COMPARATIVE STUDY OF TOOL-FLOWS FOR RAPID PROTOTYPING OF SOFTWARE-DEFINED RADIO DIGITAL SIGNAL PROCESSING

A dissertation submitted to the Department of Electrical Engineering,
UNIVERSITY OF CAPE TOWN, in fulfilment of the requirements for the degree of

Master of Science

at the

University of Cape Town

by

KHOBATHA SETETEMELA

Supervised by :
DR SIMON WINBERG

February 8, 2019

Declaration

I know the meaning of plagiarism and declare that all the work in this dissertation, save for that which is properly acknowledged and referenced, is my own. It is being submitted for the degree of **Master of Science** in Electrical Engineering at the University of Cape Town. This work has not been submitted before for any other degree or examination in any other university.

Signature of Author: Signed by candidate
University of Cape Town

Cape Town
February 8, 2019

ABSTRACT

This dissertation is a comparative study of tool-flows for rapid prototyping of SDR DSP operations on programmable hardware platforms. The study is divided into two parts, focusing on high-level tool-flows for implementing SDR DSP operations on FPGA and GPU platforms respectively. In this dissertation, the term 'tool-flow' refers to a tool or a chain of tools that facilitate the mapping of an application description specified in a programming language into one or more programmable hardware platforms. High-level tool-flows use different techniques, such as high-level synthesis to allow the designer to specify the application from a high level of abstraction and achieve improved productivity without significant degradation in the design's performance. SDR is an emerging communications technology that is driven by - among other factors – increasing demands for high-speed, interoperable and versatile communications systems. The key idea in SDR is the need to implement as many as possible of the radio functions that were traditionally defined in fixed hardware, in software on programmable hardware processors instead. The most commonly used processors are based on complex parallel computing architectures in order to support the high-speed processing demands of SDR applications, and they include FPGAs, GPUs and multicore general-purpose processors (GPPs) and DSPs. The architectural complexity of these processors results in a corresponding increase in programming methodologies which however impedes their wider adoption in suitable applications domains, including SDR DSP. In an effort to address this, a plethora of different high-level tool-flows have been developed. Several comparative studies of these tool-flows have been done to help – among other benefits – designers in choosing high-level tools to use. However, there are few studies that focus on SDR DSP operations, and most existing comparative studies are not based on well-defined comparison criteria.

The approach implemented in this dissertation is to use a system engineering design process, firstly, to define the qualitative comparison criteria in the form of a specification for an ideal high-level SDR DSP tool-flow and, secondly, to implement a FIR filter case study in each of the tool-flows to enable a quantitative comparison in terms of programming effort and performance. The study considers Migen- and MyHDL-based open-source tool-flows for FPGA targets, and CUDA and Open Computing Language (OpenCL) for GPU targets. The ideal high-level SDR DSP tool-flow specification was defined and used to conduct a comparative study of the tools across three main design categories, which included high-level modelling, verification and implementation. For tool-flows targeting GPU platforms, the FIR case study was implemented using each of the tools; it was compiled, executed on a GPU server consisting of 2 GTX Titan-X GPUs and an Intel Core i7 GPP, and lastly profiled. The tools were moreover compared in terms of programming effort, memory transfers cost and overall operation time. With regard to tool-flows with FPGA targets, the FIR case study was developed by using each tool, and then implemented on a Xilinx 7 FPGA and compared in terms of programming effort, logic utilization and timing performance.

ACKNOWLEDGEMENTS

I would like acknowledge the **NRF and South African Radio Astronomy Observatory (SARAO)** for affording me the funding opportunity to complete my postgraduate studies.

To my supervisors **Dr Simon Winberg** and **Prof Michael Inggs**, who believed in me from the beginning of this dissertation and who never ceased to encourage, guide and kindly nudge me forward when I could not see the finish line, I am heartily indebted. This dissertation would not be possible without your emotional and technical support.

To my fellow students in the SDRG and RRSG research groups at UCT **Valerie, Shaun and Matthew**, for always willing to discuss ideas and lend a hand, thank you.

To my friends, **Sibonelo Madlopha, Dr Lerato Mohapi and Potjo Nthama** for your words of encouragement in due season which kept me going, thank you.

To my parents, **Make and Babe Dlamini and 'Me' Mpho**, who supported me emotionally, spiritually and financially throughout this work, I shall forever be indebted to you.

To my loving wife and best friend, **Lomaswati Setetemela**, thank you for the kindness, the loving care and the patience that you gave me. I cannot thank you enough.

Now unto Him who is able to keep me from falling, and to present me faultless before the presence of His glory with exceeding joy, To the only wise God my Saviour, be glory and majesty, dominion and power, both now and ever. Amen.

CONTENTS

Abstract i

Acknowledgements iii

Contents iv

List of Figures ix

List of Tables xi

List of Abbreviations xii

1 Introduction 1
- 1.1 Background . . . 1
  - 1.1.1 Software Defined Radios . . . 2
  - 1.1.2 Field Programmable Gate Arrays . . . 3
    - 1.1.2.1 FPGA Hardware . . . 3
    - 1.1.2.2 FPGA Configuration . . . 4
    - 1.1.2.3 HDL Design Methodology for FPGAs . . . 4
    - 1.1.2.4 High-Level Design Methodology for FPGAs . . . 5
  - 1.1.3 Graphics Processing Units . . . 6
    - 1.1.3.1 GPU Hardware . . . 6
    - 1.1.3.2 High-Level Design Methodology for GPUs . . . 7
- 1.2 Problem Statement . . . 7
- 1.3 Objectives . . . 8
- 1.4 Dissemination Strategy . . . 9
  - 1.4.1 Objectives and Approach . . . 9
  - 1.4.2 Target Audiences and Plan . . . 9
- 1.5 Dissertation Overview . . . 11

2 Literature Review 14
- 2.1 Software Defined Radio . . . 15
  - 2.1.1 Brief History of SDR Technology . . . 15
  - 2.1.2 SDR Hardware Architecture . . . 16
    - 2.1.2.1 The Radio Frequency Front-End Subsystem . . . 17
    - 2.1.2.2 Analog-to-Digital Converters (ADCs) . . . 18
    - 2.1.2.3 Digital-to-Analog Converters (DACs) . . . 19
    - 2.1.2.4 Programmable Digital Signal Processing . . . 19

2.1.3 SDR DSP Waveforms Functionality Characteristics . . . 21
2.1.3.1 Computational Complexity . . . 22
2.1.3.2 Parallelism . . . 22
2.1.3.3 Data Independency . . . 23
2.1.3.4 Arithmetic Requirements . . . 24
2.1.4 Programmable DSP Hardware for SDR . . . 24
2.1.4.1 GPP . . . 24
2.1.4.2 DSP Processor . . . 24
2.1.4.3 FPGA . . . 25
2.1.4.4 GPU . . . 26
2.2 Field Programmable Gate Array . . . 26
2.2.1 Programming Architecture . . . 26
2.2.1.1 SRAM-based Programming Technology . . . 26
2.2.1.2 Flash Programming Technology . . . 27
2.2.1.3 Antifuse Programming Technology . . . 27
2.2.2 Routing Architectures . . . 28
2.2.2.1 Island-Style Routing Architecture . . . 28
2.2.2.2 Hierarchical Routing Architecture . . . 28
2.2.3 FPGA-based SDR DSP Platforms . . . 29
2.2.3.1 RHINO . . . 29
2.2.3.2 ROACH . . . 30
2.2.3.3 Ettus USRP Platform . . . 31
2.2.4 High-Level Programming Tools . . . 32
2.2.4.1 MyHDL . . . 32
2.2.4.2 Migen . . . 33
2.3 Graphics Processing Units . . . 33
2.3.1 Hardware Architecture . . . 33
2.3.2 GPU-based SDR DSP Platforms . . . 35
2.3.2.1 GPU with GPP . . . 35
2.3.2.2 GPU with FPGA and GPP . . . 36
2.3.3 High-Level Programming Tools . . . 37
2.3.3.1 CUDA . . . 37
2.3.3.2 OpenCL . . . 37
2.4 High-Level Tool-Flows Techniques . . . 38
2.4.1 Modelling . . . 38
2.4.1.1 Model-based . . . 38
2.4.1.2 Algorithmic . . . 38
2.4.1.3 System Level . . . 38
2.4.2 Code generation . . . 38
2.4.2.1 High-Level Synthesis . . . 38
2.4.2.2 Electronic System-Level Synthesis . . . 40
2.5 High-Level Tool-Flows in SDR . . . 41
2.5.1 Existing Tools . . . 41
2.5.1.1 GNU Radio . . . 41
2.5.1.2 OptiSDR . . . 41
2.5.2 Past Tool-Flow Surveys . . . 42
2.5.2.1 Performance Measurement . . . 42
2.5.2.2 Design Productivity Measurement . . . 42
2.5.3 Standardisation . . . 43

2.5.3.1 SCA . . . 43
2.5.3.2 STRS . . . 45
2.6 Chapter Summary . . . 45

3 Methodology **47**
3.1 Systematic Development of the Comparison Criteria . . . 47
3.1.1 System Engineering Process . . . 48
3.1.2 Ideal Tool-Flow Requirements . . . 48
3.2 FPGA Tools Study Process . . . 50
3.2.1 Tools Selection . . . 50
3.2.2 Case Study Selection . . . 50
3.2.3 Gateway Development Process . . . 51
3.2.4 Tool-Flow Evaluation Process . . . 52
3.2.4.1 Design Productivity . . . 52
3.2.4.2 Operational Efficiency . . . 53
3.3 GPU Tools Study Process . . . 53
3.3.1 Tools Selection . . . 53
3.3.2 Case Study Selection . . . 54
3.3.3 Software Development Process . . . 54
3.3.4 Tool-Flow Evaluation Process . . . 54
3.3.4.1 Coding Effort . . . 54
3.3.4.2 Operational Efficiency . . . 55
3.4 Chapter Summary . . . 56

4 Systematic Analysis and Design of an Ideal High-Level SDR Tool-Flow **58**
4.1 Tool-Flow System Conceptualisation . . . 58
4.1.1 Stakeholder Identification . . . 58
4.1.2 Stakeholder Requirements . . . 59
4.2 Tool-Flow System Definition . . . 61
4.2.1 Functional Analysis . . . 61
4.2.1.1 Functional Architecture . . . 61
4.2.1.2 Functional Structure and Interface . . . 63
4.2.2 System Requirements . . . 64
4.2.2.1 Functional Requirements . . . 64
4.2.2.2 Performance Requirements . . . 67
4.2.2.3 Usability Requirements . . . 67
4.2.2.4 Interface Requirements . . . 70
4.2.3 Ideal High-Level FPGA Tool-flow System Architecture . . . 70
4.2.3.1 High-Level Modeling . . . 70
4.2.3.2 Code Generation . . . 71
4.2.3.3 Verification . . . 71
4.2.4 Ideal High-Level GPU Tool-flow System Architecture . . . 72
4.2.4.1 High-Level Modeling . . . 72
4.2.4.2 Algorithm Verification . . . 73
4.2.4.3 Code Generation . . . 73
4.3 Chapter Summary . . . 73

5 Evaluation of Tool-Flows for Rapid Prototyping SDR DSP Operations on FPGAs **74**
5.1 Evaluation Criteria . . . 74
5.1.1 Proposed Ideal High-Level FPGA SDR Toolflow . . . 74

5.1.2 Design Productivity and Performance . . . 76
5.2 Qualitative Study . . . 76
5.2.1 MyHDL Evaluation . . . 77
5.2.2 Migen Evaluation . . . 79
5.3 Design Productivity and Performance Study . . . 84
5.3.1 Performance Results . . . 84
5.3.1.1 MyHDL . . . 84
5.3.1.2 Migen . . . 85
5.3.1.3 Comparison . . . 86
5.3.2 Design Productivity Results . . . 87
5.4 Chapter Summary . . . 88

6 Evaluation of Tool-Flows for Rapid Prototyping SDR DSP on GPUs 89
6.1 Evaluation Criteria . . . 89
6.1.1 Proposed Ideal High-Level GPU SDR Design Flow . . . 89
6.1.2 Design Productivity and Performance . . . 89
6.2 Qualitative Study . . . 91
6.2.1 CUDA . . . 91
6.2.2 OpenCL . . . 92
6.3 Coding Effort Results . . . 93
6.4 Performance Results . . . 93
6.4.1 Execution Times . . . 94
6.4.2 Throughput . . . 95
6.4.3 Power Consumption . . . 95
6.5 Chapter Summary . . . 96

7 Conclusion and Future Work 98
7.1 Ideal High-Level Tool-Flow Specification . . . 98
7.1.1 FPGA Tool-Flow . . . 98
7.1.2 GPU Tool-Flow . . . 98
7.2 Evaluations of Existing SDR DSP Tool-Flows . . . 99
7.2.1 FPGA SDR DSP Tools . . . 99
7.2.2 GPU SDR DSP Tools . . . 100
7.3 Conclusion . . . 101
7.4 Future Work . . . 101
7.4.1 Increase number of case study programmers for statistical significance . . . 101
7.4.2 Use more and larger case study application . . . 101
7.4.3 VHDL code generation capability for Migen . . . 101
7.4.4 SDR DSP primitives for Migen . . . 101

A Systematic Design of an Ideal High-Level SDR Tool-Flow 110
A.1 Stakeholders Identification . . . 110
A.2 Original Stakeholder Requirements . . . 111
A.2.1 Standardisation Bodies . . . 111
A.2.2 Users . . . 113
A.2.3 Research and Development . . . 113
A.2.4 Hardware Manufacturers . . . 114
A.3 Functional Analysis . . . 115
A.4 System Requirements . . . 120

B FPGA Tools - Source Code **123**
B.1 MyHDL . . . 123
B.1.1 FIR Filter High-Level Specification . . . 123
B.1.2 FIR Filter High-Level Test Bench . . . 123
B.1.3 FIR Filter Generated HDL . . . 125
B.1.4 FIR Filter HDL Test Bench . . . 127
B.2 Migen . . . 127
B.2.1 FIR Filter High-Level Specification . . . 127
B.2.2 FIR Filter High-Level Test Bench . . . 128
B.2.3 FIR Filter Generated HDL . . . 129
B.2.4 FIR Filter HDL Test Bench . . . 132

C GPU Tools - Source Code **133**
C.1 OpenCL . . . 133
C.1.1 FIR Filter High-Level Specification . . . 133
C.1.2 FIR Filter High-Level Test Bench . . . 133
C.2 CUDA . . . 139
C.2.1 FIR Filter High-Level Specification . . . 139
C.2.2 FIR Filter High-Level Test Bench . . . 140

LIST OF FIGURES

1.1 Block diagram of the ideal SDR transceiver . . . 2
1.2 Practical SDR - The MeerKAT receptor . . . 3
1.3 An architecture of a typical modern FPGA . . . 4
1.4 A typical HDL-based FPGA design flow . . . 5
1.5 Typical high-level design methodology. . . . 6
1.6 Architecture of the NVIDIA's GTX Titan X GPU . . . 7
1.7 Overview of the literature review showing the key concepts that underlie this dissertation . . . 11
1.8 Steps for evaluation of FPGA- and GPU-based tool-flows . . . 12

2.1 Overview of the literature review showing the key concepts that underlie this dissertation . . . 15
2.2 Architectural overview of the SPEAKeasy SDR system . . . 16
2.3 Software-defined Radio block diagram . . . 17
2.4 ADC performance (2010-2018): ENOB versus sampling frequency . . . 19
2.5 Generic architecture of a digital down converter (DDC) . . . 20
2.6 Generic architecture of a digital up converter (DUC) . . . 21
2.7 Simple Parallel Implementation of a FIR filter . . . 23
2.8 Trade-off between reconfigurability and development time for FPGA, ASIC, DSP, GPP, and hybrid GPP/FPGA-centric SDR architectures. . . . 25
2.9 SRAM cell . . . 27
2.10 Island-style routing . . . 28
2.11 Hierarchical FPGA routing architecture . . . 29
2.12 Architecture of RHINO FPGA-based SDR platform . . . 30
2.13 ROACH FPGA-based SDR platform architecture . . . 31
2.14 USRP N210 FPGA-based SDR platform architecture . . . 32
2.15 Architecture of the NVIDIA's GTX Titan X GPU . . . 35
2.16 CPU and GPU system architecture . . . 36
2.17 System architecture of the WARP GPU-based SDR platform . . . 37
2.18 High-level synthesis design steps . . . 40
2.19 GNU Radio design flow architecture [1]. . . . 41
2.20 Conceptual architecture of the SCA SDR standard . . . 43
2.21 STRS layered architecture . . . 45

3.1 The System Engineering Process . . . 49
3.2 Waterfall software/hardware development process . . . 52
3.3 Pageable and pinned data transfers from host to GPU . . . 55
3.4 Steps for evaluation of FPGA- and GPU-based tool-flows . . . 56

4.1 Original requirements of the End-Users stakeholder group for the ideal high-level SDR tool-flow. . 60
4.2 Functional architecture diagram for the high-level SDR DSP tool-flow 62
4.3 Functional architecture diagram for the hardware abstraction function. 63
4.4 High-level FPGA-based SDR DSP tool-flow functional structure and interface diagram 64
4.5 Percentage of FPGA project time spent in verification 66
4.6 Logical architecture of Ideal SDR Tool-Flow 71
4.7 The Ideal High-Level SDR GPU Design Methodology 72

5.1 Logical architecture of the ideal high-level tool-flow for SDR DSP development on FPGA platforms. 75
5.2 MyHDL FIR filter RTL simulation results 78
5.3 MyHDL's code generation scheme 79
5.4 FIR filter FPGA utilization results for migen and MyHDL designs 86

6.1 The Ideal High-Level SDR GPU Design Methodology 90
6.2 FIR filter kernel and memory transfer times for CUDA 95
6.3 FIR filter kernel and data transfer times for OpenCL 96
6.4 FIR filter throughputs for CUDA and OpenCL 97

7.1 Logical architecture of the proposed Ideal SDR Tool-Flow 99
7.2 The Ideal High-Level SDR GPU Design Methodology 100

A.1 Original Standardisation bodies stakeholder group requirements 112
A.2 Original End-Users stakeholder group requirements 113
A.3 Original R&D team stakeholder group requirements 114
A.4 Original Hardware Manufacturers stakeholder group requirements 115
A.5 High-level FPGA-based SDR DSP Toolflow Functional Requirements 116
A.6 Hardware abstraction 117
A.7 Design verification 118
A.8 Design optimisation 118
A.9 Code Generation 119
A.10 High-level FPGA-based SDR DSP Toolflow functional structure and interface diagram 120

LIST OF TABLES

1.1 Conference papers published as part of the dissemination strategy. 10
1.2 Journal papers under review as part of the dissemination strategy. 10
1.3 Seminars part of the dissemination plan. 10

2.1 Computational complexity of common SDR DSP kernels. 22
2.2 State of the art FPGA High-level design flows for SDR DSP. 34
2.3 State of the art GPU High-level design flows. 39
2.4 Past studies involving comparative evaluation of high-level tool-flows 44

3.1 Stakeholder Requirements with regard to High-Level FPGA SDR DSP Tool-flow. 49
3.2 FIR filter specification. 51

4.1 Identification of Stakeholders for the Ideal High-Level SDR Design Flow. 59
4.2 Stakeholder Requirements with regard to High-Level FPGA SDR DSP Tool-flow. 60
4.3 Functional system requirements for the high-level FPGA-based SDR design flow. 68
4.4 System requirements for the high-level FPGA-based SDR design flow. 69

5.1 System requirements for the high-level FPGA-based SDR design flow. 76
5.2 FIR filter specification 76
5.3 Evaluation of high-level design tools for FPGA-based SDR design. 84
5.4 Performance of FIR filter VHDL code generated using MyHDL against that of a hand-crafted design 85
5.5 Performance of FIR filter Verilog code generated using Migen. 86
5.6 Summary of coding effort for developing and testing the FIR filter waveform in Migen and MyHDL. 87
5.7 Summary of coding effort and performance results for the FIR filter waveform in Migen and MyHDL. 87
5.8 Gain in NRE design time G_{NRE}, quality loss L_Q and design productivity P_D of Migen vs manual HDL and MyHDL vs manual HDL 87

6.1 FIR filter specification. 91
6.2 Summary of coding effort for developing and testing the FIR filter waveform in CUDA and OpenCL. 93
6.3 FIR filter kernel and memory transfer times for CUDA and OpenCL under paged host to device and device to host memory transfer. 94
6.4 FIR filter kernel and memory transfer times for CUDA and OpenCL under pinned host to device and device to host memory transfer. 95
6.5 Power consumption results for CUDA and OpenCL 96

A.1 Identification of Stakeholders for the Ideal High-Level SDR Design Flow. 111
A.2 Functional system requirements for the high-level FPGA-based SDR design flow. 121
A.3 System requirements for the high-level FPGA-based SDR design flow. 122

LIST OF ABBREVIATIONS

- **ADC** – analog-to digital converter
- **API** – application programming interface
- **APP** – accelerated parallel processing
- **ASICs** – application-specific integrated circuits
- **BEE11** – Berkeley Emulation Engine 1
- **CASPER** – Collaboration for Astronomy Signal Processing and Electronics Research
- **CF** – core framework
- **CLBs** – configurable logic blocks
- **CMOS** – complementary metal-oxide-semiconductor
- **CORBA** – Common Architecture Request Broker Architecture
- **CPU** – central processing unit
- **CTM** – Close-to- Metal
- **DAC** – digital-to-analog converter
- **DBB** – digital baseband
- **DBE** – digital back-end
- **DDC** – digital down-conversion
- **DDR5** – double data rate type 5
- **DDS** – direct digital synthesiser
- **DFE** – digital front-end
- **DoD** – US Department of Defense
- **DRAM** – dynamic random access memory
- **DSP** – digital signal processing
- **DSPs** – digital signal processors
- **DUC** – digital up-conversion
- **EDA** – electronic design automation
- **EEPROM** – electrically erasable programmable read-only memory

- **ENOB** – effective number of bits
- **ESL** – electronic System Level
- **FFT** – fast Fourier transform
- **FHDL** – fragmented hardware description language
- **FIR** – finite impulse response
- **FPGA** – field programmable gate array
- **FSM** – finite state machine
- **GPP** – general purpose processor
- **GPU** – graphics processing unit
- **GPGPU** – general purpose computing on graphics processing unit
- **GRC** – GNU Radio Companion
- **GSPS** – gigasamples per second
- **HDL** – hardware description language
- **HPC** – high performance computing
- **HLS** – high-level synthesis
- **IDL** – interface definition language
- **IF** – intermediate frequency
- **I/O** – input/output
- **IOB** – input/output block
- **ITRS** – International Technology Roadmap for Semiconductors
- **JTAG** – Joint Test Action Group
- **LMS** – Lightweight Modular Staging
- **LUTs** – lookup tables
- **MAC** – multiply and accumulate
- **MEMS** – micro-electro-mechanical systems
- **MIMO** – multiple-input and multiple-output
- **MoC** – model of computation
- **NASA** – National Aeronautics and Space Administration
- **NRE** – Non-recurring engineering
- **NVCC** – NVIDIA's CUDA Compiler
- **OFDM** – orthogonal frequency-division multiplexing
- **OpenCL** – Open Computing Language
- **OS** – operating system
- **PCIe** – peripheral component interconnect express

- **PLI** – procedural language interface
- **POSIX** – Portable Operating System Interface
- **PPL** – Pervasive Parallelism Laboratory
- **PROM** – programmable read-only memory
- **PTX** – parallel thread execution
- **R&D** – Research Development
- **RFFE** – radio frequency front-end
- **RHINO** – Reconfigurable Hardware Interface for computiNg and radiO
- **ROACH** – Reconfigurable Open Architecture Computing Hardware
- **RRSG** – Radar Remote Sensing Group
- **RTL** – register-transfer level
- **RTOS** – real-time operating system
- **SAR** – successive approximation register
- **SCA** – Software Communications Architecture
- **SDF** – Synchronous Dataflow
- **SDK** – software development kit
- **SDR** – software defined radio
- **SDRG** – Software Defined Radio Research Group
- **SE** – system engineering
- **SEU** – single event upset
- **SFDR** – spurious-free dynamic range
- **S&H** – sample-and-hold
- **SKA** – Square Kilometre Array
- **SLOC** – source lines of code
- **SMs** – stream multiprocessors
- **SNR** – signal-to-noise ratio
- **SoCs** – systems on Chips
- **SoI** – system-of-interest
- **SPI** – serial peripheral interface
- **SRAM** – static random access memory
- **SARAO** – South African Radio Astronomy Observatory
- **STRS** – Space Telecommunications Radio System
- **SWIG** – Simplified Wrapper and Interface Generator
- **TLBs** – transactional lookaside buffers

- **TLM** – transaction-level modelling
- **TLP** – textual programming language
- **UHD** – USRP-Hardware-Driver
- **USB** – universal serial bus
- **USRP** – Universal Software Radio Peripheral
- **VCD** – value change dump
- **VPI** – verilog procedural interface
- **WARP** – Wireless Open-ccess Research Platform
- **XML** – extensible mark-up language

CHAPTER 1

INTRODUCTION

This dissertation presents a comparative study of tool-flows for the rapid prototyping of the digital signal processing (DSP) operations of software defined radio. Increasingly high demands for flexibility, interoperability and speed in communication systems are driving the adoption of the emerging SDR technology [2]. SDR is about moving as many of the radio functions that are traditionally implemented on static hardware to programmable hardware platforms, such as field programmable gate arrays (FPGAs), graphics processing units (GPUs) and digital signal processors (DSPs) [3]. Due to the power wall that microprocessor technology hit a few years ago, parallel computing has now become the main approach for meeting the high processing demands of modern high-performance applications, including SDR [4]. However, the adoption of parallel computing platforms, such as FPGAs and GPUs, to meet the speed and flexibility demands of SDR, introduces complexity in programming paradigms. For example, standard FPGA programming methodology is based on register-transfer level (RTL) languages, such as VHSIC Hardware Description Language (VHDL) and Verilog. These languages are difficult to learn and cumbersome to use, given the current increasing design complexities [5].

In an effort to simplify the programming effort required for these platforms without significantly degrading performance, high-level tool-flows targeting one or more of these high-performance platforms are an active area of research in both industry and academia. In this dissertation, we use the term 'tool-flow' to refer to a tool or a sequence of tools that work in concert to facilitate the mapping of an application description, specified in a programming language, onto one or more programmable hardware platforms. Adoption of and interest in adopting high-level design tool-flows by SDR designers is on the rise [3, 6, 7, 8]. Given that there is a growing plethora of different high-level tool-flows developed by both industry and academia, some specifically for SDR DSP and some for generic applications, there is a need for a comparative study of these existing tools [9]. This study can help by providing a detailed analysis of the effectiveness of the tools and thereby guide designers in their tool choices. While such studies have been conducted in the literature [10, 9, 11, 12], at the time of writing this dissertation, there was none targeting the SDR domain. Therefore, this dissertation seeks to address this identified lack of SDR-based tool-flows evaluation, in the hope that the results of the study will be helpful and useful to other domains as well.

This chapter proceeds as follows: section 1.1 provides a brief conceptual background on the key themes, including SDRs, SDR DSP hardware and high-level tool-flows. A description of the problem that motivated this study is given in Section 1.2. Section 1.3 states the overarching aim of this dissertation, which is followed by an outline of key supporting objectives. A plan of how the findings from this work will be disseminated is described in Section 1.4. The chapter ends with a brief overview of the dissertation.

1.1 BACKGROUND

This project is a comparative study of tool-flows with regard to fast development of SDR DSP operations on programmable computing hardware. In basic terms, a tool-flow is a selection, configuration and sequencing of tools in order to undertake development. In this dissertation, the focus is on tool-flows that support design entry at higher levels of hardware and software design abstractions. These kind of tool-flows are referred to as high-

level tool-flows. GPPs, DSPs, GPUs and FPGAs are the most common programmable platforms for implementing SDR DSP operations. Due to time constraints, the study focuses only on selected tool-flows that target GPU and FPGA platforms. This study is specifically aimed at providing qualitative and quantitative insights about the high-level design capabilities of existing tool-flows with respect to SDR DSP applications. The practical importance of the insights from this study include informing SDR DSP designers in making the correct tool choices and identifying areas of improvement for tool designers. The study is carried out within a conceptual context covering the following key topics: SDR, FPGAs, GPUs and high-level tool-flows. A brief conceptual background of these topics is given in this section.

1.1.1 Software Defined Radios

Increasingly high demands for flexibility, interoperability and speed in communication systems are driving the adoption of the emerging technology known as software defined radio (SDR) [2]. According to the wireless innovation forum (formerly the SDR forum) [3]:

> "A software-defined radio is a radio in which some or all of the physical layer functions are software defined."

These radio functions include operations such as channelisation, modulation and demodulation – traditionally, these functions would have been implemented in hardware, using discrete static components, such as transistors and capacitors, or placed on application-specific integrated circuits (ASICs). These traditional systems would have had little flexibility, however; if different frequencies or modulation strategies were needed, then the whole system would have had to be redesigned and refabricated. In comparison, SDR systems are much more flexible: for instance, modulation schemes can be changed rapidly by running different software. The SDR approach was made feasible largely due to the availability of high-speed, low-power sampling and processing technologies. Accordingly, SDR solutions are generally realised by means of various programmable devices, such as digital signal processors (DSPs), multicore processors, graphics processing units (GPUs) and field programmable gate arrays (FPGAs).

Ideally, an SDR should support "any radio waveform at any carrier frequency and at any bandwidth" [3]. As illustrated in Figure 1.1, an ideal SDR architecture consists of a programmable or reconfigurable digital signal processing platform connected directly to an antenna through digital-to-analog and analog-to digital converters (DAC/ADC). The received signal is sampled and digitised directly from the antenna, and then fed to the DSP subsystem, where application-specific radio functions such as channel selection, demodulation and decoding are performed.

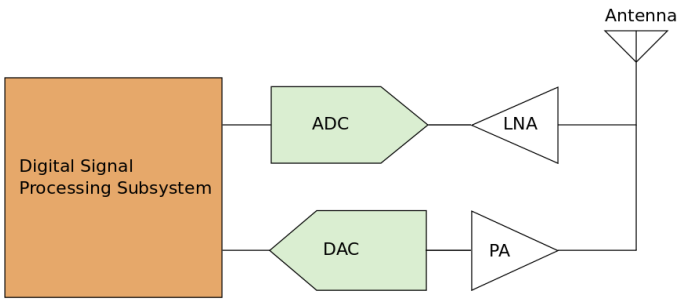

Figure 1.1: The ideal SDR architecture. The antenna is connected directly to the digital signal processing system through the converters. (Image adapted from: [13]).

The ideal SDR cannot be realised practically due to current technical limitations in antennas, digitisers and DSP platforms [3]. Instead, practical implementations, such as the MeerKAT radio telescope receptor [14], which is shown in Figure 1.2, are discussed in the literature. Unlike the ideal SDR concept, however, the MeerKAT receptor has a limited frequency range of 1 - 1.75GHz. The digitiser is placed as close to the receiver as is practical to ensure the highest quality and most stable RF pass-band. Functions such as RF amplification, level control and bandpass equalisation are performed in analog in the RF unit. The digital unit directly converts RF analog signals into digital signals, performs digital down-conversion, channelization, and outputs the results to the correlator via a commodity Ethernet data link. The correlator employs a mixed FPGA/GPU architecture for the signal processing nodes.

Although FPGAs and GPUs are excellent engines for the flexibility and high processing demands of SDR, they are generally difficult to program using traditional low-level tools, due to their complex architectures. This is especially the case for FPGAs, more so than for GPUs. Currently, there exists a growing plethora of high-level design tools aimed at simplifying the development of FPGAs and GPUs. Our aim in this dissertation is to conduct a comparative investigation of the effectiveness of applying some of these tools for developing SDR waveforms for FPGA and GPU platforms respectively.

