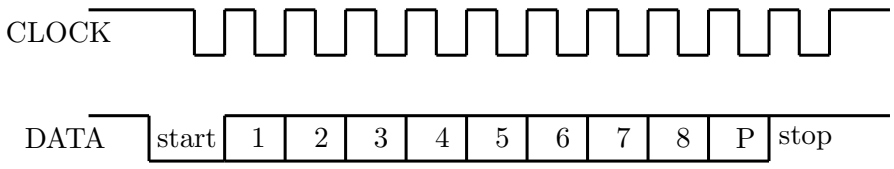


Figure 4.4: The layout of a PS2 keyboard packet

The keyboard sends its clock over one line and the serial data over another. The clock is generated by the keyboard itself, and is generally between 20 and 30 kHz. As can be seen in Figure 4.4, the keyboard data should always be read on the *negative* clock edge.

When the 8 bits from the keyboard have been read, they can be interpreted. Every key on the keyboard has a 'make' and a 'break' code. The make code signifies that a key has been pressed, while the break code indicates a key release. In most cases the break code of a key is the make code preceded by the hexadecimal number $F0$. Examples of several make and break codes are displayed in Table 4.6. The full table of scancodes is too large to display here, however it can easily be found on the Internet [28]. We can see here that there are several composite keys. These keys have a make code that is preceded by $E0$ and a break code that is preceded by $F0$, $E0$. The two keys that have a different control sequence are PRNT SCRN and PAUSE and are listed in the table in a special section. The Pause key is unique in that it has no break code, and is the only key that uses $E1$ in its make sequence.

The keyboard interface was designed and implemented in the fashion the theory suggests. 11 bits are read serially based on the keyboard clock. The parity check is done by using solely XOR gates in the following fashion:

$$(b0)\ XOR\ (b1)\ XOR\ (b2) \ldots XOR\ (b7)\ XOR\ P = C$$

If the parity check succeeds, C should be 1. If, at any point, the transmission has corrupted one of the bits, C will be 0, and the result should be discarded.

Once the 8 bit vector has been verified it is transferred to the 100 MHz clock domain and put on the output of the module for 3 100 MHz clock cycles. The

Table 4.6: Several examples of keyboard scancodes

Key	Make	Break
A	1C	F0, 1C
B	32	F0, 32
C	21	F0, 21
-	4E	F0, 4E
R CTRL	E0, 14	F0, E0, 14
R ALT	E0, 11	F0, E0, 11
PRNT SCRN	E0, 12, E0, 7C	E0, F0, 7C, E0, F0, 12
PAUSE	E1,14,77, E1,F0,14, F0,77	

system then goes into a wait state, awaiting the next keystroke or key release.

A wrapper is built around this structure to control the main device. The keyboard input is handled according to the state machine depicted in Figure 4.5. What happens is: When a key gets pressed the state machine will get data_valid = 1, and will therefore move to state 1. In state 1 the key data will be read and assumed to be a basic key. If at this point a different key is pressed before the first one is released, the machine will go back to state 0 and start over with the new key. As only single keys should be pressed to control the device, this should not be a problem. Now, assuming no additional keys have been pressed, the machine will wait in state 1 for the key to be released. Once the key is released the machine will pick up the break code and move to state 2. The process will hold here until the previous break code is done. Now if the same key identifier does not follow, another key has been pressed in the mean time, and we return to state 0. If we do find the valid key identifier, we move to state 3 where we output the found key, and we then wait for a data to become unready. Once the data is unready, we return to state 0.

The remainder of the wrapper checks the key identifiers coming out of the previously described process. If the keystroke matches one if the controls in use, the predefined control activity, such as toggling or incrementing a control signal, is executed and the process will wait for the next input.

4.5 Conclusion

This chapter described all the steps taken in implementing the the various modules that this design comprises. The system was implemented using Xilinx PlanAhead, for which it had to comply to several strict requirements. All requirements were met and the implementation of the system using a single partial reconfiguration region was successful.

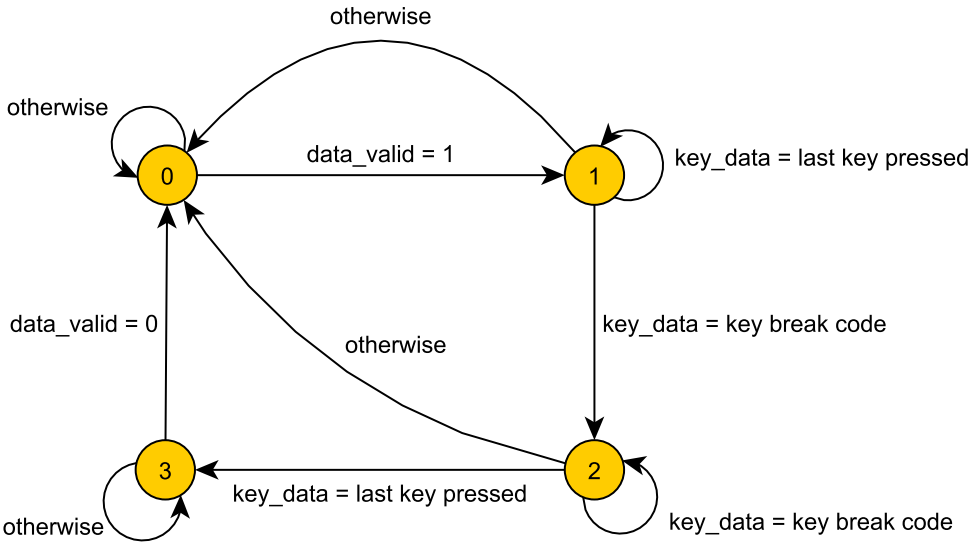


Figure 4.5: The keyboard state machine

Initially, FFT cores provided by Xilinx were used in this project. However, there were several reasons why this approach did not suffice. This resulted in the design and implementation of a custom FFT. Although this FFT is not able to finish all required processing within the single sample boundary, it is deemed sufficient for the current implementation.

Of all the proposed effects, there were several modules that were not implemented. The repeater module needs a buffer that is too large to realize on-chip, and the wah effect was left out because the variable band-pass filter proved to be too difficult to build in the time domain. Furthermore, all effects apart from the filtering module were implemented in the time domain, due to several unforeseen constraints discussed in earlier chapters. All other effects have been implemented successfully.

The effects that were initially intended to be implemented in the frequency domain are static, i.e. there are no control parameters available to change the effect, due to the fragile nature of the implementation used. The effects implemented in the time domain are customizable within certain bounds, either restricted by the hardware or manually restricted to operate within the bounds set by the theory.

5 Results

This chapter contains all the results of the system's final test phase. Please note that all data acquired in the implementation phase that was used to improve and modify the modules in this system is conveyed in Chapter 4, and will not be discussed further in this chapter.

First, the results regarding partial reconfiguration that were measured and deduced by implementing this project will be discussed in Section 5.1. Section 5.2 will then discuss the remaining test results from the FFT that were not directly used in the implementation phase. Consequently, Section 5.3 will discuss the test methods and results for the effect modules. The results regarding the support modules of this project are mentioned in Section 5.4, after which Section 5.5 will recapitulate the final results of this project.

5.1 Partial Reconfiguration

One of the goals of this project was to investigate the feasibility of partial reconfiguration. Throughout this thesis, the advantages and drawbacks of partial reconfiguration, whether theoretical, technology bound, considering the toolchain or otherwise, have been thoroughly discussed. Is this section the feasibility of partial reconfiguration in regard to this project will be discussed.

