A Fixed-Point DSP Architecture for Software-Defined Radio

WOUTER KRIEGLER

Thesis presented in partial fulfilment of the requirements for the degree
Master of Science in Electronic Engineering
at the University of Stellenbosch.

SUPERVISOR: DR G-J VAN ROOYEN

March 2009
Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the owner of the copyright thereof (unless to the extent explicitly otherwise stated) and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

..............................................................................................
Signature

..............................................................................................
Date
Summary

Due to ever evolving wireless communication standards and technologies, the need for more flexible radio terminals are becoming more sought after in order to adapt to these new standards. Software-defined radio offers a solution to this demand. Software-defined radio is a radio communication system where signal processing components that have typically been implemented in hardware are replaced by reconfigurable and re-useable software modules running on a digital processor.

The need exists to rapidly create new SDR applications without designing an entire system from the ground up, and without specialised knowledge of the target platform. This thesis initially describes the design of a generic SDR architecture that is highly reconfigurable and promotes a high level of code re-use. The research forms part of a larger project to design a domain-specific language (DSL) in which to describe SDR functionality in a platform-independent way. In this thesis, the code synthesis from the DSL is extended to support the Freescale DSP563xx family.
Opsomming

Te danke aan die konstante vooruitgang van draadlose kommunikasie standaarde en tegnologieë, word daar radio terminale benodig wat meer buigsaam is. Sagteware gedefinieerde radio (SDR) bied 'n oplossing tot hierdie aanvraag. 'n Sagteware gedefinieerde radio is 'n radio kommunikasie stelsel waar seinverwerking komponente wat voorheen in hardware geïmplementeer is, nou deur her-konfigureerbare en her-bruikbare sagteware modules vervang word wat op 'n digitale verwerker uitvoer.

Die behoefte bestaan om spoedig nuwe SDR toepassings te ontwikkel sonder die nodigheid om 'n totale sisteem van die grondvlak af te ontwerp, en sonder dat die gebruiker 'n gespesialiseerde agtergrond van die teiken hardeware platform het. Hierdie tesis beskryf aanvanklik die ontwerp van 'n generiese SDR argitektuur wat 'n hoë vlak van her-konfigureerbaarheid bied, en kode her-gebruik bevorder. Die navorsing maak deel uit van 'n groter projek om 'n domein-spesifieke taal (DSL) te ontwikkel waarin SDR funtionaliteit in 'n platform onafhanklike manier beskryf kan word. In hierdie tesis word die kode sintese vanuit die DSL uitgebrei om die Freescale DSP563xx familie van digitale seinververkers te ondersteun.
Acknowledgements

- I would like to thank my supervisor, Dr Gert-Jan van Rooyen, for his constant support and guidance.

- Thank you to my parents for their support, and for giving me the opportunity to study at a leading institution such as Stellenbosch University.

- I would like to thank my girlfriend for her support during my studies.

- Thank you to all my friends for the support and good times.
Contents

Summary ii
Opsomming iii
Acknowledgements iv
Nomenclature xiv
Terms of Reference xvi

1 Introduction 1
   1.1 The Need For Software-Defined Radios . . . . . . . . . . . . . . . . . . . . 1
   1.2 Defining Software Defined Radio . . . . . . . . . . . . . . . . . . . . . . . . 2
   1.3 Project Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
   1.4 Thesis Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2 Background Study 6
   2.1 Stellenbosch University SDR Project . . . . . . . . . . . . . . . . . . . . . 6
   2.2 Hardware Platform . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
      2.2.1 DSP Processors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
      2.2.2 Development Platforms for DSPs . . . . . . . . . . . . . . . . . . . 9
      2.2.3 Target DSP Platform . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
   2.3 Fixed-Point Arithmetic . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
      2.3.1 Fixed-Point Representation . . . . . . . . . . . . . . . . . . . . . . . 11
      2.3.2 Fixed-point Calculations . . . . . . . . . . . . . . . . . . . . . . . . 13
   2.4 Introduction to XML . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
      2.4.1 Describing SDR in XML . . . . . . . . . . . . . . . . . . . . . . . . . 16
   2.5 XSL Transformations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
CONTENTS

2.5.1 Introduction to XSLT ........................................... 20
2.5.2 XSLT Instructions .............................................. 21
2.6 Conclusion .......................................................... 25

3 Architecture Design and Reference Implementation ........................................... 26
3.1 Generic SDR Topology ............................................. 26
  3.1.1 The Converter .................................................. 28
  3.1.2 The Attributes ................................................ 30
  3.1.3 The Queues .................................................... 32
  3.1.4 The Process ................................................... 37
  3.1.5 Linking Converters ........................................... 39
  3.1.6 Creating an SDR Application ................................. 42
3.2 Static Scheduler .................................................... 43
  3.2.1 Defining the System ........................................ 43
  3.2.2 SDF Constraints .............................................. 45
  3.2.3 Sequential Scheduling ........................................ 47
3.3 Conclusion .......................................................... 49

4 Hardware Implementation .................................................. 51
4.1 Memory Management ............................................. 51
4.2 Fixed-Point Arithmetic .......................................... 53
  4.2.1 Fixed-point Representation on DSP ........................ 53
  4.2.2 Fixed-point Calculations on the DSP ...................... 56
  4.2.3 Optimal Scale Factor ........................................ 61
  4.2.4 Rescale ....................................................... 62
  4.2.5 Merge with SDR Topology ................................... 62
4.3 ADC and DAC ....................................................... 66
  4.3.1 CODEC and CODEC Configuration ......................... 66
  4.3.2 Data Transfer ................................................ 68
4.4 Conclusion .......................................................... 70

5 Translation Layers ....................................................... 71
5.1 XML and XSLT ...................................................... 71
  5.1.1 Creating an SDR Application ................................. 71
5.2 XSLT Transformation ............................................. 77
### CONTENTS

5.2.1 Converter Definitions Transform ................................................. 78
5.2.2 Node Declarations and Configuration Transform .......................... 81
5.2.3 Node Link Definitions Transform .................................................. 83
5.2.4 Schedule Transform ................................................................. 83
5.2.5 XSLT Translation Layer Conclusion ............................................. 84

5.3 Lexical Analysis ............................................................................. 84
  5.3.1 Generating a Compiler ............................................................... 84
  5.3.2 Generating a Flex Scanner ......................................................... 86
  5.3.3 Generating a Bison Parser ......................................................... 90

5.4 Conclusion .................................................................................... 94

6 Tests and Results ............................................................................ 96
  6.1 Simulation Results ......................................................................... 96
  6.2 Hardware Results .......................................................................... 99
  6.3 Conclusion .................................................................................... 103

7 Conclusions ..................................................................................... 104
  7.1 Review of Results .......................................................................... 104
  7.2 Futher Work .................................................................................. 105
  7.3 Final Remarks ............................................................................... 106

A Appendix ........................................................................................ 110
  A.1 Contents of Accompanying CD .................................................... 110
  A.2 Code Listing ............................................................................... 111
List of Figures

1.1 A typical model for software-defined radios illustrating the different frequency ranges. ........................................... 3
1.2 A software-defined radio interfacing with a variety of networks. ........... 3
1.3 The Stellenbosch University SDR project uses XML as a DSL in which to specify radio functionality. DSL compilers then exist to automatically generate a HLL compatible with the desired hardware platform. ........... 4

2.1 Performance versus reconfigurability comparison between various baseband processing elements. Reproduced from [4] ......................... 9
2.2 Q4.20 format on a 24-bit word length representing the value 6.7890625. ... 11
2.3 Q[7].[17] format. ........................................................................ 14
2.4 Q[4].[20] format. ........................................................................ 14
2.5 Tree structure of a converter described in XML. ................................ 18
2.6 Process flow of XSL transformations. ............................................ 20

3.1 Block diagram of a phased-locked loop. ....................................... 27
3.2 Linear PLL model used for the demodulation of an FM signal. ............ 27
3.3 Block diagram of an FM demodulator, where each node can be modeled as a converter. ................................................................. 28
3.4 A complete SDR application (FM demodulator of Figure 3.3) showing each converter with its own set of input buffers and attributes. ............. 29
3.5 Converter and queue layout. ......................................................... 33
3.6 Two converters sharing a single queue. The second output queue of Converter_A points to the first input queue of Converter_B, thus linking them. 34
3.7 A typical buffer .......................................................................... 35
3.8 Diagram showing the layout of a typical converter. .......................... 40
3.9 A sine wave and a constant value linked to a sum converter to create a sine
wave with a DC offset. .................................................. 42
3.10 SDF graph for an FM demodulator. ................................. 44
3.11 An SDF graph. ......................................................... 44
3.12 An SDF graph with consistent sample rates, but no periodic admissible
sequential schedule. ....................................................... 46
3.13 Buffer sizes for the input queues of the nodes in Figure 3.11 during a single
iteration period. .......................................................... 50
4.1 Fixed-size block memory allocation strategy ........................ 52
4.2 The value \(-0.26171275\) represented using the \texttt{fract} data type. .... 54
4.3 Two values that have the same bit layout but representing two different
values. ........................................................................ 55
4.4 A fixed point value represented using different scale factors. ........ 56
4.5 Two values that have been aligned. Each value also contains one redundant
integer bit. ................................................................. 58
4.6 Addition operation resulting in an answer containing redundant integer bits. 61
4.7 Physical connections between the DSP56311 and the CS4218 audio codec.
Reproduced from [28]. .................................................... 67
4.8 Serial communication between the DSP56311 and the CS4218 in serial
mode 4. ....................................................................... 67
5.1 A complete SDR application consisting of various XML converter definitions. 73
5.2 Visual representation of a complete XML definition of the SDR application
illustrated in Figure 5.1. .................................................. 76
5.3 Overview of XSLT routines. ............................................. 78
5.4 Examples of how floating-point expressions will be converted to fixed-point
expressions. .................................................................. 85
5.5 Program flow for a complete compiler generated using Flex and Bison. ... 86
5.6 Program flow for the scanner. ........................................ 89
5.7 Program flow for the parser. .......................................... 95
6.1 A simple AM modulation-demodulation scheme using synchronous demod-
ulation. .................................................................... 97
6.2 Results obtained from both the DSP and Matlab implementations of the test application. Only a single demodulated curve is visible since the results are so closely matched. .................................................. 98
6.3 Test setup consisting of a DSP56311 evaluation module, a Picoscope3224 and a PC. .......................................................... 99
6.4 An AM modulation scheme. ................................................. 100
6.5 Time and frequency analysis of an AM modulated signal. The message signal has a frequency of 187.5Hz, and the carrier signal has a frequency of 6kHz. ................................................................. 100
6.6 Comparison of a machine generated AM modulation application and a hand-coded AM modulation application. .................. 101
List of Tables

2.1 Summary of XPath expression axes. Reproduced from [18]................. 23

4.1 Comparison of four memory management strategies......................... 53

6.1 Summary of processor loads incurred by the various SDR components in
Figure 6.4. ........................................................................... 102

6.2 Summary of processor loads incurred by various SDR components not in-
cluded in the test setup in Figure 6.4.............................................. 102

A.1 Summary of the contents of the accompanying CD......................... 110
## Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 An XML entity</td>
<td>15</td>
</tr>
<tr>
<td>2.2 XML definition of a Gain converter</td>
<td>19</td>
</tr>
<tr>
<td>2.3 Example of an xsl:variable</td>
<td>24</td>
</tr>
<tr>
<td>2.4 Example of a xsl:if element testing if a given variable is of type int</td>
<td>24</td>
</tr>
<tr>
<td>3.1 Definition of a Converter</td>
<td>30</td>
</tr>
<tr>
<td>3.2 Creating an instance of a converter</td>
<td>30</td>
</tr>
<tr>
<td>3.3 Definition of an attribute structure for a sine wave generator</td>
<td>31</td>
</tr>
<tr>
<td>3.4 Creating an instance of an attribute structure</td>
<td>31</td>
</tr>
<tr>
<td>3.5 Calling the setter function for an attribute structure</td>
<td>31</td>
</tr>
<tr>
<td>3.6 Setter function for an attribute structure</td>
<td>32</td>
</tr>
<tr>
<td>3.7 Definition of the QUEUE structure</td>
<td>34</td>
</tr>
<tr>
<td>3.8 Process of creating a new buffer</td>
<td>36</td>
</tr>
<tr>
<td>3.9 The setQueue function</td>
<td>37</td>
</tr>
<tr>
<td>3.10 Declaring a function pointer</td>
<td>37</td>
</tr>
<tr>
<td>3.11 Assigning a function to a process pointer</td>
<td>37</td>
</tr>
<tr>
<td>3.12 Calling a process with an attribute structure as argument</td>
<td>38</td>
</tr>
<tr>
<td>3.13 Function prototypes for popQueue() and pushQueue()</td>
<td>38</td>
</tr>
<tr>
<td>3.14 Example of the process function of an amplifier converter</td>
<td>41</td>
</tr>
<tr>
<td>3.15 The output ports of two converters are linked to the input ports of a third</td>
<td>41</td>
</tr>
<tr>
<td>4.1 Definition of the MY_FRACt structure</td>
<td>54</td>
</tr>
<tr>
<td>4.2 Function prototypes for _fract expressions.</td>
<td>57</td>
</tr>
<tr>
<td>4.3 C code for performing fixed-point multiplication</td>
<td>59</td>
</tr>
<tr>
<td>4.4 C code for performing fixed-point division</td>
<td>60</td>
</tr>
<tr>
<td>4.5 QUEUE implementation using a single scale factor for each element in the buffer</td>
<td>63</td>
</tr>
</tbody>
</table>
4.6 When a value is sent to the input queue of a connecting node, the value must first be rescaled to be compatible with the scale factor of the queue, as illustrated here. ........................................ 63
4.7 QUEUE implementation using individual scale factors for each element in the buffer. ........................................ 65
4.8 An attribute structure implemented using the MY_FRACT data type to represent fixed-point variables. ........................................ 65
4.9 Declaration of the receive and transmit buffers of the ADC and DAC modules. 68
5.1 Subdivisions of a complete SDR XML definition, enclosed within the <System> element. ........................................ 72
5.2 Component declaration and configuration of an amplifier node. ........ 75
5.3 Links definition between two nodes. ........................................ 75
5.4 Proposed schedule for a simple amplifier system listed in XML format. . 76
5.5 Extract of the XSLT transform that identifies attributes of type double in a XML converter definition, and adds them to the attribute structure for the current converter. ........................................ 79
5.6 XSLT transform for generating a setter function that will initialize fixed-point values. ........................................ 80
5.7 XSLT transform for selecting and copying the process definition to the output document. ........................................ 81
5.8 XSLT transform for determining the number of nodes in a SDR application, generating a list of CONVERTER pointers, and initializing these pointers. . 82
5.9 The overall layout a Flex input file. ........................................ 86
5.10 The Overall Layout of a Bison Grammar. ........................................ 90
5.11 Declaring two token types, along with the associativity and precedence of the tokens. ........................................ 92
5.12 A Bison rule for constructing a mathematical addition expression. ....... 93
A.1 Matlab code for calculating the schedule of the system defined in Fig: 3.11 111
A.2 Complete example of configuring a new converter of type "MUL". ........ 112
### Nomenclature

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM</td>
<td>Amplitude Modulation</td>
</tr>
<tr>
<td>ADC</td>
<td>Analogue-to-Digital Converter</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuits</td>
</tr>
<tr>
<td>BNF</td>
<td>Backus-Naur Form</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Style Sheet</td>
</tr>
<tr>
<td>DAC</td>
<td>Digital-to-Analogue Converter</td>
</tr>
<tr>
<td>DDS</td>
<td>Direct-Digital Synthesis</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain Specific Language</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>DTD</td>
<td>Document Type Definition</td>
</tr>
<tr>
<td>EDE</td>
<td>Embedded Development Environment</td>
</tr>
<tr>
<td>EOF</td>
<td>End-of-File</td>
</tr>
<tr>
<td>FIFO</td>
<td>First-In-First-Out</td>
</tr>
<tr>
<td>FM</td>
<td>Frequency Modulation</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FPU</td>
<td>Floating-point Unit</td>
</tr>
<tr>
<td>FSK</td>
<td>Frequency Shift Keying</td>
</tr>
<tr>
<td>GPP</td>
<td>General Purpose Processor</td>
</tr>
<tr>
<td>HLL</td>
<td>High-Level Language</td>
</tr>
<tr>
<td>IF</td>
<td>Intermediate Frequency</td>
</tr>
<tr>
<td>JTRS</td>
<td>Joint Tactical Radio System</td>
</tr>
<tr>
<td>LSB</td>
<td>Least Significant Bit</td>
</tr>
<tr>
<td>LPF</td>
<td>Low Pass Filter</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bit</td>
</tr>
<tr>
<td>OOP</td>
<td>Object-Oriented Programming</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PASS</td>
<td>Periodic Admissible Sequential Schedule</td>
</tr>
</tbody>
</table>
PLL Phased-Lock Loop
PM Phase Modulation
QF Fractional Bits
QI Integer Bits
QoS Quality of Service
RF Radio Frequency
RTOS Real Time Operating System
SCA Software Communications Architecture
SDF Synchronous Data Flow
SDR Software-Defined Radio
SFDR Spurious Free Dynamic Range
SNR Signal-to-Noise Ratio
SQNR Signal-to-Quantization-Noise Ratio
URI Uniform Resource Identifier
VCO Voltage Controlled Oscillator
WL Word Length
XML Extensible Markup Language
XSL Extensible Stylesheet Language
XSLT Extensible Stylesheet Language Transformation
Terms of Reference

This thesis was commissioned by Dr G-J van Rooyen of Stellenbosch University on 1 January 2007. His specific instructions were to extend the current Stellenbosch University Software Defined Radio framework, which currently supports code synthesis for microprocessors (C++) and FPGAs (VHDL), to also support code synthesis for DSPs (C).

The following subtasks were prescribed:

1. Create a functional prototype of the proposed extension in ANSI C on a microprocessor.
2. Port the C code to execute correctly on the Freescale DSP563xx family of DSPs.
3. Implement an XSLT translation layer, that can synthesise DSP563xx C from a generic XML definition of a software-defined radio.
4. If necessary, extend the current Stellenbosch University XML schema for radio definition, so that DSP targets are supported transparently.
5. Create a demonstration radio application that is defined in the top-level XML schema, and demonstrate that correctly executable DSP code can be synthesised directly.
Chapter 1

Introduction

1.1 The Need For Software-Defined Radios

Wireless communications is developing at a furious pace due to the emergence of new standards and protocols. This has led to a large demand for wireless Internet connectivity with integrated services that offer global coverage and user-controlled quality of service (QoS) [2]. With this demand for wireless Internet connectivity comes a desire for future-proof radios, which keep radio hardware and software from becoming obsolete as new standards and techniques become available.

The rate of technology advancement and innovation is increasing and predicting changes in communications standards are problematic. Consequently wireless system manufacturers and service providers must upgrade their systems accordingly to include new advancements or even fix bugs when they arise. This upgrade is most often expensive and time consuming increasing the need for future-proof radios.

Current technologies for voice, video, and data all use different packet structures, data structures, and signal processing techniques [2]. To provide integrated solutions a device must be capable of delivering various services, either by itself or by communicating with other devices providing complimentary services. These complimentary services might vary due to the physical location of the user. To successfully communicate with the supporting technologies the radio has to decode signals of the various devices using different air-interfaces, modulation techniques, or even protocols. To implement such devices the radio needs to be reconfigured based on the systems it will be interfacing with and the functions it will support.