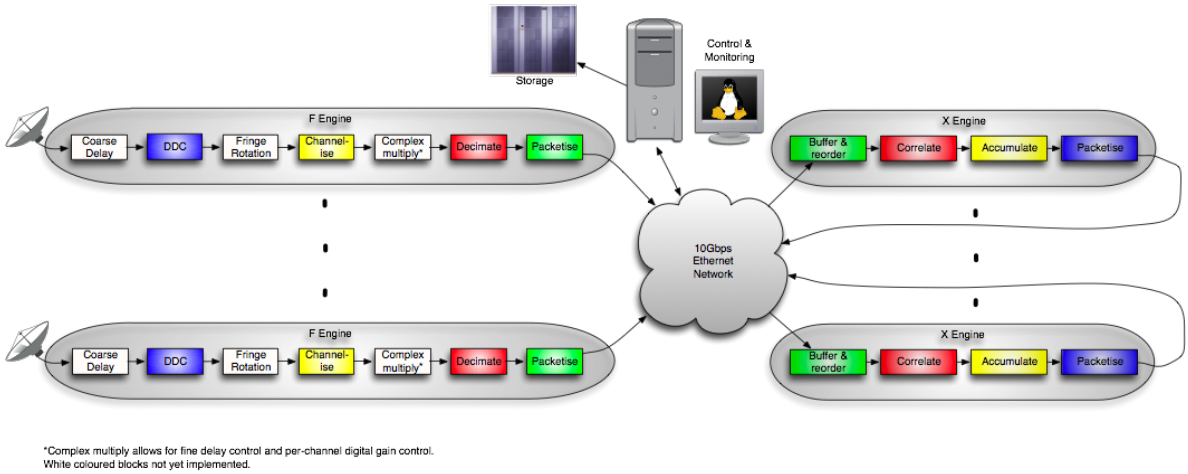

Figure 1.2: The MeerKAT receptor is an example of a practical SDR implementation. Due to practical limitations, some functions are implemented in analogue through dedicated hardware and the rest are implemented digitally through flexible hardware. (Image Source: [14])

1.1.2 Field Programmable Gate Arrays

A simplified architecture of a typical FPGA device is described and the conventional FPGA design methodology based on a hardware description language (HDL) is outlined. The subsections below look at FPGA hardware (1.1.2.1), configuration (1.1.2.2), HDL-based design methodology for FPGAs (1.1.2.3), and high-level design methodology for FPGAs (1.1.2.4).

1.1.2.1 FPGA Hardware

An FPGA is a pre-defined logic substrate that can be configured, in seconds, through user-modifiable code to implement various digital circuits. Figure 1.3 illustrates a simplified architecture of a typical modern FPGA device. The architecture is a highly dense and parallel structure consisting of four main types of elements: configurable logic blocks (CLBs), programmable interconnect infrastructure, programmable input/output blocks (IOBs) and various specialised extending logic blocks (e.g. memory and DSP blocks). The CLB is the basic logic unit of an FPGA. The programmable interconnect routes digital signals between CLBs and to and from the IOBs, which interface the FPGA with the external environment. Modern FPGA devices embed dedicated functional blocks, such as DSP slices, transceiver circuitry and hard microprocessors within the FPGA logic fabric, to save on the

usage of logic cells, and to improve design performance and productivity [15, 16].

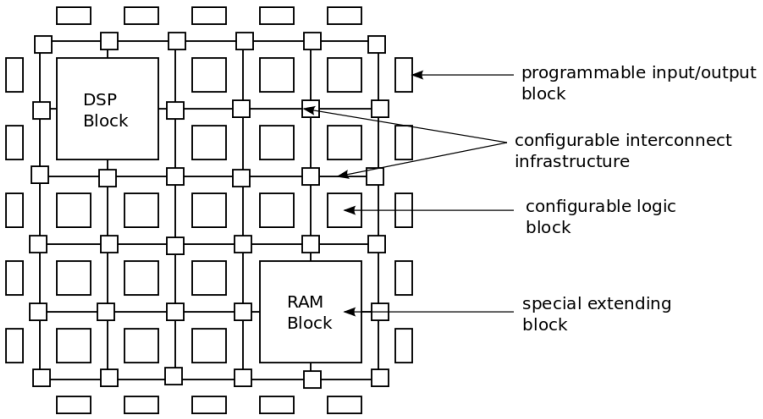

Figure 1.3: An architecture of a typical modern FPGA showing its four main elements: configurable logic blocks, programmable interconnect, programmable input/output blocks and special extending blocks. (Image adapted from: [15])

1.1.2.2 ***FPGA Configuration***

An FPGA is configured by writing a stream of 0's and 1's into its programming memory. The memory functions as programming points for the FPGA's CLBs and routing infrastructure. Most modern FPGAs store the configuration bitstream in static random access memory (SRAM). Since SRAM is volatile, the configuration needs to be downloaded upon power up. The bitstream can also be automatically loaded from non-volatile memories, such as programmable read-only memory (PROM) and serial peripheral interface (SPI) Flash. Microprocessors, microcontrollers and digital signal processors can be used to download the configuration bitstream to the FPGA's SRAM. Several other FPGA configuration approaches are used too, including the most popular Joint Test Action Group (JTAG) and universal serial bus (USB) interfaces [16].

Therefore, the process of designing an FPGA application essentially entails generating a configuration bitstream and then loading it into the device's programming memory. The standard FPGA design flow uses HDLs to specify the application intent, and a set of hardware compiler tools to generate the bitstream from the specification. An overview of this methodology is covered in the following subsection.

1.1.2.3 ***HDL Design Methodology for FPGAs***

The standard FPGA design methodology is illustrated in Figure 1.4. HDLs are used to capture the application specification at the register transfer level (RTL). VHDL and Verilog are the most widely used HDLs. Unlike common software languages, HDLs are inherently parallel in nature and provide various constructs for describing both the structure and the behaviour of the application circuit.

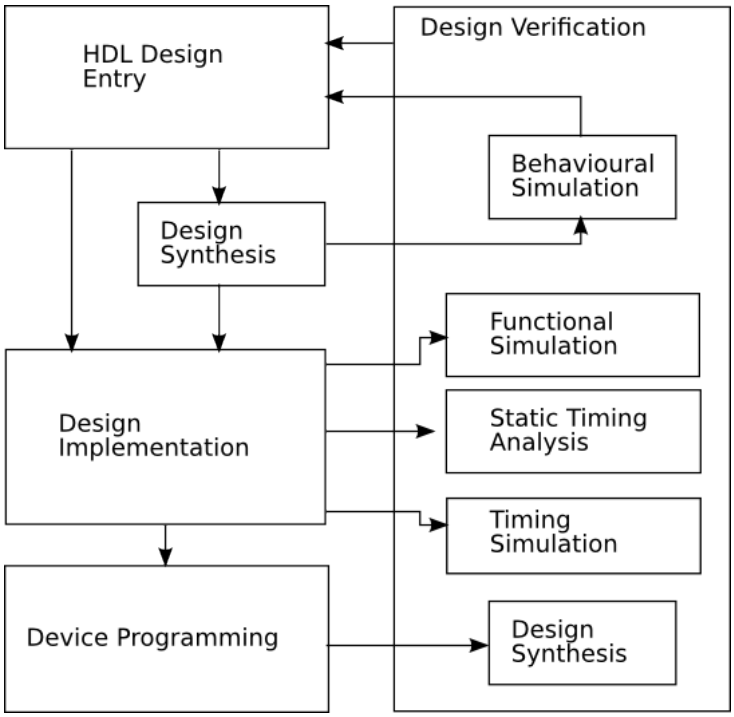

Figure 1.4: A typical HDL-based FPGA design flow [17].

Once captured by using an HDL, the application specification is verified for functional correctness through logic simulation. A logic synthesis tool then translates the RTL code into an optimised network of gates called a netlist which is then fitted into the actual FPGA's logic resources through a place-and-route process, using FPGA vendor-specific tools. Finally, the vendor tools are used to generate a configuration bitstream that is used to program the FPGA device.

The density of FPGAs has been increasing steadily, thus enabling more and more complex and large applications to be implemented on a single chip. However, the standard HDL-based FPGA design methodology is not well suited to cope efficiently with this increasing complexity. As a result, FPGA vendors, electronic design automation (EDA) tool developers and FPGA application designers (including SDR DSP) are shifting toward high-level design languages and tools. The following subsection gives an overview of the high-level design methodology.

1.1.2.4 ***High-Level Design Methodology for FPGAs***

High-level design is a broad term for modern design methodologies that seek to allow designers to describe hardware productively at higher levels of abstraction (above RTL) and thus not worry much about complex, low-level hardware details [18, 9].

A typical high-level FPGA design methodology is based on either algorithmic or system-level design-entry as illustrated in Figure 1.5. The general practice is to augment existing high-level software languages, such as C++, Python, and Scala, to produce new hardware languages, such as SystemC [19], MyHDL [20] and Chisel [21] respectively. A high-level synthesis engine automatically converts the high-level design descriptions into RTL code for FPGA implementation using vendor tools.

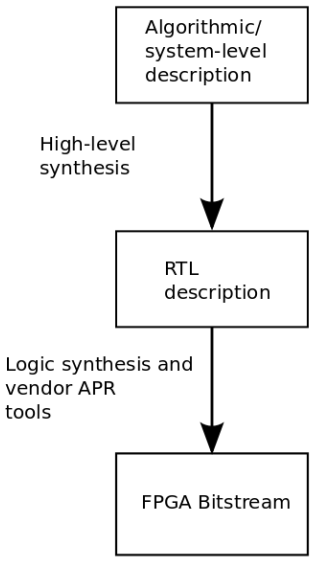

Figure 1.5: Typical high-level design flow. In the high-level design methodology, the design abstraction level is raised above RTL, to either behavioural level or to system-level or both, in order to simplify design entry and improve productivity without incurring significant losses in performance. (Image adapted from: [22])

1.1.3 **Graphics Processing Units**

In addition to FPGAs, our study also compares high-level design methodologies for rapid prototyping of SDR DSP operations on GPU platforms. A description of GPU devices, covering their internal hardware makeup (1.1.3.1) and common programming methodologies (1.1.3.2), is presented in this subsection.

*1.1.3.1 **GPU Hardware***

Graphics processing units (GPUs) are complex evolving chips that deliver high theoretical computational power at relatively low cost. GPUs were originally made for handling computer graphics processing. However, as the chips evolved in both complexity and computational power, it was discovered that they could be used efficiently to also perform high-performance scientific operations that were traditionally handled by the central processing unit (CPU). Therefore, GPUs are presently used in various applications besides 3D computations, including SDR DSP, bioinformatics, computational finance and cryptography, using a technique known as General Purpose computing on Graphics Processing Units (GPGPU) [23, 24].

Architecturally, the graphics processing unit (GPU is quite a different chip compared to a conventional general purpose processor (GPP). Figure 1.6 shows the architecture of the NVIDIA's GTX Titan X GPU that was used for experiments in this dissertation. The GPU connects to the main memory via a peripheral component interconnect express (PCIe) connector. It has its own memory space, 12GB in size. Generally, the power of the GPU resides more in its large number of processing cores, called Streaming Multiprocessors (SMs). Titan X has 3072 cores. Due to their architecture, GPUs primarily cope with problems that exhibit high levels of data parallelism, such as many of the SDR DSP algorithms.

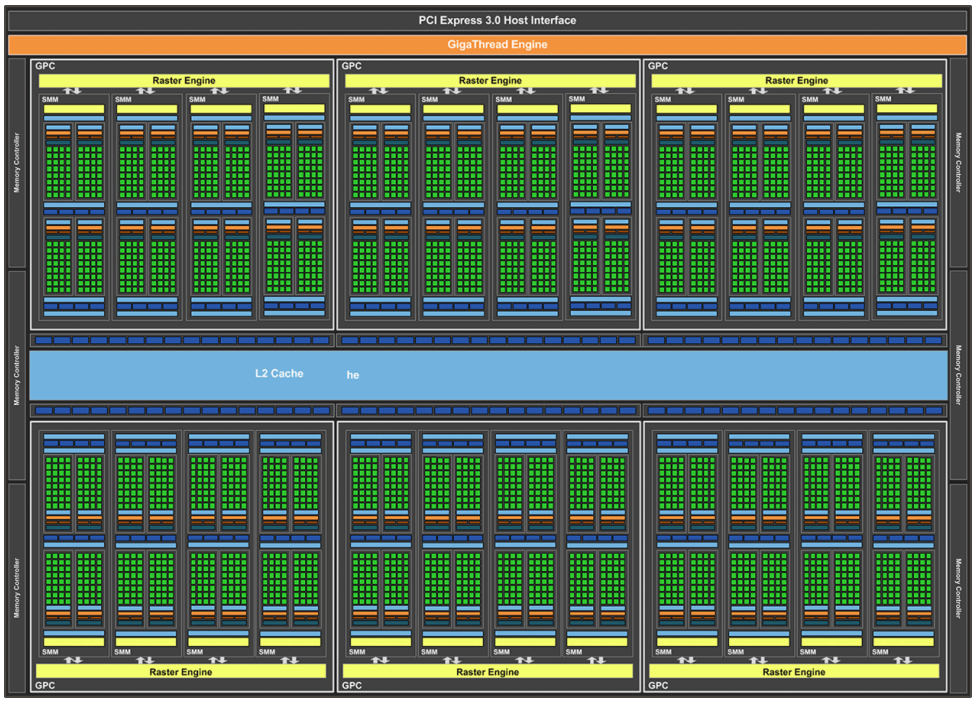

Figure 1.6: Architecture of the NVIDIA's GTX Titan X GPU [25].

1.1.3.2 High-Level Design Methodology for GPUs

The large number of cores present in GPUs require a strong and constraining form of structuring, which induces significant changes and complexity into the programming paradigm, especially with regard to task distribution [26].

Traditionally, mapping non-graphics applications to GPUs required re-writing the entire application in terms of graphical primitives, using graphical languages such as OpenGL or DirectX. Because this approach was tedious and error-prone, several high-level academic and third-party design frameworks appeared, which abstracted away the low-level GPU programming details. However, beginning in 2007, GPU vendors started to release general-purpose GPU programming languages, such as AMD Close-to- Metal (CTM) and NVIDIA CUDA. Presently, NVIDIA CUDA [27], DirectCompute [28], and OpenCL [29] are the most common GPU general-purpose languages.

1.2 Problem Statement

Digital signal processing (DSP) is at the heart of the digital revolution that has brought us many applications, such as communications, radio astronomy, radar and biomedical systems. While DSP applications were traditionally implemented using ASICs and DSPs, there is now a notable growth of interest in the use of FPGAs [30, 31] and GPUs [32], especially for high-performance DSP applications, such as the correlator designed for the Square Kilometre Array (SKA) radio telescope. In particular, SDR DSP, because of its flexible high-speed processing requirements, is one main example of an application domain that could benefit greatly from the flexibility and high compute density of FPGA and GPU technologies [33].

However, although GPUs and FPGAs possess excellent features that can enable a variety of high-performance DSP applications, some application designers still experience major difficulties in using them efficiently, because of their complex programming requirements [5]. The lack of effective and easy-to-use design methodologies is a major barrier to the widespread adoption of FPGA technology [34]. The conventional methodology is difficult to use, as it forces designers to work at a low register-transfer-level (RTL), using HDLs such as VHDL or Verilog to build solutions from first principles. This results in lengthy development times that may increase risk factors

for development companies, such as missing market opportunities. The conventional FPGA design methodology is proving inadequate to cope with the increasing complexity of FPGA devices and thus suffers from low design productivity, a problem known in the literature as the design productivity gap [35].

High-level design, which involves specifying designs at higher levels of abstraction, is one of the most promising approaches for bridging the design productivity gap for modern processing platforms, including FPGAs and GPUs. It is much more productive in cost and design time to describe a design in high-level behavioural or algorithmic code than in RTL for FPGAs and low-level C code for GPUs, provided that comparable performance levels can be achieved.

While there have been significant efforts in the last few decades to develop new high-level design languages, methodologies, tools and frameworks to bridge the FPGA and GPU design productivity gap, little work has been done to evaluate the relative capabilities of existing high-level solutions. In fact, the few available publications on the subject focus primarily on proprietary tools [36, 10, 9, 37, 38], and very few consider open-source solutions [39, 40]. Further, there is no systematic approach in conducting the evaluations. It is often not clear what evaluation criteria were used or how the criteria were synthesised. This dissertation thus presents a systematic and comparative evaluation of the capabilities of a selection of open-source high-level design tools to facilitate fast prototyping of SDR DSP algorithms on FPGA and GPU platforms.

1.3 OBJECTIVES

The overarching aim of this dissertation is to

> "study and compare high-level tool-flows for fast prototyping of SDR DSP operations on FPGA and GPU platforms".

The study seeks to determine how capable high-level tool-flows are, relative to one another, in facilitating rapid development of quality SDR DSP applications on FPGAs and GPUs. In order for the comparative study to be meaningful and fair, effective and comprehensive criteria should be used. The criteria used in related non-SDR domain studies generally evaluate tool-flows only in terms of the quality of the generated applications designs. There are few criteria that allow the tool-flow's capability to not only produce quality designs to be evaluated, but also to speed up the application development process. Given that the tool-flows will be studied for a specific domain (the development of SDR DSP applications), the criteria need take this into account. For example, the criteria should specify the expected functional and performance capabilities of a good SDR DSP tool-flow for FPGA and GPU platforms respectively. Currently, such a comprehensive specification does not exist.

The following set of sub-objectives have been designed in response to the main objective:

1. **Define an *"Ideal High-Level SDR Tool-Flow"* specification:** Develop systematic and reliable evaluation criteria to be used in benchmarking the capabilities of a selection of existing high-level design tools in order to streamline the SDR DSP development process for FPGA and GPU platforms.

2. **Conduct a comparative evaluation of existing high-level FPGA design tools:** Using the systematic evaluation methodology developed in 1, benchmark the capabilities of a selection of existing high-level design tools to facilitate easy and fast prototyping of SDR DSP applications on FPGA platforms.

3. **Conduct a comparative evaluation of existing high-level GPU design tools:** Using the systematic evaluation methodology developed in 1, benchmark the capabilities of a selection of existing high-level design tools to facilitate easy and fast prototyping of SDR DSP applications on GPU platforms.

4. **Recommend high-level design tools for fast prototyping SDR DSP on FPGAs and GPUs:** Based on the comparative evaluations conducted according to 2 and 3, recommend a high-level design tool that is best suited to be used as is, or modified, to simplify and accelerate the design and implementation of SDRs on FPGAs and GPUs respectively.

1.4 DISSEMINATION STRATEGY

Several research materials have been produced from this project since its inception. The dissemination strategy outlines a set of activities and methods that have been and will be undertaken to facilitate communication of these research outputs to relevant audiences. The strategy includes conference presentations, academic seminars and publication of articles in journals. Each of the activities targets a variety of audiences. Besides simply communicating the knowledge and findings produced in this research, dissemination activities conducted during the course of this dissertation also provided opportunities where valuable feedback was received and used to improve the quality of the study.

An outline of the dissemination strategy used in this dissertation is given in Subsections 1.4.1 (Objectives and Approach) and 1.4.2 (Target Audiences and Plan). The types of activities and methods that will be used to disseminate information from this project and also to present a concrete working plan, showing past and future dissemination activities, are discussed below.

1.4.1 Objectives and Approach

The planned dissemination activities include:

- **Conference presentations** - Presentation of the results of the undertaken investigations to high profile relevant conferences, as listed in the dissemination plan in the following subsection. Conference presentations and publications were chosen for use as short-term feedback platforms on several relatively smaller aspects of this study. Therefore, various factors, such as shorter response times and the possibility of in-person meetings and feedback, were considered when selecting specific conference options.
- **Journal publications** - Publication of the results of the undertaken investigations to high profile journals, as mentioned in the dissemination plan subsection that follows. Journals are a long-term approach, and they accelerate the wide dissemination of research knowledge by publishing high- quality articles that provide more detail on the work.
- **Seminar presentations** - Presentation of academic seminars on topics related to the investigations conducted in this dissertation. These seminars will be used to raise awareness among academics and researchers about high-level design methodologies and high-performance DSP platforms for SDR. It is envisaged that this will introduce others to the various technologies and tools covered in this dissertation and encourage them to consider or even apply them to their projects.

1.4.2 Target Audiences and Plan

The planned dissemination activities presented in the previous subsection are targeted at the following three main audiences.

1. **Designers of SDR DSP Applications** - This group includes all people and organisations that develop software and firmware for SDR applications. Our targets, but is not limited to, the following specific organisations: the Digital Back-End (DBE) team at SKA South Africa, the Radar Remote Sensing Group (RRSG) and the Software Defined Radio Research Group (SDRG) at UCT.
2. **SDR DSP and EDA Researchers** - All individuals and organisations involved in research and development of new improved technologies, frameworks, tools and methodologies for modelling, designing, implementing and verifying SDR DSP operations fall under this group. This work targets, but is not limited to, the following specific SDR DSP and EDA researchers: Milkymist labs, MyHDL community, the DBE team at SKA South Africa, Collaboration for Astronomy Signal Processing and Electronics Research (CASPER) at Berkeley University and the Pervasive Parallelism Laboratory (PPL) at Stanford University.
3. **Standardisation Organisations** - Part of the deliverables from this study is the proposed systematised specification of an ideal high-level tool-flow for rapid prototyping of SDR DSP operations. It is hoped that

the proposed methodology will be of value to standardisation organisations, in both the SDR DSP and EDA domains. Therefore, some of the specific standardisation organisations targeted by this work include, but are not limited to, the Wireless Innovation Forum (formerly the SDR Forum) and Software Communications Architecture (SCA).

The planned dissemination activities (past and future) involving seminars, conference papers and journal papers are outlined in Tables 1.3, 1.1, 1.2 respectively below.

Table 1.1: Conference papers published as part of the dissemination strategy.

Name	**Date and Venue**	**Audience**	**Paper Title**
IEEE Radio and Antenna Days of the Indian Ocean (RADIO)	Sept 2015, Mauritius	SDR hardware and software platforms and technologies researchers and designers	Evaluation of High-Level open-source tool-flows for rapid prototyping of Software Defined Radios[41]
IEEE International Conference on Mechanical and Intelligent Manufacturing Technologies(ICMIMT)	Feb 2018, South Africa	Information Technology Researchers	Systematic design of an ideal tool-flow for accelerating big data applications on FPGA platforms[42]

Table 1.2: Journal papers under review as part of the dissemination strategy.

Name	**Publisher**	**International/Local**	**Paper Title**
Advances in Science, Technology and Engineering Systems Journal (ASTESJ)	ASTESJ	International	Systematic Design of an Ideal Toolflow for Accelerating Big Data Applications on FPGA Platforms
International Journal of High Performance Computing and Networking (IJHPCN)	SCOPUS	Parallel Computing	Comparative Study of Tool-Flows for Rapid Prototyping of SDR DSP Operations

Table 1.3: Seminars part of the dissemination plan.

Venue	**Title**	**Audience**
University of Cape Town (UCT)	Design of an Ideal High-Level Tool-Flow for SDR DSP	Remote Radar Sensing Group (RRSG)
UCT	Introduction to SDR Technology: Design Tools and Hardware	Communications Technologies Students and Researchers
National University of Lesotho (NUL)	Design of a SDR - based Set-Top Box System for Lesotho's Digital Migration	Computer and Electronic Engineering Students and Academics

Continued on next page

Table 1.3 – *Continued from previous page*

Where	What	Audience
NUL	Introduction to FPGA Technology	Faculty of Science and Technology Students and Academics

1.5 Dissertation Overview

The aim of this dissertation is to survey the capabilities of various high-level electronic design tools to streamline the process of developing SDR DSP applications on FPGA platforms. While much work has been done to build a variety of high-level design languages, tools and frameworks targeting FPGAs and other devices, there are no systematic comparative surveys of existing high-level design solutions. This gap has motivated the research summarized in this dissertation.

Chapter 1, Introduction, sets the scene by providing a brief conceptual background of FPGAs, SDRs and high-level design. FPGAs are presented as sophisticated hybrid computing engines that possess the flexibility of software plus the performance of hardware. A simplified architecture of a typical FPGA device, showing the positions and roles of its main elements – CLBs, IOBs, programmable interconnect and dedicated function blocks – is described. SDR is introduced as an emerging technology that adds flexibility to radio communications by leveraging flexible hardware platforms, such as FPGAs. High-level design – in other words, designing hardware at higher levels of abstraction, such as behavioural or system level – is introduced as the glue that pulls hardware and applications together, offering a promising solution to the current gaps and challenges in the design industry.

Chapter 2, Literature Review, is a critical discussion of the four main themes, shown in Figure 1.7, that underpin this research: FPGAs, SDR DSP, High-Level Design and EDA Tool Surveys. The review defines the literature context in which the subject matter of this dissertation is placed, and further strengthens the motivation for this research by identifying gaps in the literature on the subject of high-level design tool surveys for FPGA-based SDR DSP.

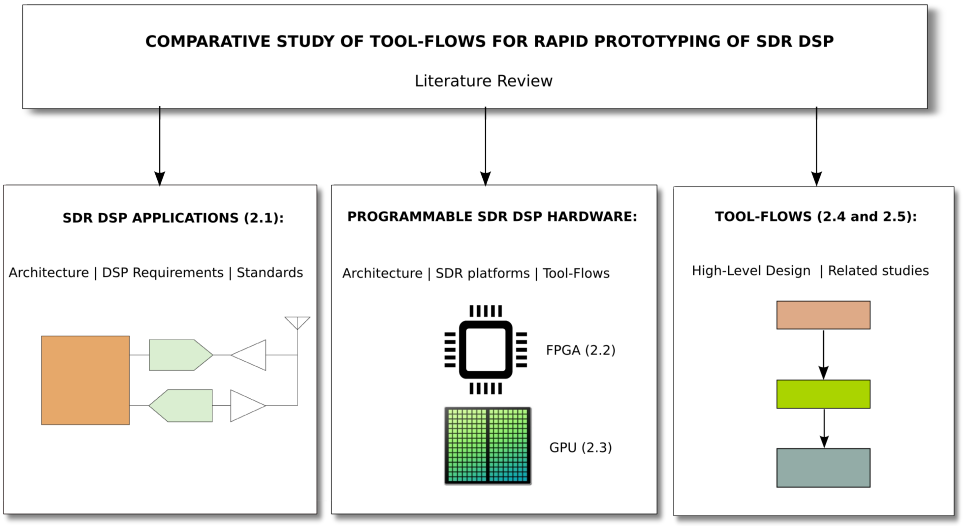

Figure 1.7: Overview of the literature review showing the key concepts that underlie this dissertation.

We first present a review of SDR technology that focuses especially on the architecture of a typical SDR waveform and associated computational characteristics.

Since SDR DSP operations can be designed on various programmable hardware platforms, including one or a combination of GPPs, DSPs, GPUs and FPGAs, a review of these platforms, especially their specific programming architectures and programming requirements, is essential in order to define a more holistic comparison criteria for the various tool-flows. The Chapter presents an architectural and design review of both FPGA and GPU technologies.

An exhaustive study of all available high-level tool-flows falls beyond the scope of this dissertation. Instead, the approach adopted in this dissertation is to select, from the current landscape of existing tool-flows, a few of these for the comparative study. Therefore, it is necessary to introduce and review the high-level tool-flows landscape in order to identify and briefly characterise the pool of existing tools from which we can choose. An up-to-date survey of the high-level tools and tool-flows currently available for digital signal processing and specifically SDR is therefore presented.

Chapter 3, Methodology, presents the research methodology that is adopted in this dissertation to realise the original aim of this project of performing a comparative study of high-level tool-flows for rapid prototyping of SDR DSP operations on FPGA and GPU platforms. The methodology is organised into two main parts, the first part focuses on FPGA tool-flows and the second on GPU tool-flows. Figure 1.8 shows a summary of steps that are followed to evaluate both FPGA and GPU tool-flows.

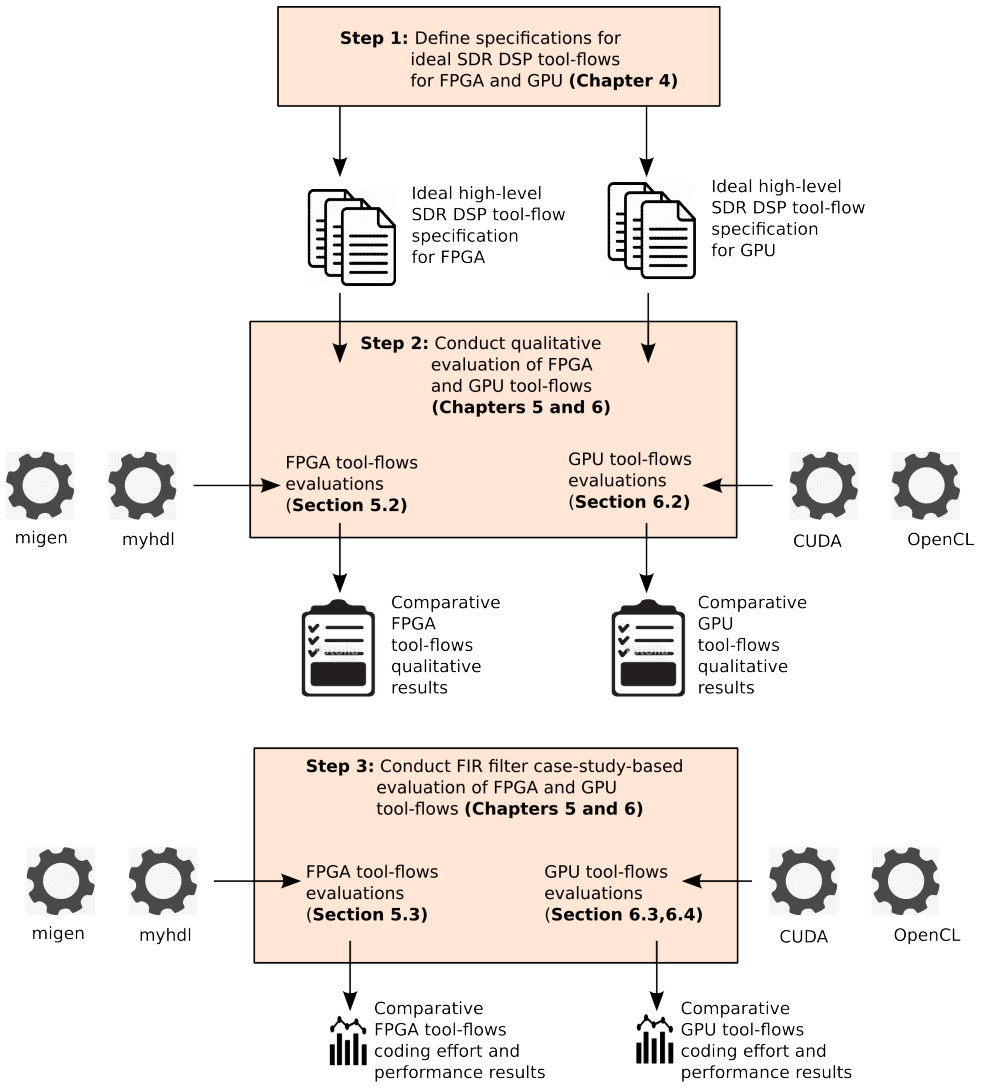

Figure 1.8: Steps for evaluation of FPGA- and GPU-based tool-flows.

The approach is to first develop a comprehensive criteria to use in comparing the tools. In order to produce a holistic criteria, a system engineering process is followed to develop the criteria in the form of a system specification for an ideal high-level SDR DSP tool-flow on FPGA and GPU platforms. Once the comparison criteria is defined, the next step in the approach is to perform a study of tool-flows for rapid prototyping SDR DSP on FPGA and GPU platforms respectively. For each of the two platforms, this entails i) selecting representative tools to study, ii) evaluating the selected tools against the systematic criteria iii) following a waterfall development process to implement a FIR filter case study application using each of the selected tools and iv) using the case study to comparatively evaluate the tools in terms of design productivity and performance. The tools, methods and metrics used to evaluate both the design efficiency and performance of each of the selected FPGA and GPU tool-flows are described.

Chapter 4, Systematic Design of an Ideal High-Level SDR DSP Tool-Flow, describes the systematic development of the system requirements for an ideal high-level tool-flow for fast prototyping of SDR DSP on FPGA platforms. While several high-level design flows exist both in the literature and in industry, there has generally been little work done to define what constitutes a true high-level SDR DSP design flow. As a result, some previous works on comparative tool surveys employ loosely defined comparison criteria, which in turn affect the objectivity of the results. However, development of a complete specification of an ideal high-level FPGA-based SDR DSP design flow is beyond the scope of this dissertation. We only provide a description that will be sufficient for defining the objective tool evaluation criteria in Chapters 5 and 6.

Chapter 5, Evaluation of High-Level Tool-Flows for Implementing SDR DSP on FPGAs, presents a comparative evaluation of the capabilities of Migen and MyHDL high-level design tools to streamline the process of developing SDR DSP on FPGA platforms. Each of the tools is evaluated against the ideal tool-flow system requirements according to four main categories: specification, validation and verification, code generation and implementation. A simple bandpass FIR filter waveform is designed and implemented, using each tool in order to gain hands-on knowledge of the capabilities of the tools, beyond what is said concerning them in the literature. The gaps and strengths identified in each tool are discussed. The generated RTL implementations are synthesized and compared in terms of timing and logic utilization performance metrics.

Chapter 6, Evaluation of High-Level Tool-Flows for Implementing SDR DSP on GPUs, presents a comparative evaluation of the capabilities of OpenCL and CUDA to streamline the process of developing SDR DSP on GPU platforms. Each of the tools is qualitatively evaluated against the ideal tool-flow system requirements grouped into four main categories: specification, validation and verification, code generation and implementation. A simple bandpass FIR filter waveform is then implemented, using each tool in order to gain hands-on knowledge of the capabilities of the tools, beyond what is said concerning them in the literature. The gaps and strengths identified in each tool are discussed. The two implementations are profiled and compared in terms of running time and power consumption performance metrics.