Table 5.1: The configuration time for the Virtex-II Pro (XC2VP30)

No. of Frames	Frame Length (bits)	Configuration Bits	Total no. of Bits (incl. header)	Download Time (ms)		
				SelectMap	Serial	JTAG
1,756	6,592	11,575,552	11,589,984	28.97	231.80	351.21

Table 5.1 [29], shows the size of the configuration file and the time it takes to configure the device via several interfaces. The most notable fact in this table is the difference between the configuration times for the various interfaces. The JTAG port is used to configure the device from a PC USB interface, while the SelectMap interface can be used to configure the device internally, either by using a Xilinx built-in function such as the PROM memory or the SystemACE Compact Flash device. We can verify the file size shown in the table ($11,589,984$ bits or $1,448,748$ bytes) with the size of the actual merged full bitfile generated by PlanAhead, being $1,448,817$, ignoring the small size variation.

If we now assume that the programming time scales linearly with the bitfile

size, we can deduct some indication of the configuration time of the partial reconfigurable units in this project. The largest partial bitfile in this project is 82,350 bytes, which coincidentally corresponds to exactly 100 frames, resulting in a scaling factor of about 17 compared to the full size FPGA. If we scale the configuration time accordingly, we end up with 1,7 ms for the SelectMap and 20 ms for the JTAG.

Unfortunately, a 20 ms delay, resulting in an audible silence, is quite resource expensive to hide completely. Since we are dealing with a direct input/output model, buffering can never continuously solve this problem. Although we could build a buffer that fills when the device is initially programmed, we can not refill this buffer when a partial reconfiguration is performed, and the delay will persist once the buffer is emptied, while initially adding an unwanted delay. For the SelectMap, the story is somewhat different. While 20 ms is a noticeable delay, roughly 2 ms should not be audible to the general listener. This adds another argument in favor of implementing this project as a self-contained platform, using the SystemACE Compact Flash interface to store and load the partial reconfigurable units.

In order to measure the reconfiguration time, we need to connect to the DONE signal of the FPGA. This signal, however, does not behave as the theory suggests. In fact, it stays in the high state throughout the reconfiguration. As such, we need a different method to measure the time it takes to write a different module to the partial reconfiguration region on the FPGA. We know that the output bus macros behave erratically throughout the reconfiguration. Since there is no direct use for the control outputs of the effect modules and they should remain low during normal operation, we can check these outputs for changes. As soon as one of the control outputs becomes high, a counter is started with a grace period of 1000 cycles, meaning the counter will continue while there are at most 1000 cycles between two high pulses. We can then read out this counter and relate it to a reconfiguration time. Although this is not an air tight method, it provides us with an indication of the reconfiguration time. The averaged measurement of this reconfiguration time resulted in a value of 27.8 ms. This value is conform the order of magnitude of the theoretical estimate of 20 ms.

This last paragraph is dedicated to an idea to solve the problem of hiding the reconfiguration time in this project, although this was not implemented in this project. According to the Xilinx Partial Reconfiguration User Guide [10], there should be a signal that is deasserted when configuration is in progress and asserted when finished. However, this guide does not provide any information on which signal or pin this is. The Virtex-II User Guide [29] mentions a 'done' signal that functions like this, that is bit 12 of the STAT register, but during this project it was not discovered how to acquire this signal ("[the STAT register] can be read using the JTAG or SelectMAP port for debugging purposes"). However, if one manages to read this signal, there could be another use for it other than using it to disable the Bus Macros. We could use this signal to enable and disable a 'shortcut', as shown in Figure 5.1, thereby bypassing the PRR during configuration time, and

ensuring a continuous sound output. In this figure, the done signal is used as an enable on the bus macros and as a select signal for the multiplexers. In the current version of the system, the counter mentioned in the last paragraph also functions as the bypass enable. Although this does not completely hide the reconfiguration process, it softens the distortion caused by reconfiguration.

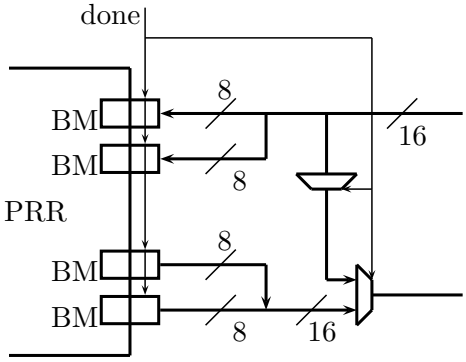


Figure 5.1: Creating a PRR bypass. Blocks marked 'BM' are Bus Macros

5.2 Fast Fourier Transform

There are several results related to the FFT that have not had a place in this thesis, the first of which is the impact of the fixed point strategy on the on the accuracy of the result. At every single one of the seven stages of the FFT, the calculated values from the multiplication are truncated. This leads to a worst case error of 0.999 repetitive being rounded down to 0. Since we then add two numbers, both carrying a worst case error of approximately 1, the worst case error per stage is 2. Chaining seven stages then yields a worst case error of 14 on one transform. The multiplication does not yield a multiplicative error, as any multiplication done within the FFT is done with a twiddle factor, which is always smaller then or equal to one. The same holds for the reverse transform.

Given that we do not know what is in between the forward and the reverse transform, we can only obtain a worst case error for both transforms. Adding all numbers, the worst case error is:

$$e_{worst\ case} = 2\ per\ stage\ *7\ stages\ *2\ transforms\ = 28$$

This is equivalent to $\frac{28}{2^{15}} * 100\% = 0.0854\%$ of the full dynamic range of the 16 bit two's complement (15 bits were used for the calculation as the error can occur with both positive and negative numbers). MATLAB tests of several real songs, scaled to full range, yield a maximum error of 23, indicating that the found worst case number of 28 is viable.

Table 5.2 presents the amount of resources of the device that are used by a single instance of the FFT module.

Table 5.2: FFT device utilization

Logic Utilization	Used	Available	Utilization
Number of Slices	622	13696	4%
Number of Slice Flip Flops	804	27392	2%
Number of 4 input LUTs	894	27392	3%
Number of BRAMs	6	136	4%
Number of MULT18X18s	8	136	5%

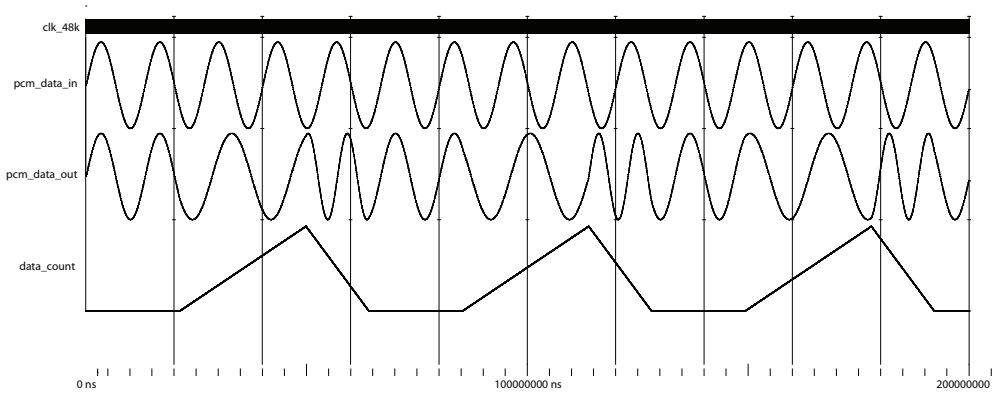


Figure 5.2: the output of the pitch modulation testbench

5.3 Effects

Since all but one of the effects were implemented in the time domain, They will be summed up here, ending with the filtering effect, being the only one left in the frequency domain. The effects that were not implemented will –of course– be left out. As all effect modules are capable of running at speeds greater than the system clock of 100 MHz, the maximum clock cycle statistics will be left out as they pose no constraint.

