RadiO Modeling Environment (ROME)

Prepared by:

Shaun Katz

Department of Electrical Engineering
University of Cape Town

Prepared for:

Dr Simon Winberg

Radar and Remote Sensing Group
University of Cape Town

· August 2013 ·

Submitted to the Department of Electrical Engineering at the University of Cape Town in fulfilment of the academic requirements for a Master of Science degree in Electrical Engineering.

Key Words: software defined radio; hardware description languages; field programmable gate arrays; electronic design automation; xilinx platform studio

Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this masters dissertation from the work(s) of other people, has been attributed and has been cited and referenced.
3. This masters dissertation is my own work.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as their own work or part thereof.

Name: **Shaun Katz**

Signed: Date: November 14, 2013

Acknowledgements

First and foremost, I would like to acknowledge my project supervisor, **Dr Simon Winberg**, for proposing this research topic and organising access to necessary equipment as well as for his guidance and persistent help.

I would also like to thank the members of the Radar Remote Sensing Group (RRSG) as well as **Prof Mike Inggs** and **Dr Amit Mishra**.

Thanks should be given to the **Square Kilometer Array (SKA) South Africa** for the financial assistance given towards this research.

Abstract

At the University of Cape Town, there are a range of undergraduate and postgraduate projects related to the prototyping of and experimenting with Software Defined Radio (SDR) and radar systems [1]. Students often spend a large portion of their research time developing processing modules from scratch [1]. Unfortunately, the fact that students spend much of their time building and integrating fundamental functional blocks, results in students not having sufficient time to devote to the main focus of their research [1].

High level abstraction at the design entry stage of a typical Field Programmable Gate Array (FPGA) design flow, has shown a reduction in the development time of complex systems [2]. Such abstraction techniques, discussed in [2], include: graphical representations of system blocks; automatic code generation of parametric blocks and Intellectual Property (IP) reuse [3].

Many commercial Electronic Design Automation (EDA) products for FPGA development use graphical visualisations of Hardware Description Language (HDL) code to assist engineers [3]. However, these tools usually don't abstract clock or bus signals [3]. Furthermore, reuse of HDL modules tends to involve much time spent connecting a range of commonly understood connectors. Automating these connection tasks could save time and simplify designs [3].

This dissertation presents a model-based rapid prototyping tool for use in SDR on a reconfigurable computing platform. The tool aims to facilitate a novice developer, student or researcher with the development of and experimentation with SDR processing applications deployed on a FPGA platform.

The developed tool, which integrates with Xilinx Platform Studio (XPS), is presented in this dissertation. The results, of testing the tool by implementing example designs, are also presented in this dissertation.

Key Words: software defined radio; hardware description languages; field programmable gate arrays; electronic design automation; xilinx platform studio

Contents

Acronyms

ADC Analog to Digital Converter.

DnD Drag and Drop.

EDA Electronic Design Automation.

EDK Embedded Development Kit.

FPGA Field Programmable Gate Array.

GPMC General Purpose Memory Controller.

GUI Graphical User Interface.

HDL Hardware Description Language.

IP Intellectual Property.

ISE Integrated Software Environment.

MHS Microprocessor Hardware Specification.

MPD Microprocessor Peripheral Definition.

MVC Model-View-Controller.

NCO Numerically Controlled Oscillator.

RHINO Reconfigurable Hardware Interface for computiNg and radiO.

ROM Read-Only Memory.

ROME RadiO Modeling Environment.

SDR Software Defined Radio.

SoC System on Chip.

VHDL VHSIC Hardware Description Language.

VLSI Very-Large-Scale Integration.

XML Extensible Markup Language.

XPS Xilinx Platform Studio.

XST Xilinx Synthesis Tool.

ZUI Zoomable User Interface.

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This dissertation presents a model-based rapid prototyping tool for use in Software Defined Radio (SDR) on a reconfigurable computing platform. The tool aims to facilitate a novice developer, student or researcher with the development of and experimentation with SDR processing applications deployed on a Field Programmable Gate Array (FPGA) platform.

This chapter will provide a brief background to the work presented in this dissertation, followed by an analysis of the initial user requirements for the project. The chapter will present a set of functional requirement to satisfy these user requirement. Finally, the chapter concludes with a description of the structure of this dissertation.

1.1 Background

In order to fully understand the context of this dissertation, a brief background will be presented. This section first presents the concept of Software Defined Radio, followed by an introduction to the Reconfigurable Hardware Interface for computiNg and radiO (RHINO) platform. The section concludes with brief description of high level abstraction and data flow modeling.

1.1.1 Software Defined Radio (SDR)

Software-defined radio (SDR), refers to wireless communication in which transceiver modulation and demodulation is handled by a computer [5]. The underlying goal of SDR is to replace as many Very-Large-Scale Integration (VLSI) and analog components, of the transceiver, with programmable devices [5]. The ideal case is often considered to be an Analog to Digital Converter (ADC) connected directly to an antenna [5]. SDR technology is being used in various other field besides telecommunications. These various other application include radio astronomy and radar. Section 2.1 presents the concept of SDR in greater detail.

1.1.2 Software Defined Radio Group (SDRG)

The Software Defined Radio Group (SDRG) is a research group at the University of Cape Town (UCT). The group's focus is on the research and development of SDR and radar as well as other remote sensing technologies. In the SDRG, there are a range of undergraduate and postgraduate projects related to the prototyping of and experimenting with SDR and radar systems [1]. Students often spend a significant portion of their research time developing processing modules from scratch [1]. Unfortunately, the fact that students spend much of their time building and integrating fundamental functional blocks, results in students not having sufficient time to devote to the main focus of their research [1].

1.1.3 Reconfigurable Hardware Interface for computiNg and radiO (RHINO)

The RHINO is a SDR platform developed by the Software Defined Radio Group (SDRG) at the University of Cape Town's Department of Electrical Engineering [6]. The RHINO was designed for use in research as well as education [6] and aims to facilitate in the development of the necessary skills required for SDR [6]. For more detail on the RHINO platform refer to section 2.1.3.

1.1.4 High Level Abstraction

High level abstraction at the design entry stage of a typical FPGA design flow, has shown a reduction in the development time of complex systems [2]. Such abstraction techniques, discussed in [2], include: graphical representations of system blocks; automatic code generation of parametric blocks and Intellectual Property (IP) reuse [3].

Many commercial Electronic Design Automation (EDA) products for FPGA development use graphical visualisations of Hardware Description Language (HDL) code to assist engineers [3]. However, these tools usually don't abstract clock or bus signals [3]. Furthermore, reuse of HDL modules tends to involve a considerable amount of time spent connecting a range of commonly understood connectors. Automating these connection tasks could save time and simplify designs [3].

1.1.5 Data Flow Modelling

In order to improve the automation of designing a system from the initial specification, research efforts have turned to modeling methods to specify, analyze, verify, and synthesize systems [7]. The main motivation for using models system design is abstraction [7]. Abstraction helps to better understand a complex system by hiding irrelevant information [7].

The data flow paradigm is widely recognized in industry and academia [8]. This is evident in the popularity of MathWorks Simulink and National Instruments LabVIEW [8]. Many have argued that data flow modeling is a natural method for representing digital signal processing systems [8]. Modeling methods can help to better characterize, understand and thus improve SDRs [8].

1.2 Objective

The main objective of this project is to develop an easy-to-use modelling tool, for use in a SDR and FPGA training context. The tool will facilitate a student, or researcher not highly experienced in HDL coding and SDR processing, with the deployment and experimentation of SDR processing on a Reconfigurable Hardware Interface for computiNg and radiO (RHINO) platform. The developed application should be capable of automatically generating synthesisable HDL code from a graphical system-level block diagram representation.

1.2.1 Initial User Requirements

A project proposal was presented before undertaking this dissertation. The initial user requirements for the tool, which made up the project proposal, are:

U1. Easy to use.

U2. Import/integrate existing and future HDL designs as graphical blocks.

U3. Connect blocks together using a specified interfacing standard.

The term "Easy to use", in U1, is defined, in the context of this dissertation, to mean that the tool is required to make use of some programming techniques common to Graphical User Interfaces (GUIs) and EDA tools. These techniques include, but are not limited to: saving and restoring designs, as well as copy/paste and undo/redo functionality. U3 refers to a "specified interfacing standard" for the connection of blocks. This requirement means that the tool is required to abstract commonly understood connections away from the user. In other words, if two block are connected to each other using a common bus interface, the tool is required to connect these two blocks with one single connection rather than the multiple connections that make up the bus interface.

1.2.2 Functional Requirements

A number of functional requirements, and "nice-to-haves", were developed in order to satisfy the user requirements. Table 1.1 lists the user requirements and the functional requirements that satisfy them. These functional requirements are listed as:

F1. Provide a list of blocks that can be dragged and dropped into a design area.

F2. These blocks need to be able to support ports and parameters. Changes to these parameter values, in the GUI, should reflect in the generated HDL.

F3. The tool must have the ability to connect blocks together.

F4. Blocks need to be able to be resized. Scaling of blocks can help to emphasize importance or to reduce clutter and improve the use of the design space.

F5. The GUI will have to redraw blocks in the event that a parameter changes. A parameter can change the number of I/O ports on a block.

F6. It would be very useful to have colour coded blocks, i.e. interfaces, controllers, primitives, DSP blocks etc.

F7. Provide the ability to group a number of blocks together to create another block. This hierarchy of sub-systems could help to reduce the complexity of large designs.

F8. Provide the ability to group wires into buses. This refers to both grouping the individual bits of a vector into one connector as well as grouping many connectors together to form a bus of connectors.

F9. The GUI could provide a view of the data path's source code, where one can follow instantiations back to the implementation of modules and visa-versa.

F10. The GUI could show real-time any syntax errors in the design.

F11. The compilation and synthesis tools must be hooked into the GUI and able to be run from within the design-flow.

The functional requirements, F6, F7, F9 and F10 are deemed low priority. However these requirements are still considered "nice-to-haves".

1.2.3 Purpose of the study

Students at UCT often spend a large portion of their research time developing processing modules from scratch [1]. Winberg et al. explain, in [1], that students spend a considerable amount of their time getting these fundamental HDL building blocks together, and that this results in students not having sufficient time to devote to the main focus of their research.

There are many significant advantages to a text-based design methodology [9]. HDL code is said to be easier to parameterize and regular structures easier to replicate than

Table 1.1: Satisfying Requirements: Table of functional requirements and the corresponding user requirement they satisfy. The table also presents expected difficulty level associated with implementation of the functional requirement.

Number	**Function**	**Requirement(s) Satisfied**	**Expected Difficulty (1-10)**
F1	Library of available blocks	U1, U2	2
F2	Ports & parameters	U1, U2	1
F3	Connecting blocks	U1, U3	2
F4	Resizing	U1	3
F5	Redrawing blocks	U1	5
F6	Colour coding blocks	U1	3
F7	Grouping of blocks	U1	7
F8	Grouping of wires	U1, U3	7
F9	Link to source code	U1	8
F10	Real-time syntax errors	U1	8
F11	Link to synthesis tools	U1	3

a schematic diagram [10]. Hardware behavior and structure can be described in various levels of abstraction by a designer in HDLs [11]. However, people find the textual representation, of a structural HDL model, less illustrative and harder to read than the schematic one [9, 12]. Moreover, design errors are easier to detect in a schematic representation than in its textual form [9, 12].

Graphical visualization, of a HDL model, is not only useful in verification but in design documentation as well [12]. Providing a graphical representation of a designed model allows other designers to understand designs easier and faster [12].

Visualization techniques are useful in an education context as well [12]. Visualization of a model can help students imagine how a HDL design is supposed to function [12].

It is for the aforementioned reasons that a number of commercial EDA tools provide some form of graphical visualization of structural HDL models [12]. However, these tools usually don't abstract clock or bus signals [3] thus making the interconnection of blocks a time consuming task. Furthermore, these commercial tools are either too expensive or complex for use in education [9, 11, 12].

The proposed tool aims to facilitate a novice developer, student or researcher with the development of and experimentation with SDR processing applications deployed on a

FPGA platform.

1.3 Scope and Limitations

The scope of this project is to design and test a model-based SDR prototyping tool for the purpose of proving the concept. The scope of this project is limited by time and budget constraints. Due to these constraints:

- The study is limited to the generation of synthesisable HDL code and will not focus on verification or simulation techniques.
- The study is limited to structural code generation and will not focus on behavioural code generation.
- The study is limited by the information attainable from proprietary software tools.
- The functional requirements, F6, F7, F9 and F10 are deemed low priority and thus fall outside the scope of this study.

1.4 Plan of Development

This dissertation will focus on the development and testing of a model-based prototyping tool for use in SDR on a reconfigurable computing platform. This includes an investigation into SDR, current system-level design tools, string parsing techniques, as well as an investigation into a suitable methods of HDL code generation. The structure of this dissertation is described in this section.

Chapter 2 starts by introducing SDR, its history and applications followed by a review of hardware platforms currently in use. Various design entry methodologies, providing different levels of abstraction, are then analyzed. The chapter goes on to reviews three tools commonly used to build SDR applications. Techniques for VHSIC Hardware Description Language (VHDL) analysis and visualisation as well as code generation are presented. The chapter concludes with an introduction to the Xilinx Platform Studio (XPS) design tool and its associated files.

Chapter 3 begins by explaining the processes involved in the research and development of the work presented in this dissertation. The chapter goes on to list the various elements that make up the development and experimentation environments. A set of experiments, that will be performed in order to verify the correctness of the designed tool, will then be presented. The chapter concludes with an explanation on how data, resulting from experimentation, will be collected and analyzed.

Chapter 4 explains the overall design methodology of RadiO Modeling Environment (ROME). The chapter presents an introduction to the proposed system followed by an explanation of the methodology behind its design. The chapter concludes with a comprehensive look at all the graphical components of ROME and how they are implemented.

Chapters 5, 6 and 7 present three experiments designed to test the functionality of the proposed system. Chapter 5 presents the design of a waveform generator as well as the results from this experiment. Similarly chapters 6 and 7 present the design of and the results from a wishbone-based adder and a digital FM demodulator chain respectively.

Chapter 8 discusses the results of the experiments presented in chapters 5, 6 and 7 as well as presents conclusions, about the system, based on the discussion. Finally the chapter concludes with ideas for future work.

Chapter 2

Literature Review

The fields of SDR and VHDL design are rich in history. This chapter starts by introducing SDR, its history and applications followed by a review of hardware platforms currently in use. Various design entry methodologies, providing different levels of abstraction, are then analyzed. The chapter goes on to reviews three tools commonly used to build SDR applications. Techniques for VHDL analysis and visualisation as well as code generation are presented. The chapter concludes with an introduction to the XPS design tool and its associated files.

2.1 Software Defined Radio (SDR)

This section introduces the concept of SDR, its history and applications followed by a review of hardware platforms currently in use.

2.1.1 Overview

Radio transceiver design has changed dramatically due to ever increasing computing power [13]. Traditionally, an analog radio consists of a super-heterodyne transceiver [13]. Here signals, from an antenna, are converted down to an Intermediate Frequency (IF) where filtering takes place [13]. Later the filtered signals are converted down to baseband and finally demodulated [13]. Figure 2.1.a illustrates a traditional analog radio receiver.

A digital radio transceiver on the other hand is made up of two parts: a radio Front-End (FE) and a digital Back-End (BE) [13]. The FE focuses on narrowband frequency down-conversion followed by an ADC [13]. The BE is responsible for digital processing functions such as filtering, (de)modulation and channel (de)coding [13]. Figure 2.1.b illustrates a digital radio receiver.

SDR architectures have come about due to an increasing need for more flexible and adaptable systems [13, 14, 15]. In a SDR system the typical functionality of a radio

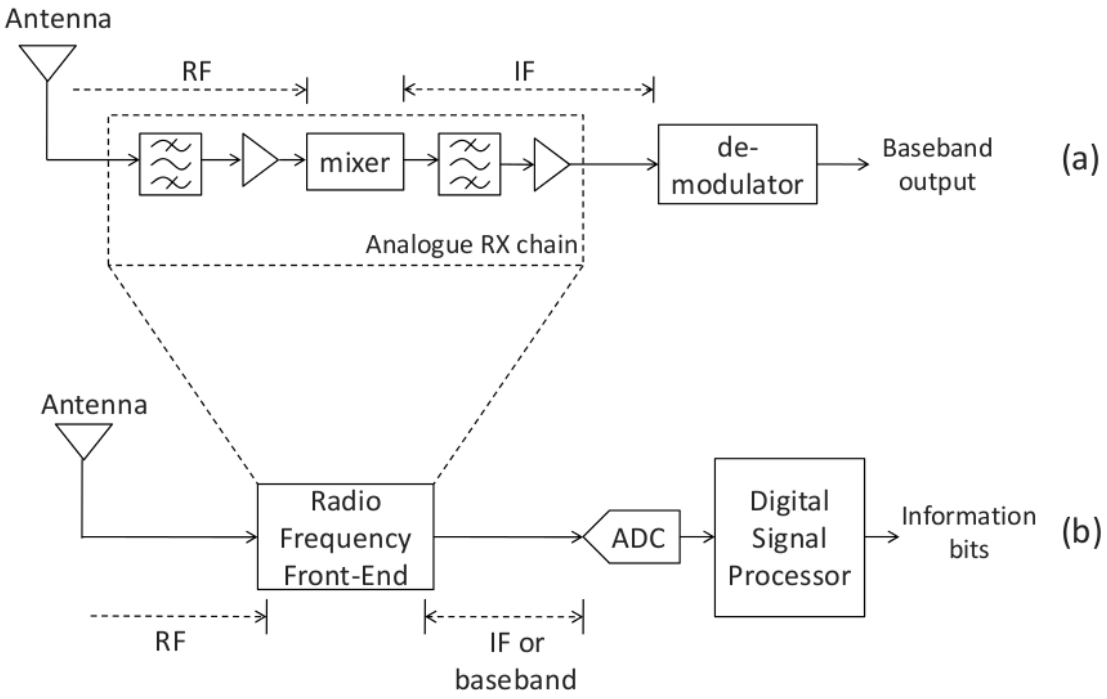


Figure 2.1: Block diagram of an analog and digital receivers: illustrating a block diagram representation of an Analog (a) and a Digital (b) hardware receiver. Adapted from [13].

interface, usually implemented with dedicated hardware, is defined by software [15]. This software defined interface allows SDR systems to dynamically adapt to their radio environment [15].

The underlying design goal of SDR is to be as general as possible and to rely on software for the implementation of new functions [13]. According to Buracchini [15], this is achieved by:

- Shifting the boundary between analog and digital towards RF, by making use of wideband ADC/DAC as near to the antenna as possible [15].
- Replacing as many VLSI and analog components, of the transceiver, with programmable devices [5] such as DSPs, FPGAs or General Purpose Processors (GPP) [13, 15].

The ideal SDR, often referred to in the literature as a Software (SW) Radio [14, 15], consists of an antenna, ADC/DAC, and programmable device [1, 5, 13, 15]. Figure 2.2 shows a block diagram of the ideal SW radio initially proposed by [14]. In Mitola's [14] design, all RF and baseband signals, of the receiver, are digital due to the ADC at the antenna [16]. According to Valerio [13], the following three conditions must be met before such a system can be realized:

- The antenna must operate at all frequencies of interest.

- The ADC/DAC must be able to sample at rates greater than twice the maximum frequency of interest.
- The digital processor must have enough processing power to handle the intended signal processing.

Valerio goes on to state, in [13], that, in practice, truly ideal software-defined radios are only realizable for certain simple applications such as AM radio. Although current digital processor technology is capable for most applications, antennas are usually designed for a specific frequency band [13]. Not to mention the extraordinary specifications required by the ADC [13, 16]. Abidi [16] explains, that in order to digitize cellular or WLAN signals, in the band from 800MHz to 5.5GHz, one would need a 12 bit , 11 GS/s ADC. At that time [16], as well as at the time of this report, such specifications are impossible.

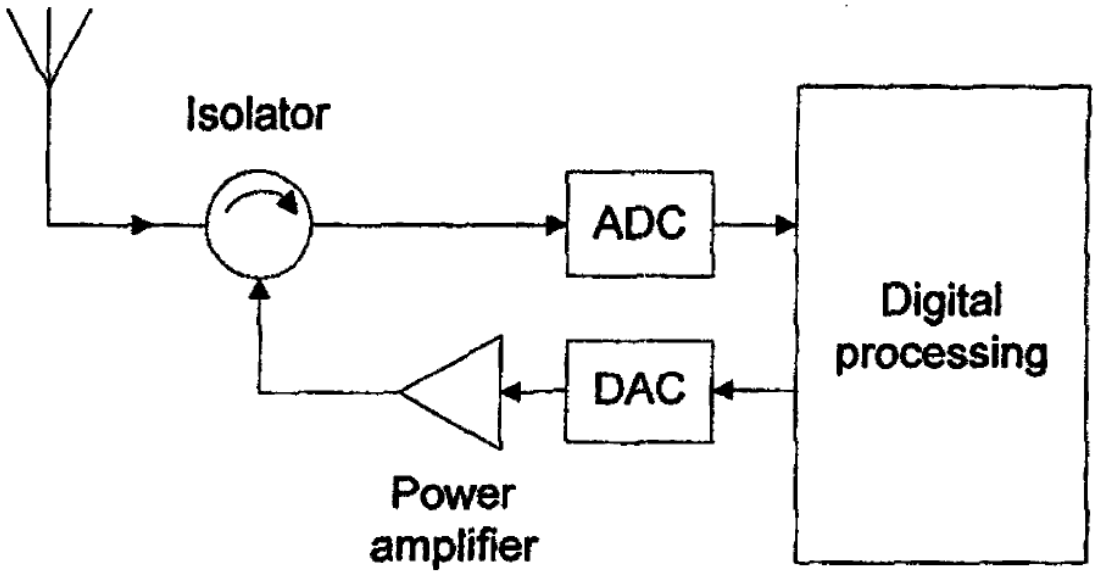


Figure 2.2: Block diagram of an ideal SW radio: illustrating a block diagram representation of an ideal software radio transceiver. Adapted from [16].

Due to these technological limitations, a typical software-defined radio architecture follows a more realisable configuration. Figure 2.3 shows the block diagram of a typical SDR. This configuration consists of a wideband radio FE, which shifts a portion of the spectrum to IF before digitization [13]. Valerio [13] notes a fundamental difference between the wideband radio FE of a SDR and the narrowband radio FE common to digital transceivers.

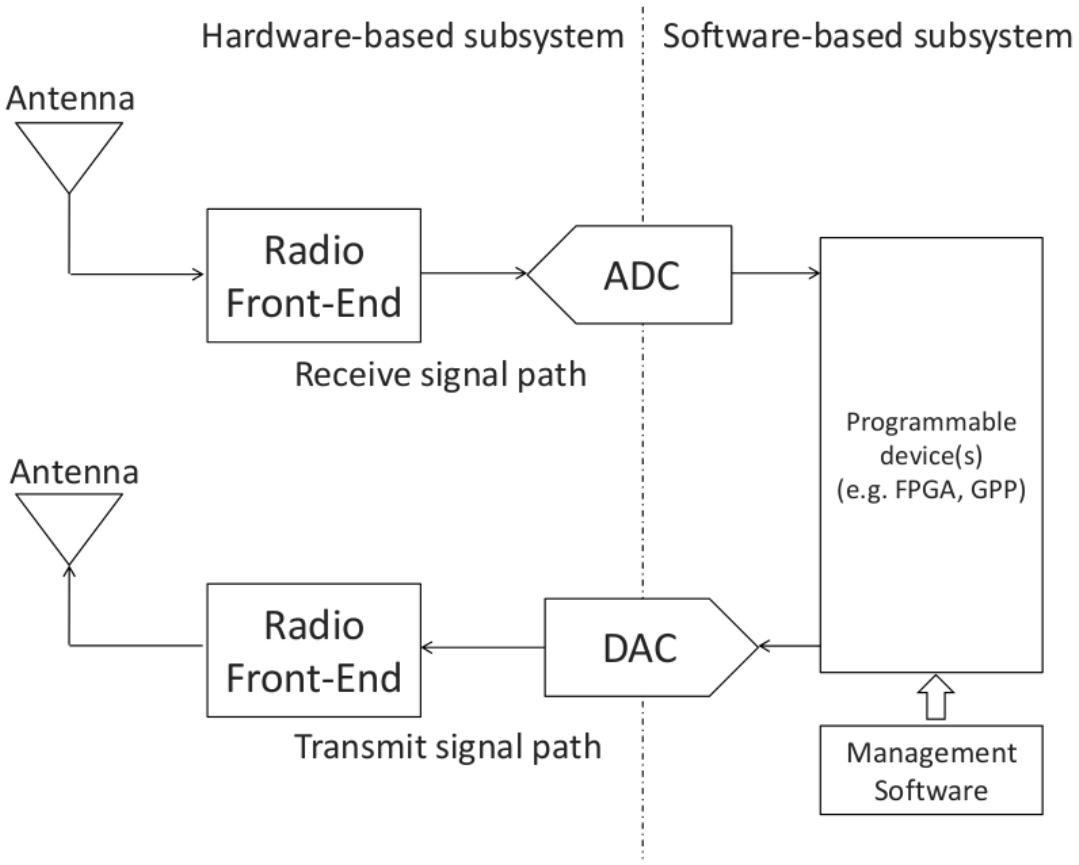


Figure 2.3: Block diagram of a typical SDR: illustrating a block diagram representation of a typical software defined radio transceiver. Adapted from [13].

2.1.2 Applications for Software-Defined Radio

SDR technology is being used in various other fields besides telecommunications. These various other application include radio astronomy and radar. This section will introduce the concepts of radio astronomy and radar as well as the use of SDR technology in these fields.

Radar

Radar (Radio Detection and Ranging) is a method of detecting targets using electromagnetic waves. Radar works by broadcasting a radio signal and receiving the reflected echoes. Information about a targets direction, relative to the receiver, range and speed can be calculated by processing these reflections.

Traditionally, a radar system is built using dedicated hardware [17]. Such hardware is tailored to the task to be achieved and offers little or no reconfigurability [17]. However,

the possibilities of radar are broad, covering fields like: short and long range surveillance on the ground and in the air; target detection, recognition and tracking; weapon guidance; etc [17]. Debatty proposes, in [17], the use of SDR to create multifunctional Software-Defined Radars.

Radio Astronomy

Radio astronomy consists of measuring the electromagnetic emissions from distant radiating sources [18]. This type of astronomy addresses a broad range of thermal and non-thermal radiation [18]. As opposed to optical astronomy, which uses lenses and mirrors to capture the light waves emitted by celestial bodies, radio astronomy uses radio antennae to detect the radio waves [19]. Radio astronomy aims to deduce, from analysis of these electromagnetic radiation distributions, what objects are radiating as well as what causes specific emission characteristics [18].

Scott explains, in [19], that, like radar, radio astronomy instrumentation is highly specialized. Custom, dedicated instruments are built for each individual radio astronomy applications [19]. The flexibility, and reconfigurability of SDRs has been recognized as a means of optimizing radio astronomy telescopes [20].

2.1.3 SDR Platforms

There have been a number of platforms developed to solve the SDR problem. Many of these platforms make use of reprogrammable architectures such as GPPs and DSPs as well as reconfigurable FPGA devices [21]. Tan et al. present a GPP-based SDR platform in [22]. A number of DSP-based platforms include, but are not limited to, those presented in [23, 24, 25].

Safadi et al. [26] suggest however, that a hybrid approach offers the best performance for a SDR implementation. This hybrid approach, which makes use of various computing technologies, falls into the field of heterogeneous computing [8].

Numerous SDR platforms have been proposed in the literature which make use of hybrid processing technologies. These include, but are not limited to, those platform presented in [26, 27]. This section will review two heterogeneous SDR platforms used in education namely: the Universal Software Radio Peripheral (USRP) and the Reconfigurable Hardware Interface for computatioN and radiO (RHINO).

Universal Software Radio Peripheral (USRP)

Ettus Research^{TM}'s Universal Software Radio Peripheral (USRP) [28] and their collection of daughterboards [29] [13, 30] provide users with a comprehensive platform for preforming

SDR processing. Typically, the USRP is used in conjunction with GNU Radio [28]. As mentioned in section 2.3.2, the goal of GNU Radio is to promote a wider access to the EM spectrum [30]. Inggs [30] points out that an implication of this underlying goal is that, for the end-user to preform software-defined radio processing, the user should not require general purpose computing equipment beyond that which is readily available off the shelf. Figure 2.4 shows a block diagram of the USRP platform and illustrates the connections between the mainboard and daughterboards.

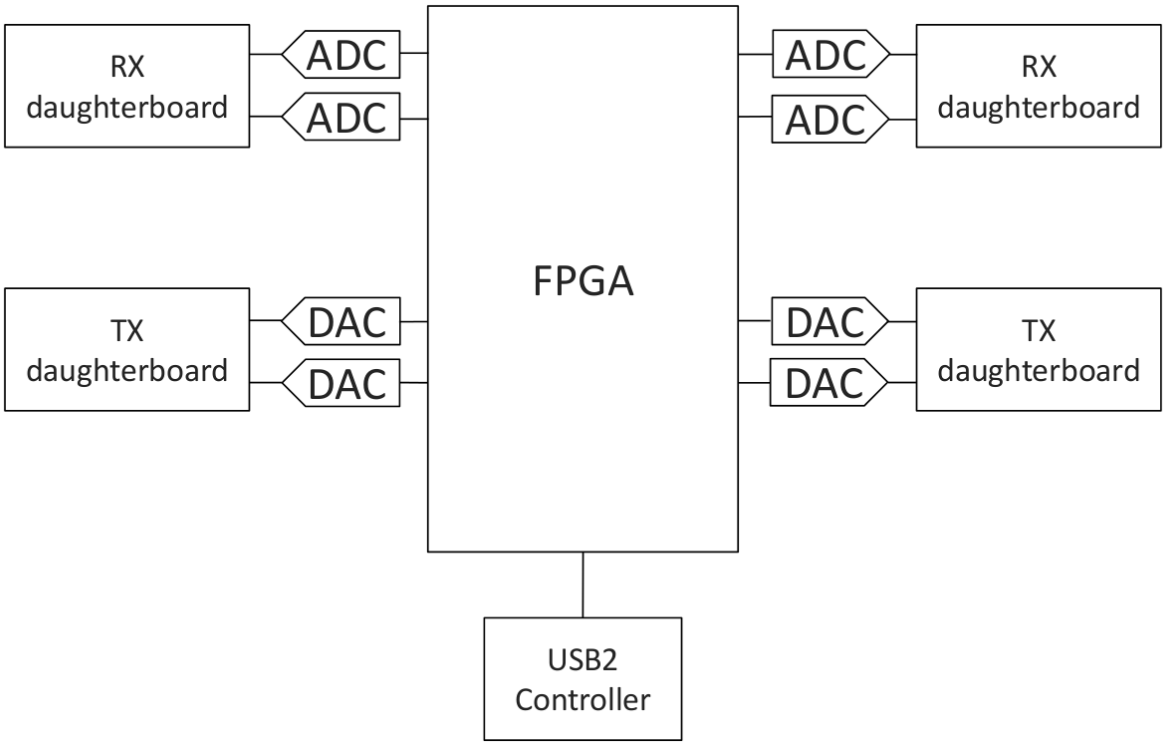


Figure 2.4: The USRP Platform: illustrating a block diagram representation of the USRP platform. Adapted from [13].

The ADCs/DACs are connected to daughterboards which handle the radio FE implementation [13]. Digital radio tasks performed on the USRP are divided between the internal FPGA and an external CPU through a USB2 connection [13]. However user's generally would not modify the FPGA [30]. Instead, general purpose tasks, such as up/down conversion, decimation, and interpolation, are performed on the FPGA [13, 30]. While the user's SDR application-specific tasks, on the other hand, are performed on the host CPU [13, 30]. Such tasks include: modulation and demodulation. According to Inggs [30] this approach is simple and cost effective however, the end-user can not leverage the full potential and advantages of performing signal processing on a FPGA.

Reconfigurable Hardware Interface for computiNg and radiO (RHINO)

The RHINO, is a SDR platform developed by the Software Defined Radio Group (SDRG) at the University of Cape Town's Department of Electrical Engineering [6]. The RHINO was designed for use in research as well as education [6] and aims to facilitate in the development of the necessary skills required for SDR [6].

A Xilinx XC6SLX150T FPGA is the primary means by which the RHINO platform performs digital signal processing [8]. Connected to the RHINO's FPGA is 512MB of DDR3 RAM, 2 FMC mezzanine card slots and 2 10 Gbps Ethernet [8].

The RHINO has a Texas Instrument Sitara ARM Cortex A8 processor which provides the platform with command and control support [8]. This processor has 256 MB of permanent storage, as well as 256 MB of DDR2 RAM available to it. Additionally it supports USB, SD Cards, HDMI and Audio (in- /out) peripheral interfaces [8]. The ARM's General Purpose Memory Controller (GPMC) interface provided and an interconnection bus with the FPGA [8].

BORPH Linux [31], installed on the processor, manages configuration and control of the FPGA [8]. The control signal, managed by BORPH, are communicated over the GPMC-based bus between the processor and the FPGA [8].

2.2 Design Entry

A typical FPGA design flow consists of a combination of tools that implement a designed system [10]. Figure 2.5 illustrates a typical design flow for an FPGA design. The current flows are typically very modular [10]. These modules take the design from a system specification, through synthesis, placement and routing, to an eventual bitstream for programming.

This section will focus on the design entry stage of the typical flow. Design entry is the way a designer describes the hardware intended for implementation [10]. There are a number of different ways to describe a design [10]. These methods include, but are not restricted to, Hardware Description Language (HDL) as well as schematic and High-level Synthesis (HLS) entry [10]. This section will briefly introduce the aforementioned design entry methods.

2.2.1 Abstraction Levels

Digital systems can be described at different abstraction levels in order to manage the design and description of complex systems [32]. Typically a design can be described at

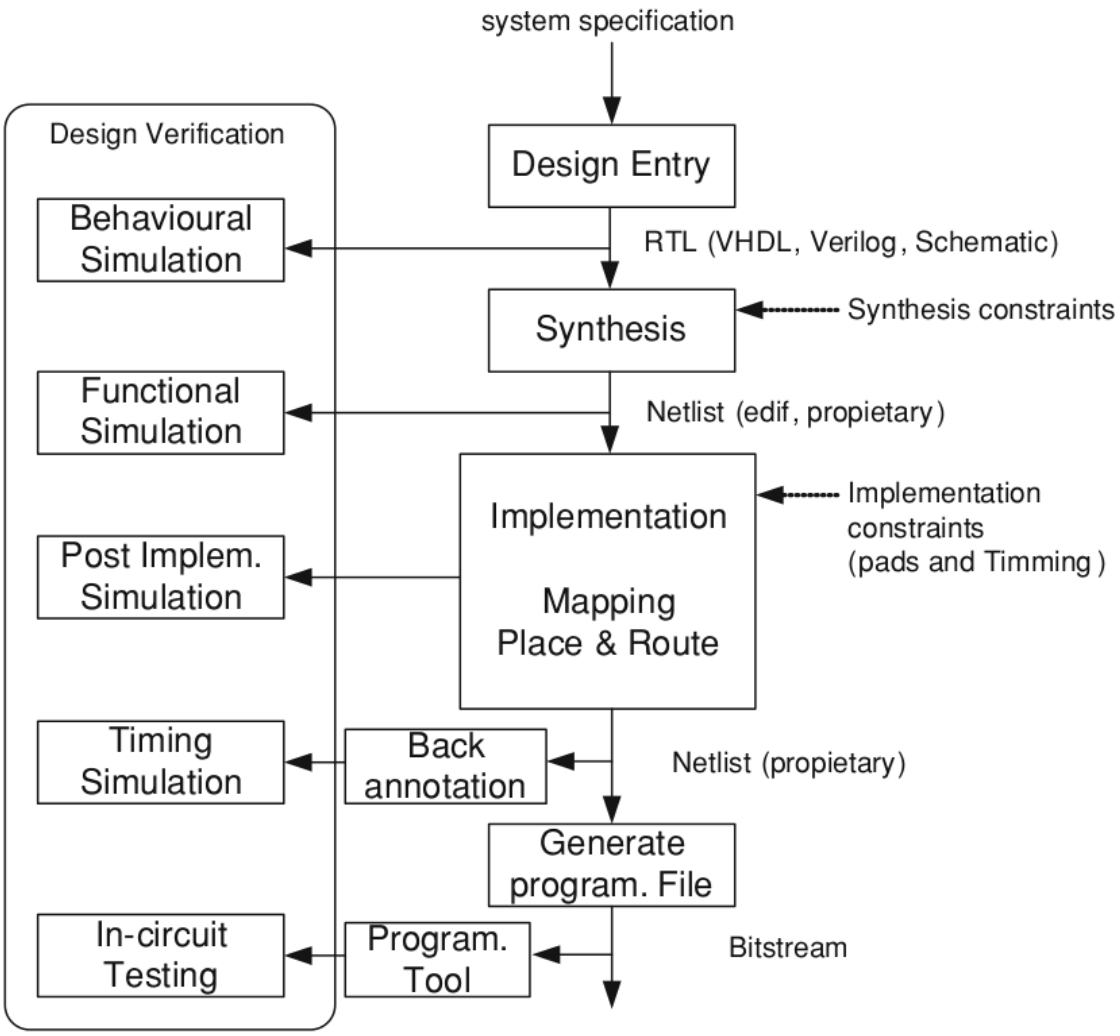


Figure 2.5: Typical Design Flow: illustrating a typical design flow for FPGA design. Adapted from [10].

either the behavioral or structural level. Figure 2.6 illustrates the relationship between the behavioural and structural levels as well as their relationship with the physical layer.

Digital systems are typically made up of smaller subsystems [33]. At the structural level, a system is described as the interconnection of gates and components [32]. This design model allows for large complex systems to be build up from the interconnection of smaller, simpler predefined systems [33].

The behavioral model, on the other hand, sits at a higher level of abstraction. At the behavioral level, a system is described in terms of what its function is rather than its interconnected components [32].