Chapter 7, Conclusions and Future Work, concludes the dissertation and outlines recommended future work. In this dissertation a comparative study of high-level tool-flows for rapid prototyping of SDR DSP operations on FPGAs and GPUs has been presented. Migen and MyHDL were studied under FPGA-based tool-flows and OpenCL and CUDA were studied under GPU-based tool-flows. A systematised specification of an ideal high-level SDR DSP tool-flow has been developed and used as a criteria to qualitatively evaluate the features of the tools. In addition, a FIR filter case study was implemented in each of the tools and used to study and compare the tools in terms of design productivity and design performance.

CHAPTER 2

LITERATURE REVIEW

The aim of this dissertation is to conduct a comparative study of high-level tool-flows. It specifically focuses on tool-flows for the rapid prototyping of SDR DSP operations on FPGA and GPU platforms. The approach is, firstly, to select the FPGA and GPU tool-flows that will be considered for the study. Secondly, to define the systematic criteria and select a suitable case study SDR DSP application that will be used to compare the tool-flows.

In order to realise the approach and thereby achieve the objectives of this dissertation, a review of several concepts and techniques that underlie this study is necessary and is summarised in Figure 2.1 and expanded in this chapter.

Section 2.1 presents a review of SDR technology that focuses especially on the architecture of a typical SDR waveform and associated computational characteristics.

Since SDR DSP operations can be designed on various programmable hardware platforms, including one or a combination of GPPs, DSPs, GPUs and FPGAs, a review of these platforms, especially their specific programming architectures and programming requirements, is essential in order to define a more holistic comparison criteria for the various tool-flows. Section 2.2 presents an architectural and design review of FPGA technology, while GPUs are reviewed in Section 2.3.

An exhaustive study of all available high-level tool-flows falls beyond the scope of this dissertation. Instead, the approach adopted in this dissertation is to select, from the current landscape of existing tool-flows, a few of these for the comparative study. Therefore, it is necessary to introduce and review the high-level tool-flows landscape in order to identify and briefly characterise the pool of existing tools from which we can choose (Section 2.4). An up-to-date survey of the high-level tools and tool-flows currently available for digital signal processing and specifically SDR is therefore presented in Section 2.5.

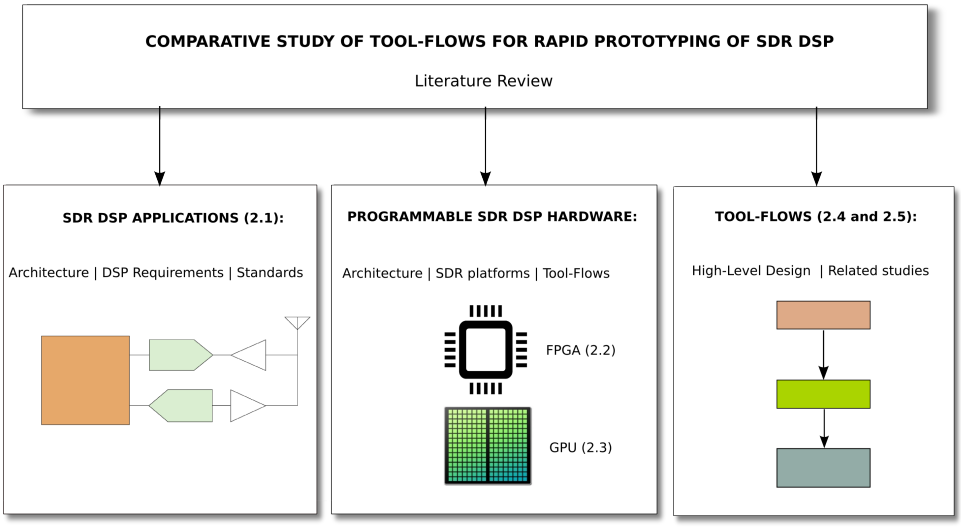

Figure 2.1: Overview of the literature review showing the key concepts that underlie this dissertation.

2.1 SOFTWARE DEFINED RADIO

This high-level tool-flow study focuses specifically on the software defined radio (SDR) application domain. Invented a couple of decades ago, SDR is a technology that emphasizes the need for as many functions of a radio system as possible to be defined in flexible hardware, as opposed to the traditional fixed hardware implementations [3]. According to the Wireless Innovation Forum, one of the top ten most wanted innovations in SDR technology are high-level techniques for the efficient porting of SDR DSP operations onto complex signal processing platforms [43].

This section gives a brief historical background (2.1.1) of the SDR technology, reviews the main parts of a modern SDR hardware architecture (2.1.2), discusses the general computational features and requirements of SDR DSP waveforms (2.1.3), and ends with a brief review of the key modern computing processors (2.1.4) that are used to provide the required flexibility for SDR functions.

2.1.1 Brief History of SDR Technology

Early wireless instruments were hardware defined: they were developed to support a fixed radio waveform format, as enabled by the implementation hardware.

By the early 1990s, it became a challenge to keep up with the rapid technological advancements in wireless communications, using the traditional hardware-defined radio architecture. Taking advantage of new radio protocols by using the traditional radio architecture required the hardware to be redesigned, but this is both costly and time-consuming.

The lack of interoperability between wireless equipment posed another challenge [44]. For example, a hardware-defined wireless instrument that had been built to support a specific radio protocol could not interoperate with equipment that was supporting a different protocol. An example of this occurred during the U.S. invasion of the Caribbean island of Grenada in 1979, where the lack of a fully integrated and interoperable communication system posed a major challenge to the invading forces. For example, the lack of coordination in respect of the use of different radio frequencies prevented communications between Marines and Rangers of the U.S. military.

Consequently, in 1984, in an effort to address the limitations of the traditional hardware-defined radio architecture, a team at E-Systems, Inc. (now Raytheon) built a digital baseband receiver, which they referred to as a “software radio”. This marked the first emergence of the software-defined radio concept. The receiver provided programmable

interference cancellation and demodulation for broadband signals through the use of adaptive filters, along with shared-memory multiple array processors.

In 1991, DARPA's SPEAKeasy became the first military program that specifically required the physical layer functions of a radio to be defined in software [45]. The goal of the SPEAKeasy program was to demonstrate a single radio that could operate from 2MHz to 2GHz and support interoperability among multiple radio devices. Moreover, to future-proof the radio hardware, the system was further required to support the deployment of new wireless protocols and modulations. An architectural overview of the SPEAKeasy software radio is shown in Figure 2.2.

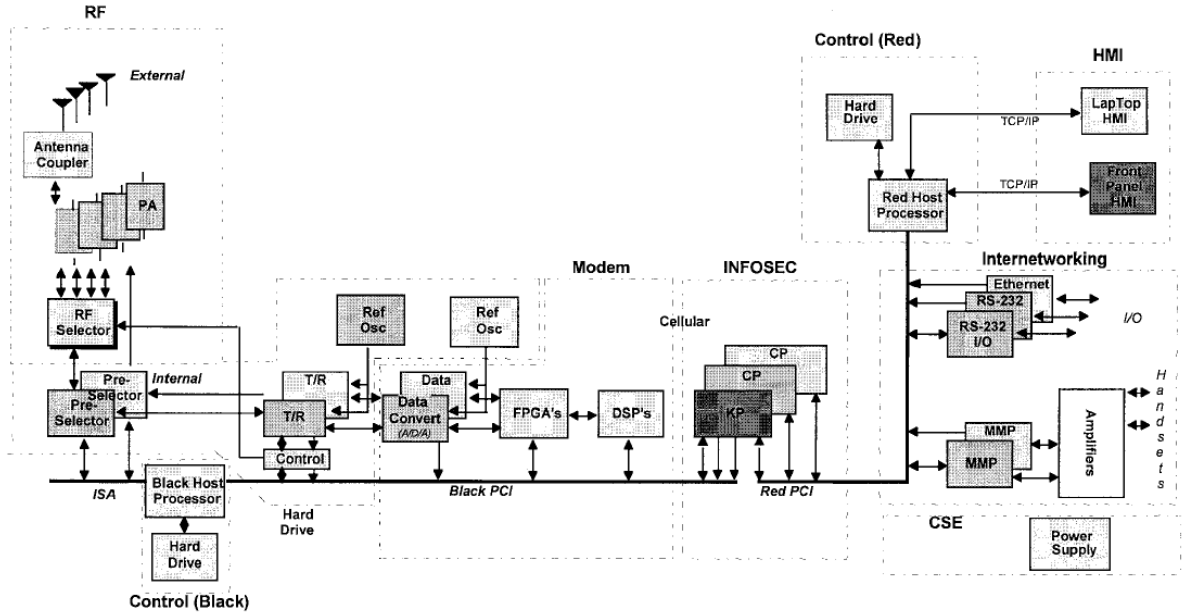

Figure 2.2: Architectural overview of the SPEAKeasy SDR system (Image source: [45]).

Joseph Mitola independently reinvented the term "software radio" in 1991 to refer to a plan to develop a Global System for Mobile communications (GSM) base station that would combine a digital receiver with E-System's digitally controlled communications jammers for a true software-based transceiver. The term "software defined radio" is credited to Mitola, who published the first paper [13] on the topic in 1992; it refers to a class of radios that can be reprogrammed and reconfigured via software.

2.1.2 SDR Hardware Architecture

A typical SDR hardware architecture (Figure 2.3) is a traditional hardware-defined super-heterodyne transceiver, with as many operations as possible moved to programmable hardware devices. The architecture can be organised into the following three main sections:

- **Analog signal processing subsystem** - Due to technological limitations, not all radio functions can be moved to programmable hardware. The analog subsystem includes all the analog SDR hardware elements concerned with capturing and processing the radio signals in analog form.
- **Digitisation subsystem** - The main goal of SDR technology is to perform as many radio operations as possible in software on digital programmable hardware. The digitisation subsystem includes the hardware elements required to convert radio signals from analog to digital and vice versa.
- **Programmable digital signal processing subsystem** - This subsystem concerns all the programmable hardware devices used to implement the flexible operations of the SDR system.

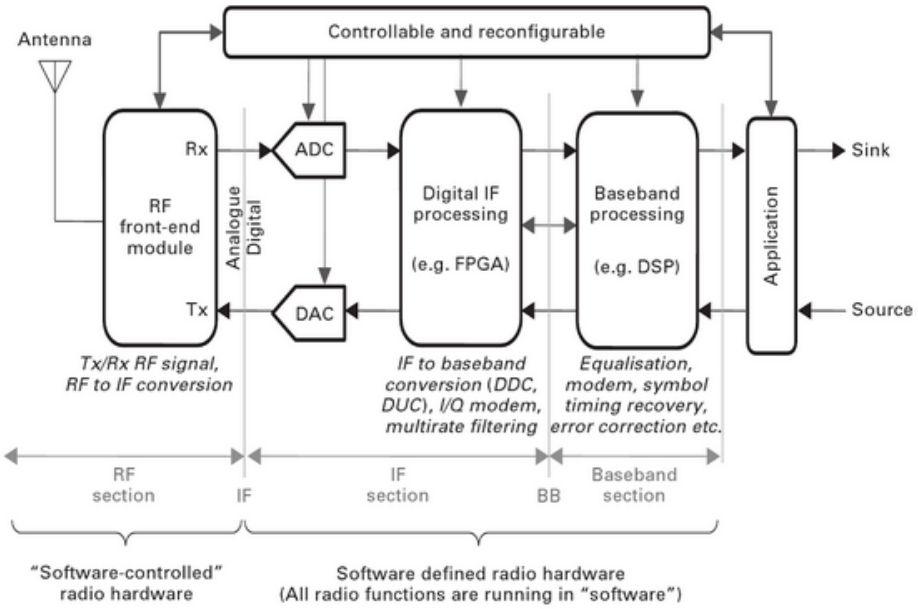

Figure 2.3: Software-defined Radio block diagram (Image adapted from: [46]).

The following subsections will review the radio frequency front-end subsystem (2.1.2.1), Analog-to-digital converters (ADCs) (2.1.2.2) and digital-to-analog converters (DACs) (2.1.2.3), as well as at programmable digital signal processing (2.1.2.4).

2.1.2.1 The Radio Frequency Front-End Subsystem

An ideal SDR system is required to implement all the radio operations digitally in flexible hardware. However, due to technology limitations, this cannot yet be realized in practice. In practice, therefore, SDR systems consist of the analog part in the form of the radio frequency front-end (RFFE) signal processing subsystem. The RFFE includes analog circuitry responsible for reception/transmission of the carrier signal, and up/down-conversion of analog signals to either passband or baseband respectively, or to an intermediate frequency (IF) [47]. The main circuitry elements are:

- antennae for signal reception/transmission
- front-end filters and amplifiers for signal conditioning and
- front-end mixers for up/down conversion.

A brief description and review of these main RFFE elements is presented below. Although the RFFE comprises radio operations that are practically difficult to implement flexibly on programmable hardware, various attempts have been made, both in academia and industry, to develop new techniques and/or re-use existing ones to achieve some degree of flexibility, even in the RFFE [3, 48]. In the case of antennae, smart wideband antenna technology is a trend towards providing flexibility in signal reception/transmission for SDR systems. Further techniques, which are used to develop other programmable RFFE modules, such as bandpass filters for channel selection, are discussed too.

Smart Wideband Antenna Technology

An antenna is a key component of any radio system, including SDR systems. In basic terms, an antenna is a passive device that converts an information signal (in a wired system) to electromagnetic waves that travel in space and can be received by another antenna. The receiving antenna changes the received electromagnetic wave into an electrical signal at its terminals, so that it can then be transformed into intelligible information by the receiver [49].

Gain and frequency response are two key antenna parameters when designing an antenna. A fundamental trade-off exists between the beam-width of an antenna and the gain: if the antenna is required to radiate uniformly in all directions, then a low gain is achieved. Conversely, if the antenna is focused to radiate in one direction, a higher gain can be achieved. For example, parabolic dish antennas have a relatively narrow beam-width and a high gain, but must be directed at the communication partner. A group of identical non-directional antennas can be used to emulate a directional antenna. The process of dynamically defining the required antenna radiation pattern falls in the category of smart antennas [3].

Two main approaches are used to support a wide range of operating frequencies for an SDR antenna:

1. Wideband antenna - design the antenna to give reasonable performance over the entire frequency band.
2. Tunable antenna - design the antenna to give good performance over a narrow band, yet with support for tuning over a wider frequency range.

Each of these two schemes has its own unique advantages and disadvantages. Neither of the two is better than the other across all applications. For example, a cognitive radio that is required to monitor the spectrum and search for available bandwidth needs to use a wideband antenna; once an available band is identified, a tunable antenna can then be used to simplify the filtering requirements and improve quality.

Fundamentally, tunable antennas are based on the idea of selectively adding to and removing segments from an antenna to achieve the desired antenna geometry, or including capacitors and inductors to achieve the desired resonance frequency. The addition and removal of segments can be achieved by closing and opening switches between them. The switches can either be made from micro-electro-mechanical systems (MEMS) or semiconductor devices. For example, a wideband antenna capable of being tuned from 6.9 to 13.8 GHz, by using MEMS-based microblowers to reconfigure the shape of the antenna, is reported in [50].

Programmable RF Modules

Besides the antennae, efforts are underway to design programmability into other key elements of the RFFE subsystem too, especially in respect of the RF bandpass filters that are used for channel selection. The task of designing fully programmable bandpass filters is a challenge that remains to be solved. Bandpass filters are used in both transmitters and receivers to ensure good sensitivity and efficient channel usage. They are typically among the most costly RF elements, and also provide the least flexibility. In SDR, flexible bandpass filters are achieved by requiring the filters to be either electronically programmable or to be stacked to produce a filter bank [51, 52, 53].

2.1.2.2 Analog-to-Digital Converters (ADCs)

SDR systems hinge on analog-to-digital converters (ADCs) and digital-to-analog-converters (DACs). They provide a connection between the analog and digital parts of the chain in a wireless system. An ADC translates an analog signal into a digital form by using a two-step process: sample-and-hold (S/H) and digital quantization. The performance of the ADC is mainly limited by the noise from the S/H operation, and by the signal distortion introduced in the quantization step [54].

ADC performance can be measured in terms of several metrics, including signal-to-noise ratio (SNR), sampling rate, resolution and spurious-free dynamic range (SFDR). Resolution is defined in terms of the effective number of bits (ENOB). ENOB measures the effective performance of an ADC compared to that of an ideal ADC, in the presence of noise and distortion. Specifically, the effective resolution of an ADC is the actual resolution (in terms of number of bits) that remains after reduction in the nominal ADC resolution, due to both noise and distortion. SFDR is a measure of the available ADC dynamic range, without taking into account interference and distortion due to spurious noise. The three most commonly used ADC performance parameters are resolution, sampling rate and power dissipation. These parameters are typically combined to create two common figures-of-merit, which are defined in Equations 2.1 and 2.2 below [54].

$$P = 2^B * f_s \tag{2.1}$$

$$F = \frac{2^B * f_s}{P_{diss}} \tag{2.2}$$

Where B is the ENOB, f_s is the sampling rate, and P_{diss} is the power dissipation. P evaluates the combined performance due to resolution and speed, whereas F evaluates power efficiency in terms of speed and resolution.

Most modern ADCs can be grouped according to their respective architectures. Each ADC architecture is best suited for a specific area of application, with certain ranges of resolutions and sampling rates [54]. There are six main types of architectures for modern ADCs: flash, half-flash, folding, successive approximation register (SAR), pipelined and sigma-delta [54, 55]. Figure 2.4 shows a performance overview in terms of resolution versus sampling rate of ADC devices over the period 2010-2018. From the graph, it can be seen that ADC speed is inversely proportional to effective resolution. The performance of the various ADC architectures in terms of resolution and speed is presented in [54].

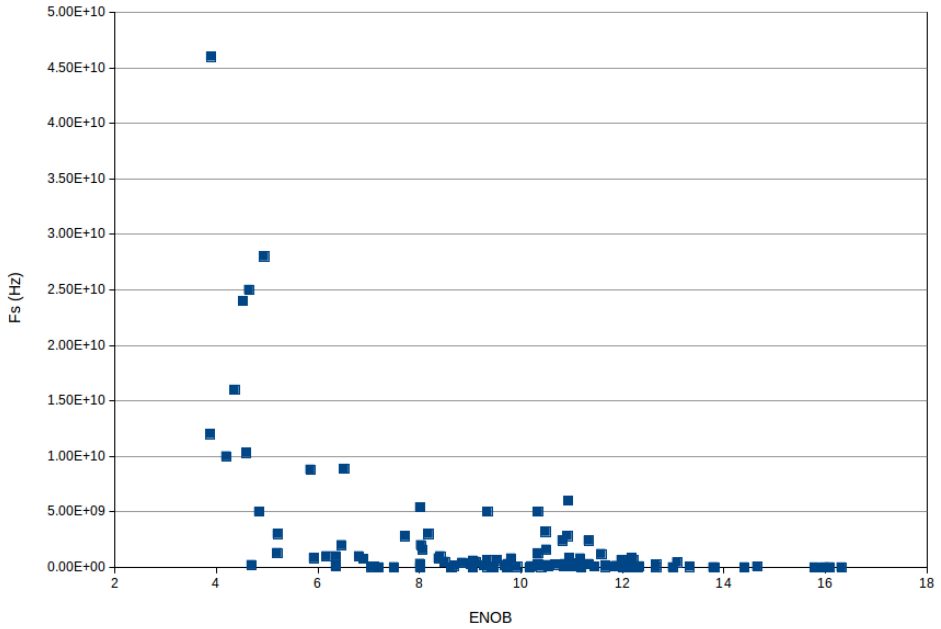

Figure 2.4: ADC performance (2010-2018): ENOB versus sampling frequency [56].

2.1.2.3 Digital-to-Analog Converters (DACs)

As shown in Figure 2.3, a digital-to-analog converter (DAC) is an essential part of every SDR transmitter chain. It is an electronic circuit that is used to perform the reverse function of an ADC – that of converting a digital signal into an analog form. A knowledge of the inner workings of a DAC device is not necessary when choosing tool-flows and developing SDR DSP applications. However, different SDR applications do have different DAC requirements and therefore a knowledge of the key metrics used to characterise the performance of DACs is essential to an SDR designer [49, 50]. There are several DAC architectures and their suitability for particular applications is determined by specific figures of merit, including: resolution, maximum sampling frequency and dynamic range. DACs are easier to design than ADCs and their architectures generally mirror those of ADCs [3].

2.1.2.4 Programmable Digital Signal Processing

The programmable DSP subsystem of an SDR system consists of all radio operations that can be implemented in software, on a variety of flexible hardware platforms, including GPUs, DSPs, multicore GPPs and FPGAs. Especially in relation to this subsystem, there is an increasing need for improved programming methodologies

that can facilitate the efficient porting of SDR operations to the various hardware platforms [43]. Therefore, it is necessary to review these subsystem, focusing especially on the representative DSP operations and their processing characteristics, in order to identify and formulate sound tool-flow comparison criteria, to choose an appropriate case study application and, more generally, to study the different tools in relation to rapid prototyping of SDR DSP.

The programmable DSP subsystem of an SDR system can be grouped into digital front-end (DFE) and digital baseband. The front-end comprises all digital functions that are used to select and translate a desired signal to baseband and vice versa. The digital baseband covers all signal processing performed at baseband level. Both the DFE and digital baseband modules are reviewed below.

Digital Front-End (DFE)

The DFE performs all signal processing operations at IF, immediately after the ADC in a receiver chain, and immediately before the DAC in a transmitter chain. The specific implementations might differ across SDR implementations, depending on design goals, but generally, the two functions of the DFE are digital down-conversion (DDC) and digital up-conversion (DUC).

Digital Down converter (DDC) The role of a DDC is to convert a digitised signal to a lower frequency signal in order to simplify the subsequent radio stages. The architecture of a typical DDC is shown in Figure 2.5. Generally, the DDC consists of three main components: a direct digital synthesiser (DDS), a filter and a down-sampler (which may be built into the filtering module) [57].

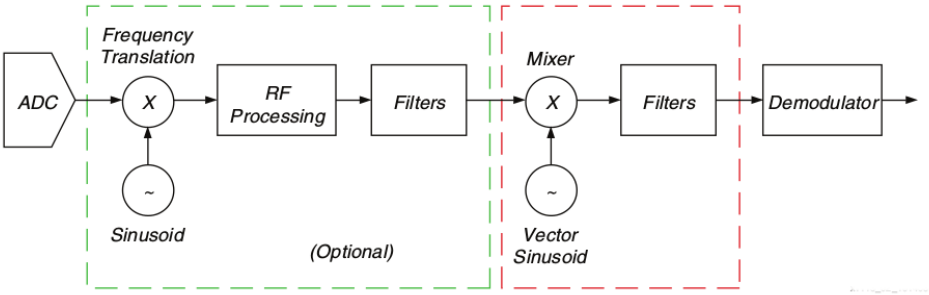

Figure 2.5: Generic architecture of a digital down converter (DDC). Key components consisting of a filter, mixer and vector sinusoid generator (e.g DDS) are shown [57].

Digital Up converter (DUC) The function of a DUC is to move one or more signals from baseband to passband in preparation for wireless propagation to a next communication partner through an antenna. Figure 2.6 shows a generic architecture of a DUC. Typically, the passband signal consists of information bearing carrier signals located at one or more radio frequencies (RF) or intermediate frequencies (IF). The operation of the DUC entails several tasks, including: interpolation, filtering and mixing. Interpolation is performed to raise the sample rate of the signal. Filtering is used to shape the spectrum to the desired form, and mixing translates the baseband signal to required carrier frequencies [57]..

Digital Baseband (DBB)

All SDR DSP operations that are performed at baseband fall under the digital baseband subsystem. The digital baseband subsystem of a SDR waveform can be generalised into three main stages: filter, modem and codec [58]. For example, in a receiver, these stages implement the final signal processing functions at baseband level that are necessary to extract and make sense of the actual communicated information for rendering on an output device.

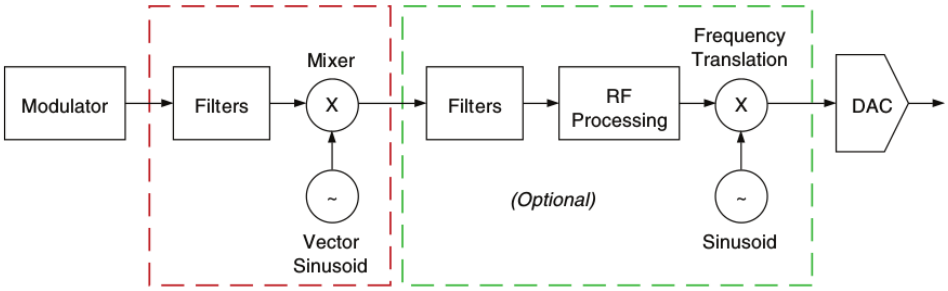

Figure 2.6: Generic architecture of a digital up converter (DUC). Key components comprising of a filter, mixer and vector sinusoid generator (e.g DDS) are shown [57].

Filter stage Filters are required in transmitters and receivers at the digital baseband stage in order to select the correct band. For example, filters such as root-raised cosine and finite impulse response (FIR) filters are used. Given their high computational density, their regularity, and the commonality among the various algorithms involved, full programmability would add insufficient value to compensate for the additional power consumption. Therefore, for SDR implementations, configurable multi-standard filters are commonly used [59].

The FIR filter is the most common type of a software-defined digital filter, because of its excellent stability and linear phase. A FIR filter is a significant processing unit in most modern SDR waveforms [60] and is used as one of the case studies in most SDR hardware and software platforms studies [61, 62, 63, 64].

Modem stage The modem stage (also known as the "inner transceiver") is the stage that differs most across various wireless standards; it is responsible for signal conditioning, which involves operations such as rake reception, correlation, synchronization, fast Fourier transform (FFT), interference cancellation and so forth. The stage is further diversified by the ongoing evolution of modulation schemes and wireless standards to increase performance and throughput. It is at this stage that vendors distinguish themselves by differentiating their products from those of their competitors, improving performance by improving the algorithms. At this stage, performance can be gained or lost through choice or optimization of the right or the wrong algorithms. Since this is the part that is most affected by various wireless standards and implementations, the modem stage should be kept flexible by making it highly programmable [58, 59].

Codec stage The codec stage is also called the "outer transceiver"; it performs a variety of functions, including: (de)multiplexing, (de)puncturing and (de)interleaving, and uses a variety of channel codecs (e.g. convolution, Turbo, Reed-Solomon, Viterbi). The performance of the majority of the functions involved at this stage is influenced by standard algorithms, and thus there is little differentiation among the various vendors. The stage comprises standard operations, which are generally similar among different wireless standards and have high processing requirements. Therefore, flexibility is not an essential requirement for the codec stage, and is therefore commonly implemented in ASIC chips [58, 59].

2.1.3 SDR DSP Waveforms Functionality Characteristics

Generally, DSP operations can be characterised as computationally demanding, highly parallel and data independent, and at times possessing very low arithmetic requirements [65]. It is necessary to understand these characteristics properly in order to estimate their influence accurately in using high-level tool-flows to map DSP operations onto processing architectures, such as FPGAs and GPUs.

A FIR filter example is used in this subsection to discuss the key characteristics of SDR DSP applications in terms of their computational complexity, parallelism, data independence and arithmetic requirements. An understanding of these characteristics is essential in order to define meaningful evaluation criteria for studying the capabilities of

tools to streamline the development of SDR DSP operations.

2.1.3.1 Computational Complexity

DSP operations can exhibit high levels of computational complexity. Table 2-complexity-sdr-kernels shows the computational complexity expressions for several common SDR DSP operations for both real and complex input where for matrices operations, the matrices are of dimensions mxn and nxp, for FFT the size is n, and the vectors are of size n. For example, consider the N-tap FIR filter algorithm shown in Equation 2.3. The operations show that a_0 must first be multiplied with x_k, then the product of a_1 with x_{k-1} should be computed, to which it must be added, and so on. With the tap size of n, this implies that the FIR filter operation requires n products and n-1 summations in order to compute y_k, as shown in Equation 2.4 below.

$$y_k = \sum_{i=0}^{n-1} a_i x_{k-1} \tag{2.3}$$

$$y_k = a_0 x_k + a_1 x_{k-1} + a_2 x_{k-2} + ... + a_{k-1} x_{k-n+1} \tag{2.4}$$

Given that another computation starts when the next sample arrives, the number of operations required per cycle is fixed at 2N per sample. For implementation on a processor, taking into account the additional cycles due to the load/store cycle, the effective processing speed requirement could be 6N per sample.

Considering an audio application with a sampling rate of 48 KHz, a 128-tap filter will need 36.9 Mega samples per second, which might seem realisable for some technologies, but for image processing operations with sampling rates such as 8.5MHz, the speed requirement sky-rockets and results in processing rates of 10 gigasamples per second (GSPS) [65].

Table 2.1: Computational complexity of common SDR DSP kernels.

SDR DSP Kernel	**Coding effort**	
	Real Input	***Complex Input***
Matrix-matrix multiplication	$2mnp$	$5nlog_2n$
Fast Fourier transform	$\frac{5}{2}nlog_2n$	$5nlog_2n$
Forward or back substitution	n^2	$4n^2$
Eigenvalue decomposition: eigenvalues only	$\frac{4}{3}n^3$	$\frac{16}{3}n^3$
Eigenvalue decomposition: eigenvalues and eigenvectors	$9n^3$	$23n^3$
Finite impulse response filter	$n^2 - n$	$4n^2 - 4n$

2.1.3.2 Parallelism

DSP algorithms are generally inherently suited to parallel processing [66]. This can be seen with the FIR filter operation described in Equation 2.3; the FIR operation can either be executed sequentially on a sequential processor or in parallel on a parallel processor as illustrated in Figure 2.7. For parallel processing, each element in the figure is implemented as a hardware processing unit and therefore for a 128-tap filter, 128 registers for delay elements, 128 multipliers for computing the products and an 128-input addition will be required.

In this scheme, we have the required hardware configuration to process a single round of the operation in one sampling period. A parallel platform with high degree of concurrency and sufficiently large memory storage capability will be able to handle this computation within the necessary time budget [65]. Examples of parallel FIR

filter implementations on the FPGA and GPU are reported in [67] and [63] respectively.

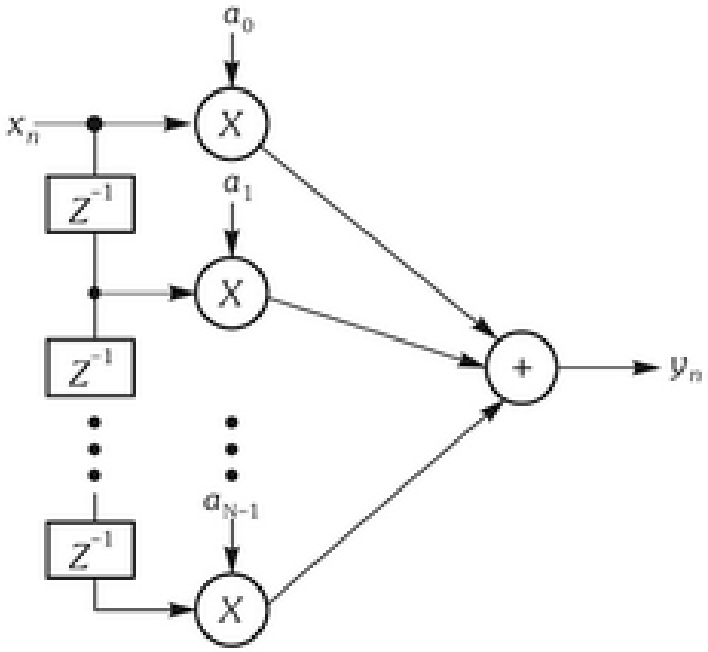

Figure 2.7: Simple Parallel Implementation of a FIR filter [65].

2.1.3.3 ***Data Independency***

Data independence is one of the key requirements for efficient parallel implementation of operations. It makes it possible for computations to be ordered in order to reduce memory and data storage requirements and to increase processing speeds by taking advantage of the available hardware resources on a parallel computing platform. For example, consider below, the following N rounds of the FIR filter operation described in Equation 2.4 above:

$$y_n = a_0x_n + a_1x_{n-1} + a_2x_{n-2} + ... + a_{N-1}x_{n-N+1} \tag{2.5}$$

$$y_{n+1} = a_0x_{n+1} + a_1x_n + a_2x_{n-1} + ... + a_{N-1}x_{n-N+2} \tag{2.6}$$

$$y_{n+2} = a_0x_{n+2} + a_1x_{n+1} + a_2x_n + ... + a_{N-1}x_{n-N+3} \tag{2.7}$$

$$y_{n+N-1} = a_0x_{n+N-1} + a_1x_{n+N} + a_2x_{n+N+1} + ... + a_{N-1}x_n \tag{2.8}$$

It is obvious from the equations that each of the N computations require the x_n data. The entire calculation consisting of the N computations can be performed, such that N simultaneous calculations are carried out for y_n , y_{n+1} , . . ., y_{n+N-1}, by using x_n and therefore removing any need to keep it. This scheme presents a new requirement to keep intermediate accumulator values. This presents the algorithm designer with several different approaches of optimising the algorithm, given specific design constraints, such as speed, resource utilization (for FPGA implementations) and power consumption [65].