Pitch Modulation Several effects depend on the notion of pitch modulation. Figure 5.2 demonstrates the performance of the pitch modulation scheme explained in Chapter 4. The graph marked 'data_count' represents the number of values in the FIFO. When this graph shows an upwards ramp, the output of the FIFO is read back at a slower rate than the sample frequency, lowering the pitch. The downwards ramp results in an increase in the pitch, after which the FIFO is empty. The cycle then restarts.

Delay Being one of the easier effects to implement, this effect achieves a good performance without introducing any unwanted effects. The downside to this effect is that it requires 2 out of 8 full columns of BRAMs in order to buffer 500 ms of audio for a single channel. Although this amount is available, using 2 channels

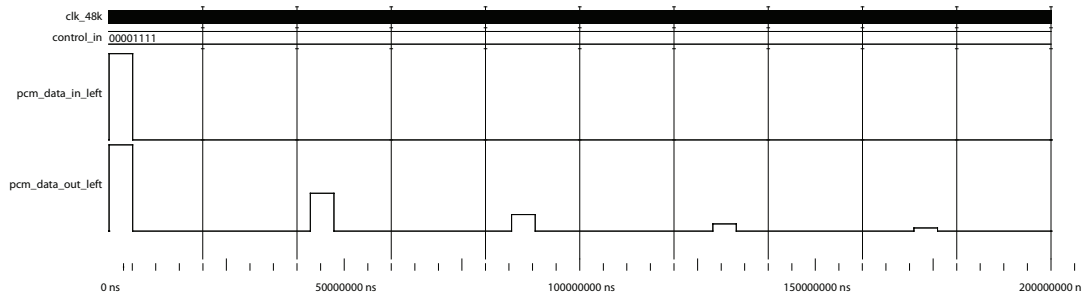


Figure 5.3: the output of the delay testbench

would exceed the available memory. If at some point in the future dual channel audio is required, the buffers should be moved to the DDR RAM.

Figure 5.3 shows the output of the testbench of the delay effect. The test bench excites the input signal for several clock cycles, after which the input is reset to zero. Both the time delay (43 seconds in this test) and the gradual decay are clearly visible in the output.

The number of device resources used by the module are presented in Table 5.3.

Table 5.3: Delay device utilization

Logic Utilization	Used	Available	Utilization
Number of Slices	187	13696	1%
Number of Slice Flip Flops	193	27392	0%
Number of 4 input LUTs	293	27392	2%
Number of BRAMs	30	136	22%
Number of MULT18X18s	1	136	0%

Chorus The behavior of the chorus testbench is demonstrated in Figure 5.4. We see the three different phase modulators and the sum of all signals in the output.

As mentioned before, the chorus effect is not as easy to implement, because it requires a continuously lowered or raised pitch, which we can not achieve. As such, this effect does not perform as well as we would like it to. Although the effect is still audible, it is hard to recognize as a chorus effect.

Table 5.4 shows the device utilization of the chorus module.

Table 5.4: Chorus device utilization

Logic Utilization	Used	Available	Utilization
Number of Slices	493	13696	3%
Number of Slice Flip Flops	524	27392	1%
Number of 4 input LUTs	852	27392	3%
Number of BRAMs	3	136	2%

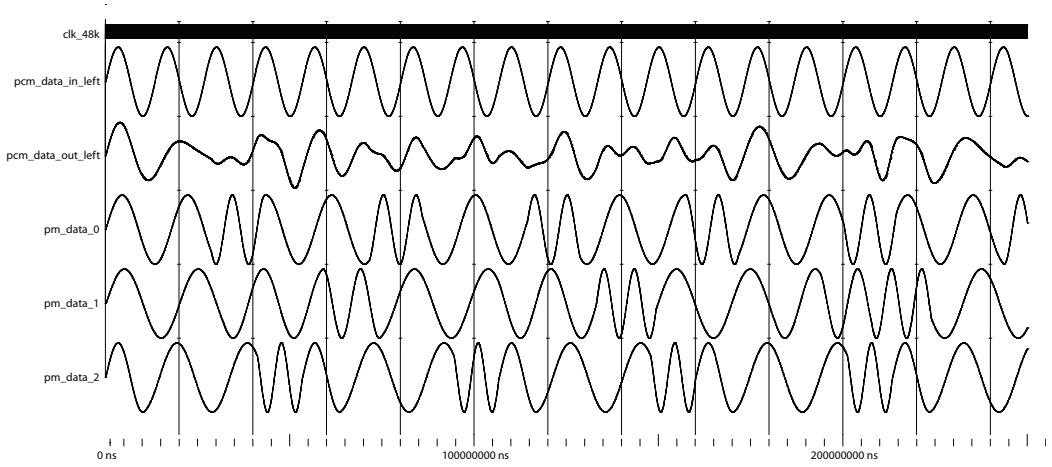


Figure 5.4: the output of the chorus testbench

Reverb The implementation of the Schroeder reverberator is quite complex. As such, Figure 5.5 shows several intermediate signals besides the input and output. The signals labeled `pm_data_[0:3]` are the outputs of the 4 comb filters, `combed` is the composite comb filter output and `allpass_in` is the rescaled version of combed. `trimmed` is then the output of the first all-pass filter and `allpassed` the output of the second all-pass filter. The output is a mixed signal of the input and the `allpassed` signal.

Table 5.5 displays the FPGA usage results of the implementation.

Table 5.5: Reverb device utilization

Logic Utilization	Used	Available	Utilization
Number of Slices	627	13696	4%
Number of Slice Flip Flops	914	27392	3%
Number of 4 input LUTs	856	27392	3%
Number of BRAMs	24	136	17%
Number of MULT18X18s	3	136	2%

The reverb effect is quite hard to control correctly. As we need to mix several signals together, we get an averaged volume for the composite signal, which can differ quite strongly from the original volume. We can see in the figure that after the direct sound, we get a small composite sound that is slightly delayed. This is the effect of reverberation.

Volume Control Adjusting the volume is one of the easier effects, and as such functions very well. Considering everybody would know what a change in volume sounds like, there is little more to say here.