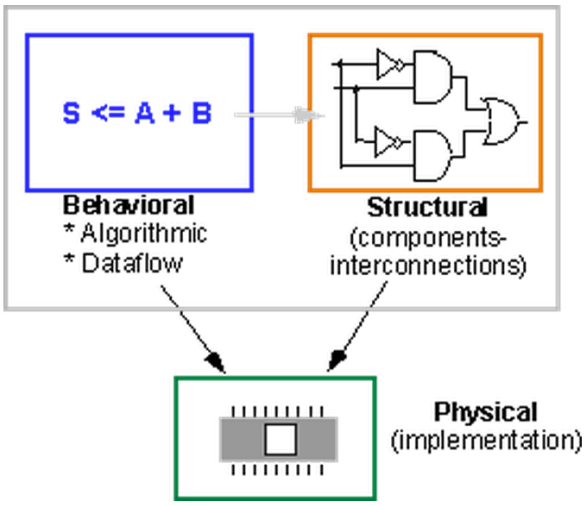


Figure 2.6: Levels of Abstraction: illustrates the relationship between the behavioural and structural levels as well as their relationship with the physical layer. Adapted from [32].

2.2.2 Hardware Description Language (HDL) Design Entry

There are a large number of HDLs available these days [34], however VHDL and Verilog are regarded as the two industry standard languages [34, 10, 35]. Most FPGA tools support both VHDL and Verilog and even support the mixing of both languages in a single project [10]. Although the decision to use any one language over the rest does not restrict the functionality of the end-product, it may affect the design process in a number of other ways [34]. These include, but are not restricted to, the readability and re-usability of code, the speed of development and the probability and number of bugs [34].

VHSIC Hardware Description Language (VHDL) VHDL was developed on behalf of the United Stated Department of Defense during the later part of the '70s and early '80s [35]. Borrowing much of its concepts and syntax from the ADA programming language [36, 37], VHDL is capable of describing very complex behavior [35]. The language was designed to uniformly document the functionality of parts and to work identically on any simulator [37]. Proposed as an IEEE standard in 1986 and eventually adopted in 1987, VHDL has gone through a number of revisions [35]. The VHDL standard, IEEE 1076, was updated in 1993 and again in 2008 [10].

Verilog Verilog HDL is a general-purpose hardware description language which originated at Gateway Design Automation in 1983 [38]. IEEE 1364 was originally accepted as an IEEE standard in 1995 [38]. Subsequently the standard was updated in 2001 and again in 2005 [38]. The language is said to be easy to learn, for experienced C programmers, due to its syntax similarity to the C programming language [38].

2.2.3 Schematic Design Entry

Schematic capture is a design entry methodology where in a user describes a logic circuit graphically [39]. A circuit is described with interconnecting graphical symbols which represent logic functions [39].

EDA tools fall into a category of software tools developed for use in the designing and testing of electronic systems such as: printed circuit boards, application-specific integrated circuits (ASICs) and reconfigurable hardware platforms like FPGAs [10].

Schematic editors require an appropriate library of logic symbols as well a translator to convert the schematic into the vendor specific netlist format used by the FPGA [39]. Both the library and translator are typically supplied as a package by the FPGA or EDA vendor [39]. FPGA symbol libraries and netlist interfaces exist for many popular schematic editors, including Viewdraw from Viewlogic Systems, SDT from OrCAD Systems, Design Architect from Mentor Graphics, and Composer from Cadence Design Systems [39].

Medrano et al. evaluate a number of EDA schematic capture tools, in [40], namely: gschem, Kicad and TinyCAD. There are numerous other open-source schematic capture tools presented in the literature, namely: Kactus2 [41], Fritzing [42], QElectotech [43] and Qucs (Quite universal circuit simulator) [44]. Caneda (Circuits and Networks EDA) is an open-source attempt to port Qucs from Qt3 to Qt4.

2.2.4 Intellectual Property (IP) Entry

This methodology involves the use and reuse of previously developed cores. The use of pre-designed and pre-verified design blocks promote a higher level of design reuse [45]. Meiyappan et al. explain, in [45], that design reuse is the most promising technique for increasing the productivity of designers.

Parameterization is considered by many to be an essential part of making reusable IP cores [45, 46]. A single highly parameterized core can represent many non-parameterizable or "static" cores due to the flexibility associated with parameterization [46]. A study shows that, under ideal conditions, design time is halved when (re)using static cores and reduced by as much as an eighth when (re)using parameterizable cores [47].

Due to the wide variety of tools which facilitate with the delivery of IP cores, integration of cores, from different sources, can be difficult [46]. A study has shown that, for IP reuse to be viable, the costs associated with reuse must not exceed 30% of the cost of recreating the respective core from start to finish [46, 47]. The term "socketable IP"

has been used by Xilinx to refer to IP blocks that can be connected together without significant cost [48].

2.2.5 Electronic System Level (ESL)

This field has developed in order to address the increasing needs, of system developers, for more efficient and less time-consuming development methodologies [10]. Several names are used, in the literature, to refer to these methodologies such as: Electronic System Level (ESL), High-level synthesis (HLS), algorithmic synthesis, or behavioral synthesis [10].

The field aims to raise the level of abstraction by making use of languages primarily developed for software systems, such as C/C++ and Java, to describe digital hardware systems [49].

The input to these tools is a high-level language such as: C/C++, Java or Python instead of a traditional HDL [10]. Most of these tools however generate an HDL representation, of the designed system, in order to interface with tradition design tools [10]. Table 2.1 lists a number of HLS tools, presented in the literature, as well as their base high level language.

Table 2.1: High Level Sythesis Tools: listing HLS tools and their base high level language.

Base Language	**ESL Language**
C/C++	SystemC [50], Handle-C, FpgaC [51], Impulse C [52]
Java	JHDL [53]
Python	MyHDL [54], Migen [55]
Haskel	CλaSH [34], Lava [34]

2.3 Tools For FPGA-based Design

There are many different approaches, described in the literature, for implementing SDR systems [8]. Those approaches which make use of reconfigurable technology have been found to be the most popular [8].

This section will briefly introduce three existing frameworks for implementing SDRs on reconfigurable hardware. These frameworks include: Xilinx ISE, GNU Radio, and a Simulink-based flow. These frameworks offer a varying levels of abstraction. Figure 2.7 presents a block diagram which illustrates the level of abstraction these frameworks offer.

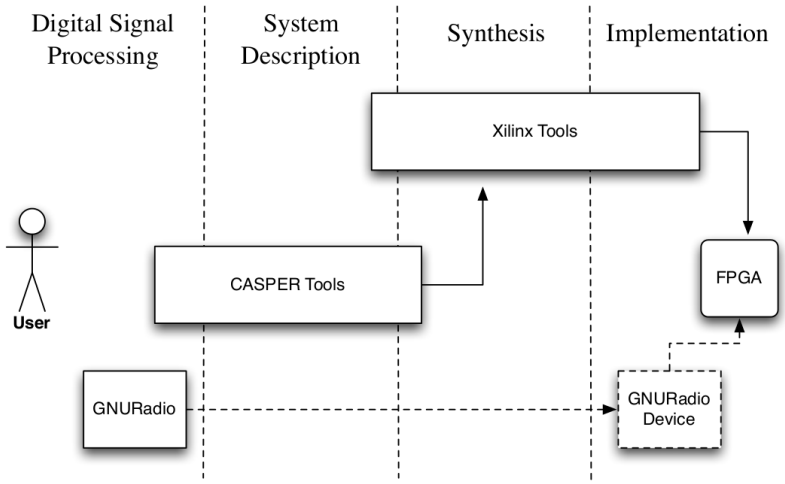


Figure 2.7: Abstraction Levels of SDR Frameworks: illustration of the varying levels of abstraction offered by some existing SDR frameworks. Adapted from [30].

Table 2.2: Stages of the Xilinx Design Flow: describing the various stages which constitute the Xilinx design flow. Adapted from [8].

Stage	**Description**
Design Entry	Xilinx allows users to specify designs utilising either text (Hardware Description Languages) or graphical (block diagram) or a combination of the two.
Design Synthesis	The synthesis of the design is the process whereby the user's design is prepared for later implementation on the target FPGA platform. The synthesis process is also required in order to create accurate models for simulation of the user's design.
Design Implementation	The implementation stage performs 'place and route' operations, which describes the layout of the logic on the FPGA, as well as the interconnection routes between logic elements.
Device Programming	Xilinx provides a mechinism for generating the bitstream required to program and configure the FPGA.

2.3.1 Xilinx ISE

Commercial FPGA vendors provide suites of comprehensive back-end tools for implementing and mapping to their hardware chips [56]. Integrated Software Environment (ISE) is a GUI tool, provided by Xilinx, to facilitate with the configuration of their FPGAs [8, 30]. The ISE environment consists of a variety of tools which support the primary and verification flows of the Xilinx design flow [8, 30]. Figure 2.8 illustrates the design flow for Xilinx FPGAs. A description of each of the stages involved in the Xilinx design flow is provided in Table 2.2.

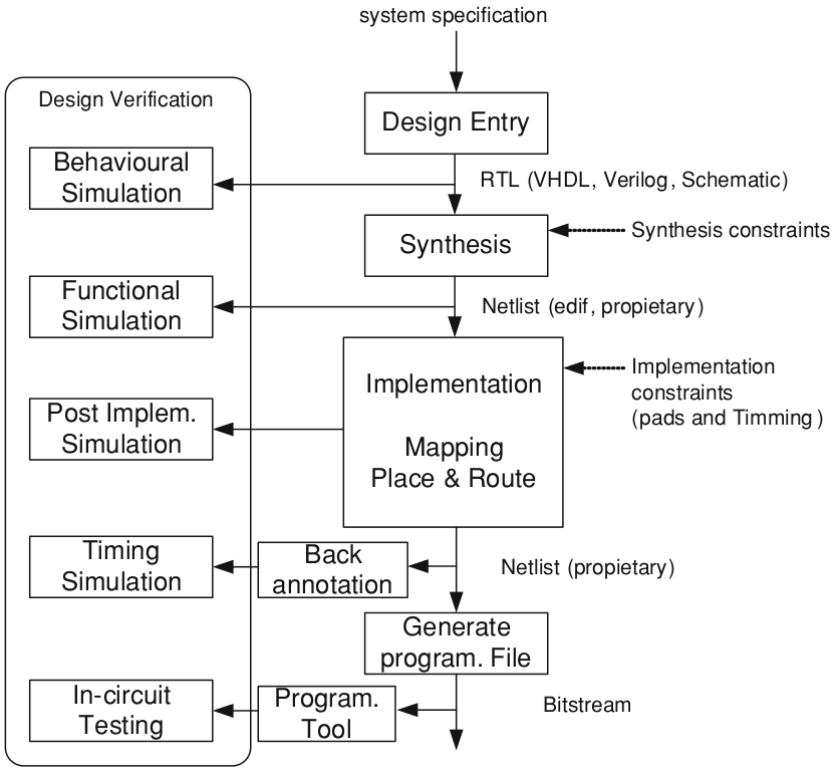


Figure 2.8: Xilinx Design Flow Diagram: block diagram illustrating the primary and verification stages of the Xilinx design flow for FPGA design. Adapted from [10].

Inggs [8, 30] and Chang [56] note that even with these tools, implementing applications on FPGA platforms requires the end-user to have a considerable amount of technical knowledge. Inggs explains in [8, 30] that the applicability of such tool is limiting due to plethora of complex nomenclatures associated with Xilinx products.

2.3.2 GNU Radio

GNU Radio is a free and open source toolkit for generating software-defined radio systems [13, 30, 57]. The framework has gained interest in the research community, due to

the universal move away from closed source firmware and proprietary chips with minimal customizability [13]. The GNU Radio toolkit comprise a library of signal processing primitives which are implemented in C++ [13, 30, 57]. A radio can be built by "glueing" these primitives together to form a graph [57]. Where the nodes of the graph are the signal processing primitives and the edges of the graph represent the data flow between connected primitives [57]. Blossom explains [57] that primitives, can be seen as blocks that process the infinite streams of data which flow from their inputs to their outputs.

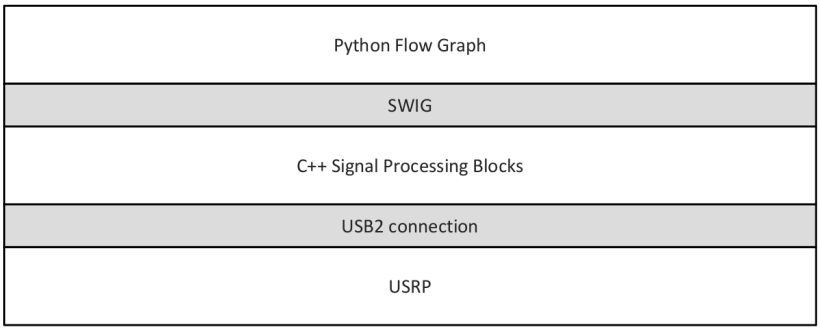


Figure 2.9: GNU Radio Architecture: illustrating the architecture of the GNU Radio toolkit. Adapted from [13].

Figure 2.9 shows the architecture of the GNU Radio toolkit. In this architecture, C++ is used for low-level programming while Python is used at a higher level to generate graphs [13]. The flow graphs can be constructed using C++, however it is far easier to connect primitives together using Python [57]. This is achieved through the use of SWIG2 [58], which provides an interface between Python and the C++ primitives [13, 58].

Listing 9.1 provides a code snippet of a tone generator implemented in GNU Radio. After all the required primitives have been defined and parameterised, a flow graph, *fg*, is initiated. The graph is built using *fg.connect* statements. Where the first and second arguments refer to the source and destination primitives respectively. A chain is established by using the destination primitive of the first statement, e.g. *chan_filter*, as the source primitive in the second statement.

The goal of GNU Radio is to promote a wider access to the EM spectrum [30]. Inggs [30] points out that an implication of this underlying goal is that, for the end-user to preform software-defined radio processing, the user should not require general purpose computing equipment beyond that which is readily available off the shelf. Typically GNU Radio is used in conjunction with the Ettus Research^{TM}'s Universal Software Radio Peripheral (USRP) [28] and their collection of daughterboards [29] [13, 30] as mentioned in section 2.1.3. Valerio [13] points out that, although the developers of GNU radio suggest

using the USRP, the toolkit is not restricted to this hardware platform.

GNU Radio Companion (GRC)

Due to the steep learning curve associated with new languages/libraries, a graphical interface was developed for GNU Radio [59]. The GNU Radio Companion (GRC) [60] is a GUI tool which generates GNU Radio flow graphs in a similar fashion to Labview and Simulink [59, 61]. Manicka [59] divides GRC into six components. These component are presented and described in Table 2.3.

Table 2.3: Components of GNU Radio Companion: describing the various components which make up the GRC tool. Adapted from [59].

Component	Description
Flow Graph	Signal processing blocks are interconnected to form a flow graph. GRC provides a scrollable window where signal processing blocks are placed and interconnected.
Processing Blocks	A signal processing block performs a signal processing task. These blocks are represented, in GRC, as rectangles with a label, indicating the name of the block, and a list of the blocks parameters.
Parameters	A parameter effect the function of the signal processing block. These parameters could, for example, alter the sampling rate or gain of the block. In GRC, parameters are displayed within the rectangle representing the signal processing block.
Sockets	Socket is the name given to the inputs and outputs of a signal processing block. A socket is represented, in GRC, by a smaller rectangle attached to the sides of the signal processing block and are typically labeled either "in" or "out". In GRC, sockets are colour coded with respect to their data type. Complex, floating point, integer, short, and byte data types would be represented by the colours blue, red, green, yellow, and purple respectively.
Connections	A connection joins the output socket of a one signal processing block to the input socket of another. Connections are represented, in GRC, by a line between two sockets. GRC prevents a user from connecting sockets of differing data types.
Variables	A variable value is accessible to all elements of the flow graph. Like signal processing blocks, variables are represented, in GRC, by a rectangular block. Variables can be used to share values with multiple parameters. For instance, if all parameters associated with sampling rate use the same sample_rate variable, changing this variable once is easier than changing all sampling rate parameters.

Figure 2.10 shows an implementation of a tone generator in GNU Radio Companion. This implementation closely resembles that of listing 9.1. However, in this implementation a noisy signal source is introduced as a third input to the addition block.

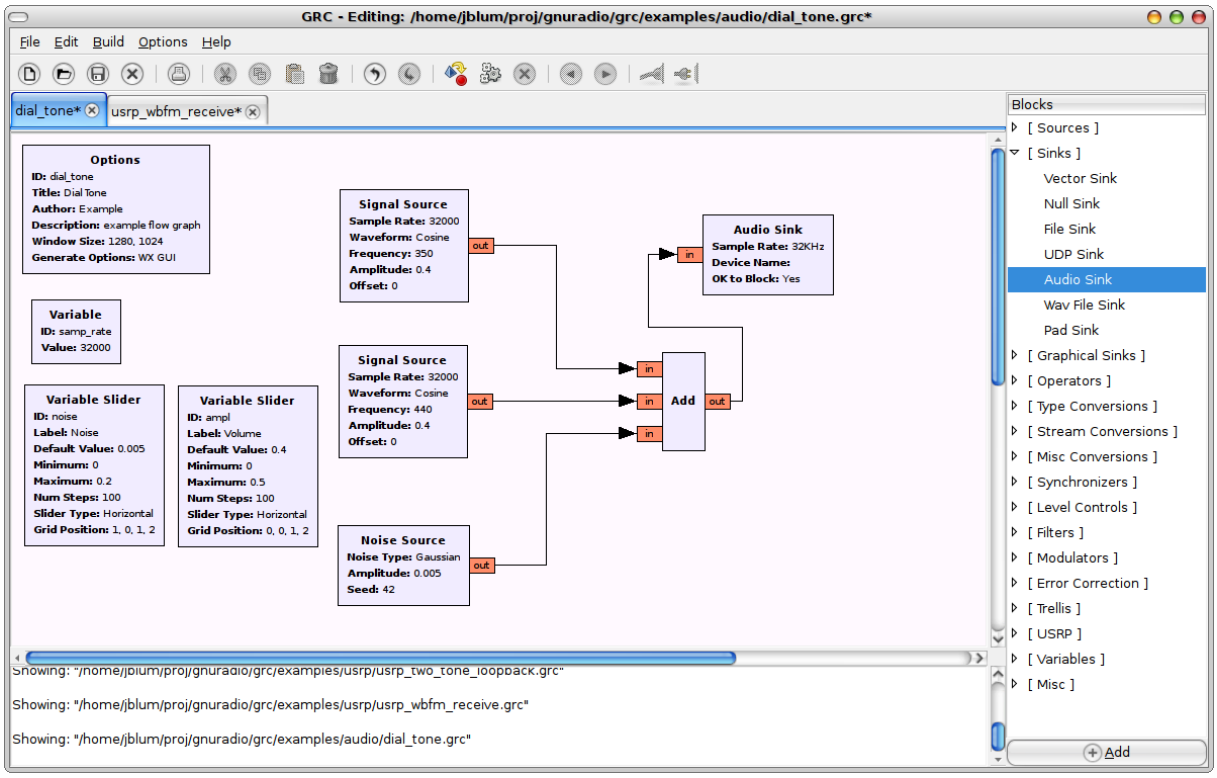


Figure 2.10: GNU Radio Companion Tone-Gen example: illustrating the use of GRC in building the tone generator described in Listing 9.1. Adapted from [62].

2.3.3 Simulink-based Design Flow

CASPER (Collaboration for Astronomy Signal Processing and Electronics Research) is a collaborative research group consisting of 13 collaborating institutions from 5 continents [63]. The group aims to simplify the design flow for radio astronomy instrumentation in order to reduce their production time. [8, 30, 63]. This is achieved by facilitating collaborative development through the promotion of design reuse of open-source hardware [64], software and gateware [8, 30, 63].

MSSGE (Matlab/Simulink/System Generator/EDK) is a Simulink-based toolflow commonly used to develop on CASPER's FPGA-based processing boards [65]. MSSGE extends on the original BEE toolflow [56] developed at the Berkeley Wireless Research Center (BWRC) [65].

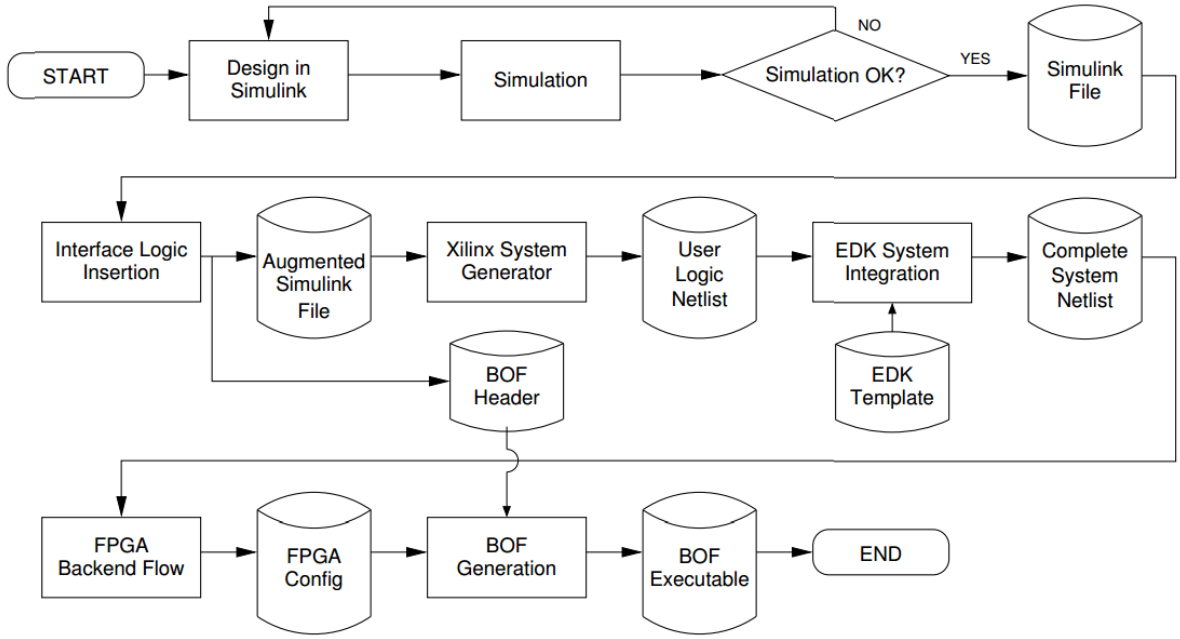


Figure 2.11: Simulink-based Design Flow: illustrating an automated Simulink-based design flow for FPGA design. Adapted from [31].

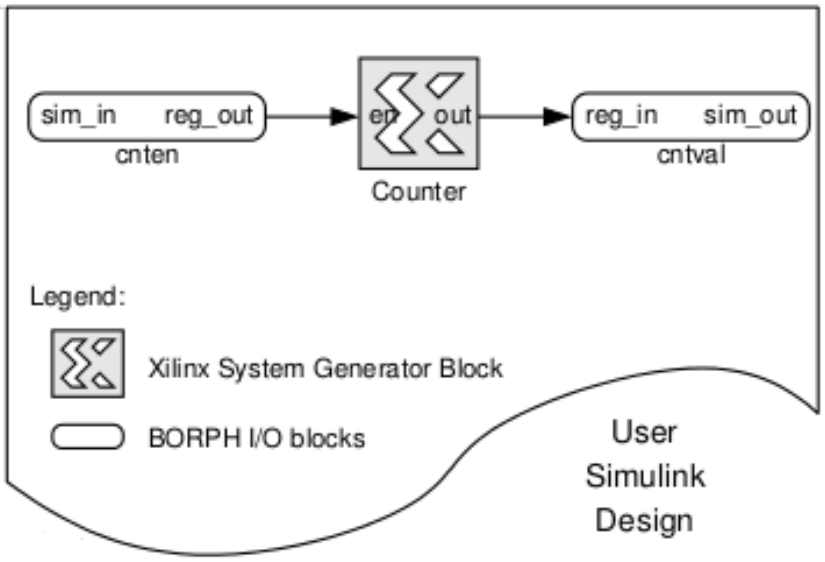


Figure 2.12: Simulink-based Design Flow Example: illustrating the design of a simple counter in the Simulink-based flow. Adapted from [31].

Figure 2.11 shows a block diagram representation of the Simulink-based design flow typically used in conjunction with the CASPER libraries and hardware platforms. Firstly, a design is described, graphically, in Simulink [31]. Users are provided with a wide range

of Xilinx System Generator (XSG) blocks, as well a CASPER hardware specific blocks, to utilise within a design [31]. Figure 2.12 shows a simple design consisting of a XSG *Counter* and two *registers*. The registers can be accessed through BORPH's IOREG virtual file system described in [31]. A completed design can then be simulated within the Simulink environment in order to verify its correctness [31]. Once a design has been tested and verified, through simulation, the user can proceed to the hardware generation stages of the design flow [31]. A netlist is compiled for each XSG block present in the Simulink design. These generated netlists are then tied together and synthesized with Xilinx Embedded Development Kit (EDK).

Although CASPER's platforms have successfully reduced the time and cost of developing radio astronomy instrumentation, Inggs [8, 30] identifies problems with the simulink-based toolflow. According to Inggs [8, 30], the software which makes up much of the toolchain, such as Xilinx and Matlab tools, are prohibitively expensive. Inggs goes on to state that the proprietary nature of the toolchain prevents users from improving the performance and functionality of the tools [8, 30].

2.4 VHDL Analysis

High level language analysis typically involves mapping source code onto an intermediate representation [66]. This is done by parsing the source into tokens in order to recognize the syntax of the language [66].

There are many tools that automate most tasks of lexical analyzer and parser generation for a variety of language grammars [66]. Bohm explains, [66], that this wide availability is due to extensive research in the field of scanning and parsing.

VHDL is a strongly typed description language with a context sensitive grammar [67]. Rahmati et al. [67] explain that, due to the language's strict restrictions, VHDL has become one of the most difficult language to analyze [67]. VHDL parsers are considered complex to implement due to the flexibility of the grammar [68]. However a few VHDL parsers have been implemented [12, 69]. Macko et al. discuss an ANTLRv3 parser developed to analyse VHDL code in [12] and Griffin mentions the development of a Bison/Flex based VHDL parser in [69].

2.5 Intermediate Representations

There have been numerous intermediate formats proposed to represent VHDL models [70]. There are two main approaches when it comes to VHDL data representation, namely the XML-based and object oriented approaches [11]. Jelemenska et al. [11] explain, that gen-

erally a XML based representation is more flexible than an object oriented approach [11].

Due to the complexity involved in processing the VHDL grammar, Reshadi et al. proposed HDML (Hardware Description Markup Language) [68]. HDML is an object-oriented data structure for representing VHDL [68]. HDML was developed to fulfill the needs of hardware designers as well as CAD tool designers [68].

Similarly, an ad-hoc XML schema was developed as an intermediate representation medium for VHDL models in [12]. Macko et al. explain, in [12], that XML was chosen due to its ability to preserve the hierarchical structure of VHDL designs as well as its extensive use and the availability of libraries [12].

IP-XACT is a standard described in IEEE Std 1685-2009 [71] that has been adopted by many in the FPGA industry [48]. The standard aims to provide EDA tools, IP providers and system design communities with a well-defined specification for documenting IP, in a vendor neutral manner, using meta-data [71, 72]. This meta-data fully represents the components and designs within the system [71] and can be used to configure, integrate, and verify an IP core in System on Chip (SoC) design environments [72].

Kamppi et al. make use of IP-XACT in a proposed graphical embedded system development environment presented in [41]. Once a design is complete, a VHDL generator converts the hierarchical IP-XACT design into a structural top-level HDL entity and generates the meta-data for the generated HDL component [41].

Arnesen et al. [46] discuss however, that the IP-XACT standard does not provide enough support for reconfigurable computing IP and that the standard my be better suited to SoC design [46]. IP-XACT lacks strong enough support for the parametric nature of reconfigurable IP according to [46]. Arnesen et al. [46] further mentions, that IP-XACT better suits predefined bus interfaces, common in SoCs, rather than arbitrary, IP specific interfaces present in reconfigurable computing [46].

A number of IP-XACT extension schemas have been proposed to combat some of the limitations present in the standard. Perry et al. present XCI, in [73], as a "bespoke" IP-XACT in order to address the problem of heterogeneous system design abstraction.While the XML schema presented by Rollins et al. [47], addresses limitations of IP-XACT as well as introduces better support for reconfigurable computing IPs [47]. This extension introduces support for parameterization, mathematical expressions, high level data types and tool generators [46]. However, this extension has become bulky and difficult to manage [46]. It is for this reason that Arnesen et al. [46] refine this initial attempt, with CHREC XML, by organising meta-data into three abstraction layers [46]. The ideas

presented in [46] were later used by Neely et al. in their system presented in [48].

Numerous other Extensible Markup Language (XML) schemas have been developed for use in context to VHDL, system design and modeling. These schemas include, but are not limited to, the following: NetPDL presented by Risso et al. in [74] as well as the Modeling Markup Language (MoML) [75] which aims to specify the interconnections of parameterized, hierarchical components.

2.6 VHDL Visualization

Commercial tool like Aldec's Active-HDL and Xilinx ISE provide editable graphical visualizations and schematic editors [12]. The Code2Graphics tool, packaged with Active-HDL, is capable of converting HDL code into a structural, graphical representation [12]. Modifications in the graphical model are reflected in the generated HDL code using the Graphics2Code tool [12]. However little or no abstraction, of clocking signals or bus interfaces, is offered.

Macko et al. present a VHDL visualization tool in [9] and [12]. Similarly a visualisation system for Verilog is discussed in [11]. However these visualisations are not editable and no system is provided for conversion back to the textual form.

2.7 VHDL Code Generation

Most of the high level synthesis tools, mentioned in section 2.2.5, generate an HDL representation of a designed system [10]. GEZEL [76], HDLmaker [77] and SIGNAL [78] are hardware description languages presented in the literature that provide VHDL and Verilog generation from a higher level description.

VHDL code is not only generated from higher level textual languages but from graphical representations as well. The Unified Modeling Language (UML) is commonly used in the literature to represent hardware systems [79, 80, 81]. The literature present methods for generating VHDL code form UML descriptions [79, 80, 81].

2.8 Xilinx Platform Studio (XPS)

XPS is part of the Xilinx Design Suite. The tool is typically used to build, connect and configure embedded processor-based systems. XPS makes use of graphical design views and wizards to facilitate with system development. It has the ability to configure and integrate plug and play IP cores from an IP catalog [82].

In XPS, processor and peripheral cores are known as *pcores* [82]. A pcore is defined as a directory structure. Figure 2.13 illustrates the directory structure for a typical pcore. In Figure 2.13, directories are represented by rounded blocks while files are represented by ovals.

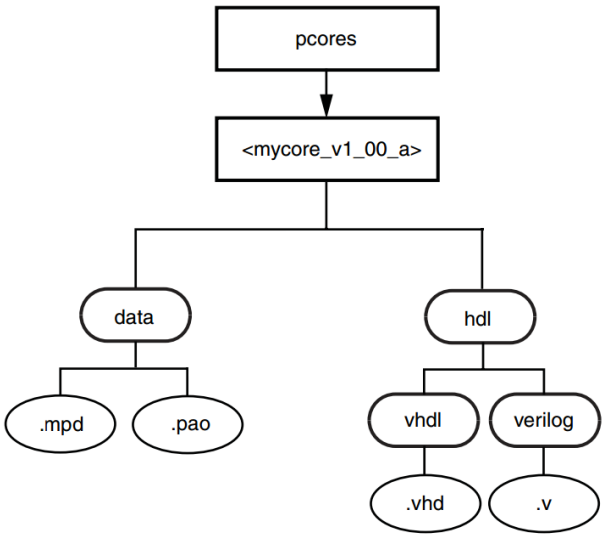


Figure 2.13: pCore Directory Structure: illustrating the structure of the pcores directory. Adapted from [82]

2.8.1 Microprocessor Peripheral Definition (MPD)

The Microprocessor Peripheral Definition (MPD) file defines the interface of the peripheral. The file lists ports and parameters associated to the core as well as the default parameter values and default port connectivity for bus interfaces. Parameters that you set in this file are mapped to VHDL generics or to parameters for Verilog. [82]

MPD files are text files and follow a specific format. The file consist of assignment commands which define options, busing interfaces, parameters and ports. The following is a typical component definition:

BEGIN peripheral_name
...
command name = value
...
END

The definition starts by defining the component's name and follows with a series of

assignments. Finally the definition terminates with the *END* keyword.

2.8.2 Microprocessor Hardware Specification (MHS)

The Microprocessor Hardware Specification (MHS) file defines the architecture of a design. This is achieved by instantiating and configuring bus architectures and components as well as their interconnections. Parameters that you set in this file override those in the MPD files and are mapped to VHDL generics or to parameters for Verilog. [82]

MHS files are text files and follow a specific format. The file consist of component instantiations which define parameters, busing interfaces and ports. The following is a typical component instantiation:

BEGIN peripheral_name
PARAMETER INSTANCE = instance_name
PARAMETER HW_VER = value # e.g. 1.00.a
...
command name = value
...
END

The instantiation starts by defining the component's name, followed by the instance's name and version. After these initial compulsory definition follows a series of assignments. These can include other parameters as well as port and bus interface assignments. Finally the instantiation terminates with the *END* keyword.

It should be noted that it is possible to generate invalid VHDL or Verilog, from a MHS description, even if the MHS file is syntactically correct [82]. This is due to the fact that MHS has a neutral format and sits on top of hardware description languages like VHDL and Verilog [82].

2.8.3 Bus Interfaces and Abstractions

In XPS, a bus interface is a grouping of related ports. Several components often have many of the same ports. This requires redundant port declarations for each component. For example, every component that connects to a bus will require the same ports defined and connected. A bus interface provides a high level of abstraction for the connectivity of a common interfaces. In its simplest form, a bus interface can be considered a bundle of signals. In this way components can use a bus interface as if it were a single port.

Bus standards can be defined using the BUS_STD keyword, in the core's MPD file,

as follows:

OPTION BUS_STD = value

, where the value can be one of the predefined bus standards supported by Xilinx's or a user defined standard.

Another core can be defined to connect to a bus standard by default. This is achieved by using the BUS_INTERFACE keyword, in the core's MPD file, as follows:

BUS_INTERFACE BUS=bus_label, BUS_STD=bus_std, BUS_TYPE=bus_type

, where the BUS keyword labels the name of the interface while the BUS_STD and BUS_TYPE keywords define the standard and relationship type respectively. The standard can be set to one of the predefined bus standards supported by Xilinx's or a user defined standard. The bus type represents the relationship of the interface to the connection. In other words the type can represent a master or a slave relationship.

Finally an instantiated component's can be connected to a bus through the use of the BUS_INTERFACE command, in the MHS description, as follows:

BUS_INTERFACE bus_label = bus_instance_name

,where the label represents the component's interface and the name represent the bus that the instance is connecting to.

2.8.4 HDL Wrapper and Top-level Generation

The Hardware Platform Generation tool (Platgen) synthesizes each IP core instance, found in a given hardware design, using Xilinx Synthesis Tool (XST) compiler. Platgen generates wrappers for each of the instantiated IP cores as well as generates the top-level HDL file that interconnects all the IP wrappers.

Figure 2.14 illustrates the Platgen flow. The figure shows the process of taking a MHS system definition as well as IP core MPD definitions and producing system-level HDL and wrappers. The diagram also shows how the top level HDL, wrappers and pcore HDLs are passed to XST for synthesis.

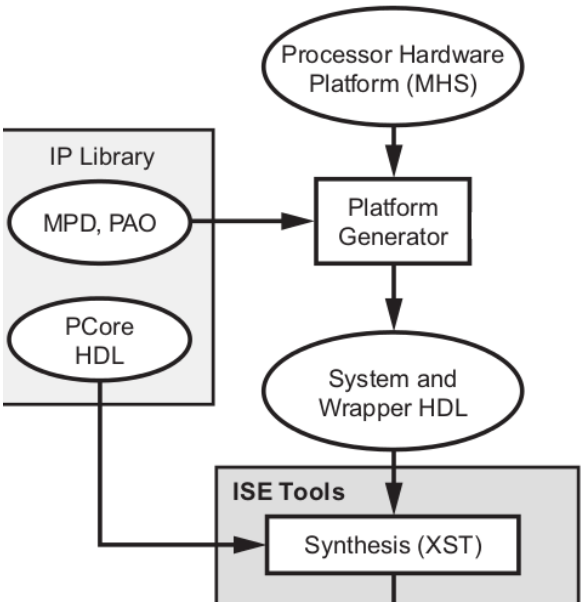


Figure 2.14: PlatGen design flow: illustrating HDL code generation from a MHS specification. Adapted from [83].

Chapter 3

Methodology

The chapter begins by explaining the processes involved in the research and development of the work presented in this dissertation. The chapter goes on to list the various elements that make up the development and experimentation environments. A set of experiments, that will be performed in order to verify the correctness of the designed tool, will then be presented. The chapter concludes with an explanation on how data, resulting from experimentation, will be collected and analyzed.

3.1 Phases of Development

Figure 3.1 is a block diagram illustrating the flow of processes involved in the research of the work presented in this dissertation. This section will describe the overall flow as well as describe each process involved in the flow.

Initial Meetings
Firstly, meetings with supervisors and experts in the field will be scheduled. During these meetings a Terms of Reference is drawn up. The Terms of Reference should specify what is required of the system as well as the expected deliverables. As part of the Terms of Reference, initial user requirements for the project are established.