2.1.3.4 Arithmetic Requirements

Generally, different DSP applications and technologies have different arithmetic requirements, including precision, fixed-point vs floating-point processing and so on. Word length is commonly used to characterise the arithmetic requirements of DSP operations. In many DSP applications, the requirement of internal precision can be significantly minimised by constraining the word length requirements of the input data. This is defined by the resolution of the A/D device. The internal word length can be reduced, depending on the kind of calculations required. The low arithmetic requirement is key, especially for FPGA designs, since most FPGAs do not support dedicated floating-point flexibility as yet. The reduced word length implies small memory requirements, high speed implementations because adder and multiplier speeds are determined by input word lengths and smaller area. Therefore, there have been significant efforts in determining largest word lengths [65].

2.1.4 Programmable DSP Hardware for SDR

The four main computing processors that are used to implement flexible functions of a SDR system are digital signal processors (DSPs), general purpose processors (GPPs), field programmable gate arrays (FPGAs) and graphics processing units (GPUs). Each of these devices differ in terms of their level of re-configurability, development cycle, cost and performance (Figure 2.8) best suited for different SDR functions. A good understanding of these platforms and their programming requirements will be beneficial in specifying requirements for the ideal high-level tool-flow for mapping SDR operations onto these devices.

2.1.4.1 GPP

A general-purpose processor (GPP) is generally called a central processing unit (CPU) and is like the processor that typically powers personal computers. GPPs are designed to perform a widest possible range of operations including file processing, web browsing, multimedia display, networking, scientific computations, etc. The generic architecture of these devices makes it impossible to optimise them for specific application domains. GPPs are designed to perform well in fixed and floating-point arithmetic and control-oriented functions such as branching and logic. Therefore, they are suitable for implementing most of the functions of the SDR programmable DSP subsystem, beginning with the physical layer up to the protocol and network stacks [3].

GPPs provide the easiest and most mature development methodology. The largest number of programmers is conversant with GPP languages such as C, C++ and Java and development tools. The tools provide high software portability among GPPs from various vendors. GPPs support a wide range of operating systems (OS), from full fledged graphical interfaces such as MS Windows to light real-time options such as FreeRTOS. The OS provides a well-defined abstraction layer that facilitates development of portable software. However, unless real-time operating systems (RTOSs) along with predictable GPP devices, are used when appropriate, usage of standard general purpose OS makes software execution on a GPP unpredictable. It is important to note however that not all real-time applications such as in SDR require an RTOS. For example, for a software-defined TV recorder, the input data might be sampled at say, 10MSps and to avoid frame loss, must be processed at that rate or more. Received and decoded data is stored on disk. This way, the application is clearly real-time. However, there is no hard constraint on when a particular frame is saved to disk. As long as the input buffers are large enough, the software can run on a non- real-time OS. [3].

Multi-core architectures of GPPs can enhance processing performance by executing multiple operations in parallel [47].

2.1.4.2 DSP Processor

A digital signal processor (DSP) is a microprocessor with an architecture optimised for the fast processing requirements of digital signal processing. The optimisations are performed at different architectural levels including hardware and software. DSPs provide excellent support for multiply-acculate (MAC) and fused multiply-add (FMA) operations which are used extensively in many DSP algorithms. Further, they are typically streamlined for data streaming operations and use optimised memory schemes that allow them to load instructions or data simultaneously. Transistors that a GPP uses for complex cache and peripheral interfaces are typically eliminated to

reduce power consumption. DSPs generally out-perform GPPs in terms of power efficiency. DSPs are best suited for datapath-oriented operations such as filtering and Fast Fourier transform (FFT) rather than control-intensive tasks such as the protocol stack. In practice, a typical DSP-based SDR platform would use a DSP with a GPP to implement the protocol stack [3, 47].

The DSP development environment is generally more complex than that for a GPP. DSPs provide a significantly limited OS support. High-level development tools, especially C/C++ compilers, are supported for most DSP, but performance critical applications are generally implemented in assembly. If there is an OS requirement, a real-time OS is almost always used. To achieve optimal application performance from a DSP, the developer must be very familiar with the internal architecture of the device. DSPs are used extensively in software-defined cellular base stations and in radios that require low power and have modest data rate requirements [3].

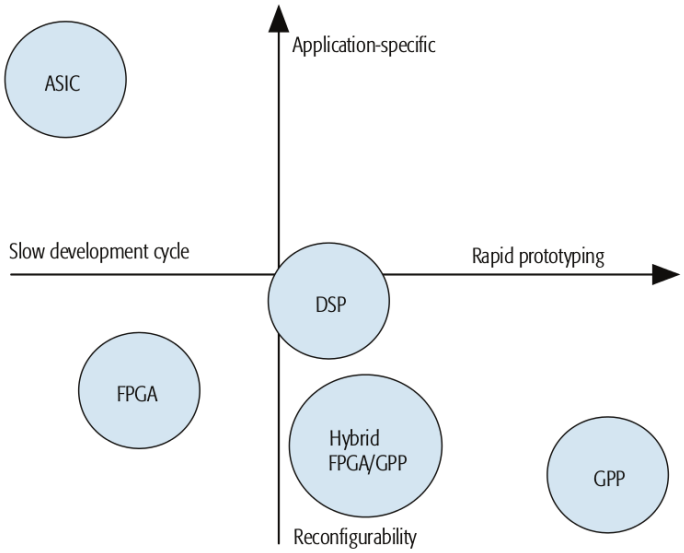

Figure 2.8: Trade-off between reconfigurability and development time for FPGA,ASIC, DSP, GPP, and hybrid GPP/FPGA-centric SDR architectures [47].

2.1.4.3 ***FPGA***

An FPGA is a microchip that is designed to be configured after manufacture. It provides a pre-defined dense infrastructure of logic elements, which can be configured, multiple times, by means of user software to implement various digital circuits. The architecture of an FPGA is shown in Figure 1.3. It is a highly dense and parallel structure consisting of configurable logic blocks (CLBs), programmable interconnect infrastructure, programmable input/output blocks (IOBs) and various specialised extending logic blocks (e.g. memory and DSP blocks). The CLB is the basic logic unit of an FPGA. The programmable interconnect routes digital signals between CLBs, and to and from the IOBs, which interface the FPGA with the external environment. Modern FPGA devices embed dedicated functional blocks, such as DSP slices, transceiver circuitry and hard microprocessors within the FPGA logic fabric to save on the usage of logic cells, and to improve design performance and productivity. For example, Xilinx's Virtex 7 FPGA provides 2 million logic cells, together with 96 built-in multi-gigabit transceivers at 28.05Gbps and 1,200 I/O pins [68, 69].

FPGAs can be classified into three types based on the configuration scheme they support: SRAM, flash and anti-fuse. On one hand, SRAM and flash FPGAs use volatile static memory and non-volatile cells respectively to store the configuration data. On the other hand, anti-fuse FPGAs use unique one-time-programmable chemical switches. A more detailed review of these three FPGA configuration technologies is presented in Section 2.2.1. Generally, mostly SRAM-based FPGAs have been used in terrestrial and space SDR implementations. Flash devices cannot meet the high signal processing requirements of modern SDR. FPGAs excel in high-speed datapath-

oriented operations, but are not suitable for control-oriented operations. A soft or hard GPP microchip is often used with the FPGA [70]. The GPP is then used to handle control-oriented processing [3, 47].

The FPGA's development flow and environment vary greatly from those used for DSP and GPP. It is complex and suffers from relatively lowest developer productivity. The traditional FPGA development flow was presented in Section 1.1.2.3. Section 2.2.4 provides a review of modern high-level development methodologies and tools that are aimed at stream-lining the FPGA's design flow.

*2.1.4.4 **GPU***

Graphics processing units (GPUs) are complex evolving processing technologies that deliver high computational power at relatively low cost. GPUs were originally made for handling computer graphics processing. However, as the chips evolved in both complexity and computational power, it was discovered that they could also be used efficiently to perform high-performance scientific computing applications that were traditionally handled by the central processing unit (CPU). Therefore, GPUs are presently used in various applications, apart from 3D computations, including SDR DSP, bioinformatics, computational finance and cryptography, using a technique known as general purpose computing on graphics processing units (GPGPU) [3, 52].

Architecturally, a GPU is basically a dense array of floating-point multipliers with efficient memory access. In fact, a GPU can be regarded as a very specialised DSP. Much of the physical layer functions of an SDR can be defined as a linear chain of arithmetic operations and thus can be efficiently implemented on a GPU. In terms of performance, a GPU outperforms a DSP and a GPP by factors of at least 5 and 3 respectively, on selected datapath-intensive operations. In terms of theoretical peak performance, GPUs are more efficient than DSPs and GPPs but provide a difficult programming environment [71, 72]. Section 2.3.1 provides an expanded review of the GPU architecture and in Section 2.3.3 a review of modern GPU development methodologies is presented.

2.2 FIELD PROGRAMMABLE GATE ARRAY

Despite their high associated programming cost, FPGAs are gaining popularity in SDR prototyping. Given that they are also one of the target implementation platforms that this study focuses on, it is necessary to have a sufficiently deep knowledge of their programming hardware architectures (2.2.1) and routing architectures (2.2.2); we also give examples of existing FPGA-based complete SDR platforms (2.2.3), as well as programming methodologies and tools (2.2.4).

2.2.1 Programming Architecture

Modern FPGA devices are based on one or a combination of three main programming technologies: SRAM, flash and antifuse[68]. The kind of FPGA programming mechanism used can affect the ultimate quality and usefulness of FPGA applications. Knowledge of an FPGA's programming technology is essential in the design and evaluation of device-dependent implementation tools, such as place and route. The main FPGA programming architectures and their relation to tool-flow design are reviewed in this subsection. A review of these can contribute in the formulation of an ideal criteria for high-level tool-flow for development of SDR DSP applications.

*2.2.1.1 **SRAM-based Programming Technology***

SRAM-based FPGAs store the FPGA's configuration data in static memory cells, which are distributed throughout the FPGA's logic and routing fabric. SRAM memory consumes low power, and is reliable and simple to implement. This is because, unlike dynamic random access memory (DRAM), SRAM does not require a refresh circuit, but instead exhibits data remanence. The SRAM cells are used to program the routing interconnect of FPGAs and the configurable logic blocks, which implement logic functions. Since SRAM is volatile, SRAM-based FPGAs must be configured upon power up. A typical SRAM cell (see Figure 2.9) is built from six transistors. A single bit is stored in four transistors, which form two cross-coupled inverters. The cell has two stable states, which are used to denote 0 and 1.

SRAM is the most commonly used FPGA programming technology. This is because, compared to other tech-

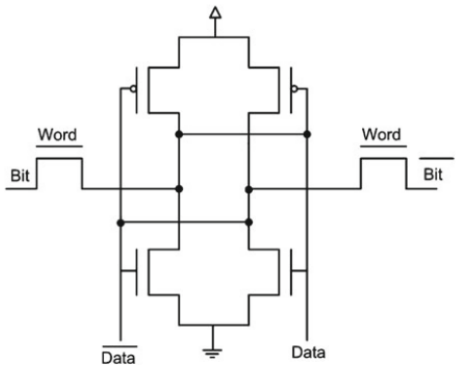

Figure 2.9: SRAM cell [69].

nologies, SRAM offers excellent re-programmability, energy efficiency and higher speed at smaller geometry. Conversely, although it is commonly used, SRAM-based programming technology does have several disadvantages. An SRAM cell is built from six transistors, compared with two for flash and one for antifuse. This makes SRAM expensive due to its high transistor count and effective large area requirement. In addition, since SRAM cells are volatile in design, external storage devices such as PROM are required to provide permanent storage of the FPGA's configuration bitstream. This requirement for additional devices increases the cost and area overhead of SRAM-based FPGAs [69].

SRAM is included in FPGA chips such as the Cyclone, Stratix series of Altera and Spartan, Virtex series of Xilinx.

2.2.1.2 Flash Programming Technology

Besides SRAM, FPGAs can also be configured through flash-based programming technology. Flash-based FPGAs are a combination of FPGAs with internal flash memory. Flash is a popular non-volatile type of memory developed from electrically erasable programmable read-only memory (EEPROM) in the early 1980s at Toshiba. As early as the late 1990s, Actel developed the first industrial flash-based FPGA, ProASIC, intended for space applications [61]. Among others, one of the key features of flash that led to it being considered as a programming technology for the ProASIC FPGA, was its ability to offer reprogrammability without sacrificing non-volatility, as does an SRAM-based FPGA. Flash-based FPGAs consume less power, do not require an external configuration device and are much less susceptible to radiation effects such as single event upset (SEU).

One major drawback of flash programming technology is that, unlike SRAM, flash-based FPGAs cannot be re-configured an infinite number of times. A buildup of charge in the oxide over time makes it difficult to erase and re-program the FPGA. In addition, flash-based programming technology is not based on standard complementary metal-oxide-semiconductor (CMOS) processes. Examples of modern flash-based FPGA families include Igloo and ProASIC3 from Actel.

2.2.1.3 Antifuse Programming Technology

Antifuse is the simplest FPGA programming technology, when compared to SRAM and flash. An antifuse is the opposite of a fuse; it consists of two terminals, such that the unprogrammed state of the terminals link represent a logic zero; when a voltage is applied between them, the antifuse blows, producing a connection of low resistance, which represents a logic one. The connection created between the antifuse terminals is permanent and thus antifuse-based FPGAs do not support re-programmability [15, 16].

Advantages of antifuse programming technology include small area usage, lower resistance and non-volatility. However, there are also significant disadvantages associated with antifuse technology. It is not based on standard CMOS processes and it cannot be reprogrammed. Examples of antifuse-based FPGA devices include Axcelerator by Actel [15, 16].

2.2.2 **Routing Architectures**

FPGA devices use either island-style or hierarchical routing of logic elements to produce a complex design. Each of these routing styles has different benefits and downsides, and can also affect the performance of a development tool. An understanding of these different routing styles is necessary, as the FPGA routing style influences the design of the implementation tools.

2.2.2.1 Island-Style Routing Architecture

A typical island-style FPGA, shown in Figure 2.10, consists of logic blocks ordered in a two-dimensional mesh with interconnect infrastructure equally distributed throughout the mesh. IOBs are also linked to the programmable interconnect network. The routing infrastructure consists of wire segments and configurable switches that are connected in two-dimensional channels between the logic blocks [69].

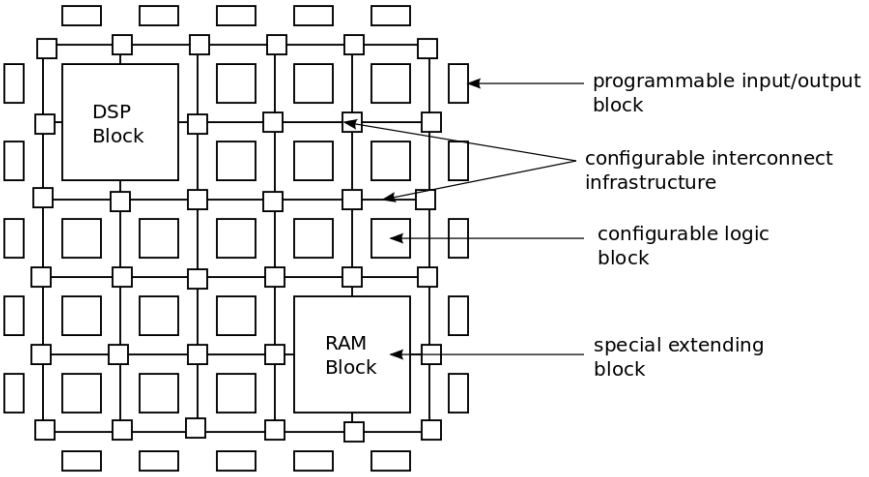

Figure 2.10: Island-style routing FPGA architecture showing configurable and embedded logic blocks as islands in a sea of routing interconnect [15].

The majority of SRAM-based FPGA platforms are based on the island-style routing architecture. This has number of advantages. Because different length interconnect wires are located near logic blocks, it is possible to perform efficient wiring for a variety of design net lengths. In addition, the physical layout of the architecture allows for the easy creation of regular logic tiles and FPGA arrays. Due to the regularity, hardware properties, such as minimum interconnect delays between logic blocks, can be quickly estimated [69]..

2.2.2.2 Hierarchical Routing Architecture

The majority of FPGA designs manifest a locality of connections; in other words, they exhibit hierarchical assignment and interconnection of wires between logic blocks. The hierarchical FPGA routing style takes advantage of this locality by organising the CLBs into clusters, which are interconnected to create a hierarchical structure. Interconnections between CLBs that make up a cluster are done through segments of wires at the first hierarchy level. For blocks located in different clusters, connections are made by traversing several hierarchical levels. The bandwidth of a signal within the FPGA varies depending on the level of hierarchy. Generally, it is broadest at the highest level. Examples of FPGA families that are based on such a hierarchical routing architecture include Altera Flex10K [73], Apex and ApexII architectures [69]..

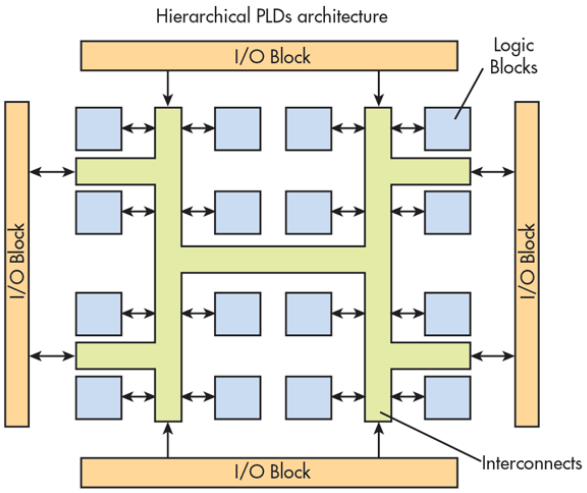

Figure 2.11: Hierarchical FPGA routing architecturecite (Image source [15]).

2.2.3 **FPGA-based SDR DSP Platforms**

A comprehensive list of FPGA-based SDR platforms is presented in [47]. Three of these platforms, Reconfigurable Hardware Interface for computiNg and radiO (RHINO), Reconfigurable Open Architecture Computing Hardware (ROACH) and Ettus Universal Software Radio Peripheral (USRP), are reviewed in detail in this subsection.

2.2.3.1 ***RHINO***

RHINO is an open hardware, FPGA-based computing platform that was developed at the University of Cape Town, with the goal of providing a hardware platform for both embedded systems education and FPGAs that would be easy to use, easy to learn, and affordable to a broad audience [70, 74].

The architecture of the RHINO platform is shown in Figure 2.12. The architecture is divided into three main components: the control processor, the FPGA, and monitoring and management. The function of the control processor is to configure and supervise the FPGA. An ARM Cortex microcontroller is used as the control processor. The FPGA subsystem is in the form of a Spartan 6 FPGA, which provides logic resources for implementation of various SDR DSP operations, including DDC, DUC and different modulators and demodulators. Spartan was selected because, when RHINO was being designed in 2010, it was found to be the cheapest and best performing option, using tools with which the RHINO design team and industry partners were familiar [70].

RHINO was intended to be a cheaper platform than ROACH and Berkeley Emulation Engine 1 (BEE1) or 2. It is architecturally essentially equivalent to ROACH but with lower-cost FPGAs. It also provides a range of power meters that enable direct power measurement for various modules, including the FPGA, as opposed to relying just on ISE report for power measurement. The power meter chips increase the unit price by about 100 to 200 Dollars which renders the platform less affordable, but it was a design compromise due to the research benefits provided by the meters.

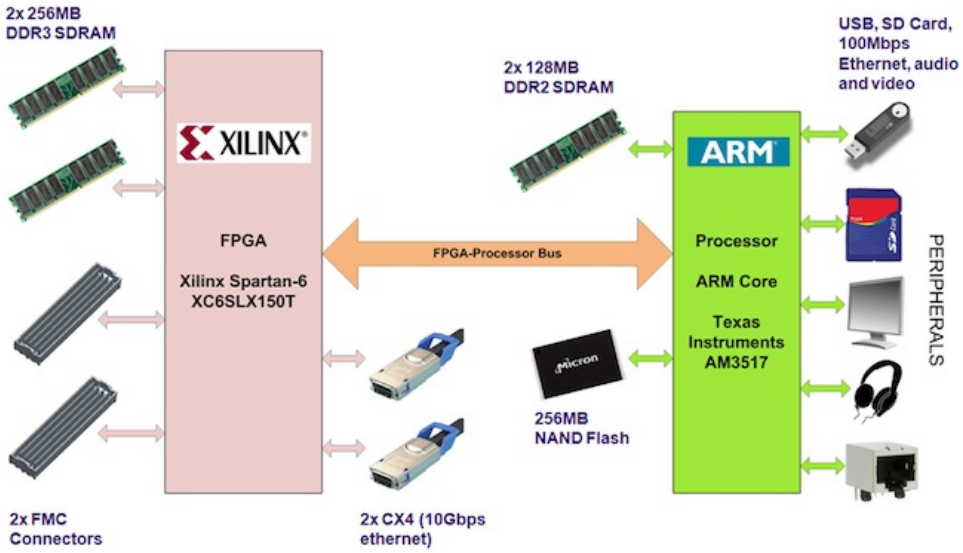

Figure 2.12: Architecture of RHINO FPGA-based SDR platform [75].

2.2.3.2 *ROACH*

The Reconfigurable Open Architecture Computing Hardware (ROACH) is a signal processing platform designed by CASPER for SKA-SA, primarily for radio astronomy applications. It is built into a 1U ATX computer format and, as shown in Figure 2.13, hosts a Xilinx Virtex 6 FPGA and a PowerPC CPU [76]. The FPGA serves as a processing element, while the CPU manages the board and provides an interface to the FPGA's memory-mapped bus. ROACH2 has two Z-DOK expansion connectors that are typically used to connect to ADC cards. It provides a maximum of 80 Gbps of bidirectional digital interface capacity.

A custom tool-flow, MSSGE, which consists of MATLAB Simulink and System Generator, is used to facilitate waveform design, debug and deployment on the platform. An alternative open-source and thus more cost-effective tool-flow for the platform is presented in [77].

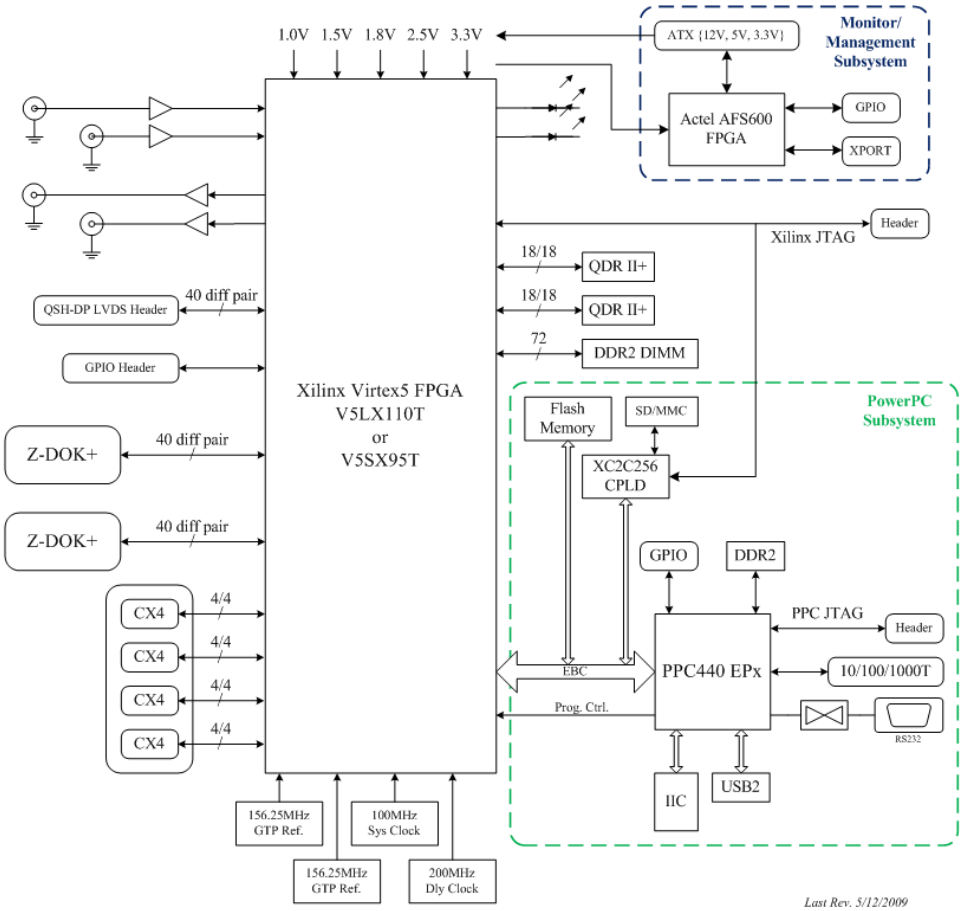

Figure 2.13: ROACH FPGA-based SDR platform architecture [78].

2.2.3.3 Ettus USRP Platform

Universal Software Radio Peripheral (USRP) is a range of software-defined hardware platforms developed by Ettus Research that target a wide spectrum of wireless applications from DC - 6.0GHz. The platforms are categorised into four main classes: the Networked (N)-series, the Bus (B)-series, the Embedded (E)-series, and the X-series. The X-series is a family of high-performance platforms for development of next generation wireless communications systems [47].

The N- and X-series family of platforms consists of an FPGA processor motherboard that is connected to several daughter-boards by high-speed ADCs and DACs. Figure 2.14 illustrates the architecture of one of the N-series platforms, the USRP N-210. The frequency range and bandwidth capabilities supported by the boards are DC-6.0 GHz and 10–160 MHz respectively, which makes it possible to have flexible analog front-end circuitry for different applications [47].

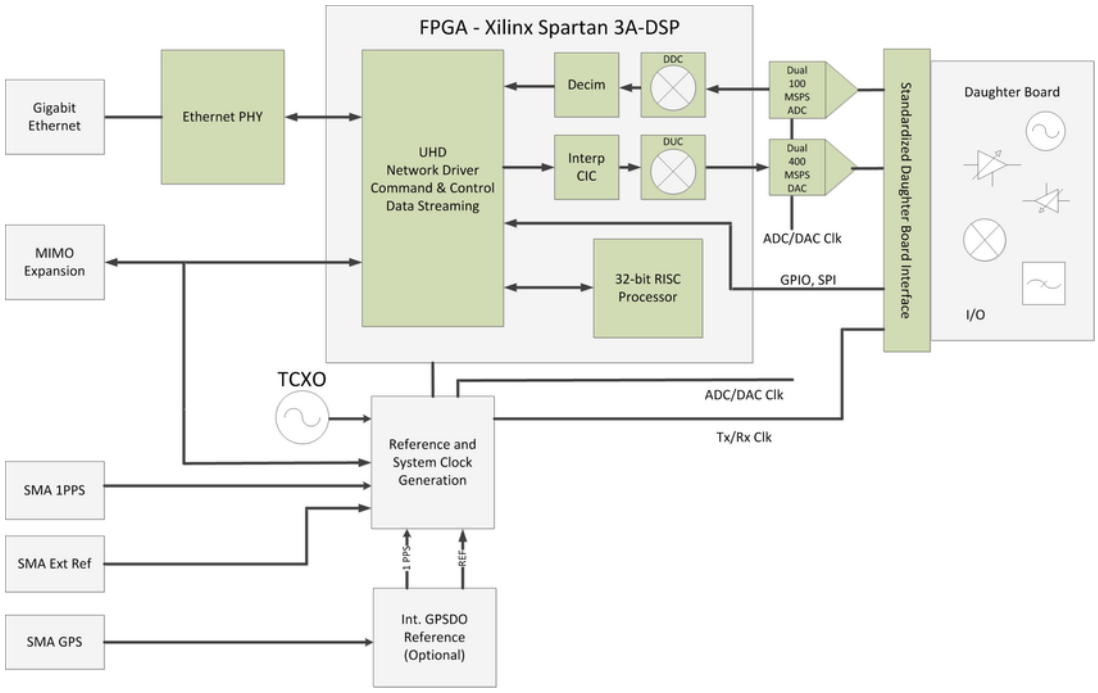

Figure 2.14: USRP N210 FPGA-based SDR platform architecture [79].

Control processors for the configuration and control of USRP FPGA processors can either be in the form of GPPs, softcore processors deployed on the FPGA or embedded Systems on Chips (SoCs) processors such as ARM, which allow portable SDR solutions. For example, for the X, N and B-210 platforms, the signal processing FPGA is controlled by means of a host GPP through external high-speed data bus connections. Transceiver architectures are differentiated across daughter-boards while ADC, DAC speeds and resolutions vary across different models of the USRP series [47].

USRP platforms are compatible with a wide variety of development tools, including GNU Radio, LabView and Mathworks, through the USRP-Hardware-Driver (UHD) application programming interface (API) [80, 47].

2.2.4 High-Level Programming Tools

This subsection reviews a few representative high-level FPGA programming tools. A comprehensive survey of existing high-level FPGA tools is shown in Table 2.2. The survey provides a pool from which specific tools will be selected for comparative evaluation in Chapter 5.

2.2.4.1 MyHDL

MyHDL [20] is a free and open-source Python package developed by Jan Decaluwe that makes it possible for Python to be used to as a hardware description and verification language. Like the Migen Python toolbox [81], the goal of MyHDL is to streamline the conventional HDL-based hardware design methodology by empowering hardware designers with the rich elegance and simplicity of the high-level Python language. With MyHDL, hardware designers can use Python code to model, verify and automatically convert their Python designs to Verilog or VHDL for synthesis and implementation, using a traditional design flow [82].

MyHDL supports high-level modeling of digital hardware designs using the Python programming language. It is aimed at saving hardware designers from the difficult, time-consuming and error prone approach of modeling hardware using traditional HDLs such as VHDL and Verilog by empowering them with the elegant and simple modeling capabilities of the Python language.

2.2.4.2 Migen

Migen [81] is a Python-based toolbox for automating, beyond the conventional HDL-based approach, the process of building complex digital hardware. It is a product of the Milkymist laboratories (M-Labs), a company and community that builds open hardware platforms and design solutions. Built upon the popular Python software language, Migen design toolbox leverages the rich modern software design concepts in Python, such as object-oriented programming and metaprogramming to allow designers to build digital hardware more simply, neatly and productively. At the core of Migen's infrastructure is the fragmented hardware description layer (FHDL) – a formal hardware modeling system that replaces the dominant event-driven paradigm in conventional HDLs with the "notions of combinatorial and synchronous statements" [83].

2.3 GRAPHICS PROCESSING UNITS

In addition to FPGAs, this study also focuses on GPUs as DSP platforms. GPU tool-flows are considered, firstly, because GPUs, though originally designed for graphics processing, have been adopted significantly in other applications, including SDR DSP because of their attractive support for data-parallelism and floating-point arithmetic. Secondly, GPU-based tool-flow studies reported in the literature have focused mainly on other application domains to the exclusion of SDR. As explained earlier in the problem statement (1.2) and objectives (1.3) sections of this dissertation, the objective of this study is to contribute to the literature by providing this missing yet important comparison. Therefore, this section reviews GPUs, beginning with a description of the hardware architecture of modern GPU chips in Subsection 2.3.1. Typical GPU-based SDR DSP hardware platforms and programming frameworks are reviewed in subsections 2.3.2 and 2.3.3 respectively.

2.3.1 Hardware Architecture

Until the past few years, GPUs were primarily used for graphical operations, such as video and image editing, accelerating graphics-related processing, etc. However, due to their massively parallel architecture, recent developments in GPU hardware and related programming frameworks have given rise to general purpose computing on graphics processing units (GPGPUs). A GPU has a large number of processing cores (typically around 2500+ to date), in contrast to a multicore CPU, which has fewer (typically around 2+ to date). These cores are located inside streaming multiprocessors (SMs). For example, the GTX Titan X device shown in Figure 2.15 has 28 SMs and a total of 3584 cores. In addition to the processing cores, a GPU has its own high throughput double data rate type 5 (DDR5) memory, which is many times faster than a typical DDR3 memory. GPU performance has increased significantly in the past few years compared to that of CPUs. Recently, NVIDIA has launched the Tesla series of GPUs, which are specifically designed for high-performance computing. NVIDIA has also released the CUDA framework, which made GPU programming accessible to all programmers without delving into the hardware details. These developments suggest that GPGPUs are indeed gaining more popularity [23, 24].