Figure 5.6 shows the testbench behavior of the volume controller. We can see

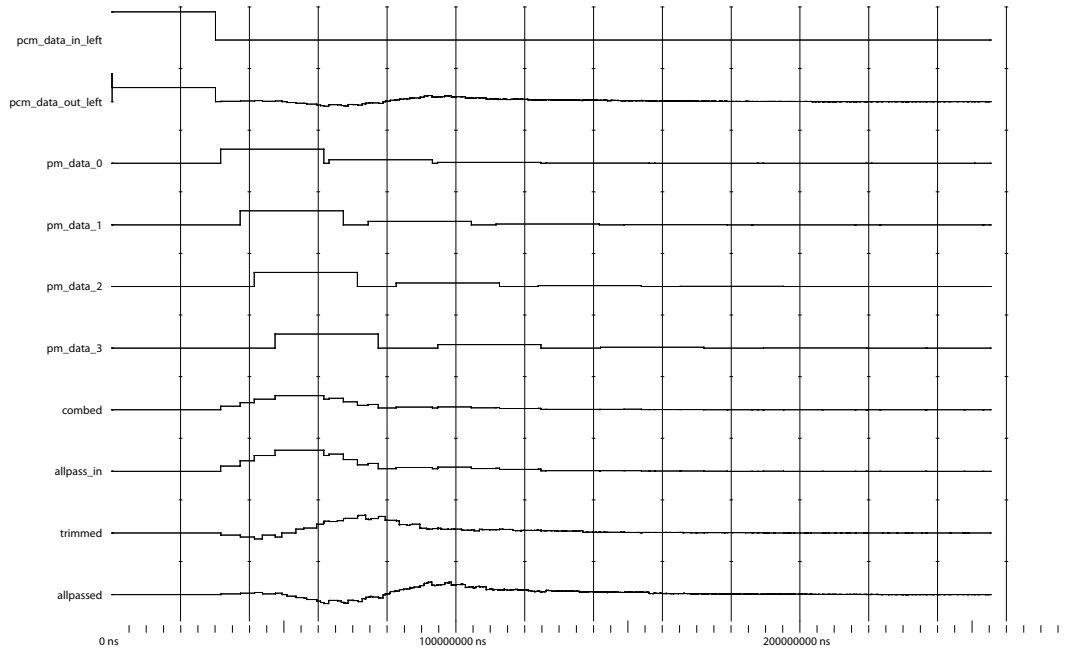


Figure 5.5: the output of the reverb testbench

in the first have of the figure that when we try to amplify a signal that already has a large amplitude that the module shows erratic behavior. The second half shows the attenuation property.

In Table 5.6 the device usage statistics are presented.

Table 5.6: Volume controller device utilization

Logic Utilization	Used	Available	Utilization
Number of Slices	16	13696	0%
Number of Slice Flip Flops	16	27392	0%
Number of 4 input LUTs	33	27392	0%
Number of MULT18X18s	2	136	1%

Tremolo The testbench result of the tremolo effect can be seen in Figure 5.7. The waveform clearly shows the sine wave superposed on the input constant. When the value of the input changes, the amplitude of the sine wave adjusts accordingly. The full period of the sine corresponds to approximately 70 ms, which translates to the intended 15 Hz tremolo.

The amount of device resources used by this module are shown in Table 5.7.

By creating an obvious vibrating effect, the tremolo is one of the most notable and successful effects in this library.

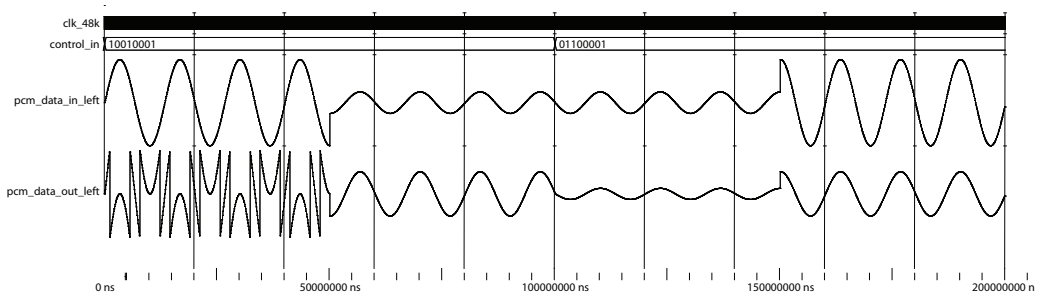


Figure 5.6: the output of the volume control testbench

Table 5.7: Tremolo device utilization

Logic Utilization	Used	Available	Utilization
Number of Slices	167	13696	1%
Number of Slice Flip Flops	130	27392	0%
Number of 4 input LUTs	315	27392	1%
Number of MULT18X18s	2	136	1%

Distortion The distortion effect may be the easiest effect in this series to identify. Most people recognize the sound as belonging to any form of rock music, in which is finds an extensive use. The drawback if this effect is that when used on multiple instruments at once, such as a finalized song, distortion will start to sound as an unpleasant static effect. Heavier settings of the distortion effect are best when combined with an amplification effect, as a low boundary will diminish the average amplitude of the output.

The output of the testbench of the distortion effect is visualized in Figure 5.8. In the second half of the plot the threshold is lowered and we see that the low amplitude signal which remained unchanged before is now clipped as well.

Table 5.8 shows the number of device resources used by this module.

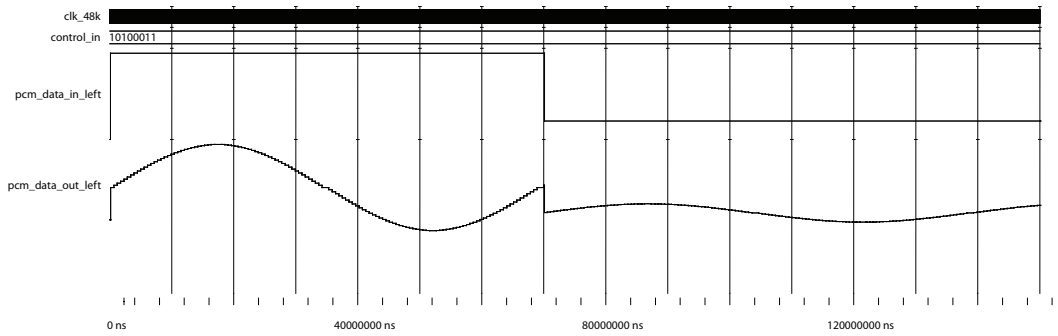


Figure 5.7: the output of the tremolo testbench

Table 5.8: Distortion device utilization

Logic Utilization	Used	Available	Utilization
Number of Slices	80	13696	0%
Number of Slice Flip Flops	32	27392	0%
Number of 4 input LUTs	161	27392	0%

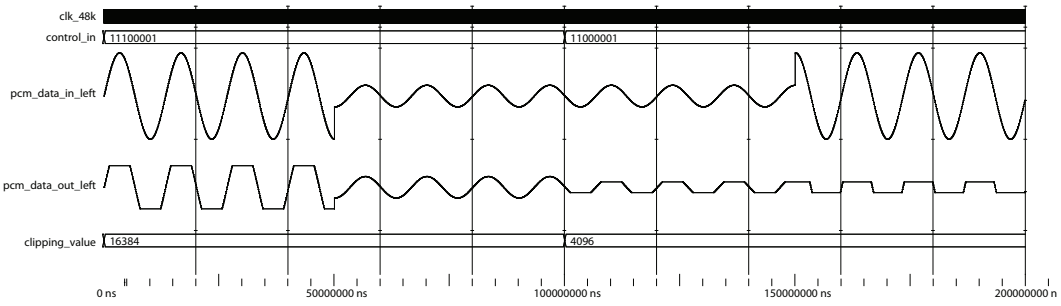


Figure 5.8: the output of the distortion testbench

Compression/Expansion Compression has a soothing effect on any sound, diminishing loud components, thereby making room for the softer noises to be heard clearly. Expansion, on the other hand, increases the gap between soft and loud sections, creating a sense of a screaming person in a room of whispers. Both effects function correctly, although expansion creates an imbalance in the sound that is unpleasant to hear in its current form.

Figure 5.9 demonstrates the functionality of the compressor/expander. The first half of the graph the compression effect is selected. We can see that the effect turns on after 21 ms of high amplitude sound, which is equivalent to 1023 samples at 48 kHz. Once the amplitude of the input signal is lower than the threshold for 1023 samples, the effect turns off. The second half of the plot shows the expansion behavior, exhibiting the exact opposite effect: diminishing the low amplitude sound, while leaving the high amplitude sound intact.

The device utilization values are shown in Table 5.9.