Requirements Review
The requirements established during initial meetings are often vague and ambiguous. During a requirements review stage, these initial requirements are studied and analysed in order to clearly understand their meaning and scope. Figure 3.2 illustrates the flow of this requirements review stage. The outcome of this stage is a set of clearer, but not necessarily unambiguous, set of reviewed requirements. These reviewed requirements are compared with the initial user requirements to make sure that the new set of requirements still satisfy the initial ones. This could repeat multiple times until a set of commonly agreed upon functional requirements are established. These functional require-

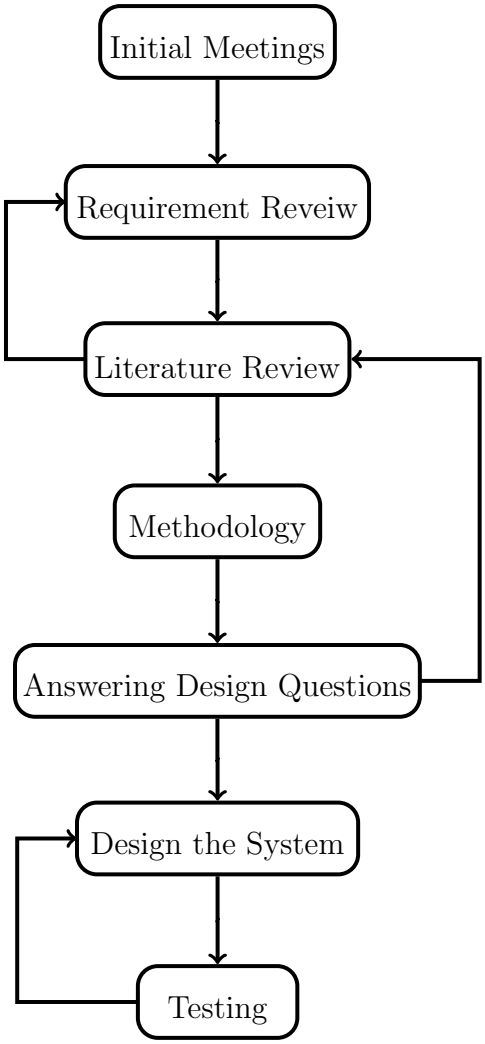


Figure 3.1: Methodology Outline: illustrating the flow of processes involved in the research of the work presented in this dissertation.

ments clearly specify what the end product should do as well as the scope to which the system needs to satisfy these requirements.

Figure 3.2: Requirements Review: illustrating the flow of the requirements review stage.

Literature Review

In order to fully understand what the requirements of the system should be, literature and related work needs to be researched and reviewed. After an analysis of the successes and shortcomings of past work one can better understand what is required of the system.

With a better understanding of what the "state of the art" is as well as what others have discovered, invented and developed, the requirements of the system may need further review. The output of the literature review stage is a body of research as well as a clearer understanding of where the project fits in with the work of others.

Designing the System

The design of the system will follow a Model-View-Controller (MVC) architecture. Figure 3.3 presents a block diagram illustration of the steps involved during the design stage. First, the *model* will be developed, followed by the *controller* that manipulates the *model*. Lastly the *view* will be built in order to visually represent the *model* as well as call the *controller* to update it. Each phase of the system design stage will be accompanied by a testing procedure.

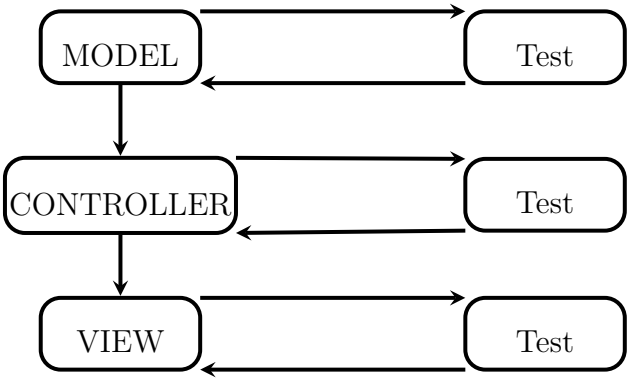


Figure 3.3: System Design Phases: illustrating the flow of the system design stage.

MODEL: The design stage begins with the development of an object-oriented (OO) library. This library will facilitate with the storage and manipulation of SDR elements. SDR blocks will be broken down into their atomic components and a Class will be written for each atomic element. During this phase of development, the OO library will go through testing.

CONTROLLER: In order for a user to interact with the *model*, a *controller* needs to be developed. The *controller* will be responsible for preforming actions on the *model* such as: adding, deleting, moving and connecting components. During the design of the controller, it will be tested in isolation and together with *model* library.

VIEW: After a suitable OO library has been defined, to store the atomic elements that make up SDR components, and functions designed to interact with this library, a graphical *view* can be developed to represent each of the OO classes. While developing the view, it will be tested together with the *model* and the *controller*.

3.2 Development and Experimentation Environments

A number of different hardware devices as well as software packages and libraries will be used during the development and experimentation of the proposed tool. This section lists the specifications for these environments as well as the versions of the software packages and libraries installed on the various hardware platforms.

3.2.1 Environment

The specifications for the development and experimentation environment are:

- **Desktop Computer**
  - **Memory:** 2,0 GiB
  - **Processor:** Intel Core2 Duo CPU E7500 @ 2.93GHz x 2
  - **OS type:** 64-bit
- **RHINO SDR Platform**
  - **FPGA:** Xilinx XC6SLX150T
  - **Processor:** Texas Instrument Sitara AM3517
  - **Operating System:** BORPH

3.2.2 Programs/Applications

Development, testing and experimentation will be preformed in the Linux environment. This Linux environment will be the Ubuntu 12.04 LTS, 64-bit, distribution. Installed software packages for the development and experimentation are:

- **Xilinx 14.1:** The Xilinx software suite typically used in conjunction with Xilinx FPGAs.
- **Qt Creator 2.4.1:** Qt Creator is an easy to use Integrated Development Environment (IDE) used together with the QtSDK library.
- **Python 2.7:** Python was chosen because it provides a platform for rapid prototype development as well as the ability to easily integrate with Qt/C++ application through libraries like PythonQt. Python also provide a platform for graphing results using the Matplotlib library.

3.2.3 Libraries

Libraries which will facilitate with the development of the tool, as well as the development of the experiments, will be installed on the development machine. Installed software packages for the development and experimentation are:

- **QtSDK 4.8.1:** Qt was chosen because it provides a platform for rapid GUI development.
- **PythonQt 2.1 for Qt4.8 [84]:** PythonQt provides the ability to integrate Python and Qt/C++ applications.
- **Matplotlib 1.3.0 [85]:** Matplotlib provides Python with the ability to plot and graph data in an easy to use way.
- **Adaptagrams:** Adaptagrams is a library for creating adaptive diagramming applications. The library has been used in the development of numerous, commonly used tools such as Inkscape and Graphiz. Github repository tree: b04ce8ac39 [86].

3.3 Experiments

A number of experiments will be performed in order to validate the correctness of the designed tool. This section outlines these experiments, the steps involved in the experimentation process as well as the purpose of each experiment.

3.3.1 Waveform Generator

The Waveform Generator experiment will involve building a waveform generator application in the proposed tool as well as generating the VHDL for the model. The experiment will include changing parameters in the graphical model in order to see the changes reflected in the generated VHDL.

Figure 3.4 illustrates a block diagram of steps involved in this experiment. The waveform generator use in the experiment will be based on the Numerically Controlled Oscillator (NCO) presented in [87]. However the design will be modified to support generic frequency and waveform types (e.g. Sine, Cosine, Square and Sawtooth). Once the design has been parameterised and its performance verified through simulation, the waveform generator will be imported into the proposed tool. A model will then be built using the imported waveform generator module and parameters changed within the GUI environment. Finally the VHDL, generated by the tool, will be verified against the expected results. The process of changing parameters and verifying the subsequently generated VHDL will be repeated for various waveform types and frequencies.

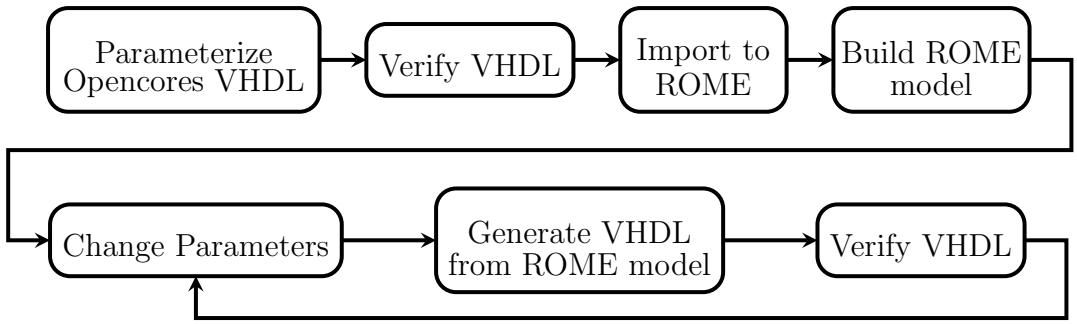


Figure 3.4: Waveform Generator Experiment: illustrating an outline of this section.

The purpose of this experiment is to test whether generics/parameters can be changed from within the proposed tool and whether these changes will be reflected in the generated VHDL.

3.3.2 Software Accessible Adder

The Software Accessible Adder experiment will involve building a bus based software accessible adder application in the proposed tool as well as generating the VHDL for the model. The experiment will include setting the values of two registers, on the FPGA, from the processor on the RHINO platform as well as reading the result of an addition operation from a third register.

Figure 3.5 illustrates a block diagram of steps involved in this experiment. Once a bus based adder has been developed and verified, it will be imported into the proposed tool. A model will then be built, in the GUI environment, using the imported modules. The level of abstraction of the model will be compared with that of Xilinx ISE and XPS. Finally the VHDL, generated by the tool, will be verified by implementing the code on the RHINO platform. The values of the two FPGA-based registers will be set from the processor and the value of the third register will be verified against the expected result of the addition operation.

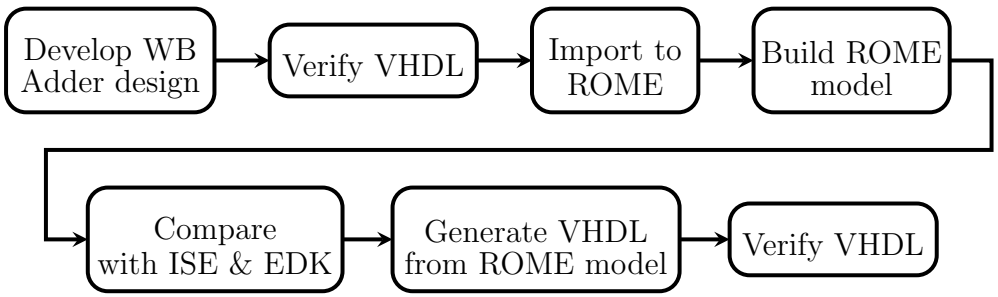


Figure 3.5: Wishbone Adder Experiment: illustrating an outline of this section.

The purpose of this experiment is to demonstrate the proposed tool's ability to abstract a bus structure away from the user as well as to compare the abstraction method of the proposed tool with that of Xilinx ISE and XPS.

3.3.3 Digital FM Receiver

The Digital FM Receiver experiment will involve building a frequency demodulation chain in the proposed tool as well as generating the VHDL for the model. The experiment will involve simulating the generated VHDL from a graphical model, and verifying the results.

Figure 3.6 illustrates a block diagram of steps involved in this experiment. First, an OpenCores design of a digital FM receiver, presented in [88], will be verified through simulation. The building blocks of the design will be imported into the proposed tool's GUI environment and a graphical model will be built using these blocks. The tool will be used to generate VHDL and this generated code will then be verified through simulation. The output waveforms produced through simulation will be validated against those presented in [88].

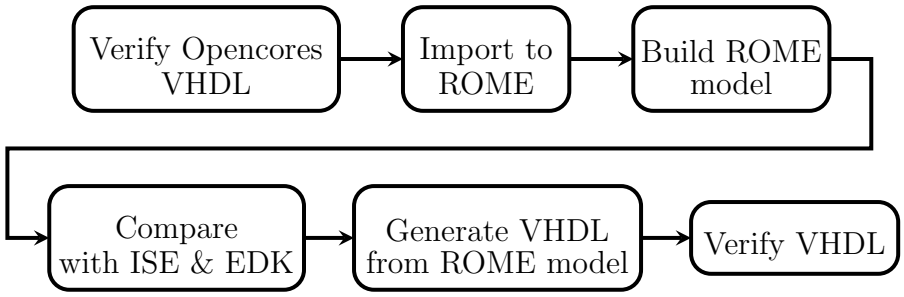


Figure 3.6: Digital FM Receiver Experiment: illustrating an outline of this section.

The purpose of this experiment is to demonstrate the use of the proposed tool in the context of a typical SDR application. The experiment aims to show that a datapath based SDR design can be built and verified using the proposed tool.

3.4 Data Collection Methods

Various different methods will be used to capture data resulting from the experiments mentioned in section 3.3. This section will briefly describe these data capturing methods.

All graphical output will be recorded through the use of screenshots. These screenshot may be cropped in order to focus on imported aspect of the output being analysed.

In order to verify the correctness of a VHDL design, the code will be simulated in Xilinx's ISim simulator.

A VHDL entity, file_src.vhd, will be created which will facilitate with the input of data into the simulator environment. This entity will synchronously read signed binary values from a text file and output these values on the raising edge of a clock cycle. Similarly, a VHDL entity, file_sink.vhd, will be created to facilitate with the collection of simulated data from the ISim environment. The entity will synchronously write data, that arrives at its input, to a text file as a signed binary number. Both file I/O entities will have generic file locations. This means that the input and output files will be set as parameter values.

A Python script will be written in order to visualise simulator input and simulated output data. The script will read the desired file line by line. A twos complement operation will be performed on each signed binary number. The values will be converted to an integer and finally plotted in two dimensions. The x-dimension will represent time as a progression on samples, while the y-dimension will represent each sample value.

The processor on the RHINO platform will be used to verify the correctness of designs that either, can not be simulated or to verify that a design is implemented correctly on a FPGA. The RHINO's Linux kernel, BORPH [31], will be used to communicate data to the RHINO's FPGA as well to access data produced by the FPGA. This will be achieved by reading data from and writing data to virtual *IOREG* files created by BORPH for all hardware processes [31].

Chapter 4

RadiO Modelling Environment (ROME)

This chapter presents an introduction to the proposed system followed by an explanation of the methodology behind its design. The chapter concludes with a comprehensive look at all the graphical components of ROME and how they are implemented.

4.1 Introduction to ROME

ROME is a graphical modelling environment, designed to facilitate students and researchers with the development of and experimentation with SDR applications implemented on reconfigurable computing platforms. The tool was developed on the Qt/C++ framework due to its cross-platform capabilities and extensive graphical libraries.

The design of ROME draws much inspiration from existing SDR frameworks. Some of these existing frameworks are mentioned in section 2.3. ROME's front-end graphical interface was inspired by GNU Radio Companion and Xilinx ISE, while ROME's back-end was inspired by CASPER's MSSGE Simulink-based design flow. Like CASPER's MSSGE, ROME relies heavily on the XPS framework mentioned in section 2.8.

ROME can be seen as an alternative graphical interface to the XPS GUI. Like MSSGE and the XPS GUI, ROME sits on top of the XPS command-line interface. Figure 4.1 is a block diagram illustrating ROME's design flow.

First ROME's design environment is used to build a SDR system made up of pcores from an IP library. When a design is completed, a MHS file is generated, by ROME, which describes the designed system. A Python script, called XPS_GEN, was developed in order to link the Xilinx tools into the ROME environment. The script draws its inspiration from the *CASPER_XPS* Matlab script used in CASPER's MSSGE toolflow.

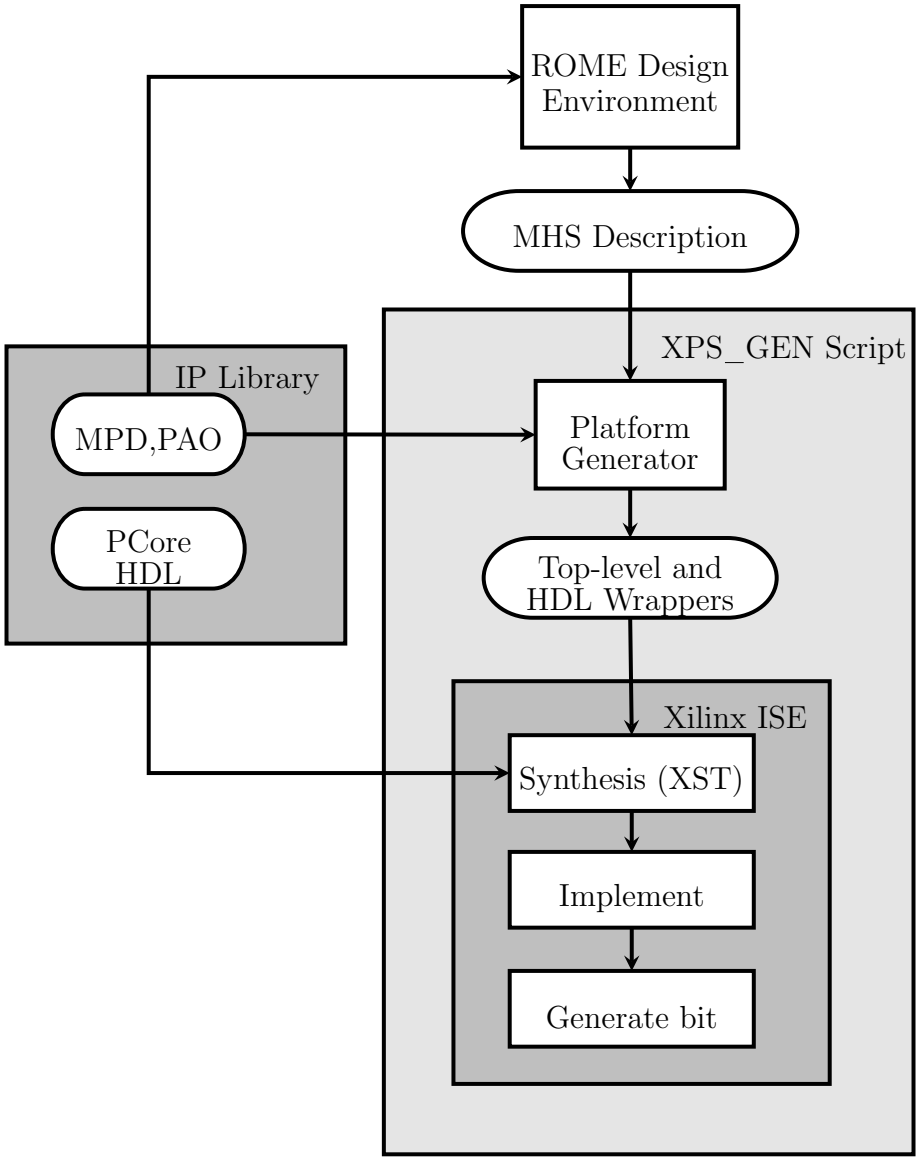


Figure 4.1: ROME Design Flow: illustrating design flow for the ROME environment.

The script coordinates operation for generating HDL wrappers, synthesising, place and route as well as generating a bitstream file for programming FPGAs.

4.2 Design Methodology

The methodology behind ROME follows a Model-View-Controller (MVC) styled design. Figure 4.2 illustrates the overview of the MVC design. Firstly, there is a model which holds all information about a user's design. The user is able to see a block diagram representation of their design through the model display or view. These models can be manipulated and altered by the user through a model manipulator or controller. The changes made to the model, through the controller, are updated in the view. In this way

the user receives feedback in the form of confirmation that their changes, to the model, have been committed successfully.

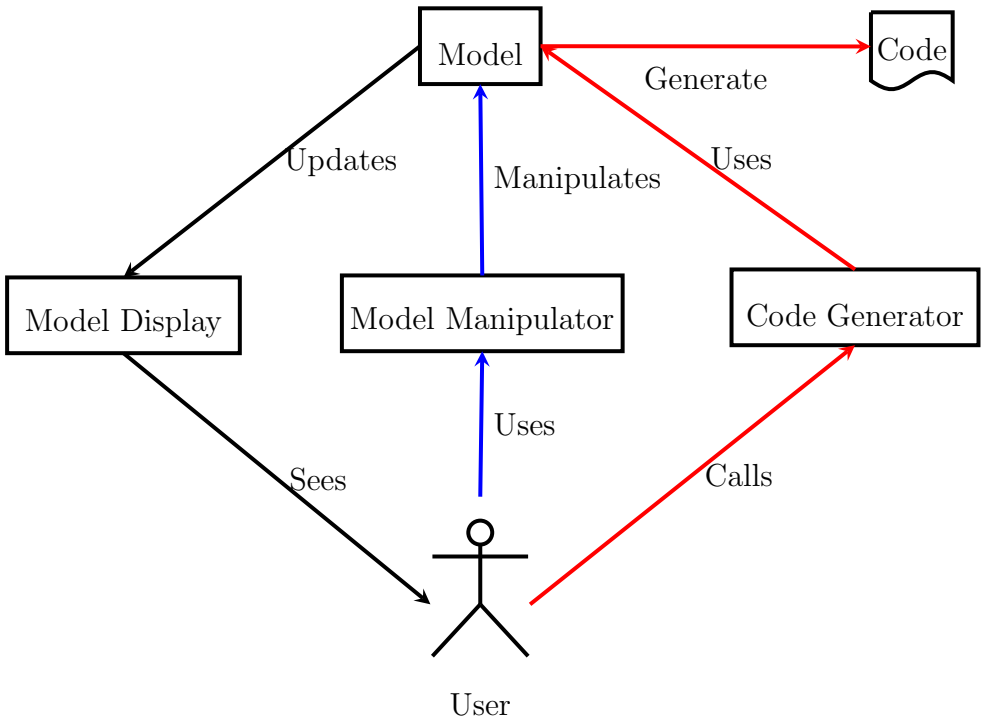


Figure 4.2: MVC Rome: illustrating an outline of this Model-View-Controller design methodology.

4.2.1 Model

The model of ROME's MVC implementation is represented by three C++ classes, namely: Component, Port and Wire. Each of these model classes, which inherit a Qt class, hold information about the specific model element, information on how the controller can change the model element, as well as information on how the model element should be drawn/represented by the view. Figure 4.3 presents a UML diagram illustrating the relationship between these model classes and the Qt framework.

A Component is equivalent to an Entity in VHDL and a Component in Verilog. The Component class inherits from the QGraphicsItemGroup, Qt, class. Listing 4.1 shows the structure of the Component Class constructor function.

Listing 4.1: Component Class Constructor

```
Component(QString name, QList<Port*> ports = QList<Port*>(), QString id = QString(),
    QGraphicsItem *parent = 0, QGraphicsScene *scene = 0);
```

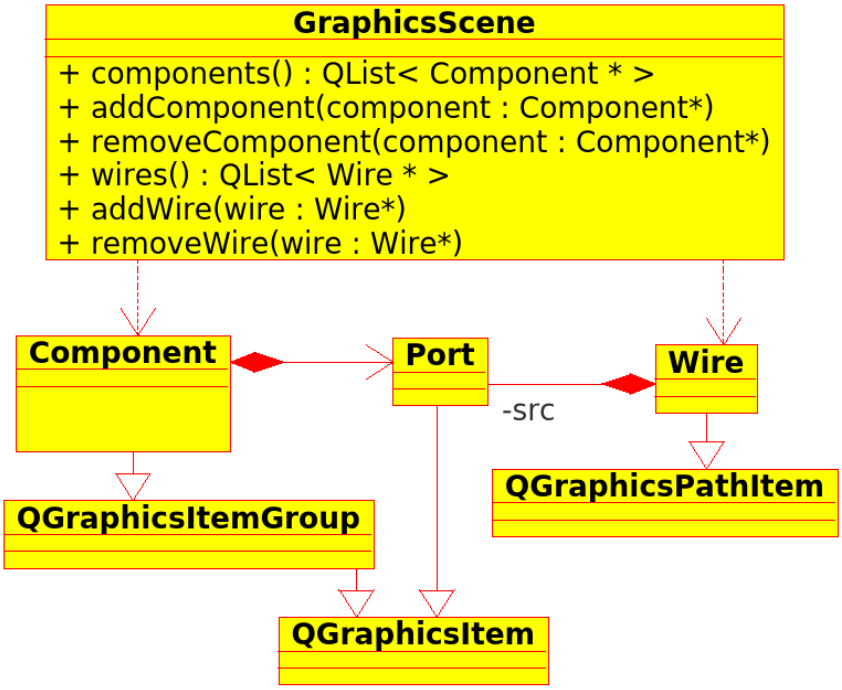


Figure 4.3: UML Class Diagram: illustrating the relationships between Components, Ports and Wires as well as their relationship with the canvas on which they are placed.

Each instance of the Component class has a name, an id, a list of ports as well as two maps: one for internal data and the other for external data. The internal data map holds internal variable values of a component. This is equivalent to Generics in VHDL and Parameters in Verilog. The external data map holds information about the component's connectivity between its own ports and the external pins of the FPGA. The class includes "get and set" functions, that provide the controller modules with access to alter class variables. Other functions include:

A Port, in ROME, is equivalent to Ports in VHDL and Verilog. The Port class inherits from the QGraphicsItem, Qt, class. Listing 4.2 shows the structure of the Port Class constructor function.

Listing 4.2: Port Class Constructor

```
Port(QString name = QString(), Direction direction = IN,
    DataType type = STD_LOGIC, QString msb = QString(), QString lsb = QString(),
    QString id = QString(), QGraphicsItem * parent = 0, QGraphicsScene *scene = 0);
```

Each instance of the Port class has a name, an id, direction and type. If the Port instance represents a vector the msb and lsb fields represent the most and least significant bits of the vector.

A Wire, in ROME, is equivalent to signal in VHDL or a wire in Verilog. The Wire class inherits from the QGraphicsPathItem, Qt, class. Listing 4.3 shows the structure of the wire Class constructor function.

Listing 4.3: Wire Class Constructor

```
Wire(Port* source, Port* destination, QString id = QString(),
        QGraphicsItem *parent = 0, QGraphicsScene *scene = 0);
```

Each instance of the Wire class has an id as well as a source and destination Port. The source and destination fields point to the memory location of the source and destination Port instances.

4.2.2 View

The view in ROME is responsible for representing the model graphically. This is achieved by drawing each element of the model on a "graphical canvas" which is visible to the user. The view not only displays the current state of the model but also allows access to the user to change and manipulate the model through the controller. In this way the view acts as an intermediary between the model and the user. The user can alter the model graphically through the view. The view then relays the user's request to the model through the control protocols.

This can be illustrated through a simple example. The user can see the model represented graphically by the view. The user drags the graphical representation of a component to another location on the screen. This action calls the controller's method associated with moving objects of a model. This controller method then calls the moved component's function which alters its x,y location values.

The view in ROME is also responsible for other graphical transformations commonly found graphical tools. These include: panning, zooming, selecting, layering, drag and drop and routing.

4.2.3 Controller

The controller in ROME is responsible for scheduling all interactions between the view and the model. In other words, the controller handles: changing component parameter values; the adding, moving and deleting of components and their connections as well as the undoing and redoing of these actions.

4.3 Graphical Components of ROME

Figure 4.4 shows the elements that make up the GUI of the tool. These elements are:

1. Main Menu: The main menu provides the user with functions commonly found in GUIs and EDA tools. Functions such as: Open, Save, Undo, and Redo.

2. Block Library: The block library is a list of all the available SDR building blocks. These blocks can be dragged from the library and dropped onto the design canvas.

3. Design Canvas: The design canvas provides a platform on which users can build SDR applications by connecting up blocks.

4. SDR Block: A SDR block in ROME is represented by a rectangle with its name positioned on the top of the block.

5. Property Menu: A user can access the properties of a SDR block by double clicking the block. The user is presented with a property menu for the selected block. In this menu the user can connect the block to the external environment or alter internal properties of the block such as: address, and data widths.

6. Compilation Menu: The compilation menu provides the user with access to the different stages of the Xilinx design flow.

7. Output Log: All output and error messages, from the Xilinx design flow, are displayed in the output log.

4.3.1 Main Menu

The main menu provides the user with functions commonly found in GUIs and EDA tools. Functions such as: Open, Save, Undo, and Redo. This section will describe how these functions were implemented in ROME.

Saving and Restoring Designs

One of the features common to numerous GUI applications is the ability to save the state of the application and the ability to restore the tool to a previously saved state. This feature was implemented in ROME in order to directly satisfy user requirement U1 mentioned in section 1.2.1.

In ROME a design schematic can be saved by writing all the information, necessary to reconstruct the design, to a file. A XML schema was developed in order to format this information in a formalised way. The developed XML schema was inspired by the

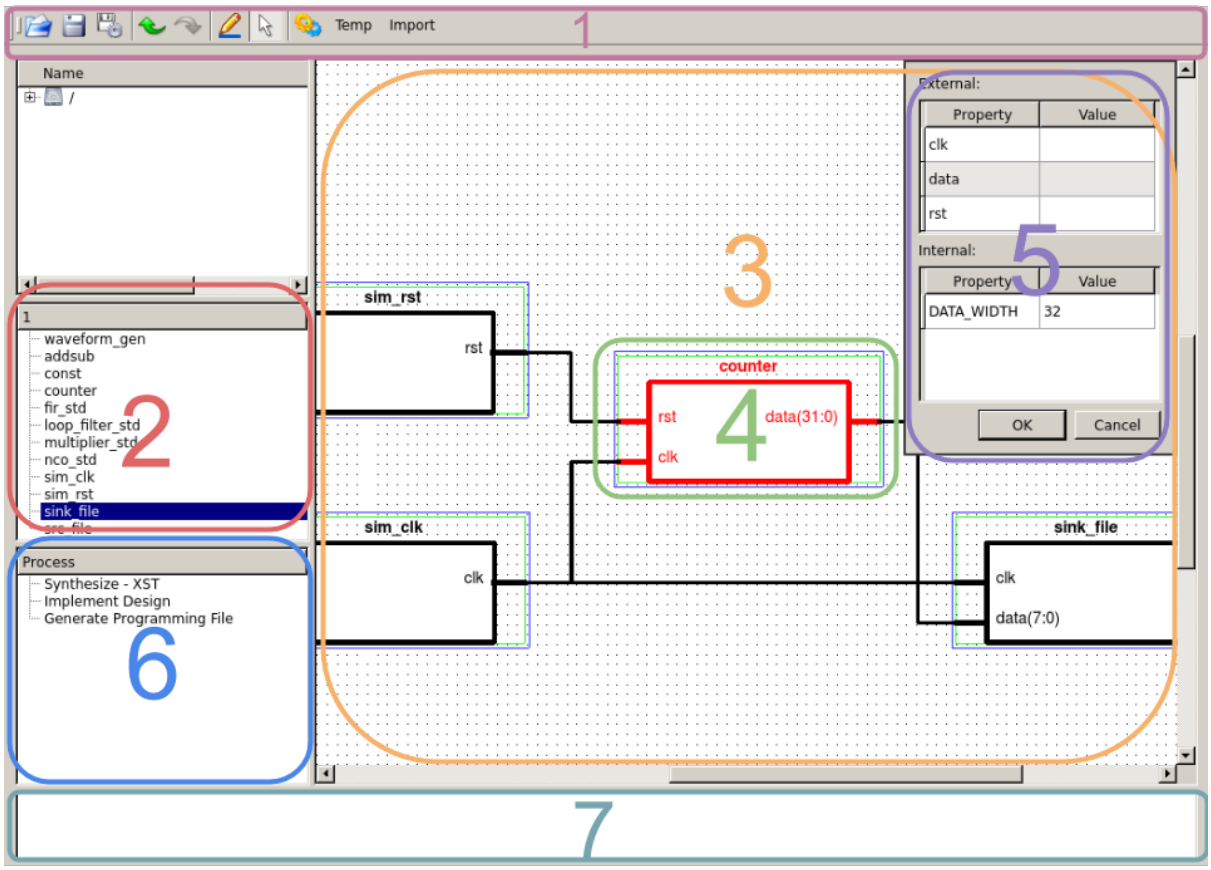


Figure 4.4: ROME's Graphical Components: illustrating the various elements that make up ROME's graphical user interface.

MoML schema mentioned in section 2.5. The schema consists of Wire and Component elements. Each Component element can also comprise of nested Port elements. Figure 4.5, represents an example design in the ROME environment and Listing 4.4 represent the corresponding XML file. In Figure 4.5, the example design comprises two components namely Func1 and Func2. The components, Func1 and Func2, comprise three ports and two ports respectively. A wire connects Func1 and Func2 at ports Z_O and X_I. In Listing 4.4, the wire element has the attributes "dest" and "src". This shows that the wire $W15$ connects port $P3$ on component $C8$ to port $P1$ on component $C9$.

Figure 4.6 illustrates the algorithm used to save ROME designs to an XML file. ROME makes use of Qt's QDomDocument classes [89] to create and parse XML formated files. First a *root* element is created. This element is named "Design" and acts as the highest most element of the XML file. The algorithm then iterates through all Wire objects present in the user's design. For each of these wire objects: create a corresponding XML wire element (named "Wire"), set all the relevant attributes of this element and finally append the wire element to the root element. A similar procedure is done for all Compo-

Listing 4.4: XML Schema: illustrating the developed XML schema for saving design structure to file. Associated to the design represented in Figure 4.5.

```
<?xml version="1.0" encoding="utf-8"?>
<Design>
  <Wire dest="C9:P1" id="W15" src="C8:P3"/>
  <Component x="24" y="41" id="C8" name="func1">
    <Ports>
      <Port dir="IN" type="LOGIC" id="P1" name="x1_I"/>
      <Port dir="IN" type="LOGIC" id="P2" name="x2_I"/>
      <Port dir="OUT" type="LOGIC" id="P3" name="z_O"/>
    </Ports>
  </Component>
  <Component x="24" y="54" id="C9" name="func2">
    <Ports>
      <Port dir="IN" type="LOGIC" id="P1" name="x_I"/>
      <Port dir="OUT" type="LOGIC" id="P2" name="y_O"/>
    </Ports>
  </Component>
</Design>
```

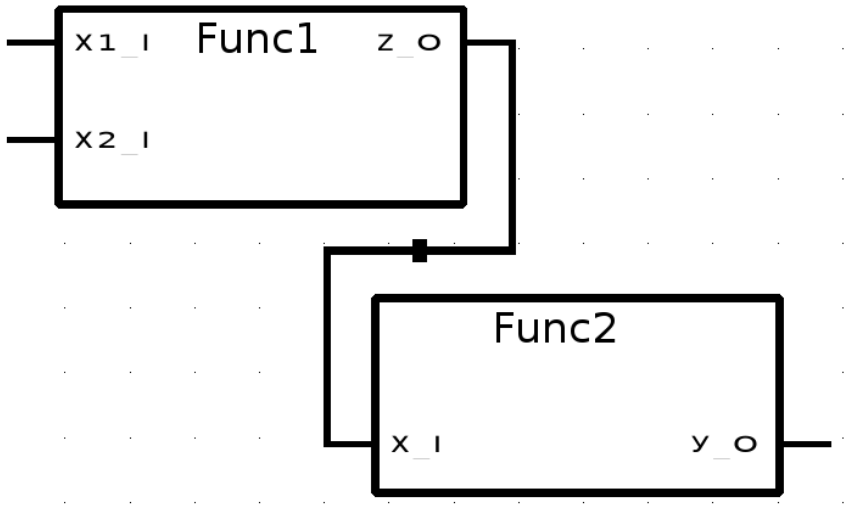


Figure 4.5: Example System: illustrating the connectivity of two components. Associated to the XML file presented in Listing 4.4.

nent objects in the user's design. However a component element can have "Property" and "Ports" as child elements. For each component element created, the algorithm iterates though the component object's properties and creates associated property elements and appends them to the current component element. Similarly, if the component has ports, a "Ports" element is created and appended to the corresponding component. Each port element is then created and appended to the "Ports" element.

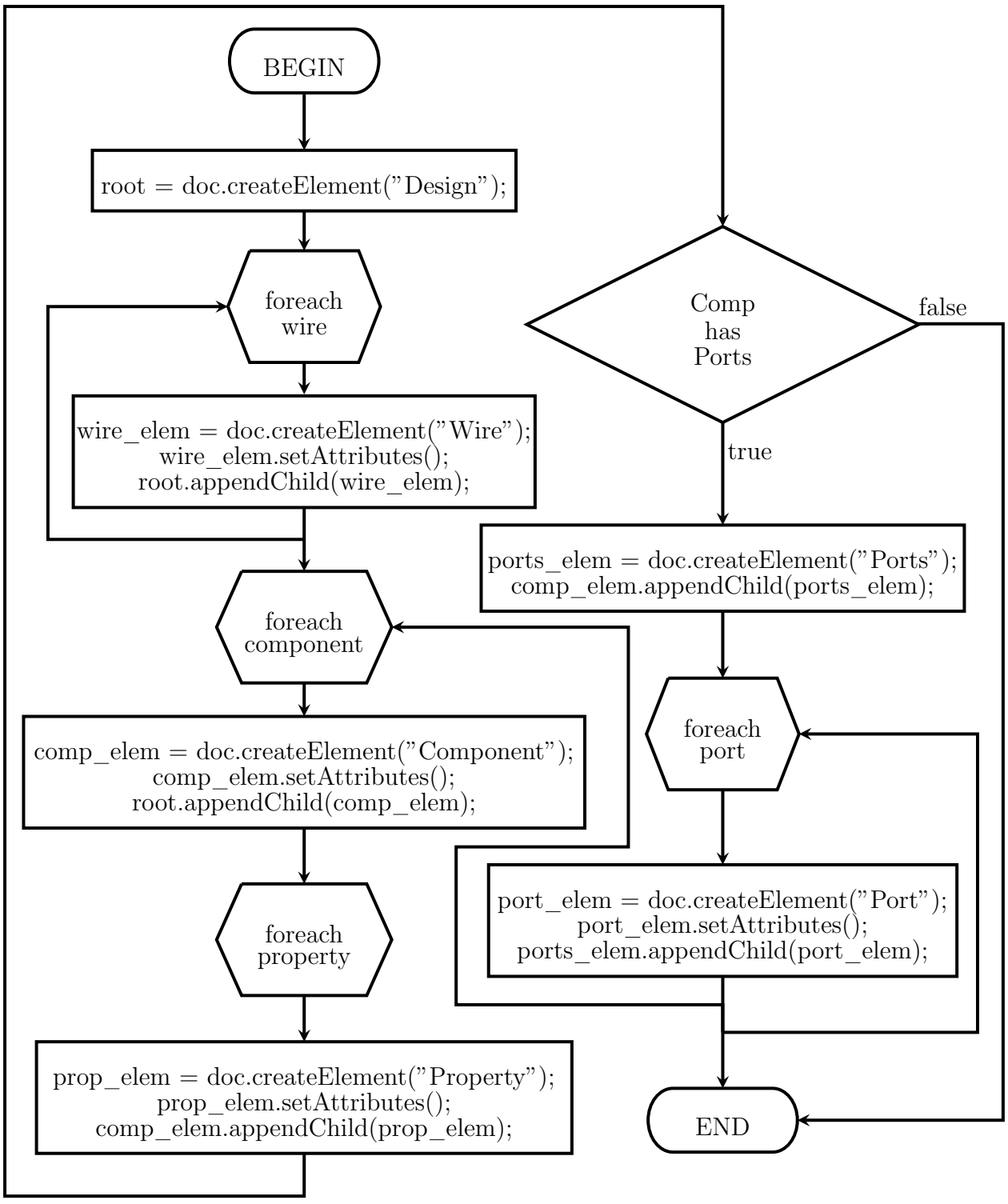


Figure 4.6: Saving Algorithm: A block diagram illustrating the algorithm used to save ROME designs to an XML file.