Cross-functionality and flexibility of traditional hardware-based radios are limited,
and can only be changed by physically altering the hardware. This restricts the number of waveforms and functions supported by the application, and ultimately reduces the lifespan of such a device [1]. *Software-Defined Radio* (SDR) technology however resolves this problem by allowing multi-mode, multi-band and multi-function wireless devices that can be enhanced and upgraded using software [1].

### 1.2 Defining Software Defined Radio

*SDR Forum* defines SDR as [1]:

>a collection of hardware and software technologies where some or all of the radios operating functions are implemented through modifiable software or firmware operating on programmable processing technologies. These devices include field programmable gate arrays (FPGA), digital signal processors (DSP), general purpose processors (GPP), programmable System on Chip (SoC) or other application specific programmable processors.

This essentially means that a single piece of hardware can perform different functions depending on the software currently loaded. Although an SDR is usually composed of both software and hardware segments, a SDR is said to receive its flexibility through software while the hardware platform stays fixed. SDR then represents a shift from fixed, hardware-intensive radios to reconfigurable, software-intensive radios [2].

An ideal SDR implementation would convert a signal to the digital domain as early as possible to provide as much flexibility as possible. Due to high carrier frequencies used in the telecommunications industry, this constraint often places a huge performance demand on the digital processor. Ideal SDR implementations can therefore not yet be fully realized due to technological and cost limitations [2].

SDRs may therefore still require some form of radio frequency (RF) front-end to handle a wide range of carrier frequencies. The RF front-end will be responsible for mixing received signals down to a more manageable intermediate frequency (IF) before analogue-to-digital conversion is done. A typical model for an SDR is shown in Figure 1.1, with a RF front-end preceding the baseband processing segment.

By implementing radios using such a flexible platform compromised of both hardware and software, radio behavior can be changed with ease. This flexibility allows SDR to
seamlessly integrate into multiple networks that may drastically differ in both air and data interfaces as depicted in Figure 1.2.

1.3 Project Motivation

To implement systems such as those depicted in Figure 1.2, it must be possible to change the behavior of an SDR with ease. New applications have to be developed rapidly, and deployed to various hardware platforms without difficulty. The reconfigurability and
flexibility of an SDR system is therefore some of its most significant qualities.

Since SDR represents a shift to software intensive radios, a reconfigurable software architecture is needed to realize these qualities. If software is custom written for each SDR application, even a subtle change in the radio functionality may require an extensive code overhaul. This thesis aims to develop and implement a highly reconfigurable SDR architecture that allows new applications to be developed rapidly.

However, a highly reconfigurable system may not be highly flexible, in the sense that a significant coding effort may be needed to adapt the SDR application for different hardware platforms. Developing new SDR applications will also often require an extensive knowledge of the target hardware platform. This thesis will further aim to deploy new SDR applications to a specific hardware platform without prior knowledge of that hardware platform.

To accomplish this, SDR applications must be defined in a hardware independent manner. For this reason a domain-specific language (DSL) is evolved in which SDR applications can be defined. This will serve as a single top-level definition of an SDR from which various hardware-specific high-level languages (HLL) can be generated using DSL compilers as illustrated in Figure 1.3.

**Figure 1.3:** The Stellenbosch University SDR project uses XML as a DSL in which to specify radio functionality. DSL compilers then exist to automatically generate a HLL compatible with the desired hardware platform.
Chapter 1 — Introduction

This thesis will detail the design of an SDR architecture, compatible with the Freescale DSP563xx family of digital signal processors (DSP). The existing set of DSL translators will be extended to include the translation to a HLL supported by this hardware platform (Figure 1.3).

1.4 Thesis Overview

This thesis will start by reviewing the current SDR project at Stellenbosch University in Chapter 2. We will then give an overview of the target hardware platform for this project, and the numeric representation supported. The chapter will conclude with an introduction on the meta language XML, and how current SDR applications are represented using this meta language.

In Chapter 3 we will develop a reference implementation of a generic SDR topology, based on the current C++ architecture. This chapter will also discuss static scheduling of SDR applications.

Chapter 4 will implement the generic SDR topology created in the previous chapter, on the target hardware platform. Common embedded design issues are also addressed in this chapter.

In Chapter 5 we will expand the current DSL to define radio applications for the target hardware platform. The various translation layers used to synthesise the HLL supported by the hardware platform are then introduced.

We will then conduct various system tests to verify the functionality of the developed SDR architecture and code synthesis process in Chapter 6.

The thesis will conclude in Chapter 7, with an overview of the thesis and suggested improvements.
Chapter 2

Background Study

This chapter will introduce some of the concepts used during this project. The chapter starts with a brief overview of the SDR project at Stellenbosch University, and identifies the gap in the research upon which this project is based. A discussion regarding the various hardware platforms suitable for SDR follows, identifying the target hardware platform for this project. The numeric representation compatible with the target hardware platform is then discussed in detail. The chapter concludes with an introduction on the meta language used to define SDR applications.

2.1 Stellenbosch University SDR Project

The SDR philosophy at Stellenbosch University is to decompose complex signal processing applications into modular, reusable building blocks. These building blocks can be individually configured and linked together to create a near endless amount of SDR applications. By using this approach, highly reconfigurable SDR applications can be developed. Previous research projects have developed and implemented such an architecture on a general purpose processor (GPP) [5].

In order to increase hardware portability of this reconfigurable architecture, a radio description language was developed in which to specify the functionality of an SDR application in a hardware independent manner. The XML meta-markup language was used to specify these DSL modules. To keep the architecture as reconfigurable as possible, the individual building blocks comprising a SDR application are each described using XML. These building blocks were dubbed converters, since they would normally convert some input signal to an output signal. A converter will normally contain these features:
1. Zero or more input ports from where tokens are received, whether these are signals, control information or data.

2. A set of attributes that defines the converter. For example, a amplify element would contain an attribute specifying its gain.

3. All converters will contain a process definition that specifies how input tokens are converted to output tokens.

4. Zero or more output ports where tokens will be placed after being processed.

Section 2.4.1 will provide a more in depth overview of how converters are specified using XML.

The XSLT language was then introduced to translate these generic platform independent DSL modules, to functional HLL code for the desired hardware platform. Previous research projects at Stellenbosch University have successfully implemented DSL translation to two hardware platforms. Figure 1.3 shows a project overview, in which a fully functional implementation to a microprocessor has been completed [5], and a prototype implementation for a VHDL-based hardware platform for FPGA has been demonstrated [6]. This illustrates the re-use of code: A single generic XML definition for an SDR radio exists; different DSL translators then generate code, compatible with a variety of platforms.

To complete the project, the goal was set to synthesise HLL code compatible with an embedded microprocessor.

2.2 Hardware Platform

As described in Chapter 1, a typical SDR system will consist of a digital processor that is usually preceded by some form of RF front-end. The digital processor of an SDR system can be considered as the heart of the system since it is responsible for implementing waveform processing. Four key considerations when examining a digital hardware platform include [2]:

• **Flexibility.** The ability to accept to a wide variety of air-interfaces, protocols and data rates.

• **Modularity.** The ability to upgrade and replace subsystems of an SDR environment.
• **Scalability.** The ability to enhance and improve the capability of an SDR.

• **Performance.** Performance includes the power consumption, cost, and computational capabilities of a digital processor.

The three interrelated properties, flexibility, modularity and scalability, can be summarised as the level of reconfigurability of a digital processor. Reconfiguration is one of the most significant characteristics of SDR, and the chosen hardware platform should complement this feature.

Digital signal processing is done mainly on three hardware platforms: Microprocessors (µP), FPGAs, and application specific integrated circuits (ASIC) [2]. Microprocessor-based platforms include DSPs and general purpose processors (GPP) such as desktop computers. A trade-off between reconfigurability and performance exist between these hardware platforms as depicted in Figure 2.1. DSPs tend to have a high level of reconfigurability due to the inherent microprocessor-based architecture, and the support of high level programming languages such as C. This flexibility comes at the price of reduced performance due to the sequential processing nature of microprocessors. An ASIC is the most specialised hardware platform and is implemented on fixed circuitry, resulting in the most optimized implementation regarding both speed and power consumption. Due to more complex circuit design and sophisticated hardware layout tools, ASIC provide a much lower level of reconfigurability, and can only be used in the specific application for which it has been designed [2].

DSP platforms clearly offer the highest level of reconfigurability, and is an obvious choice for implementing SDR architectures offering high levels of flexibility.

### 2.2.1 DSP Processors

A DSP is essentially a microprocessor optimised for digital signal processing applications. Special functions and instructions are included in DSP architectures to support high performance, numerically intensive tasks. Typically DSPs consist of an arithmetic logic unit (ALU), multiply accumulate (MAC) unit, and data and address buses. MAC operations are highly optimised for DSPs, allowing for much improved performance over other microprocessor-based platforms such as GPPs. Furthermore, if a Harvard architecture is implemented, DSPs are able to access both data and program memory simultaneously [4].

DSP processors fall into two major categories based on the way they represent numerical values and implement numerical operations: floating-point and fixed-point arithmetic.
2.2.2 Development Platforms for DSPs

Well-established programming methods exist today that allows DSPs to be reconfigured with ease [4]. DSPs are most commonly programmed in either Assembly or a high-level language (HLL). Assembly programming usually performs better than a HLL, at the expense of having a longer development cycle. However, when programming using a HLL, rapid software development and program debugging is possible.

By using a HLL, applications can be developed on familiar platforms such as GPPs. This prototype implementation can then be ported to the target DSP platform with relative ease at a later stage [2]. This is a useful feature, and will be exploited in this

---

1It is not uncommon for compiled HLL code for a DSP to be an order of magnitude slower than hand-coded assembly [2]
A programmer should retain an overall familiarity with the target DSP architecture since this will assist in writing optimal code for the target. Furthermore it will help during debugging, and when hand-optimizing portions of the compiler generated assembly.

2.2.3 Target DSP Platform

The target hardware platform for this project is the Freescale DSP56311 fixed-point 24-bit digital signal processor. Fixed-point arithmetic will therefore form an integral part of this project, and is discussed in more detail in Section 2.3. The \textit{DSP56311 Evaluation Module} (DSP56311EVM) is used as development and test platform. The main features of the DSP56311EMV include [28]:

- DSP56311 24-bit digital signal processor.
- FSRAM for expansion memory and flash memory for stand-alone operation.
- 16-bit CD-quality audio codec.

The \textit{TASKING Embedded Development Environment} (EDE) in conjunction with the TASKING DSP56xxx Software Development Toolset from Altium will be used for this project. The TASKING DSP56xxx Software Development Toolset is a programming package for the Freescale DSP56xxx family of DSPs, supporting fully compliant C++ and ANSI-C compilers. The toolset also includes the \textit{Crossview Pro Debugger} with an instruction set simulator.

2.3 Fixed-Point Arithmetic

On many embedded hardware platforms there is no hardware support for floating-point arithmetic. This is attributed to the increased cost to implement a \textit{floating-point unit} (FPU) on the processor. Software support is then needed to emulate floating-point arithmetic and can significantly limit the speed at which algorithms execute.

By using \textit{fixed-point} (integer) mathematics, algorithms may execute significantly faster due to the integer math hardware support in a large number of processors. The trade-off is that the range and accuracy may be reduced when compared to the floating-point equivalent representation. The target hardware platform for this project does not include a FPU. Real data must therefore be represented using the fixed-point arithmetic.
2.3.1 Fixed-Point Representation

The two’s complement signed fixed-point representation will be used during this project. This representation requires the programmer to create a virtual decimal point in between two bit locations for a given length of data. This can be illustrated using Q-point notation.

\[
Q[I][F]
\]  

where \( Q_I \) is the number of integer bits and \( Q_F \) is the number of fractional bits. Signed integer variable types will be used and the integer bits will therefore include a sign bit. The number of integer bits plus the number of fractional bits must be equal to the total number of bits used to represent the fixed-point value. The sum of \( Q_I \) and \( Q_F \) corresponds to the various data type frame widths on a given processor, and is know as the word length (\( W_L \)). For the Motorola DSP56311 the word length is 24 bits.

To represent a floating-point value the number must be viewed as two separate parts, the integer value and the fractional value. The integer range is determined by the number of bits used to represent the integer part (\( Q_I \)). The resolution of the fractional part is determined by the amount of fractional bits (\( Q_F \)). The fixed-point range of a signed integer \( \alpha \) is:

\[
(-2^{Q_I - 1}) \leq \alpha \leq (2^{Q_I - 1} - 1)
\]

This equation can then be used to calculate the number of integer bits needed to represent a value. Equation 2.3 can be used calculate the number of bits needed for a positive value, and Equation 2.4 can be used calculate the number of bits needed to represent a negative integer:

\[
Q_I \geq \log_2 (\alpha + 1) + 1
\]

\[
Q_I \geq \log_2 (-\alpha) + 1
\]

These equations will always yield a positive value. An integer value may not always be produced, however. When a non-integer value is calculated, the value must be rounded up.
to get the correct answer. This can be incorporated with Equation 2.3 and Equation 2.4 to produce the following generalised equation which will always produce an answer such that \( Q_I \geq 1 \).

\[
Q_I = \lceil \log_2 (\max (|\alpha_{\text{min}}|, |\alpha_{\text{max}}|) + 1) \rceil + 1 \tag{2.5}
\]

We can implement a simple example to illustrate the use of these equations. To compute the number of integer bits needed to represent a signed variable within the ranges \(-13.456 \leq \alpha \leq 13.456:\)

\[
Q_I = \lceil \log_2 (\max (|-13.456, 13.456|) + 1) \rceil + 1
\]

\[
Q_I = \lceil \log_2 (14.456) \rceil + 1
\]

\[
Q_I = \lceil 3.854 \rceil + 1
\]

\[
Q_I = 4 + 1 = 5 \tag{2.6}
\]

To verify that \( \alpha \) is bounded by the minimum and maximum values of a 5-bit signed value we use Equation 2.2:

\[
(-2^{Q_I-1}) \leq \alpha \leq (2^{Q_I-1} - 1)
\]

\[
(-2^5) \leq \{-13.456, 13.456\} \leq (2^4 - 1)
\]

\[
-16 \leq \{-13.456, 13.456\} \leq 15 \tag{2.7}
\]

This confirms that \( \alpha \) can be represented using a 5-bit signed value.

The resolution of a fixed-point value is determined by the number of fractional bits \( (Q_F) \). The resolution, \( \epsilon \), is given by:

\[
\epsilon = \frac{1}{2^{Q_F}} \tag{2.8}
\]

Clearly, the more fractional bits used, the higher the resolution of the value. To calculate the number of fractional bits required for a given resolution the following equation can be used:

\[
Q_F = \lceil \log_2 \left( \frac{1}{\epsilon} \right) \rceil \tag{2.9}
\]

For example, when a resolution of \( \epsilon \leq 0.0001 \) is required, at least 14 fractional bits are needed:

\[
Q_F = \lceil \log_2 \left( \frac{1}{0.0001} \right) \rceil = 14 \tag{2.10}
\]

\( \lceil \cdot \rceil \) is the ceiling function, which rounds its argument to the nearest integer towards plus infinity.
As mentioned earlier, a fixed-point value is represented using a number of integer bits and a number of fractional bits. The sum of these bits must be equal to the \( W_L \) of the data type. The number of integer bits needed to represent the integer part (range) of the value will receive preference over the fractional bits. The number of bits remaining \((W_L - Q_I)\) may then be used to represent the fractional part of the value. This will limit the resolution of the fractional data. If a higher resolution is needed for a given range the \( W_L \) must be increased to provide the required resolution.

Once the required characteristics of a fixed-point value have been calculated based on the \( W_L \), range, and resolution of a floating-point value, a fixed-point representation of the floating-point value can be calculated. Equation 2.11 shows the relationship between floating-point and fixed-point values.

\[
F_{\text{Fixedpoint}} = F_{\text{Floatingpoint}} \times 2^{Q_F} \tag{2.11}
\]

Fixed-point representations may only contain integer values. The result obtained from equation 2.11 must therefore be rounded towards 0. For example, to represent the floating-point value 25.678 with a resolution of \( \epsilon = 0.0001 \), we need \( Q_I = 6 \) [Equation 2.5] and \( Q_F = 14 \) [Equation 2.9]. To accommodate these specifications a WL of 20 bits is required, or \( Q[6].[14] \) in Q-notation. Suppose that a 24-bit processor is used. There are now 4 extra bits which can be used to increase the resolution of the value. The value can then be represented in \( Q[6].[18] \) format. The resolution is then increased to \( 3.815 \times 10^{-6} \), while still accommodating the required range.

### 2.3.2 Fixed-point Calculations

#### Addition and Subtraction

Consider Figure 2.3 and Figure 2.4. These figures illustrate the bit layout of two fixed-point values using a \( W_L = 24 \), a \( Q[7].[17] \) and a \( Q[4].[20] \) value respectively. Addition is pure integer addition, since both of these values are really integer values with a virtual radix point. Care must be taken to first align the decimal places of the values being added. The value with the smallest number of integer bits must be right-shifted and sign-extended to align the decimal places of the two values. During addition, bit overflow may occur. If a mechanism exists to detect if overflow has occurred (carry-flag or overflow-flag), both values must be right-shifted and sign-extended one decimal place before addition to create an extra integer bit to ensure that overflow will not occur. Alternatively, both values may
Multiplication and Division

When performing multiplication the $W_L$ of the product will be the sum of the WL’s of the two values being multiplied. The number of integer and fractional bits in the product will be the sum of the corresponding integer and fractional bits of the values being multiplied. If a $Q[4].[20]$ value is multiplied with a $Q[10].[14]$ value the product will be a $Q[14].[34]$ number as described by Equation 2.12 and Equation 2.13.

\[
Q_I \text{ Product} = Q_I \text{ Multiplicand} + Q_I \text{ Multiplier} \tag{2.12}
\]
\[
Q_F \text{ Product} = Q_F \text{ Multiplicand} + Q_F \text{ Multiplier} \tag{2.13}
\]

Typically when fixed-point multiplication algorithms are implemented the answer will be scaled back to the original $W_L$ for later use in a similar algorithm. To achieve this the number of fractional bits will be reduced until the desired $W_L$ is reached. Division is handled in a similar way. Section 4.2 will describe how these algorithms are implemented on a DSP.
2.4 Introduction to XML

*Extensible markup language* (XML) is a meta-markup language that lends itself to structured declarative definitions of systems. A markup language is a mechanism for identifying structures or data in a document. Data is included in XML documents as strings of text, and data is surrounded by text markup that describes the data [19]. A unit of data and markup is called an *element*. Since XML is a *meta*-markup language, users can define their own set of tags to categorise data as needed [18]. XML is a flexible language able to accommodate a near endless variety of data definitions or structures. Tags created by users can be documented in a *document type definitions* (DTD). A DTD is a vocabulary and syntax describing certain types of XML documents. For example, Chemical Markup Language (CML) describes a vocabulary and syntax for managing molecular information.

XML describes a document’s structure and meaning, but it does not describe the formatting of the documents. The user can determine suitable and meaningful names for tags according to their own discretion. Tags do not need be part of an existing standard or specification and can be chosen to fit the information being described. XML markup makes it easy for computer automated software to locate tags when it is described in a meaningful manner. Other markup languages such as HTML do not allow user-defined tags which makes it more difficult to identify and understand the structure and data of a document. XML is an incredibly simple data format containing self-describing data. Consider the example in Listing 2.1. Without any familiarity with XML the user can determine that this fragment describes a data variable called *gain*.

```
<attribute type="double">
    <name>gain</name>
    <default>10</default>
    <access>Read/Write</access>
    <description>Amplifier gain</description>
</attribute>
```

**Listing 2.1: An XML entity**

XML can be used as more than just a data format. Several related technologies exist to help accomplish this [18]:

- *Cascading Style Sheets* (CSS) are used to let a browser know in advance how to display a XML document since no formatting information exists in the XML docu-
The Extensible Stylesheet Language (XSL) is a more powerful style language designed specifically for XML documents.