Table 2.2: State of the art FPGA High-level design flows for SDR DSP.

License	Tool	By	Input	Output	TB	Domain	FP	FixP	Year	Status
Open source	MyHDL	Milkymist	Python	Verilog	Yes	All	No	Yes	2003	Active
Open source	Migen	Milkymist	Python	Verilog	Yes	All	No	Yes	2011	Active
Open source	Ptolemy	UC Berkeley	Schematic, Java	Java	Yes	All	No	Yes	1997	Active
Open source	Delite	Stanford Uni.	Scala	C++, CUDA, OpenCL	No	ML, SDR, Graphs	Yes	Yes	2011	Active
Open source	LegUp	Uni. of Toronto	C	Verilog	Yes	All	Yes	No	2011	Active
Open source	Chisel	UC Berkeley	Scala	C++, Verilog	No	All	Yes	Yes	2012	Active
Open source	Catapult	Mentor	SystemC, C, C++	VHDL, Verilog	Yes	All	Yes	Yes	2004	Active
Proprietary	Vivado	Xilinx	C++/SystemC	Verilog, VHDL	Yes	All	Yes	Yes	2013	Active
Proprietary	Bluespec	Bluespec Inc	BSV	Verilog	No	All	No	No	2007	Active
Proprietary	Cynthesizer	Cadence	SystemC/C++	SystemC RTL, Verilog, VHDL	No	All	Yes	Yes	2009	Active
Proprietary	HDL Coder	Mathworks	Matlab, Simulink	Verilog, VHDL	No	All	Yes	Yes	2012	Active
Proprietary	Cyberworkbench	NEC	C/C++, SystemC	Verilog, VHDL	No	All	Yes	Yes	2011	Active
Proprietary	Impulse C	Impulse Acc.	C	VHDL	No	All	Yes	Yes	2009	Active
Proprietary	Synmphony C	Synopsis	C/C++	RTL	Yes	All	No	Yes	2010	Active
Proprietary	ROCCC	Jacquard Comp.	C subset	VHDL	Yes	Streaming	Yes	No	2010	Active

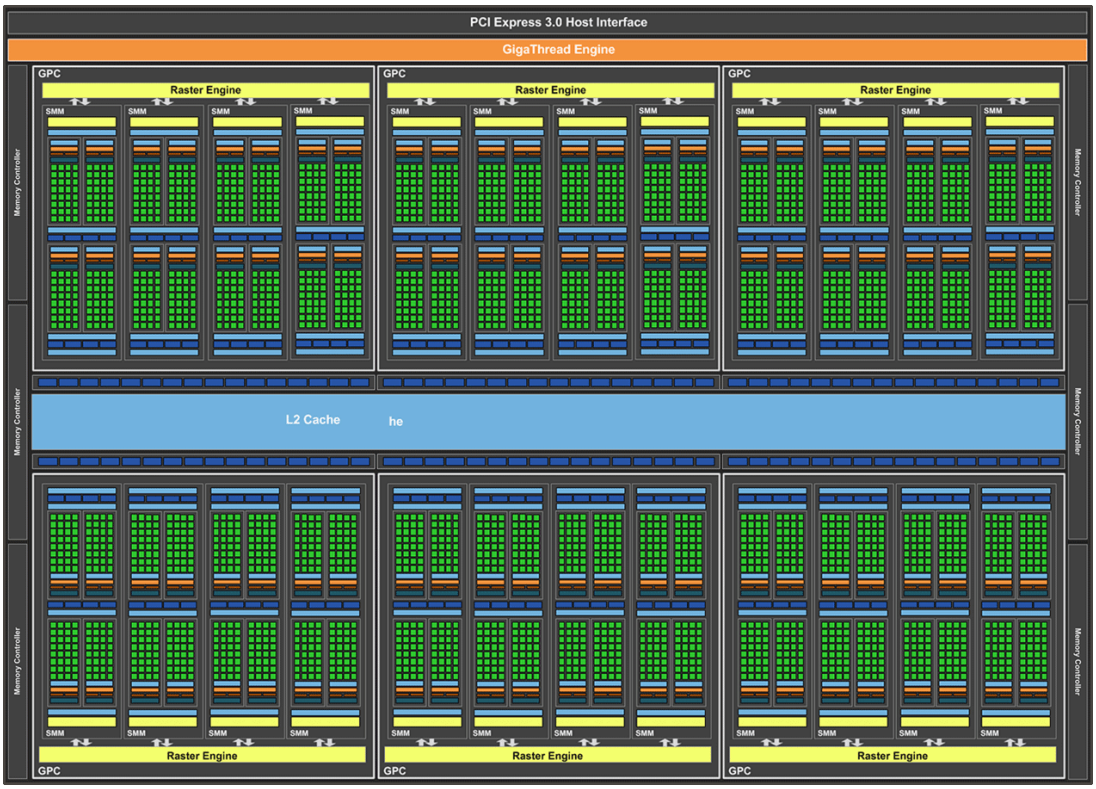

Figure 2.15: Architecture of the NVIDIA's GTX Titan X GPU [25].

A GPU-based uplink detector for massive multiple-input and multiple-output (MIMO) SDR systems is presented in [84]. The CUDA 7.0 Toolkit was used for design implementation, debugging and performance profiling. The design achieves at least 250Mbps throughput with less than 4ms latency, by using two of the latest GPUs for supporting a 128 x 16 antenna system, thus demonstrating the efficacy of GPU-accelerated massive MIMO SDR systems.

The feasibility of using GPUs as baseband processors for supporting software-defined base-station functions to achieve both real-time high-performance and high reconfigurability is reported in [72]. The whole OFDM RX baseband chain, including frame synchronisation, carrier frequency offset correction, FFT and equalisation, demodulator and decoder were implemented on a GPU server. The server comprised of four NVIDIA GTX Titan GPUs and a six-core 3.2GHz Intel i7-3930K CPU. Nsight Eclipse and the CUDA Toolkit were used to design, debug and profile the implementation. The GPU-based implementation can not only achieve less than 3ms latency and more than 50Mbps throughput for processing streaming frames in real-time, but it also offers software-defined flexibility and scalability for supporting future wireless standards.

2.3.2 GPU-based SDR DSP Platforms

Unlike FPGAs, GPUs cannot operate alone in a SDR implementation environment. Rather, in its simplest configuration, a GPU works with a host GPP processor to manage its operations. Therefore, typical GPU-based SDR platforms consist of either a GPU and a GPP, or a GPU, a host GPP and an additional accelerator, such as an FPGA. This subsection reviews some examples of GPU-based SDR platforms under these two categories.

2.3.2.1 GPU with GPP

The GPU cannot work as a stand-alone processor (Figure 2.16); it needs the CPU to initiate and keep track of kernel (functional program) execution [85]. Different bus standards are used to interface one or more GPU devices to the host CPU. All devices from key GPU vendors support the peripheral component interconnect express (PCIe)

bus.

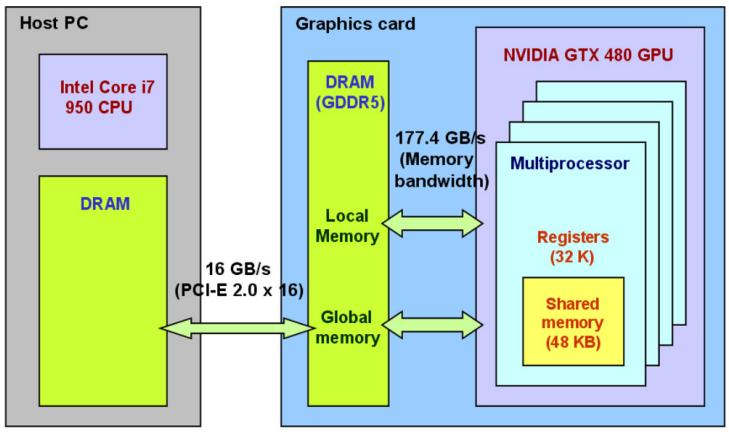

Figure 2.16: CPU and GPU system architecture showing the interconnect bus [84].

A GPU-based SDR platform is used to accelerate MIMO uplink detection in [84]. A parallel GPU-based implementation of the Viterbi decoder is presented in [86]. The GPU-based SDR platform was a normal PC workstation consisting of an NVIDIA 8800 GTX GPU. An implementation of a wideband high-performance channeliser on a GPU-based SDR platform is reported in [87].

2.3.2.2 GPU with FPGA and GPP

CPUs, in addition to GPUs, are used in concert with FPGAs to build high-performance heterogeneous SDR platforms. Each of these platforms possesses their own distinctive strengths, which make them suitable for the efficient implementation of different algorithms of the SDR signal chain. CPUs are excellent for control-intensive tasks, such as managing data transfer to and from both the GPU and the FPGA; GPUs offer very high data-parallelism, whereas FPGAs provide hardware-like performance. Although combining the different computing processors into one heterogeneous SDR platform improves the compute density and efficiency of the platform, it significantly increases the complexity of programming methodologies and tools.

Li [72] reports on the implementation of a single-input single-output orthogonal frequency-division multiplexing (OFDM) WiFi uplink system on a SDR platform, consisting of a CPU, a GPU server and an FPGA. Figure 2.17 shows the architecture of the GPU-based platform used and the associated design methodology.

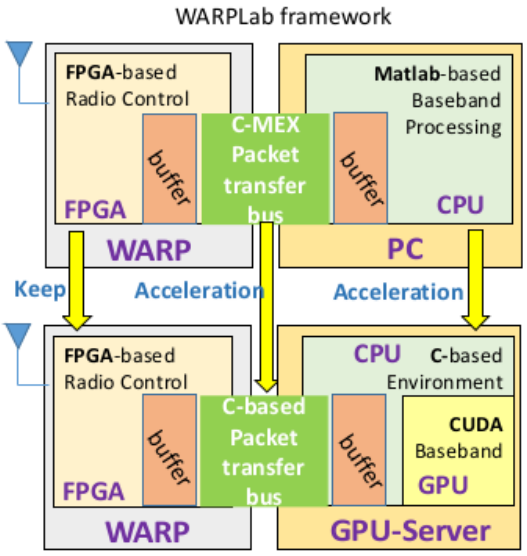

Figure 2.17: System architecture of the WARP GPU-based SDR platform [72].

2.3.3 High-Level Programming Tools

CUDA and OpenCL are the main high-level programming tools for GPUs. These two tools are briefly reviewed in this subsection, and a survey of other high-level GPU programming methodologies is presented in Table 2.3.

2.3.3.1 CUDA

CUDA [27] is a parallel computing platform and application programming framework developed by NVIDIA for general-purpose computing on NVIDIA GPUs. The architecture of NVIDIA GPUs consists of a scalable array of highly threaded Streaming Multiprocessors (SMs). This is illustrated in Figure 2.15. CUDA provides a scalable programming model and infrastructure that grants an application developer access to the GPU's instruction set and to several other parallel computing constructs for development and execution of kernels [27].

CUDA is aimed at enabling GPU designers to easily develop applications that clearly scale their parallelism in order to take advantage of the increasing number of available processing cores. Further, the aim is to provide a relatively simple programming model with a low learning curve, especially for C/C++ programmers.

In order to achieve this, CUDA has three key abstractions at its core – a hierarchy of shared memories, thread groups and barrier synchronization. These are made available to the GPU programmer in the form of minimal extensions to C/C++ standard languages. These abstractions hide the underlying low-level complexity of the GPU hardware and yet allow the designer to transparently explore various granularities of data and thread parallelism in applications. They enable the designer to easily divide the problem into chunks that can be handled independently in parallel by thread blocks, and to divide each sub-problem chunk into finer granules that can be processed in concert by all threads inside the block [88].

2.3.3.2 OpenCL

OpenCL (Open Computing Language) is an application programming interface and tool-flow developed to simplify the process of writing programs for heterogeneous computing platforms. Heterogeneous platforms are systems that use more than one kind of computing device in concert. Devices include a combination of GPPs, DSPs, FPGAs and GPUs. The challenge of writing applications for heterogeneous platforms is that each of the processors is typically parallel and has its own unique parallel programming model. Therefore, a heterogeneous system programmer must be familiar with each of these. OpenCL lends itself to addressing this challenge and also provides a scalable programming model that allows developers to transparently leverage the increasing levels of parallelism

on heterogeneous architectures. OpenCL targets multi-core CPUs, DSPs, FPGAs and GPUs [89].

2.4 HIGH-LEVEL TOOL-FLOWS TECHNIQUES

In basic terms, in the context of digital systems, high-level design refers to modelling and hardware implementations of digital systems from a higher (above RTL) level of abstraction. High-level design is a very active area of research under several key categories, including high-level modelling (2.4.1) and code generation techniques (2.4.2). These techniques are also reviewed in this subsection.

2.4.1 Modelling

Functional specification and modelling of applications using high-level tools can be performed by using model-based, algorithmic, or system-level languages and techniques.

2.4.1.1 Model-based

Model-based design is now an established approach to develop efficient solutions to complex engineering problems. In this method, complicated systems can be created by using mathematical models, which represent system components and their interactions with their surrounding environment. These models have many applications in the design process, including system simulation, stability analysis, and control algorithm design. By introducing advanced, automated code-generation technology, another application of these models has become viable. These models can furthermore be used as the input to an automatic code generation tool. Advanced, state-of-the-art code generators can produce optimized, embeddable C source codes from these models [90].

2.4.1.2 Algorithmic

Typically, the specification for a new circuit is first described at a very abstract level. More particularly, relationships between a set of inputs and a set of outputs are described using a set of computations. This is referred to as an "algorithmic level" design or an "algorithmic specification", and is often described using conventional computer programming languages, such as, for example, C++.

2.4.1.3 System Level

In system-level modelling, a system is specified in such a way that there is no differentiation between software and hardware parts. One generic model encompassing both is written. SystemC is a common system level specification language.

SystemC is a set of C++ classes and macros, which provide an event-driven simulation kernel in C++, together with signals, events, and synchronization primitives, deliberately mimicking the hardware description languages VHDL and Verilog. While such languages are often used for RTL descriptions, SystemC is generally applied to system-level modelling, architectural exploration, software development, functional verification, and high-level synthesis. SystemC is often associated with Electronic System Level (ESL) design, and with Transaction-Level Modelling (TLM) [18].

2.4.2 Code generation

In order to support mapping to standard implementation tools and interfaces, the automatic generation of implementation code is an important requirement of every high-level design tool-flow. Key techniques used in modern high-level tool-flows are high-level synthesis and electronic system-level synthesis.

2.4.2.1 High-Level Synthesis

High-level synthesis (HLS) is a digital design mechanism that automatically maps a behavioural specification to an RTL description. The mechanism was first proposed in the 60s to further automate electronic design and increase productivity. However, it was only in the recent decades that chip designers have shown an interest in HLS due to the exponential increase in design sizes and complexity, according to Moore's Law. In principle,

Table 2.3: State of the art GPU High-level design flows.

License	Tool	By	Approach	Input	Platform	sdk?	compiler?	profiler?	Year	Status
Open source	OpenCL	Khronos Group	device-level API	C/C++	GPU FPGA GPP	No	No	Yes	2009	Active
Open source	PyCUDA	Andreas Kloeckner	API bindings	Python	NVIDIA GPU	All	No	Yes	2008	Active
Proprietary	CUDA	NVIDIA	language integration, device-level API	C/C++	NVIDIA GPU	Yes	Yes	Yes	2008	Active
Proprietary	DirectCompute	Microsoft	device-level API	C/C++	NVIDIA GPU	Yes	Yes	Yes	2008	Active
Proprietary	Brook+	AMD	device-level API	C	AMD GPU	Yes	Yes	No	2007	Active
Proprietary	Peakstream	Peakstream	device-level API	C/C++	ATI (AMD) GPU	Yes	Yes	Yes	2006	Inactive
Proprietary	RapidMind	RapidMind	device-level API	C++	NVIDIA GPU, ATI (AMD) GPU	No	Yes	No	2004	Inactive

modern HLS efforts are similar to earlier HLS efforts. As illustrated in the figure below, high-level synthesis takes a behavioural description of a digital circuit along with performance goals from the user and automatically transforms the specification into a structural RTL design, consisting of a datapath plus a control unit [91, 92, 93].

Figure 2.18 shows in detail the intermediate stages that are involved in high-level synthesis. A description of each is given below:

- **Compilation** - of the source into an internal representation, usually a data flow graph with a control flow graph.
- **Transformations** - of the internal representation into a form more suitable for high-level synthesis. The transformations involve both compiler-like and hardware-specific transformations.
- **Scheduling** - assigns each operation to a time step. Scheduling is sometimes called control synthesis or control step scheduling.
- **Allocation (binding)** - assigns each operation to a piece of hardware. It involves both the selection of the types and quantity of hardware modules from a library and the mapping of each operation to the selected hardware. Allocation is sometimes called data path synthesis or data path allocation.
- **Generation** - produces the design that is passed to logic synthesis and finite state machine synthesis. The final design is commonly generated in an RTL language.

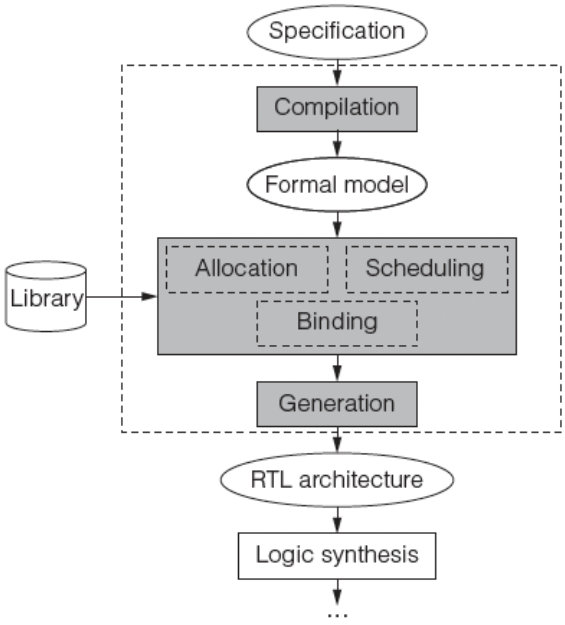

Figure 2.18: High-level synthesis design steps [94].

2.4.2.2 Electronic System-Level Synthesis

Electronic System-Level (ESL) design focuses on building electronic systems from a system-level specification [85]. The system-level of abstraction is above the behavioural level and focuses on a complete system, with its hardware and software parts. ESL compiles with the need for hardware and software co-design, while building upon the legacy hardware design representation. Like HLS, ESL focuses on increasing design productivity to keep up with the increasing complexity and density of modern designs. True ESL synthesis tools demonstrate the ability to combine design tasks within one complete flow that can generate systems across hardware and software boundaries from an algorithmic system specification [18].

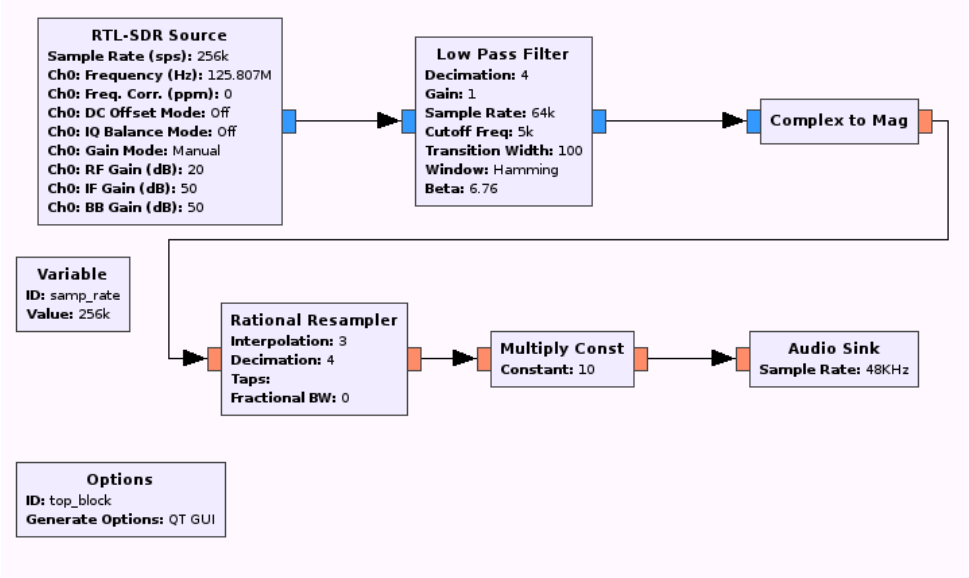

Figure 2.19: GNU Radio design flow architecture [1].

2.5 HIGH-LEVEL TOOL-FLOWS IN SDR

This section reviews existing high-level tool-flows that can be used for developing SDR DSP applications (2.5.1), and reviews related past comparative tool-flow evaluation studies (2.5.2) and key standardisation work done with regard to high-level design methodologies for SDR DSP (2.5.3).

2.5.1 Existing Tools

2.5.1.1 GNU Radio

GNU Radio [1] is a software toolkit for implementing software-defined radios on general purpose microprocessors. The toolkit consists of an extensible library of communications and signal processing blocks, a graphical frontend (GNU Radio Companion) for constructing SDR models, and a scheduler that manages model execution [95].

GNU Radio's supplied performance-critical signal processing blocks are primarily implemented in C++. Python is used to construct SDR application models by gluing multiple blocks together into signal flow graphs that implement a designer's desired functionality (e.g. FM receiver). Simplified Wrapper and Interface Generator (SWIG) is used to provide an interface between Python and C++ blocks. Code generation is supported only for models specified graphically, using the GNU Radio Companion (GRC) graphical tool. The GRC uses Cheetah templates to automate generation of Python source code from graphical flow graphs. The generated Python code declares blocks, glues them together and invokes the scheduler to manage execution of the user's SDR application on a general purpose processor. While the GNU Radio scheduler is data driven, it is complex and not based on any specific formal models of computations for SDR implementation such as synchronous data flow (SDF) [95].

2.5.1.2 OptiSDR

OptiSDR is a domain-specific language and tool-flow aimed at facilitating development and implementation of SDR applications on heterogeneous computing architectures, comprising GPPs, GPUs and FPGAs. OptiSDR is planned around a simple but highly object-oriented programming structure. OptiSDR adopts Lightweight Modular Staging (LMS), presented in Delite for a runtime code generation approach. The LMS framework provides to Delite, and eventually upwards in the hierarchy to OptiSDR, a library of core components for building high-performance code generators [63].

Delite uses LMS for code generation, hence OptiSDR builds upon the same underlying technique provided in

the framework. LMS provides a library of core components for building high-performance code generators and embedded compilers in Scala.

In Delite, code generators built using LMS include generators for C++, CUDA-C, OpenCL, and Scala. OptiSDR possesses CUDA code generation for SDR constructs, such as vector/matrix data types (e.g., the DenseVector, type), and test data generators such as sine, sum, and cos, that we used to implement our DSP algorithms and to generate test data.

2.5.2 Past Tool-Flow Surveys

This sub-section presents related literature material that focuses on evaluation of high-level tool-flows for FPGA and GPU platforms. The aim is to ensure that the various key aspects of the comparative study carried out in this dissertation, such as tool-flow selection and evaluation criteria draw from a large body of work presented in related tool-flow studies. Therefore, studies presented in this sub-section were selected based on their usage of high-level tool-flows, type of test platform (must be either GPU and/or FPGA) and usage of clear comparison criteria. The application domain was not a major deciding factor and thus surveys outside of the SDR domain are considered as well as they can provide methods, techniques and concepts that can be re-used in a different domain.

A summary of the past comparative studies of high-level tool-flows is presented in Table 2.4. Most of the published studies focus on FPGA platforms and GPUs are second on the list. In terms of the tools, FPGA-based studies cover a wider variety of tool-flows than GPUs. The majority of the GPU studies are based on CUDA and OpenCL tool-flows whereas FPGA-based studies focus mainly on tool-flows based on C/C++ or dialects thereof. No study is reported in the literature that focuses on the Python-based tool-flows.

2.5.2.1 Performance Measurement

In [10], Bluespec and Catapult-C high-level design tools are quantitatively compared in the development of algorithmic IP. The Reed-Solomon decoder is used as a case study. The tool-flows are compared in terms of productivity, area and performance metrics. Source lines of code (SLOC) are used for simple productivity measurement. Synthesis process results, including number of look-up tables, flip flops, block RAMs and equivalent gate count, were used for design area measurement, whereas maximum achievable frequency, throughput and data rate were used as performance metrics. The study concludes that Catapult is suited for fast prototyping of hardware designs, where fine-tuned performance is not a goal. Bluespec, in contrast, offers well-defined semantics that enable the development of high-performance hardware, while keeping the benefits of high-level abstraction.

In [96], a performance comparison of CUDA and OpenCL is present, using complex, near-identical adiabatic quantum algorithms kernels. The performance metrics used are: GPU operation time, end-to-end running time, kernel running time and data transfer time. The study finds that CUDA out-performs OpenCL in terms of data transfer performance. The execution of CUDA kernels was also consistently faster than that of OpenCL and, in conclusion, the study recommends CUDA as a more suitable choice for applications with high-performance goals.

Different metrics are used to measure the quality of designs developed using the tools and the efficiency of these tools. Performance metrics, including logic utilization for FPGAs and timing for both FPGAs and GPUs, are used to measure the quality of the designs. Formal models based on the SLOC are used to compute the productivity of the programming associated with the tool-flows.

2.5.2.2 Design Productivity Measurement

In [97], a generic empirical method is presented to evaluate the design productivity of a wide variety of high-level design tool-flows. The proposed approach is based on both design efficiency and implementation quality. Design efficiency is measured in terms of system development time and code properties. Non-recurring engineering (NRE) design time, NRE verification time and system integration time are the metrics used under system development time, whereas SLOC and number of characters in the code are used to evaluate code properties. Conversely, implementation quality is measured in terms of the number of lookup tables (LUTs), registers, RAM blocks, DSP cores and processing latency and minimum operating period. In order to demonstrate the design productivity scheme, the study compares an HLS compiler based on CAPH with hand-written HDL, using an interpolation

filter case study. A design productivity increase of 2.3x is measured for the high-level tool-flow.

In [98], the basic relationships between a number of various development variables, such as size, effort, size of staff and productivity, are examined. In this paper, design productivity is measured as the ratio of delivered SLOC to the total effort (in man months) needed to develop the software application.

2.5.3 **Standardisation**

Standards are necessary in SDR DSP applications development to guarantee interoperability among design tools, applications and hardware platforms. Therefore, support for key SDR standards should be part of the requirements of the ideal high-level SDR DSP tool-flow specification that will be defined in Chapter 4 of this dissertation and used as a criteria to compare the tool-flows in Chapters 5 and 6. This subsection reviews key standards relating to the design of applications in the context of SDR.

*2.5.3.1 **SCA***

The Software Communications Architecture (SCA) is an open architecture and framework created by the US Department of Defense (DoD) to provide a standardised approach for the development and deployment of SDR waveforms. The goals of SCA are to improve waveform portability and interoperability among communication instruments and also to achieve higher design productivity. The SCA achieves these goals by defining a standardised way of ensuring that waveform software development is independent from the underlying hardware.

Figure 2.20 shows the architecture of the SCA standard. The SCA defines the following key components:

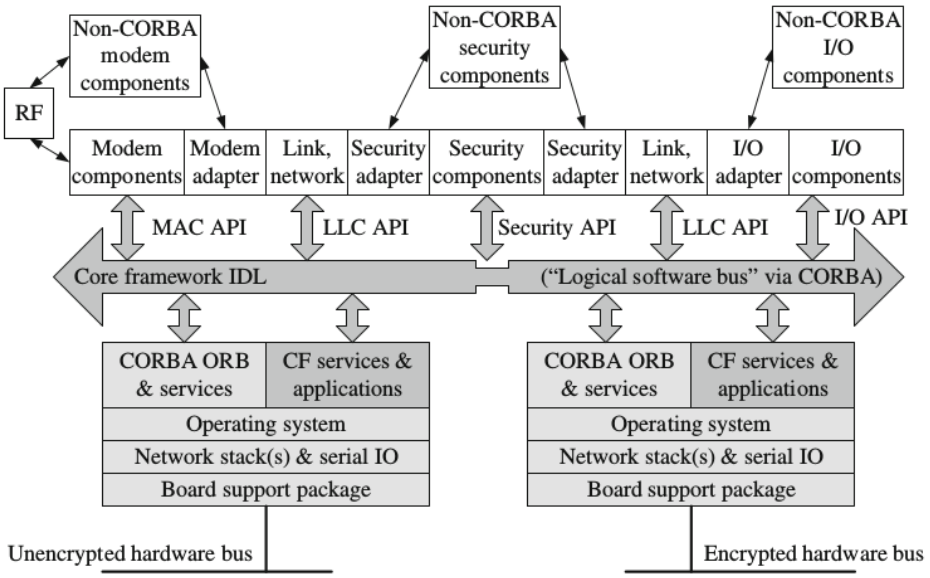

Figure 2.20: Conceptual architecture of the SCA SDR standard (lightly shaded blocks are custom off the shelf, darker one represent the SCA core framework, unshaded blocks are application-specific) [3].

- Operating system - a real-time operating system (RTOS) to manage the hardware on which the radio is deployed. The RTOS should provide POSIX support.

- Middleware layer - for interfacing the various elements of the radio together, including managing transfer of data between them. CORBA is one of the main software architectures used by the SCA to enable communications between different software developed using different languages and running on different processors.

Table 2.4: Past studies involving comparative evaluation of high-level tool-flows

Year	Tools	Hardware	Domain	Quantitative	Case study	Metrics	Qualitative	Ref
2009	Bluespec, Catapult C	FPGA	DSP	Yes	Reed Solomon decoder	SLOC, FPGA resource utilization, Frequency, Throughput, Data rate	No	[10]
2010	System generator, Bluespec	FPGA	Image processing	Yes	Convolution filter	SLOC, Logic utilization, Max frequency	Yes	[99]
2010	CUDA, OpenCL	GPU	Computational Science	Yes	Computational Science Adiabatic QUantum Algorithms	End-to-end running times, kernel running time, data transfer time	No	[96]
2012	CUDA, OpenCL	FPGA	Neural networks	Yes	Hodgkin–Huxley and Morris–Lecar models	Computation time, communication time, other overhead time	No	[100]
2012	AccelDSP, Agility Compiler Bluespec, Catapult C, Compaan, C-to-Silicon, Cyberworkbench, DK Design suite, Impulse C, ROCC, Synphony C	FPGA	Image processing	Yes	Sobel edge detector, WLAN baseband processor	Logic Utilization	No	[9]
2014	AccelDSP, Agility Compiler Bluespec, Catapult C, Compaan, C-to-Silicon, Cyberworkbench, DK Design suite, Impulse C, ROCC, Synphony C	FPGA	Computational Science	Yes	Tri-Diagonal Matrix Algorithm	Logic utilization, run time, solution error	No	[37]
2015	OpenCL, Vivado	FPGA	Machine Vision	Yes	Disparity map calculation	FPGA resource utilization	No	[101]
2016	OpenCL, Vivado	FPGA	General	Yes	Benchmark suite (17 kernels)	FPGA resource utilization	No	[11]
2017	CUDA, OpenCL, OpenMP, OpenACC	GPU	General	Yes	SPEC benchmark suite	programming productivity, execution time, communication time, energy consumption	No	[12]

- Interface definition language (IDL) - a group of interface definition language classes are used to provide abstract interfaces to various components that comprise a radio instrument. A combination of these IDL classes, and their implementation code constitute the core framework (CF).

- Domain profile - this is an XML framework that provides the capability to specify radio components, their inter-connections and functional properties.

- Application programming interfaces (APIs) - APIs for common interfaces such as Ethernet, Video and USB.

A thorough review of the SCA is beyond the scope of this project. The reader is encouraged to consider [102, 103] for in-depth SCA reviews.

*2.5.3.2 **STRS***

SDR is also used for satellite and space applications. One of the key design goals for space instruments is long life cycles. Therefore, space-based SDRs should also be built to support long life cycles. Providing easy to use and effective techniques for modifying the functionality of the space radio can lengthen the mission life, increase data collection and also prevent a mission failure. SDR has been a subject of study in the National Aeronautics and Space Administration (NASA) for more than a decade, and a need for a standard has been realised. Generally, commercial platforms are 10 years ahead of the hardware that is used for space instruments. Therefore, space radios, compared to terrestrial ones, have higher processing infrastructure limitations. The SCA is not suitable for most NASA space missions due to its large size and high computational demands. Therefore, NASA has created a light standard for space SDRs, called the 'Space Telecommunications Radio System' or STRS [104].

Figure 2.21 shows the architecture of the STRS standard.