Undo/Redo Stack

ROME implements Qt's Undo Framework [90] for handling undo/redo functionality. The framework is based on the idea that all editing actions are implemented by instantiating

command objects and storing them on a command stack. The command object has knowledge of how to "undo" the change in order to restore the diagram to its previous state. It is thus possible to undo a sequence of changes by calling the undo function of the command object while traversing the stack downwards. Similarly, changes can be re-applied by calling the redo function while traversing the stack upwards. This feature was implemented in ROME in order to directly satisfy user requirement U1 mentioned in section 1.2.1.

4.3.2 Block Library

The block library is a list of all the available SDR building blocks. Figure 4.7 presents a screenshot of the block library, GUI component, illustrating available SDR blocks. These blocks can be dragged from the library and dropped onto the design canvas. This section will describe the process involved in importing HDL modules into ROME as well as how ROME compiles an IP labrary form these imported modules. This GUI component was implemented in ROME, in order to satisfy functional requirement F1 mentioned in section 1.2.2.

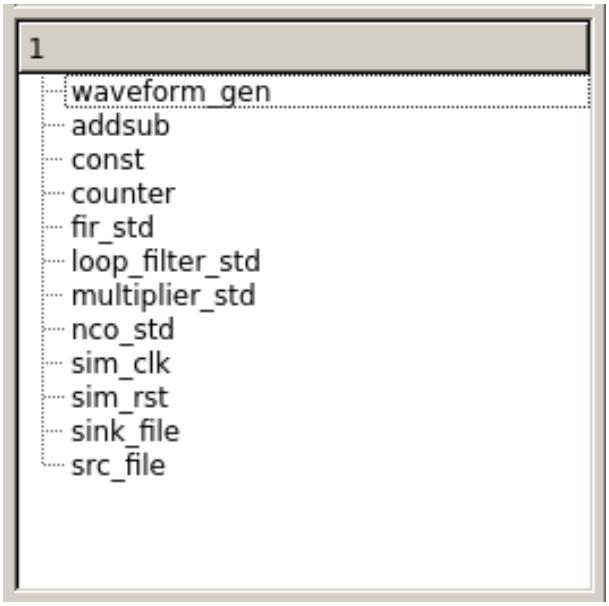


Figure 4.7: Block Library: illustrating a library of available SDR building blocks.

Importing Existing HDL Modules

The block library is a list of all available cores found in the pcores directory. Populating this list requires the user to first import existing HDL modules. In order to import existing HDL modules to the tool, all relevant structural information must first be extracted. This information includes: the number of ports as well as the direction, data type and size of each port.

HDL modules can be imported into ROME by using Xilinx Embedded Development Kit (EDK)'s Create/Import Peripheral Wizard. The wizard is a binary shipped with the Xilinx Suite as part of the Xilinx EDK package. It is typically used in order to import existing pcores to a new XPS project or to create an entirely new peripheral core. The tool can be called in one of two ways. It can be called with or without a GUI. The tool takes either HDL or NGC netlists as input and outputs a pcore directory structure, for the core, which includes a MPD file as well as the original HDL or NGC input files.

Figure 4.8 presents a screenshot of the GUI interface of the *createip* wizard, while the terminal command is presented as:

```
createip -batch -import "module" -ver "version" -dir . -lang mixed
    -type peripheral -hdl work "first_file.vhd" -hdl work "secornd_file.vhd"
```

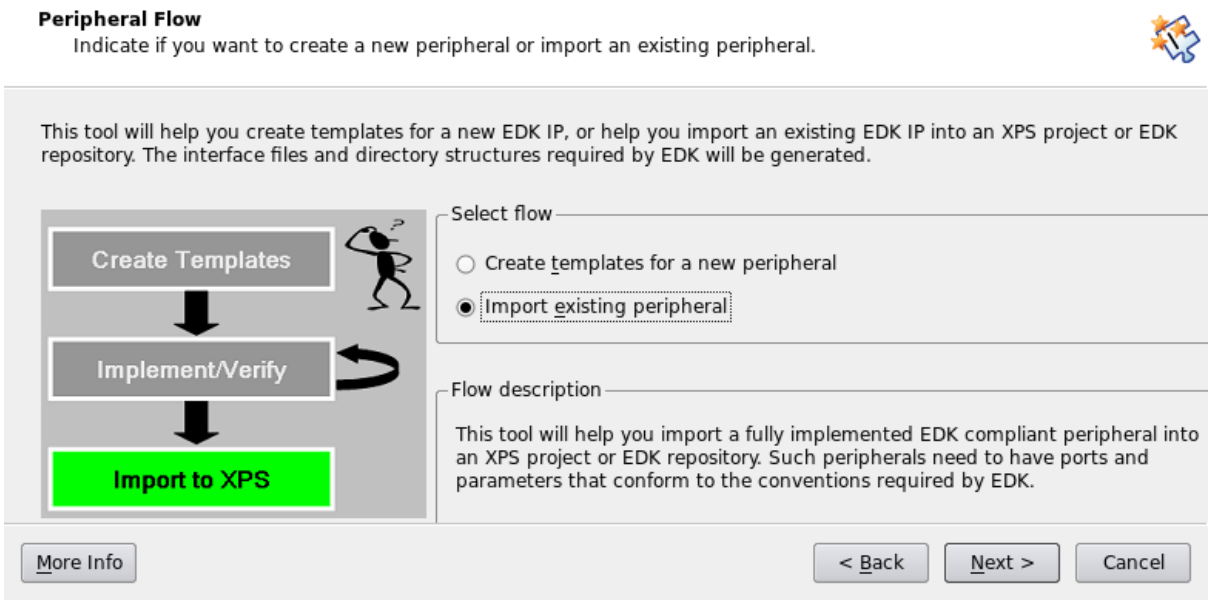


Figure 4.8: Xilinx Create/Import Peripheral Wizard: illustrating a screenshot of the createip wizard.

As mentioned in section 2.8.1, a MPD file is used by XPS to define the interface of a peripheral. The file contains a list ports and parameters as well as ports' default connectivity to bus interfaces and default parameter values.

Populating The Block Library

Before one can start building SDR applications in ROME, the block library needs to be populated. The library is a list of all SDR building blocks that are available to the

user. That is to say, it is a list of all the cores found in the application's search path. On startup, ROME updates the block library by searching the library path for Xilinx EDK pcores. Figure 4.9 is a flowchart illustrating the algorithm used to update the block library.

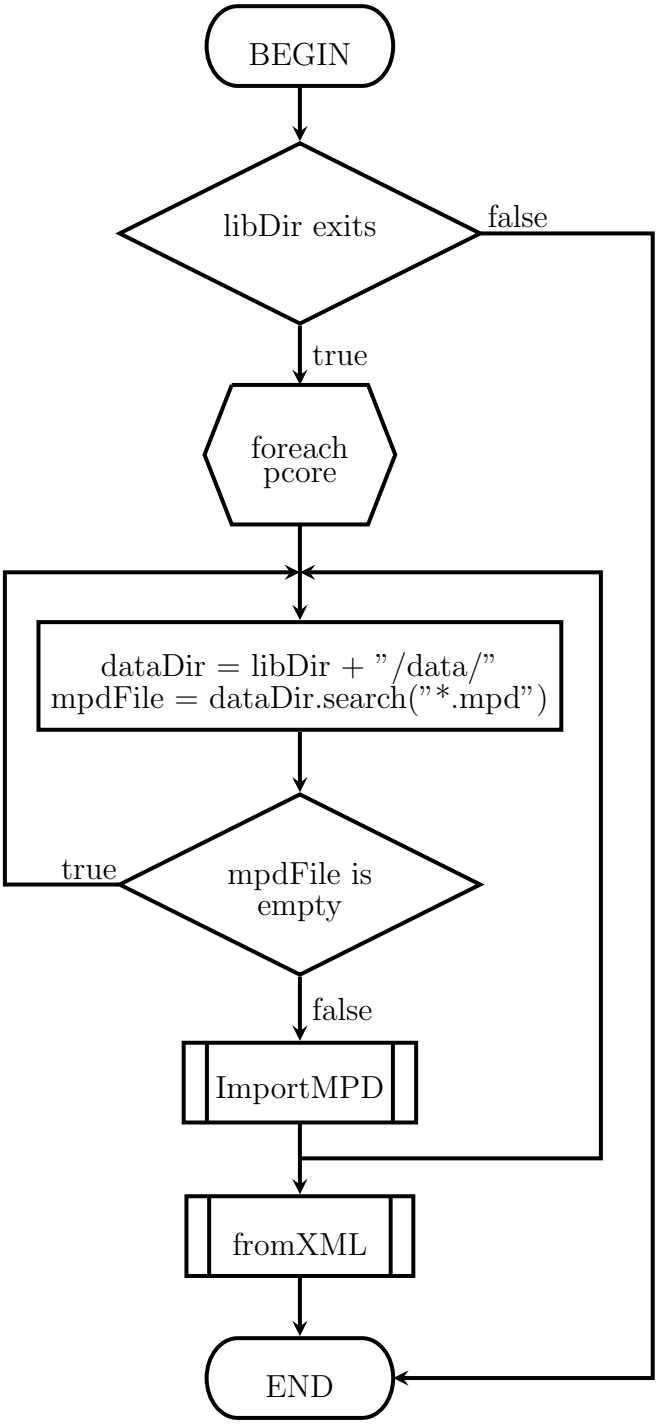


Figure 4.9: Update Block Library: a block diagram illustrating the algorithm used to update the block library.

The algorithm begins by checking whether the pcore directory exists. If so, search for the MPD file associated with each of the cores in the pcore directory. This is followed by parsing the information in all these MPD files and writing all relevant information to a file in a XML format. Listing 4.5 illustrates the format of the library XML file. Finally the XML file is parsed and each of the entries are placed in the library view.

Searching for available MPD files in the pcores directory is achieved through a recursive directory search algorithm. Once an MPD file is found, all relevant information is extracted from it. This is achieved through a regular expression based parser. MPD assignment commands use the following the format:

```
command name = value
```

The MPD parser extracts *name* and *value* information from MPD assignment commands such as: OPTION, PARAMETER, and PORT. The regular expression,

```
(^COMMAND\\s*)|(\\s*,\\s*)
```

, extracts a list of assignment expressions, for the given command, separated by a comma. Further extraction, of *names* and *values*, is achieved through the regular expression,

```
^\\s*([!-~^,=]*)\\s*=\\s*([!-~^,=]*)\\s*$
```

, when applied to each assignment expression in a loop. Bus standards are then extracted from the OPTION command, Data types and ranges from the PARAMETER command and directions and data width are extracted from the PORT command. All this extracted information is then written to a file in a XML format.

4.3.3 Design Canvas

The design canvas provides a platform on which users can build SDR applications by connecting up blocks. This section will describe the implementation of features such as: panning & zooming, drag and drop as well as automatic connection routing.

Panning & Zooming

The ROME environment is a Zoomable User Interface (ZUI). This means that the user is able to change the scale of the visible area of the view. In other words, the user can zoom in and out of objects of interest. The user can not only change the scale of the visible view, but can change the co-ordinates of the visible area as well. This is known

Listing 4.5: ROME Library XML file: illustrating a typical library file used to populate the block library.

```
<library>
 <component name="counter">
  <parameter value="32" type="NATURAL" name="DATA_WIDTH"/>
  <port mode="IN" name="rst"/>
  <port mode="IN" name="clk"/>
  <port lrange="(DATA_WIDTH-1)" mode="OUT" rrange="0" name="data"/>
 </component>
 <component name="fir_std">
  <port mode="IN" name="clock"/>
  <port mode="IN" name="reset"/>
  <port lrange="11" mode="IN" rrange="0" name="data_in"/>
  <port lrange="11" mode="OUT" rrange="0" name="data_out"/>
 </component>
 <component name="loop_filter_std">
  <port mode="IN" name="CLK"/>
  <port mode="IN" name="RESET"/>
  <port lrange="7" mode="IN" rrange="0" name="C"/>
  <port lrange="11" mode="OUT" rrange="0" name="D1"/>
  <port lrange="11" mode="OUT" rrange="0" name="D2"/>
 </component>
 <component name="multiplier_std">
  <port mode="IN" name="CLK"/>
  <port mode="IN" name="RESET"/>
  <port lrange="7" mode="IN" rrange="0" name="input1"/>
  <port lrange="7" mode="IN" rrange="0" name="input2"/>
  <port lrange="7" mode="OUT" rrange="0" name="output"/>
 </component>
</library>
```

as panning. The user can pan across the virtual view in both the horizontal and vertical axes. These features were implemented in ROME, in order to satisfy functional requirement F4 mentioned in section 1.2.2. Although F4 refers to the resizing of component blocks, panning & zooming functionality provides much the same benefits as resizable components.

The panning and zooming functionality is accessed through the mouse. The user can distinguish these mouse actions through the use of keyboard modifiers keys. Zooming is achieved by holding down the Ctrl key, on the keyboard, in conjunction with the scroll buttons on the mouse. Similarly panning is achieved by holding down the Alt key while clicking and dragging the canvas in the direction of choice.

Panning

Listing 4.6 provides the code used to implement the panning functionality in ROME. The implementation is provided in the *mouseMoveEvent* function which is inherited from Qt's QGraphicsView class. The modifier key is first checked. If the modifier key is not the Alt key, call the default functionality provided by the QGraphicsView class. However if the modifier key is the Alt key, the algorithm then determines how much the scene needs to be panned. This is achieved by subtracting the current pan point by the previous one. After a record of the current pan point is saved, the center of the scene is changed by the calculated panning difference.

Listing 4.6: Implementation of Panning functionality.

```
/* Provides the ability to Pan the visual area of the scene */
void GraphicsView::mouseMoveEvent(QMouseEvent* event) {

    if(event->modifiers() != Qt::AltModifier){
        QGraphicsView::mouseMoveEvent(event);
        return;
    }
    if(!LastPanPoint.isNull()) {
        //Get how much we panned
        QPointF delta = mapToScene(LastPanPoint) - mapToScene(event->pos());
        LastPanPoint = event->pos();

        //Update the center ie. do the pan
        SetCenter(GetCenter() + delta);
    }
}
```

Zooming

Listing 4.7 provides the code used to implement the zooming functionality in ROME. The implementation is provided in the *wheelEvent* function which is inherited from Qt's QGraphicsView class. Once it has been determined that the Control key is being used as a modifier, the current mouse position is mapped to the scenes co-ordinate space and QGraphicsView's default *scale* transformation function is called. After scaling is complete, the scene's center is shifted to accommodate for a horizontal or vertical translation

that may have occurred during the scaling process. A new center is calculated by adding the offset which resulted due to scaling.

Listing 4.7: Implementation of Zooming functionality.

```
/* Provides the ability to Zoom in and out (ie. scale the) visual area of the scene */
void GraphicsView::wheelEvent(QWheelEvent* event) {

    if(event->modifiers() != Qt::ControlModifier){
        QGraphicsView::wheelEvent(event);
        return;
    }
    //Get the position of the mouse before scaling, in scene coords
    QPointF pointBeforeScale(mapToScene(event->pos()));

    //Get the original screen centerpoint
    QPointF screenCenter = GetCenter();

    //Scale the view ie. do the zoom
    double scaleFactor = 1.15; //How fast we zoom
    if(event->delta() > 0) { //Zoom in
        scale(scaleFactor, scaleFactor);
    } else { //Zooming out
        scale(1.0 / scaleFactor, 1.0 / scaleFactor);
    }

    //Get the position after scaling, in scene coords
    QPointF pointAfterScale(mapToScene(event->pos()));
    //Get the offset of how the screen moved
    QPointF offset = pointBeforeScale - pointAfterScale;
    //Adjust to the new center for correct zooming
    QPointF newCenter = screenCenter + offset;
    SetCenter(newCenter);
}
```

Layering

When objects are moved around the graphical canvas, it is possible for objects to overlap. The view in ROME is capable of layering objects. In other words, should the object of interest overlap with another object, ROME will place the object of interest in front of any other object. Listing 4.8 presents the code that implements this functionality. The user can focus on any object by selecting it. All selected objects are placed on a layer on top, or in front, of all unselected objects.

Listing 4.8: Implementation of Layering functionality.

```
 QList<QGraphicsItem *> overlapItems = collidingItems();

 qreal zValue = 0;
 foreach (QGraphicsItem *item, overlapItems) {
      if (item->zValue() >= zValue && item->type() == Component::Type)
          zValue = item->zValue() + 0.1;
 }
setZValue(zValue);
```

Drag and Drop

Typically Drag and Drop (DnD) provides the user with a visual mechanism for transferring data between different applications or within an application [91]. ROME relies heavily on the Qt framework for DnD functionality. This feature was implemented, in

ROME, in order to satisfy functional requirement F1 mentioned in section 1.2.2. Documentation about how Qt's drag and drop system works, and how to use it, is available at [91].

The view in ROME handles two types of DnD operations, namely external and internal. An example of the first type of DnD operation is: dragging a component from the Block Library and dropping it on the graphical canvas. In this case, the graphical view is accepting a drop event external to itself. Moving an object from one location on the canvas to another is achieved by dragging it from its current location and dropping it on its final intended destination. This DnD operation is an example of an internal operation because the drop event originated from within the graphical view itself.

Automatic Routing

ROME relies heavily on the Adaptagrams project [92]. Adaptagrams is a set of four C++ libraries for adaptive diagramming applications. The project is part of research performed at Monash University under the Monash Adaptive Visualisation Lab (MArVL) [93]. libavoid is one of the C++ libraries [86] which make up the Adaptagrams project. The library is responsible for providing object-avoiding orthogonal [94] or polyline connector routing in diagram editor tools [95]. This feature was implemented in ROME, in order to satisfy user requirement U1 mentioned in section 1.2.1.

4.3.4 SDR Block

A SDR block in ROME is the graphical representation of a signal processing component. Each model instantiation of a component is represented graphically as a named rectangle with labels along its edges representing the component's ports. This GUI component was implemented in ROME, in order to satisfy functional requirement F5 mentioned in section 1.2.2.

The model classes presented in Section 4.2.1 implement the *paint* function, inherited from Qt's QGraphicsItem class, which describes how each class should be represented graphically on the design canvas. Qt's Paint System is discussed in greater detail in [96].

The *paint* function is called every time the display is refreshed. Instances of the Component class are "(re)painted" first. Since a Component consists of only a rectangle and a label, the painting procedure need only draw a rectangle and text. Once the Component's *paint* function is called, all of the Ports associated with the component are drawn.

Figure 4.15 illustrates a block diagram representation of the algorithm implemented by each instantiated ports *paint* function. The algorithm starts by checking whether the component, the given port belongs to, has been selected by the user. If so, the port will

be represented in red, else the port is represented in black. In ROME, input ports are positioned on the left of the component block while output and bidirectional ports are positioned on the right. If the port is an input port, a line is created extending from the left of the component towards the left and the font is set to left alignment. Similarly, if the port is an output or bidirectional port, the font is aligned to the right and a line is created from the right of the component extending towards the right. Before the port can be drawn, the algorithm checks if the port is a vector. If the port is representing a vector, the upper and lower ranges of the vector width are appended to the port label. Finally, the created line as well as a correctly aligned label are drawn on the design canvas.

4.3.5 Property Menu

ROME provides access to SDR processing blocks' properties from the GUI. A user can access the properties of a SDR block by double clicking the block. The user is presented with a property menu for the selected block. Figure 4.10 is a screenshot of a typical property menu in ROME. In this menu the user can connect the block to the external environment or alter internal properties of the block such as: address, and data widths. This GUI component was implemented in ROME, in order to satisfy functional requirements F2 and F5 mentioned in section 1.2.2.

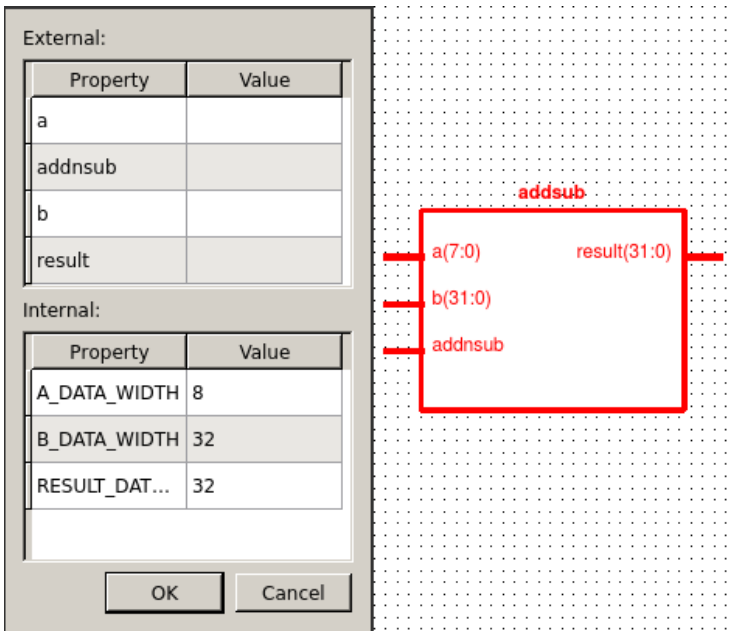


Figure 4.10: Property Menu: a screenshot illustrating the property menu for a adder/-subtracter SDR processing block.

Internal properties, like data widths, can have an effect on how the SDR block would be represented graphically. Data width can be used as a Generic or Parameter values in

order to increase flexibility of reusable VHDL or Verilog code respectively. In such a case the upper or lower range values of a port can change. ROME will redraw the SDR block, on the design canvas, with the newly set port ranges. ROME not only supports simple numerical value changes but supports more complex expressions as well. Expression are parsed, decoded and evaluated by Python through the PythonQt framework [84]. PythonQt is Python binding for the Qt framework which offers a simple way to embed the scriptable Python language into a C++ Qt applications.

Listing 4.9 illustrate the procedure used by ROME to evaluate expressions. After initializing PythonQt objects, variables and values are added to the evaluator. The added variable provide context to the expression. The expression can then be evaluated in the context of the added variable. In this way property values can be calculated when the expression depends on other property values or even other property expressions. For example an output port's data width may depend on the data width of the input ports. Addition, subtraction, multiplication, division and exponent operators are supports as well as the use of brackets.

Listing 4.9: Implementation of port expression evaluation functionality.

```
/* Provides the ability to evaluate expressions using Python */
QVariant Parser::evalExpression(QString expression, QMap<QString,QVariant> data)
{
    PythonQt::init();
    PythonQtObjectPtr mainModule =
            PythonQt::self()->getMainModule();
    // Add context variables
    for(int i=0; i< data.count(); i++){
        QString sdata = (data.begin() + i).key();
        QVariant vdata = QVariant((data.begin() + i).value().toReal());
        mainModule.addVariable(sdata,vdata);
    }
    QVariant result = mainModule.evalScript(expression, Py_eval_input);
    return result;
}
```

4.3.6 Compilation Menu

The compilation menu in ROME provides a link to the Xilinx tool for synthesis, place and route mapping, and bitstream generation. The user can select these different stages in the FPGA design flow, see Figure 2.5, through the graphical list of processes. This GUI component was implemented in ROME, in order to satisfy functional requirement F11 mentioned in section 1.2.2. Figure 4.11 is a screenshot of the compilation menu in ROME illustrating the list of available compilation processes. These processes include: Synthesis, Implement and Generate Programming File.

ROME links the compilation process, selected in the compilation menu, to the equivalent Xilinx operation through a Python script known as xps_gen.py. Listing 9.2 provides the full Python implementation of xps_gen used by ROME. ROME forks a new process

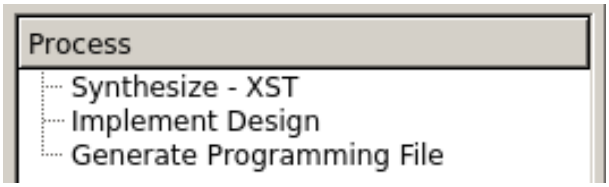


Figure 4.11: Compilation Menu: a screenshot illustrating the compilation options offered in the ROME environment.

for the execution of the Python script. This allows the user to continue using the GUI environment while the compilation process occurs in the background. The script first determines which compilation process was selected and then sets up the Python environment for the installed version of the Xilinx tools. Figure 4.12 illustrates the flow of the xps_gen script.

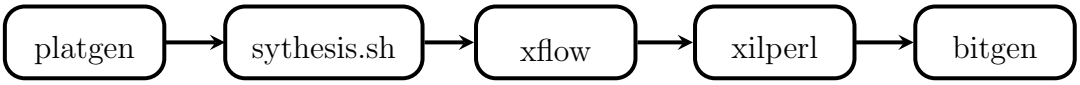


Figure 4.12: XPS Gen Flow: a block diagram illustrating the flow of operations performed by the xps_gen script.

The Python script calls Xilinx EDK's Platform Generator (PlatGen) with a generated MHS file as well as the MPD files, mentioned in Section 4.3.2, as inputs. Figure 4.13 illustrates the design flow for PlatGen. PlatGen generates HDL wrappers for each of the IP blocks in the design. A top level HDL file is then generated which includes the generated HDL wrappers in its hierarchy. The HDL code produced by PlatGen can then be passed to the ISE tool for synthesis. The xps_gen script runs a synthesis script, synthesis.sh, which is generated by PlatGen in order to synthesize the design. The synthesis script generates a Xilinx netlist file known as a NGC file for the design.

The Python script passes a set of option, a constraint file (UCF), associated with the design, as well as the synthesised netlist to the XFLOW tool. XFLOW, together with XILPERL, implement the design. Finally, a bitstream is produced by calling the BITGEN tool from the xps_gen Python script.

4.3.7 Output Log

The output log provides the user with feedback on the progress of the compilation process. The log links to the execution of the xps_gen Python script. Stdout messages are displayed in black text, while stderr messages are displayed in red text. Figure 4.14 is a screenshot of the output log illustrating the resulting messages displayed while preforming the synthesis operation.

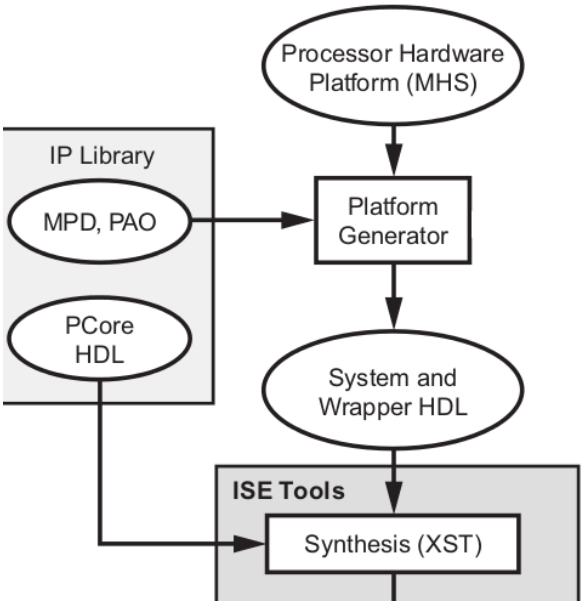


Figure 4.13: PlatGen design flow: illustrating HDL code generation from a MHS specification. Adapted from [83].

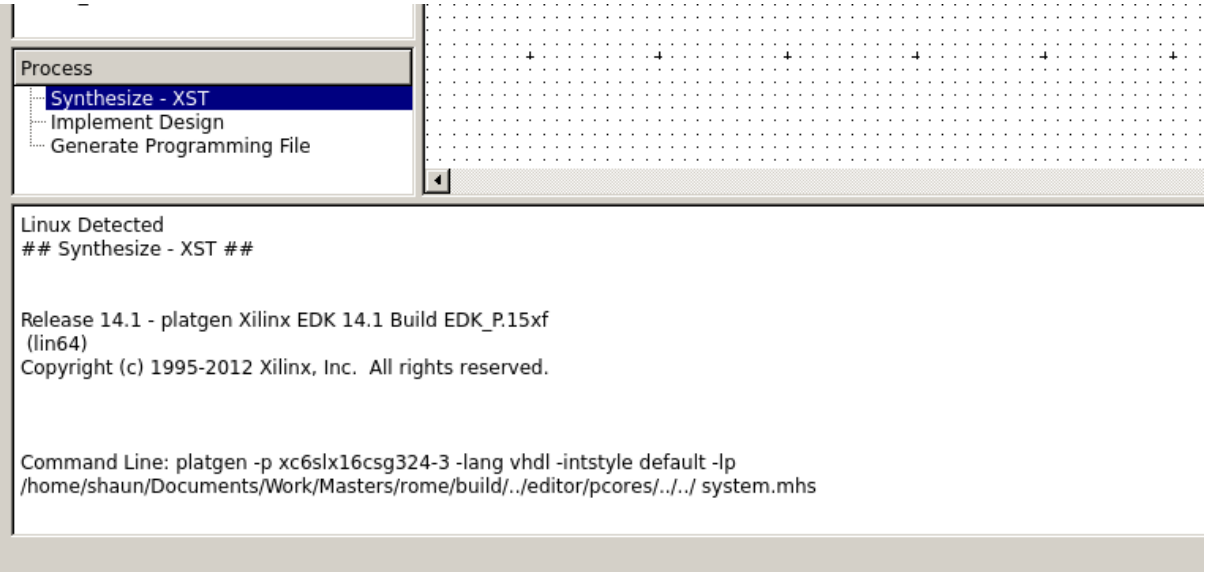


Figure 4.14: Output Log: a screenshot illustrating the resulting messages displayed while preforming the synthesis operation.

4.4 Conclusion

This chapter explained the overall design methodology of ROME. The chapter presented an introduction to the proposed system followed by an explanation of the methodology behind its design. The chapter then concluded with a comprehensive look at all the graphical components of ROME and how they are implemented.

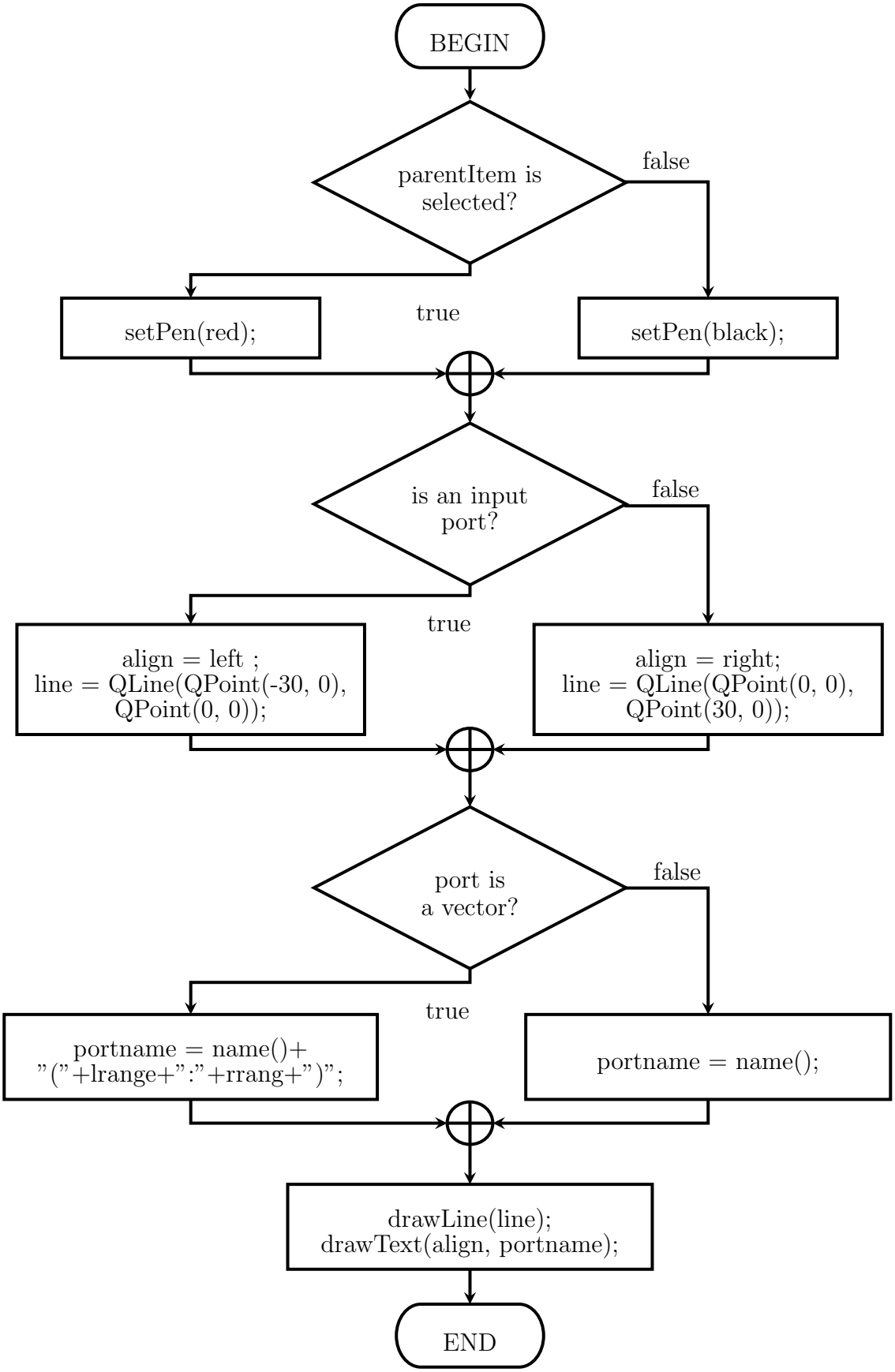


Figure 4.15: Drawing SDR Blocks: a block diagram illustrating the procedure implemented in the *paint* function of the Port class.

Chapter 5

Case Study: Waveform Generator

5.1 Overview

This case study comprises a parameterizable waveform generator. Figure 5.1 shows a block diagram illustrating the architecture of the waveform generator. The system consists of a phase accumulator cascaded with a quantize phase register, a SIN/COS Look Up Table (LUT), a SQUARE/SAWTOOTH wave generator and a wave multiplexer.

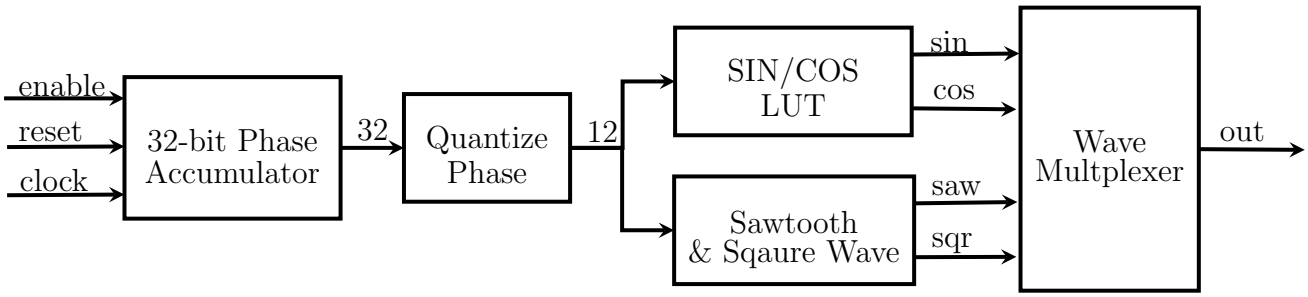


Figure 5.1: Waveform Generator: block diagram illustrating the design of a waveform generator. Adapted from [87].

The waveform generator will produce different types of waveforms, as well as waveforms at varying frequencies based on tunable parameters. Table 5.1 lists the tunable parameters as well as valid parameter options. The *phase_inc* parameter, is known as the phase increment and, is used to control the frequency of the 12-bit output waveform. The waveform type is determined by the *wave* parameter.

The available output waveform types include: sine, cosine, square and sawtooth signals. The generated signal is accessed from the *output* port. Table 5.2 lists the ports which interface control and data signal.

Table 5.1: Waveform Generator Parameters: lists the parameters of the Waveform Generator component.

Generic	Type	Description
phase_inc	Integer	Phase increment determines the output frequency through $Fout = \frac{Phase\_inc * Fclk}{2^{32}}$
wave	String	Type of Waveform to generate. Options include: "SINE", "COSINE", "SQUARE", "SAW"

Table 5.2: Waveform Generator Ports: lists the ports of the Waveform Generator component.

Port	Type	I/O	Description
clk	Logic	in	System clock
reset	Logic	in	Asyncronous system reset
en	Logic	in	Clock enable
output	12-bit vector	out	Waveform output signals

5.2 Components

Figure 5.3 shows a block diagram illustrating the architecture of the waveform generator. The system consists of a phase accumulator cascaded with a quantize phase register, a SIN/COS Look Up Table (LUT), a SQUARE/SAWTOOTH wave generator and a wave multiplexer.

5.2.1 Phase Accumulator

The phase accumulator increments, every clock cycle, by the *phase_inc* parameter value. The frequency of the generated output waveform is determined by the value of *phase_inc* through the formula: $Fout = \frac{Phase\_inc * Fclk}{2^{32}}$. Listing 5.1 illustrates the VHDL implementation of the phase accumulation as well as asynchronous reset and clock enable operations.