XSL Transformations (XSLT) is an XML application that describes transformations from an input XML document to another in either the same or different XML vocabularies. The output document does not necessarily have to be an XML document [18].

XLinks and XPointers are used to link to other XML documents and individual parts of a document respectively.

XPath is a non-XML syntax used by both XLinks and XPointers for identifying particular pieces of XML documents.

XML provides full support for the Unicode character set. XML defines character references that allow you to use ASCII to encode characters not available in your native character set.

XML is a meta-markup language for designing domain-specific markup languages [18]. Each XML-based markup language related to a specific field is called an XML application. We will now give an overview of how DSL modules are defined in the Stellenbosch University SDR project, using XML.

2.4.1 Describing SDR in XML

As mentioned earlier, the DSL modules in a SDR application are referred to as converters. A complete SDR application may consist of many converters. More than one type of converter is usually present in a SDR application and there may be several instances of each converter type. We will first describe how a single converter is defined using XML. This definition will then be used as a building block for describing a complete SDR application in XML in Section 5.1.1.

The XML definition of a converter must contain the necessary information to define all aspects of a converter, since the high-level language (HLL) equivalent version of the converter will be generated from this XML definition. The root element of each XML converter definition is therefore <converter>. Converters consist of the following elements:
• **Input queue(s).** Each input queue must have a unique `<name>` specific to the converter (but not necessarily the other converters in the SDR application) to distinguish it from the other possible input queues. A data type for each queue should also be specified. For this architecture the data type of a queue will be forced to **double**. We will expand on this restriction in the coming chapters.

• **Output queue(s).** Each output queue must have a unique `<name>` specific to the converter (but not necessarily the other converters in the SDR application) to distinguish is it from the other possible output queues. As with input queues, a data type should be specified for each output queue. The type will however be ignored and forced to **double**.

• **Attribute(s).** Each attribute of the converter must have a unique `<name>` specific to that converter, but not necessarily the other converters in the SDR application. In addition, each attribute must have a data type. The data type should be specified using the ANSI-C standard. In order to ensure that an attribute is always initialized, a `<default>` value for the attribute should be specified. If an initial value for a attribute is not specified when a converter is first created, this default value should be used.

• **Process definition.** Each converter will contain a single `<code>` segment containing a process definition describing how input samples are converted to output samples. The syntax of the code should also be specified. For this architecture code should be written in the ANSI-C standard. This restriction will be discussed further in Section 5.3.

Figure 2.5 shows the layout of a typical converter, in this case an **amplifier**. The complete XML definition of the converter is shown in Listing 2.2. The individual converter elements have descriptive and meaningful names, making the XML markup both human and machine readable. The representation of information is hierarchical and has a clear tree structure. These characteristics will be of crucial importance once XML converter definitions must be translated to other high-level languages.
Figure 2.5: Tree structure of a converter described in XML.
Chapter 2 — Background Study

Listing 2.2: XML definition of a Gain converter
2.5 XSL Transformations

2.5.1 Introduction to XSLT

The Extensible Stylesheet Language Transformation (XSLT) provides elements that define rules for how one XML document is transformed into another XML document. Though XSLT is designed primarily for XML-to-XML or XML-to-HTML transformations, XSLT processors can also be made to output essentially arbitrary text [18]. During an XSL transformation, an XSLT processor reads both an XML document and an XSLT stylesheet. The XSLT processor executes instructions found in the XSL stylesheets and outputs a new document as illustrated in Figure 2.6.

![Figure 2.6: Process flow of XSL transformations.](image)

A well-formed XML document is a tree. The top node is called the root and connected to it are children nodes, each connected to zero or more children and so forth. An important property of a tree is that each node and its children also form a tree. A tree is thus an hierarchical structure of trees. This is clearly evident in Figure 5.2. XSLT processors model XML documents as a tree containing nodes, where nodes include the following [18]:

- The root
- Elements
- Text
- Attributes
An XSLT processor accepts as input a tree represented as an XML document and produces as output a new tree. XSLT contains operators for identifying and selecting nodes from a tree. If the selected node is an element then it may be an entire tree in itself. XSLT documents contain template rules. A template rule has a pattern that specifies which nodes to match, and what to output when the pattern is matched. An XSLT processor traverses an input XML document starting at the root and following an order defined by the template rules. When the processor visits a node it compares that node with the pattern of each template rule in the XSLT stylesheet. If the node matches, the processor outputs the rule’s template.

Each template rule is an \texttt{xsl:template} element. The \texttt{match} attribute of the element \texttt{xsl:template}, contains the pattern of the rule. The format of the output is defined in the contents of the \texttt{xsl:template} element. Instructions in the template for performing tasks are defined by various XSLT elements. XSLT elements are identified by an \texttt{xsl:} prefix on the element name. All elements that do not contain an \texttt{xsl:} prefix is part of the output tree or result. The content of a \texttt{xsl:template} element contains the actual template that will transform input to output. A template will generally contain both literal text that will appear in the output as well as XSLT instructions that will identify and copy data from the input XML document to the output.

### 2.5.2 XSLT Instructions

This section will introduce some of the XSLT instructions and related technologies commonly used in templates to identify and process XML data. This is by no means a complete guide to XSLT, but touches on a variety of concepts and features used in this thesis.

The \texttt{match} attribute of the \texttt{xsl:template} element allows you to express exactly which nodes you want to match and which nodes not to match. Match patterns allows you to match nodes by element name, child element, descendants and attributes. Below is a brief summary of the most commonly used match operators:

- **Matching the root node.** To specify the root node in a rule the “/” character should be assigned to the \texttt{match} attribute. XSLT stylesheets generally start with a rule to match the root node in order to output the root node of the output document.

- **Matching element names.** One of the most basic patterns to match is single element. The name of the element to be matched is simply assigned to the \texttt{match} attribute.
• **Wilds cards.** To match more that one node the asterisk wildcard (*) can be assigned to the match attribute used instead of an element name. The asterisk wildcard will match all elements. When more than one rule matches a node, the more specific rule takes precedence.

• **Matching children.** The “/” symbol can be used to match specified hierarchies of elements when used between two element names. For example converter/name will match only name elements that are direct children of converter elements, and ignore other name elements. Patterns can be stringed together to specify deeper matches. For example ConverterDefinitions/converter/name matches only name elements whose parent is a converter element whose parent is a ConverterDefinitions element. The * wildcard can be combined with these patterns to substitute for any arbitrary element name in a hierarchy. The rule converter/* will match all children of the converter element.

• **Matching descendants.** The “//” symbol can be used so select all elements of a given type no matter how deep they occur in the hierarchy. For example, the ConverterDefinitions//name rule will match all name descendants, no matter how deep they are, of ConverterDefinitions.

• **Matching attributes.** The “@” sign can be used to match attribute names by prefixing an ”@” sign to the attribute name to be matched. The buffer/@name will match name attributes of buffer elements.

• **Matching multiple patterns.** The “|” operator can be used to match multiple patterns. If a node matches either patterns the template will activate. For example, the rule input|output will match both input and output elements.

The select attribute is used by many XSLT instructions to specify a particular set of children on which to perform the desired instruction. The patterns used by the select attribute are the same kind as those of the match attribute in the xsl:template element. More complex patterns for matching or selecting nodes are possible when using XPath. The XPath Language provides a means of identifying a particular element, group of elements, text fragment, or other part of an XML document. XPath not only allows you to select nodes through the criteria of match patterns, but also by referring to ancestor nodes, parent nodes, sibling nodes, preceding nodes and following nodes. XPath provides
### Table 2.1: Summary of XPath expression axes. Reproduced from [18].

<table>
<thead>
<tr>
<th>Axis</th>
<th>Selects From</th>
</tr>
</thead>
<tbody>
<tr>
<td>ancestor</td>
<td>The parent of the context node, the parent of the parent of the context node, the parent of the parent of the context node and so forth back to the root.</td>
</tr>
<tr>
<td>ancestor-or-self</td>
<td>The ancestors of the context node and the context node itself.</td>
</tr>
<tr>
<td>attribute</td>
<td>The attributes of the context node.</td>
</tr>
<tr>
<td>child</td>
<td>The immediate children of the context node.</td>
</tr>
<tr>
<td>descendant</td>
<td>The children of the context node, the children of the children of the context node and so forth.</td>
</tr>
<tr>
<td>descendant-or-self</td>
<td>The context node itself and its descendants.</td>
</tr>
<tr>
<td>following</td>
<td>All nodes that start after the end of the context node, excluding attributes and namespace nodes.</td>
</tr>
<tr>
<td>following-sibling</td>
<td>All nodes that start after the end of the context node, and have the same parent as the context node.</td>
</tr>
<tr>
<td>namespace</td>
<td>The namespace of the context node.</td>
</tr>
<tr>
<td>parent</td>
<td>The unique parent of the context node.</td>
</tr>
<tr>
<td>preceding</td>
<td>All nodes that finish before the beginning of the context node, excluding attributes and namespace nodes.</td>
</tr>
<tr>
<td>preceding-sibling</td>
<td>All nodes that finish before the beginning of the context node, and have the same parent as the context node.</td>
</tr>
<tr>
<td>self</td>
<td>The context node.</td>
</tr>
</tbody>
</table>

a number of axes allowing you to select from different parts of the tree relative to a particular node in the tree called the context node.

Table 2.1 provides a summary of the axes and their meanings. When specifying an axis, a rule is limited to expressions that only select nodes as indicated by Table 2.1. An axis is usually followed by a double colon (::), and a pattern that further focus the node set.

The value of an XML node can be computed and copied to an output document using the `xsl:value-of` element. The select attribute of an `xsl:value-of` element is used to specify exactly which node value to compute. For example, `<xsl:value-of
select="name"/> element can be used compute the name of an element and copy it to an output document. Any XPath expression can be used with the select attribute to specify which nodes to process.

When multiple items exist that can be process by an XSLT instruction such as xsl:value-of, only the first item will be processed. For example, when the value of multiple converter elements must be computed using xsl:value-of, only the first converter value will be computed. Usually this is not the desired outcome. It is possible to process each of the nodes using the xsl:for-each element. Each element chosen by its select attribute is processed in turn.

It is possible to declare constants in XSLT to help clean up code. The xsl:variable element defines a string that can be used elsewhere in the stylesheet. The name attribute of xsl:variable provides a reference to this defined string. The string is defined in the contents of the element. Alternatively the select attribute can be used to assign a value to the variable. For example, Listing 2.3 declares a variable named buffer_name and assigns the value of the name attribute of a buffer element to it. The value of a variable is accessed by prefixing a dollar ($) sign to the name of the variable.

```xml
<xsl:template match="/component/buffer">
   <xsl:variable name="buffer_name" select="@name"/>
</xsl:template>
```

Listing 2.3: Example of an xsl:variable.

XSLT provides an xsl:if element for making choices that influence the output based on the input. The test attribute of the xsl:if element contains an expression that evaluates to either true or false. If the expression is true the contents of the element are output, otherwise not. Consider the example in Listing 2.4. The test attribute of xsl:if is assigned the expression "@type='int'". Thus if the type attribute of some arbitrary element is equal to ‘int’, the expression will evaluate to true and the contents of the xsl:if element will will be output.

```xml
<xsl:if test="@type='int'">
   int <xsl:value-of select="./name"/>
</xsl:if>
```

Listing 2.4: Example of a xsl:if element testing if a given variable is of type int.
2.6 Conclusion

This chapter started by firstly giving some background surrounding the SDR project at Stellenbosch University, and identifying the gap in the research which this thesis aims to fulfill. The target hardware platform was then discussed, and the fixed-point arithmetic supported by the platform was introduced. The chapter concluded with an overview on XML and XSLT which is used to define SDR applications in a platform independent manner. This will be further discussed in Chapter 5.
Chapter 3

Architecture Design and Reference Implementation

In the previous chapter, C was identified as development language for a DSP-based platform. This chapter will focus on the implementation of a C-based architecture that plans to duplicate the design patterns of the current SDR architecture developed at Stellenbosch University. This includes the emulation of OOP principles in a non-OOP programming language. Furthermore, the architecture must promote code that can be easily machine generated, since the ultimate goal of this project is the automatic generation of a high-level language.

It was chosen to first implement a standard C reference design on a microprocessor based platform. This is due to the convenience of the platform as development environment. The architecture was developed on the Linux operating system, using a standard GCC compiler. Chapter 4 will describe how this design is ported to a DSP platform.

3.1 Generic SDR Topology

Telecommunications networks or signal processing applications can be modeled as a series of interconnected nodes and sub-systems, where a sub-system in turn consists of a series of nodes. A SDR application can be modeled in the same way. Consider Figure 3.1. This image shows a simple block diagram for a FM demodulator using a phase-locked loop (PLL). Such a system contains the following elements:

1. A phase detector
2. A loop filter

3. A loop amplifier

4. A voltage-controlled oscillator (VCO)

These system elements are the nodes referred to earlier. Each of these nodes perform a specific function, be it the amplification of a signal or multiplication of two signals. These nodes are then connected to each other to create a radio application. Each of these nodes can be represented using an equivalent mathematical function, that can be implemented using software. Figure 3.2 shows the equivalent linear model of a PLL. This model now contains a series of nodes performing these mathematical functions.

Software modules can now be created to perform the action of these nodes. A node will receive the necessary amount of data needed to execute. This data will then be processed according to the mathematical properties of the node, and the result will be passed to the next node in the system. Nodes will execute sequentially to carry out the task of the radio application.

Figure 3.1: Block diagram of a phased-locked loop.

Figure 3.2: Linear PLL model used for the demodulation of an FM signal.
3.1.1 The Converter

As discussed earlier, a node in a signal processing network is responsible for performing mathematical operations on tokens. Tokens are consumed by the node, processed and then passed to the next node in the network. Streams of input tokens are therefore converted to streams of output tokens. Tokens may be data, digital representation of analogue signals or control information. From here on we will refer to a node as a converter.

As discussed in the previous chapter, a converter consists of the following main elements:

1. A list of input queues where data samples will reside that still have to be processed.
2. A process describing the mathematical operation of the converter.
3. A list of attributes used by the converter to perform its action.
4. A list of output queues where samples will reside that have been processed.

![Block diagram of an FM demodulator, where each node can be modeled as a converter.](image)

Each converter must contain at least one input or one output. Each converter must further contain a process declaration that describes how input samples must be converted to output samples. Most converters will also contain a list of attributes that will be used by the process to adjust or configure the task of the converter. This may be the gain value of a amplifier node, or the filter coefficients of a filter node. These converters can
then be connected to each other to create a complete radio application as illustrated in Figure 3.4.

![Figure 3.4: A complete SDR application (FM demodulator of Figure 3.3) showing each converter with its own set of input buffers and attributes.](image)

We can create a data structure containing these components. Listing 3.1 shows the C code for the definition of a `CONVERTER` structure. This may resemble a class construct in a object-oriented programming (OOP) language. Classes enables the programmer to model objects that have *attributes* and *behaviors* [15]. Once a class has been defined, the class definition can be used to declare objects of that class. Since we are not using an OOP language, a structure will be used instead, but OOP concepts will be emulated where necessary.

The `CONVERTER` structure contains pointers to the four main elements of a converter. The first is a pointer to a process using C *function pointers*. A number of functions can be specified elsewhere in the program and linked to instances of the different converters using function pointers. The next two are pointers to a list of input and output queues. A converter may have more than one input or output queue. A input and output list is created that in turn contains pointers to the individual input and output queues. The final pointer is one that points to an attribute structure for the converter. The attribute structure will be discussed in Section 3.1.2.

When a new radio application is developed, an instance of each converter in the system
typedef struct sdr_converter{
    void (*process)(void *state);
    QUEUEptr* input_list;
    QUEUEptr* output_list;
    void *attributes;
} CONVERTER;

Listing 3.1: Definition of a Converter

must be declared. This means memory must be allocated for the converter and a reference to that converter must be created. A array of type CONVERTER* is created for this reason. Each element in the array will then serve as a reference to a converter instance. An instance of a converter is created by allocating memory for that converter. The returned pointer will be of type CONVERTER*. An array element can then be assigned to that converter. Listing 3.2 contains an example of this process.

CONVERTER *SDF[NUM_OF_CONVERTERS];

SDF[0] = (CONVERTER *) malloc(sizeof(CONVERTER));
SDF[1] = (CONVERTER *) malloc(sizeof(CONVERTER));
SDF[2] = (CONVERTER *) malloc(sizeof(CONVERTER));

Listing 3.2: Creating an instance of a converter.

An empty converter instance now exist. The next sections will describe the individual components of a converter, and then finally how to create and configure a new converter.

3.1.2 The Attributes

A converter may contain a set of attributes that are parameters to its processing algorithm. For example, an amplifier will have a gain attribute. The value of this attribute will be used by the process function to perform the action of the converter. Each converter will have a unique attribute structure associated with it. This structure will be specific to the type of the converter. Each converter in the system will have its own attribute instance. If multiple converters of the same type exist in a system, each will have an instance of the attribute structure.
typedef struct sdr_src_sin {
    CONVERTER *parent_struct;
    double frequency;
    double amplitude;
    double phase;
    double DC_offset;
} SDR_SRC_SINE;

Listing 3.3: Definition of an attribute structure for a sine wave generator.

A attribute present in all structures is the parent pointer. This value will always point to the current converter. This will serve as a convenient method to access input and output queues. See Section 3.1.3 and 3.1.4 for more details.

Listing 3.3 shows the attribute structure of a sine wave generator. This structure contains all the attributes needed to create a sine wave. When a sine wave converter is created, an instance of this structure will be created by allocating memory for it. The returned pointer will depend on the type of attribute structure. This value can now be assigned to the attribute pointer in the CONVERTER structure, which was defined as a void pointer.

    SDF[0]->attributes = (SDR_SRC_SINE*) malloc(sizeof(SDR_SRC_SINE));

Listing 3.4: Creating an instance of an attribute structure.

Each attribute structure must be accompanied by a setter function. A similar technique is used in OOP, where class constructors are used to initialize data members when an object is created from a class. The setter function will receive three sets of data. The first is a pointer to the attribute structure that must be configured. The second is the values of the attributes that must be configured. The third is a pointer to the current converter, or parent converter. Listing 3.5 provides an example of a setter function call, and listing 3.6 provides an example of a setter function.

    set_sdr_src_sine(SDF[0]->attributes, 0.001, 10, 0, 0, SDF[0]);

Listing 3.5: Calling the setter function for an attribute structure.
void set_sdr_src_sine(  SDR_SRC_SINE * sdr_src_sinePtr,
                        double xfrequency, double xamplitude,
                        double xphase, double xDC_offset,
                        CONVERTER * xparent_struct){
    sdr_src_sinePtr->frequency = xfrequency;
    sdr_src_sinePtr->amplitude = xamplitude;
    sdr_src_sinePtr->phase = xphase;
    sdr_src_sinePtr->DC_offset = xDC_offset;
    sdr_src_sinePtr->parent_struct = xparent_struct;
}

Listing 3.6: Setter function for an attribute structure.

3.1.3 The Queues

We have defined a converter as a node consisting of input ports and output ports. The
input ports are the source of data to be processed by the converter, and the output
port is the destination of processed data. Some converters may have more than one one
input or output port. For example, a basic sum converter has two input ports, and one
output port. It would therefore be impractical to define the input and output ports within
the CONVERTER structure, since the number of ports for most converters will differ. As
discussed earlier it is more convenient to declare a pointer to an array or a list, where
pointers to the individual ports are located, as illustrated in Figure 3.5.