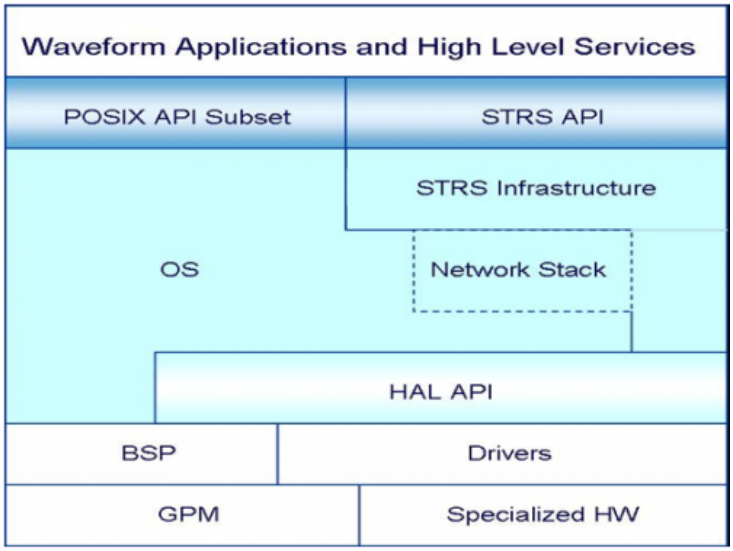

Figure 2.21: STRS layered architecture [105].

2.6 CHAPTER SUMMARY

This chapter reviewed the literature in relation to various concepts, including SDR, FPGAs, GPUs and high-level design. The review of the hardware architecture and application requirements of SDR DSP platforms and operations, presented in 2.1 has provided a solid theoretical framework. This is necessary for specifying the requirements of the ideal high-level SDR DSP tool-flow system in Chapter 4 and for choosing a suitable example SDR application to use in evaluating the tool-flows in Chapters 5 and 6.

Sections 2.2 and 2.3 reviewed the two main implementation devices of SDR DSP operations, viz. FPGAs and GPUs, which are the focal target devices of the tool-flow comparisons carried out in this dissertation. For both devices, the review included aspects pertaining to both architectural design and a sampling of key platforms and programming methodologies used. The content covered in these sections is also used to inform the process of formulating holistic comparison criteria for the tool-flows.

Further, high-level design techniques and tools were reviewed in sections 2.4 and 2.5. The best techniques for use in the high-level modelling and code-generation stages of the high-level design methodology were identified and reviewed. A comprehensive review of past studies involving tool-flows comparisons for both FPGAs and GPUs was presented. This review identified the approaches used and the tools considered in the comparisons.

In the next chapter, we will describe the research methodology that is adopted in this dissertation to realise the original aim of this project to perform a comparative study of high-level tool-flows for rapid prototyping SDR DSP applications on FPGA and GPU platforms.

CHAPTER 3

METHODOLOGY

The aim of this dissertation is to conduct a comparative study of tool-flows for the rapid prototyping of SDR DSP operations on FPGA and GPU platforms. The main motivation is the proliferation of different high-level design tool-flows aimed at addressing the current design productivity gap affecting modern high-performance computing hardware. The availability of numerous different high-level tool-flows necessitates their comparative study. This study will help to reveal the various strengths and weaknesses of existing tools so as to inform tool choices and identify the areas in high-level design technologies that still require improvement. Due to various benefits provided by open-source software including cost effectiveness, freedom in use, distribution and modification, the study focuses mostly on open-source tool-flows. In addition, only FPGA and GPU architectures are considered. However, the methodology is such that the study can easily be extended to other SDR DSP processors, such as multicore GPPs and DSP processors. As discussed in the literature review (section 2.5), there are many high-level tools to be considered in this study. Instead of looking at all of them, however, the approach in this dissertation is to select and study only two tools for FPGA platforms and an additional two for GPU platforms.

The specific steps undertaken in this dissertation to achieve the objectives which were outlined in Section 1.3 are:

1. Developing comprehensive criteria to use in comparing the tools
2. Performing a study of tools for rapid prototyping of SDR DSP on FPGA platforms
   (a) Selecting tools to study
   (b) Evaluating the features of each of the tools against the criteria
   (c) Implementing a case study application in each of the tools
   (d) Evaluating each of the tools in terms of the case study application
3. Performing a study of tools for rapid prototyping of SDR DSP on GPU platforms
   (a) Selecting tools to study
   (b) Evaluating the features of each of the tools against the criteria
   (c) Implementing a case study application in each of the tools
   (d) Evaluating each of the tools in terms of the case study application

The sections that follow will expand on each of these steps to describe and discuss the rationale behind the techniques, design processes, tools and metrics used. Section 3.1 thus presents the systematic methodology used to develop the criteria for comparison. Sections 3.2 and 3.3 describe the methodologies followed to evaluate FPGA and GPU tool-flows respectively.

3.1 SYSTEMATIC DEVELOPMENT OF THE COMPARISON CRITERIA

A motivation for developing comprehensive criteria for comparing tool-flows was given in section 1.2. This section outlines the systematic approach followed to develop the criteria.

3.1.1 System Engineering Process

A system engineering approach is used in this dissertation to develop a comprehensive comparison criteria, in the form of an ideal high-level tool-flow specification, for the tool-flows. The ideal tool-flow specification is aimed at defining clearly, and in detail, the key design goals and features that should be incorporated in standard SDR DSP design tools for FPGA and GPU platforms respectively.

System engineering (SE) is a holistic interdisciplinary approach that enables the building of successful systems. A system is a set of integrated elements that work together to perform a particular purpose, and successful systems are those that best satisfy the purposes and needs for which they have been built [106]. In this dissertation, the ideal high-level tool-flow for rapid prototyping of SDR DSP operations is the system under consideration. A system engineering process is thus followed to design a holistic specification for the ideal tool-flow system. As shown in Figure 3.1, the process is comprehensive, iterative and recursive. It includes the following main steps which are each supported by system analysis and control tools:

- **Requirements Analysis** - this is the first step in terms of the System Engineering Process. The purpose of this step is to develop functional and performance requirements for the system; the step translates stakeholder requirements into a set of requirements that describe what the system must do, and how well it must do it. The requirements must be understandable, unambiguous, comprehensive, complete and concise [107].
- **Functional Analysis/Allocation** - this step is concerned with a functional analysis of the system functions identified in the requirements analysis step. Functional analysis is performed by refining higher-level functions identified during requirements analysis into lower-level functions. Performance requirements of the higher-level requirements are assigned to lower-level functions. The output of the functional analysis is the functional architecture of the system. The system's functional architecture provides a logical description of what the system does, and how well it does it [107].
- **Design Synthesis** - this step focuses on defining the system in terms of the components, which together constitute the system. The components might include physical and software elements. The output of the synthesis step is typically called the physical architecture of the system [107].

The following subsection (3.1.2) describes the starting requirements which are used in Chapter 4 as inputs to the system engineering process that produces the ideal tool-flow specification which is subsequently used in Chapter 5 and 6 as part of the criteria for evaluating FPGA and GPU tool-flows respectively.

3.1.2 Ideal Tool-Flow Requirements

A review of the literature has revealed that there is currently no global standard that defines what constitutes an ideal high-level tool-flow for development of SDR waveforms. Key efforts in this direction include:

- the high-level synthesis design methodology for digital hardware design, targeting devices, such as FPGAs. A review of high-level synthesis is given in Section 2.4.
- the electronic system-level design methodology, which targets multiple different computing processors, including FPGAs, GPUs, and multicore GPPs from one high-level input specification. Section 2.4 covers an overview and review of the ESL methodology.
- the software communications architecture framework, which aims to enable the implementation of SDR waveforms on heterogeneous platforms, comprising several different processors, such as GPPs, DSPS, or FPGAs. The SCA framework for SDR is reviewed in Subsection 2.5.3.

In order to improve the objectivity of the comparative study, a specification of the ideal high-level tool-flow is developed in this dissertation and used as the baseline criteria in investigating the tools. The starting requirements of the ideal tool-flow were elicited through email and interview correspondence, and by reviewing the literature on the subject. In addition, in line with the system engineering methodology, a list of key system stakeholders were first identified and then the starting requirements elicited from each of the stakeholder groups. A tabular listing of all stakeholder requirements, from the four stakeholder classes, is provided in Table 3.1 below.

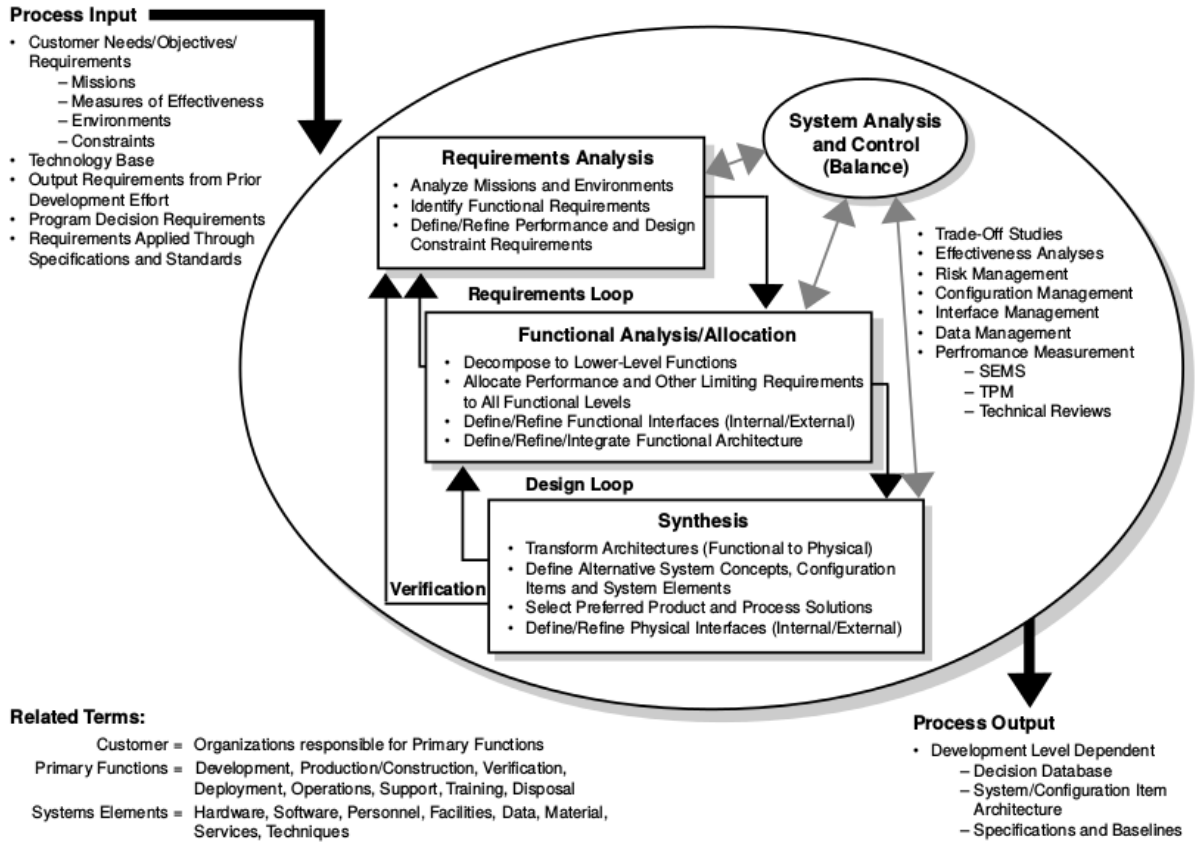

Figure 3.1: The System Engineering Process [108].

Table 3.1: Stakeholder Requirements with regard to High-Level FPGA SDR DSP Tool-flow.

Stakeholder	**Requirement**	**ID**
End-Users (EU)	Should be easy to use	EU-R1
	Should be reliable	EU-R2
	Should produce portable designs	EU-R3
	Should support design reuse	EU-R4
	Should abstract low-level hardware details	EU-R5
	Should detect design errors early in the development cycle	EU-R6
	Should deploy user SDR waveform on a hardware platform	EU-R7
	Should generate bit-accurate models for FPGA designs	EU-R9
	Should support automatic design optimisations	EU-R10
	Should generate quality designs	EU-R11
Research & Development (RD)	Should support design reuse	RD-1
	Should generate quality designs	RD-2
	Should easily integrate with third-party SDR waveform design tools	RD-3
	Should be based upon established design techniques and technologies	RD-4
	Should be designed for easy upgrading and customisations	RD-5
Hardware Vendors (HV)	Should generate portable designs	HV-1
	Should support automatic design optimisations	HV-2
	Should support one-click generation of implementation code	HV-3
Standardisation (SB)	Should produce portable designs	SB-1

Continued on next page

Table 3.1 – *Continued from previous page*

Stakeholder	Requirement	ID
	Should support multiple high-level design paradigms Should easily integrate with other third-party SDR waveform design tools	SB-2 SB-3
	Should be based upon relevant established SDR and EDA standards	SB-4

The system engineering process, documented in both Chapter 4 and Appendix A is followed to transform the above requirements into a system specification that consists of functional and non-functional system requirements and a logical architecture of the ideal high-level tool-flow system for rapid prototyping of SDR DSP on SDR platforms, including FPGAs and GPUs.

3.2 FPGA TOOLS STUDY PROCESS

Two main categories of high-level tool-flows are considered for the comparative study in this dissertation, namely, FPGA and GPU tool-flows. The project methodology is also branched to cover the two categories of the study. This section begins with a description of the key elements of the process followed to study the FPGA tools.

Subsection 3.2.1 explains which FPGA tools were selected for study, out of many existing high-level tool-flows, and why they were chosen. Subsection 3.2.2 introduces the SDR DSP application that is used as a case study for the FPGA tool-flows in this dissertation. Subsection 3.2.3 presents the software development process used in developing the case study application, and Subsection 3.2.4 describes the configurations and metrics used to evaluate the design efficiency and design quality of each tool-flow in terms of the case study application.

3.2.1 Tools Selection

A comprehensive list of existing high-level tool-flows is presented in Section 2.5 of the literature review. This study focuses only on a few selected open-source tools, namely, MyHDL and Migen. Due to time constraints, the number of tool-flows to be studied is limited only to two out of over ten tools available in the literature. Further, due to various benefits provided by open-source software including cost effectiveness, freedom in use, distribution and modification, the study focuses mostly on open-source tool-flows. Migen and MyHDL are among the most used open-source high-level digital hardware design tools and though both based on the Python software language, they follow different approaches when it comes to high-level design and therefore have been selected for evaluation in this dissertation.

Migen and MyHDL are among key open-source design tools targeted at improving adoption of FPGA technology by stream-lining the traditional FPGA design process through introduction of high-level, software-like design techniques. Both tools are sufficiently mature and have been used to develop various hardware design applications in both academia and industry [77, 74, 109, 110]. In addition, they are based on different high-level digital hardware design techniques and thus are a suitable choice as they provide an opportunity to study and compare different high-level FPGA design techniques. There are not many open-source high-level design tools targeting FPGAs in the literature. Besides Migen and MyHDL, other notable tools include LegUp [111] and Chisel [21].

3.2.2 Case Study Selection

In addition to evaluating the tools by investigating their features against the ideal tool-flow specifications, the approach is also to select and implement a case study SDR DSP application, using each of the tools being studied. According to the review of the SDR architecture presented in Subsection 2.1.2 of the literature review, the programmable DSP subsystem constitutes all the operations of the SDR system that can be implemented on programmable platforms, including FPGAs and GPUs. The operations, from which a case study can be selected, include filters, mixers, digital synthesisers, FFT, demodulators, and codec algorithms, such as Reed Solomon and

Viterbi. A FIR filter operation is selected and used as a case study in this dissertation.

A FIR filter is chosen because, although not the most fundamental, digital filtering is certainly the oldest discipline in the field of DSP and thus SDR waveform development [66]. Digital filtering is so widespread that the quantity of literature pertaining to it exceeds that of any other topic in DSP. Unlike other key SDR algorithms, such as the Viterbi decoder and the OFDM demodulator, as shown in subsection 2.5.2, a FIR filter is widely used in similar tool-flow studies and is representative of typical operations found in many SDR algorithms.

The specification of the FIR filter used for the case study is shown in Table 3.2 below. The FIR coefficients that were used to realise the above FIR filter, were generated using the Remez tool in Python. The originally generated coefficients are floating-point numbers and are converted into fixed-point for FPGA deployment. Floating-point coefficients are avoided, mainly because, although they enhance precision, more FPGA resources are required to handle floating-point arithmetic. Furthermore, algorithmic precision is not a goal in this study.

Table 3.2: FIR filter specification.

Design criteria	**Specification**
Type	Lowpass
Passband	0 - 8MHz
Stopband	8 - 10MHz
Order	63
Passband ripple	2.3%
Stopband attenuation	40dB

3.2.3 **Gateway Development Process**

This project follows a version of the waterfall methodology (shown in Figure 3.2) that has been adapted for gateware development but follows the similar structure. In particular, implementation of unit testing involves functional simulation prior to testing on hardware. Several studies have used adapted versions of the waterfall process for gateware development [112, 77, 113].

Firstly, the case study requirements are gathered. This entails characterising the FIR filter waveform to be developed by describing its key properties, such as number of taps, data size, filter type, sampling frequency, bandwidth, pass-band and stop-band. The specific FIR filter case study requirements are described in 3.2.2.

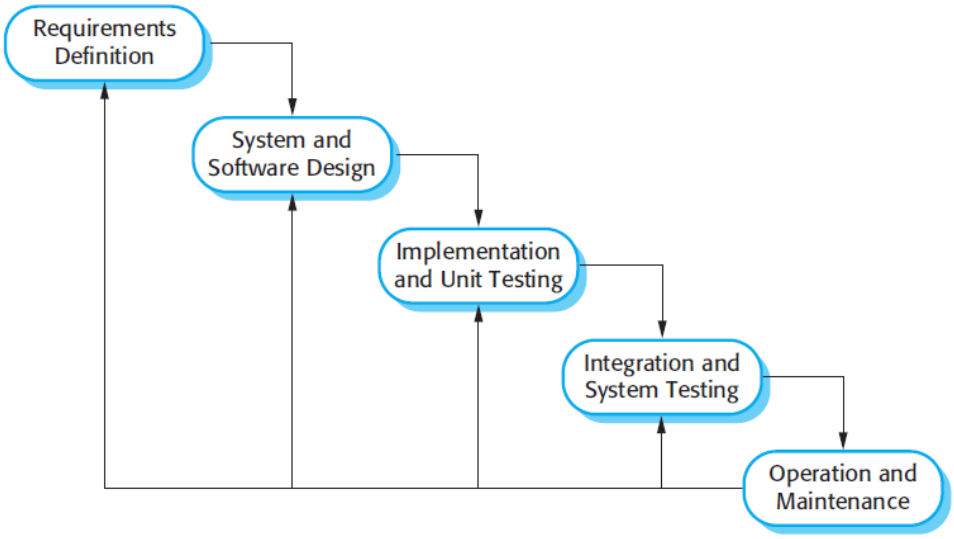

Figure 3.2: Waterfall software/hardware development process [114].

A **gateware design specification** of the FIR filter case study application is then produced from the initial requirements. This stage entails analysis and elaboration of the original software requirements to produce a software design specification consisting of functional and non-functional requirements. The online FIR filter design tool known as TFilter [115] was used to generate the filter coefficients according to given filter requirements. In addition, an algorithmic design of the serial FIR filter operation and the associated test bench was developed.

The **implementation** of the case study design begins once the design specification has been completed. The FIR filter specification developed in the design specification stage was implemented, using MyHDL and Migen separately. The development machine used was a Lenovo Y500 laptop with an Intel Core i7 processor and running Ubuntu 14.04 LTS, MyHDL version 1.0dev, Migen 0.5dev and Python 3.4.3. Once implemented, each design was **tested** by means of functional simulation, and implemented on a Xilinx Virtex 7 FPGA using the ISE 14.7 tool.

Ongoing **maintenance** of the implementation was carried out to fix the bugs. All the MyHDL and Migen source code written to develop and test the FIR filter case study waveform was stored on and maintained from an online Git repository.

3.2.4 Tool-Flow Evaluation Process

Besides the qualitative study guided by the ideal high-level tool-flow, the following metrics are used in this study to evaluate the resultant quality of the implementation designs produced from and the associated design productivity of each tool. Logic utilisation, power consumption and speed are the metrics used to measure the quality of the FPGA designs. Coding effort is then used to estimate the design productivity associated with each tool. These metrics are used because they are well understood and, as shown in Subsection 2.5.2 of the literature review, are commonly used to assess the quality of FPGA designs.

3.2.4.1 Design Productivity

Empirical measurement of software design productivity is a complex problem and an active area of research. Several approaches are used. The various schemes are reviewed in Subsection 2.5.2 of the literature review . The number of source lines of code (SLOC) is the most commonly used metric. It has the advantage of being well-defined and easy to repeat. It moreover provides an approximate measure of the size of a program. However, using the SLOC metric alone to measure design productivity is not enough. Additional measurable factors that affect design productivity include total design time, verification time and the quality of the final design. Good design productivity cannot merely be based on how fast a design can be prototyped, but the quality of the developed

design is also equally important. The method of measuring design productivity proposed and demonstrated in [97] is used in this dissertation to measure the design productivity of the FPGA tool-flows. Compared to other schemes, the selected one is comprehensive, easy to follow and is repeatable. The scheme defines design productivity (P_D) as a trade-off between design efficiency (G_{NRE}) and quality (L_Q) as show through the equations below.

$$G_{NRE} = \frac{t_{verif}^{HDL} + t_{verif}^{HDL}}{t_{design}^{HLT} + t_{design}^{HLT}} \tag{3.1}$$

$$L_Q = \frac{\alpha_1 \times lut_{design}^{HDL} + \alpha_1 \times ff_{design}^{HDL} + \alpha_1 \times sr_{design}^{HDL} + \alpha_1 \times dsp_{design}^{HDL}}{\alpha_1 \times lut_{norm}^{HDL} + \alpha_1 \times ff_{norm}^{HDL} + \alpha_1 \times sr_{norm}^{HDL} + \alpha_1 \times dsp_{norm}^{HDL}} \tag{3.2}$$

$$P_D = \frac{G_{NRE}}{L_Q} \tag{3.3}$$

Where t_{design}^{HDL} and t_{verif}^{HDL} are reference measures of time spent manually writing the FIR filter module and the associated test-bench respectively. t_{design}^{HLT} and t_{verif}^{HLT} are measures of time spent developing - using each of the high-level tool-flows (Migen or MyHDL) - the FIR filter module and test-bench respectively. lut_{design}^{HDL} , ff_{design}^{HDL}, sr_{design}^{HDL} and dsp_{design}^{HDL} are the number of LUTs, flip flops, slice registers and DSP cores obtained from a high-level tool-flow. lut_{norm}^{HDL} , ff_{norm}^{HDL}, sr_{norm}^{HDL} and dsp_{norm}^{HDL} are the number of LUTs, flip flops, slice registers and DSP cores obtained from an FPGA implementation of the reference manually written HDL design. $\alpha_i s$ are normalising coefficients and are all set to 1 in this study.

*3.2.4.2 **Operational Efficiency***

We use logic resource utilisation, speed and power consumption to characterise the performance of the designs generated using the study tools. These are commonly used metrics when quantifying the operational efficiency of an FPGA programming methodology[[116]. Resource utilisation is given in terms of the number of slice registers, slice LUTs, IOBs and DSP cores required by a design.

The first two metrics, logic utilisation and operating frequency (speed), can be obtained easily from the output of the place and route tools. Throughput is calculated by simple analysis or by testing after deployment [116]. Power consumption can be measured using Xilinx Power Estimator (XPE) [117]. XPE is tightly integrated with Xilinx ISE design suite to enable designers migrate designs for power estimation.

3.3 GPU TOOLS STUDY PROCESS

This dissertation also conducts a comparative study of tool-flows for rapid prototyping of SDR DSP operation on GPU platforms. As explained earlier, the approach followed is, firstly, to select a sample of high-level GPU tool-flows to consider for the evaluation, and to evaluate them against the ideal specification; thereafter, a suitable case study application is selected and implemented, using the tools for design effort and performance tests. This section describes this study approach in detail.

Subsections 3.3.1 and 3.3.2 presents the tools and case studies selected respectively. Subsection 3.3.3 describes the software methodology used to implement the case study application, whereas the evaluation metrics and tools used to evaluate the coding effort and performance of the tool-flows are described in Subsection 3.3.4. .

3.3.1 Tools Selection

Subsection 2.3.3 presented a comprehensive list of existing high-level tools for GPU platforms. Due to time constraints, this study focuses only on few selected tools, namely, CUDA and OpenCL. CUDA [27] is the most commonly used development tool-flow for GPU programming. OpenCL [29] is a framework for developing programs that can be executed across heterogeneous platforms, consisting of CPUs, GPUs, DSPs and FPGAs.

3.3.2 Case Study Selection

GPUs are used to implement a variety of radio functions at different stages of the SDR programmable subsystem. For examples, GPUs have been used to implement decoders [86, 118, 119], detectors [120, 121, 122] and channelisers [123]. A FIR filter case study is used to evaluate the GPU tool-flows. Digital filters, especially FIR filters, are integral components in SDR waveform detectors and channelisers. In addition, unlike other key SDR algorithms, such as the Viterbi decoder and the OFDM demodulator, a FIR filter is simple enough to be implemented quickly, using the different high-level tools being investigated in this work while also being representative of the key features found in many SDR algorithms.

3.3.3 Software Development Process

The waterfall software development methodology is followed to design, implement and test the FIR filter case studies in each of the tool-flows. The methodology is described in Subsection 3.2.3.

The first step entails defining the **requirements** of the FIR filter waveform. The required characteristics of the FIR filter are similar to those of the FPGA tool-flows studies, which were presented in Table 3.2. The FIR filter should be a low-pass with 80MHz bandwidth, 30 taps and 2.3% passband ripple.

Once the requirements have been defined, they are transformed into an implementable **specification** through a process of analysis and design. In this project, a FIR filter analysis and design tool called TFilter [115] was used to generate the filter specification in the form of coefficients that produce the desired FIR filter response. These coefficients were stored in floating-point format. Unlike FPGAs, GPUs are excellent floating-point arithmetic processors, and thus there is no need to convert the coefficients to fixed-point format. In addition to the coefficients used to specify the functionality of the filter, Equation 2.3 also provides a reference mathematical specification of the serial operation of the FIR filter.

With the specification ready, the FIR filter waveform is **implemented** according to the given specification, using CUDA and OpenCL high-level GPU tool-flows. The implementations were done on a GPU server with the following hardware: an Intel (R) Core i7 960 @ 3.2GHz processor with 24GiB DIMM RAM, 33MHz PCI Express bus and 2 x GTX Titan-X GPUs with 12GB memory. The server had the following tool-flow configurations: Version 9.1.85 of the CUDA compilation tools and Version 1.2 of the OpenCL driver. The tool-flows were used to compile, deploy and **test** the different FIR filter waveforms on the GPU server. The performances of the respective waveforms was measured over various input samples sizes.

An ongoing **maintenance** of the implementations was carried out to fix the bugs. All the CUDA and OpenCL source code written to develop and test the FIR filter case study waveform were stored on and maintained from an online Git repository.

3.3.4 Tool-Flow Evaluation Process

This dissertation is a comparative study of high-level tool-flows for fast prototyping of SDR DSP operations on both FPGA and GPU platforms. Section 3.2 has outlined the methodology that was used to evaluate the FPGA tool-flows using the FIR filter case study application. Section 3.3 deals with the study of GPU tool-flows; it describes the methodology, comprising both the process and the measurement metrics, which was used in this dissertation to study the GPU tool-flows. The tool-flows are evaluated in terms of the efficiency of their design processes and also with regard to the quality of the designs that they produce. Subsection 3.3.4.1 outlines the approach and the metrics used to study the design efficiency of each tool-flow empirically. Subsection 3.3.4.2 focuses on the approach and the metrics for studying the performance of the designs generated using each tool.

3.3.4.1 Coding Effort

A simplified coding effort measurement scheme based on SLOC was used.

3.3.4.2 Operational Efficiency

Operating efficiency quantitatively characterizes the results of a programming effort. Kernel execution time, memory transfer time, throughput and power consumption are used to measure and compare the operational efficiency of each of the tool-flows in this dissertation.

Three main test configurations that are set up are: i) cache warm-up, ii) performance testing under paged memory, iii) performance testing under pinned memory. Benchmarking measurements should be made under steady-state conditions as far as possible, to avoid erroneous results due to external conditions. For example, CPUs generally suffer from initial overhead due to "cold" caches and transactional lookaside buffers (TLBs), which result in expensive memory access misses. Therefore, when performing benchmark studies, it is necessary that a "warm-up" test be done to ensure that the system is under steady-state. Warm-up tests were run for CUDA and OpenCL before taking any benchmark measurements.

The first benchmark test measures the performances of CUDA and OpenCL in terms of total execution time, kernel execution time, memory transfer time and power consumption under paged host memory configuration.

The GPU does not have direct access to pageable host memory. Instead, GPU requests for data transfers from the pageable host memory first entail the allocation of temporary "pinned" host memory by the GPU programming tool-flow, to which host data from the pageable memory is copied and then transferred from the "pinned" memory directly to the GPU memory. Figure 3.3 illustrates pageable and pinned data transfers.

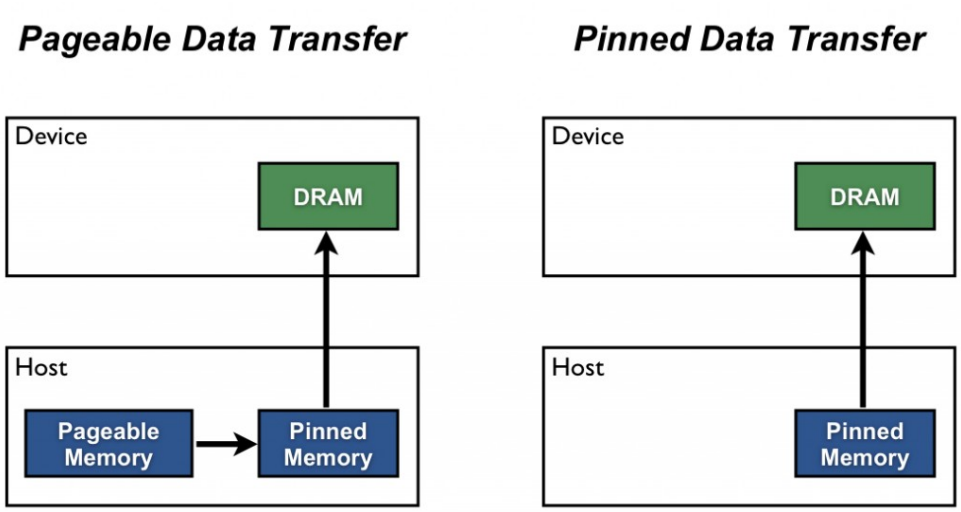

Figure 3.3: Pageable and pinned data transfers from host to GPU (Image source: [124]).

The second benchmark test considers the FIR filter algorithm under pinned host memory configuration. The test measures the performances of CUDA and OpenCL in terms of total execution, kernel execution time, memory transfer time and power consumption.

We use the execution time (T) and the Energy (E) to express the performance and energy consumption. T is defined as the total amount of time that an application needs from the start till the end of the execution, whereas E is defined as the total amount of energy consumed by the system (including host CPUs and accelerators) from the beginning until the end of the execution.

For CUDA, all the operational efficiency measurements, except power consumption, were done using the NVIDIA profiler (nvprof) [125] tool. The nvprof profiling tool enables the collection and viewing of profiling data via the command-line. It supports the collection of profile data for CUDA-related tasks on both the host and the device, such as kernel execution and memory transfers. For OpenCL, all profiling activities, except for power consumption, were performed using OpenCL's profiling events. To improve the precision of the measurements,

each measurement was repeated ten times, and then an average was recorded.

The K20Power [126] profiling tool was used for power and energy profiling in both CUDA and OpenCL. It is a power and energy profiler for GPU applications executing on K20 GPUs. It performs automatic correction for the slow ramping behaviour due to the built-in power sensor; it records the full power profile and calculates the energy used by the GPU during kernel execution.

3.4 CHAPTER SUMMARY

This chapter has described the methodology followed to realise the original aim of this project, namely, to conduct a comparative study of high-level tool-flows for rapid prototyping of SDR DSP operations on FPGA and GPU platforms. The methodology is organised into two main parts: the first part focuses on FPGA tool-flows, and the second on GPU tool-flows. Figure 3.4 shows a summary of steps that are followed to evaluate both FPGA and GPU tool-flows.