Listing 5.1: Phase Accumulator: VHDL code illustrating the phase accumulator functionality.

```
-------------------------------------------------------------------------------
-- Phase accumulator increments by 'phase_inc' every clock cycle --
-- Output frequency determined by formula: Phase_inc = (Fout/Fclk)*2^32 --
-- E.g. Fout = 36MHz, Fclk = 100MHz, Phase_inc = 36*2^32/100 --
-- Frequency resolution is 100MHz/2^32 = 0.00233Hz --
-------------------------------------------------------------------------------

    phase_acc_reg: process(clk, reset)
    begin
        if reset = '0' then
            phase_acc <= (others => '0');
        elsif clk'event and clk = '1' then
            if en = '1' then
```

```
        --phase_acc <= unsigned(phase_acc) + unsigned(phase_inc);
                phase_acc <= unsigned(phase_acc) + phase_inc;
            end if;
        end if;
    end process phase_acc_reg;
```

5.2.2 Quantize Phase

The phase quantization operation is achieved by tapping off the top 12 most significant bits of the accumulated phase. The tapping operation is achieved by:
lut_addr <= **phase_acc**(31 downto 20); in VHDL. This quantized phase is then used to address the SIN/COS LUT as well as in the generation of the square and sawtooth waveforms.

5.2.3 SIN/COS LUT

The SIN/COS LUT consists of two Read-Only Memorys (ROMs) which store preprocessed sine and cosine values. Each ROM holds 4096 12-bits amplitude values. The amplitude values range between -2047 and 2047. In this configuration, a phase resolution of, $2Pi/4096 = 0.088$, degrees can be achieved. Listing 5.2 represents the VHDL entity declaration for the SIN/COS LUT illustrating the input and output ports of the entity.

Listing 5.2: SIN/COS Look Up Table (LUT): VHDL code illustration the instantiation of the LUT.

```
entity sincos_lut is

port (

   clk : in std_logic;
   en : in std_logic;
   addr : in std_logic_vector(11 downto 0);
   sin_out : out std_logic_vector(11 downto 0);
   cos_out : out std_logic_vector(11 downto 0)

   );

end entity;
```

5.2.4 Square & Sawtooth Wave Generation

The square and sawtooth waveforms can be derived directly from the quantized accumulated phase. A square wave can be generated by placing a threshold on the phase value. This thresholding is achieved by:
squ_out <= ‘‘011111111111’’ when **lut_addr_reg**(11) = ‘1’ else ‘‘100000000000’’; in VHDL. If the most significant bit of the phase is high then the output should be high. Similarly if the most significant bit of the phase is low then the output should be low.

A sawtooth wave can be generated by incrementing the signal every time the phase

is incremented. This can by achieved by:
saw_out <= **lut_addr_reg**; in VHDL. Due to the fact that the quantized accumulated phase is a 12-bit standard logic vector, the phase will reset back to zero periodically as the accumulated value exceeds 12-bits. This is actually the desired effect for the sawtooth wave, so the generated sawtooth can be equated to the accumulating phase value.

5.2.5 Output Multiplexer

The type of waveform produced at the output of the waveform generator is determined by the *wave* parameter. Listing 5.3 illustrates the VHDL implementation of the output multiplexer. If the *wave* parameter is set to "$SINE''$ then the generated sine output signal is placed on the *output* port. Similarly, if the *wave* parameter is set to "$COSINE''$, "$SQUARE''$ or "SAW'' then the generated cosine, square or sawtooth signal is place on the *output* port respectively.

Listing 5.3: Multiplexer Description: VHDL code illustrating the implementation of the waveform multiplexer.

```
    latch_output: process(clk)
    begin
        if clk'event and clk = '1' then
            if en = '1' then
                if wave = "SINE" then
                    output <= sin_out;
                elsif wave = "COSINE" then
                    output <= cos_out;
                elsif wave = "SQAURE" then
                    output <= squ_out;
                elsif wave = "SAW" then
                    output <= saw_out;
                end if;
            end if;
        end if;
    end process latch_output;

end rtl;
```

5.3 Results

The results of implementing the waveform generator module in the ROME environment are presented in this section.

5.3.1 Importing The Waveform Generator Module

The VHDL code represented in Listing 9.3 implements the waveform generator. The VHDL implementation was imported into ROME using Xilinx's Create/Import Peripheral Wizard (createip) as mentioned in section 4.3.2. Figure 5.2 is a summary provided by createip apon successful completion of the import process. It is mentioned in the summary that a new pcore was generated from the two source file namely, sincos_lut.vhd and waveform_gen.vhd. It is then concluded in the summary that a new MPD file was

created in the pcore's data directory. The MPD file generated by the createip wizard is provided in Listing 9.4.

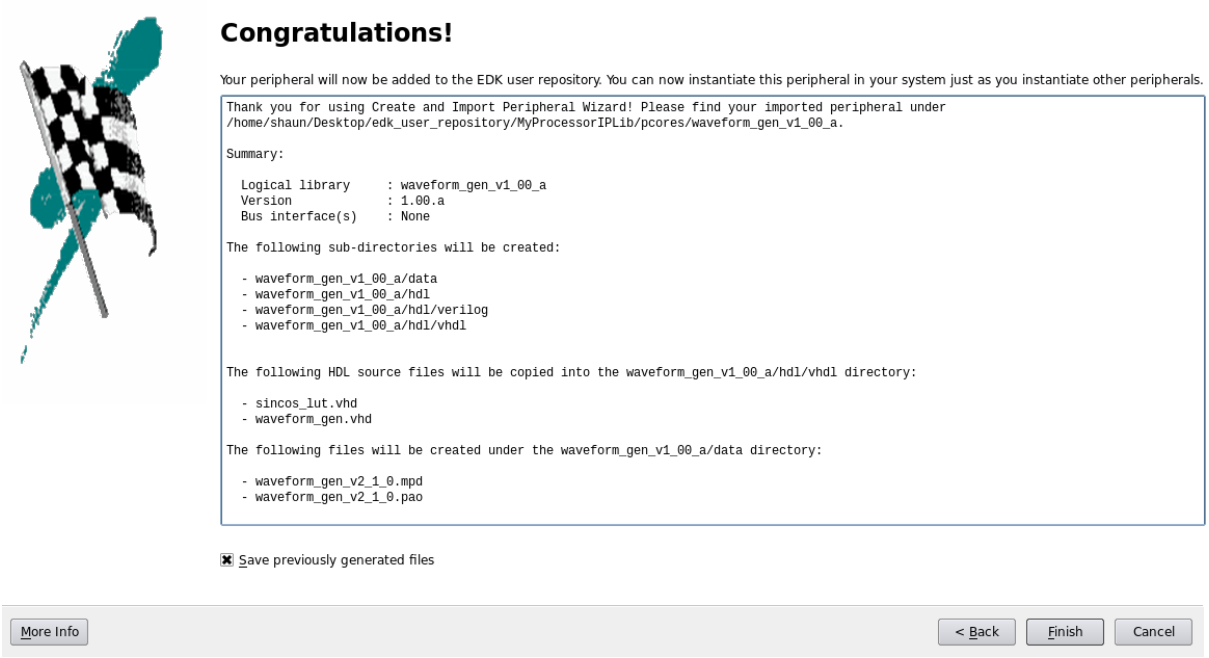


Figure 5.2: Waveform Generator Import Summary: a screenshot illustrating the summary produced by createip wizard after importing the waveform generator.

5.3.2 Building The Waveform Generator Application

Figure 5.3 illustrates the waveform generator application built in the ROME environment. For the purpose of testing the functionality of the waveform generator module within the ROME environment, the module was connected to simulation blocks namely: a simulated clock and reset as well as a block that sinks all simulated output to a file. These blocks are labeled sim_clk, sim_rst and sink_file. Since VHDL generics and Verilog parameters are exposed to the user through the GUI, the destination file and data widths can be adjusted graphically. Figure 5.3 also illustrates the property menu for the waveform generator module.

Listing 9.5 illustrates the design file generated by ROME upon saving the waveform generator application. Together with the XML design file, ROME created the MHS file presented in Listing 9.6. Listing 5.4 presents a portion of the generated MHS file. Note that changes to parameter value through the GUI are represented in the generated MHS file.

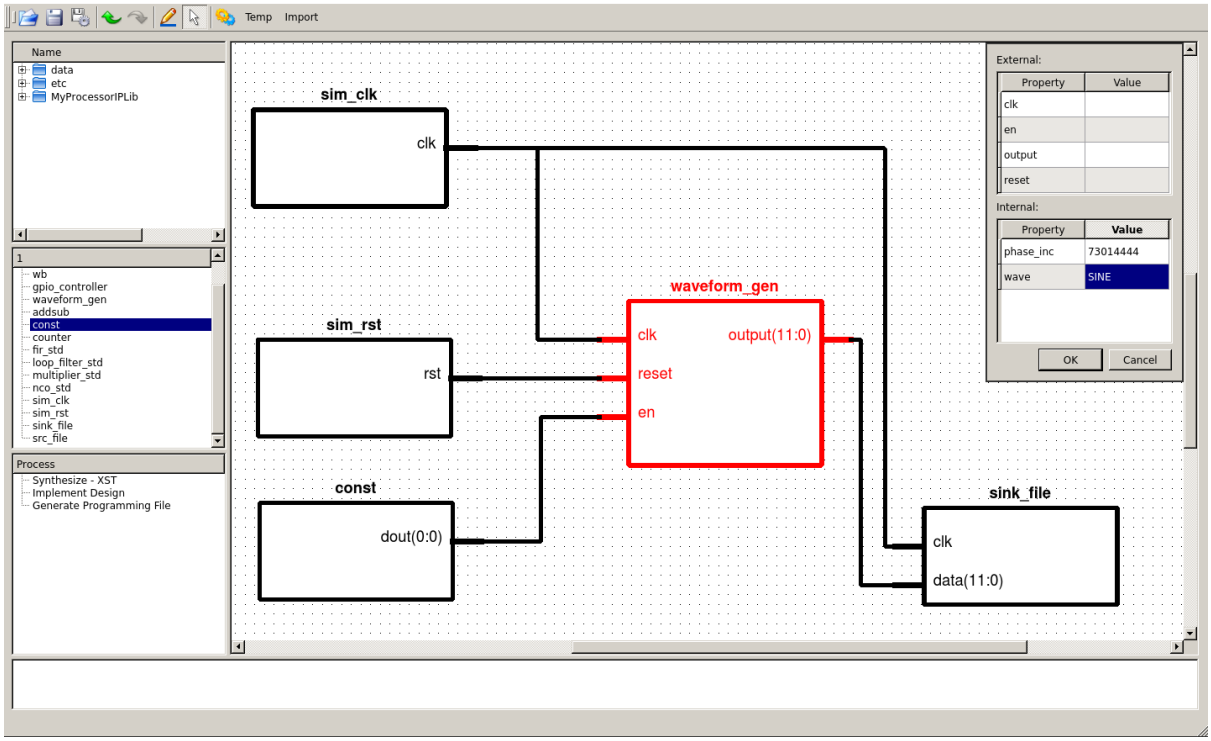


Figure 5.3: Waveform Generator in ROME: a screenshot illustrating the waveform generator example implemented in ROME.

Listing 5.4: Generated MHS code illustrating parameter value propagation.

```
##############################################################################
BEGIN waveform_gen
        PARAMETER INSTANCE = waveform_genC3
        PARAMETER HW_VER = 1.00.a
        PARAMETER phase_inc = 73014444
        PARAMETER wave = SINE
        PORT clk = W3
        PORT reset = W2
        PORT en = W6
        PORT output = W4
END
```

5.3.3 VHDL Code Generation

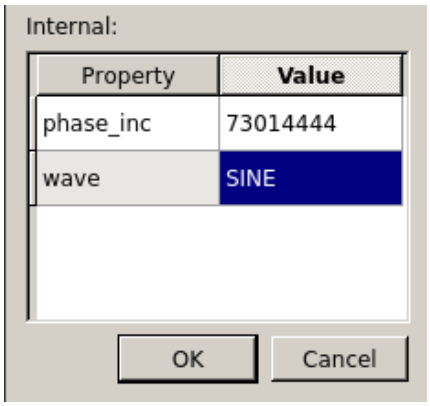
Listing 9.7 presents the logged output as a result of calling xps_gen from with in the ROME environment. According to this output log: a top level and a top level stub VHDL file as well as VHDL wrappers, for each instance, were generated successfully.

5.3.4 Simulation

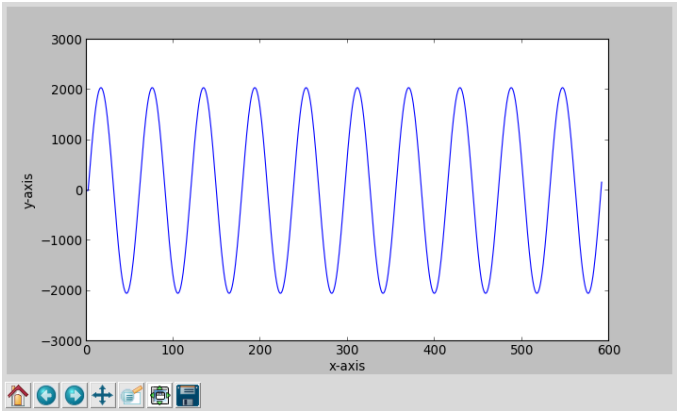
The generated VHDL files were simulated in Xilinx's iSim with a 100MHz clock for a period of 6μs. The resulting waveform was written to a file and the file's contents represented using Matplotlib's pyplot. Figure 5.4 illustrates the effect, on the generated output signal, of changing the value of the *wave* parameter. All waveforms presented in

Figure 5.4 oscillate at 1.7MHz. Figures 5.4b, 5.4d, 5.4f and 5.4h represent the resulting waveforms due the *wave* parameter being changed to "SINE", "COSINE", "SQUARE" and "SAW" respectively.

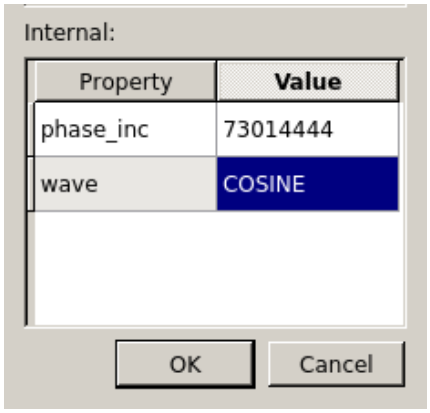
Similarly Figure 5.5 illustrates the effect, on the generated output signal, of changing the value of the *phase_inc* parameter. Both waveforms presented in Figure 5.5 are sine waves. Figures 5.5b and 5.5d represent the resulting sine waves due a change in the *phase_inc* parameter. Figures 5.5b represents a sine wave at twice the default frequency (or 3.4MHz) while Figures 5.5d represents a sine wave at half the default frequency (or 850 KHz). Note that ROME handles the evaluation of the multiplication and division expressions, for the phase_inc parameter, in Figure 5.5.



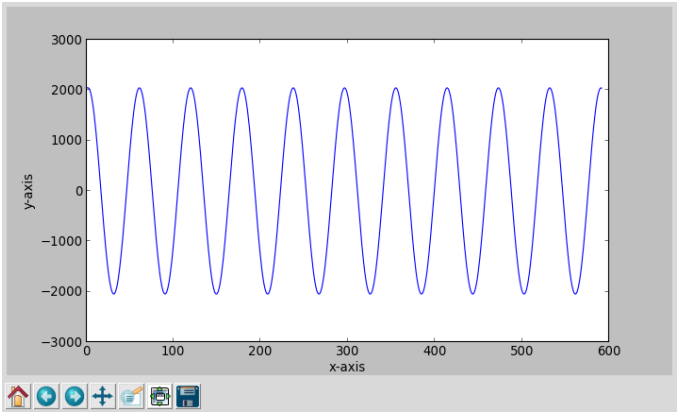
(a) Wave Parameter = SINE



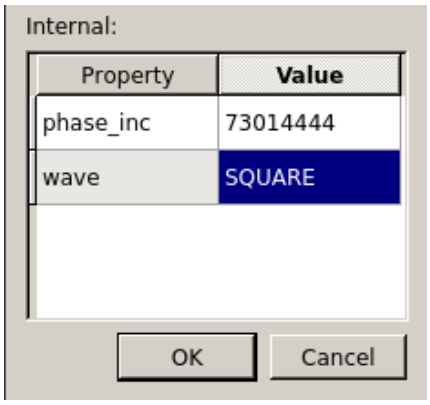
(b) 1.7 MHz Sine Wave



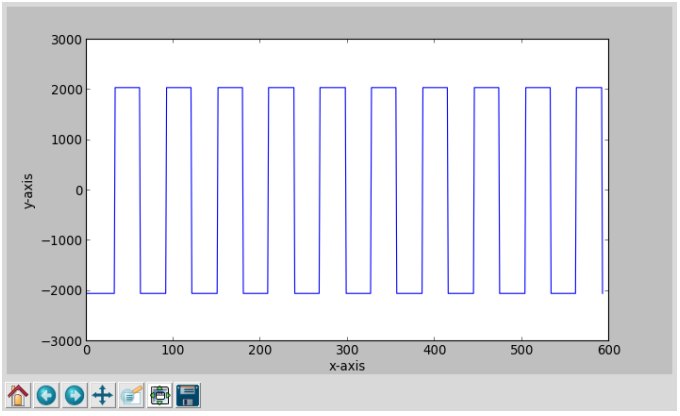
(c) Wave Parameter = COSINE



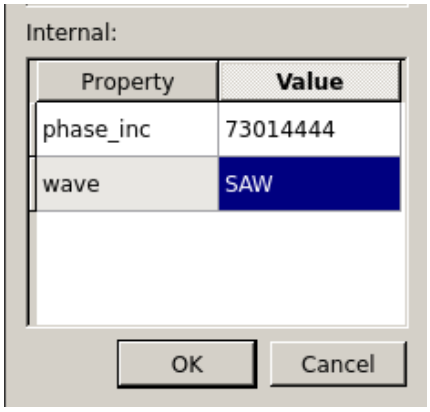
(d) 1.7 MHz Cosine Wave



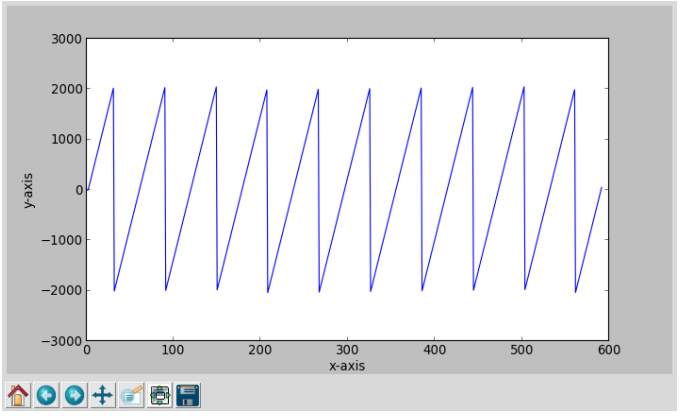
(e) Wave Parameter = SQUARE



(f) 1.7 MHz Square Wave

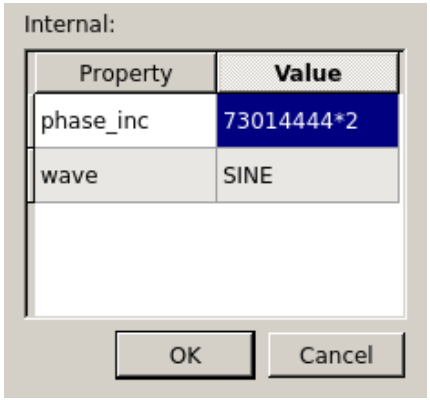


(g) Wave Parameter = SAW

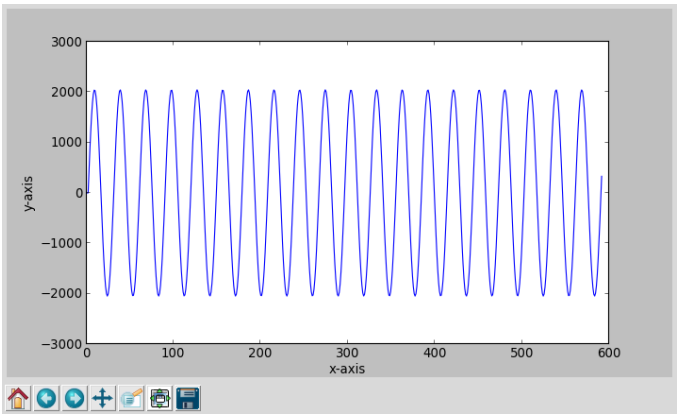


(h) 1.7 MHz Sawtooth Wave

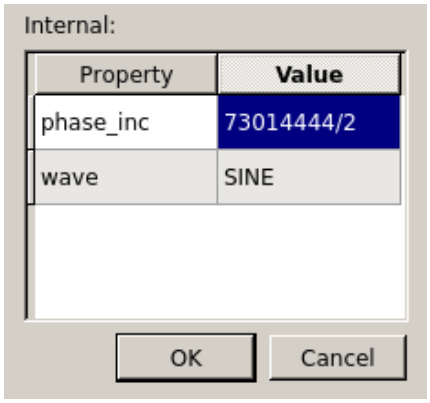
Figure 5.4: Wave Parameter: illustrating the effect of changing the value of the wave parameter.



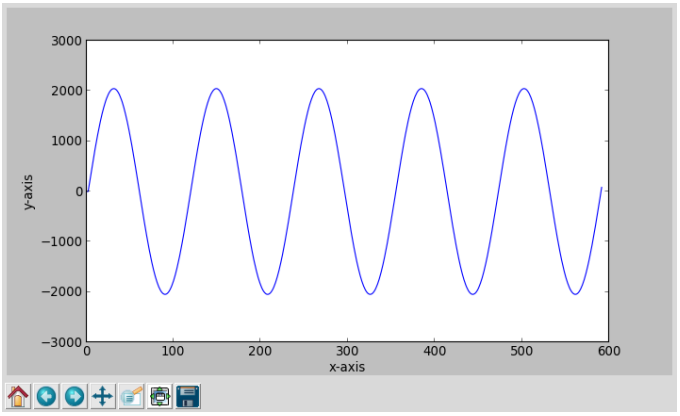
(a) Phase_inc = 73014444*2



(b) 3.4 MHz Sine Wave



(c) Phase_inc = 73014444/2



(d) 850 KHz Sine Wave

Figure 5.5: Phase Increment Parameter: illustrating the effect of changing the value of the phase_inc parameter.

Chapter 6

Case Study: Wishbone Adder

6.1 Overview

This study comprises a software accessible adder. Figure 6.1 presents a block diagram illustrating the design of a software accessible adder for implementation on the RHINO platform. The system consists of: a wishbone bus to handle inter-module communication; wishbone compliant registers to store values; an adder to compute additions and a bridge to handle communication between the ARM processor's GPMC interface, on the RHINO, and the wishbone bus.

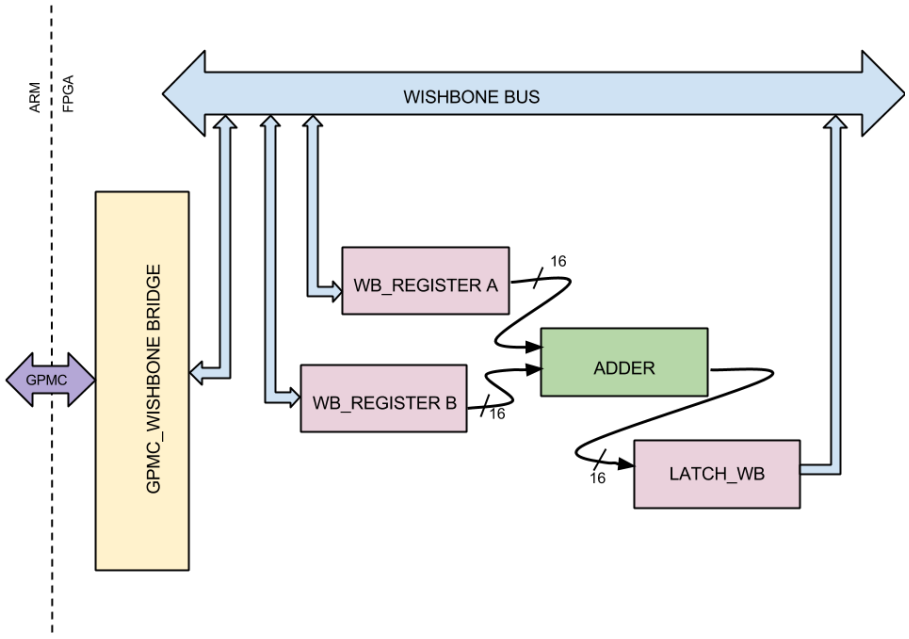


Figure 6.1: Wishbone Adder: block diagram illustrating the design of a wishbone adder.

The design for this study was inspired by a tutorial for CASPER's Simulink-based Toolflow [97]. A screenshot of the adder implemented in CASPER's tutorial is presented

in Figure 6.2.

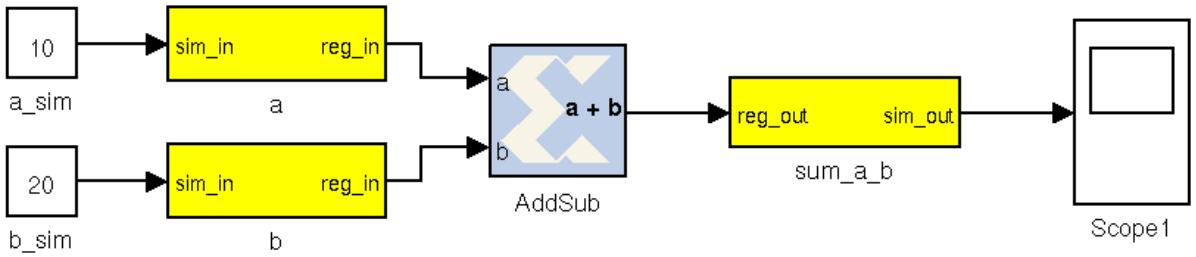


Figure 6.2: CASPER Adder: a screenshot of an implementation of an adder using CASPER's Simulink-based Toolflow. Adapted from [97].

6.2 Components

The system consists of: a wishbone bus to handle inter-module communication; wishbone compliant registers to store values; an adder to compute additions and a bridge to handle communication between the ARM processor's GPMC interface, on the RHINO, and the wishbone bus.

6.2.1 Wishbone Bus

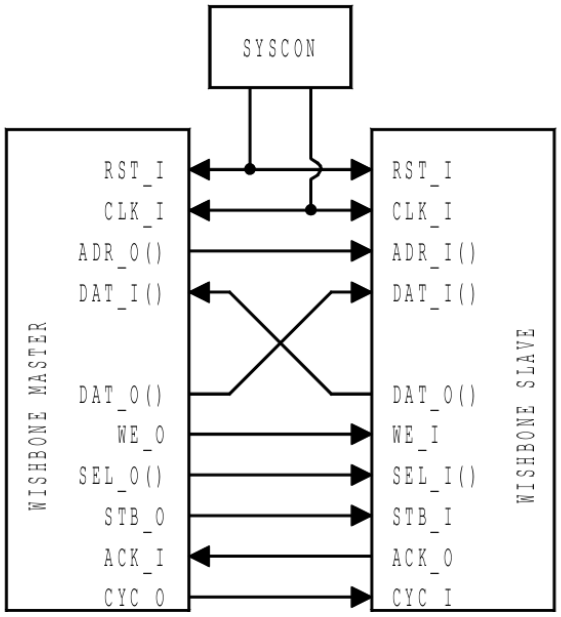
Wishbone is a flexible public domain design methodology for use in system-on-chip (soc) applications. Wishbone tries to establish a standard for inter-module communication and data exchange. The standard aims to encourage design reuse, portability and reliability of designs.

Figure 6.3a presents a block diagram illustrating the interconnection between a wishbone master and a wishbone compliant slave module. Signal descriptions about the various signals used in the wishbone interconnect are available in the wishbone standard [98]. A portion of the VHDL implementation of the wishbone bus interconnection is presented in Figure 6.3b. The full VHDL implementation of the wishbone bus is presented in Listing 9.8. In Figure 6.3b all master signals are prefixed with a $M\_$ while all slave signals are prefixed with a $S\_$.

Figures 6.5b and 6.6b present clock signal diagrams illustrating the signalling on the wishbone bus interconnection interface during read and write operations respectively.

6.2.2 GPMC Wishbone Bridge

The GPMC is a 16-bit external memory controller implemented by the ARM processor on the RHINO platform. The GPMC data access engine provides a flexible model for



(a) WB InterCon Diagram. Adapted from [98].

```
architecture BEHAVIORAL of wb is
begin
    S_WB_ADR <= M_WB_ADR;
    S_WB_DAT_I <= M_WB_DAT_O;
    S_WB_WE <= M_WB_WE;
    S_WB_STB <= M_WB_STB;
    S_WB_CYC <= M_WB_CYC;

    M_WB_DAT_I <= S_WB_DAT_O;
    M_WB_ACK <= S_WB_ACK;

end BEHAVIORAL;
```

(b) WB InterCon VHDL

Figure 6.3: Wishbone Interconnection: (a) block diagram and (b) VHDL code illustrating the interconnection between master and slaves on a wishbone bus.

communication with all standard memory types [19]. On the RHINO platform, the processor's GPMC interface is used as a data bus between the AM3517 processor and the Spartan6 FPGA.

The GPMC's 1GB address space is memory mapped onto the processor's address space [19]. In this way, applications can access external devices by simply reading from or writing to appropriate memory addresses. The processor then handles the bus transactions in the background, completely invisibly to the user application.

The GPMC Wishbone Bridge was developed in order to allow a user application to access internal devices implemented on the FPGA. This enables the processor to act as a master on the wishbone bus. In this way, all slaves connected to the wishbone bus are addressable from within the processor's memory map.

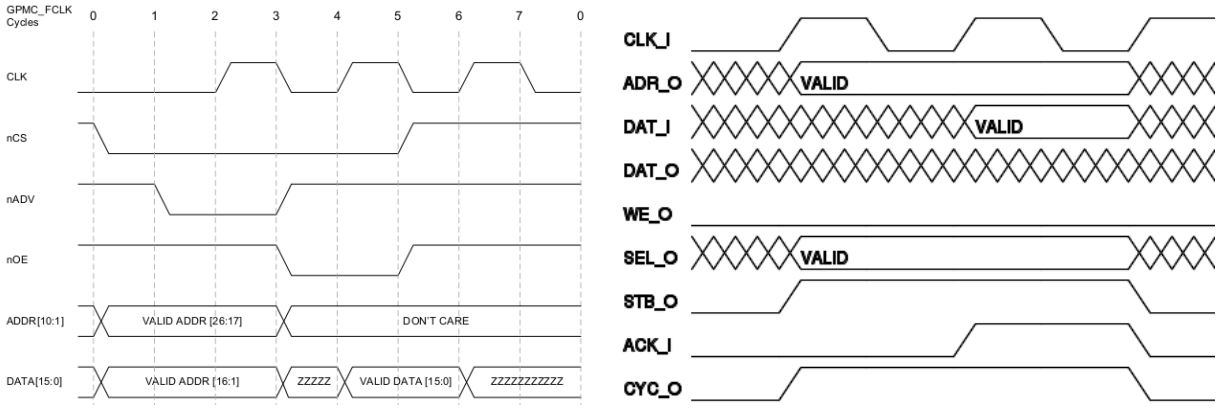
Figure 6.4 provides a description of the various signals present in the GPMC interface as well as their relevance to the FPGA. On the RHINO platform, the chip select lines CS1 to CS7 are connected to the FPGA [19], while CS0 is connected to a NAND flash present on the RHINO and thus can be ignored.

Figures 6.5a and 6.6a present clock signal diagrams illustrating the signalling on the GPMC interface during read and write operations respectively. The Bridge translates

GPMC Control Line	Full name	Direction	Description
nCS0 to nCS7	Chip select 0 to 7	Output	Enables a specific external memory device.
CLK	Clock	Output	Bus clock
nWE	Write Enable	Output	Indicates valid data on bus during write
nOE	Output Enable	Output	Indicates peripheral can write to data bus.
nADV_ALE	Address Valid	Output	Indicates valid address present on bus.
CLE	Control Latch Enable	Output	Not used by FPGA.
nWP	Write Protect	Output	Enables write protect on memory device.
WAIT0 to WAIT2	Wait inputs	Input	Indicates that memory device still busy.

Figure 6.4: GPMC Control Lines: Describes the control lines of the GPMC interface. Adapted from [19].

GPMC signals to wishbone signals and visa versa.



(a) GPMC Read Cycle [19]

(b) Wishbone Read Cycle [98]

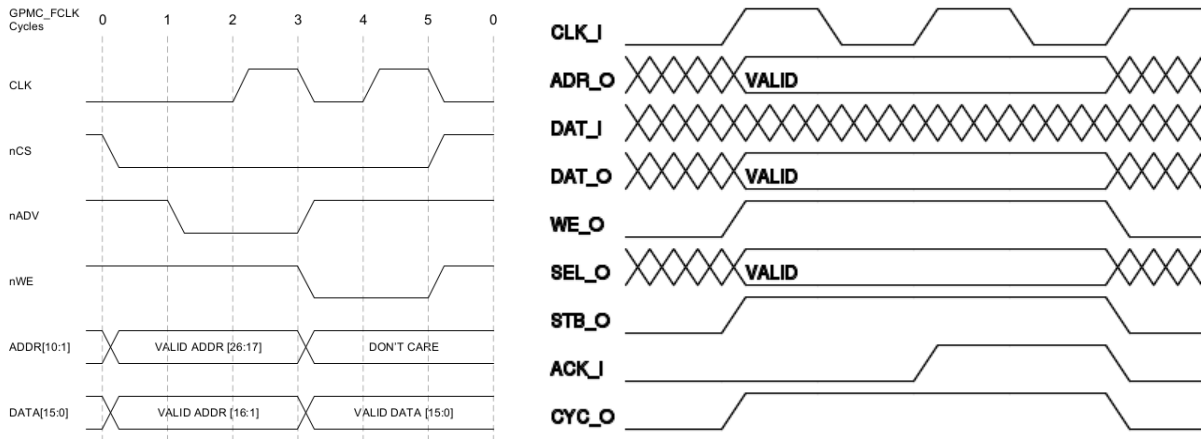
Figure 6.5: GPMC/Wishbone Read Cycle: signal diagrams illustrating the signal timing required for a (a) GPMC and (b) Wishbone read operations.

A full VHDL implementation of the GPMC Wishbone Bridge is presented in Listing 9.10. The VHDL port declaration for the GPMC Wishbone Bridge entity is presented in Listing 6.1.

Listing 6.1: GPMC Wishbone Bridge Entity Declaration

```
-- GPMC INTERFACE
gpmc_a : in std_logic_vector(10 downto 1);
gpmc_d : inout std_logic_vector(15 downto 0);
gpmc_clk_i : in std_logic;
gpmc_n_cs : in std_logic_vector(6 downto 0);
gpmc_n_we : in std_logic;
gpmc_n_oe : in std_logic;
gpmc_n_adv_ale : in std_logic;
gpmc_n_wp : in std_logic;
gpmc_busy_0 : out std_logic;
gpmc_busy_1 : out std_logic;

-- WISHBONE MASTER INTERFACE
RST_I,CLK_I : in std_logic;
ADR_O : out std_logic_vector (15 downto 0);
```



(a) GPMC Write Cycle [19]

(b) Wishbone Write Cycle [98]

Figure 6.6: GPMC/Wishbone Write Cycle: signal diagrams illustrating the signal timing required for a (a) GPMC and (b) Wishbone write operations.

```
DAT_O : out std_logic_vector (15 downto 0);
WE_O,STB_O,CYC_O : out std_logic;
DAT_I : in std_logic_vector (15 downto 0);
ACK_I : in std_logic
```

6.2.3 Wishbone Register

The Wishbone Register module stores values set by the user from the processor. The module is fully wishbone compliant and acts as a slave on the wishbone bus. Each register on the bus is individually and uniquely accessible due to a parameterizable address. When connected to the RHINO's processor through the wishbone bus and GPMC Wishbone bridge, the module is addressable from the processor's memory map. Figure 6.7 presents a schematic diagram of a 16-bit wishbone compliant register with a non-wishbone output port. The parameterizable address is not depicted in the Figure 6.7. The state of the output port can be monitored by the processor (or any wishbone master) by routing the output data lines back to [DAT_O(16..0)]. During read cycles the AND gate prevents erroneous data from being latched into the register.
Listing 9.12 provides a full VHDL implementation of the Wishbone Register. The code was inspired by the 16-bit SLAVE output port presented in [98], however modified to include parameterizable addressing.

6.2.4 Adder/Subtracter

The Adder/Subtracter is a two input module with parameterized data width. Table 6.1 lists and describes the ports of the module. The design unit multiplexes add and subtract operations with an addnsub input. The full VHDL implementation of the adder/subtracter module is presented in Listing 9.14. The code in Listing 9.14 is an adaptation of

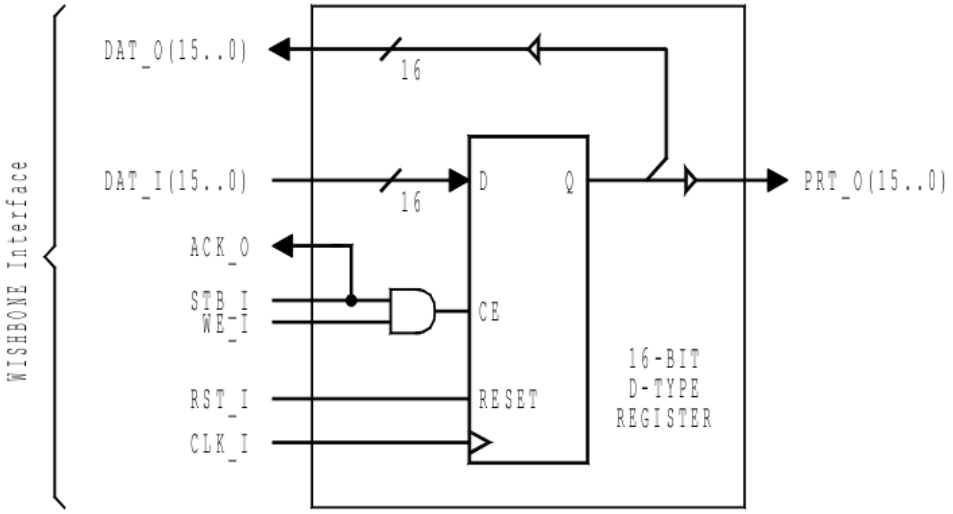


Figure 6.7: 16-bit Wishbone Register: a circuit diagram illustrating the design of a 16-bit wishbone compliant register. Adapted from [98].

the adder/subtracter module presented in [99].