Once a reference pointer to each of the queues have been created, memory can be
allocated for each of the queues. A design choice was made to only allocate memory for
input queues. It would be wasteful to allocate memory for both an input queue and an
output queue, because data samples that reside in an input queue, would also exist in the
output queue of the converter from where the data was obtained. This effectively doubles
the memory requirements for buffers.

Data will be consumed from an input buffer and supplied to an output buffer. We
can use a single buffer that is shared between two converters to act as both an input and
output queue. When a converter produces a data sample, it will be written to the buffer.
When the connecting converter consumes a sample, it will retrieve a data sample from
the same buffer. This then acts as a link between two converters.

Figure 3.6 shows two converters. Converter_A has a list of output queues. This list
Figure 3.5: Converter and queue layout.

contains pointers to two output queues. Converter B has a list of input queues. This list contains pointers to two input queues. In this case output_queue_1 of Converter A is connected to input_queue_0 of Converter B. Data samples will be added to the head of the queue, and removed from the tail of the queue. This is known as a first-in-first-out (FIFO) queue. FIFO queue are common in communications systems, where queues hold data packets in route to their destinations.

The samples or tokens in a communications system may range from data, signals, or control information. For this project we have chosen to implement all tokens as doubles. This was done mainly for ease of implementation, since the ANSI-C version is only a first draft implementation. However, all tokens types that may occur in queues may not be compatible with the double data type. By adapting the way converters interpret the double values in queues, other data types can be emulated.

Listing. 3.7 shows the structure declaration of QUEUE. QUEUE contains 7 elements. The two pointers, buffer_start and buffer_end, contain the addresses of the first and last elements in a queue respectively. Buffers are implemented as circular queues. In Section 3.2 it is shown that a SDF graph with consistent sample rates, can be scheduled with a finite amount of memory. For such a system the number of tokens that resides in each buffer can be determined at any given time. The maximum number of tokens that can occur in each buffer can be determined by examining the sizes of these buffers during a single iteration period. Since this is an periodic schedule, the maximum buffer size of
Figure 3.6: Two converters sharing a single queue. The second output queue of Converter A points to the first input queue of Converter B, thus linking them.

typedef struct sdr_queue {
    unsigned int size;
    double *buffer_start;
    double *buffer_end;
    double *head;
    double *tail;
    int full;
} QUEUE;

Listing 3.7: Definition of the QUEUE structure.

A node will remain constant for each iteration. As a result, a fixed amount of memory is allocated for a queue, starting at buffer_start and ending at buffer_end. This memory space will remain static, and will not grow dynamically during program execution.

It should be made clear that the QUEUE structure does not contain the actual buffer, but only information about where the buffer is located in memory.

The next set of pointers are head and tail. They are responsible for keeping track of the front and back of the queue. Samples are added (pushed) to the head of the queue, and samples are removed (popped) from the tail of the queue. The final two members are size and full. The total number of elements that the queue can support is the size of the queue. When a queue is full the flag full will be set to “1”, otherwise it
will be set to “0”. The members size and flag may seem redundant, since both values can be calculated using previous data members. By defining members to represent data that must otherwise be calculated using other variables may offer a slight performance increase.

As mentioned earlier, buffers are implemented as circular queues. A queue will only exist in a static memory space. Data is added at the head of the queue. When the head pointer gets to the end of the allocated memory space (buffer.end), it must wrap to the start address (buffer.start). Similarly, the tail pointer must wrap to the start address when it reaches the end of the buffer. Buffer overrun will occur when the tail pointer reaches the head pointer.

Figure 3.7 shows a buffer of size 8. Currently five data spots are occupied and three are free. It should be noted that head always points to a memory location containing a data sample, while tail always points to the first free memory location in the buffer. When a buffer retrieve (pop) operation occurs, the value at location head will be returned and the pointer value will be increased. Currently two pop operations can be executed before the pointer needs to be reset to the start address. When a buffer write (push) operation occurs, the given value is written to address tail and the pointer value is increased. When tail points to buffer.end, the value will first be written to the memory location and then reset to the start address.

![Figure 3.7: A typical buffer](image)

When the tail pointer reaches head, the buffer is full and no further push operations can be made. The data to be added will be lost. Precautions will be taken to ensure that this will never happen. Section 3.2 will introduce a technique to calculate the schedule for a data flow network, such as an SDR application. We will also be able to calculate the maximum number of samples that will ever exist in a queue connecting two nodes. This can be used to allocate an sufficient amount of memory for each buffer, preventing buffer overflow.
When the head pointer reaches the tail, the buffer is empty and no further pop operations can be made. Once again precautions will be taken to ensure this situation never presents itself. The schedule calculated in Section 3.2 will ensure a node will only execute when the sufficient number of data samples is available in the input buffer of that node.

When creating a new buffer the following three steps must be followed:

1. Allocate memory for the buffer.
2. Allocate memory for the QUEUE structure.
3. Set up the QUEUE structure.

\[
\text{Ptr} = \text{malloc(}\text{sizeof(double)} \times 5\text{)};
\text{SDF[0]->input_list[0]} = \text{malloc(}\text{sizeof(QUEUE)})
;\text{setQueue(SDF[0]->input_list[0],5,Ptr,Ptr+sizeof(double)*4,Ptr,Ptr);} \\
\]

Listing 3.8: Process of creating a new buffer.

Listing 3.8 provides the code format of these three steps. Memory must be allocated for the buffer itself. The size of buffers will be calculated in Section 3.2. Ptr is a temporary pointer of type double, and will point to the first element in the buffer. Memory must now be allocated for a QUEUE structure that will contain the information of the buffer. The QUEUE structure must now be configured. The setQueue() function is used for this. Six arguments will be sent to setQueue. The first is a reference pointer to the QUEUE structure that must be configured. This pointer will be used to assign values to the individual elements of QUEUE. The size attribute of the buffer is set first. The flag full is set to “0”, indicating that the buffer is not full. The buffer_start and buffer_end values must be set next. The temporary pointer, Ptr, contains the address of the first element of the queue and must be assigned to buffer_start. Since the size of the buffer is known, the address of the last element can be calculated by adding size - 1 to the Ptr value. This address is then assigned to buffer_end. Finally the head and tail pointers must be set. The buffer will be empty at this stage. Both of these pointers will therefore be assigned the address of the first element in the buffer. Listing 3.9 contains the code for the setQueue() function.
void setQueue( QUEUE* ptr, unsigned int s, double *start,  
    double *end, double *h, double *t){
    ptr->size = s;
    ptr->full = 0;
    ptr->buffer_start = start;
    ptr->buffer_end = end;
    ptr->head = h;
    ptr->tail = t;
}

Listing 3.9: The setQueue function.

3.1.4 The Process

The final element of a converter is the process function that converts input token streams into output token streams. Each definition of a converter must include a process function. Every type of converter will have its own unique process function, that will be shared by all the instances of that converter type. This is accomplished using function pointers. Listing 3.10 shows how a function pointer, called process, is declared. It receives a void pointer, state, as argument and does not return a result.

void (*process)(void *state);

Listing 3.10: Declaring a function pointer.

When a new converter is created a function must be assign to it. This means the process pointer must be assigned to a function. A function name is really the starting address in memory of the code that performs the function’s task. We then just have to assign that address to the process pointer in a converter structure. Listing 3.11 shows how this is done. A function called sdr_src_sine_process is defined somewhere in program memory. The process pointer of a converter instance is then assigned the value of the address of the function.

SDF[0]->process = sdr_src_sine_process;

Listing 3.11: Assigning a function to a process pointer.
As mentioned earlier, only a single instance of each function is declared. More than one instance of a converter using that function may be declared. Therefore, when a function is called, an argument must be passed to the function that contains the attributes of the current converter. For example, when two amplifier converters exist in the same system, they may have different gain values. They will also contain different input and output queues. The attribute structure of each of these converters contains the necessary information to inform the function which variable values to use during execution. In addition to the variable values, the attribute structure also contains a pointer to the parent node. The parent node contains the pointers to the input and output lists. This will be used by the function to identify the input and output queues of the converter. Listing 3.12 provides an example of calling a function pointer.

```c
SDF[0]->process( SDF[0]->attributes );
```

**Listing 3.12:** Calling a process with an attribute structure as argument.

The purpose of a converter is processing tokens from input ports and then writing tokens to an output port(s). Tokens are retrieved from an input port using the `popQueue()` function. The input port from which tokens are received is sent as argument to the `popQueue()` function. Recall that a list containing pointers to all input buffers is defined in `CONVERTER` (page 30). We will then only send the appropriate pointer from that list to the `popQueue()` function. The `popQueue()` function will return the value at `head`. When tokens are written to an output port the, `pushQueue()` function is used. The output port to which tokens will be written is sent as argument to `pushQueue()` along with the value to be written. The `pushQueue()` function will not return any result. Listing 3.13 shows the function prototypes for the `popQueue()` and `pushQueue()` functions.

```c
double popQueue( QUEUE *);
void pushQueue( QUEUE *, double );
```

**Listing 3.13:** Function prototypes for `popQueue()` and `pushQueue()`.

When a `process` function is called, it receives one argument, *node*. This is a pointer to an attribute structure. A process function will always receive attribute structures of the same type, since each function is specific to a converter type. Consider Listing 3.14. This is a process function for an amplifier converter. This function will always receive
pointers to attribute structures of type \texttt{SDR\_BASIC\_AMP}. Listing 3.3 contains an attribute structure for a \textit{sine wave generator}. The process function for that converter will always receive an attribute structure of type \texttt{SDR\_SRC\_SINE}.

The received attribute structure contains the necessary information about the converter executing. Not just of the type of converter, but also of the specific instance of the converter that is executing. Attributes within the structure is reached by prefixing the attribute name with \texttt{node->}. For example, the attribute \texttt{frequency} within the \textit{sine wave generator} attribute structure is retrieved using \texttt{node->frequency}. The value of an attribute can be updated simply by assigning a new value to it. For example, this would be necessary for the phase angle in a \textit{sine wave generator}.

If tokens are to be retrieved from the first input port the argument

\begin{verbatim}
node->parent_struct->input_list[0]
\end{verbatim}

will be sent to the \texttt{popQueue()} function. This is somewhat cumbersome. A temporary pointer, \texttt{parent}, of type \texttt{CONVERTER} is rather declared. The address of the parent structure is then assigned to this pointer. This creates a convenient method for accessing the input and output lists of a converter. The argument \texttt{parent->input_list[0]} can now be sent to the \texttt{popQueue()} function to retrieve tokens from the first input port. Similarly tokens can be written to the first output port by sending \texttt{parent->output_list[0]} to the function \texttt{pushQueue()} along with the value to be written.

Listing 3.14 provides an example of a complete process function of an \textit{amplifier} converter. The attribute structure is also provided. Figure 3.8 gives a visual representation of a typical converter after configuration.

### 3.1.5 Linking Converters

Once all the converters for an SDR application have been created they must be linked together. As discussed earlier, an input queue and an output queue share the same buffer. The converter containing the input queue is responsible for allocating memory for the buffer and configuring the \texttt{QUEUE} structure. A buffer now exists, and the input port of the consuming converter is connected to it. The output port of the producing converter must now be connected to the same buffer. This is done by assigning the address of the existing \texttt{QUEUE} structure to the appropriate output list entry. The output port of the producing converter and the input port of the consuming converter now both point to the same \texttt{QUEUE}. 
Figure 3.8: Diagram showing the layout of a typical converter.
typedef struct sdr_basic_amp {
    double gain;
    CONVERTER *parent_struct;
} SDR_BASIC_AMP;

void sdr_basic_amp_process(SDR_BASIC_AMP *node)
{
    CONVERTER *parent;
    parent = node -> parent_struct;

    pushQueue( parent -> output_list[0],
               popQueue(parent -> input_list[0]) * node -> gain );
}

Listing 3.14: Example of the process function of an amplifier converter.

The function `linkQueue()` is responsible for linking two ports to a single QUEUE. The function receives two sets of arguments. The first is the address of the output list of the producing converter and the index of the port in the list that must be connected. The second is the address of the input list of the consuming converter and the index in the list containing the address of the QUEUE structure. The `linkQueue()` function assigns the address of the QUEUE structure to the element at the output list index.

Listing 3.15 shows how two converters are linked to a third. The first output port of converter SDF[0] is linked to the first input port of converter SDF[2]. The first output port of converter SDF[1] is then linked to the second input port of converter SDF[2]. The converters are now linked in the configuration shown in Figure 3.9, where the sine wave is converter SDF[0], the constant is converter SDF[1] and the sum converter is SDF[2].

```
linkQueue(SDF[0] -> output_list, 0, SDF[2] -> input_list, 0);
linkQueue(SDF[1] -> output_list, 0, SDF[2] -> input_list, 1);
```

Listing 3.15: The output ports of two converters are linked to the input ports of a third.
3.1.6 Creating an SDR Application

A complete SDR application will consist of many converters. Once memory for each of the converters has been allocated as described in Listing 3.2 (page 30), each converter must be configured using the techniques discussed earlier.

The following steps must be taken when configuring a new converter:

1. Assign the process pointer to the converter.

2. Allocate memory for the attribute structure of this converter type.

3. Set the values of the attribute structure using the setter function of that structure.

4. Allocate memory for the input list that will contain pointers to all the input QUEUE structures.

5. Allocate memory for an input buffer.

6. Allocate memory for a QUEUE structure.

7. Set up the attributes of the QUEUE structure according to the allocated buffer in (5).

8. Repeat steps 5 through 7 for all input queues.

9. Allocate memory for the output list that will contain pointers to the connecting input QUEUE of other converters.

Listing A.2 on page 112 contains an example of configuring a multiply converter. This converter contains two input ports and one output port. The input buffers should be able to store at least 5 data samples. The associated function is called `sdr_basic_mul_process` and the attribute structure is called `SDR_BASIC_MUL`.

Once all the converters in the system have been created and configured they must be linked to each other to create the SDR application. This is the final step when creating an SDR application.
3.2 Static Scheduler

Once all converters in the SDR network have been configured and linked together they must execute in a such a manner that buffer over-run and under-run errors will not occur in the system. This means that converters must execute at the right time and for the correct number of times. If a sum converter is scheduled to execute before at least one data sample is available in both of its input buffers, a buffer under-run error will occur. Similarly a node can only execute if there is free space in the output port. Typically a converter will only execute until the converter depending on it has received a sufficient amount of data samples. This section will discuss how such a schedule can be calculated. We will also be able to calculate the maximum number of samples that can occur in each buffer during a single program cycle. This information will be used when allocating memory for a buffer when a converter is created and configured.

3.2.1 Defining the System

A signal processing task or telecommunications application can be broken into subtasks that execute concurrently. Such a task can be a FM demodulator as described in Figure 3.2.

Data flow programs for signal processing are directed graphs where each node represents a function and each arc represents a signal path. Synchronous Data Flow (SDF) is a special case of data flow in which the number of data samples produces or consumed by each node on each invocation is known prior to compile time [9]. The advantage of SDF is that the scheduling of nodes can be calculated at compile time since this is a static system and does not have to be done at runtime. This then removes the runtime scheduling overhead. Lee and Messerschmitt [8] propose a static scheduling algorithm for such SDF networks.

Figure 3.10 shows an SDF graph for the FM demodulator illustrated in Figure 3.2. The input and output of each node has a number next to it. The value next to the input of a node indicates the number of tokens that will be consumed by the node. Similarly the value next to the output of a node indicates the number of tokens produced by the node. For SDF graphs these values are specified a priori. It should be noted the the number of tokens produced or consumed is independent of the data. Figure 3.10 is a homogeneous SDF graph. An SDF graph where each node produces and consumes only a single token on each invocation is called a homogeneous SDF graph.
Figure 3.10: SDF graph for an FM demodulator.

Figure 3.11: An SDF graph.

Not all SDF graphs will be homogeneous. For demonstration purposes the SDF graph in Figure 3.11 has been defined. An SDF graph can be described using a topology matrix (\(\Gamma\)). A topology matrix is constructed by first numbering all nodes and arcs in the SDF graph, and assigning a column to each node and a row to each connection arc:

\[
\Gamma = \begin{bmatrix}
  a & b & c & d \\
  e & f & g & h \\
  i & j & k & l \\
  \vdots & \vdots & \vdots & \vdots \\
  \text{arc}_1 & \text{arc}_2 & \text{arc}_3 & \text{arc}_n
\end{bmatrix}
\]

The \((i,j)\)th entry in the matrix corresponds to the number of tokens produced by node \(j\), on arc \(i\). If the node produces tokens on an arc the number will be positive, and when a node consumes data on an arc the number will be negative. An arc can only be connected to two nodes. Each row in the matrix will therefore only have two non-zero entries. Connections to the outside world are ignored. When a node only has inputs from the outside, that node is considered not to have any inputs and can be scheduled at any time.

The exception to this rule is a self-loop. A self loop occurs when a node is connected to itself. This will result in a single entry in a topology matrix row. The value will be the net difference of the number of tokens produced and consumed when the node
is invoked. This value should always be zero to produce a correctly constructed SDF graph and topology matrix. If the entry is positive, the number of tokens that the node produces will be more than the number of tokens the node can consume. The number of tokens in the input buffer would then just increase indefinitely. Similarly, if the entry is negative the node will not produce enough tokens to be consumed at each invocation and buffer under run errors will occur. Section 3.2.2 will introduce a method for determining whether an SDF is constructed correctly.

The topology matrix for the SDF in Figure 3.11 is given by Equation 3.2.

\[
\Gamma = \begin{bmatrix}
1 & -1 & 0 & 0 & 0 \\
1 & 0 & -3 & 0 & 0 \\
0 & 1 & -3 & 0 & 0 \\
0 & 0 & 1 & -1 & 0 \\
0 & 0 & 0 & 1 & -2
\end{bmatrix}
\]  

(3.2)

Let’s observe the first row. This describes the samples produced and consumed on arc 1. There are entries in the first and second column. The entry in the first column corresponds to the number of tokens produced by node 1, the entry in the second column corresponds to the number of tokens consumed by node 2. When the complete topology matrix is generated, the algorithm described by Lee and Messerschmitt [8] can be followed to calculate the schedule for the SDF graph.

An admissible sequential schedule (\(\phi\)) is defined as an ordered list of nodes such that, if the nodes are executed in the sequence given by \(\phi\), the number of tokens in the buffers will remain nonnegative and bounded [9]. A periodic admissible sequential schedule (PASS) is a periodic and infinite admissible sequential schedule, where one period is defined as the nodes in the list \(\phi\). We assume that an infinite number of tokens are expected as input to the SDF network. The iteration period is the length of one cycle of a blocked schedule, and will execute periodically.

It may not always be possible to calculate a valid PASS for an SDF graph. The next section will discuss how to determine if a valid periodic schedule exists for a given SDF graph.

### 3.2.2 SDF Constraints

1. **Consistent sample rate**: Before we start our calculations we must first check if a system can be scheduled with a finite amount of memory. This is important since
a scheduling algorithm will usually be implemented on a processor with definite memory constraints. For a system to use a finite amount of memory it must have consistent sample rates. For consistent sample rates:

\[ \text{rank}(\Gamma) = s - 1 \]  

(3.3)

where \( s \) is the number of nodes in the system. This will ensure that a periodic schedule using finite memory always exists [8]. Buffer over-run errors will therefore never occur.