Table 5.9: Compression / Expansion device utilization

Logic Utilization	Used	Available	Utilization
Number of Slices	100	13696	0%
Number of Slice Flip Flops	97	27392	0%
Number of 4 input LUTs	191	27392	0%
Number of MULT18X18s	1	136	0%

Octaver As mentioned in the theory, the octaver introduces a distorting effect apart from its intended use. Knowing in advance that this effect will occur, this

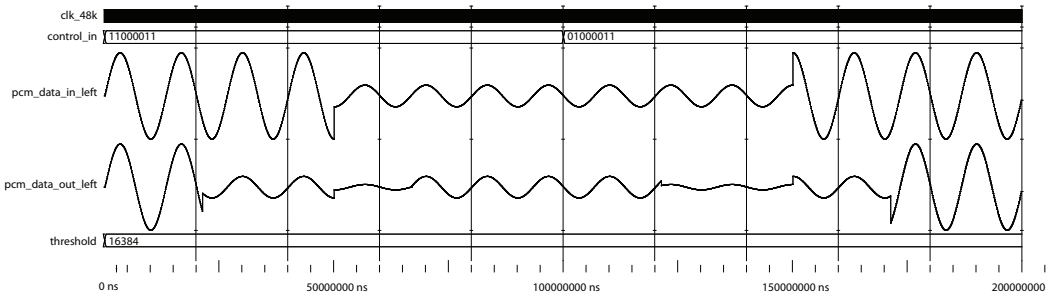


Figure 5.9: the output of the compression/expansion testbench

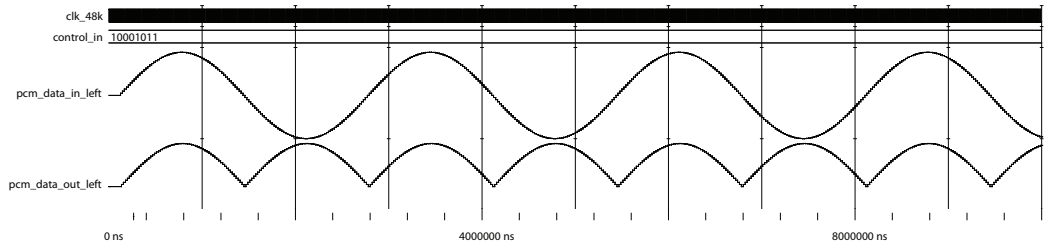


Figure 5.10: the output of the octaver testbench

effect functions very well, as is to be expected from such a straightforward implementation. As with the distortion effect, this effect is best used on a single instrument source.

The behavior of the module, depicted in Figure 5.10, is clear and correct. In order to best visualize the functionality, this testbench uses a sine input.

Table 5.10 presents the device resource utilization.

Table 5.10: Octaver device utilization

Logic Utilization	Used	Available	Utilization
Number of Slices	25	13696	0%
Number of Slice Flip Flops	32	27392	0%
Number of 4 input LUTs	40	27392	0%

Vibrato Although the implementation of this effect is quite successful, it is very hard to distinguish from the much easier to implement more tremolo effect.

The amount of device resources used by this module are shown in Table 5.11.

Phasing and Flanging Since these effects were constructed with an experimental approach, they do not function as well as we would want them to. Although the effects are in fact distinguishable, further tweaking will be required to bring out the full potential of this approach.

Table 5.11: Vibrato device utilization

Logic Utilization	Used	Available	Utilization
Number of Slices	162	13696	1%
Number of Slice Flip Flops	162	27392	0%
Number of 4 input LUTs	173	27392	0%
Number of BRAMs	1	136	0%

Filtering The filtering effect is functional for every setting. However, due to the smearing effect explained in the theory, it introduces quite a lot of static. Where the low-pass setting negates most of this static, the band-pass and especially the high-pass filters are susceptible to this effect and are as such of less use. The effects are, however, still quite usable to demonstrate the relation between the sounds and the visual output of the FFT.

Table 5.12 shows the FPGA resource usage of the filtering module.

Table 5.12: Filter device utilization

Logic Utilization	Used	Available	Utilization
Number of Slices	70	13696	0%
Number of Slice Flip Flops	55	27392	0%
Number of 4 input LUTs	110	27392	0%

Minimum PRR size Is this section, all device resource usage statistics were presented. As all modules are mandated to fit within the PRR, the final size of the PRR should exceed the values stated in Table 5.13. Since we already need to use 22% of the available BRAM units, the remainder of the constraints are of little importance, as spanning enough columns to facilitate all the BRAMs automatically provides enough of the other resources.

Table 5.13: Minimum PRR size

Logic Utilization	Used	Available	Utilization
Number of Slices	627	13696	4%
Number of Slice Flip Flops	914	27392	3%
Number of 4 input LUTs	856	27392	3%
Number of BRAMs	30	136	22%
Number of MULT18X18s	3	136	2%

5.4 Support Modules

The statement that holds for the sound effects largely holds for the video output and the keyboard input as well. As most modules in this project communicate

with outside peripherals, it is quite difficult to obtain data that can be visualized easily, and the data that can be visualized is hard to interpret. As such, only the verification procedure and its results for both modules will be described here.

VGA interface Although it is quite time-consuming to verify the VGA interface in this fashion, this interface was verified by synthesizing the VHDL and programming it to the board. The big advantage of this method is that once programmed, one can see immediately if the made changes had the desired effect. The first implementation, resulting in a functioning blank screen, was working correctly at the first attempt, making it unnecessary to apply any other verification method, as any further work was comprised of displaying information on that blank screen.

The device usage of the VGA module is presented in Table 5.14. We can see that the VGA interface unit is very large. This is due to the fact that many conditions to write to the correct section of the screen are required. Furthermore, an extra column has been added to the table, as this module does not comply to the 100 MHz system clock frequency. As such, this entire module runs on the 40 MHz pixel clock that is used to generate the correct frame rate for the VGA display.

Table 5.14: VGA module device resource usage

Logic Utilization	Used	Available	Utilization
Number of Slices	1532	13696	11%
Number of Slice Flip Flops	693	27392	2%
Number of 4 input LUTs	2828	27392	10%
Number of BRAMs	1	136	0%
Number of MULT18X18s	4	136	2%
Maximum Clock Frequency			84.729 MHz

Keyboard Interface The verification method that was used to test the keyboard interface was similar to the one used for the VGA interface. Once the VHDL file was synthesizing correctly, the FPGA was programmed with this core and the on-board LEDS were used to display the state of the module. From this state one can derive any errors present in the keyboard interface, and the corresponding section of code could be analyzed further for errors. Any errors present were eliminated using this method.

Table 5.15 presents the amount of resources in the device used by the keyboard interface.

5.5 Conclusion

The baseline of this project was to create a device not only *capable* of partial reconfiguration, but able to effectively make use of this strategy. When researching

Table 5.15: VGA module device resource usage

Logic Utilization	Used	Available	Utilization
Number of Slices	210	13696	1%
Number of Slice Flip Flops	112	27392	0%
Number of 4 input LUTs	394	27392	1%

the effectiveness of partial reconfiguration in this project, it was discovered that in the context of sound, the delay of a full reconfiguration of the device would produce a disturbing silence, while the configuration of a partial module, that takes in the order of ten times less time, only causes a small disturbance. In this case, The SelectMap device, that handles internal reprogramming, again performs about ten times better still, making the effect of a partial reconfiguration completely inaudible in contrast to the JTAG interface. The reconfiguration time was measured and amounted to 27.8 ms, confirming the theoretical estimate.