Table 6.1: Adder/Subtracter: lists the ports of the adder/subtracter component.

Port	I/O	Description
a, b	Input	Data inputs to adder/subtracter with parameterized width
addnsub	Input	Multiplexing input for add and subtract operations
result	Output	The output addition/subtraction, of the two inputs, with parameterized width

6.3 Results

The results of implementing the wishbone adder design in the ROME environment are presented in this section.

6.3.1 Importing IP Core

Due to the fact that the wishbone busing standard is not one of the "Xilinx-known" bus standards [100], a user-defined wishbone bus was created. The process of defining a new bus standard in XPS is mentioned in section 2.8.3. This was achieved by editing the MPD files, generated by the createip wizard for all cores with a wishbone interface.

Listing 6.2 presents a portion of the MPD file, generated by the createip wizard for the Wishbone Bus core. A full version of this file is presented in Listing 9.9. The MPD file was modified to include bus abstraction functionality. Lines 12 and 13 were added to the file to define the module as a bus and to define the standard of the bus to be wishbone. The master and slave interfaces of the wishbone bus were then defined by modifying lines 30 to 46.

Listing 6.2: Wishbone Bus MPD file

```
11 ## Peripheral Options
12 OPTION IPTYPE = BUS
13 OPTION BUS_STD = WB
```

```
30 ### WISHBONE MASTER INTERFACE
31 PORT M_WB_ADR = M_WB_ADR, DIR = I, VEC = [15:0]
32 PORT M_WB_DAT_O = M_WB_DAT_O, DIR = I, VEC = [15:0]
33 PORT M_WB_WE = M_WB_WE, DIR = I
34 PORT M_WB_STB = M_WB_STB, DIR = I
35 PORT M_WB_CYC = M_WB_CYC, DIR = I
36 PORT M_WB_DAT_I = M_WB_DAT_I, DIR = O, VEC = [15:0]
37 PORT M_WB_ACK = M_WB_ACK, DIR = O
38
39 ### WISHBONE SLAVE INTERFACE
40 PORT S_WB_ADR = S_WB_ADR, DIR = O, VEC = [15:0]
41 PORT S_WB_DAT_I = S_WB_DAT_I, DIR = O, VEC = [15:0]
42 PORT S_WB_WE = S_WB_WE, DIR = O
43 PORT S_WB_STB = S_WB_STB, DIR = O
44 PORT S_WB_CYC = S_WB_CYC, DIR = O
45 PORT S_WB_DAT_O = S_WB_DAT_O, DIR = I, VEC = [15:0]
46 PORT S_WB_ACK = S_WB_ACK, DIR = I
```

Before the GPMC Wishbone Bridge could be connected to the master interface of the newly defined wishbone bus, the MPD file for the bridge core was modified. Listing 6.3 presents a portion of the MPD file, generated by the createip wizard for the GPMC Wishbone Bridge core. A full version of this file is presented in Listing 9.11. Lines 19 was added to the file in order to specify to which bus interface the core connects to. The bus standard and type are specified in line 19. The wishbone master interfaces was then defined by modifying lines 36 to 42.

Listing 6.3: GPMC Wishbone Bridge MPD file

```
18 ## Bus Interfaces
19 BUS_INTERFACE BUS=MWB, BUS_STD=WB, BUS_TYPE=MASTER
```

```
36 PORT ADR_O = "M_WB_ADR", DIR = O, VEC = [15:0], BUS = MWB
37 PORT DAT_O = "M_WB_DAT_O", DIR = O, VEC = [15:0], BUS = MWB
38 PORT WE_O = "M_WB_WE", DIR = O, BUS = MWB
39 PORT STB_O = "M_WB_STB", DIR = O, BUS = MWB
40 PORT CYC_O = "M_WB_CYC", DIR = O, BUS = MWB
41 PORT DAT_I = "M_WB_DAT_I", DIR = I, VEC = [15:0], BUS = MWB
42 PORT ACK_I = "M_WB_ACK", DIR = I, BUS = MWB
```

Similarly, in order for the Wishbone Register to be connected to the slave interface of

the previously defined wishbone bus, the MPD file for the register core was modified. Listing 6.4 presents a portion of the MPD file, generated by the createip wizard for the Wishbone Register core. A full version of this file is presented in Listing 9.13. Lines 19 was added to the file in order to specify to which bus interface the core connects to. The bus standard and type are specified in line 19. The wishbone slave interface was then defined by modifying lines 28 to 34.

Listing 6.4: Wishbone Register MPD file

```
18 ## Bus Interfaces
19 BUS_INTERFACE BUS=SWB, BUS_STD=WB, BUS_TYPE=SLAVE
```

```
28 PORT ADR_I = S_WB_ADR, DIR = I, VEC = [15:0], BUS = SWB
29 PORT DAT_I = S_WB_DAT_I, DIR = I, VEC = [15:0], BUS = SWB
30 PORT WE_I = S_WB_WE, DIR = I, BUS = SWB
31 PORT STB_I = S_WB_STB, DIR = I, BUS = SWB
32 PORT CYC_I = S_WB_CYC, DIR = I, BUS = SWB
33 PORT DAT_O = S_WB_DAT_O, DIR = O, VEC = [15:0], BUS = SWB
34 PORT ACK_O = S_WB_ACK, DIR = O, BUS = SWB
35 PORT PRT_O = "", DIR = O, VEC = [15:0]
```

6.3.2 Abstraction

Figure 6.8 presents a screenshot of the Wishbone Adder design implemented in the ROME environment. Note that all the wishbone interfaces are abstracted away from the top level diagram. For comparative purposes, the Wishbone Adder design was implemented in both the Xilinx ISE and Xilinx XPS design tools. Figures 6.9a and 6.9b present screenshots illustrating the Wishbone Adder design implemented in XPS and ISE respectively.

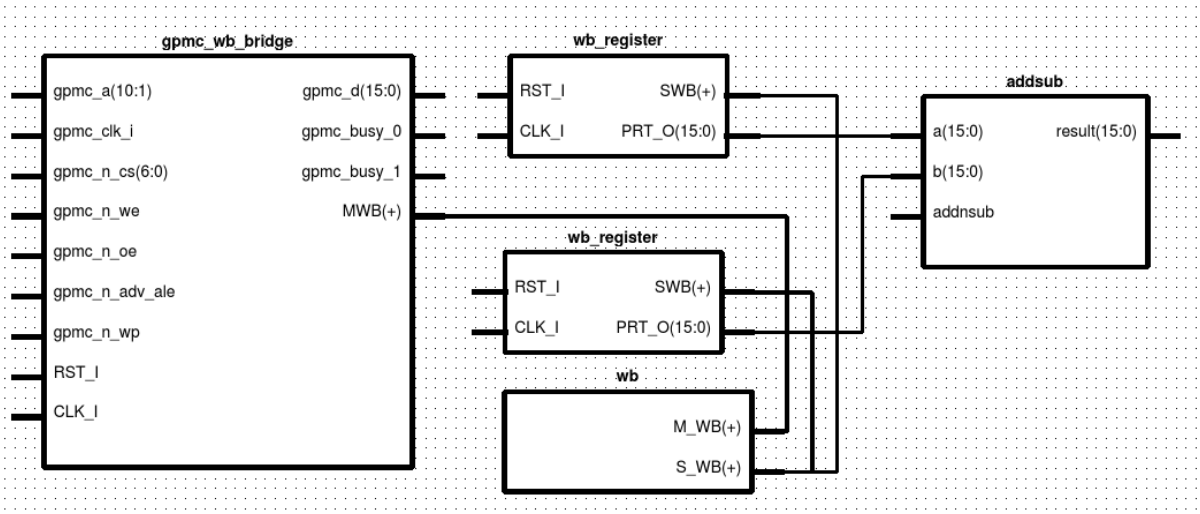
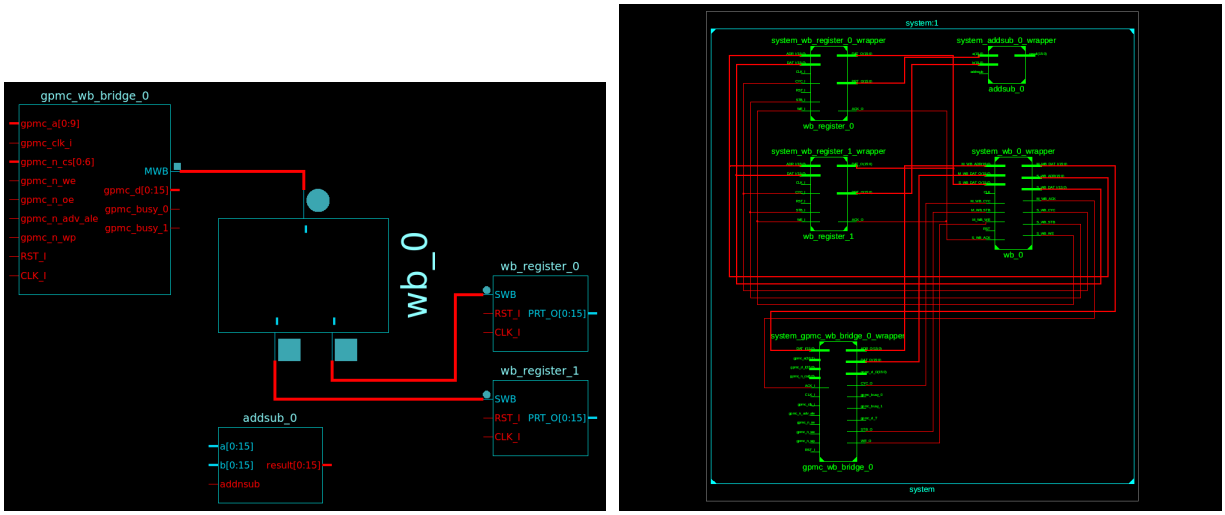


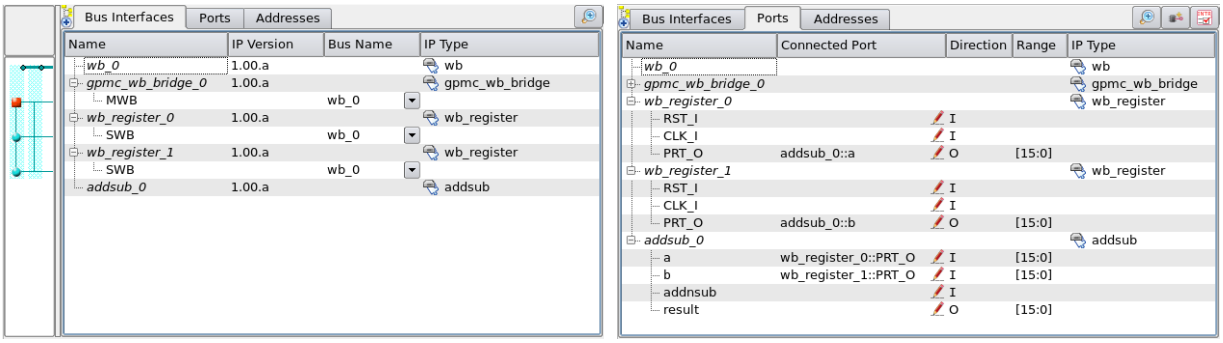
Figure 6.8: Wishbone Adder Implemented in ROME: a screenshot illustrating the wishbone adder example implemented in the ROME environment.



(a) XPS GUI (b) Xilinx ISE

Figure 6.9: Wishbone Adder in Xilinx: screenshots illustrating the wishbone adder examples implemented in (a) XPS GUI and (b) Xilinx ISE.

Figures 6.10a and 6.10b present XPS screenshots of the bus interface and ports views respectively. The bus interface view provides the user with the ability to connect cores to a bus, while the ports view provides the user with a mechanism for inter-port connections.



(a) XPS Bus View. (b) XPS Ports View.

Figure 6.10: Wishbone Adder in XPS: screenshots illustrating the wishbone adder example in XPS's (a) bus and (b) ports views.

6.3.3 VHDL Code Generation

Listing 9.15 presents the saved MHS file for the Wishbone Adder design. A portion of the saved MHS file is presented in Listing 6.5. Note that the wishbone slave interface of the Wishbone Register core instantiated at line 37, is connected to the instantiation of the Wishbone Bus core through the BUS_INTERFACE SWB = wb_0 command at line 41.

Listing 6.5: Wishbone Adder MHS file

```
3  ##############################################################################
4  BEGIN wb
5   PARAMETER INSTANCE = wb_0
6   PARAMETER HW_VER = 1.00.a
7  END
```

```
36 ##############################################################################
37 BEGIN wb_register
38  PARAMETER INSTANCE = wb_register_C2
39  PARAMETER HW_VER = 1.00.a
40  PARAMETER C_BASEADDR = 0x0002
41  BUS_INTERFACE SWB = wb_0
42  PORT PRT_O = W4
43 END
```

A programming bitstream was then successfully generated from with in the ROME environment. Figure 6.11 presents a screenshot of the output log after running the "Generate Programming File" process in the compilation menu.

6.3.4 Verification

The generated bitstream, discussed in section 6.3.3, was then verified on the RHINO. The bitstream was used to generate a BOF executable file which can run on, the RHINO's linux kernal, BORPH. Figure 6.12 presents a screenshot of the BORPH terminal on the RHINO platform. As mentioned in section 2.3.3, the BORPH operating system maps the FPGA registers onto a virtual file system. This provides users with the ability to read and write to registers on the FPGA by reading and writing to these virtual files. In figure 6.12 the Wishbone Registers are written to with the "echo" command and their values read using the "od" command. After the two registers, A and B, had been set to the values 6 and 4 respectively, the output register was read. The last "od" command in figure 6.12 reveals the values of registers A and B in columns 2 and 3 respectively as well as the result of their addition in column 4.

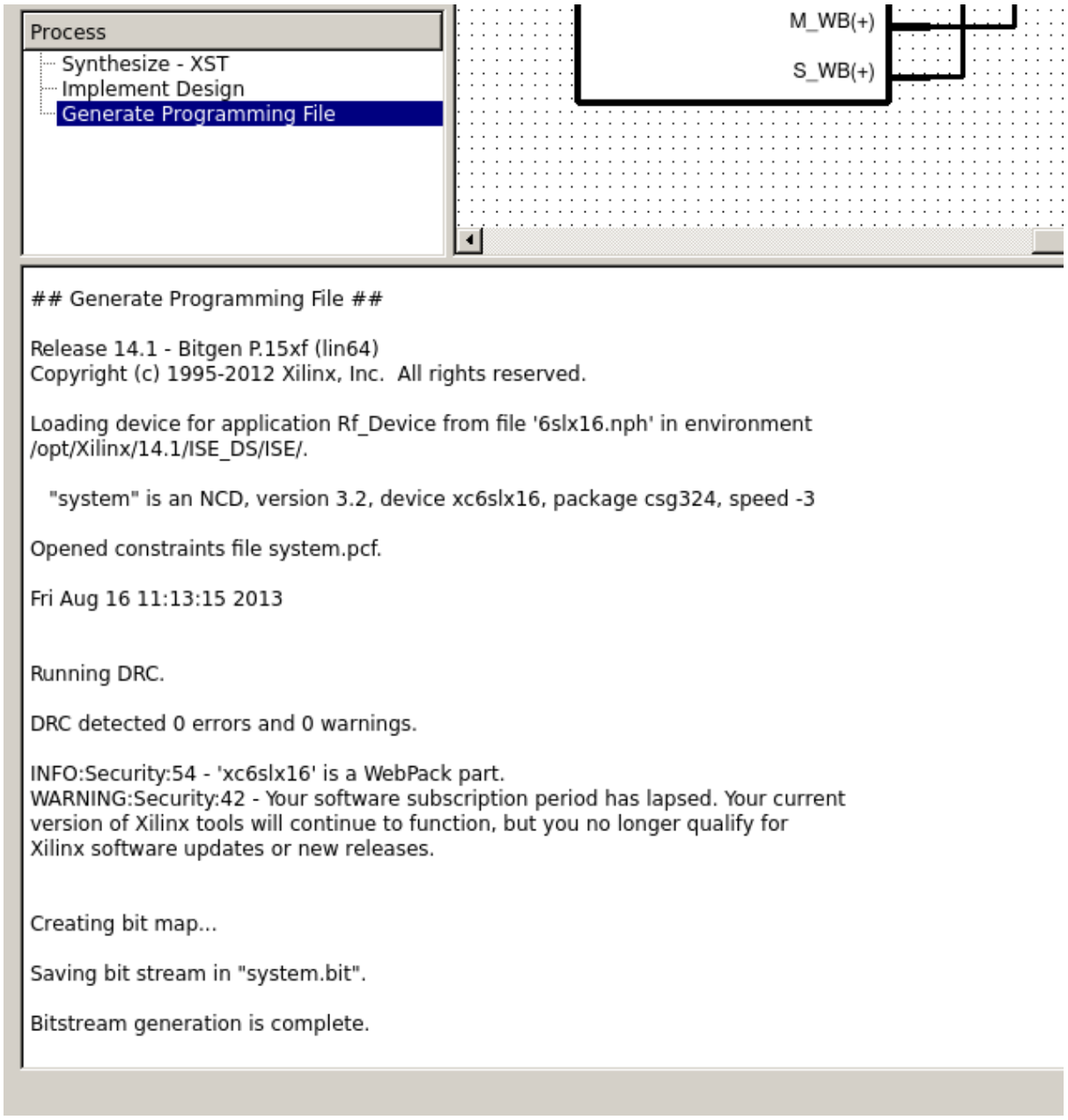


Figure 6.11: Wishbone Adder Bitstream Generation: screenshot illustrating the result of the bitstream generation operation for the wishbone adder example.

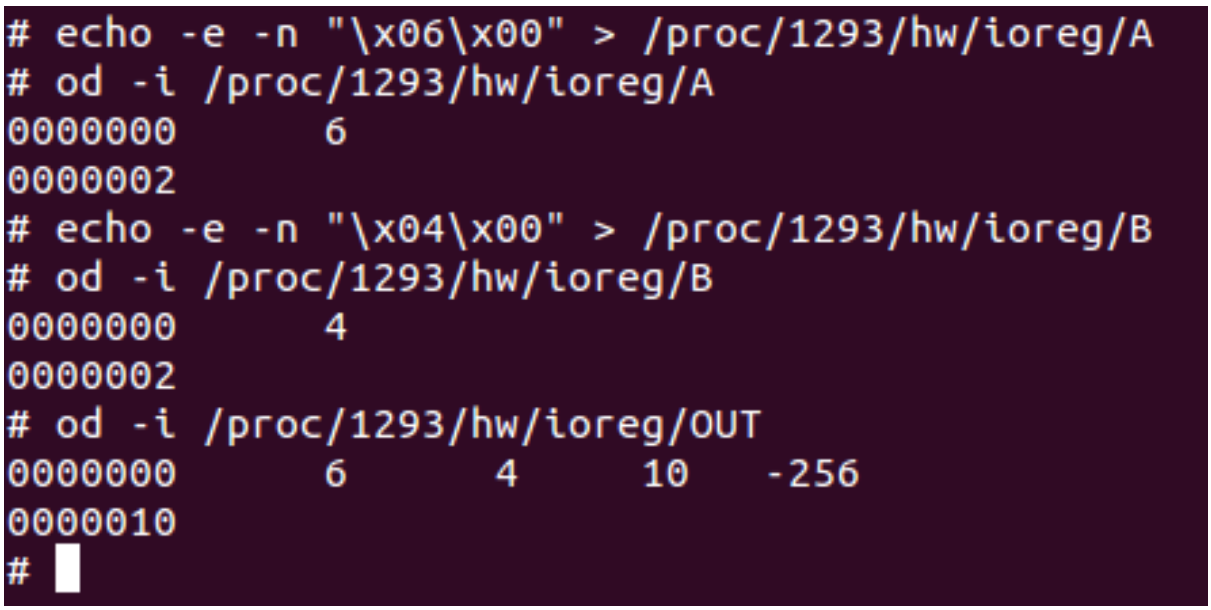


Figure 6.12: Wishbone Adder Verification: a screenshot of the BORPH terminal illustrating the results of write to and read from the registers on the FPGA.

Chapter 7

Case Study: Digital FM Receiver

7.1 Overview

This study comprises a digital frequency modulation (FM) receiver. The design is adapted from [88] and consists of a Numerically Controlled Oscillator (NCO), Multiplier and Loop Filter cascaded with a Finite Impulse Response (FIR) Low pass filter. The reader is encouraged to refer to the work presented in [88] for more detail on the design of this FM receiver. Figure 7.1 presents a block diagram illustrating the design of the Phase Lock Loop (PLL) based FM receiver described in [88].

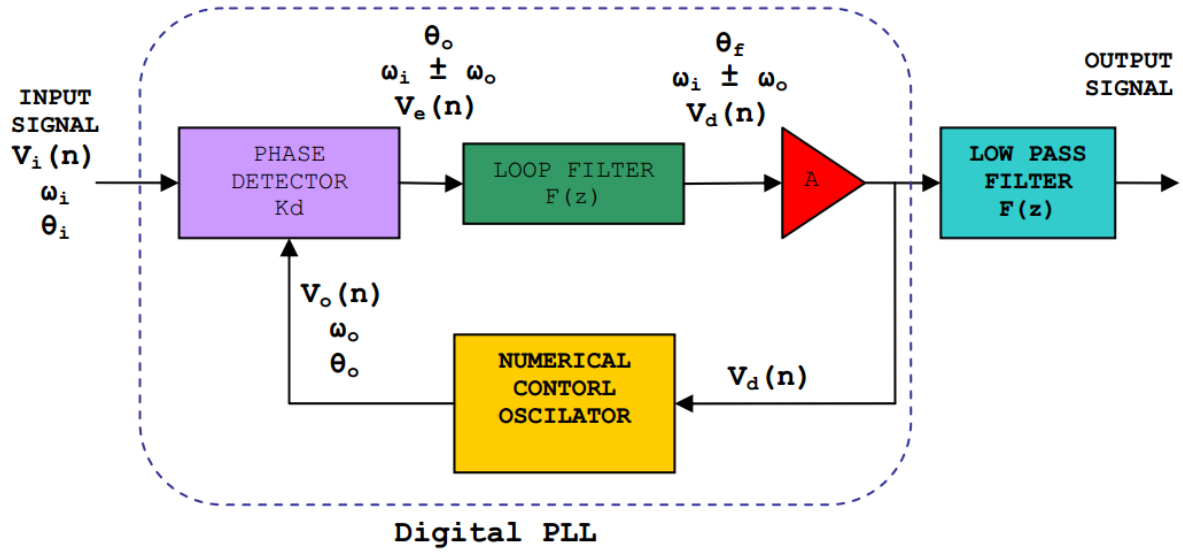


Figure 7.1: All Digital FM Receiver Circuit: block diagram illustrating the design of a PLL-based FM receiver. Adapted from [88].

7.2 Components

7.2.1 Phase Detector

The phase detector component is used to detect the phase error between the input FM modulated signal and the output of the NCO within the PLL. This is achieved by multiplying the two inputs to the phase detector. The multiplication of these inputs produces a 16-bit number, which is then scaled down by discarding the 8 most significant bits. A list of all ports and description for the phase detector component are presented in Table 7.1

Table 7.1: Phase Detector: lists the ports of the phase detector component.

Port	**Type**	**I/O**	**Description**
clk	Logic	in	System clock
reset	Logic	in	Asyncronous system reset
input1	8-bit Vector	in	FM modulated input signal
input2	8-bit Signed	in	Input from NCO
output	8-bit Signed	out	Product of input signals

7.2.2 Loop Filter

The loop filter module is used to remove high frequency components from the circuit. This is implemented in VHDL by performing a sign extension of the 8-bit input signal into a 12-bit output multiplied by 15/16 that is added back to the 8-bit input. A list of all ports and description for the loop filter component are presented in Table 7.2

Table 7.2: Loop Filter: lists the ports of the loop filter component.

Port	**Type**	**I/O**	**Description**
clk	Logic	in	System clock
reset	Logic	in	Asyncronous system reset
C	8-bit Signed	in	Input signal from the Phase Detector
D1	12-bit Signed	out	Output filtered signal
D2	12-bit Signed	out	Output filtered signal (D1 scaled by 1/1024)

7.2.3 Numerically Controlled Oscillator (NCO)

The NCO component is implemented as an integrator circuit that accumulates the input values and maps them to a cosine ROM. A list of all ports and description for the NCO component are presented in Table 7.3

Table 7.3: Numerically Controlled Oscillator (NCO): lists the ports of the NCO component.

Port	**Type**	**I/O**	**Description**
clk	Logic	in	System clock
reset	Logic	in	Asyncronous system reset
din	12-bit Signed	in	Input signal from the Loop Filter
dout	8-bit Signed	out	Output data from Cosine ROM

7.2.4 Digital Low Pass FIR Filter

The low pass filter is used to shape the output signal through a 16 tap FIR filter. This filter is essentially an average filter since the filtered output is equal to the average of the last 16 sample values. The VHDL implementation of this is just a 4-bit logical shift to the right. A list of all ports and description for the FIR component are presented in Table 7.4

Table 7.4: Digital Low Pass FIR Filter: lists the ports of the digital low pass FIR filter.

Port	**Type**	**I/O**	**Description**
clk	Logic	in	System clock
reset	Logic	in	Asyncronous system reset
data_in	12-bit Signed	in	Input signal from the Loop Filter
data_out	12-bit Vector	out	Output FM demodulated signal

7.3 Results

The results of implementing the FM receiver design in the ROME environment are presented in this section.

7.3.1 Importing IP Core

Figure 7.2 illustrates the Frequency Modulation (FM) receiver chain built and implemented in the ROME environment. The blocks labeled sim_clk, sim_rst, src_file and sink_file, are for simulation purposes, supporting the functionality of reading FM data from a text file and writing the demodulated data to another file. Since VHDL generics and Verilog parameters are exposed to the user through the GUI, source and destination files as well as filter coefficients and data widths can be adjusted graphically.

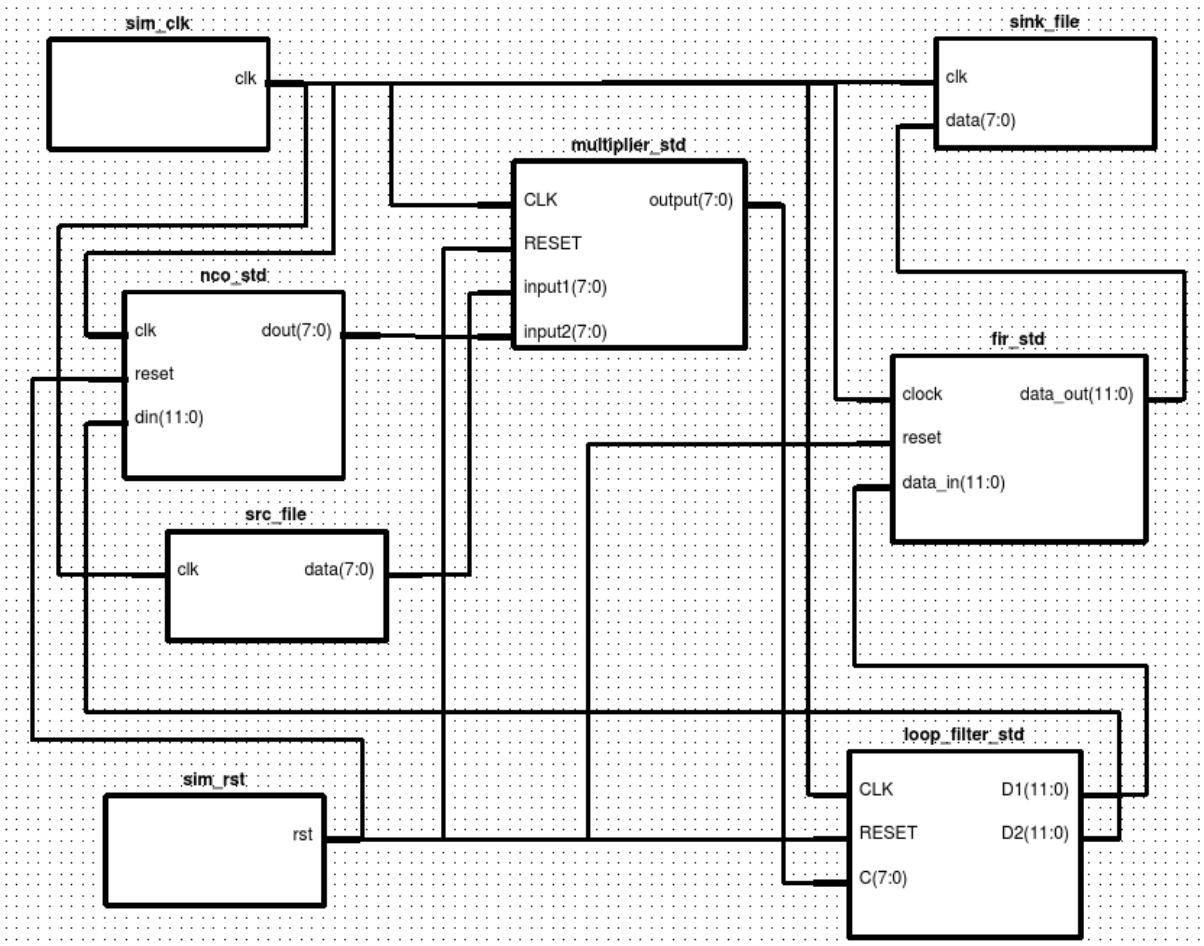


Figure 7.2: FM Receiver in ROME: a screenshot illustrating the FM receiver example implemented in the ROME environment.

7.3.2 Automatic Routing

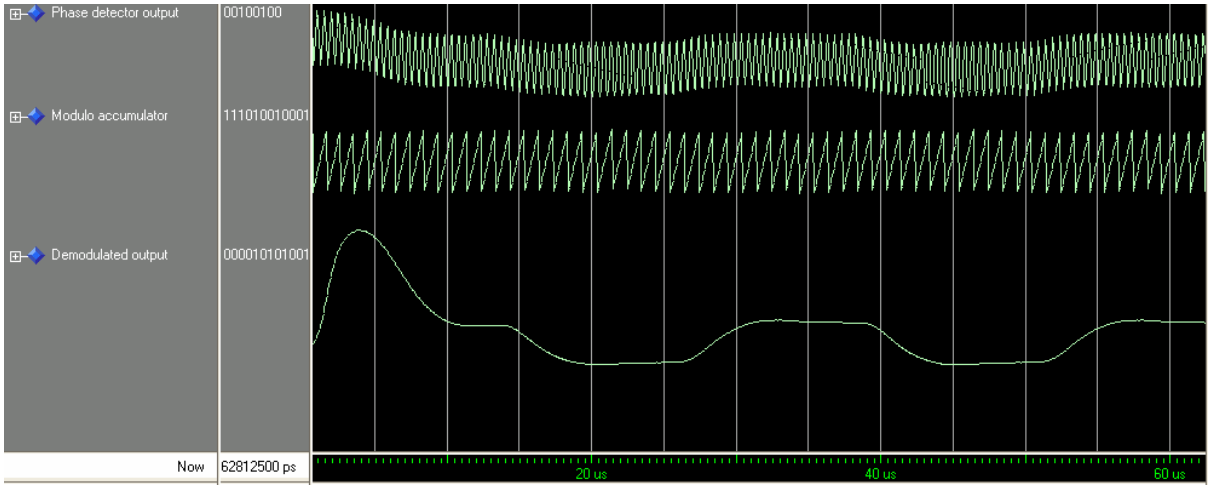
Figure 7.2 demonstrates ROME's automatic routing abilities. ROME's ability to automatically route orthogonal connections, which consist of a sequence of horizontal and vertical line segments, provides a clean environment for designing datapath-based designs.

7.3.3 VHDL Code Generation

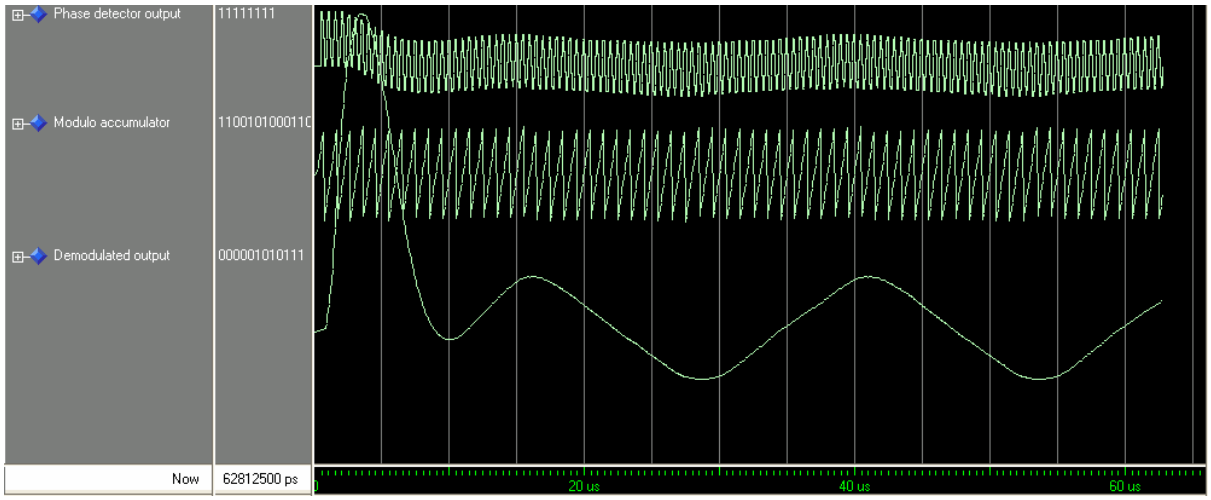
Listing 9.18 presents the logged output as a result of calling xps_gen from with in the ROME environment. According to this output log: a top level and a top level stub VHDL file as well as VHDL wrappers, for each instance, were generated successfully.

7.3.4 Simulation and Validation

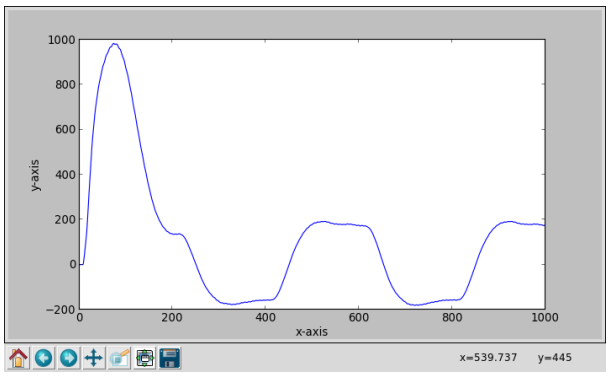
The generated VHDL files were simulated in Xilinx's iSim with a modulated square wave data and a modulated triangle wave data provided by [88]. The resulting demodulated waveforms were written to a file and the file's contents represented using Matplotlib's pyplot. Figures 7.3a and 7.3b illustrate the results for the demodulated square wave and the demodulated triangle wave, presented in [88], respectively. Figures 7.3c and 7.3d present pyplots that illustrate the results of simulating the ROME generated vhdl in iSim. Figures 7.3c presents the results from demodulating the modulated square wave, while Figures 7.3d presents the results from demodulating the modulated triangle wave. Unfortunately, Rahmatullah did not provide his results, in [88], as raw data. Thus, it was not possible to validate the results properly. Had this data been provided, the demodulated waveforms produced by ROME could be correlated with those presented in [88]. However a form of validation can be achieved by performing a visual inspection and comparison between Figures 7.3a and 7.3c and between Figures 7.3b and 7.3d



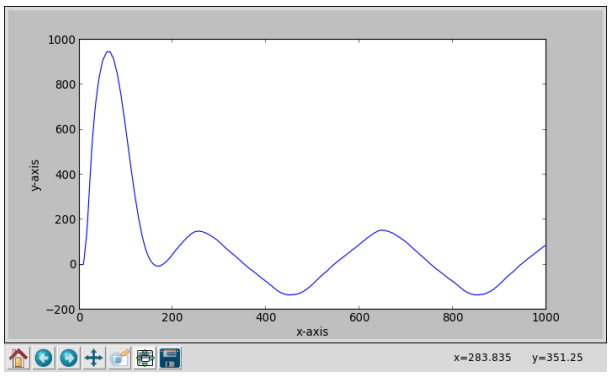
(a) ModelSim Square Wave. Adapted from [88]



(b) ModelSim Tri-Wave. Adapted from [88]



(c) Square Wave



(d) Triangle Wave

Figure 7.3: FM Demodulation Results: screenshots illustrating the expected results for a modulated (a) square wave and (b) triangle wave as well as the simulated results for a modulated (c) square wave and (d) triangle wave.

Chapter 8

Conclusions

This chapter discusses the results of the experiments presented in chapters 5, 6 and 7 as well as presents conclusions, about the system, based on the discussion. Finally, ideas for future work are explored.

8.1 Discussion of Results

A discussion of results, of the experiments presented in chapters 5, 6 and 7, will be presented in this section as well as conclusions, about the system, based on this discussion.

8.1.1 Case Study: Waveform Generator

The purpose of the Waveform Generator experiment was to test whether changes could be made to generics/parameters, from within the GUI of the ROME environment, and whether these changes would reflect in the VHDL code that ROME generates.

The MHS file pesented in Listing 5.4 shows that the *phase_inc* and *wave* parameters, set from the property menu within ROME, are reflected in the generated MHS file. The output log of ROME, presented in Listing 9.7, shows that ROME successfully generated VHDL wrappers for the waveform generator design. The log also shows that the design was successfully synthesised by XST. The synthesis process took 53.00*seconds* to complete.