2. **Sufficient delays**: Directed loops within an SDF graph without a sufficient number of delays will result in a scheduling error, and must be identified by the user. This can occur even when the SDF graph has consistent sample rates. Directed loops usually occur when samples are fed back using some sort of feedback loop. Figure 3.12 contains an SDF graph that contains a directed loop. This system has consistent sample rates, however no periodic admissible sequential schedule exists. Node 1 requests a sample from node 2 to execute, however node 2 requests a sample from node 1 to execute. Therefore the directed loop must contain a sufficient amount of delays to initialise execution. This implies that the first sample consumed by the destination node will not be produced by the source node, but will be part of the initial state of the buffer.

Scheduling algorithms will always run to completion if a sequential schedule does exist, and fail otherwise [8].

![Figure 3.12: An SDF graph with consistent sample rates, but no periodic admissible sequential schedule.](image-url)
3.2.3 Sequential Scheduling

If an SDF graph has consistent sample rates, solving the nullspace of $\Gamma$ results in the positive integer vector $q$ [9].

$$\Gamma q = 0 \quad (3.4)$$

The dimension of this vector is $s$, the same as the number of nodes in the system, and indicates the number of times each node must be scheduled in one period of a PASS:

$$q = \begin{bmatrix} 6 \\ 6 \\ 2 \\ 2 \\ 1 \end{bmatrix} \quad (3.5)$$

Each entry corresponds to a node in the system as defined in the topology matrix. Entry $q(i)$ corresponds to node $i$. The order in which these nodes will be scheduled is still unknown. Each time a node is scheduled we will decrease this vector until it reaches zero when all nodes have been scheduled.

Each arc in the SDF graph can be represented as a FIFO buffer. Each node needs a buffer to store the next token it has to consume. Some nodes may need to consume more than one token to execute, therefore the number of tokens present in each buffer will vary during execution. The vector $b(n)$ is defined to represent the buffers of the nodes and the number of tokens present in each at time $n$. For the purpose of scheduling it does not matter how long each node executes, and the index $n$ can only be incremented each time a node executed and a new node starts.

Because this will be a solution for a single processor (sequential execution), only one node can execute at a time. We must therefore define another vector $v(n)$ to indicate which node is executing:

$$v(n) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.6)$$

This is a vector with $s$ entries, where each position corresponds to a node in the system. When a certain node is invoked, the corresponding entry in the vector is set to one.
For the SDF defined in Figure 3.11, the vector $v(n)$ can take on any one of five values depending on which node is executing.

When a node executes it consumes tokens from its input buffer(s), and produces tokens on its output buffer(s). The size of the buffers at any time, $n$, during execution is given by:

$$b(n + 1) = b(n) + \Gamma v(n)$$  \hspace{1cm} (3.7)

The buffer vector ($b$) is a function of the previous buffer sizes and the topology matrix ($\Gamma$). If the SDF contains directed loops the initial state of the buffers must be set before recursion of 3.7 starts. The SDF graph in Fig. 3.11 does not contain any directed loops and the initial state of the buffer vector is given by:

$$b(0) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$  \hspace{1cm} (3.8)

The algorithm described in [8] can now be used to calculate the order in which nodes must execute. For a given vector $q$, such that $\Gamma q = 0$, and an initial state for the buffers $b(0)$, a node $i$ is said to be runnable if that node has not been scheduled $q_i$ times and that running it will not cause a buffer size to become negative. This is called a class $S$ algorithm. A class $S$ algorithm is an algorithm that schedules a node if it is runnable and updates the buffer vector $b(n)$. The algorithm stops when no more nodes are runnable. If the algorithm stops before all nodes have been scheduled, deadlock has occurred.

Equation 3.9 is introduced to update the value of the vector $q$. Each time a node has been scheduled, the entry in the vector corresponding to the scheduled node must be decreased. When all the entries in the vector reach 0, scheduling is finished. A class $S$ algorithm will run to completion if a periodic admissible sequential schedule exist for an SDF graph. This also guarantees that there are no directed loops with an insufficient number of delays present in the system.

$$q(n + 1) = q(n) - v(n)$$  \hspace{1cm} (3.9)

The final class $S$ algorithm can be summarized with the following steps:

1. Solve for the smallest positive integer vector $q \in \eta(\Gamma)$. This has been done, and the result is shown in Equation 3.5.
2. Create the initial state of the buffer vector $b(0)$, with a sufficient amount of delays if any directed loops are present in the SDF graph.

3. For each node $i$, if the node is runnable and has been scheduled less that $q_i$ times, schedule the node.

4. If each node has been scheduled $q_i$ times, scheduling is finished.

5. If no nodes can be scheduled, deadlock has occurred. This indicates that an error in the SDF graph exists.

6. If $q \neq 0$, repeat steps (2) - (4).

The algorithm was implemented in Matlab and the source code can be found in Listing A.1 on page 111. The calculated schedule is $\phi = \{1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 4, 4, 5\}$.

Figure 3.13 shows a visual representation of the sizes of the various buffers during a single iteration period. It should be noted that the buffer sizes at the start and end of the iteration period are the same. This verifies that we have consistent sample rates in our system, and that buffers can be implemented using a finite amount of memory.

This information can now be used to allocate a sufficient amount of memory for the input buffers of converters to ensure that buffer over-run errors will not occur.

3.3 Conclusion

This chapter describes the detailed design of a highly reconfigurable SDR topology. The converter is introduced to represent a node in a signal processing application, along with its four main elements: input queues, output queues, attributes, and a process definition. The topology was designed in standard ANSI-C, and the next chapter will focus on porting the developed architecture to the target hardware platform.

This chapter additionally introduces a static scheduling algorithm for calculating the desired execution order of the nodes in an SDR application.
Figure 3.13: Buffer sizes for the input queues of the nodes in Figure 3.11 during a single iteration period.
Chapter 4

Hardware Implementation

4.1 Memory Management

The architecture described in Section 3.1 relies heavily on the ability to allocate memory for various data structures such as buffers, structures and lists. Memory allocation is usually handled by the host operating system (OS). The memory management scheme is responsible for allocation portions of memory to programs at their request, and for freeing memory when it is no longer used. On embedded platforms such as a digital signal processor (DSP), usually no OS is present and therefore no memory management scheme exists. Memory allocation then becomes the responsibility of the programmer.

Three popular contiguous memory allocation schemes include the first-fit, best-fit and worst-fit strategies. Contiguous memory is defined as a single continuous memory block.

1. **First fit**: Allocate memory at the first hole that is big enough to accommodate the requested memory block. Searching can start at the beginning of data memory, or at the position where the previous first fit ended (next fit).

2. **Best fit**: Search the entire memory space and allocate memory at the smallest hole that is big enough to satisfy the memory request. The option produces the smallest leftover hole.

3. **Worst fit**: Search the entire memory space and allocate memory at the largest hole. This option leaves the largest leftover hole after allocation. This may be more desirable than a small leftover hole obtained from a best fit strategy.

These memory allocation strategies each suffer from various levels of external memory fragmentation. External fragmentation occurs when there is enough total memory to
satisfy a memory allocation request, but not enough contiguous memory. This happens when memory is continuously allocated and deallocated and the memory space is broken into small memory blocks that are generally too small to be used. One solution to fragmentation is **compaction**. The goal of compaction is to shuffle allocated memory in order to group all free memory together in one large block. Compaction may not always be feasible, due to complexity and performance constraints.

A solution to external fragmentation is to divide the memory space into **fixed-size blocks**. Whenever memory allocation occurs a fixed block would be assigned. The allocated block will almost never be the exact size of the requested memory. The difference between the block allocated and the memory size requested is called **internal fragmentation**. Internal fragmentation also results in un-utilised memory, but whenever memory is freed the entire block becomes available for allocation. The fixed-size block method can be expanded by creating memory partitions each containing a series of blocks with a certain block size as illustrated in fig. 4.1. When a memory request is made the partition containing block sizes closest to the requested size is chosen. The μC/OS-II RTOS employs a **fixed-size block** memory allocation strategy [25].

Table 4.1 compares the four strategies discussed. They differ mainly in performance and level of memory fragmentation. Memory allocation will only occur during the initial topology setup. Memory allocation during waveform execution would never occur. Performance is therefore not of such a big concern. Memory fragmentation is of more im-

![Figure 4.1: Fixed-size block memory allocation strategy](image-url)
4.2 Fixed-Point Arithmetic

4.2.1 Fixed-point Representation on DSP

Fixed-point arithmetic was discussed in Section 2.3. A fixed-point value can be split up into two separate parts. The integer part and the fractional part, or rather the integer bits and the fractional bits representing the value. These two parts are enclosed within a single data frame and separated with a virtual decimal place.
typedef struct my_fract {
    _fract val;    // The unscaled value
    int scale;     // The scale factor
} MY_FRACT;

Listing 4.1: Definition of the MY_FRACT structure.

Traditionally the decimal place of a fixed-point value would be fixed at a certain position for all values during program execution, regardless of the value currently represented. However, this can limit both the range and resolution of a value. By using a separate variable to store the position of the decimal place, instead of choosing a fixed position for the decimal place, it is possible to represent a value with increased flexibility.

Two variables will therefore be needed to represent a fixed-point value. One to store the actual data bits, integer and fractional, and another to store the position of the decimal place separating the integer and fractional data. To envelope both of these variables a structure was declared housing both values. Listing 4.1 shows the declaration of the MY_FRACT structure.

The TASKING EDE supports a data type _fract with which to do fixed-point arithmetic. This data type is capable of representing values in the range \([-1; 1)\) using 24 data bits. The MSB is a sign-and-value bit, while the following 23 bits are fractional bits. A double precision fixed-point data type long _fract using 48 bits also exist. Throughout this design the single precision version will be used for ease of implementation. However, a minimal coding effort will be needed to adapt the implementation to support the double precision version. Figure 4.2 indicates how the _fract data type is represented on the DSP.

![Figure 4.2: The value \(-0.26171275\) represented using the _fract data type.](image-url)
data bits of a fixed-point value. A second value exists which indicates where the decimal place is located. By varying the position of the decimal place we can control how many integer bits and fractional bits must be used to represent a fixed point value. Consider the two values shown in Figure 4.3. Both of these values have the same bit layout, 6AC000. However by placing the decimal point at different locations we are able to represent two different values.

A) 3.3359375 Represented in Q[3][21] Format

| 2² | 2¹ | 2⁰ | 2⁻¹ | 2⁻² | 2⁻³ | 2⁻⁴ | 2⁻⁵ | 2⁻⁶ | 2⁻⁷ | 2⁻⁸ | 2⁻⁹ | 2⁻¹⁰ | 2⁻¹¹ | 2⁻¹² | 2⁻¹³ | 2⁻¹⁴ | 2⁻¹⁵ | 2⁻¹⁶ | 2⁻¹⁷ |
|----|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0  | 1  | 1  | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |

B) 53.375 Represented in Q[7][17] Format

<table>
<thead>
<tr>
<th>2⁻⁶</th>
<th>2⁻⁵</th>
<th>2⁻⁴</th>
<th>2⁻³</th>
<th>2⁻²</th>
<th>2⁻¹</th>
<th>2⁰</th>
<th>2¹</th>
<th>2²</th>
<th>2³</th>
<th>2⁴</th>
<th>2⁵</th>
<th>2⁶</th>
<th>2⁷</th>
<th>2⁸</th>
<th>2⁹</th>
<th>2¹⁰</th>
<th>2¹¹</th>
<th>2¹²</th>
<th>2¹³</th>
<th>2¹⁴</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 4.3:** Two values that have the same bit layout but representing two different values.

Since the _fract_ data type can only represent values within the range [−1; 1), floating point values outside this range must first be scaled down in order to exist within these boundaries. To scale a value to within the range [−1; 1) the value must be dived by a number larger than itself. This denominator should be a power of 2 since division can be simplified to an arithmetic shift-left which is not computationally demanding. When a value has been divided by this number, a value within the range [−1; 1) is obtained. This value can now be represented using the _fract_ data type. Consider the following example. To rescale the value 53.375 has to be divided by the first power of 2 bigger than itself, 2⁶ or 64.

\[
-1 \leq 0.833984375 = \frac{53.375}{64} < 1 \tag{4.1}
\]

Equation 2.5 specifies that 7 integer bits are needed to represent the value 53.375 using two’s complement notation. The rest of the bits will be used to represent the fractional part. The frame width of the _fract_ data type is 24 bits, which means 17 bits will be used to represent the fractional data. To keep the resolution of a value as high as possible the number of integer bits used to represent a fixed-point value will always be as small as possible while still providing the sufficient range. This will be referred to as the scale.
factor of a fixed-point value. The scale factor includes the sign-bit. Equation 4.2 can be used to retrieve the original value, using the scaled down value and the scale factor of the value, both stored in the MY_FRACT structure.

\[
\text{Original value} = \text{MY_FRACT.value} \times 2^{(\text{MY_FRACT.scale} - 1)} \tag{4.2}
\]

Figure 4.4 shows how the value 53.375 is represented using fixed-point arithmetic. In the first case only the necessary amount of integer bits are used to represent the value while in the second case there are six redundant integer bits. They serve no purpose other than decreasing the resolution of the value. In this project all fixed-point values will always be optimally scaled to maximise the resolution of the value. To scale a value, the number of redundant bits are determined, zeros for a positive value and ones for a negative value. The value is then left-shifted the appropriate amount, and the bits shifted out are discarded.

<table>
<thead>
<tr>
<th>A) 53.375 Using scale factor of 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B) 53.375 Using scale factor of 14</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2^13 2^12 2^11 2^10 2^9 2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0 1 0 1 1 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

**Figure 4.4**: A fixed point value represented using different scale factors.

### 4.2.2 Fixed-point Calculations on the DSP

Custom written functions are now needed to perform fixed-point calculations using this representation of fixed-point values. Functions are generally called with parameters of type MY_FRACT. The MY_FRACT structure contains all the needed information to reconstruct the original values. When a calculation has been done, the result is stored in a MY_FRACT structure and returned.

Four basic mathematical operations have been implemented, *addition*, *subtraction*, *multiplication* and *division*. The function prototypes can be seen in Listing 4.2. These functions receive two fixed-point values each, stored in a MY_FRACT structure. The answer
Listing 4.2: Function prototypes for _fract expressions.

is then returned in the form of a MY_FRACT structure. Five equality operations have also been implemented (Listing 4.2). Equality operations will always only return a one (true) or a zero (false).

Addition and Subtraction

When performing addition and subtraction care must be taken to first align the decimal places of the two values before the calculation starts. Consider bit 23 of Figure 4.3. In the first example the bit carries a weight of $2^1$ and in the second example the bit carries a weight of $2^6$. Since fixed-point addition and subtraction is just integer math, these bits must first be aligned to ensure the corresponding bits carry the same weight. The larger of the two values (the value with the larger scale factor) can not be adjusted to suite the scale factor of the smaller value. The value would have to be left-shifted resulting in a loss of integer bits and ultimately a change in value. This can not be allowed. The smaller of the two values will therefore be right-shifted and sign-extended until aligned with the larger value. This will result in a loss in resolution, but this will not influence the resolution of the answer since the larger value ultimately determines the final resolution.

When the values have been aligned, addition and subtraction is performed using two's compliment integer math. Special care must be taken to ensure bit overflow does not occur. When two values are added the number of integer bits needed to represent the answer may be more than the amount of integer bits used to represent the values being added or subtracted. When two values are added the maximum value the answer can
Figure 4.5: Two values that have been aligned. Each value also contains one redundant integer bit.

be will be twice that of the largest value. To represent such a value only one additional integer bit is needed (4.3):

\[ Q_{I_{\text{Answer}}} \leq Q_{I_{\text{Maxvalue}}} + 1 \]  \hspace{1cm} (4.3)

This is due to the fact that an arithmetic left-shift is equivalent to doubling a value. To ensure that an additional bit is always available should overflow occur, both values are right-shifted one additional bit before calculation. This results in one redundant integer bit in each value that will be used if carry overflow occurs.

When performing fixed-point addition or subtraction the following steps must be taken:

1. Determine the value with the largest scale factor.

2. Increase the scale factor of the largest value by one to accommodate possible carry-bit overflow.

3. Right-shift the value with the smaller scale factor until the decimal places of the two values are aligned.

4. Perform integer addition or subtraction.

5. Store answers obtained in a \texttt{MY\_FRACT} structure and return the value.

\section*{Multiplication}

When performing multiplication and division on fixed-point values the individual decimal places do not have to be aligned. The position of the decimal place in the answer must just be determined after calculation based on the scale factors of the multiplicand and multiplier. Consider the values \(A\) and \(B\) in Example 4.4. Using Equation 2.5, the scale
factor (integer bits) of each value can be calculated. The values can be reconstructed
using Equation 4.2.

\[ A = 128.0 \]
\[ a.val = 0.5 \]
\[ a.scale = 9 \]
\[ B = 25.5 \]
\[ b.val = 0.796875 \]
\[ b.scale = 6 \] (4.4)

Using basic algebra the multiplication of \( A \) and \( B \) can be simplified as follows:

\[ A \times B = 128.0 \times 25.5 = (a.2^{9-1}) (b.2^{6-1}) \]
\[ = (a.b) (2^{9-1} \cdot 2^{6-1}) \]
\[ = (a.b) (2^{15-2}) \] (4.5)

It should be noted that the power of two values are not the same as the scale factors as
calculated in Example 4.4. That is because the sign bit is included in the scale factor.
Multiplication can be done by separating the scaled down value and the scale factor of the
values. The scaled down values are multiplied, and the power of two values are summed.

When the calculation is done, the sign bit in each scale factor first has to be ignored,
thus reducing the scale factor by one. This is illustrated in Example 4.5. The exponent
value of the answer is then two less than the sum of the scale factors. Once again this
value does not include a sign bit. To adjust this back to a scale factor the value must be
increased by one to include a sign bit. The C code to perform this calculation is shown
in Listing 4.3.

\begin{verbatim}
answ = a.val*b.val;
answ_scale = a.scale + b.scale - 1;
\end{verbatim}

\textbf{Listing 4.3:} \textit{C code for performing fixed-point multiplication.}

\section*{Division}

Division is handled in a similar way. Using the same values defined in 4.4 the following
calculation can be done. Once again the calculation can be split into two separate parts.
answ = a.val / b.val;
answ_scale = a.scale - b.scale + 1;

Listing 4.4: C code for performing fixed-point division.

The scaled-down values can be divided and the scale factors can be subtracted from each other:

$$\frac{A}{B} = \frac{128.0}{25.5} = \frac{(a.2^{9-1})}{(b.2^{6-1})} = \left(\frac{a}{b}\right)2^{(9-1)-(6-1)} = \left(\frac{a}{b}\right)2^{3} \quad (4.6)$$

When performing fixed-point division there are two possible concerns. The first is when the scaled-down value of the numerator is bigger than the denominator. When division is performed a value greater than one will occur. This is not acceptable since the _fract_ data type can only accommodate values within the range $[-1; 1)$. This can be avoided by increasing the scale factor of the numerator, thus further reducing the scaled-down value. The second problem that can occur is when the scale factor of the denominator is greater than the numerator. When the exponent values are subtracted from each other, the answer will have a negative scale factor. To avoid this problem the scale factor of the numerator must be increased until it is at least one larger than that of the denominator.

When performing fixed-point division the following steps must be taken:

1. Determine whether the scaled-down value of the numerator is greater than the denominator.

2. If so, increase the scale factor of the numerator (thus reducing the scaled down value) until the scaled down value is less than the denominator.

3. Determine if the scale factor of the numerator is less than that of the denominator.

4. If so, increase the scale factor of the numerator until it is at least one more than the denominator.

5. Perform fixed-point division using the code shown in Listing 4.4.
6. Store answer in a MY_FRACT structure, and return the value.