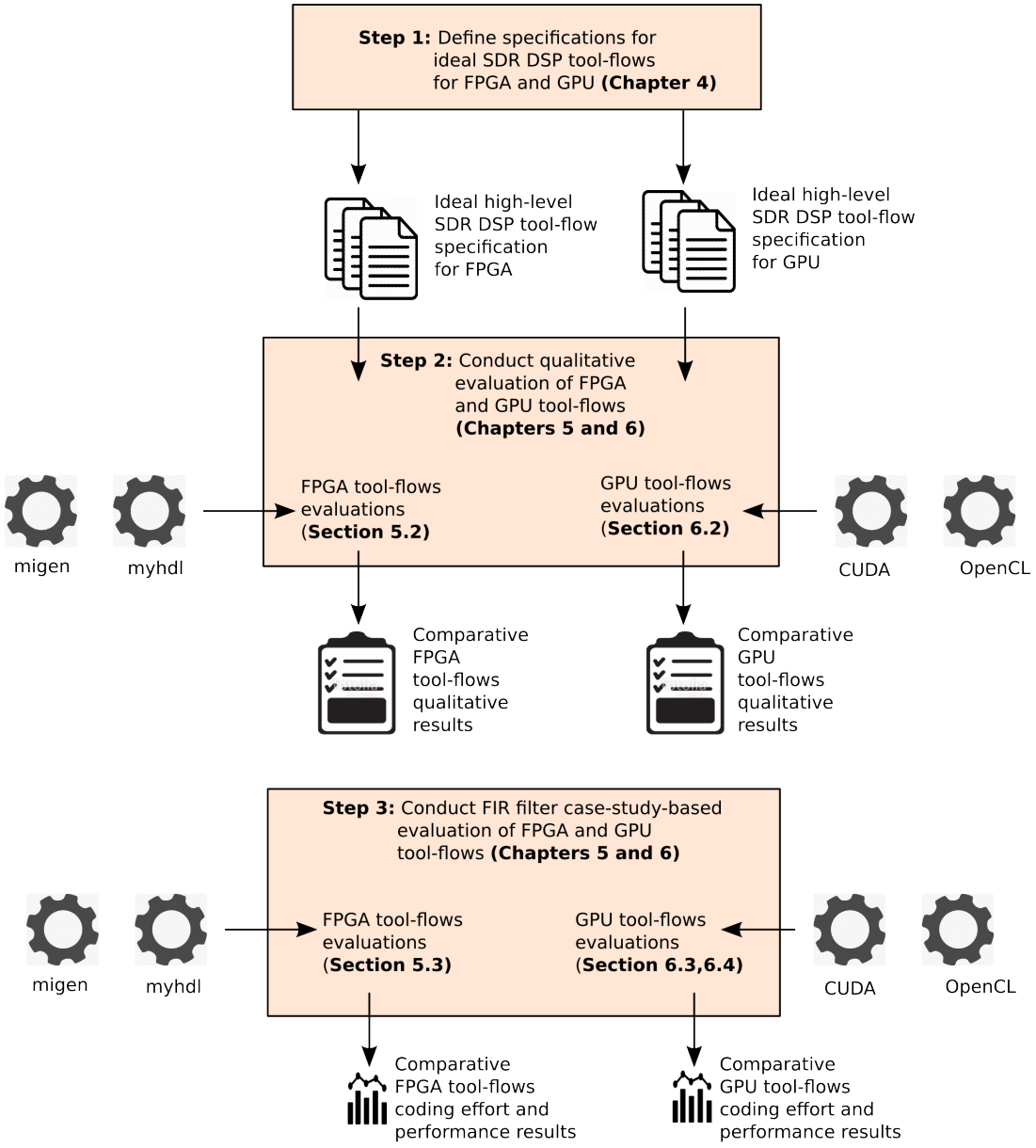

Figure 3.4: Steps for evaluation of FPGA- and GPU-based tool-flows.

The approach is to first develop the comprehensive criteria to use in comparing the tools. A system engineering process is followed to develop these criteria in the form of system specifications for an ideal high-level SDR DSP tool-flow on FPGA and GPU platforms. Once the comparison criteria have been defined, the next step in the approach is to perform a study of tool-flows for rapid prototyping of SDR DSP on FPGA and GPU platforms respectively. For each of the two platforms, this entails: i) selecting representative tools to study, ii) evaluating the selected tools against the systematic criteria, iii) following a waterfall software development process to implement a FIR filter case study application using each of the selected tools, and iv) using the case study to do a comparative evaluation of the tools in terms of design productivity and performance. The tools, methods and metrics used to evaluate both the design efficiency and performance of each of the selected FPGA and GPU tool-flows were described.

Chapter 4 details the first step of the methodology, which involves the systematic design of criteria to evaluate and compare selected tool-flows for rapid prototyping of SDR DSP.

CHAPTER 4

SYSTEMATIC ANALYSIS AND DESIGN OF AN IDEAL HIGH-LEVEL SDR TOOL-FLOW

This chapter describes the development of the system requirements for an ideal high-level tool-flow for fast prototyping of SDR DSP on heterogeneous platforms. The focus is on platforms that comprise one or a combination of GPUs, multicore GPPs and FPGAs. While several high-level design flows exist both in the literature and in industry, there has generally been little work done to define what constitutes a true high-level SDR DSP design flow. As a result, some previous works on comparative tool surveys use loosely defined comparison criteria, which in turn affect the objectivity of the results. Development of a complete specification of an ideal high-level SDR DSP design flow for heterogeneous platforms is beyond the scope of this dissertation. Only a description that would be sufficient to aid in defining objective tool evaluation criteria in Chapters 5 and 6 of this dissertation, is developed in this chapter.

Starting tool-flow features and requirements collected from the literature survey in Chapter 2 are listed in Section 4.1. The requirements are then analysed and elaborated on in Subsection 4.2.1 to produce the functional architecture, the functional structure and the interface diagrams of the ideal tool-flow system. Finally, a set of system requirements, derived from the original tool-flow features, is described in Subsection 4.2.2, and the logical architectures of the proposed ideal tool-flow systems for FPGA and GPU targets are presented.

4.1 TOOL-FLOW SYSTEM CONCEPTUALISATION

This section describes the high-level SDR tool-flow system concept by identifying the stakeholders (4.1.1) and their requirements (4.1.2). It thus analyses the problem space, and the needs and requirements of the tool-flow system.

4.1.1 Stakeholder Identification

Stakeholder classes for the ideal high-level SDR tool-flow system were identified by considering each stage of the system life cycle and identifying a list of main stakeholders who have an interest in the system. See Table 4.1 for a list of identified stakeholders. The list is not exhaustive, but it is large enough to aid in developing a comprehensive requirements basis for the ideal tool-flow.

Table 4.1: Identification of Stakeholders for the Ideal High-Level SDR Design Flow.

Life Cycle Stage	**Stakeholders**
Engineering	EDA tools engineers
	EDA standards bodies
	SDR waveform developers
	FPGA-based SDR platform makers
Development	EDA tools developers
	EDA standards bodies
Operation	EDA SDR waveform developers
	EDA Researchers
Maintenance	EDA tools developers

The stakeholders are further organised into four main classes which are described below.

- **End-Users** - this group consists of end-users of the ideal high-level SDR tool-flow. Members are: DSP gateware engineers (MeerKAT [127] digital-backend (DBE) team); SDR researchers (Software-Defined Radio Group at the University of Cape Town).

- **Research & Development** - this group comprises scientists and engineers who are involved in the design and construction of the ideal high-level SDR tool-flow. Members are: High-level hardware design technology researchers; SDR researchers; Software engineers; Gateware engineers.

- **Hardware Manufacturers** - this group consists of manufacturers of chips and boards that are used to implement SDR applications. Members are: SDR FPGA board makers (UCT RHINO team, ROACH team); manufacturers of FPGA devices (Xilinx, Altera).

- **Standardisation Bodies** - this group comprises relevant standardisation bodies who direct the design and development of compatible, interoperable and good quality design tools and applications. Members are: SDR standardisation bodies (Wireless Innovation Forum [128]); FPGA EDA standardisation bodies.

The requirements gathered from the different stakeholders identified above are presented in the subsection that follows. They capture the high-level SDR DSP tool-flow concept from the stakeholders' points of view, and form the basis for defining the tool-flow system in the subsequent subsection.

4.1.2 Stakeholder Requirements

Stakeholder requirements are a list of non-technical requirements that describe the needs of the system to be built. In this dissertation, the system of interest is the ideal high-level tool-flow for fast prototyping of SDR DSP waveforms on FPGA and GPU platforms.

A sample of the stakeholder requirements card from the end-users stakeholder group is shown in Figure 4.1. This subsection does not provide a complete listing of all the stakeholder requirements cards for the ideal tool-flow system. The rest of the cards can be accessed in Appendix A.2 of this dissertation. A complete set of numbered stakeholder requirements, from all the stakeholder groups, are however presented in Table 4.2.

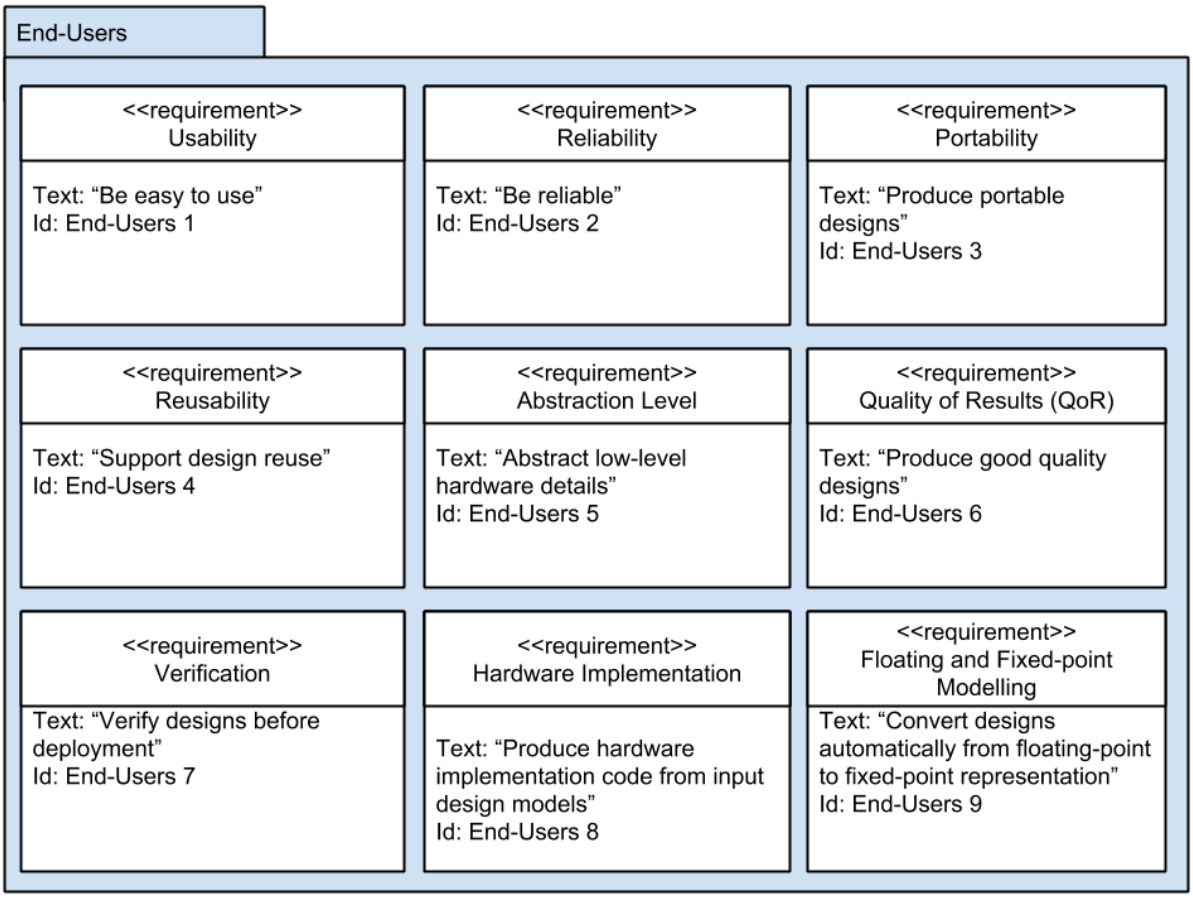

Figure 4.1: Original requirements of the End-Users stakeholder group for the ideal high-level SDR tool-flow.

The stakeholder requirements presented in this chapter are the result of a process consisting of interviews and email correspondence with representatives from the various stakeholder groups identified in Subsection 4.1.1 above. Additionally, some of the requirements were informed directly by means of literature published by the stakeholders themselves on the subject.

Table 4.2: Stakeholder Requirements with regard to High-Level FPGA SDR DSP Tool-flow.

Stakeholder	**Requirement**	**ID**
End-Users	Should be easy to use	EU-R1
	Should be reliable	EU-R2
	Should produce portable designs	EU-R3
	Should support design reuse	EU-R4
	Should abstract low-level hardware details	EU-R5
	Should detect design errors early in the development cycle	EU-R6
	Should deploy user SDR waveform on a hardware platform	EU-R7
	Should generate bit-accurate models for FPGA designs	EU-R9
	Should support automatic design optimisations	EU-R10
	Should generate quality designs	EU-R11
R & D	Should support design reuse	RD-1
	Should generate quality designs	RD-2

Continued on next page

Table 4.2 – *Continued from previous page*

Stakeholder	Requirement	ID
	Should easily integrate with third-party SDR waveform design tools	RD-3
	Should be based upon established design techniques and technologies	RD-4
	Should be designed for easy upgrading and customisations	RD-5
HW Vendors	Should generate portable designs	HV-1
	Should support automatic design optimisations	HV-2
	Should support one-click generation of implementation code	HV-3
Standardisation	Should produce portable designs	SB-1
	Should support multiple high-level design paradigms Should easily integrate with other third-party SDR waveform design tools	SB-2 SB-3
	Should be based upon relevant established SDR and EDA standards	SB-4

In the section that follows, the stakeholder requirements listed in this section are analysed by means of the system engineering design techniques, which were covered earlier in the literature review (Chapter 2) in order to create a clear system-level description of the high-level SDR DSP tool-flow.

4.2 TOOL-FLOW SYSTEM DEFINITION

This section focuses on describing, in detail, a system-of-interest (SoI) to satisfy the identified high-level SDR DSP tool-flow requirements. It starts with a functional analysis (4.2.1). The functional architecture of the system is described in Subsection 4.2.1.1. Finally, the functional structure and interface of the system is described in Subsection 4.2.1.2.

Thereafter, the system requirements are presented (4.2.2) in terms of four categories, viz. function, performance, usability, and interface. Lastly, the system architectures of the ideal high-level tool-flows with regard to FPGAs (4.2.3) and GPUs (4.2.4) are discussed in greater detail.

4.2.1 Functional Analysis

A function refers to a specific or discrete action (or series of actions) that is necessary to achieve a given objective, in other words, an operation that the system must perform. Functional analysis is a process of translating the stakeholders' requirements into detailed design criteria. The purpose of functional analysis is to present an overall integrated description of the system's functional architecture, operational scenario and interface.

4.2.1.1 Functional Architecture

The functional architecture of a system is a collection of functions and corresponding sub-functions that describes the input to output processing done by the system to realise its mission. According to the analysis of the original stakeholders' requirements presented in the previous subsection, the high-level SDR DSP tool-flow system should be able to perform five key functions, namely, hardware abstraction, floating- to fixed-point conversion, design verification, code generation and design optimisation, all of which are presented in the functional architecture diagram shown in 4.2.

Hardware abstraction refers to the system's ability to hide away the low-level implementation hardware details from the designer and thus to enable the designer to focus on domain application tasks. This implies that the

tool-flow should provide design techniques and languages for capturing designs at higher levels of abstraction. For example, for FPGA platforms, this means an ability to target the device from an algorithmic application model.

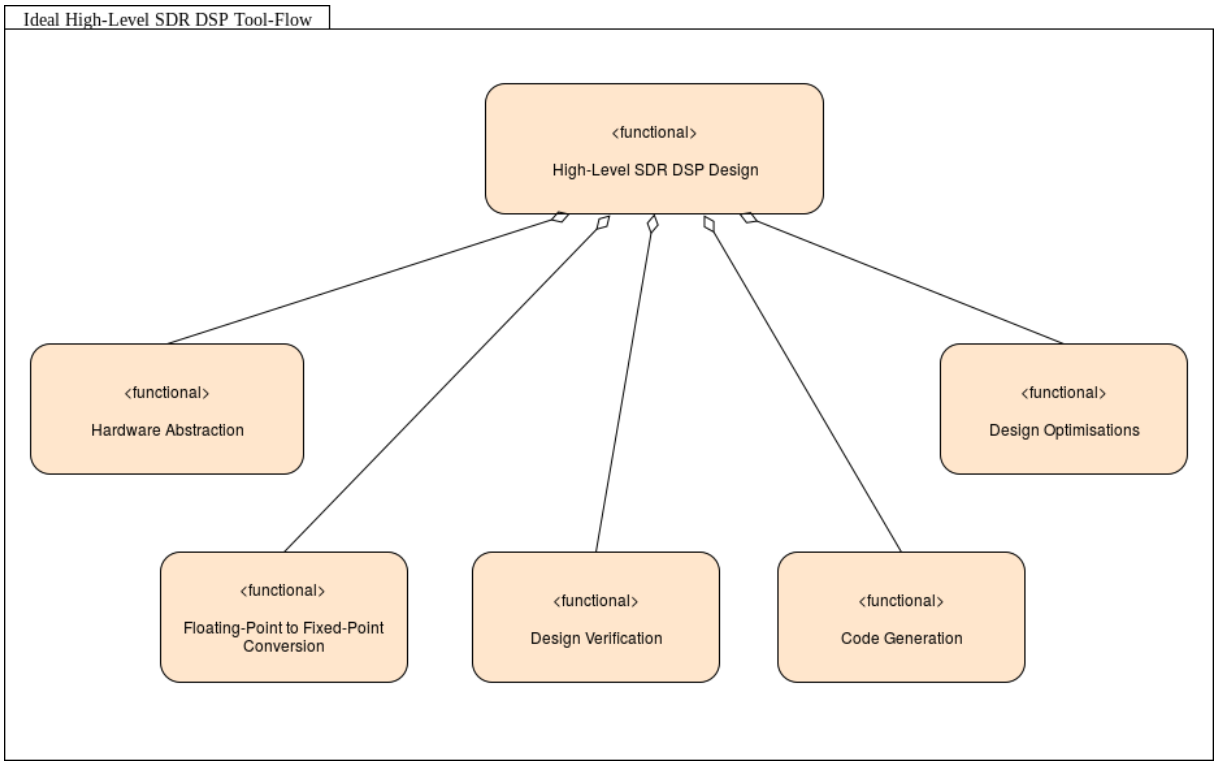

Figure 4.2: Functional architecture diagram for the high-level SDR DSP tool-flow. Key system functions are shown.

Floating to fixed-point conversion originates directly from the stakeholder requirements. This is intended to cater for the FPGA targets of the tool-flow. Implementation of floating point operations on FPGAs is very costly in terms of resources [15]. Therefore, FPGA designs are typically first translated to the fixed-point format prior to FPGA deployment. This is not necessary for GPU targets since they are optimised for floating point operations. This is an important function of the tool-flow.

Two main approaches are used in the literature to support hardware abstraction in high-level design: model-driven development, textual specifications for the high-level language, and virtual hardware prototypes. Model-driven development uses graphical models and pre-built application components so that designers can visually construct complex applications. High-level hardware design languages raise the level of abstraction above RTL to behavioural or system level, and thus enable the user to specify the hardware application's intent as though it were software. Therefore, the hardware abstraction function of the tool-flow system is broken down into two main sub-functions, viz. model-driven and high-level design specifications, as shown in Figure 4.3. Functional decomposition of the rest of the tool-flow functions is documented in Appendix A.

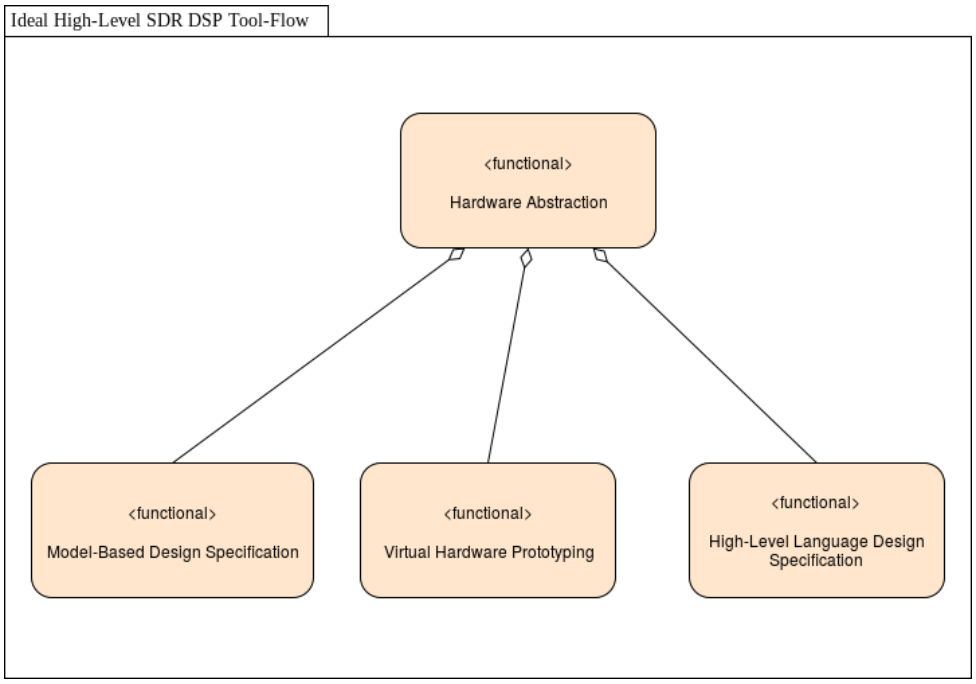

Figure 4.3: Functional architecture diagram for the hardware abstraction function.

*4.2.1.2 **Functional Structure and Interface***

The functional structure and interface diagram (shown in Figure 4.4) provides a summary of the tool-flow system's primary functions and how the system is envisioned to interconnect with external systems.

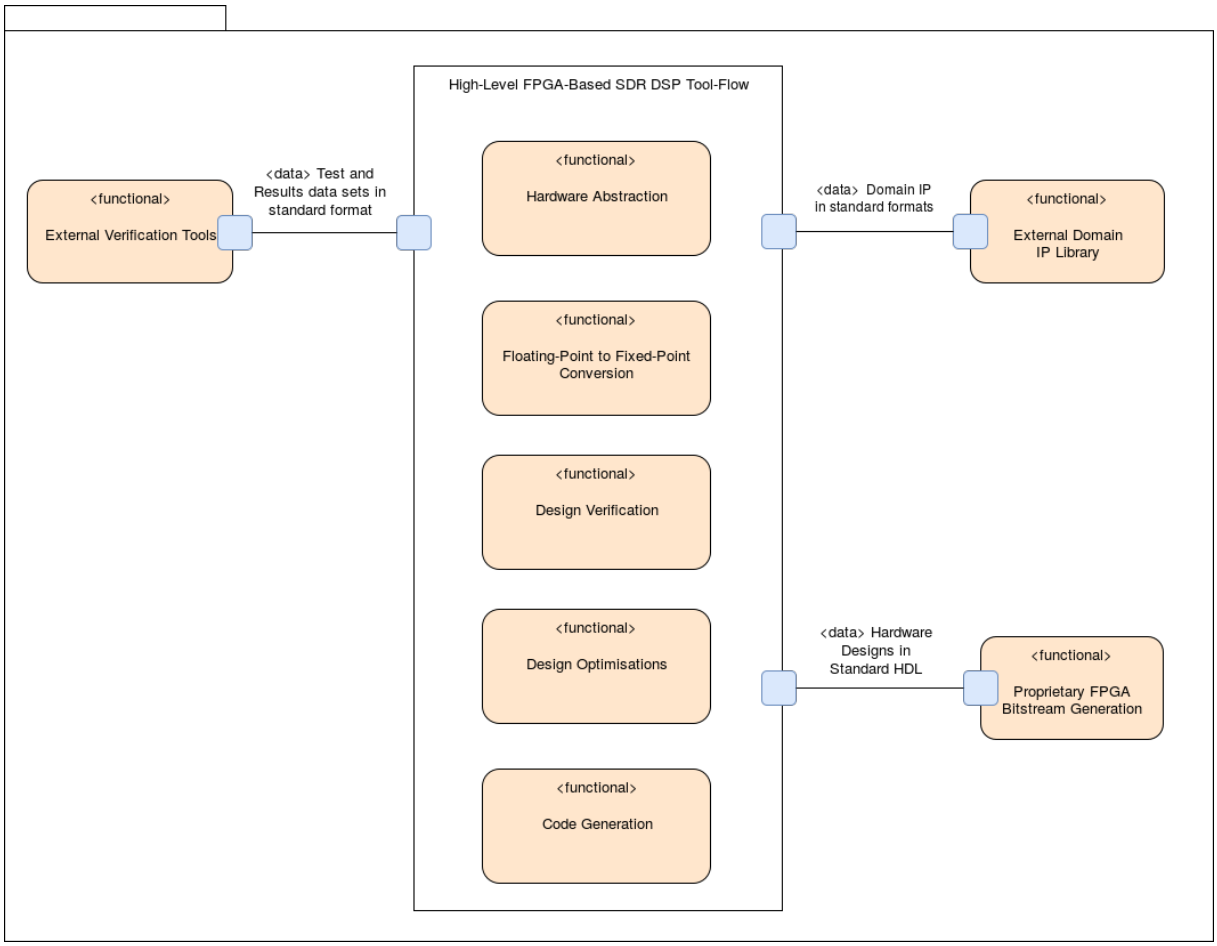

Figure 4.4: High-level FPGA-based SDR DSP tool-flow functional structure and interface diagram.

Functional analysis of the high-level SDR FPGA tool-flow was presented in this subsection. The output of the analysis is a more detailed set of functional and non-functional requirements of the tool-flow system which are presented and explained in the subsection that follows.

4.2.2 System Requirements

This subsection describes the system requirements of the high-level SDR tool-flow. System requirements are all of the requirements, at the system level, that describe the functions which the system as a whole should fulfil to satisfy the stakeholder needs and requirements.

The requirements are grouped into functional and non-functional categories.

4.2.2.1 Functional Requirements

This sub-section lists and discusses the functional requirements of the ideal high-level SDR DSP tool-flow system. Functional requirements describe the functions that the high-level SDR tool-flow system should perform in operation. They define quantity (how many), quality (how well), coverage (how far), time lines (when and how long) and availability (how often) [107]. The functional requirements of the tool-flow system are derived by using the technical knowledge on tool-flows, FPGAs, GPUs and SDR waveforms reviewed in the literature review Chapter to transform the high-level stakeholder requirements into technical requirements stating what the the tool-flow system and its components must accomplish in order to fulfil its mission.

FR1: Design entry must be above RTL

In order to reduce the present design productivity challenge affecting FPGA applications design especially, the design abstraction level needs to be lifted above RTL. Developing FPGA applications at RTL poses several challenges to SDR waveform developers. At RTL, the designer develops an application by describing how synchronous and combinatorial circuit modules inter-connect together to give it it's functionality. This is an error-prone, tedious process that requires knowledge and experience in digital hardware design; a knowledge possessed by only a few even among SDR waveform designers[5]. Majority of SDR applications designers are however conversant in algorithmic languages such as MATLAB, Python, C++ and use them to model the functionality of algorithms before hardware implementation[129]. Therefore, the ideal tool-flow should allow designers to target FPGA platforms from algorithmic SDR waveform specifications.

Thus, the ideal tool-flow should enable designers to target FPGA platforms from SDR waveform specifications at algorithmic and system levels of abstraction.

The waveform typically includes data-flow and control-flow oriented parts[130]. For example channelisation requires a data-flow MoC while a Virterbi decoder operation is a control-flow-oriented operation. Therefore, a variety of high-level design paradigms and computing models suitable for modelling the different operations of a SDR waveform should be supported. Particularly, the tool-flow should support design entry through both textual and component-based approaches. Textual capture allows the designer more and finer control over the application but suffers from relatively poor intuition. On the contrary, the component-based approach is relatively easy to understand and quicker to assemble a design yet is cumbersome to use for big designs and the type of applications that can be developed is limited by the library of available primitives.

FR2: Automatic floating to fixed-point conversion must be supported

FPGAs are fixed-point devices. That is, they are not efficient in implementing floating-point operations except through the use of specialised soft/hard floating-point arithmetic processing cores. Therefore, for efficient FPGA implementation, SDR waveform designers usually first need to convert input and internal algorithmic data from floating to fixed-point and perform fixed-point analysis to ensure that algorithmic correctness is not lost. The process of converting application models from floating to fixed-point has been identified as one of the most difficult aspect of implementing an algorithm on an FPGA[131].

Therefore, in order to facilitate efficient implementation of SDR waveform designs on FPGA platforms, the tool-flow should provide facilities that enable designers to automatically translate their waveform models from floating- to fixed-point format. The facilities should include verification capabilities that allow the designer to evaluate the quality of the generated fixed-point model through quantization analysis process[41].

FR3: The tool-flow must support fast design verification before and after implementation

Several studies have shown that FPGA verification, compared to other SDR platforms such as GPUs and GPPs, consume majority of design time [132]. Figure 4.5 shows that an average 48% of total FPGA design time was spent in verification related tasks in 2016. It is also a known fact that high-level based design entry promotes rapid functional verification of designs and effectively minimises the current design productivity gap[133]. Therefore, on top of supporting design capture of higher levels of abstraction, the high-level specifications should also be executable. Verification is not a major challenge among existing GPU programming methodologies. However, key GPU verification challenges include the complex and evolving HW/SW ecosystem and software managed coherence. The ideal tool-flow should provide efficient and easy to use tools to debug, validate and profile designs in terms of metrics such as energy, memory cost and execution time[41].

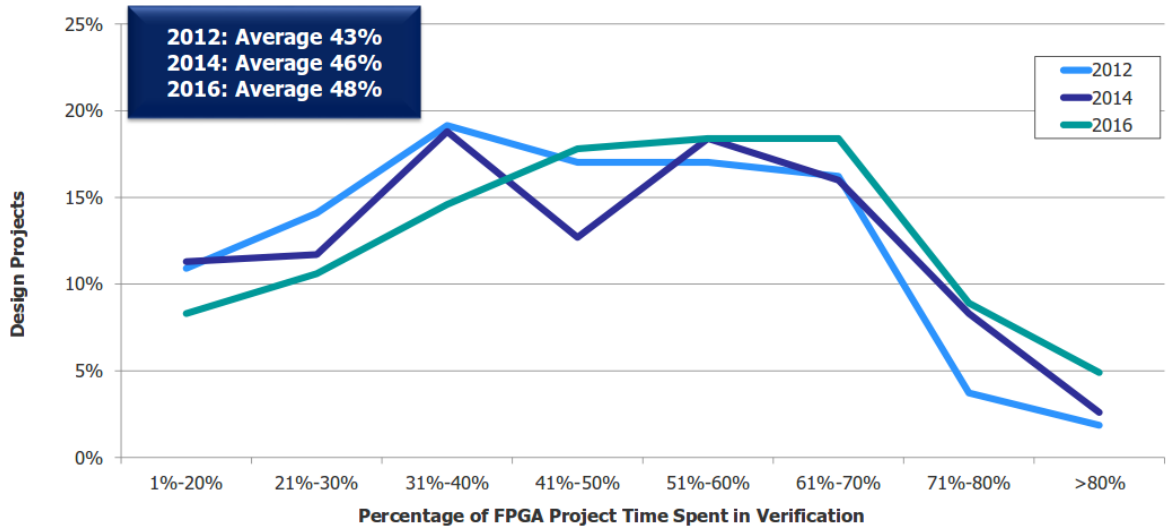

Figure 4.5: Percentage of FPGA project time spent in verification (Source: [134])

FR4: The tool-flow must support automatic generation of implementation code

According to the Wireless Innovation Forum, techniques to enable efficient porting of SDR waveform applications from high-level specifications to embedded heterogeneous platforms including FPGAs, GPPs, DSPs and GPUs are required [43]. Currently, there is still a large divide between algorithm modelling and hardware implementation. For example, after using a high-level language such as MATLAB to explore the functional correctness of an algorithm, an SDR waveform designer would typically need to manually convert the high-level model to a GPU or FPGA implementation [5]. This divide between algorithm modelling and hardware implementation results in several challenges include long design times and increased design errors. Therefore, the ideal tool-flow should allow automatic code generation for programmable SDR DSP platforms such as FPGAs and GPUs. The designer should be able to perform "push-button" generation of FPGA code from high-level SDR waveform specifications. Since, in practice, FPGA configuration stream is synthesised from two main standard HDLs (VHDL and Verilog), the tool-flow should generate VHDL or Verilog equivalents of the high-level input model[41].

FR5: The code generator must support automatic design optimisations

The current design technology challenge is not only characterised by poor productivity levels, but also, increasing demands for high quality designs, especially in communication applications of the SDR domain. For example, low energy consumption and high data speeds are among key design goals of modern communication SDR waveform designs. Achieving a high quality design is not a result of chance or simply choosing an efficient algorithm. The operation must also - among others- be optimised for the specific hardware platform to be used. This process is called design space exploration and entails searching for an optimum application design for given platform constraints[135]. Performed manually, this process is tedious and skill-intensive.