Additionally, a strategy where the partial reconfiguration delay could be masked completely, by bypassing the module that is to be reconfigured at configuration time was proposed. Although this has not yet been implemented due to some complications, this could be a feasible way of masking the configuration delay.

Analyzing the FFT, it became clear that the chosen fixed point strategy introduces a maximum error of one tenth of a percent, making it a viable and accurate implementation.

When discussing the impact on the sound of all the effects, it became clear that most effects function correctly, although a few effects are difficult to identify or tell apart. Several modules were implemented with an experimental technique, making them unable to reach their full potential at this point. When this technique is fine-tuned correctly, it has the potential to add a new dimension to the available effects in this project.

6 Conclusions

This chapter will review everything that was done in the course of this project. Section 6.1 will review all conclusions, and is divided into the different chapters. Section 6.2 will summarize the main contributions of this project to the field of engineering. Finally, Section 6.3 will discuss any future work that can be performed based on the current state of the project.

6.1 Summary

In Chapter 2, all the information required to understand the work done during this project, and the processes and functions described in this thesis, is contained.

A short history of reconfigurable computing shows us that FPGAs populate a region in hardware in between the fast but rigid ASIC and the slow but flexible GPP. Rather than excelling at either speed or flexibility only, the FPGA can be designed to fit a large region of this trade-off, sacrificing speed for flexibility or vice versa. Along with the description of how an FPGA works and how it is (re)configured, the notion of partial reconfigurability is introduced, referring to a technique where only a small portion of the FPGA is configured, leaving the remainder intact. Partial reconfiguration can be divided into static and dynamic partial reconfiguration, where the former means the FPGA does not operate while it is partially reconfigured, whereas the with the latter strategy the remainder of the FPGA will continue to function whilst partially reconfiguring.

We continue by discussing several pieces of work directly related to this project. Despite the fact that there are many papers regarding partial reconfiguration, few hold a direct relation to this project. As such, several colleagues that have compiled other related work were cited, in order to form a complete picture of this field of work. Furthermore, there have been many publications by Xilinx that are relevant to what this thesis is proposing to do, as this project is implemented using their product, and therefore the majority of the citations have direct tie-ins with this company.

Apart from taking in a sounds signal and sending out a sounds signal, this project incorporates additional input and output. The input is in the form of a keyboard, allowing us to control the processes existing within the FPGA. The additional output is a visual output to a screen, displaying the sounds time or frequency spectrum, enabling us to visually analyse the sound and the difference between the unaltered and the altered sound.

As this project is for the most part built up out of custom made blocks, a large section of this chapter is dedicated to explaining the workings of these modules.

The most notable part covers the functionality of the Fourier Transform and its derivatives. The Fourier Transform is used to derive the relative frequencies from a signal. In the case of sound, the extraction of the frequency pattern provides us with a good insight into the frequency composition of the sounds, and enables us to extract certain frequencies from the signal, in order to process them separately. The remainder of this chapter handles the functionality contained within the effect modules, designed to alter the sound stream running through the system.

In Chapter 3, the design steps taken to form the final version of this project, based on the project goals, were discussed. The goal of this project is to create an audio manipulation platform using partial reconfiguration. In order to best visualize the effect a system was designed that has both an audio output and a video output. The initial system is built up out of several partial reconfigurable time domain effects, followed by a frequency transform, several partial reconfigurable frequency domain effects and a transform back to the time domain. Apart from this main path, video data is gathered from several points in this path, feeding information to the VGA output.

Due to several unforeseen constraints, the initial system was reduced to a single partial reconfigurable region, only able to reconfigure a single module partially, leaving the rest fixed. Furthermore, multiple effect previously scheduled for the frequency domain were moved to the time domain or left out completely.

In order to create the possibility for partial reconfiguration, a special interface was defined. At a later stage of the project, this standard was divided into a time domain standard and a separate frequency domain standard.

The possibility for implementing this project in the MOLEN polymorphic processor was investigated. Using the technology this project is based on, however, does not seem to be a feasible approach to accomplish this. First, the implementation of the MOLEN processor would have to be adapted to the required version of Xilinx ISE 9.3*i*PR8, which is a time consuming project. Once this is complete, the MOLEN implementation would have to be redesigned in order to fit the constraints posed by Xilinx PlanAhead.

Chapter 4 described all the steps taken in implementing the the various modules that this design comprises. The system was implemented using Xilinx PlanAhead, for which it had to comply to several strict requirements. All requirements were met and the implementation of the system using a single partial reconfiguration region was successful.

Initially, FFT cores provided by Xilinx were used in this project. However, there were several reasons why this approach did not suffice. This resulted in the design and implementation of a custom FFT. Although this FFT is not able to finish all required processing within the single sample boundary, it is deemed sufficient for the current implementation.

Of all the proposed effects, there were several modules that were not implemented. The repeater module needs a buffer that is too large to realize on-chip, and the wah effect was left out because the variable band-pass filter proved to be too difficult to build in the time domain. Furthermore, all effects apart from the

filtering module were implemented in the time domain, due to several unforeseen constraints discussed in earlier chapters. All other effects have been implemented successfully.

The effects that were initially intended to be implemented in the frequency domain are static, i.e. there are no control parameters available to change the effect, due to the fragile nature of the implementation used. The effects implemented in the time domain are customizable within certain bounds, either restricted by the hardware or manually restricted to operate within the bounds set by the theory.

Chapter 5 discussed the results regarding the implementation of the project. The baseline of this project was to create a device not only *capable* of partial reconfiguration, but able to effectively make use of this strategy. When researching the effectiveness of partial reconfiguration in this project, it was discovered that in the context of sound, the delay of a full reconfiguration of the device would produce a disturbing silence, while the configuration of a partial module, that takes in the order of ten times less time, only causes a small disturbance. In this case, The SelectMap device, that handles internal reprogramming, again performs about ten times better still, making the effect of a partial reconfiguration completely inaudible in contrast to the JTAG interface. The reconfiguration time was measured and amounted to 27.8 ms, confirming the theoretical estimate.

Additionally, a strategy where the partial reconfiguration delay could be masked completely, by bypassing the module that is to be reconfigured at configuration time was proposed. Although this has not yet been implemented due to some complications, this could be a feasible way of masking the configuration delay.

Analyzing the FFT, it became clear that the chosen fixed point strategy introduces a maximum error of one tenth of a percent, making it a viable and accurate implementation.

When discussing the impact on the sound of all the effects, it became clear that most effects function correctly, although a few effects are difficult to identify or tell apart. Several modules were implemented with an experimental technique, making them unable to reach their full potential at this point. When this technique is fine-tuned correctly, it has the potential to add a new dimension to the available effects in this project.

6.2 Main Contributions

Goals In the introduction of this thesis, the main goal of this project was formulated as follows:

- To design and implement an audio processor making use of the partial reconfiguration technique.

Additionally, several secondary goals were defined:

1. To research partial reconfiguration and select the most feasible implementation to incorporate in this project.

2. To design and implement an audio processor making use of the selected partial reconfiguration technique.

3. The to be designed system is to feature a clear user interface as well as be as autonomous as possible.

4. To investigate the possibility of implementing this project on the available implementation of the MOLEN polymorphic processor.

These goals will be revisited in this section, discussing them one by one.

ad 1. At the start of this project, extensive research into partial reconfiguration was done in order to select the suitable candidate to include in the implementation. Although there are many publications regarding partial reconfiguration, there are few working, ready to use ways of implementation. As such, the method used by Xilinx was used, deeming this the most feasible approach.

ad 2. The design and implementation was successful. Additionally, partial reconfiguration was proven to be a sensible addition to this implementation.

ad 3. The implementation features a clear interface for both output and control. At this stage, however, the configuration of the system is done using a computer. An on-board configuration solution should be added in order to achieve full autonomy.

ad 4. The possibility to incorporate this project in MOLEN was studied, but deemed impossible, at least in the scope of this project.