### 4.2.3 Optimal Scale Factor

When fixed-point calculations are done, redundant integer bits may occur in the answer. The redundancy must, however, be built into the operations to prevent possible carry-bit overflow. When two values are added together it is specified that the answer must have at least one more integer bit than the largest value being added to accommodate possible overflow. This extra bit may however be redundant. Consider the 8-bit fixed-point calculations done in Figure 4.6. The larger of the two values has been optimally scaled. The decimal place of the smaller value has been aligned with the first. To accommodate possible carry-bit overflow the answer must contain an extra integer bit. In this case the extra integer bit is redundant however. Since the answer does not need the redundant bit the value can be left-shifted one bit. The value is now optimally scaled. As mentioned previously, all values in the system will always be optimally scaled, i.e. no redundant integer bits. This results in an increase in the resolution of values.

![Figure 4.6: Addition operation resulting in an answer containing redundant integer bits.](image)

To ensure all values are always optimally scaled, all redundant bits will be discarded after a fixed-point calculation has been done. A rescale function was created that receives a MY_FRACT structure containing possible redundant integer bits. Redundant bits are identified and discarded. The optimally scaled value is then returned in a MY_FRACT structure.

Consider the answer in Figure 4.6. The most significant bit (MSB) is a zero. This is the sign bit and indicates that the value is positive. The next bit is also a zero and is redundant. By left-shifting the entire value and decreasing the scale factor by one the numerical value is retained and the redundant bit is discarded.
The following steps are taken when identifying and discarding redundant bits of a fixed-point value:

1. Determine if the value is positive or negative by identifying the MSB.
2. Determine the number of times ($N$) the MSB (sign bit) repeats itself.
3. Left-shift the value $N$ times.
4. Update the scale factor of the value by subtracting $N$.
5. Return the value in a MY_FRACT structure.

### 4.2.4 Rescale

Five equality operations have been implemented (Listing 4.2). Our representation of fixed-point values prohibits us from simply comparing two values. This is due to the fact that each value has a scale factor. Before two values can be compared their decimal places must first be aligned. The value with the smaller scale factor will be right-shifted until aligned with the value containing the larger scale factor. Once the values are aligned they can be evaluated using normal equality operators.

### 4.2.5 Merge with SDR Topology

The topology described in Chapter 3 is implemented using floating-point data types such as double. The topology will have to be adjusted to support fixed-point data types. A converter consists of input queues, output queues, attributes and a process. These segments must all support fixed-point data types.

**Queues**

We have previously specified that buffers will only contain real data, and each element in a buffer was defined as a double. This must be converted to a fixed-point equivalent implementation. Two designs have been implemented, each with a distinct advantage and disadvantage. Depending on the specific SDR application the most appropriate method can be chosen and implemented at compile time.
typedef struct sdr_queue {
    unsigned int size;
    _fract *buffer_start;
    _fract *buffer_end;
    _fract *head;
    _fract *tail;
    int full;
    int scale;
} QUEUE;

Listing 4.5: QUEUE implementation using a single scale factor for each element in the buffer.

Listing 4.5 shows the QUEUE structure for the first method. In this case each element in the buffer will be of type _fract. An additional attribute is needed to specify the scale factor of these values. The scale attribute was added for this reason (Listing 4.5).

All values in a buffer will now have the same scale factor. This means that values will have to be rescaled before they are pushed into the buffer if they have a different scale factor. The rescale() function will be used to rescale a value to the desired scale factor. The scale factor of the connecting input queue must however first be retrieved as shown in Listing 4.6.

CONVERTER *parent;
QUEUE *output;
int scale_to_be_0;

parent = node->parent_struct;
output = parent->output_list[0];
scale_to_be_0 = output->scale;

pushQueue(parent->output_list[0], rescale(value_a, value_b, scale_to_be_0));

Listing 4.6: When a value is sent to the input queue of a connecting node, the value must first be rescaled to be compatible with the scale factor of the queue, as illustrated here.
The disadvantages of this technique are:

1. Extra processing power is required to rescale a value before it is pushed into a queue.

2. Reduction in the resolution of a value.

3. Reduced dynamic range of values. The scale factor of a queue must be sufficiently large to support the range of the largest value expected in the buffer. Values that are much smaller must also be represented using this scale factor, which will result in redundant integer bits and therefore a loss in resolution.

4. Possible discarding of integer bits if the scale factor of a buffer is not sufficiently large.

5. Prior knowledge of the SDR application and the range of values expected is needed when specifying the scale factor for a buffer. Since the ultimate goal of this project is the automatic generation of a high-level language, this will not necessarily be a disadvantage. However, this may result in more complex code translation.

The advantage of this method however is a reduced buffer size since only a single scale factor has to be stored for a collection of values.

The second method stores the scale factors of each value in the buffer. Each element in the buffer is of type `MY_FRACT` (Listing 4.7). This technique has several advantages:

1. Increased resolution since each value contains its own scale factor and can therefore be optimally scaled.

2. Increased dynamic range.

3. Reduced processing power since values can just be pushed into the buffer without adapting the scale factor.

4. Prior knowledge of the SDR application and the range of values expected are not needed.

The disadvantage of this method is an increase in memory usage since a scale factor must be stored for each value. This may be undesirable, especially when large buffers are implemented.
typedef struct sdr_queue {
    unsigned int size;
    MY_FRACT *buffer_start;
    MY_FRACT *buffer_end;
    MY_FRACT *head;
    MY_FRACT *tail;
    int full;
} QUEUE;

Listing 4.7: QUEUE implementation using individual scale factors for each element in the buffer.

A buffer implementation must be chosen depending on the specific SDR application. Currently the first method has been implemented due to ease of implementation. It is recommended to implement the second method if values with a large dynamic range are expected within the system.

Attributes

If a converter contains a floating-point attribute such as a double or float, that attribute must be substituted with a fixed-point equivalent variable. It was decided to replace each floating-point variable with a MY_FRACT variable. All attributes will therefore have their own scale factor. Each attribute will also be optimally scaled, thus maximizing its resolution. Since each value has its own scale factor, values in a large dynamic range can be expressed. This means small values and large values can be stored while still maximizing its resolution. This will increase computational accuracy.

typedef struct sdr_src_sine {
    MY_FRACT frequency;
    MY_FRACT amplitude;
    MY_FRACT phase;
    MY_FRACT DC_offset;
    CONVERTER *parent_struct;
} SDR_SRC_SINE;

Listing 4.8: An attribute structure implemented using the MY_FRACT data type to represent fixed-point variables.
Process

The process definition of a converter is a function containing code which specifies how to convert input tokens to output tokens. These functions are written in a high-level language, in our case C. When a converter contains floating-point variables, expressions containing these variables will most probably exist in the process definition. These expressions are not compatible with our fixed-point processor. Expressions containing even a single floating-point variable must therefore be substituted with fixed-point equivalent expressions. Section 4.2.2 introduced functions capable of performing fixed-point calculations. Section 5.3 will describe how process definitions containing floating-point expressions are evaluated and translated to fixed-point equivalent expressions incorporating these functions.

4.3 ADC and DAC

Thus far we have only considered SDR applications that have no connection to the outside world. However, practical SDR applications will have to receive and process analogue signals. The analogue-to-digital converter (ADC) and digital-to-analogue converter (DAC) modules are introduced to perform these tasks.

4.3.1 CODEC and CODEC Configuration

The CS4218 Stereo Audio Codec (CODEC) [31], located on the DSP56311EVM, will be used to perform digital-to-analogue (D/A) and analogue-to-digital (A/D) conversion. The DSP will be able to receive data from the CODEC, process the data, and transmit the data back to the CODEC. The DSP will communicate with the CODEC via the DSP’s two synchronous serial interfaces, ESSI0 and ESSI1.

The CODEC has several modes of serial operation, but is physically wired to function in serial mode 4 (SM4) on the DSP56311EVM. When operating in SM4, the audio data is separated from the control data for the CODEC. Audio data will be sent via the ESSI0 port to and from the CODEC, and control data will be sent via the ESSI1 port as illustrated in Figure 4.7.

SM4 can be further configured by setting the secondary sub-mode pins of the CODEC. The sub-modes specify whether the CODEC operates in master or slave mode, and the number of bits per frame. For the DSP56311EVM the secondary modes are physically
configured to sub-mode 0, indicating that the CODEC operates in master mode with a frame size of 32 bits. Sub-mode 0 further specifies that each frame contains two 16-bit words. The left channel 16-bit word and the right channel 16-bit word is sent to and from the CODEC with the most significant bit (MSB) first (Figure 4.8).

Before data transfer can occur the CODEC must first be configured. This is done by sending control information via the ESSI1 port of the DSP. For further information about the configuration of the CODEC, refer to Appendix A.1.
4.3.2 Data Transfer

Audio data is sent via the ESSI0 port to and from the DSP. The ESSI interface has been configured to operate in synchronous network mode. Synchronous mode specifies that the transmitter and receiver use common clock and synchronization signals. In network mode, two data words will be received or transmitted during each frame, to successfully interface with the CODEC. It is further specified that the 32-bit frame will be comprised of two 16-bit words, where each 16-bit value is left aligned to bit 23 to represent a two's complement value.

Reception

Whenever data is received on the ESSI0 port, a receive interrupt will be generated. As shown in Figure 4.8, the left channel will be received first, then the right channel. When the first 16-bit frame has been received an ESSI Receive Data Interrupt will be generated, indicating that the left channel may be read. A buffer with two data slots, RX_Buffer, was created to store the left and right channels of a received data frame (Listing 4.9). The received data will be transferred to RX_Buffer[0]. This will also clear the ESSI receive register. When the second 16-bit frame has been received an ESSI Receive Data Interrupt will again be generated, indicating that the right channel may be read. The received data will be transferred to RX_Buffer[1], again clearing the ESSI receive register. An ESSI Receive Last Slot Interrupt will then be generated, indicating that the entire 32-bit frame has been successfully received.

Listing 4.9: Declaration of the receive and transmit buffers of the ADC and DAC modules.

```
_fract RX_Buffer[2];
_fract TX_Buffer[2];
```

The received signals can now be easily accessed by the ADC module. Received data will be left-aligned with bit 23. The CODEC uses two’s compliment representation and we can therefore store received data in a _fract variable with an accompanying scale factor of “1”.
Transmission

Transmission is handled in a similar fashion. The left channel will be transmitted first, then the right. A buffer with two data slots, \texttt{TX\_Buffer}, was created to store data to be transmitted to the CODEC (Listing 4.9). \texttt{TX\_Buffer}[0] will hold the left channel and \texttt{TX\_Buffer}[1] will hold the right channel. During program execution whenever a data sample is to be transmitted by the DAC module, that value should be written to the appropriate position in \texttt{TX\_Buffer}.

A \textit{ESSI Transmit Data Interrupt} will be generated when data is to be transmitted and the transmit register is empty. The contents of \texttt{TX\_Buffer}[0] will be written to the transmit register, and the ESSI port will transmit the word. A \textit{ESSI Transmit Data Interrupt} will again be generated when data is to be transmitted and the transmit register is empty. The contents of \texttt{TX\_Buffer}[1] will then be written to the transmit register, and the ESSI port will transmit the second word. The \textit{ESSI Transmit Last Slot Interrupt} will be generated when transmission of the last word is about to start, indicating that this frame has successfully been serviced.

Synchronization

To synchronize the DSP and the CODEC, the CODEC will generate a synchronization pulse (SSYNC) whenever a new data frame is about to start, as illustrated in Figure 4.8. Thus, whenever a SYNNC pulse has been received it would indicate that the previous data frame has been fully received, and transferred to \texttt{RX\_Buffer}. To ensure that a new data sample is always in the \texttt{RX\_Buffer}, a \texttt{Wait\_Frame\_Sync()} function was created. This function will pause program execution until a SSYNC pulse has been detected. When the pulse is detected data can be read from \texttt{RX\_Buffer}, and normal program execution can resume.

When program execution reaches the end of an iteration cycle, the processed data must usually be transmitted via the ESSI port to the CODEC. Processed data will be transferred to the \texttt{TX\_Buffer} as soon as they become available. The program will then start execution from the beginning where it will once again wait for a frame SSYNC pulse. When the SSYNC pulse has been received, data in the \texttt{TX\_Buffer} will be transmitted to the CODEC (Figure 4.8).
4.4 Conclusion

This chapter describes how to port the SDR architecture developed in Chapter 3 to be compatible with the target hardware platform for this project, the Motorola DSP56311. Issues such as memory management and numeric representation are discussed. The main focus of this chapter is describing how floating-point values can be represented using the fixed-point arithmetic supported by the hardware platform. Furthermore, methods for performing typical mathematical operations are introduced.

Finally the chapter concludes by introducing the ADC and DAC modules used to connect an SDR application to the outside world.
Chapter 5

Translation Layers

Chapters 3 and 4 develops and implements a platform specific SDR architecture. This architecture is implemented using the C language, and an extensive knowledge of the target hardware platform is required to create new SDR applications when using this architecture. Users would first need to gain a thorough understanding of the architecture’s surrounding code before implementing new SDR applications. The need exists to separate the *SDR application design* from the *boilerplate system code*.

This chapter addresses this issue by specifying SDR applications using generic DSL modules introduced in Section 2.4.1. A domain specific language (DSL) translator can then generate the desired high-level language (HLL) supported by the underlying SDR architecture. Various levels of translation exist and each will be discussed in detail during this chapter.

5.1 XML and XSLT

5.1.1 Creating an SDR Application

In Chapter 2 generic DSL modules were introduced that were mostly hardware independent. DSL modules were specified in XML, and used to define converters. These converter definitions will now be used as a foundation for creating SDR applications. A complete XML definition of an SDR application will consist of the following subdivisions:

1. *Importing Converter Definitions*. This section is responsible for importing all XML converter definitions that will be used by the SDR application. The imported converter definitions are enclosed within the `<ConverterDefinitions>` element.
2. **Node Declarations and Configuration.** The declaration and configuration of each converter instance that will make up the SDR application will be listed in this section. An instance of a converter will from here on be referred to as a *node*. All nodes will be listed in the `<Nodes>` element.

3. **Node Link Definitions.** When all nodes have been declared and configured, the connections between nodes will be defined in this section. All node link definitions will be included in the `<Links>` element.

4. **Sequential Schedule.** The order in which nodes must execute will be defined in this section. The schedule will be listed in the `<Schedule>` element.

The root element of a complete SDR XML definition is `<System>` (Listing 5.1). The above-mentioned sections will be enclosed within this root element. Each of these sections will now be discussed in more detail.

```xml
<System>
  <ConverterDefinitions>
  </ConverterDefinitions>
  <Nodes>
  </Nodes>
  <Links>
  </Links>
  <Schedule>
  </Schedule>
</System>
```

**Listing 5.1:** *Subdivisions of a complete SDR XML definition, enclosed within the `<System>` element.*

**Converter Definitions**

Before a node can be declared and configured, each converter type that will be used in the application must be imported from an XML library. An XML library is a collection of generic converter definitions that can be used to create SDR applications which can then be translated to various hardware platforms. When all converter types have been imported the individual converter instances that will make up the SDR application can
be declared and configured. Consider the trivial example in Figure 5.1. This SDR system consists of five base converter types:

- Analogue-to-Digital Converter
- Sine-wave Generator
- Multiplier
- Amplifier
- Digital-to-Analogue Converter

Each unique converter present in the application must be imported before separate instances can be declared and configured. The ENTITY type attribute can be used to link external data into an XML document. In this case the external data is an XML document containing a converter definition. The value of the ENTITY attribute is the name of an unparsed general entity which links to the external data. To link the entity to the external data, the SYSTEM keyword and a URI to the data is used:

```xml
ENTITY amplifier SYSTEM "xml_lib/sdr_basic_gain.xml"
```

This ENTITY declaration associates the name amplifier with the converter definition at “xml_lib/sdr_basic_gain.xml”. The general entity reference &amplifier can now be placed within the XML document and the parser will replace it with the contents of the external
data. When using external parsed entities the external document should ideally be well formed. All elements that start inside the entity must therefore finish inside the same entity.

All external XML converter definitions will be imported using ENTITY declarations. XML converter definitions will first be associated with a general entity reference. The external data will then be inserted into the main document using the entity references. External entities will be enclosed within the <ConverterDefinitions> element.

Node Declaration and Configuration

Once all converters have been imported the individual node instances that will make up the SDR application can be declared and configured. A new node is declared by adding a <component> element within the <Nodes> element. A <component> element represents a node in an SDR application, and must be configured to represent a converter that will perform a certain task. Each declared <component> must be assigned a converter type. The converter type must be one of those imported in the <ConverterDefinitions> element. Each <component> must have a name unique within the SDR application. Line one of Listing 5.2 shows how the type and name attributes are set when declaring an amplifier node.

When a new node has been declared, the attributes, if any, of this instance of the converter must be set. This is done using the <set> element. The <set> element has an attribute called <name>, used to identify the converter attribute that must be set. The attribute will be set to the value contained within the <set> element. Line two of Listing 5.2 shows how the gain attribute of an amplifier is set to “10”.

Finally the buffer size and scale factor of each input queue must be set. The <buffer> element is used for this purpose and has three attributes: name, size and scale. The name attribute is used to identify the the specific input port of the node that must be configured. The maximum buffer size is assigned to size and the scale factor of the queue is assigned to scale. Line three of Listing 5.2 shows how the input queue named Input of an amplifier converter is configured.

Node Link Definitions

When all nodes of an SDR application have been declared and configured the nodes have to be linked to each other. The output queue of one node must thus be connected to the
Listing 5.2: Component declaration and configuration of an amplifier node.

input queue of another node. The <link> element is used for this. A <link> element contains four attributes. The output queue to be linked is specified using the from and from_port attributes. The node containing the output queue is specified using the from attribute and the specific output queue within the node is specified with the from_port attribute. The input queue is identified similarly using the to and to_port attributes. Listing 5.3 shows the XML syntax for linking two nodes.

Listing 5.3: Links definition between two nodes.

Sequential Schedule

When all nodes have been declared and linked together it represents an SDF graph, and a schedule can be calculated using the method described in Section 3.2. The calculated order in which nodes execute must be defined in this section. This section does not necessarily need to be defined by hand, and can be the result of machine translation. A complete XML definition of an SDR application may need to pass through several layers of code translation before the final HLL version is completed.

To add a node to the scheduling order a <Node> element is added. The <Node> element has a single attribute, name. The name of the node to be scheduled is assigned to name. Listing 5.4 gives an example of the proposed schedule for an SDR application that reads a sample from an ADC. The sample is then amplified and written to a DAC.
Listing 5.4: Proposed schedule for a simple amplifier system listed in XML format.

```
<Node name="name_ad_converter" />
<Node name="name_amplifier" />
<Node name="name_da_converter" />
```

**Figure 5.2:** Visual representation of a complete XML definition of the SDR application illustrated in Figure 5.1.
5.2 XSLT Transformation

When a complete XML definition of an SDR application has been created, XSL transformations will be used to generate the corresponding C code compatible with the DSP56311. A complete SDR definition is comprised of four sections, and each of these sections will be used to generate a specific segment of the C code. It was chosen to implement the final C program in two separate files. One containing the definitions of all converters to be used in the applications, and the other containing the declaration and configuration of the individual converter instances in the application.

The definition of converters, including the converter-specific attribute structures, attribute setter functions, and converter process definitions, will be implemented in the file “sdr.h”. The declaration of individual converter instances (nodes), configuration of the nodes, linking of nodes, and declaration of the sequential execution order of nodes will be implemented in the file “sdr.c”.

A complete definition of an SDR application consists of four segments. Each of these segments will serve as the primary resource for generating a specific section of the HLL implementation. The four segments are:

1. **Converter Definitions.** This segment will be used to generate the attribute structures of all converters, and the attribute setter functions of these converters. It will also be used to generate the process functions of all converters. These transformations will be implemented in the XSLT stylesheet “sdr.h.xsl”, and will be responsible for generating the file “sdr.h”.