Figures 5.4 and 5.5 verify the correctness of the VHDL wrappers, generated by ROME. It is evident from Figure 5.4 that the *wave* parameter setting was successfully translated from the ROME environment to the generated code. Upon investigation of the figures, 5.4 and 5.5, the *phase_inc* parameter can be verified as well. In Figure 5.4, the frequency of the produced waves, $Fout = (phase\_inc * Fclk)/2^{32}$, should be $1.7MHz$ when $Fclk = 100MHz$. In the simulated time, of 6 micro seconds, the waveforms produce 11 peaks. Since, $freq = phase/time$, the frequency of the waveforms in Figure 5.4,

is $1.83MHz$. Similarly, the frequencies can be validated for Figure 5.5. Figure 5.5b oscillates 21 times in 6μs thus oscillating at $3.5MHz$. While Figure 5.5d oscillates 6 times, in the 6μs simulation, thus oscillating at a frequency of $1MHz$.

When comparing these produced waveforms with their expected outputs, it can be seen that there is an error of approx. $0.1MHz$. This can be attributed to the fact that the waveform generator core only supports numeric_std arithmetic and does not support fixed or floating point arithmetic.

In conclusion the Waveform Generator experiment showed that, generic/parameter values can be set from within the ROME environment and these values are reflected in the VHDL wrappers generated by the tool.

8.1.2 Case Study: Wishbone Adder

The purpose of this experiment was to demonstrate ROME's ability to abstract a bus structure away from the user as well as to compare the abstraction method of the proposed tool with that of Xilinx ISE and XPS.

ROME required the MPD files, for all cores with a wishbone interface, be prepared prior to importing them into the ROME environment. For this experiment the preparation process involved altering: 16 lines of code from the wishbone bus MPD as well as 8 lines from both the GPMC_Wishbone bridge and register MPDs.

When comparing ROME's abstraction ability (Figure 6.8) to that of XPS (Figure 6.9a) and ISE (Figure 6.9a), it can be seen that ROME is comparable with XPS. Both XPS and ROME successfully abstract away the wishbone bus structure, both representing the wishbone interconnection with a single connection where as ISE offers no bus abstraction at all. It can be seen from Figure 6.9a that XPS represents master interfaces with a square and slave interfaces with a circle. The wishbone interface are also placed on the top and bottom of the bus symbol in XPS. The diagram in Figure 6.9a however cannot be edited as this is not an available function of the tool. XPS designers are required to build models in views like those illustrated in Figures 6.10a and 6.10b. Conversely, both ROME and ISE offer editable schematic design entry.

This experiment also serves to show that Xilinx's flow can be called from within the ROME environment. Figure 6.11 demonstrates that ROME is capable of synthesising and implementing designs as well as generating programming bitstreams.

In conclusion the Wishbone Adder experiment demonstrated that ROME is capable of abstracting a bus structure away from the user. This abstraction ability is comparable

to that of XPS's diagram viewer, however ROME's schematic models are editable where as XPS's are not.

8.1.3 Case Study: Digital FM Receiver

The purpose of the FM Receiver experiment was to demonstrate the use of the ROME environment in the context of a typical SDR application. The experiment aimed to show that a data path based SDR design could be built and verified using the ROME environment.

Figure 7.2 demonstrates ROME's automatic orthogonal routing abilities. ROME's ability to automatically route connections provides a clean environment for designing data path-based designs.

The simulated results shown in Figures 7.3c and 7.3d serve to validate that designs can be built and validated with ROME in a SDR context.

8.2 Recommendations for Future Work

VHDL is a strongly typed description language with a context sensitive grammar [67]. As mentioned in section 2.4, due to the language's strict restrictions and wide prospects, VHDL has become one of the most difficult language to analyze [67]. It is for this reason, the Xilinx tools were chosen to transform VHDL modules into a simpler intermediate format. Relying on these well tested commercial tools, means that ROME has similar compatibilities and features as the Xilinx tools. However this also means that ROME has similar issues as the these tools as well. For example, as mentioned in section 2.8.2, it is possible to generate invalid VHDL or Verilog, irrespective of the syntactical correctness of the generated MHS, since MHS is a neutral format that lies on top of the HDL [100]. It is therefore recommended that further work be done in order to reduce ROME's dependency on the Xilinx tools.

During a review of initial user requirements for this project, a set of functional requirement were developed in order to satisfy the initial user requirement. Refer to section 1.2.2 for a full list of these requirements. As mentioned in section 1.2.2, functional requirements, F6, F7, F9 and F10 were deemed low priority and out of the scope of this dissertation. A list, describing these low priority requirement:

F6. It would be very useful to have colour coded blocks, i.e. interfaces, controllers, primitives, DSP blocks etc.

F7. Provide the ability to group a number of blocks together to create another block. This hierarchy of sub-systems could help to reduce the complexity of large designs.

F9. The GUI could provide a view of the data path's source code, where one can follow instantiations back to the implementation of modules and visa-versa.

F10. The GUI could show real-time any syntax errors in the design.

However these requirements are still considered "nice-to-haves". It is therefore recommended that work be done to implement these functional requirements in future iterations of this work.

Section 1.1.4 emphasizes the importance of high level abstraction techniques in reducing the development time of complex systems. Graphical signal processing environments like MATLAB Simulink and the GnuRadio Companion provide clock signal abstraction. In other words clocking signals are not visibly connected to the processing blocks. Although ROME does offer bus interface abstraction, the tool does not support clock signal abstraction. Therefore, further work towards the abstraction of clocking signals is recommended.

An issue with the project's initial requirements was presented in section 1.2.1. A less ambiguous definition for the term "easy to use" was presented. Although ROME makes use of common practices in GUI and EDA design for making tools easier to use, a survey, on the effects of these practices on ease of use, has not been performed. It is therefore recommended that a survey, involving professional and novice developers, be performed in order to analyse and quantify the degree of ease that ROME offers.

8.3 Summary

This paper reported on the design for a tool for rapid prototyping of software defined radio applications on a reconfigurable computing platform. A tool was developed with the ability to: add, delete, move and connect graphical blocks; save and restore designs as well as generate top-level HDL code.

High level abstraction techniques were used to reduce the time to develop complex systems. The tool, presented in this dissertation, made use of abstraction techniques such as: graphical representations of system blocks; automatic code generation of parametric blocks; Intellectual Property (IP) reuse and bus interface abstraction.

A set of experiments were performed in the context of three case studies. The Waveform Generator experiment showed that, generic/parameter values can be set from within the ROME environment and these values are then reflected in the VHDL wrappers generated by the tool. The Wishbone Adder experiment demonstrated that ROME is capable of abstracting a bus structure away from the user. This abstraction ability is comparable to that of XPS's diagram viewer, however ROME's schematic models are editable where as

XPS's are not. Finally the FM Receiver experiment, served to prove that SDR designs can be built and validated within ROME environment.

Although the developed tool is limited in its functionality, it serves to prove the concept of the design as well as supplies a platform for future study on the effects of high level abstraction on the productivity of SDR developers and students.

Bibliography

[1] S. Winberg, A. Mishra, and B. Raw, "Rhino blocks pulse-doppler radar framework," in *Parallel Distributed and Grid Computing (PDGC), 2012 2nd IEEE International Conference on*, pp. 876 –881, dec. 2012.

[2] D. Castells, M. Monton, R. Pla, D. Novo, A. Portero, O. Navas, J. Farre, L. Ribas, and C. J., "Comparing design flows for structural system level specifications facing fpga platforms," in *XIX Conference on Design of Circuits and Integrated Systems.(DCIS)*, November 2004.

[3] S. Katz, S. Winberg, and A. Mishra, "Rapid model-based prototyping tool for sdr on a rc platform," in *Parallel Distributed and Grid Computing (PDGC), 2012 2nd IEEE International Conference on*, pp. 711–715, 2012.

[4] "Gnu radio: Learning by example." [ONLINE] Available: `http://www.joshknows.com/gnuradio#example`, 2013.

[5] M. Sadiku and C. Akujuobi, "Software-defined radio: a brief overview," *Potentials, IEEE*, vol. 23, pp. 14 – 15, oct.-nov. 2004.

[6] S. Winberg, A. Langman, and S. Scott, "The rhino platform: charging towards innovation and skills development in software defined radio," in *Proceedings of the South African Institute of Computer Scientists and Information Technologists Conference on Knowledge, Innovation and Leadership in a Diverse, Multidisciplinary Environment*, SAICSIT '11, (New York, NY, USA), pp. 334–337, ACM, 2011.

[7] F. R. Wagner, F. A. Nascimento, and M. F. Oliveira, "Model-driven engineering of complex embedded systems: Concepts and tools,"

[8] G. E. Inggs, "Putting the pieces together: The systematic development of a software defined radio toolflow for the rhino project," Master's thesis, University of Cape Town, 2011.

[9] D. Macko and K. Jelemenska, "Vhdlvisualizer: Hdl model visualization with simulation-based verification," in *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2012 IEEE 15th International Symposium on*, pp. 199 –200, april 2012.

[10] J.-P. Deschamps, G. D. Sutter, E. Cant, J.-P. Deschamps, G. D. Sutter, and E. Cant, "Eda tools," in *Guide to FPGA Implementation of Arithmetic Functions*, vol. 149 of *Lecture Notes in Electrical Engineering*, pp. 127–151, Springer Netherlands, 2012. 10.1007/978-94-007-2987-2_6.

[11] K. Jelemenska, M. Nosal, and P. Cicak, "Visualization of verilog digital systems models," in *International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering (CISSE 10), Bridgeport, Connecticut, USA*, Dec 2010.

[12] D. Macko and K. Jelemenska, "Vhdl structural model visualization," in *EUROCON - International Conference on Computer as a Tool (EUROCON), 2011 IEEE*, pp. 1 –4, april 2011.

[13] D. Valerio, "Open source software-defined radio: A survey on gnu-radio and its applications," *Forschungszentrum Telekommunikation Wien, Vienna, Technical Report FTW-TR-2008-002*, 2008.

[14] J. Mitola, "The software radio architecture," *Communications Magazine, IEEE*, vol. 33, no. 5, pp. 26–38, 1995.

[15] E. Buracchini, "The software radio concept," *Communications Magazine, IEEE*, vol. 38, no. 9, pp. 138–143, 2000.

[16] A. Abidi, "The path to the software-defined radio receiver," *Solid-State Circuits, IEEE Journal of*, vol. 42, no. 5, pp. 954–966, 2007.

[17] T. Debatty, "Software defined radar a state of the art," in *Cognitive Information Processing (CIP), 2010 2nd International Workshop on*, pp. 253–257, 2010.

[18] B. F. Burke and F. Graham-Smith, *An introduction to radio astronomy*. Cambridge University Press, 2009.

[19] S. Scott, "Rhino: Reconfigurable hardware interface for computation and radio," Master's thesis, Department of Electrical Engineering at the University of Cape Town, 2011.

[20] "Rasdr at sara conference in july 2013." [ONLINE] Available: `http://www.radio-astronomy.org/node/139`, 2013.

[21] J.-P. Delahaye, C. Moy, P. Leray, and J. Palicot, "Managing dynamic partial reconfiguration on heterogeneous sdr platforms," in *SDR Forum Technical Conference*, vol. 5, 2005.

[22] K. Tan, H. Liu, J. Zhang, Y. Zhang, J. Fang, and G. M. Voelker, "Sora: high-performance software radio using general-purpose multi-core processors," *Commun. ACM*, vol. 54, pp. 99–107, Jan. 2011.

[23] M.-J. SHEEN, L. Seung-Hwan, C. Kyoung-Rok, *et al.*, "A dsp-based reconfigurable sdr platform for 3g systems," *IEICE transactions on communications*, vol. 88, no. 2, pp. 678–686, 2005.

[24] H.-M. Bluethgen, C. Grassmann, W. Raab, and U. Ramacher, "A programmable platform for software-defined radio," in *System-on-Chip, 2003. Proceedings. International Symposium on*, pp. 15–, 2003.

[25] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "Soda: A high-performance dsp architecture for software-defined radio," *Micro, IEEE*, vol. 27, no. 1, pp. 114–123, 2007.

[26] M. Safadi and D. Ndzi, "Digital hardware choices for software radio (sdr) baseband implementation," in *Information and Communication Technologies, 2006. ICTTA'06. 2nd*, vol. 2, pp. 2623–2628, IEEE, 2006.

[27] L. Belanger, "Advanced sdr platform eases multiprotocol radio development," *RF DESIGN*, vol. 30, no. 1, p. 36, 2007.

[28] "Ettus research products: Usrp table." [ONLINE] Available: `https://www.ettus.com/product#tabs-2`, 2013.

[29] "Ettus research products: Usrp daughterboard table." [ONLINE] Available: `https://www.ettus.com/product#tabs-2`, 2013.

[30] G. Inggs, D. Thomas, and S. Winberg, "Building a rhino harness a software defined radio toolflow for rapid prototyping upon fpgas," in *Industrial Technology (ICIT), 2013 IEEE International Conference on*, pp. 1098–1103, IEEE, 2013.

[31] H. K.-H. So, *BORPH: An operating system for FPGA-based reconfigurable computers*. ProQuest, 2007.

[32] J. V. der Spiegel, "Vhdl tutorial." [ONLINE] Available: `http://www.seas.upenn.edu/~ese171/vhdl/vhdl_primer.html`, May 2013. University of Pennsylvania Department of Electrical and Systems Engineering.

[33] P. P. Chu, *FPGA prototyping by VHDL examples: Xilinx Spartan-3 version*. Wiley-Interscience, 2008.

[34] M. Hofstra, "Comparing hardware description languages." [Online] Available: `http://referaat.cs.utwente.nl/TSConIT/download.php?id=1144`, June 2012.

[35] D. L. Perry, *VHDL: Programming by Example*. New York, NY, USA: McGraw-Hill, Inc., 4 ed., 2002.

[36] A. Dewey, "The vhsic hardware description language (vhdl) program," in *Proceedings of the 21st Design Automation Conference*, DAC '84, (Piscataway, NJ, USA), pp. 556–557, IEEE Press, 1984.

[37] D. MacMillen, R. Camposano, D. Hill, and T. Williams, "An industrial view of electronic design automation," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 19, pp. 1428 –1448, dec 2000.

[38] S. Palnitkar, *Verilog® hdl: a guide to digital design and synthesis, second edition*. Upper Saddle River, NJ, USA: Prentice Hall Press, second ed., 2003.

[39] B. Fawcett, "Tools to speed fpga development," *Spectrum, IEEE*, vol. 31, no. 11, pp. 88–94, 1994.

[40] C. Medrano, I. Plaza, M. Castro, F. Garci anda Sevilla, J. Marti andnez Calero, J. Felix, and M. Corbala andn, "A review of electronic engineering design free software tools," in *Education Engineering (EDUCON), 2010 IEEE*, pp. 1867 –1871, april 2010.

[41] A. Kamppi, L. Matilainen, J. Maatta, E. Salminen, T. Hamalainen, and M. Hannikainen, "Kactus2: Environment for embedded product development using ip-xact and mcapi," in *Digital System Design (DSD), 2011 14th Euromicro Conference on*, pp. 262–265, 2011.

[42] "Fritzing." [Online] Available: `http://fritzing.org/developer/`.

[43] "Qelectrotech." [Online] Available: `http://qelectrotech.org/contact.html`.

[44] M. E. Brinson and S. Jahn, "Qucs: A gpl software package for circuit simulation, compact device modelling and circuit macromodelling from dc to rf and beyond," *International Journal of Numerical Modelling: Electronic Networks, Devices and Fields*, vol. 22, no. 4, pp. 297–319, 2009.

[45] S. Meiyappan, K. Jaramillo, and P. Chambers, "10 tips for generating reusable vhdl," August 1999.

[46] A. Arnesen, N. Rollins, and M. Wirthlin, "A multi-layered xml schema and design tool for reusing and integrating fpga ip," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pp. 472–475, 2009.

[47] N. Rollins, A. Arnesen, and M. Wirthlin, "An xml schema for representing reusable ip cores for reconfigurable computing," in *Aerospace and Electronics Conference, 2008. NAECON 2008. IEEE National*, pp. 190–197, IEEE, 2008.

[48] C. Neely, G. Brebner, and W. Shang, "Shapeup: A high-level design approach to simplify module interconnection on fpgas," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pp. 141–148, 2010.

[49] T. Balderas-Contreras, R. Cumplido, and G. Rodríguez, "Synthesizing vhdl from activity models in uml 2," *International Journal of Circuit Theory and Applications*, 2012.

[50] O. S. Initiative, "Ieee standard systemc language reference manual," *IEEE Computer Society*, pp. 1666–2005, 2006.

[51] "Fpga c compiler." [ONLINE] Available: `http://sourceforge.net/projects/fpgac/`, 2013.

[52] "Impulse c." [ONLINE] Available: `http://www.impulseaccelerated.com/`, 2013.

[53] P. Bellows and B. Hutchings, "Jhdl-an hdl for reconfigurable systems," in *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pp. 175–184, IEEE, 1998.

[54] J. Decaluwe, "Myhdl: a python-based hardware description language," *Linux journal*, vol. 2004, no. 127, p. 5, 2004.

[55] "Migen: A python toolbox for building complex digital hardware." [ONLINE] Available: `https://github.com/milkymist/migen`, 2013.

[56] C. Chang, *Design and applications of a reconfigurable computing system for high performance digital signal processing*. PhD thesis, UNIVERSITY OF CALIFORNIA, 2005.

[57] E. Blossom, "Gnu radio: tools for exploring the radio frequency spectrum," *Linux J.*, vol. 2004, pp. 4–, June 2004.

[58] D. M. Beazley *et al.*, "Swig: An easy to use tool for integrating scripting languages with c and c++," in *Proceedings of the 4th USENIX Tcl/Tk workshop*, pp. 129–139, 1996.

[59] N. Manicka, *GNU radio testbed*. PhD thesis, University of Delaware, 2007.

[60] "Gnu radio companion wiki." [ONLINE] Available: `http://gnuradio.org/redmine/projects/gnuradio/wiki/GNURadioCompanion`, 2013.

[61] C.-Y. Chen, F.-H. Tseng, K.-D. Chang, H.-C. Chao, J.-L. Chen, *et al.*, "Reconfigurable software defined radio and its applications," *Tamkang Journal of Science and Engineering*, vol. 13, no. 1, p. 2938, 2010.

[62] "Gnu radio companion." [ONLINE] Available: http://www.joshknows.com/grc, 2013.

[63] "Casper: Collaboration for astronomy signal processing and electronics research." [ONLINE] Available: https://casper.ssl.berkeley.edu/, 2013.

[64] "Rapid development of radio astronomy instrumentation using open source fpga boards, tools and libraries." [ONLINE] Available: https://casper.berkeley.edu/presentations/casper_oxford.pdf, 2013.

[65] "Casper: Mssge toolflow." [ONLINE] Available: https://casper.ssl.berkeley.edu/wiki/MSSGE_Toolflow, 2013.

[66] I. Bohm, "Automatic code generation using dynamic programming techniques," Master's thesis, Johannes Kepler University of Linz, 2007.

[67] D. Rahmati, A. S. Zebardast, M. H. Reshadi, and Z. Navabi, "Handling complex vhdl semantics with an oo intermediate format," in *Electrical and Computer Engineering, 2001. Canadian Conference on*, vol. 2, pp. 1273–1278, IEEE, 2001.

[68] M. Reshadi, B. Goji-Ara, and Z. Navabi, "Hdml: compiled vhdl in xml," in *VHDL International Users Forum Fall Workshop, 2000. Proceedings*, pp. 69 –74, 2000.

[69] D. R. Griffin, "A vhdl interpreter for model-based diagnoses.," Master's thesis, AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING, 1992.

[70] W. Ecker, M. Heuchling, J. Mades, T. Schneider, A. Windisch, and K. Yang, "Using xml for vhdl model representation," in *World Computer Congress*, 2000.

[71] *IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows.* IEEE Std 1685™-2009, 3 Park Avenue, New York, NY 10016-5997, USA: IEEE, Feb. 18 2010.

[72] W. Kruijtzer, P. van der Wolf, E. de Kock, J. Stuyt, W. Ecker, A. Mayer, S. Hustin, C. Amerijckx, S. de Paoli, and E. Vaumorin, "Industrial ip integration flows based on ip-xact standards," in *Design, Automation and Test in Europe, 2008. DATE '08*, pp. 32–37, 2008.

[73] T. Perry, R. Walke, and K. Benkrid, "An extensible code generation framework for heterogeneous architectures based on ip-xact," in *Programmable Logic (SPL), 2011 VII Southern Conference on*, pp. 81–86, 2011.

[74] F. Risso and M. Baldi, "Netpdl: An extensible xml-based language for packet header description," *Computer Networks*, vol. 50, no. 5, pp. 688 – 706, 2006.

[75] E. A. Lee and S. Neuendorffer, *MoML: A Modeling Markup Language in SML: Version 0.4*. Citeseer, 2000.

[76] "Gezel: Hardware/software codesign environment." [Online] Available: `http://rijndael.ece.vt.edu/gezel2/`.

[77] "Hdlmaker." [Online] Available: `http://www.polybus.com/hdlmaker/hdlmaker.html`.

[78] A. Kountouris and C. Wolinski, "A method for the generation of hdl code at the rtl level from a high-level formal specification language," in *Circuits and Systems, 1997. Proceedings of the 40th Midwest Symposium on*, vol. 2, pp. 1095–1098 vol.2, 1997.

[79] R. Damasevicius and V. Stuikys, "Application of uml for hardware design based on design process model," in *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, pp. 244–249, IEEE Press, 2004.

[80] T. G. Moreira, M. A. Wehrmeister, C. E. Pereira, J.-F. Pétin, and E. Levrat, "Generating vhdl source code from uml models of embedded systems," in *Distributed, Parallel and Biologically Inspired Systems*, pp. 125–136, Springer, 2010.

[81] S. Le Beux, P. Marquet, A. Honoré, and J.-L. Dekeyser, "A model driven engineering design flow to generate vhdl," in *Proceedings of the International Workshop on Model Driven Design for Automotive Safety Embedded Systems (ModEasy'07)*, pp. 15–22, 2007.

[82] "Platform specification format reference manual." [ONLINE] Available: `http://www.xilinx.com/support/documentation/sw_manuals/edk10_psf_rm.pdf`, 2013.

[83] Xilinx, *Embedded System Tools Reference Manual EDK*, ug111 (v14.1) ed., April 2012.

[84] "Pythonqt: Introduction." [ONLINE] Available: `http://pythonqt.sourceforge.net/`, 2013.

[85] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[86] "Github repository: mjwybrow/adaptagrams." [ONLINE] Available: `https://github.com/mjwybrow/adaptagrams/tree/b04ce8ac3900a1b08abd6c6322def6a7e71dbe8c`, 2013.

[87] S. Doherty, *NCO / Periodic waveform generator*. OpenCores.org, Oct 2008.

[88] N. Rahmatullah, *Design of All Digital FM Receiver Circuit*. Institut Teknologi Bandung, March 2005.

[89] "Qtxml: Qdomdocument class." [ONLINE] Available: `http://qt-project.org/doc/qt-5.0/qtxml/qdomdocument.html`, 2013.

[90] "Overview of qt's undo framework." [ONLINE] Available: `http://qt-project.org/doc/qt-4.8/qundo.html`, 2013.

[91] "Qt project documentation: Drag and drop." [ONLINE] Available: `http://qt-project.org/doc/qt-4.8/dnd.html`, 2013.

[92] "Adaptagrapms: Tools for adaptive diagrams." [ONLINE] Available: `http://www.adaptagrams.org`, 2013.

[93] "Marvl: Monash adaptive visualisation lab." [Online] Available: `http://marvl.infotech.monash.edu.au/`.

[94] M. Wybrow, K. Marriott, and P. J. Stuckey, "Orthogonal connector routing," in *Graph Drawing*, pp. 219–231, Springer, 2010.

[95] "Adaptagrapms: libavoid." [ONLINE] Available: `http://www.adaptagrams.org/documentation/libavoid.html`, 2013.

[96] "Qt paint system." [ONLINE] Available: `http://qt-project.org/doc/qt-4.8/paintsystem.html`, 2013.

[97] "Casper toolflow - tutorial 1: Introduction to simulink." [ONLINE] Available: `https://casper.berkeley.edu/wiki/Introduction_to_Simulink`, 2013.

[98] "Wishbone b4: Wishbone system-on-chip (soc)interconnection architecturefor portable ip cores." [ONLINE] Available: `cdn.opencores.org/downloads/wbspec_b4.pdf`, 2013.

[99] "Altera - vhdl: Adder/subtractor." [ONLINE] Available: `http://www.altera.com/support/examples/vhdl/vhd-add-sub.html`, 2013.

[100] Xilinx, *Platform Specification Format Reference Manual Embedded Development Kit (EDK) 13.2*, ug642 (v13.2) ed., July 2011.

Chapter 9

Appendices

9.1 Appendix: The Attached CD

The directory structure of the attached CD is the following:

- **case_studies/** : This directory holds implementations of the experiments performed in this dissertation.
  - **fm_receiver/** : This is the directory for the FM Receiver case study experiment. The directory has both ROME and ISE implementations.
  - **waveform_gen/** : This is the directory for the Waveform Generator case study experiment. The directory has both ROME and ISE implementations.
  - **wishbone_adder/** : This is the directory for the Wishbone Adder case study experiment. The directory has both ROME and XPS implementations.
- **data_collection/** : This directory hold implementations of tools that helped with data collection.
  - **file io/** : This is the directory for all simulation VHDL files which helped with reading simulation data in to and writing simulated data out of the simulation environment.
  - **pyplotting/** : This is the directory for the implementation of a pyplot based script to plot simulation data for visualisation.
- **rome/** : This is the directory for the entire Qt/C++ implementation of ROME
- **dissertation.pdf** : This is a digital PDF copy of this dissertation.

9.2 Appendix: Misc

Listing 9.1: GNU Radio Tone-Gen Example: GNU Radio Python code illustrating the way in which systems are described with in the GNU Radio framework. Adapted from [4]

```
#!/usr/bin/env python
from gnuradio import gr
from gnuradio import audio

#create the flow graph
tb = gr.top_block()

#create the signal sources
#parameters: samp_rate, type, output freq, amplitude, offset
src1 = gr.sig_source_f(32000, gr.GR_SIN_WAVE, 350, .5, 0)
src2 = gr.sig_source_f(32000, gr.GR_SIN_WAVE, 440, .5, 0)

#an adder to combine the sources
#the _ff indicates float input and float output
adder = gr.add_ff()

#create a signal sink
sink = audio.sink(32000)

#connect the adder
#the adder has multiple inputs...
#we must use this syntax to choose which input to use
tb.connect(src1, (adder, 0))
tb.connect(src2, (adder, 1))

#connect the adder to the sink
tb.connect(adder, sink)

#run the flow graph
tb.run())
```

Listing 9.2: XPS_GEN: a Python script used to coordinate compilation processes between ROME and the Xilinx tools.

```
#-------------------------------------------------------------------------------
# AUTHOR: Shaun Katz
# University of Cape Town (UCT)
# Department of Electrical Engineering
# Software Defined Radio Group (SDRG)
# DATE: 24/04/2012
#-------------------------------------------------------------------------------

import os
import sys
import platform
from optparse import OptionParser

parser = OptionParser()
parser.add_option("-s", "--synthesis",
                  action="store_true", dest="synthesis", default=False,
                  help="Synthesize - XST")
parser.add_option("-i", "--implement",
                  action="store_true", dest="implement", default=False,
                  help="Implement Design")
parser.add_option("-g", "--generate",
                  action="store_true", dest="generate", default=False,
                  help="Generate Programming File")
parser.add_option("-l", "--library_path",
                  action="store", dest="library_path", default=os.getcwd()+'/pcores/',
                  help="Library Search Path")
(options, args) = parser.parse_args()

# #os.system("xps_gen")

PLATFORM = platform.system()
XILINX_VER = '14.1'
if (PLATFORM == 'Linux') :
        print 'Linux Detected'
        os.environ['XILINX'] = '/opt/Xilinx/' + XILINX_VER + '/ISE_DS/ISE'
        os.environ['PATH'] += ':/opt/Xilinx/' + XILINX_VER + '/ISE_DS/EDK/bin/lin64'
        os.environ['PATH'] += ':/opt/Xilinx/' + XILINX_VER + '/ISE_DS/ISE/bin/lin64'
```

```
        CMD_PLATGEN = "platgen -p xc6slx16csg324-3 -lang vhdl -intstyle default -lp " + options.library_path + " system.mhs"
        CMD_SYNTHESIS = "cd synthesis && ./synthesis.sh && cd .."
        CMD_COPY_OPT = "cp etc/fast_runtime.opt implementation/fast_runtime.opt"
        CMD_COPY_UCF = "cp data/system.ucf implementation/system.ucf"
        CMD_XFLOW = "xflow -wd implementation -p xc6slx16csg324-3 -implement fast_runtime.opt system.ngc"
        CMD_XILPERL = "xilperl /opt/Xilinx/" + XILINX_VER + "/ISE_DS/EDK/data/fpga_impl/observe_par.pl -error yes
            implementation/system.par"
        CMD_COPY_UT = "cp etc/bitgen.ut implementation/bitgen.ut"
        CMD_BITGEN = "cd implementation && bitgen -w -f bitgen.ut system && cd .."
elif (PLATFORM == 'Windows') :
        print 'Windows Detected'
        os.environ['XILINX'] = 'C:/Xilinx/13.4/ISE_DS/ISE'
        os.environ['PATH'] += ';C:/Xilinx/13.4/ISE_DS/EDK/bin/nt64'
        os.environ['PATH'] += ';C:/Xilinx/13.3/ISE_DS/ISE/bin/nt64'

        CMD_PLATGEN = "platgen -p xc6slx16csg324-3 -lang vhdl -intstyle default -lp " + options.library_path + " system.mhs"
        CMD_SYNTHESIS = "cd synthesis & synthesis.cmd & cd .."
        CMD_COPY_OPT = "echo f|xcopy etc\\fast_runtime.opt implementation\\fast_runtime.opt /Y"
        CMD_COPY_UCF = "echo f|xcopy data\\system.ucf implementation\\system.ucf /Y /I"
        CMD_XFLOW = "xflow.exe -wd implementation -p xc6slx16csg324-3 -implement fast_runtime.opt system.ngc"
        CMD_XILPERL = "xilperl C:/Xilinx/13.4/ISE_DS/EDK/data/fpga_impl/observe_par.pl -error yes implementation/system.par"
        CMD_COPY_UT = "echo f|xcopy etc\\bitgen.ut implementation\\bitgen.ut /Y"
        CMD_BITGEN = "cd implementation & bitgen -w -f bitgen.ut system & cd .."
else :
        print 'Platform Not Supported'
        exit()

if (options.synthesis):
        #print('#########################')
        print '## Synthesize - XST ##'
        #print('#########################')
        sys.stdout.flush()
        os.system(CMD_PLATGEN)
        os.system(CMD_SYNTHESIS)

if (options.implement):
        #print('#########################')
        print('## Implement Design ##')
        # print('#########################')
        sys.stdout.flush()
        os.system(CMD_COPY_OPT)
        os.system(CMD_COPY_UCF)
        os.system(CMD_XFLOW)
        os.system(CMD_XILPERL)

if (options.generate):
        #print('###################################')
        print('## Generate Programming File ##')
        #print('###################################')
        sys.stdout.flush()
        os.system(CMD_COPY_UT)
        os.system(CMD_BITGEN)
```

9.3 Appendix: Waveform Generator

Listing 9.3: Waveform Generator VHDL: VHDL implementation of a waveform generator.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity waveform_gen is
generic(
            phase_inc : integer := 73014444; -- Fout = 1.7MHz when Fclk = 100MHz
                                             -- phase_inc = (Fout/Fclk)*(2^32)
            wave : string := "SINE" ); -- Options: "SINE", "COSINE", "SQUARE", "SAW"
port (
  -- system signals
  clk : in std_logic;
  reset : in std_logic;

  -- clock-enable
  en : in std_logic;

  -- Output waveform
  output : out std_logic_vector(11 downto 0));
end entity;


architecture rtl of waveform_gen is

    component sincos_lut
        port (
                clk : in std_logic;
                en : in std_logic;
                addr : in std_logic_vector(11 downto 0);
                sin_out : out std_logic_vector(11 downto 0);
                cos_out : out std_logic_vector(11 downto 0));
    end component;

    signal phase_acc : std_logic_vector(32-1 downto 0);
    signal lut_addr : std_logic_vector(11 downto 0);
    signal lut_addr_reg : std_logic_vector(11 downto 0);

    signal sin_out : std_logic_vector(11 downto 0);
    signal cos_out : std_logic_vector(11 downto 0);
    signal squ_out : std_logic_vector(11 downto 0);
    signal saw_out : std_logic_vector(11 downto 0);

begin

-------------------------------------------------------------------------------
-- Phase accumulator increments by 'phase_inc' every clock cycle --
-- Output frequency determined by formula: Phase_inc = (Fout/Fclk)*2^32 --
-- E.g. Fout = 36MHz, Fclk = 100MHz, Phase_inc = 36*2^32/100 --
-- Frequency resolution is 100MHz/2^32 = 0.00233Hz --
-------------------------------------------------------------------------------

    phase_acc_reg: process(clk, reset)
    begin
        if reset = '0' then
            phase_acc <= (others => '0');
        elsif clk'event and clk = '1' then
            if en = '1' then
        --phase_acc <= unsigned(phase_acc) + unsigned(phase_inc);
                phase_acc <= unsigned(phase_acc) + phase_inc;
            end if;
        end if;
    end process phase_acc_reg;

-------------------------------------------------------------------------
-- use top 12-bits of phase accumulator to address the SIN/COS LUT --
-------------------------------------------------------------------------

    lut_addr <= phase_acc(31 downto 20);

-------------------------------------------------------------------------
-- SIN/COS LUT is 4096 by 12-bit ROM --
-- 12-bit output allows sin/cos amplitudes between 2047 and -2047 --
-- (-2048 not used to keep the output signal perfectly symmetrical) --
-- Phase resolution is 2Pi/4096 = 0.088 degrees --
-------------------------------------------------------------------------
```

```
    lut: sincos_lut
    port map (
                  clk => clk,
                  en => en,
                  addr => lut_addr,
                  sin_out => sin_out,
                  cos_out => cos_out );

-----------------------------------
-- Hide the latency of the LUT --
-----------------------------------

    delay_regs: process(clk)
    begin
        if clk'event and clk = '1' then
            if en = '1' then
                lut_addr_reg <= lut_addr;
            end if;
        end if;
    end process delay_regs;

-----------------------------------------------
-- Square output is msb of the accumulator --
-----------------------------------------------

    squ_out <= "011111111111" when lut_addr_reg(11) = '1' else "100000000000";

-------------------------------------------------------------
-- Sawtooth output is top 12-bits of the accumulator --
-------------------------------------------------------------

    saw_out <= lut_addr_reg;

    latch_output: process(clk)
    begin
        if clk'event and clk = '1' then
            if en = '1' then
                if wave = "SINE" then
                    output <= sin_out;
                elsif wave = "COSINE" then
                    output <= cos_out;
                elsif wave = "SQAURE" then
                    output <= squ_out;
                elsif wave = "SAW" then
                    output <= saw_out;
                end if;
            end if;
        end if;
    end process latch_output;

end rtl;
```

Listing 9.4: Waveform Generator MPD: The MPD file generated while importing the waveform generator module to the ROME environment

```
##############################################################################
##
## Name : waveform_gen
## Desc : Microprocessor Peripheral Description
## : Automatically generated by PsfUtility
##
##############################################################################

BEGIN waveform_gen

## Peripheral Options
OPTION IPTYPE = PERIPHERAL
OPTION IMP_NETLIST = TRUE
OPTION HDL = MIXED
OPTION IP_GROUP = USER


## Bus Interfaces

## Generics for VHDL or Parameters for Verilog
PARAMETER phase_inc = 73014444, DT = INTEGER
PARAMETER wave = SINE, DT = STRING
```

```
## Ports
PORT clk = "", DIR = I
PORT reset = "", DIR = I
PORT en = "", DIR = I
PORT output = "", DIR = O, VEC = [11:0]

END
```

Listing 9.5: Waveform Generator XML: The XML file generated as a result of saving the waveform generator in the ROME environment.

```
<!DOCTYPE MyML>
<Design>
 <Wire x="0" c_src="C5" p_dest="P2" y="0" id="W2" c_dest="C3" p_src="P1"/>
 <Wire x="0" c_src="C4" p_dest="P1" y="0" id="W3" c_dest="C3" p_src="P1"/>
 <Wire x="0" c_src="C3" p_dest="P2" y="0" id="W4" c_dest="C7" p_src="P4"/>
 <Wire x="0" c_src="C4" p_dest="P1" y="0" id="W5" c_dest="C7" p_src="P1"/>
 <Wire x="0" c_src="C8" p_dest="P3" y="0" id="W6" c_dest="C3" p_src="P1"/>
 <Component x="1117" y="1026" id="C3" name="waveform_gen">
  <Rect width="200" x="0" y="0" height="170"/>
  <Properties>
   <Property value="73014444," type="internal" name="phase_inc"/>
   <Property value="SINE" type="internal" name="wave"/>
   <Property value="" type="external" name="clk"/>
   <Property value="" type="external" name="en"/>
   <Property value="" type="external" name="output"/>
   <Property value="" type="external" name="reset"/>
  </Properties>
  <Ports>
   <Port lrange="" x="-30" direction="IN" y="40" data_type="STD_LOGIC" rrange="" id="P1" name="clk"/>
   <Port lrange="" x="-30" direction="IN" y="80" data_type="STD_LOGIC" rrange="" id="P2" name="reset"/>
   <Port lrange="" x="-30" direction="IN" y="120" data_type="STD_LOGIC" rrange="" id="P3" name="en"/>
   <Port lrange="11" x="230" direction="OUT" y="40" data_type="STD_LOGIC_VECTOR" rrange="0" id="P4" name="
       output"/>
  </Ports>
 </Component>
 <Component x="728" y="828" id="C4" name="sim_clk">
  <Rect width="200" x="0" y="0" height="100"/>
  <Properties>
   <Property value="" type="external" name="clk"/>
  </Properties>
  <Ports>
   <Port lrange="" x="230" direction="OUT" y="40" data_type="STD_LOGIC" rrange="" id="P1" name="clk"/>
  </Ports>
 </Component>
 <Component x="733" y="1066" id="C5" name="sim_rst">
  <Rect width="200" x="0" y="0" height="100"/>
  <Properties>
   <Property value="" type="external" name="rst"/>
  </Properties>
  <Ports>
   <Port lrange="" x="230" direction="OUT" y="40" data_type="STD_LOGIC" rrange="" id="P1" name="rst"/>
  </Ports>
 </Component>
 <Component x="1423" y="1240" id="C7" name="sink_file">
  <Rect width="200" x="0" y="0" height="100"/>
  <Properties>
   <Property value="file_io.out" type="internal" name="stim_file"/>
   <Property value="12" type="internal" name="width"/>
   <Property value="" type="external" name="clk"/>
   <Property value="" type="external" name="data"/>
  </Properties>
  <Ports>
   <Port lrange="" x="-30" direction="IN" y="40" data_type="STD_LOGIC" rrange="" id="P1" name="clk"/>
   <Port lrange="(width-1)" x="-30" direction="IN" y="80" data_type="STD_LOGIC_VECTOR" rrange="0" id="P2" name
       ="data"/>
  </Ports>
 </Component>
 <Component x="734,7826086956525" y="1234,782608695652" id="C8" name="const">
  <Rect width="200" x="0" y="0" height="100"/>
  <Properties>
   <Property value="1" type="internal" name="DATA"/>
   <Property value="1" type="internal" name="DATA_WIDTH"/>
   <Property value="" type="external" name="dout"/>
  </Properties>
  <Ports>
   <Port lrange="(DATA_WIDTH-1)" x="230" direction="OUT" y="40" data_type="STD_LOGIC_VECTOR" rrange="0" id=
       "P1" name="dout"/>
  </Ports>
 </Component>
```

```
</Design>
```

Listing 9.6: Waveform Generator MHS: The MHS file generated by ROME to represent the Waveform Generator design.