Main Contributions The main contributions of this project are the following:

1. **Partial Reconfiguration** This project has added to the field of dynamic partial reconfiguration by summarizing the current status of the subject as well as by realizing a large proof-of-concept.

2. **MOLEN** By investigating the possibilities of including an approach of partial reconfiguration in the MOLEN polymorphic processor, this project has contributed knowledge to the MOLEN project.

3. **Audio platform framework** By designing and implementing this audio processor, a framework was created, upon which new audio effects can be easily implemented. Additionally, this framework has been clearly structured and designed, facilitating easy extensibility and enhancement.

4. **Demonstration platform** Because of the appeal of a sound generating and manipulation platform, this project can be used to demonstrate state of the art technology in a way that can be grasped by the general interested public.

5. **Integration in other projects** This proof-of-concept implementation was used by another project to realize and demonstrate remote partial reconfiguration over IP.

6.3 Future Work

There are several recommendation for someone who decides to improve this project. The recommendations are divided into several sections of the project.

Partial Reconfiguration The partial reconfigurability of this project can be improved in several ways:

- *Investigate and implement a partial reconfiguration approach different from the approach used by Xilinx PlanAhead.* This could improve the project by opening up multiple regions for partial reconfiguration or remove restrictions posed by the PlanAhead approach.
- *Re-implement this project with an improved version of PlanAhead.* At the time of implementation, PlanAhead 10.1.8 has several bugs preventing full implementation of the various features, among which is using multiple regions for partial reconfiguration. Future versions could have these bugs removed, opening up new features to improve this project.
- *Make the platform independent.* There are several option to achieve a platform that is not dependent on a computer for configuration. A practical suggestion is implementing a module that communicates with a mass storage device, such as the Compact Flash, through the SelectMap interface. This would not only remove the need for external reconfiguration, but also speed up the reconfiguration.
- *Implement the partial reconfiguration region bypass.* Adding a bypass for a PRR will guarantee a continuous sound output, negating the reconfiguration time.

Fourier Transform

- *Improve the speed of the FFT implementation.* At this time, the FFT implementation is quite straightforward. No optimisations have been performed and there are redundant waiting cycles to guarantee correct processing of the samples. Improving either or both of these point could result in the FFT being able to complete within one cycle of the sample clock.
- *Redesign the FFT by splitting it into several sub-FFTs.* By performing a split of the FFT, the local resolution can be improved, and several effects can be improved as a result.

Effects

- *Build additional effects.* This platform provides an easy, well defined structure for building, testing and tuning effects. As long as the added effect complies

with the set rules and template, there are no limitations other than an upper bound on the usable physical area.

- *Improve the current effects.* Although the existing effects were designed with care, there is always room for improvement. Several modules have yet to reach their full potential, for instance by opening up additional settings.
- *Implement frequency estimation.* Many effects can benefit from knowing the exact frequency of their input. Through the use of frequency estimation, this exact frequency can be approximated, providing the effects room for improvement, as well as open up opportunities for new effects to be created.

Bibliography

[1] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The Molen Polymorphic Processor," *IEEE Transactions on Computers*, pp. 1363–1375, November 2004.

[2] V. Betz and J. Rose, "Cluster-Based Logic Blocks for FPGAs: Area-Efficiency vs. Input Sharing and Size," in *IEEE Custom Integrated Circuits Conference*, pp. 551–554, May 1997.

[3] Xilinx, *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*, November 2007.

[4] Atmel, *FPSLIC on-chip Partial Reconfiguration of the Embedded AT40K FPGA*, 2002.

[5] M. Danek, P. Honzk, J. Kadlec, R. Matousek, and Z. Pohl, "Reconfigurable System on a Programmable Chip Platform," *ATMEL Applications Journal*, vol. , p. , 2005.

[6] Xilinx, "Development System Reference Guide," 2005.

[7] Xilinx, *Virtex Series Configuration Architecture User Guide*, September 2000.

[8] D. Mesquita, F. Moraes, J. Palma, L. Moller, and N. Calazans, "Remote and partial reconfiguration of FPGAs: tools and trends," in *Proceedings of IPDSP*, p. , April 2003.

[9] N. Dorairaj, E. Shiflet, and M. Goosman, "PlanAhead Software as a Platform for Partial Reconfiguration," *XCell Journal*, vol. , pp. 68–71, Fourth Quarter 2005.

[10] Xilinx, *Early Access Partial Reconfiguration User Guide*, September 2008.

[11] Xilinx and Brian Jackson, *Partial Reconfiguration Design With PlanAhead*, March 2008.

[12] Xilinx, *Development System Reference Guide*, 2005.

[13] C. Bobda and D. Murr, "Partial Reconfiguration Design," in *Introduction to Reconfigurable Computing*, pp. 213–258, Springer Netherlands, 2007.

[14] I. Page, "Closing the gap between hardware and software: hardware-software cosynthesis at Oxford," in *Hardware-Software Cosynthesis for Reconfigurable Systems (Digest No: 1996/036), IEE Colloquium on*, pp. 2/1–211, Feb 1996.

[15] A. Upegui and E. Sanchez, "Evolving hardware by dynamically reconfiguring Xilinx FPGAs," in *Lecture Notes in Computer Science*, pp. 56–65, Springer Berlin / Heidelberg, 2005.

[16] E. McDonald, "Runtime FPGA Partial Reconfiguration," vol. , pp. 1–7, March 2008.

[17] B. Blodget, P. James-Roxby, E. Keller, S. McMillan, and P. Sundararajan, "A Self-reconfiguring Platform," in *Field-Programmable Logic and Applications*, pp. 565–574, Springer Berlin / Heidelberg, 2003.

[18] H. Nyquist, "Certain topics in telegraph transmission theory," *Proceedings of the IEEE*, vol. 90, pp. 280–305, February 2002.

[19] C. Shannon, "Communication In The Presence Of Noise," *Proceedings of the IEEE*, vol. 86, pp. 447–457, February 1998.

[20] S. Lehman, "http://www.harmony-central.com/Effects/." website, .

[21] E. Doering, "http://cnx.org/content/m15491/latest/." website, 2008.

[22] B. Girod *et al.*, *Signals and Systems*, pp. 195–236. Wiley, 2001.

[23] S. Bernsee, "http://www.dspdimension.com/." website, 1999.

[24] S. Raaijmakers and S. Wong, "Run-Time Partial Reconfiguration for Removal, Placement and Routing on the Virtex-II Pro," *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, vol. , pp. 679–683, August 2007.

[25] Xilinx, "http://tinyurl.com/V2P-demo." website, .

[26] Xilinx, *Data Sheet: LogiCore Fast Fourier Transform v5.0*, October 2007.

[27] Xilinx, *Xilinx University Program Virtex-II Pro Development System: Hardware Reference manual*, March 2005.

[28] A. Chapweske, "http://tinyurl.com/scancodes." website, .

[29] Xilinx, *Virtex-II Pro Platform FPGA User Guide*, June 2003.

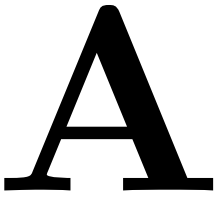
[30] J. A. Lee, *Colours of love: an exploration of the ways of loving.* New York: New Press, 1973.