2. **Node Declaration and Configuration.** This segment will be used to generate the C code that will declare and configure nodes that are present in the SDR application. These transformations will be implemented in the XSLT stylesheet “sdr.c.xsl”, and will produce the file “sdr.c”.

3. **Node Link Definitions.** This segment will be used to generate the C code that links the input and output queues of nodes. Link definitions will follow directly after the declaration and configuration of nodes, and the transformations will also be located in the “sdr.c.xsl” stylesheet. Output text will consequently be appended to the “sdr.c” file.

4. **Sequential Schedule.** The last component of an SDR application definition is the sequential execution schedule. This section will directly follow the node link defini-
The SDR XML Definition will be transformed into the file “sdr.c”. This transformation will be located in the “sdr.c.xsl” stylesheet.

Figure 5.3 gives an overview of the XSLT translation layer. A single XML SDR definition exists that will generate two files containing the HLL implementation of the SDR application. The XSLT translation process will be responsible for generating the base C code, implementing the architecture designed in Chapter 3 and 4. Chapter 4 also focused on the representation of fixed-point arithmetic. Although the XSLT translation layer will not be responsible for converting floating-point expressions to fixed-point expressions, it will be responsible for marking the code to be translated by the lexical analysis translator. It will also generate support code that will help with the lexical analysis translation layer.

We will now look at each of the translation segments in more detail.

### 5.2.1 Converter Definitions Transform

As mentioned earlier this segment is responsible for generating the converter specific attribute structure, attribute setter function, and process function of all converter types
present in the SDR application.

This stylesheet will be applied to the XML definition of a complete SDR application. This segment will therefore only match the `<ConverterDefinitions>` element, and apply the template to that element. Included in the `<ConverterDefinitions>` element are all the converter definitions. Each converter definition is encapsulated in a `<converter>` element, and the `<xsl:for-each>` element will be used to match and process each converter definition individually. Before the transform is started, a `<xsl:variable>` called `name` is created and assigned the value of the current converter name to be used throughout the transform.

The first segment that will be generated is the converter specific attribute structure of the current converter. Attributes defined in the converter will be identified and placed within the attribute structure. Attributes with a floating point data type will be replaced by a `MY_FRACT` equivalent version to resemble an attribute structure as shown in Listing 4.8. The XSLT transform for identifying and copying floating-point attributes is shown in Listing 5.5.

```c
typedef struct <xsl:value-of select="$/name"/> { 
    <xsl:for-each select="attribute|constant">
        <xsl:if test="@type='double'">
            MY_FRACT __f_<xsl:value-of select="./name"/>>;
        </xsl:if>
    </xsl:for-each>
    CONVERTER *parent_struct;
}<xsl:value-of select="translate($name,& LOWER_TO_UPPER ;)"/>;
```

**Listing 5.5:** Extract of the XSLT transform that identifies attributes of type double in a XML converter definition, and adds them to the attribute structure for the current converter.

To visually distinguish between floating-point attributes and integer attributes, the names of floating-point attributes will be prefixed with `'__f_'`, and integer attributes with `'__i_'`. This does not influence the XSLT translation layer in anyway, and will only be exploited in Section 5.3 during lexical analysis.

The next stage of translation involves the generation of an attribute setter function. The transform is created to generate a setter function resembling the example shown in Listing 3.6. However, since floating-point attributes have been replaced by fixed-point
Chapter 5 — Translation Layers

void set_<xsl:value-of select="$name"/>(
<xsl:value-of select="translate($name,& LOWER_TO_UPPER ;)"/>*<xsl:value-of select="$name"/>Ptr,
<xsl:for-each select="attribute|constant">
  <xsl:if test="@type='double'">
    _fract <xsl:value-of select="/name"/>,
    int <xsl:value-of select="/name"/>_scale,
  </xsl:if>
</xsl:for-each>CONVERTER *parent_struct)
{
  <xsl:for-each select="attribute|constant">
    <xsl:if test="@type='double'">
      <xsl:value-of select="$name"/>Ptr-&gt;__f_<xsl:value-of select="/name"/>_.val = <xsl:value-of select="/name"/>;
      <xsl:value-of select="$name"/>Ptr-&gt;__f_<xsl:value-of select="/name"/>_.scale = <xsl:value-of select="/name"/>_scale;
    </xsl:if>
  </xsl:for-each>
  <xsl:value-of select="$name"/>Ptr-&gt;parent_struct = parent_struct;
}

Listing 5.6: XSLT transform for generating a setter function that will initialize fixed-point values.

equivalent attributes, the setter function will have to set both the fixed-point value and
the scale factor of that value. Listing 5.6 shows the XSLT transform that will identify
floating-point attributes, and generate code that will set the value and scale of the at-
tribute separately. The "sdr.c.xsl" stylesheet will be responsible for generating the setter
function call and the accompanying arguments needed by the setter function to initialize
an attribute structure. This will be discussed in Section 5.2.2.

The final stage of translation will be responsible for identifying and copying the process
definition of a converter to the output file. The transform will first check if the code
segment is written in the ANSI-C format by testing the syntax attribute of the process
element. The contents of code element, within the process element, will then be copied to
the output document as illustrated in Listing 5.7. As discussed earlier, the code segment of
a converter definition may contain floating-point expressions that have to be translated to
fixed-point equivalent expressions. This transform will mark code segments by enclosing
them within #pragma CODE_START and #pragma CODE_END. These markers will be used by
Chapter 5 — Translation Layers

Listing 5.7: XSLT transform for selecting and copying the process definition to the output document.

the lexical analyzer to identify code segments to be translated to fixed-point expressions. As discussed in Section 4.2.5, tokens sent to the input queues of connecting nodes might need to be rescaled to be compatible with the scale factor of the queue. This issue has been addressed, and a solution is given in Listing 4.6. This segment of the translation will also be responsible for generating code that retrieves the scale factor(s) of the connecting input queue(s). The code segment defined in the converter definition will not currently use these retrieved scale factors. However, the lexical analysis translation layer will convert floating-point expressions into fixed-point expressions that will use the scale factors (See Section 5.3).

5.2.2 Node Declarations and Configuration Transform

This segment is the first of three transforms in the “sdr.c.xsl” stylesheet. This transform will be responsible for determining the number of nodes in the SDR application, declaring empty node instances for each node in the application, and then finally configuring each of these nodes.

The number of nodes in the system is determined by first matching the <Nodes> element within the <System> element. The number of <component> elements within the <Nodes> element can then be counted using the using the last() function as shown in Listing 5.8. This transform will then generate code that will declare a list of CONVERTER pointers. Each of these pointers are then assigned to empty CONVERTER instances to
<xsl:for-each select="*/component">
    <xsl:if test="position()=1">
        CONVERTER* SDF[<xsl:value-of select="last()"/>];
    </xsl:if>
</xsl:for-each>

<xsl:for-each select="*/component">
    SDF[<xsl:value-of select="position() - 1"/>] = (CONVERTER *)dspmalloc(sizeof(CONVERTER));
</xsl:for-each>

Listing 5.8: XSLT transform for determining the number of nodes in a SDR application, generating a list of CONVERTER pointers, and initializing these pointers.

resemble the code found in Listing 3.2.

The next stage of the transform involves the configuration of each node in the SDR application. This is done by processing each <component> element within the <Nodes> element. The generated code must adhere to the format and order described in Section 3.1.6. This will include the following:

1. Determining the position of the current <component> element being processed. This position will correspond to an array index in the previously defined CONVERTER list, and the configuration process to follow will therefore be done on that empty CONVERTER instance.

2. This transform will generate the function call for the setter function of the attribute structure of this node. Attribute names and values will be read iteratively from the <component> element, and matched in the order they appear in the converter definition. Attributes are matched in the order they appear in the converter definition since the setter function was generated from this definition. When setting fixed-point values, the value and the scale factor is set independently. Floating-point values must therefore first be converter to fixed-point equivalent versions, by calculating the fixed-point value and the scale factor. However, the lexical analyzer will be responsible for this translation. The XSLT transform will only copy the original floating-point value to the output file, and mark it to be processed by the lexical analyzer at a later stage. The values to be converted by the lexical analyzer will be
marked by enclosing them within `#pragma CODE_START` and `#pragma CODE_END`.

3. The next stage of the transformation is responsible for determining the number of input queues of the node, generating the code to allocate memory for them, and then configuring each of these queues by generating a function call for `setQueue()`.
   
   This is done by examining the `<buffer>` element within the `<component>` element. The `<buffer>` element contains information about the size and scale factor of the queue, and should be retrieved to generate code resembling that found in Listing 3.8.

4. The final component of the transform will determine the number of output queues of the node, and generate the code that will allocate memory for the output list. Output queues do not have to been configured as with input queues, since they will just be linked to input queues at a later stage.

Once all nodes in the system have been configured, the input and output queues of these nodes must be linked together to create a complete SDR application. This will be the task of the next transform.

### 5.2.3 Node Link Definitions Transform

The `<Links>` element within the `<System>` element contains information about how the input and output ports of nodes are connected. This transform will be responsible for retrieving that information and generating the code found in Listing 3.15. This is done by first determining the index of the `from_node` in the CONVERTER list by evaluating the names of the `<component>` elements. Next, the index of the `from_port` in the output list of the node is determined by evaluating the converter definition of that node. A similar method is followed to determine the index of both the `to_node` and the `to_port`.

### 5.2.4 Schedule Transform

The function of the last XSLT transform is to examine the `<Schedule>` element within `<System>`, and generate the function calls for each node in the correct order. As with the previous transform, the position that the node occupies in the CONVERTER list must first be determined. The transform will then generate the function calls, and produce the code found in Listing 3.12.
5.2.5 XSLT Translation Layer Conclusion

The set of XSLT transforms in this section was responsible for synthesizing C code from an XML specification that is compatible with the DSP563xx series of DSPs. The generated code implements the reconfigurable architecture designed in Chapters 3 and 4. The issue of converting floating-point expression to fixed-point equivalent expressions have not been fully addressed. However, the transform has identified and marked the code to be converted, and the next translation layer will be responsible for the conversion.

5.3 Lexical Analysis

The target platform for this project is the DSP56311, a fixed-point digital signal processor. Mathematical operations must therefore be done using fixed-point arithmetic. However, the code snippets contained within the `<process>` elements of converter definitions are specified using ANSI-C and will contain floating-point expressions. Since XSLT is not a text or mathematical parser, it was decided to implement an additional translation layer that will be responsible for recognizing floating-point expression and convert them to fixed-point equivalent ones.

A compiler was created for this purpose. A compiler is a computer program that translates text written in a computer language, to another computer language. In this case, standard ANSI-C code will be translated to fixed-point compatible C code. This is also known as a language rewriter, since it changes the form of expressions without changing the language.

In Section 4.2 we introduced several mathematical functions that could perform fixed-point calculations. The primary task of the compiler will be to generate C code that will use these functions to perform standard mathematical calculations. Figure 5.4 shows how standard mathematical expression will be converted to fixed-point expression by implementing the functions introduced in Section 4.2.

5.3.1 Generating a Compiler

To create a compiler, the GNU programming tools *Flex* and *Bison* were used. *Flex* (Fast Lexical Analyzer) is a tool for generating scanners. A scanner is a program which recognizes lexical patterns in text [21]. These patterns, called *rules*, are described using pairs of regular expressions and accompanying C code. The scanner analyses its input,
Figure 5.4: Examples of how floating-point expressions will be converted to fixed-point expressions.

searching for occurrences of the regular expressions. Whenever a match is found the corresponding C code is executed. Flex generates a C file called “lex.yy.c” by default, which defines a scanning routine yylex().

One of the main uses of Flex is to serve as a scanner companion for the parser generator, Bison. A parser is a program or program part that analyses a sequence of tokens using a set of grammatical rules to create a structure for further processing. Bison is a general-purpose parser generator that converts context-free grammars into a parser for that grammar [22]. A common formal system for describing such grammar rules in a human readable manner is the Backus-Naur Form (BNF). For each grammar rule an action is specified that must execute when that rule is matched. The action is specified by a sequence of C statements. The Bison parser reads a sequence of tokens as its input, and groups the tokens using the grammar rules. If the input is matched to a grammar rule, the corresponding rule is executed. Bison generates a file called “file.tab.c”, which defines the parsing routine yyparse().

The sequence of input tokens is received from the accompanying scanner. Each time a new token is needed by the parser, the lexical analyzer function, yylex(), is called. The yylex() routine is supposed to return the type of the next token in the input stream. Bison can be instructed to generate the support file called “file.tab.h”, which includes the definitions of all tokens in the formal grammar. This file is then included in the Flex scanner to be compatible with the generated parser. Figure 5.5 provides and overview of the Flex and Bison relationship.

The next section will describe how the scanner component of the compiler was created,
and how the scanner was adapted to successfully interface with the parser.

### 5.3.2 Generating a Flex Scanner

To generate a scanner, Flex reads an input file specifying the behavior of the scanner. The input file consists of three sections, as shown in Listing 5.9. When the Flex generated scanner routine `yylex()` is called for the first time, it will read text from the specified input stream and place the text in an input buffer. Tokens will be retrieved from this buffer until empty, when the buffer will be filled again. The scanner will retrieve tokens until a end-of-file (EOF) character is read, or when one of its actions specify a `return` statement. When `yylex()` is called again after stopping due to a `return` statement, scanning will continue from where it left off. Once a match is established, the text corresponding to the matched token is made available in the global character pointer `yytext`.

```plaintext
definitions
%%
rules
%%
user code
```

**Listing 5.9:** The overall layout a Flex input file.

As stated before, a Flex input file contains three sections. The *definitions section* of the input file contains name definitions to simplify the scanner specification. Name
definitions will have the form:

    name definition

The definition defines a regular expression, that can subsequently be referred to as name at a later stage. Name definitions significantly simplifies pattern definitions in the rules section to follow. For example, the regular expression:

    REAL [0-9]+"."[0-9]*

defines a sequence called REAL, which will match one or more digits ([0-9]+), followed by a single decimal point ("."), followed by zero or more digits ([0-9]*). This name definition can now be used in the rules section to match a variety of patterns that may contain a real value. Code to be copied verbatim to the generated file can also be specified in the definitions section by enclosing them in ‘%{…%}’. This may include the definition of external libraries, or the declaration of support variables to be used by the scanner.

The rules section will contain a series of rules specifying the patterns the scanner should match, and the actions that should be taken when a pattern is matched. A rule will be in the form:

    pattern action

Patterns are specified using an extended set of regular expressions. The Flex Manual should be consulted for an in depth look at the supported set of regular expressions [21]. For example, the regular expressions:

    [0-9]+"."[0-9]* { return FRACT; }

or

    REAL { return FRACT; }

will specify a rule that matches real data, and returns a token to the program that called it, in this case the Bison parser routine yyparse(). FRACT is a token value defined by the parser, and is defined in the support file generated by Bison called “file.tab.h”. This file will be included in the definitions section, allowing the scanner to interface with the Bison generated parser. As mentioned earlier, the matched text will appear in the buffer yytext. It is the responsibility of the scanner to return the semantic value of the token to the parser along with the token type. For this reason a copy of the data in yytext
is made, and stored in a buffer accessible by the parser routine. Upon returning to the parser routine, the matched text can be retrieved and used by the parser.

The *code section* of the specification will be copied verbatim to the output file “lex.yy.c”. This section contains companion functions that will call, or be called, by the scanner. This section is optional, and will be ignored during this project.

Flex provides a mechanism for conditionally activating rules, called *start conditions*. This is a useful feature for this project since only certain segments of the input file should be scanned. During the XSLT translation layer, snippets of code have been marked to be processed by the lexical analyzer. For example, rules prefixed with `<code>` will only activate when the scanner is in the start condition named `code`. A start condition is activated by using the `BEGIN` action. When the scanner starts, no start condition is activated. When a specific syntactical pattern is matched, a corresponding start condition may be activated. For example, the following rule:

```
"#pragma CODE_START" { BEGIN(code); }
```

will activate only when no start condition is activated, since there is no start condition specifier prefixed to the pattern. When this text is matched by the scanner, it will initialize the start condition `code`. Only rules prefixed with `<code>` will now activate. For example, if the following two rules exist:

```
<code>[0-9]+ { do_something(); }

[0-9]+ { do_somethingelse(); }
```

only the first rule will activate since it contains the correct start condition. Start conditions that will appear in the scanner must be listed in the *definitions section*, beginning with a `%x` followed by the name of the start condition. The `%x` specifies that the start condition is *exclusive*, and that only rules qualified with the start condition will be active. To deactivate a rule, the `BEGIN(INITIAL)` action can be used.

The scanner generated for this project must only scan certain segments of the input file. It was previously defined that code to be processed by the compiler should be enclosed within the code markers, `#pragma CODE_START` and `#pragma CODE_END`. Text in the input file falling outside these markers will just be echoed to the output file using the `ECHO` action.

Figure 5.6 gives an overview of the generated scanner.
Figure 5.6: Program flow for the scanner.
5.3.3 Generating a Bison Parser

To generate a parser, Bison reads an input file formally specifying a context-free grammar, and actions that should be taken when instances of the grammar is recognized. A Bison grammar file has four main sections as shown in Listing 5.10. The task of a Bison generated parser is to group tokens according to its grammar rules [22]. When a grammar pattern is matched the corresponding action specified in the grammar rule should execute. The tokens received by the parser should be generated by a *lexical analyzer* supplied by the user. In this case a lexical analyzer was created using Flex. Each time the parser needs a new token the lexical analyzer will be called.

The tokens returned by the lexical analyzer only contain information about the token type. For example, a INTEGER or a REAL value has been scanned. The details of the token, in this case the actual value of the integer, is not returned with the token type. However, the scanner generated by Flex will contain a copy of the value of the current token in the *yytext* buffer. This is called the *semantic value* of the token.

In order for a parser to be useful it should not only parse expressions by matching token groups, but should also produce some form of output based on the input file. For this project the task of the parser will be to recognize mathematical text streams, and produce a new mathematical text stream as output (Figure 5.4). The semantic values of tokens must therefore be accessible by the parser. To accomplish this, a pointer is passed as argument to *yylex()* , and the semantic value is sent back to the parser through this pointer reference. Since we will build strings of text, the data type for semantic values should be *char*.

```plaintext
%{
Prologue
%
Bison declarations
%%
Grammar rules
%%
Epilogue
}
```

**Listing 5.10:** The Overall Layout of a Bison Grammar.

As stated before, a Bison grammar file has four main sections (Listing 5.10). The *prologue* section contains macro definitions and declarations of functions and variables
that are used in the actions of the grammar rules. The most important definitions include:

1. Declaring the data type of semantic values. This is done by including the macro definition `#define YYSTYPE char*`.

2. Include the Flex generated scanner “lex.yy.c”.

3. Declaring the calling convention of the `yylex()` routine. Since a char* is passed as argument to the scanner routine the function prototype of `yylex()` will be altered to look as follow:

   ```c
   int yylex(YYSTYPE *yylval_param);
   ```

   It should be noted that the function prototype should also be altered on the scanner side of the compiler. The scanner will now be adapted to copy semantic values in the `yytext` buffer to this character pointer to be used by Bison.