```
PARAMETER VERSION = 2.1.0

##############################################################################
BEGIN waveform_gen
        PARAMETER INSTANCE = waveform_genC3
        PARAMETER HW_VER = 1.00.a
        PARAMETER phase_inc = 73014444
        PARAMETER wave = SINE
        PORT clk = W3
        PORT reset = W2
        PORT en = W6
        PORT output = W4
END

##############################################################################
BEGIN sim_clk
        PARAMETER INSTANCE = sim_clkC4
        PARAMETER HW_VER = 1.00.a
        PORT clk = W3 & W5
END

##############################################################################
BEGIN sim_rst
        PARAMETER INSTANCE = sim_rstC5
        PARAMETER HW_VER = 1.00.a
        PORT rst = W2
END

##############################################################################
BEGIN sink_file
        PARAMETER INSTANCE = sink_fileC7
        PARAMETER HW_VER = 1.00.a
        PARAMETER stim_file = file_io.out
        PARAMETER width = 12
        PORT clk = W5
        PORT data = W4
END

##############################################################################
BEGIN const
        PARAMETER INSTANCE = constC8
        PARAMETER HW_VER = 1.00.a
        PARAMETER DATA = 1
        PARAMETER DATA_WIDTH = 1
        PORT dout = W6
END
```

Listing 9.7: Waveform Generator Log: The output log as a result of running the Synthesis command from with in ROME.

```
Linux Detected
## Synthesize - XST ##


Release 14.1 - platgen Xilinx EDK 14.1 Build EDK_P.15xf
 (lin64)
Copyright (c) 1995-2012 Xilinx, Inc. All rights reserved.



Command Line: platgen -p xc6slx16csg324-3 -lang vhdl -intstyle default -lp
/home/shaun/Documents/Work/Masters/rome/build/../editor/pcores/../../ system.mhs



WARNING:EDK - INFO:Security:71 - If a license for part 'xc6slx16' is available,
   it will be possible to use 'XPS_TDP' instead of 'XPS'.
   WARNING:Security:42 - Your software subscription period has lapsed. Your
   current version of Xilinx tools will continue to function, but you no longer
   qualify for Xilinx software updates or new releases.
```

```
Parse /home/shaun/Desktop/waveform_gen/rome/system.mhs ...

Read MPD definitions ...


Overriding IP level properties ...


Computing clock values...

Performing IP level DRCs on properties...

Running DRC Tcl procedures for OPTION IPLEVEL_DRC_PROC...

Checking platform address map ...

Checking platform configuration ...

Checking port drivers...

Performing Clock DRCs...

Performing Reset DRCs...

Overriding system level properties...

Running system level update procedures...

Running UPDATE Tcl procedures for OPTION SYSLEVEL_UPDATE_PROC...

Running system level DRCs...

Performing System level DRCs on properties...

Running DRC Tcl procedures for OPTION SYSLEVEL_DRC_PROC...


Running UPDATE Tcl procedures for OPTION PLATGEN_SYSLEVEL_UPDATE_PROC...

Modify defaults ...

Creating stub ...

WARNING:EDK - (GLOBAL) - atleast 1 toplevel output/input needs to be defined!

Processing licensed instances ...
Completion time: 0.00 seconds

Creating hardware output directories ...

Managing hardware (BBD-specified) netlist files ...

Managing cache ...

Elaborating instances ...

Writing HDL for elaborated instances ...

Inserting wrapper level ...

Completion time: 0.00 seconds

Constructing platform-level connectivity ...
WARNING:EDK - (GLOBAL) - atleast 1 toplevel output/input needs to be defined!
Completion time: 0.00 seconds

Writing (top-level) BMM ...

Writing (top-level and wrappers) HDL ...


Generating synthesis project file ...


Running XST synthesis ...

INFO:EDK - The following instances are synthesized with XST. The MPD option
   IMP_NETLIST=TRUE indicates that a NGC file is to be produced using XST
   synthesis. IMP_NETLIST=FALSE (default) instances are not synthesized.
INSTANCE:waveform_genc3 - /home/shaun/Desktop/waveform_gen/rome/system.mhs line
4 - Running XST synthesis
```

```
INSTANCE:sim_clkc4 - /home/shaun/Desktop/waveform_gen/rome/system.mhs line 16 -
Running XST synthesis

INSTANCE:sim_rstc5 - /home/shaun/Desktop/waveform_gen/rome/system.mhs line 23 -
Running XST synthesis

INSTANCE:sink_filec7 - /home/shaun/Desktop/waveform_gen/rome/system.mhs line 30
- Running XST synthesis

writing filewriting file


Running NGCBUILD ...

INFO:EDK - NCF files should not be modified as they will be regenerated.
    If any constraint needs to be overridden, this should be done by modifying
    the data/system.ucf file.


Rebuilding cache ...

Total run time: 53.00 seconds

xst -ifn system_xst.scr -intstyle silent
Running XST synthesis ...

XST completed
```

9.4 Appendix: Wishbone Adder

Listing 9.8: Wishbone Bus VHDL: a VHDL implementation of a Wishbone bus.

```
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;
library UNISIM;
use UNISIM.Vcomponents.ALL;

entity wb is
   port (
         RST_I,CLK_I : in std_logic;

         -- WISHBONE MASTER INTERFACE
         M_WB_ADR : in std_logic_vector (15 downto 0);
         M_WB_DAT_O : in std_logic_vector (15 downto 0);
         M_WB_WE: in std_logic;
         M_WB_STB : in std_logic;
         M_WB_CYC : in std_logic;
         M_WB_DAT_I : out std_logic_vector (15 downto 0);
         M_WB_ACK : out std_logic;

         -- WISHBONE SLAVE INTERFACE
         S_WB_ADR : out std_logic_vector (15 downto 0);
         S_WB_DAT_I : out std_logic_vector (15 downto 0);
         S_WB_WE: out std_logic;
         S_WB_STB : out std_logic;
         S_WB_CYC : out std_logic;
         S_WB_DAT_O : in std_logic_vector (15 downto 0);
         S_WB_ACK : in std_logic
    );
end wb;

architecture BEHAVIORAL of wb is
begin
     S_WB_ADR <= M_WB_ADR;
     S_WB_DAT_I <= M_WB_DAT_O;
     S_WB_WE <= M_WB_WE;
     S_WB_STB <= M_WB_STB;
     S_WB_CYC <= M_WB_CYC;

     M_WB_DAT_I <= S_WB_DAT_O;
     M_WB_ACK <= S_WB_ACK;

end BEHAVIORAL;

-- synopsys translate_off
configuration CFG_wb of wb is
   for BEHAVIORAL
   end for;
end wb;
-- synopsys translate_on
```

Listing 9.9: Wishbone Bus MPD: a MPD file generated as a result of importing the bus to ROME. The generated file has been adapted to include bus abstraction functionality.

```
##############################################################################
##
## Name : wb
## Desc : Microprocessor Peripheral Description
## : Automatically generated by PsfUtility
##
##############################################################################

BEGIN wb

## Peripheral Options
OPTION IPTYPE = BUS
OPTION BUS_STD = WB
OPTION IMP_NETLIST = TRUE
OPTION STYLE = HDL
OPTION HDL = VHDL
OPTION IP_GROUP = USER
OPTION MAX_MASTERS = 16
```

```
OPTION MAX_SLAVES = 16


## Bus Interfaces

## Generics for VHDL or Parameters for Verilog

## Ports
PORT RST = RST, DIR = I
PORT CLK = CLK, DIR = I

### WISHBONE MASTER INTERFACE
PORT M_WB_ADR = M_WB_ADR, DIR = I, VEC = [15:0]
PORT M_WB_DAT_O = M_WB_DAT_O, DIR = I, VEC = [15:0]
PORT M_WB_WE = M_WB_WE, DIR = I
PORT M_WB_STB = M_WB_STB, DIR = I
PORT M_WB_CYC = M_WB_CYC, DIR = I
PORT M_WB_DAT_I = M_WB_DAT_I, DIR = O, VEC = [15:0]
PORT M_WB_ACK = M_WB_ACK, DIR = O

### WISHBONE SLAVE INTERFACE
PORT S_WB_ADR = S_WB_ADR, DIR = O, VEC = [15:0]
PORT S_WB_DAT_I = S_WB_DAT_I, DIR = O, VEC = [15:0]
PORT S_WB_WE = S_WB_WE, DIR = O
PORT S_WB_STB = S_WB_STB, DIR = O
PORT S_WB_CYC = S_WB_CYC, DIR = O
PORT S_WB_DAT_O = S_WB_DAT_O, DIR = I, VEC = [15:0]
PORT S_WB_ACK = S_WB_ACK, DIR = I

END
```

Listing 9.10: GPMC Wishbone Bridge VHDL: a VHDL implementation of a GPMC to Wishbone bridge.

```
----------------------------------------------------------------
-- Company:   University of Cape Town
-- Engineer:   Shaun Katz
--
-- Create Date: 16:51:41 06/22/2012
-- Module Name: gpmc_wb_bridge - Behavioral
--
--   <----->|gpmc BRIDGE wb|<----->
----------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library UNISIM;
use UNISIM.VComponents.all;

entity gpmc_wb_bridge is
        port (
                -- GPMC INTERFACE
                gpmc_a : in std_logic_vector(10 downto 1);
                gpmc_d : inout std_logic_vector(15 downto 0);
                gpmc_clk_i : in std_logic;
                gpmc_n_cs : in std_logic_vector(6 downto 0);
                gpmc_n_we : in std_logic;
                gpmc_n_oe : in std_logic;
                gpmc_n_adv_ale : in std_logic;
                gpmc_n_wp : in std_logic;
                gpmc_busy_0 : out std_logic;
                gpmc_busy_1 : out std_logic;

                -- WISHBONE MASTER INTERFACE
                RST_I,CLK_I : in std_logic;
                ADR_O : out std_logic_vector (15 downto 0);
                DAT_O : out std_logic_vector (15 downto 0);
                WE_O,STB_O,CYC_O : out std_logic;
                DAT_I : in std_logic_vector (15 downto 0);
                ACK_I : in std_logic
        );
end gpmc_wb_bridge;

architecture Behavioral of gpmc_wb_bridge is
    -- Define signals for the gpmc bus
    signal gpmc_clk_i_b : std_logic; --buffered gpmc_clk_i
    signal bank_addr : STD_LOGIC_VECTOR (3 downto 0) := (others => '0');
    signal addr : STD_LOGIC_VECTOR (15 downto 0) := (others => '0');
    signal we : std_logic;
```

```
    type arr is array(1 downto 0) of std_logic_vector(15 downto 0);
    signal sync_gpmc_d_DAT_O : arr;
    signal sync_DAT_I_gpmc_d : arr;

begin
        -- Buffer GPMC Bus Clock
        IBUFG_gpmc_clk_i : IBUFG
        generic map(IBUF_LOW_PWR => FALSE, IOSTANDARD => "DEFAULT")
        port map(I => gpmc_clk_i,O => gpmc_clk_i_b);

        addr(0) <= '0'; -- Due to Granularity

        -- Debug Wiring
        gpmc_busy_0 <= '0';
        gpmc_busy_1 <= '0';

        process (gpmc_clk_i_b,gpmc_n_cs,gpmc_n_oe,gpmc_n_we,gpmc_n_adv_ale,gpmc_d,gpmc_a)
        begin
                if (rising_edge(gpmc_clk_i_b)) then -- gpmc_clk_i_b'event and gpmc_clk_i_b = '1' then
                        sync_DAT_I_gpmc_d(0) <= DAT_I;
                        sync_DAT_I_gpmc_d(1) <= sync_DAT_I_gpmc_d(0);
                        --First cycle of the bus transaction record the address
                        if (gpmc_n_adv_ale = '0') then
                                bank_addr <= gpmc_a(3 downto 1) & gpmc_d(15); -- Set Address of Memory Bank
                                addr(15 downto 1) <= gpmc_d(14 downto 0); -- Set Address of Word
                        --Bus Write
                        elsif (gpmc_n_we = '0') then
                                we <= '1';
                        --Bus Read
                        elsif gpmc_n_oe = '0' then
                                we <= '0';
                        end if;
                end if;
        end process;

        process (ACK_I, we, CLK_I, addr)
        begin
                if (rising_edge(CLK_I)) then
                        sync_gpmc_d_DAT_O(0) <= gpmc_d;
                        sync_gpmc_d_DAT_O(1) <= sync_gpmc_d_DAT_O(0);
                        if (we = '1') then --Bus Write
                                CYC_O <= '1';
                                ADR_O <= addr;
                                WE_O <= '1';
                                --DAT_O <= data;
                                STB_O <= '1';
                                if (ACK_I = '1') then
                                        CYC_O <= '0';
                                        STB_O <= '0';
                                end if;
                        elsif (we = '0') then --Bus Read
                                CYC_O <= '1';
                                ADR_O <= addr;
                                WE_O <= '0';
                                STB_O <= '1';
                                if (ACK_I = '1') then
                                        --data <= DAT_I;
                                        CYC_O <= '0';
                                        STB_O <= '0';
                                end if;
                        else
                                CYC_O <= '0';
                                STB_O <= '0';
                        end if;
                end if;
        end process;

        -- Manage the tri-state bus
        gpmc_d <= sync_DAT_I_gpmc_d(1) when (gpmc_n_oe = '0') else (others => 'Z');
        DAT_O <= sync_gpmc_d_DAT_O(1);
end Behavioral;
```

Listing 9.11: GPMC Wishbone Bridge MPD: a MPD file generated as a result of importing the bridge to ROME. The generated file has been adapted to include bus abstraction functionality.

```
##################################################################
```

```
##
## Name : gpmc_wb_bridge
## Desc : Microprocessor Peripheral Description
## : Automatically generated by PsfUtility
##
##############################################################################

BEGIN gpmc_wb_bridge

## Peripheral Options
OPTION IPTYPE = PERIPHERAL
OPTION IMP_NETLIST = TRUE
OPTION HDL = MIXED
OPTION IP_GROUP = USER


## Bus Interfaces
BUS_INTERFACE BUS=MWB, BUS_STD=WB, BUS_TYPE=MASTER

## Generics for VHDL or Parameters for Verilog

## Ports
PORT gpmc_a = "", DIR = I, VEC = [10:1]
PORT gpmc_d = "", DIR = IO, VEC = [15:0]
PORT gpmc_clk_i = "", DIR = I
PORT gpmc_n_cs = "", DIR = I, VEC = [6:0]
PORT gpmc_n_we = "", DIR = I
PORT gpmc_n_oe = "", DIR = I
PORT gpmc_n_adv_ale = "", DIR = I
PORT gpmc_n_wp = "", DIR = I
PORT gpmc_busy_0 = "", DIR = O
PORT gpmc_busy_1 = "", DIR = O
PORT RST_I = "", DIR = I
PORT CLK_I = "", DIR = I
PORT ADR_O = "M_WB_ADR", DIR = O, VEC = [15:0], BUS = MWB
PORT DAT_O = "M_WB_DAT_O", DIR = O, VEC = [15:0], BUS = MWB
PORT WE_O = "M_WB_WE", DIR = O, BUS = MWB
PORT STB_O = "M_WB_STB", DIR = O, BUS = MWB
PORT CYC_O = "M_WB_CYC", DIR = O, BUS = MWB
PORT DAT_I = "M_WB_DAT_I", DIR = I, VEC = [15:0], BUS = MWB
PORT ACK_I = "M_WB_ACK", DIR = I, BUS = MWB

END
```

Listing 9.12: Wishbone Register VHDL: a VHDL implementation of a Wishbone register.

```
----------------------------------------------------------------
-- Company:   University of Cape Town
-- Engineer:   Shaun Katz
--
-- Create Date: 11:20:32 06/22/2012
-- Module Name: wb_register - Behavioral
--
--   <----->|wb REGISTER output|----->
----------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity wb_register is
        generic ( C_BASEADDR : std_logic_vector (15 downto 0) := x"0000" ;
                        C_HIGHADDR : std_logic_vector (15 downto 0) := x"0000" ));
        port (
                -- WISHBONE SLAVE INTERFACE
                RST_I,CLK_I : in std_logic;
                ADR_I : in std_logic_vector (15 downto 0);
                DAT_I : in std_logic_vector (15 downto 0);
                WE_I : in std_logic;
                STB_I : in std_logic;
                CYC_I : in std_logic;
                DAT_O : out std_logic_vector (15 downto 0);
                ACK_O : out std_logic;

                -- NON-WISHBONE PORTS
                PRT_O : out std_logic_vector (15 downto 0));
end wb_register;

architecture my_reg of wb_register is
        signal data : std_logic_vector(15 downto 0) := (others => '0');
begin
        process (RST_I,STB_I,CYC_I,CLK_I)
```

```
        begin
                if (rising_edge(CLK_I)) then
                        if (RST_I = '1') then
                                data <= (others => '0');
                                DAT_O <= (others => '0');
                                ACK_O <= '0';
                        elsif (ADR_I = C_BASEADDR) then
                                if ((STB_I and CYC_I) = '1') then
                                        if (WE_I = '0') then -- READING CYCLE
                                                DAT_O <= data;
                                                ACK_O <= '1';
                                        elsif (WE_I = '1') then -- WRITING CYCLE
                                                data <= DAT_I;
                                                ACK_O <= '1';
                                        end if;
                                elsif ((STB_I and CYC_I) = '0') then
                                        ACK_O <= '0';
                                end if;
                        end if;
                end if;
        end process;

        PRT_O <= data;
end my_reg;
```

Listing 9.13: Wishbone Register MPD: a MPD file generated as a result of importing the register to ROME. The generated file has been adapted to include bus abstraction functionality.

```
##############################################################################
##
## Name : wb_register
## Desc : Microprocessor Peripheral Description
## : Automatically generated by PsfUtility
##
##############################################################################

BEGIN wb_register

## Peripheral Options
OPTION IPTYPE = PERIPHERAL
OPTION IMP_NETLIST = TRUE
OPTION HDL = MIXED
OPTION IP_GROUP = USER


## Bus Interfaces
BUS_INTERFACE BUS=SWB, BUS_STD=WB, BUS_TYPE=SLAVE

## Generics for VHDL or Parameters for Verilog
PARAMETER C_BASEADDR = 0x0000, DT = std_logic_vector(15 downto 0)
PARAMETER C_HIGHADDR = 0x0000, DT = std_logic_vector(15 downto 0)

## Ports
PORT RST_I = "", DIR = I
PORT CLK_I = "", DIR = I
PORT ADR_I = S_WB_ADR, DIR = I, VEC = [15:0], BUS = SWB
PORT DAT_I = S_WB_DAT_I, DIR = I, VEC = [15:0], BUS = SWB
PORT WE_I = S_WB_WE, DIR = I, BUS = SWB
PORT STB_I = S_WB_STB, DIR = I, BUS = SWB
PORT CYC_I = S_WB_CYC, DIR = I, BUS = SWB
PORT DAT_O = S_WB_DAT_O, DIR = O, VEC = [15:0], BUS = SWB
PORT ACK_O = S_WB_ACK, DIR = O, BUS = SWB
PORT PRT_O = "", DIR = O, VEC = [15:0]

END
```

Listing 9.14: Adder: a VHDL implementation of a Adder/Subtracter.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY addsub IS
```

```
        GENERIC(
                A_DATA_WIDTH : natural := 32;
                B_DATA_WIDTH : natural := 32;
                RESULT_DATA_WIDTH : natural := 32
        );
        PORT
        (
                a: IN std_logic_vector(A_DATA_WIDTH-1 downto 0);
                b: IN std_logic_vector(B_DATA_WIDTH-1 downto 0);
                addnsub: IN STD_LOGIC;
                result: OUT std_logic_vector(RESULT_DATA_WIDTH-1 downto 0)
        );
END addsub;

ARCHITECTURE rtl OF addsub IS
signal res : signed(RESULT_DATA_WIDTH-1 downto 0);
BEGIN
        PROCESS (a, b, addnsub)
        BEGIN
                IF (addnsub = '1') THEN
                        res <= signed(a) + signed(b);
                ELSE
                        res <= signed(a) - signed(b);
                END IF;
        END PROCESS;
result <= std_logic_vector(res);
END rtl;
```

Listing 9.15: Wishbone Adder MHS: The MHS file generated by ROME to represent the Wishbone adder design.

```
 PARAMETER VERSION = 2.1.0

##############################################################################
BEGIN wb
 PARAMETER INSTANCE = wb_0
 PARAMETER HW_VER = 1.00.a
END

##############################################################################
BEGIN wb_register
 PARAMETER INSTANCE = wb_register_C3
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0x0000
 BUS_INTERFACE SWB = wb_0
 PORT PRT_O = W6
END

##############################################################################
BEGIN gpmc_wb_bridge
 PARAMETER INSTANCE = gpmc_wb_bridge_C1
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE MWB = wb_0
END

##############################################################################
BEGIN addsub
 PARAMETER INSTANCE = addsubC4
 PARAMETER HW_VER = 1.00.a
 PARAMETER A_DATA_WIDTH = 16
 PARAMETER B_DATA_WIDTH = 16
 PARAMETER RESULT_DATA_WIDTH = 16
 PORT a = W4
 PORT b = W6
END

##############################################################################
BEGIN wb_register
 PARAMETER INSTANCE = wb_register_C2
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0x0002
 BUS_INTERFACE SWB = wb_0
 PORT PRT_O = W4
END
```

9.5 Appendix: FM Receiver

Listing 9.16: FM Receiver XML: The XML file generated as a result of saving the FM receiver design in the ROME environment.

```
<!DOCTYPE MyML>
<Design>
 <Wire x="0" c_src="C12" p_dest="P1" y="0" id="W18" c_dest="C13" p_src="P1"/>
 <Wire x="0" c_src="C17" p_dest="P2" y="0" id="W19" c_dest="C13" p_src="P1"/>
 <Wire x="0" c_src="C12" p_dest="P1" y="0" id="W20" c_dest="C18" p_src="P1"/>
 <Wire x="0" c_src="C17" p_dest="P2" y="0" id="W21" c_dest="C18" p_src="P1"/>
 <Wire x="0" c_src="C13" p_dest="P4" y="0" id="W24" c_dest="C18" p_src="P4"/>
 <Wire x="0" c_src="C18" p_dest="P3" y="0" id="W25" c_dest="C20" p_src="P5"/>
 <Wire x="0" c_src="C17" p_dest="P2" y="0" id="W26" c_dest="C20" p_src="P1"/>
 <Wire x="0" c_src="C20" p_dest="P1" y="0" id="W27" c_dest="C12" p_src="P1"/>
 <Wire x="0" c_src="C20" p_dest="P3" y="0" id="W28" c_dest="C21" p_src="P4"/>
 <Wire x="0" c_src="C20" p_dest="P3" y="0" id="W29" c_dest="C13" p_src="P5"/>
 <Wire x="0" c_src="C12" p_dest="P1" y="0" id="W30" c_dest="C21" p_src="P1"/>
 <Wire x="0" c_src="C17" p_dest="P2" y="0" id="W31" c_dest="C21" p_src="P1"/>
 <Wire x="0" c_src="C21" p_dest="P2" y="0" id="W32" c_dest="C22" p_src="P4"/>
 <Wire x="0" c_src="C12" p_dest="P1" y="0" id="W33" c_dest="C22" p_src="P1"/>
 <Wire x="0" c_src="C23" p_dest="P3" y="0" id="W34" c_dest="C18" p_src="P2"/>
 <Wire x="0" c_src="C23" p_dest="P1" y="0" id="W35" c_dest="C12" p_src="P1"/>
 <Component x="198,3246612548828" y="427,9637756347656" id="C12" name="sim_clk">
  <Rect width="200" x="0" y="0" height="100"/>
  <Properties>
   <Property value="" type="external" name="clk"/>
  </Properties>
  <Ports>
   <Port lrange="" x="230" direction="OUT" y="40" data_type="STD_LOGIC" rrange="" id="P1" name="clk"/>
  </Ports>
 </Component>
 <Component x="253,0474700927734" y="665,3237915039062" id="C13" name="nco_std">
  <Rect width="200" x="0" y="0" height="170"/>
  <Properties>
   <Property value="" type="external" name="clk"/>
   <Property value="" type="external" name="din"/>
   <Property value="" type="external" name="dout"/>
   <Property value="" type="external" name="reset"/>
  </Properties>
  <Ports>
   <Port lrange="" x="-30" direction="IN" y="40" data_type="STD_LOGIC" rrange="" id="P1" name="clk"/>
   <Port lrange="" x="-30" direction="IN" y="80" data_type="STD_LOGIC" rrange="" id="P2" name="reset"/>
   <Port lrange="11" x="-30" direction="IN" y="120" data_type="STD_LOGIC_VECTOR" rrange="0" id="P3" name="din"
       />
   <Port lrange="7" x="230" direction="OUT" y="40" data_type="STD_LOGIC_VECTOR" rrange="0" id="P4" name="dout"
       />
  </Ports>
 </Component>
 <Component x="226,6026763916016" y="1132,129638671875" id="C17" name="sim_rst">
  <Rect width="200" x="0" y="0" height="100"/>
  <Properties>
   <Property value="" type="external" name="rst"/>
  </Properties>
  <Ports>
   <Port lrange="" x="230" direction="OUT" y="40" data_type="STD_LOGIC" rrange="" id="P1" name="rst"/>
  </Ports>
 </Component>
 <Component x="737,5941162109375" y="556,0569458007812" id="C18" name="multiplier_std">
  <Rect width="212" x="0" y="0" height="170"/>
  <Properties>
   <Property value="" type="external" name="CLK"/>
   <Property value="" type="external" name="RESET"/>
   <Property value="" type="external" name="input1"/>
   <Property value="" type="external" name="input2"/>
   <Property value="" type="external" name="output"/>
  </Properties>
  <Ports>
   <Port lrange="" x="-30" direction="IN" y="40" data_type="STD_LOGIC" rrange="" id="P1" name="CLK"/>
   <Port lrange="" x="-30" direction="IN" y="80" data_type="STD_LOGIC" rrange="" id="P2" name="RESET"/>
   <Port lrange="7" x="-30" direction="IN" y="120" data_type="STD_LOGIC_VECTOR" rrange="0" id="P3" name="input1
       "/>
   <Port lrange="7" x="-30" direction="IN" y="160" data_type="STD_LOGIC_VECTOR" rrange="0" id="P4" name="input2
       "/>
   <Port lrange="7" x="242" direction="OUT" y="40" data_type="STD_LOGIC_VECTOR" rrange="0" id="P5" name="
       output"/>
  </Ports>
 </Component>
 <Component x="1100,623291015625" y="987,1028442382812" id="C20" name="loop_filter_std">
  <Rect width="212" x="0" y="0" height="170"/>
```

```
  <Properties>
   <Property value="" type="external" name="C"/>
   <Property value="" type="external" name="CLK"/>
   <Property value="" type="external" name="D1"/>
   <Property value="" type="external" name="D2"/>
   <Property value="" type="external" name="RESET"/>
  </Properties>
  <Ports>
   <Port lrange="" x="-30" direction="IN" y="40" data_type="STD_LOGIC" rrange="" id="P1" name="CLK"/>
   <Port lrange="" x="-30" direction="IN" y="80" data_type="STD_LOGIC" rrange="" id="P2" name="RESET"/>
   <Port lrange="7" x="-30" direction="IN" y="120" data_type="STD_LOGIC_VECTOR" rrange="0" id="P3" name="C"/>
   <Port lrange="11" x="242" direction="OUT" y="40" data_type="STD_LOGIC_VECTOR" rrange="0" id="P4" name="D1"
       />
   <Port lrange="11" x="242" direction="OUT" y="80" data_type="STD_LOGIC_VECTOR" rrange="0" id="P5" name="D2"
       />
  </Ports>
 </Component>
 <Component x="1110,208618164062" y="571,153564453125" id="C21" name="fir_std">
  <Rect width="231,5" x="0" y="0" height="170"/>
  <Properties>
   <Property value="" type="external" name="clock"/>
   <Property value="" type="external" name="data_in"/>
   <Property value="" type="external" name="data_out"/>
   <Property value="" type="external" name="reset"/>
  </Properties>
  <Ports>
   <Port lrange="" x="-30" direction="IN" y="40" data_type="STD_LOGIC" rrange="" id="P1" name="clock"/>
   <Port lrange="" x="-30" direction="IN" y="80" data_type="STD_LOGIC" rrange="" id="P2" name="reset"/>
   <Port lrange="11" x="-30" direction="IN" y="120" data_type="STD_LOGIC_VECTOR" rrange="0" id="P3" name="
       data_in"/>
   <Port lrange="11" x="261,5" direction="OUT" y="40" data_type="STD_LOGIC_VECTOR" rrange="0" id="P4" name="
       data_out"/>
  </Ports>
 </Component>
 <Component x="1435,67578125" y="758,22119140625" id="C22" name="sink_file">
  <Rect width="200" x="0" y="0" height="100"/>
  <Properties>
   <Property value="file_io.out" type="internal" name="stim_file"/>
   <Property value="12" type="internal" name="width"/>
   <Property value="" type="external" name="clk"/>
   <Property value="" type="external" name="data"/>
  </Properties>
  <Ports>
   <Port lrange="" x="-30" direction="IN" y="40" data_type="STD_LOGIC" rrange="" id="P1" name="clk"/>
   <Port lrange="(width-1)" x="-30" direction="IN" y="80" data_type="STD_LOGIC_VECTOR" rrange="0" id="P2" name
       ="data"/>
  </Ports>
 </Component>
 <Component x="127,7535018920898" y="976,4017333984375" id="C23" name="src_file">
  <Rect width="200" x="0" y="0" height="100"/>
  <Properties>
   <Property value="file_io.in" type="internal" name="stim_file"/>
   <Property value="8" type="internal" name="width"/>
   <Property value="" type="external" name="clk"/>
   <Property value="" type="external" name="data"/>
  </Properties>
  <Ports>
   <Port lrange="" x="-30" direction="IN" y="40" data_type="STD_LOGIC" rrange="" id="P1" name="clk"/>
   <Port lrange="(width-1)" x="230" direction="OUT" y="40" data_type="STD_LOGIC_VECTOR" rrange="0" id="P2"
       name="data"/>
  </Ports>
 </Component>
</Design>
```

Listing 9.17: FM Receiver MHS: The MHS file generated by ROME to represent the FM receiver design.

```
PARAMETER VERSION = 2.1.0

##############################################################################
BEGIN sim_clk
        PARAMETER INSTANCE = sim_clkC12
        PARAMETER HW_VER = 1.00.a
        PORT clk = W18 & W20 & W27 & W30 & W33 & W35
END

##############################################################################
BEGIN nco_std
        PARAMETER INSTANCE = nco_stdC13
```

```
        PARAMETER HW_VER = 1.00.a
        PORT clk = W18
        PORT reset = W19
        PORT din = W29
        PORT dout = W24
END

##############################################################################
BEGIN sim_rst
        PARAMETER INSTANCE = sim_rstC17
        PARAMETER HW_VER = 1.00.a
        PORT rst = W19 & W21 & W26 & W31
END

##############################################################################
BEGIN multiplier_std
        PARAMETER INSTANCE = multiplier_stdC18
        PARAMETER HW_VER = 1.00.a
        PORT CLK = W20
        PORT RESET = W21
        PORT input1 = W34
        PORT input2 = W24
        PORT output = W25
END

##############################################################################
BEGIN loop_filter_std
        PARAMETER INSTANCE = loop_filter_stdC20
        PARAMETER HW_VER = 1.00.a
        PORT CLK = W27
        PORT RESET = W26
        PORT C = W25
        PORT D1 = W28
        PORT D2 = W29
END

##############################################################################
BEGIN fir_std
        PARAMETER INSTANCE = fir_stdC21
        PARAMETER HW_VER = 1.00.a
        PORT clock = W30
        PORT reset = W31
        PORT data_in = W28
        PORT data_out = W32
END

##############################################################################
BEGIN sink_file
        PARAMETER INSTANCE = sink_fileC22
        PARAMETER HW_VER = 1.00.a
        PARAMETER stim_file = file_io.out
        PARAMETER width = 12
        PORT clk = W33
        PORT data = W32
END

##############################################################################
BEGIN src_file
        PARAMETER INSTANCE = src_fileC23
        PARAMETER HW_VER = 1.00.a
        PARAMETER stim_file = file_io.in
        PARAMETER width = 8
        PORT clk = W35
        PORT data = W34
END
```

Listing 9.18: FM Receiver Log: The output log as a result of running the Synthesis command from with in ROME.

```
Linux Detected
## Synthesize - XST ##



Release 14.1 - platgen Xilinx EDK 14.1 Build EDK_P.15xf
 (lin64)
Copyright (c) 1995-2012 Xilinx, Inc. All rights reserved.
```

```
Command Line: platgen -p xc6slx16csg324-3 -lang vhdl -intstyle default -lp
/home/shaun/Documents/Work/Masters/rome/build/../editor/pcores/../../ system.mhs



WARNING:EDK - INFO:Security:71 - If a license for part 'xc6slx16' is available,
   it will be possible to use 'XPS_TDP' instead of 'XPS'.
   WARNING:Security:42 - Your software subscription period has lapsed. Your
   current version of Xilinx tools will continue to function, but you no longer
   qualify for Xilinx software updates or new releases.


Parse /home/shaun/Desktop/fm_receiver/system.mhs ...

Read MPD definitions ...


Overriding IP level properties ...


Computing clock values...

Performing IP level DRCs on properties...

Running DRC Tcl procedures for OPTION IPLEVEL_DRC_PROC...

Checking platform address map ...

Checking platform configuration ...

Checking port drivers...

Performing Clock DRCs...

Performing Reset DRCs...

Overriding system level properties...

Running system level update procedures...

Running UPDATE Tcl procedures for OPTION SYSLEVEL_UPDATE_PROC...

Running system level DRCs...

Performing System level DRCs on properties...

Running DRC Tcl procedures for OPTION SYSLEVEL_DRC_PROC...


Running UPDATE Tcl procedures for OPTION PLATGEN_SYSLEVEL_UPDATE_PROC...

Modify defaults ...

Creating stub ...

WARNING:EDK - (GLOBAL) - atleast 1 toplevel output/input needs to be defined!

Processing licensed instances ...
Completion time: 0.00 seconds

Creating hardware output directories ...

Managing hardware (BBD-specified) netlist files ...

Managing cache ...


Elaborating instances ...


Writing HDL for elaborated instances ...


Inserting wrapper level ...
Completion time: 0.00 seconds

Constructing platform-level connectivity ...

WARNING:EDK - (GLOBAL) - atleast 1 toplevel output/input needs to be defined!
Completion time: 0.00 seconds

Writing (top-level) BMM ...
```

```
Writing (top−level and wrappers) HDL ...

Generating synthesis project file ...


Running XST synthesis ...


INFO:EDK − The following instances are synthesized with XST. The MPD option
    IMP_NETLIST=TRUE indicates that a NGC file is to be produced using XST
    synthesis. IMP_NETLIST=FALSE (default) instances are not synthesized.
INSTANCE:sim_clkc12 − /home/shaun/Desktop/fm_receiver/system.mhs line 4 −
Running XST synthesis

INSTANCE:nco_stdc13 − /home/shaun/Desktop/fm_receiver/system.mhs line 11 −
Running XST synthesis

INSTANCE:sim_rstc17 − /home/shaun/Desktop/fm_receiver/system.mhs line 21 −
Running XST synthesis

INSTANCE:multiplier_stdc18 − /home/shaun/Desktop/fm_receiver/system.mhs line 28
− Running XST synthesis

INSTANCE:loop_filter_stdc20 − /home/shaun/Desktop/fm_receiver/system.mhs line 39
− Running XST synthesis

INSTANCE:fir_stdc21 − /home/shaun/Desktop/fm_receiver/system.mhs line 50 −
Running XST synthesis

INSTANCE:sink_filec22 − /home/shaun/Desktop/fm_receiver/system.mhs line 60 −
Running XST synthesis

writing filewriting file

INSTANCE:src_filec23 − /home/shaun/Desktop/fm_receiver/system.mhs line 70 −
Running XST synthesis


Running NGCBUILD ...

INFO:EDK − NCF files should not be modified as they will be regenerated.
    If any constraint needs to be overridden, this should be done by modifying
    the data/system.ucf file.

Rebuilding cache ...


Total run time: 150.00 seconds

xst −ifn system_xst.scr −intstyle silent
Running XST synthesis ...

XST completed
```