[31] Xilinx, "http://tinyurl.com/xilinx-early-access." website, .

List of Abbreviations

ADC	Analog to Digital Converter
ASIC	Application-Specific Integrated Circuits
BLE	Basic Logic Element
BPF	Band Pass Filter
BRAM	Block Random Access Memory
BSF	Band Stop Filter
CLB	Configurable Logic Block
DAC	Digital to Analog Converter
DCM	Digital Clock Manager
DFT	Discrete Fourier Transform
DIF	Decimation-In-Frequency
DIT	Decimation-In-Time
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
GCLK	Global CLocK
GPP	General Purpose Processor
HPF	High Pass Filter
IDFT	Inverse Discrete Fourier Transform
IFFT	Inverse Fast Fourier Transform
IOB	Input/Output Block
IOI	Input/Output Interconnect
LFO	Low Frequency Oscillator
LPF	Low Pass Filter
LUT	Lookup Table
PRAGMA	a Partially Reconfigurable Audio Generation and Manipulation Application
PRR	Partial Reconfiguration Region
PRU	Partial Reconfigurable Unit
SNR	Signal to Noise Ratio
SRAM	Static Random Access Memory
STDFT	Short-Time Discrete Fourier Transform
XPART	Xilinx Partial Reconfiguration Toolkit

About The Title



PRAGMA stands for 'a Partially Reconfigurable Audio Generation and Manipulation Application'. This title, however, has a direct meaning in the English language that has a direct reflection on how I view this project. John Lee describes PRAGMA as one form of love. More specifically: "Pragma : love that is driven by the head, not the heart; undemonstrative"[30]. I think this quite strikingly describes my view towards this project, and engineering in general. Although I love engineering, it always has been, and always will be a rational, realistic and controllable love, driven by a desire for knowledge.

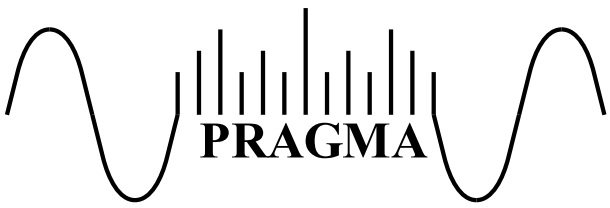


Figure A.1: The PRAGMA logo

Implementation in PlanAhead B

In order to successfully create a partial-reconfigurable design, many strict rules have to be followed. First of all, only very specific versions of the Xilinx toolkit can be used. At the time of writing this is ISE 9.2*i*04PR8 where PR8 is the toolkit for partial design, obtainable from [31] (special clearance is required to access this site, and can be requested on the Xilinx website itself). A version of PlanAhead later than the mentioned 9.2*i* is also required, however, as PlanAhead was only officially released in Xilinx suite 10.1, this is the version used to build this project (to be specific PlanAhead 10.1.08).

The design format guidelines are as follows: We need a design where there is a static part (non-reconfigurable hardware), serveral reconfigurable parts (known as a Partial Reconfigurable Units or PRUs), and a top-level entity. The top level entity should only contain black-boxed instances of the aforementioned units and Bus Macros to connect the static region to the reconfigurable region. Bus Macros (BMs) (available from [31]) are small buses, designed to tunnel the signals from and to the partial reconfiguration region (PRR). All signals, except global clocks, coming from and going into the reconfigurable portion of the design have to be routed through instantiated bus macros. The choice of the correct Bus Macro depends on the board on which the design will be implemented (in this case a xc2vp30 p:ff896 s:−6) and the direction in which the data will flow once implemented.

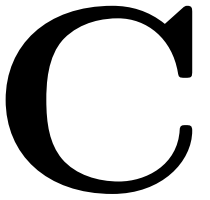
For the next phase we need to seperately synthesise all reconfigurable modules and the static part of the design. It is vital that neither the static region nor any of the reconfigurable instances contain IOBs, as only the top level entity can contain IOBs. Having synthesised the entire design it is time to start PlanAhead.

When starting PlanAhead we start a new project and we specify the synthesised top level design and the directories where the synthesised versions of the static part and the PRUs can be found. Next we need to let PlanAhead know that this project is to become a partial reconfigurable project. For this we type the command ‘hdi::pr setProject -name ¡project name¿’ into the PlanAhead console. Now we can flag the reconfigurable modules as reconfigurable using the ‘set Reconfigurable’ command. To add multiple modules to a single PRR, we can use the ‘add Reconfigurable Module’ command from the dropdown menu of a module marked as Reconfigurable. Marking a module as reconfigurable will also create a PRR in PlanAhead, that we now have to place on the device. When placing the PRR on a Virtex-II Pro device we have to keep in mind that only entire vertical strips can be reconfigured at the same time, so our PRR has to span the entire height of the device. Be sure to check the resources located inside the PRR are sufficient for every individial reconfigurable module in this PRR. Finally, the BMs

have to be placed on the device. This has to be done on the correct side of the PRR, meaning that if we have used a left-to-right (L2R) BM and the data stream direction is into the PRR, we have to place the BM on the left side of the PRR. Furthermore, the BM has to be placed in such a location, that one side is inside the PRR and the other side is located outside the PRR.

Since we should not place the static region on the device we are now ready to start testing the reconfigurable design. Running DRC will point out any violations regarding both regular and reconfigurable design. When DRC runs without errors we can continue to implement our design. First we have to execute a PlanAhead run for the static logic. Make sure that the used BMs (the .nmc files) are located in the `<project name>.runs/<floorplan name>` directory. Once completed, do the same for the reconfigurable portion of the design. Finally, when all previous steps have completed without errors, selecting 'Run PR Assemble' from the 'PlanAhead runs ¿ static' dropdown will finalise the design. The generated bitfile can be found as `'<project name>.runs/<floorplan name>/merge/static_full.bit'`.

Project Directory Structure



```
\-ISE
| \-busmacro
| \-reconfig
| | \-frequency
| | | \-freq_PR_bandpass
| | \-time
| | | \-pitch_modulator
| | | \-time_PR_chorus
| | | \-time_PR_comp_exp
| | | \-time_PR_delay
| | | \-time_PR_distortion
| | | \-time_PR_empty
| | | \-time_pr_octaver
| | | \-time_PR_reverb
| | | \-time_PR_tremolo
| | | \-time_PR_vibrato
| | | \-time_PR_volume
| \-static
| | \-ac97
| | \-clk_48k
| | \-dcm_40
| | \-fft_PR
| | \-keyboard_interface
| | \-reset_unit
| | \-VGA_interface_PR_BRAM
| \-top_level
```

```
\-PlanAhead
| \-full_time_only_juni_2009
| | \-full_time_only_juni_2009.runs
| | | \-floorplan_1
| | | | \-merge
| | | | \-pr_modules
| | | | | \-PR_time_1
| | | | | | \-chorus
| | | | | | \-comp_exp
| | | | | | \-delay
| | | | | | \-distortion
| | | | | | \-octaver
| | | | | | \-reverb
| | | | | | \-tremolo
| | | | | | \-vibrato
| | | | | | \-volumizer
| | | | \-static
| | \-full_time_only_juni_2009.data
| | | \-floorplan_1
| | | | \-earuns
| | | \-netlist
| | | | \-pr_modules
| | | | | \-PR_time_1
| | | | | | \-chorus
| | | | | | \-comp_exp
| | | | | | \-delay
| | | | | | \-distortion
| | | | | | \-octaver
| | | | | | \-reverb
| | | | | | \-tremolo
| | | | | | \-vibrato
| | | | | | \-volumizer
```