4. Declaration of various support functions and variables used by the parser.

The *Bison declarations* section of a Bison grammar defines the token types (also referred to as *terminal symbols*, since they can’t be broken down into smaller groupings) used in formulating the grammar [22]. Tokens are declared by using the `%token` keyword, followed by the name of the token type. Bison will convert these token type names to `#define` directives. These directives will be included in the file “file.tab.h” so that the scanner routine may use name to stand for this token type’s code. The *precedence* and *associativity* of tokens can also be declared in this section. Bison allows you to specify the associativity of tokens by using the precedence operators `%left` and `%right`. The order in which precedence operators are declared, defines the relative precedence of the operators, with the first declaration having the lowest precedence. For example, the two tokens declared in Listing 5.11 are both left-associative, but the MULTIPLY token type enjoys precedence over the PLUS token type. This will result in a expression, \( a*b+c \), being parsed as \((a*b)+c\) instead of \(a*(b+c)\).

The *grammar rules* section contains one or more Bison grammar rules. There must always exist at least one grammar rule. A Bison grammar rule consists of a *result*, *components*, and accompanying *actions*. *Result* is the nonterminal symbol\(^1\) that this rule

\(^1\)A *non-terminal symbol* represents a grouping of terminal symbols or even other non-terminal symbols.
describes, and components are various terminal and nonterminal symbols that are put together by this rule. Each component may also contain actions that determine the semantics of the rule. Actions are specified using C code, and will be executed each time an instance of the specified rule is recognized. The task of most actions is to compute a semantic value for the grouping built by the rule, from the semantic values of the tokens associated with that rule. Multiple rules for the same result can be written separately or can be joined with the vertical-bar character “|”, each with its own set of actions. The definition of a result with multiple rules, each with its own action will have the following form:

\[
\text{result:} \quad \text{rule1-components \{ Action \}} \\
\quad \mid \text{rule2-components \{ Action \}} \\
\quad \mid \text{rule3-components \{ Action \}} \\
\]

The C code in an action can refer to the semantic values of the components of the matched rule with the construct $n$, which refers to the semantic value of the $n$th component. The semantic value of the grouping being constructed is $$$. Bison translates these expressions to the appropriate type when it copies the values into the parser. As mentioned earlier, for this project the data type for semantic values has been set to char*. $$ can now be used to return expression constructed from the various semantic values of components. The example shown in Listing 5.12 will be responsible for matching a simple addition mathematical expression containing floating-point values, and then constructing a new fixed-point compatible expression.

For this project a Bison grammar was created that would parse mathematical expressions written in the ANSI-C format. The goal of this parser is to convert floating-point

\begin{verbatim}
%token PLUS
%token MULTIPLY

%left PLUS
%left MULTIPLY

Listing 5.11: Declaring two token types, along with the associativity and precedence of the tokens.
\end{verbatim}
exp: REAL {
    tempstr = (char*)malloc(strlen($1));
    strcpy(tempstr,$1)
    $$ = tempstr;
}

|   exp PLUS exp {
    tempstr = (char*)malloc(strlen($1)+strlen($3)+12);
    strcpy(tempstr,"fractAdd(");
    strcat(tempstr,$1);
    strcat(tempstr",");
    strcat(tempstr,$3);
    strcat(tempstr,")");
    $$ = tempstr;
}

Listing 5.12: A Bison rule for constructing a mathematical addition expression.

expressions to fixed-point equivalent expressions. Expressions containing integer only values would just be echoed to the output file. Because of the nature of a Bison generated parser, all expression will first be parsed as if they contain only integer values. If the parser detects that this expression does contain floating-point values, the entire expression will be parsed again. The reason for this is that if an expression contains only a single floating-point value the entire expression must be converted to a fixed-point expression. In some cases almost an entire expression will be parsed without encountering a floating-point value, only to detect one at the end of an expression. At this stage it is too late to convert the expression. By first parsing the entire expression as an integer only expression, thus maintaining the original format, the entire expression may be parsed again as a floating-point only expression. Any integer values within the expression will then also be converted to a fixed-point equivalent format to be compatible with the mathematical functions introduced in Section 4.2.

When re-parsing an expression, the scanner should be reconfigured to first scan the expression built during the first parse attempt. This is done by first creating a backup of the current scanner state, and then switching the scanner buffer to the text string built during the first parse attempt. When the second parse attempt has finished, the original
scanner state can be restored. An overview of the parse process is given is Figure 5.7.

5.4 Conclusion

This chapter describes the process of automating the development of SDR applications based on the architecture developed in Chapters 3 and 4. The chapter starts by describing how SDR applications can be defined in a platform independent manner using XML. Various translation layers are then introduced that uses the XML SDR definition to generate the high-level language compatible with the target hardware platform.

There are two main layers of translation. The first layer is responsible for generating the base C-code supporting the developed SDR architecture. XSLT is used to accomplish this. The second layer of translation is responsible for identifying floating-point expression within the generated code that are not compatible with the target hardware platform, and translating these to fixed-point equivalent expressions. A compiler generated by Flex and Bison is used for this.
Figure 5.7: Program flow for the parser.
Chapter 6

Tests and Results

Chapters 3 and 4 details the development of an SDR architecture, while Chapter 5 attempts to automate the creation of SDR applications using this architecture. This chapter will implement various tests to verify the functionality of this SDR architecture, and code synthesis process. We will start by implementing a simple modulation scheme in XML, and generating compatible DSP C code. The synthesised code will be simulated and compared to the results obtained from a reference Matlab implementation. Various hardware tests are then conducted to verify the functionality of the architecture on the target hardware platform. The results obtained are compared to an equivalent hand-coded implementation. Finally, the performance of the synthesised code is assessed, and again compared to an equivalent hand-coded implementation.

6.1 Simulation Results

To verify the functionality of the SDR architecture, a simple SDR application was created: namely an AM modulator and demodulator. Figure 6.1 shows a modulating signal that is modulated by multiplying it with a carrier signal. The AM modulated signal is then demodulated using synchronous (coherent) AM demodulation. The application was specified in XML, and was then transformed to C code using the translation layers described in Chapter 5. For further information on the specification of the application, consult Appendix A .The application consists of the following nodes:

- Modulating signal: A sine wave with a frequency \( f_o \) of 23.4375Hz. Sine waves are generated by means of direct-digital synthesis (DDS) [14] using a 1024-entry look-up
The value of the phase accumulator ($\Delta \phi$) is:

$$\Delta \phi = 2^R \left( \frac{f_o}{f_s} \right)$$

where $R = 10$, and the sampling frequency ($f_s$) is $24$kHz.

- **Carrier signal**: Sine wave with a frequency of $6$kHz.
- **Multiplier**: Multiplies two input samples and produces a single output.
- **Filter**: second-order IIR low pass filter (Direct form II realisation, $N = 2$ [13]) with a cutoff frequency of $0.125\frac{f_s}{2}$. The filter coefficients were calculated using the `butter` function in Matlab.
- **Print**: This module prints a sample to the screen and to an output file.

**Figure 6.1**: *A simple AM modulation-demodulation scheme using synchronous demodulation.*

The application was simulated using the *Crossview Pro Debugger* supplied with the TASKING Toolset [33]. The output of the application was written to a file and imported to Matlab. An equivalent application was written in Matlab to use as reference, and evaluate the computational accuracy of the SDR architecture. The results of both the SDR architecture and the Matlab implementations are shown in Figure 6.2. As observed in the results obtained, the two implementations produce identical outputs. This indicates that the SDR architecture has a high level of computational accuracy, even though fixed-point arithmetic was used.
Figure 6.2: Results obtained from both the DSP and Matlab implementations of the test application. Only a single demodulated curve is visible since the results are so closely matched.
6.2 Hardware Results

As discussed previously, the DSP56311 evaluation module was used as development and testing platform. The DSP56311 is programmed by a personal computer (PC), via a parallel port. The *Picoscope 3224* from Pico Technologies was used as measurement device [32]. The Picoscope 3224 has an accuracy of 1% and a 12 bit resolution, making it ideal for audio and noise measurements. The Picoscope 3224 is directly connected to a PC via a USB interface, to easily store measured results. Figure 6.3 shows the test setup.

![Test Setup Diagram](image.png)

**Figure 6.3:** *Test setup consisting of a DSP56311 evaluation module, a Picoscope 3224 and a PC.*

Since the previous section evaluated the computational accuracy of the SDR architecture, this section will evaluate the functionality and performance of the architecture on the target hardware platform. For this test a simple AM modulation application was created. A sinusoidal message signal, with a frequency of 187.5 Hz, is AM modulated by multiplying it with a 6 kHz carrier signal, as illustrated in Figure 6.4. The application was again specified in XML, and corresponding C code was synthesised by the translation layers described in Chapter 5. For further information on the specification of the application, refer to Appendix A.1.

The results obtained from the test are shown in Figure 6.5. The spectrum shows the modulated message signal around 6 kHz. From the results it can be seen that the spurious free dynamic range (SFDR) of the signal is about 50 dBm.

To place these results in perspective, an equivalent hand-coded reference AM applica-
Figure 6.4: An AM modulation scheme.

Figure 6.5: Time and frequency analysis of an AM modulated signal. The message signal has a frequency of 187.5Hz, and the carrier signal has a frequency of 6kHz.
tion was created to compare with the results obtained from the machine generated version. The results are included in Figure 6.6. By implementing the same application with hand-crafted code, and removing the overhead of the SDR architecture, similar results were obtained. The hand-coded version does not provide any noteworthy improvement over the auto generated code. This then corresponds with the results obtained from the software simulation in the previous section, and verifies that auto generated code executes correctly on the DSP56311.

![Comparison of two AM modulated signals.](image)

**Figure 6.6:** *Comparison of a machine generated AM modulation application and a hand-coded AM modulation application.*

The final component of this test was to evaluate the system performance. Table 6.1 gives a summary of the loads incurred by the various components of the application. The *Load Percentage* column indicates the percentage of processor time needed to execute that component during a single iteration period. This value includes the processing time needed to implement the underlying SDR architecture, as well as the processing time required by the process definition of each node. The *Math Load* column indicates the percentage of processor time needed to execute only the process definition of each component.
Chapter 6 — Tests and Results

<table>
<thead>
<tr>
<th>Component</th>
<th>Load Percentage</th>
<th>Math Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total System</td>
<td>46.6 %</td>
<td>42.5%</td>
</tr>
<tr>
<td>Sine wave (modulating)</td>
<td>15.6%</td>
<td>13.9%</td>
</tr>
<tr>
<td>Sine wave (carrier)</td>
<td>15.6%</td>
<td>13.9%</td>
</tr>
<tr>
<td>Multiplier</td>
<td>13%</td>
<td>11.6%</td>
</tr>
<tr>
<td>DAC</td>
<td>4.8%</td>
<td>3.1%</td>
</tr>
</tbody>
</table>

Table 6.1: Summary of processor loads incurred by the various SDR components in Figure 6.4.

As is clearly evident, these values are closely related. This indicates that the processing time needed to execute the process definition of each node, that mainly contains mathematical expressions, greatly outweighs the processing time needed to implement the underlying architecture. The performance of the hand-coded implementation was also measured. If was found that the percentage load incurred by the total system was only 9%. This value should be compared with the Math Load of the total system in Table 6.1, since this value only represents the processor time required to execute the process definitions of all nodes in the system, closely resembling the way the hand-coded implementation was created. The results indicate that the performance of the hand-coded implementation is nearly five times faster than that of the synthesised code.

Included in Table 6.2 are performance results of additional components not included in this test application. These results correspond with the previous outcome.

<table>
<thead>
<tr>
<th>Component</th>
<th>Load Percentage</th>
<th>Math Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>IIR Filter</td>
<td>68.1%</td>
<td>65%</td>
</tr>
<tr>
<td>ADC</td>
<td>5.8%</td>
<td>4.2%</td>
</tr>
<tr>
<td>Sum</td>
<td>16.6%</td>
<td>13.9%</td>
</tr>
</tbody>
</table>

Table 6.2: Summary of processor loads incurred by various SDR components not included in the test setup in Figure 6.4.
6.3 Conclusion

In this chapter the SDR architecture developed in the previous chapters was tested, along with the efficiency of the automatically generated code. Two main tests were done. The first test was a software simulation that was run using code that was automatically generated. An equivalent test application was created in Matlab to serve as reference. The results obtain indicated a high level of computational accuracy.

In the second test code was again automatically generated, this time for a hardware simulation. An equivalent application was then hand-coded to serve as reference. The results obtained once again indicated a high level of computational accuracy, with both versions achieving a SFDR of 50dB.

The performance of the automatically generated code was inspected and it was found that the processing time required for the mathematical operations overshadowed the processing time needed to implement the overhead of the underlying SDR architecture. Optimization of the mathematical operations would therefore be a good starting point for increasing the performance of the entire system.
Chapter 7

Conclusions

7.1 Review of Results

With wireless communications standards evolving at such a furious rate, the number of waveforms and protocols used by various technologies are ever expanding. Because of this, traditional hardware-based radio terminals are becoming obsolete since the functionality of such radios are fixed. Software-defined radio provides a solution to this problem by allowing radio functionality to be set in software. This gives SDR platforms the ability to adapt to new technologies by simply upgrading its software.

However, to create new SDR applications a programmer would first have to gain a thorough understanding of the target hardware platform, and this can prolong the development cycle of an SDR. At Stellenbosch University we have taken the approach to decompose complex radios into modular, re-usable building blocks. These building blocks can then in turn be used to create new SDR applications. By defining these building blocks in a platform independent manner, radio engineers can design new SDR applications without a comprehensive knowledge of the hardware platform they are targeting. For this reason, a radio description language has been developed in XML that is mostly platform independent. Source code for various hardware platforms can then be generated from this radio definition. Previous research projects have successfully implemented C++ and VHDL code synthesis.

The final stage of this project was to develop an SDR architecture for embedded platforms. The target hardware platform for this project was the Motorola DSP56311. A highly reconfigurable SDR architecture was developed and implemented on the target hardware platform. The architecture was created with automatic code generation in mind
to simplify code synthesis at a later stage. After a functioning hardware implementation was successfully designed, translation software was created to automatically synthesise HLL code from the generic DSL modules. The first layer of the translation process was responsible for generating C code from the XML definition of an SDR application using XSLT, that supported the underlying SDR architecture developed. The next layer of translation was responsible for converting the floating expressions found in the generic DSL modules to equivalent fixed-point expressions. This was done by creating a compiler, using the GNU programming tools Flex and Bison to generate a scanner and a parser respectively.

The results obtained verified the functionality of the SDR architecture developed. It was further shown that it is possible to synthesise C code from the platform independent XML specification of an SDR application.

7.2 Further Work

During the development of the SDR architecture, several areas were identified that could possibly be improved:

- Currently the hardware implementation of the SDR architecture developed in Chapter 3 lacks a proper memory management strategy. Though this does not influence the current implementation, it would be essential if the architecture is adapted to be reconfigured during runtime.

- The compiler developed in Chapter 5 currently only supports a limited subset of the ANSI-C grammar. This limits the volume of expressions recognized by the implemented compiler. It would be advantageous to expand the supported grammar, thus moving closer to an ideal code synthesis environment.

- Studying the performance results obtained in Chapter 6, it is clear the a substantial amount of overhead is introduced when performing fixed-point calculations using the functions introduced in Chapter 4. It is therefore recommended to optimise these functions, either by hand optimizing the generated assembler produced by the TASKING Toolset, or introducing more inline assembly functions in the HLL version of the code.
7.3 Final Remarks

The main purpose of this research was to further investigate the portability of SDR applications between various hardware platforms by using generic DSL modules to define an SDR application. During the course of this thesis it was proven that it is possible to synthesise C code for a DSP platform, to a certain degree. The results obtained indicates that SDR application development time can be reduced.

Synthesised C code may even be used as a starting point for creating hand coded SDR applications, if the underlying architecture is understood.
Bibliography


[27] Van Rooyen, G-J, Telecommunications 813: Extension Course, University of Stellenbosch, 2002


[33] Crossview Pro Debugger,
Appendix A

A.1 Contents of Accompanying CD

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/Compiler/</td>
<td>This directory contains the source files, and the generated files for the Flex scanner and the Bison parser</td>
</tr>
<tr>
<td>/Motorola_DSP56311/</td>
<td>This directory contains the support files needed to implement a test application on the hardware platform.</td>
</tr>
<tr>
<td>/XSLT/</td>
<td>This directory contains the source files “sdr.c.xsl” and “sdr.h.xsl” responsible for the first transformation layer</td>
</tr>
<tr>
<td>/Test/</td>
<td>This directory contains the source files needed to implement the test setup in Figure 6.4.</td>
</tr>
</tbody>
</table>

Table A.1: Summary of the contents of the accompanying CD.
A.2 Code Listing

Listing A.1: Matlab code for calculating the schedule of the system defined in Fig: 3.11

```matlab
% Populate Topology Matrix
gamma = [ 1 -1 0 0; 1 0 -3 0; 0 1 -3 0; 0 0 1 -1; 0 0 0 1 -2];
[x,y] = size(gamma); % Get number of rows(m) and columns(n) in matrix 'gamma'
if (rank(gamma) ~= y-1)
    ERROR = 1 % Inconsistent sample rates!!
else
    % Solve Parametric Solution
    Rgamma = rref(gamma); % Reduce matrix to Reduced Echelon Form
    q = null(Rgamma, 'r'); % Get RATIONAL Null space

    % Solve The Order
    b = zeros(x,1); % Create initial buffer/arc values. All arcs have initial value of 0 -> No value
    s = []; % Vector to store scheduling sequence
    buffer = [];
    hold on;
    for n = 1:y
        while (q(n) ~= 0) % If node is runnable and scheduled less than q(n) times, schedule node!
            v = zeros(1,y);
            v(n) = 1;
            b = b + gamma*v';
            buffer = [buffer b(1)];
            q = q - v';
            s = [s n]; % Add current node to schedule.
        end
    end
plot(buffer,'rs','MarkerEdgeColor','k','MarkerFaceColor','b','MarkerSize',5);
if (q ~= 0)
    % DEADLOCK
end
s %Print schedule to screen
end
```
Listing A.2: Complete example of configuring a new converter of type "MUL":

Listing A.2: Complete example of configuring a new converter of type "MUL":

// Step 1 - Assign the process pointer. */
SDF[1]−>process = sdr_basic_mul_process;

// Step 2 - Allocate memory for the attribute structure. */
SDF[1]−>attributes = (SDR_BASIC_MUL *) malloc(sizeof(SDR_BASIC_MUL));

// Step 3 - Set values of the attribute structure. */
set_sdr_basic_mul(SDF[1]−>attributes, SDF[1]);

// Step 4 - Allocate memory for the input list (That will contain 2 pointers). */
SDF[1]−>input_list = (QUEUElist) malloc(sizeof(QUEUEptr) * 2);

// Step 5 - Allocate memory for the first input buffer, with a size of "5". */
Ptr = malloc(sizeof(double) * 5);

// Step 6 - Allocate memory for a QUEUE structure. */
SDF[1]−>input_list[0] = malloc(sizeof(QUEUE));

// Step 7 - Set the attributes of the QUEUE structure. */
setQueue(SDF[1]−>input_list[0], 2, Ptr, Ptr + sizeof(double) * (2−1), Ptr, Ptr, 2);

// Step 8 - Repeat steps 5 to 7 for all input ports. */
// Step 5 - Allocate memory for the second input buffer, with a size of "5". */
Ptr = dspmalloc(sizeof(double) * 5);

// Step 6 - Allocate memory for a QUEUE structure. */
SDF[1]−>input_list[1] = malloc(sizeof(QUEUE));

// Step 7 - Set the attributes of the QUEUE structure. */
setQueue(SDF[1]−>input_list[1], 2, Ptr, Ptr + sizeof(double) * (2−1), Ptr, Ptr, 1);

// Step 9 - Allocate memory for the output list (That will contain 1 pointer). */
SDF[1]−>output_list = (QUEUElist) malloc(sizeof(QUEUEptr) * 1);