The ideal tool-flow should be able to perform a variety of automated domain- and platform-specific optimisations to any given user design to produce improve the overall design quality. For example, key FPGA design optimisations in terms of area, power and timing should be supported. Automatic support for SDR domain-specific optimisations should also be provided [41].

FR6: The tool-flow must enable implementation of the SDR design onto target FPGA

The tool-flow should provide a path to implement a final SDR waveform design onto an FPGA. Implementing a SDR waveform design on an FPGA involves storing the configuration binary stream generated using proprietary tools on the FPGA's programming points.

Ideally, the tool-flow should facilitate implementation of an SDR waveform design on an FPGA platform. Deploying the SDR waveform design on the FPGA entails loading the bitstream encoding the design into the FPGA's programming points. SRAM memory is commonly used to keep the programming bitstream in modern FPGAs. PROM and SPI flash are alternative technologies used to store the FPGA's bitstream. The process of generating bitstream for FPGA is currently still largely closed. That is, bitstream generation tools are mainly proprietary and owned by FPGA vendors.[136, 41].

*4.2.2.2 **Performance Requirements***

Performance requirements define qualitatively the extend, or how well, and under what conditions a system function or is to be performed. Generally, performance requirements are given in terms of quantity, quality, coverage and readiness [106]. These requirements are developed by performing a process of requirements analysis across all identified system functions and characterised by stating degrees of certainty in their estimate, how critical they are to system success and how they related to other requirements.

PR1: The tool-flow must support design reuse

Design reuse is a well known mechanism used in both hardware and software development to reduce design complexity and increase productivity. Two key things are necessary in supporting design re-use: a library of pre-defined components and an ability to re-use those components to develop new and bigger applications designs. Therefore, the ideal tool-flow should have a rich library of pre-defined components that are commonly used in SDR DSP waveform application development. The library components should be highly parametrisable in order to cater for multiple different design requirements. Because most of the legacy FPGA code is still in RTL, the tool-flow should support integration of pre-defined RTL components defined in VHDL or Verilog [41].

PR2: The tool-flow must enable development of good quality designs

Since "there is no free lunch", it is expected that an effective increase in design productivity as a result of an increased level of abstraction for design capture will also result in some level of reduction in the overall quality of designs. However, using several modern high-level design techniques, it is possible to check the degradation in design quality due to increased levels of abstraction such that designs with quality that is comparable to standard methodologies are obtained. Therefore, the ideal tool-flow should produce designs whose quality is not significantly lower than that of hand-written designs [41].

PR3: The tool-flow must enable development of portable designs

Waveform portability is one of of the key design requirements in current SDR technology [142]. The SDR waveform should be developed in such a way that it is independent from any target SDR implementation architecture. It should be easy to port from one SDR platform to another. Waveform portability can be achieved through several techniques including ensuring that the tool-flow and the waveform are patterned after relevant well established industry standards such as the SCA [41].

PR4: The tool-flow must operate reliably

Reliability refers to the likelihood that a tool will successfully perform its required task under given conditions. Whether, it's the automatic floating- to fixed-point conversion, automatic implementation code generation or applying platform-specific design optimisations, the ideal tool-flow is required to be reliable in all cases. Among several factors, the success of high-level tool-flows hinge on the use of accurate platform models that provide estimations of key characteristics such as latency, area and power consumption for the designs being developed. The International Technology Roadmap for Semiconductors (ITRS) highlighted that reliability and accuracy of these estimations is pivotal in realising high-level design methodologies [138].

*4.2.2.3 **Usability Requirements***

Usability requirements define the quality of the system use.

Table 4.3: Functional system requirements for the high-level FPGA-based SDR design flow.

ID	Stakeholder Requirement	System Requirement	Explanation
EU-R5	Abstract low-level hardware details	Specify SDR design functionality at algorithmic level of abstraction	Algorithmic design entry higher than RTL on the design abstraction hierarchy.
		Specify SDR design functionality at system level of abstraction	[128, 3]
EU-R6	Detect SDR design errors early in the development cycle	Simulate high-level input specification	[128]
		High level SDR specification formal verification	[137, 138]
		Automatic test bench generation	To speedup the verification process [138]
		Intelligent test bench	To speed up and streamline the verification task [138]
		Virtual platforms	[139, 140]
EU-R9	Floating and fixed point analysis and design	Floating-point to fixed-point conversion	F-point design implementation on an FPGA is difficult and costly. Fix-point models may suffice for some SDR applications.
		Bit-accurate high-level input specification simulation	[47]
EU-R8	Optimise SDR designs automatically	Domain-specific (SDR) design optimisations	[141]
		Generic design optimisations	[141]
EU-R7	Deploy SDR waveform onto FPGA platform	Generate HDL code	Must generate VHDL/Verilog for mapping with the traditional flow
		Generate FPGA bitstream	[3]
		Program FPGA	[3]
RD-3	Integrate with third-party SDR and FPGA design tools	Integrate with third-party verification tools	[3]
		Integrate with third-party FPGA bitstream generation tools	FPGA bitstream generation largely proprietary except a few [3]
		Integrate with third-party FPGA programming tools	[3]

Table 4.4: System requirements for the high-level FPGA-based SDR design flow.

Design Stage	System Requirements
High-Level Modeling	Algorithmic-level design entry?
	System-level design entry?
	SDR primitives library?
	SDR MoCs?
	Floating to fixed-point conversion?
Design Verification	High-level model simulation?
	High-level model formal verification?
	HW/SW co-verification?
	Automatic test-bench generation?
	Intelligent test-bench?
	Virtual platforms?
	Third-party verification tools integration?
Code Generation	Software code generation?
	Hardware code generation?
	Logic synthesis?
	FPGA bitstream generation?
	Domain-specific design optimisations?
	Generic design optimisations?
Design Implementation	FPGA bitstream generation?
	Third-party FPGA implementation tools integration?

UR1: The tool-flow must be easy to use

To enhance design productivity, and thus provide time-to-market advantage to the SDR waveform designer, the ideal tool-flow should be designed for ease-of-use. Given that some designers are slow to adopt FPGA technology because mainstream design methodologies are difficult to use, an easy-to-use high-level design methodology may enable a wider adoption of FPGA technology among SDR waveform designers. Ease of use is one difficult property to measure. However, there are several features and functions which when included in the tool-flow system specification, can enhance its ease of use. For example, according to Xilinx [143], these includes preconfigured push-button flows for new users, advanced analysis-driven flows for expert users, support for already familiar industry standards, simple method for creating application modules and assembling the modules together to form a system application, readable and interactive reporting facilities.

4.2.2.4 Interface Requirements

Interface requirements define how the tool-flow system is required to interact or exchange information with external systems, or how system elements within the system, interact with each other.

IR1: The tool-flow must integrate seamlessly with external design tools

External tools integration and interoperability is a well known high-level design mechanism that improves the versatility of the design process. Thus, the ideal tool-flow should allow designers to leverage external verification tools for better results as well as external FPGA implementation tools to support a wide variety of FPGA platforms.

4.2.3 Ideal High-Level FPGA Tool-flow System Architecture

This subsection presents and describes the developed architecture of the high-level tool-flow system for rapid prototyping of SDR waveform operations on FPGA and GPU platforms. The purpose of the system architecture is to define a comprehensive solution based on principles, concepts, and properties logically related and consistent with each other. The architecture has features, properties and characteristics satisfying, as far as possible, the problem or opportunity expressed by a set of system requirements and are implementable through relevant technologies.

The logical architecture of the ideal tool-flows for FPGA targets, synthesised from the system requirements, is shown in Figure 4.6. As shown on the figure, the Tool-Flow consists of three main design stages - high level modeling, code generation and hardware implementation - with intermediate verification stages between each of the design stages. Detailed descriptions of the functions and services provided in each stage are given in the subsections below.

4.2.3.1 High-Level Modeling

The high-level modeling stage is concerned with the use of high-level modeling techniques to enable the designer to efficiently capture an input SDR waveform from a given specification, translate it to fixed-point representation for FPGA deployment and verify its functional correctness before inputing it to the code generation engine for hardware generation. The actual corresponding system requirements are:

- High-level capture of user SDR models
- Platform-indepedent capture of user SDR models
- An extensible library of re-usable SDR primitives
- High-level input model based on standard SDR MoCs
- Automatic conversion of SDR models to fixed-point

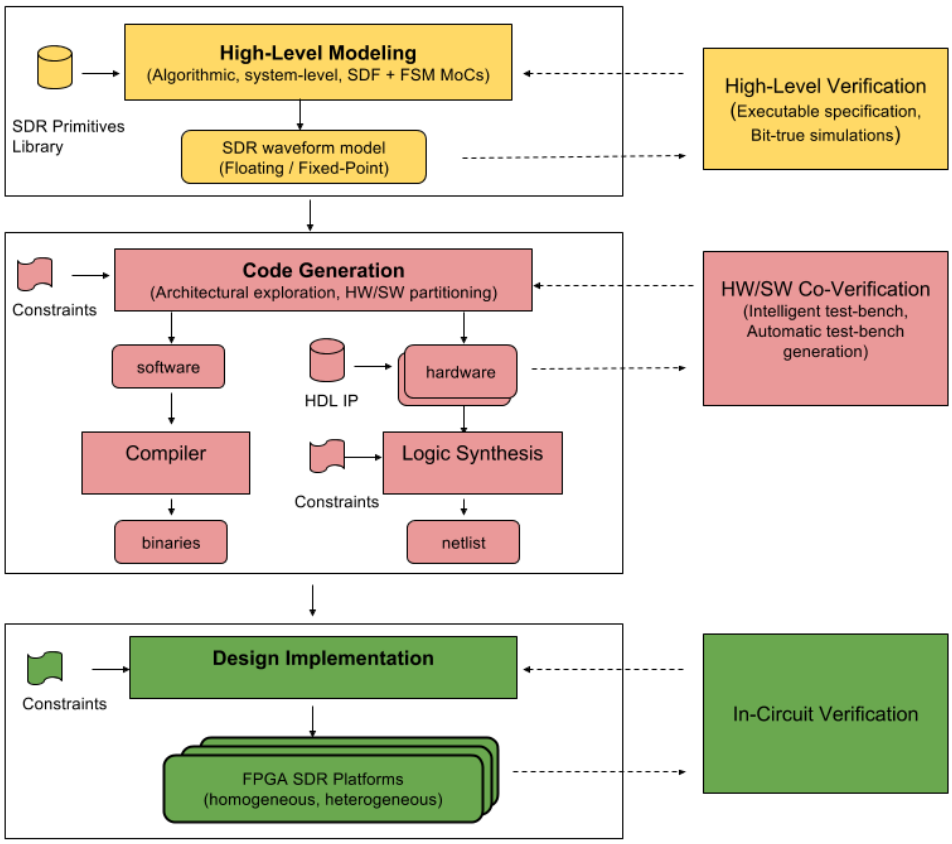

Figure 4.6: The Ideal High-Level SDR Tool-Flow logical architecture.

4.2.3.2 Code Generation

The code-generation stage is concerned with synthesis of good quality hardware code from the high-level SDR waveform specifications for deployment on target FPGA hardware. Further, the code-generation process must be such that it allows the designer to explore the design space easily and early through automatic generation of multiple designs of different architectural features from the same input high-level model. The actual listed system requirements for this design stage are:

- Synthesis of RTL code from high-leve user SDR model
- Automatic design-space-exploration
- Produce good quality RTL code

4.2.3.3 Verification

Verification encompassess all tasks, tools and methods that are used to prove the correctness and validity of the SDR waveform design at different levels of the development process. In order to narrow the current design productivity gap, the Ideal High-Level SDR tool-flow should support both a fast and reliable verification approach since a high verification effort is one of the major challenges in conventional design methodologies. The specific listed verification system requirements are:

- Functional verification of high-level SDR models

- Verification of synthesised RTL code
- Integration with external verification tools

4.2.4 Ideal High-Level GPU Tool-flow System Architecture

The logical architecture of the ideal tool-flows for GPU targets, synthesised from the system requirements, is shown in Figure 4.7. As shown on the figure, the Tool-Flow consists of three main design stages - high level modeling, code generation and hardware implementation - with intermediate verification stages between each of the design stages. Detailed descriptions of the functions and services provided in each stage are given in the subsections below.

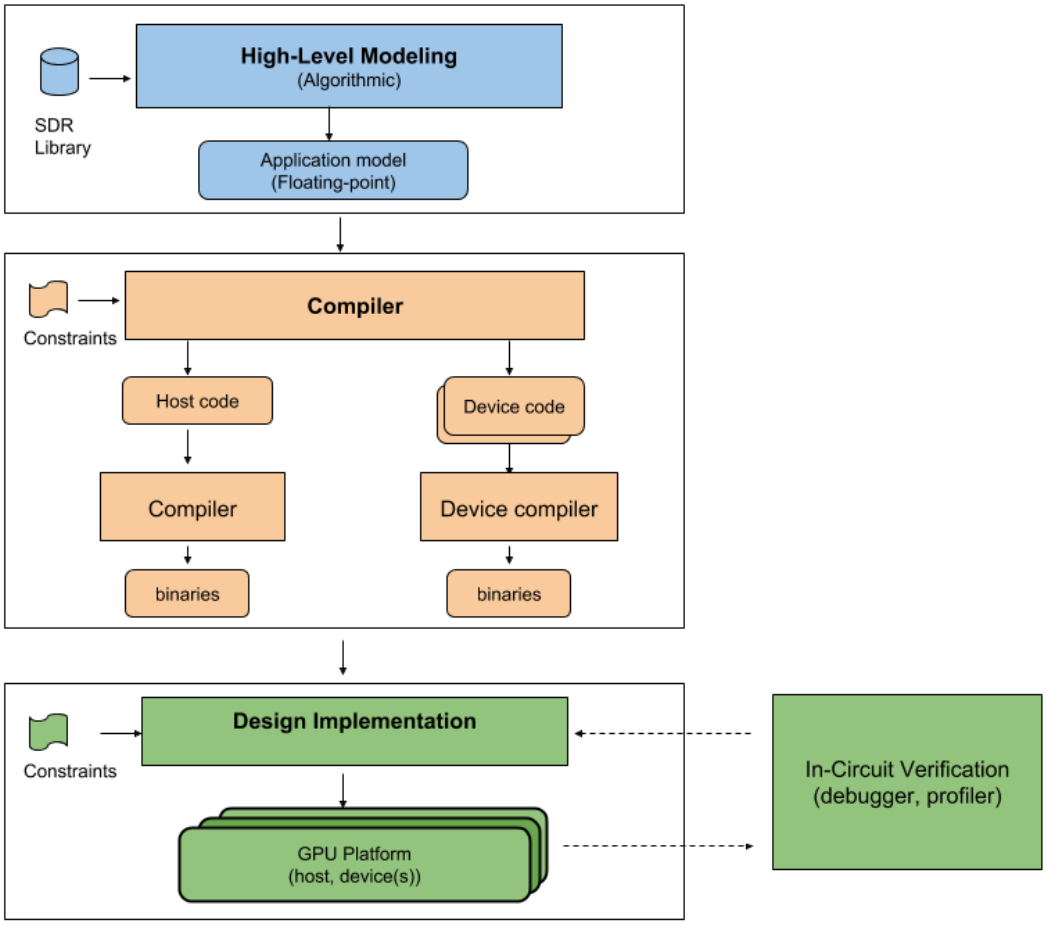

Figure 4.7: The Ideal High-Level SDR GPU Design Methodology

4.2.4.1 High-Level Modeling

GPU programming tools can be roughly classified into three categories according to the level of abstraction they support[144]:

1. High abstraction function libraries that provide commonly used algorithms with auto-generated GPU kernels.
2. Low abstraction lightweight GPU programming frameworks where programmers develop GPU kernels from scratch with no automatic code generation.

3. High abstraction compiler-based frameworks where GPU kernels are automatically generated by compilers or language runtime systems, through the use of pragmas, algorithm templates, and sophisticated program analysis techniques.

The ideal high-level toolkit should support high-level input of SDR DSP algorithms. The developer should be able to specify an application without worrying about low-level aspects such as garbage collection, device configuration and thread synchronisation. In order to improve ease of use and increase design productivity, highly parameterised SDR libraries providing commonly used SDR algorithms such as FFT, FIR filter, modulators and demodulators should be supported. In order to improve the level of expressiveness, it is essential that applications also be specified according to the synchronous dataflow model of computation which is best suited for DSP applications[145]. In addition, to reduce development time and cost, it is necessary that the high-level SDR DSP software specification be written to be easily portable from one GPU platform to another [128].

*4.2.4.2 **Algorithm Verification***

The methodology should provide efficient, reliable and easy to use facilities to debug and profile the SDR waveform application. The profiler should be of a very high resolution (1e-9) and should be able to measure GPU operations time and time taken to transfer data between device and host. Facilities for graph and waveform visualisation should be provided [71].

*4.2.4.3 **Code Generation***

The ideal high-level design methodology should support automatic generation of optimised kernel code targeting various GPU platforms from the high-level input specification[144]. The code-generator should be capable of automatically identifying both data-level and task-level parallelism inherent in the input application and exploit it to produce optimum performance implementation of the SDR DSP application. That is, the ideal design toolkit should be capable of automating optimum partitioning of the input algorithm into parts that should be deployed on the CPU and parts that can be accelerated on the GPU. For expert developers, pragmas and other directive programming techniques should be supported to factor in designer input in the code-generation process [71].

4.3 CHAPTER SUMMARY

This Chapter has discussed conceptualisation and design of a systematised Ideal High-Level SDR Tool-Flow. The system definition of the Ideal High-Level SDR Tool-Flow produced in this Chapter forms an integral part of the evaluation methodology used in Chapters 5 and 6 to conduct studies of tool-flows for rapid prototyping of SDR DSP operations on FPGA and GPU targets respectively.

CHAPTER 5

EVALUATION OF TOOL-FLOWS FOR RAPID PROTOTYPING SDR DSP OPERATIONS ON FPGAS

This chapter presents a comparative evaluation of high-level tool-flows for rapid prototyping of SDR DSP operations on FPGA platforms. The specific tool-flows being investigated are MyHDL and Migen. The evaluation methodology considers both qualitative and quantitative criteria. The qualitative criteria are based on the proposed specification of the ideal tool-flow described in Chapter 4, whereas the quantitative criteria are based on a FIR filter case study. A FIR filter is described, using both MyHDL and Migen, and then synthesised and implemented for a Xilinx Virtex 7 FPGA using ISE 14.7. Using the case study, the tool-flows are then evaluated and compared in terms of logic utilisation, power consumption and speed.

Section 5.1 describes the evaluation criteria. Qualitative and quantitative evaluations of tool-flows are presented in Sections 5.2 and 5.3 respectively, and a summary of the findings of the study is given in Section 5.4.

5.1 EVALUATION CRITERIA

This section describes the qualitative and quantitative criteria that were used to study and compare two high-level tool-flows for rapid prototyping of SDR DSP on FPGA platforms. The qualitative criteria focus on the essential features of a good SDR DSP tool-flow; they are described in Subsection 5.1.1. Subsection 5.1.2 describes the quantitative criteria, based on the FIR filter case study, which were used to evaluate and compare the programming effort and performance associated with the tools.

5.1.1 Proposed Ideal High-Level FPGA SDR Toolflow

The ideal high-level FPGA SDR tool-flow specification developed in Chapter 4 of this dissertation is used as the baseline criteria in this chapter to assess the capabilities of candidate high-level design tools for developing SDR applications on FPGA platforms. Figure 5.1 shows the logical architecture of such an ideal tool-flow. The specification outlines key requirements, which are grouped into four main categories, namely, high-level modeling, verification, code generation and design implementation. They should be supported by a standard high-level FPGA SDR design tool, according to the current state of the art in electronic design automation and SDR technologies. These requirements are listed in Table 5.1.

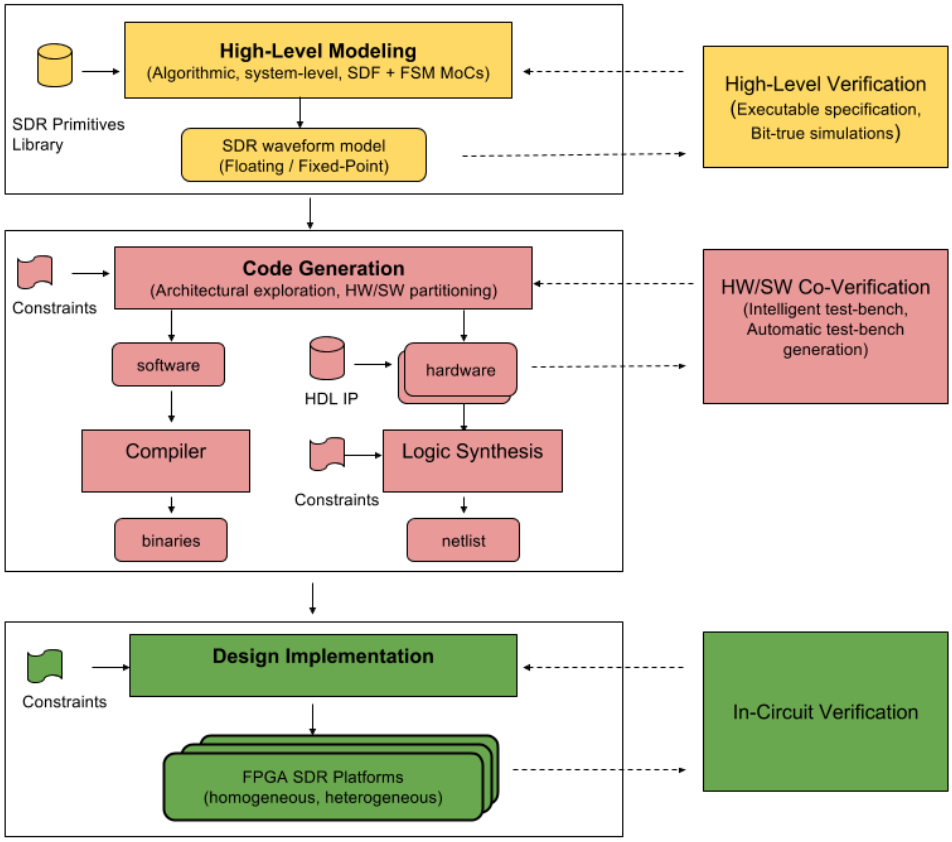

Figure 5.1: Logical architecture of the ideal high-level tool-flow for SDR DSP development on FPGA platforms.

Table 5.1: System requirements for the high-level FPGA-based SDR design flow.

Design Stage	System Requirements
High-Level Modeling	Algorithmic-level design entry
	System-level design entry
	SDR primitives library
	SDR MoCs
	Floating to fixed-point conversion
Design Verification	High-level model simulation
	High-level model formal verification
	HW/SW co-verification
	Automatic test-bench generation
	Intelligent test-bench
	Virtual platforms
	Third-party verification tools integration
Code Generation	Software code generation
	Hardware code generation
	Logic synthesis
	FPGA bitstream generation
	Domain-specific design optimisations
	Generic design optimisations
Design Implementation	FPGA bitstream generation
	Third-party FPGA implementation tools integration

5.1.2 Design Productivity and Performance

Besides the qualitative study, which entails comparing each tool with the ideal tool-flow reference, quantitative metrics are used to study the design productivity of the tool-flows and the performance of the designs they generate. Performance is measured in terms of resource use, area utilisation and maximum speed of the FPGA design. A FIR filter, based on the specification shown in Table 5.2, is used as a case study SDR DSP application. Section 3.2 of the methodology gave detailed descriptions of the quantitative evaluation process and the criteria used for these FPGA tool-flows.

Table 5.2: FIR filter specification

Design criteria	Specification
Type	Lowpass
Passband	0 - 8MHz
Stopband	8 - 10MHz
Order	63
Passband ripple	2.3%
Stopband attenuation	40dB

5.2 Qualitative Study

This section discusses and compares the features of the two tools (MyHDL in Subsection 5.2.1, and Migen in Subsection 5.2.2) in light of the baseline desired features of an ideal tool-flow, namely, high-level modeling, verifi-

cation and code generation. The strengths and limitations of each tool are discussed. Table 5.3 provides a summary of the study.

5.2.1 MyHDL Evaluation

High-Level Modeling Python's power and clarity invest MyHDL with rich high-level and Register Transfer Level (RTL) modeling properties. Both algorithmic and RTL abstraction levels are supported in MyHDL. For hardware to be synthesisable, and subsequently implementable on an FPGA, MyHDL requires it to be modeled at RTL, and a designer should stick to a given convertible subset of the syntax [20, 82]

At the core of MyHDL's support for hardware modeling is the use of Python generators to model hardware concurrency. Generators can best be understood as resumable functions. MyHDL models a hardware module as a function that returns generators. For example, in code listing 5.1, the FIR filter module is described as a function m_firfilt, which returns an explicitly clocked generator function rtl_sop. This approach makes it easy to support key hardware modeling features, such as hierarchy, named port association, arrays of instances, and conditional instantiation. To enhance the Python language for hardware modeling further, MyHDL implements classes that define traditional hardware description concepts, including a signal class to enable communication between generators, an intbv class to support bit oriented operations and an enumeration class.

In addition to the library, which adds support for traditional hardware description features, MyHDL also provides a no re-usable built-in primitives library, which targets a specific hardware design domain, such as SDR. However, in a study[77] , which proves that MyHDL is a viable tool, around which a replacement tool-flow for CASPER to target radio astronomy DSP can be designed, a SDR MyHDL library consisting of primitives, Ten Gigabit Ethernet, analog to digital converter (ADC) controllers and FIR filter DSP components has been implemented.

MyHDL provides limited support for models of computation necessary for efficient high-level modeling of SDR waveforms. Besides Inggs' work in [74], which prototypes a simplified MyHDL-based tool-flow, which targets an FPGA-based SDR platform from SDF-based high-level SDR descriptions, there is no support for the dataflow MoC in MyHDL. Instead, synthesisable hardware is modeled in a discrete-event model of computation at RTL. Support for the finite state machine (FSM) modeling is straightforward through the use of MyHDL's enumeration type and generators.

```
from myhdl import *

def FIR(clock, reset, sig_in, sig_out, coef):
    taps = [Signal(intbv(0, min=sig_in.min, max=sig_in.max))
            for ii in range(len(coef))]
    # FIR taps
    coef = tuple(coef)
    mshift = len(sig_in)-1

    @always(clock.posedge)
    def rtl_sop():
        if reset:
            for ii in range(len(coef)):
                taps[ii].next = 0
            sig_out.next = 0
        else:
            sop = 0
            for ii in range(len(coef)):
                if ii == 0:
                    taps[ii].next = sig_in
                else:
                    taps[ii].next = taps[ii-1]
                c = coef[ii]
                sop = sop + (taps[ii] * c)
            sig_out.next = (sop >> mshift)
    return rtl_sop
```

Listing 5.1: MyHDL FIR filter code example.

Verification MyHDL supports simulation and verification of hardware designs through three main approaches: MyHDL's built-in discrete event simulator, Python's unit test framework, and co-simulation with traditional HDL simulators. Formal verification and intelligent test bench techniques are not supported.

MyHDL's embedded simulator runs ontop of the Python interpreter and supports waveform visualisations, using the value change dump (VCD) standard. Figure 5.2 shows an example VCD output of the FIR filter case study design. The VCD file was generated using Migen, and viewed using GTKwave software.

MyHDL brings the power of Python's unit test framework to digital hardware design. Although popular as a software verification methodology, unit testing is still uncommon in the FPGA design domain. Unit testing has several limitations: it can only follow a limited number of execution paths, and therefore it can only test for the existence of a limited number of errors. In other words, unit testing cannot guarantee the absence of errors [146].

MyHDL can also be used as a hardware verification language for Verilog designs, by co-simulation with external HDL simulators. Co-simulation with any HDL simulator that has a procedural language interface (PLI) is supported. Currently, MyHDL supports co-simulation with two Verilog simulators: Icarus and Cver. MyHDL supports co-simulation so that test benches for HDL designs can be written in Python. Since test benches do not need to be synthesisable, this approach allows Python's rich constructs and scientific libraries, such as Numpy and Scipy, to be applied to SDR hardware design verification.

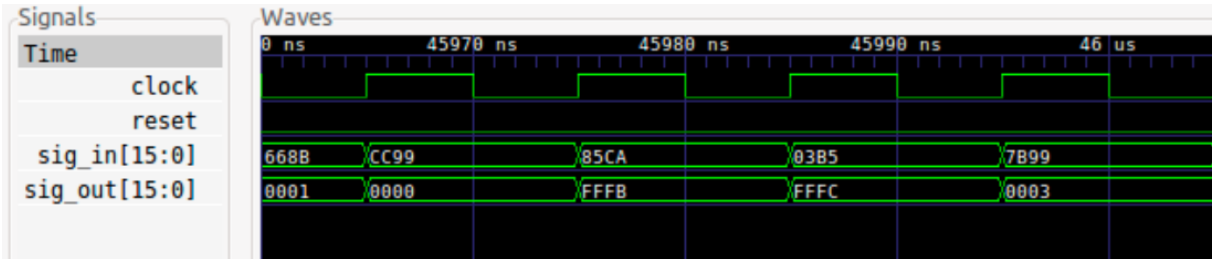

Figure 5.2: MyHDL FIR filter RTL simulation results

Code Generation MyHDL provides tools that facilitate automatic generation of synthesisable and portable Verilog or VHDL code from MyHDL designs. The generated HDL can then be fed into a traditional FPGA design flow, for synthesis, place and route, and deployment on an FPGA platform. As in other HDLs, MyHDL's convertible subset is limited, but much wider than the standard synthesis subset. It includes features that can be used for high-level modeling and test benches.

MyHDL converter operation follows the scheme shown in Figure 5.3. The converter first performs a design elaboration step, without applying convertibility limitations, to obtain a hierarchy of generators in a design. The Python introspection API is then used in a later step to analyse the hierarchy and convert it into VHDL or Verilog. The limitations of the convertible subset are only applied to the internal parts of the generators, thus allowing the rich power of Python to be used outside them.

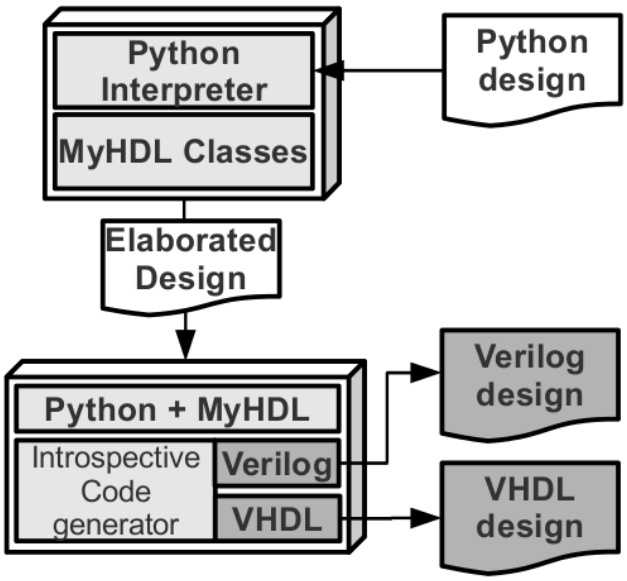

Figure 5.3: VHDL and Verilog code generation scheme used by MyHDL

Listing 5.2 shows a snippet of the FIR filter Verilog code generated using MyHDL from the semi high-level input model. The complete generated code can be seen in Appendix B.1.3. The code is highly readable. It contains 208 SLOC, including the test bench, in comparison to the 120 SLOC for the hand-crafted design. Further, there is a high correlation in identifier names between the different methods and signals in MyHDL and the produced Verilog code. This produces good traceability.

```
1  // File: firfilt.v
2  // Generated by MyHDL 0.8dev
3
4  `timescale 1ns/10ps
5
6  module firfilt (
7      sig_in,
8      sig_out,
9      clk,
10     rst
11 );
12
13
14 input signed [15:0] sig_in;
15 output signed [15:0] sig_out;
16 reg signed [15:0] sig_out;
17 input clk;
18 input rst;
19
20 reg signed [15:0] taps [0:33-1];
```

Listing 5.2: Verilog FIR filter code generated from the high-level MyHDL specification

5.2.2 Migen Evaluation

High-Level Modeling Migen supports modeling of hardware through the Python-based Fragmented Hardware Description Language (FHDL). FHDL is Migen's base language. It provides a formal system to describe signals, and synchronous and combinatorial operations performed on them. FHDL is not based on any standard model of computation. Instead, it seeks to address the limitations of the discrete event paradigm with the notions of combinatorial and synchronous statements, by enabling designers to construct the logic of their designs using a

Python program. FHDL's formal system is low-level and close to the synthesisable subset of Verilog, yet combined with Python, it enables designers to develop complex hardware structures by combining FHDL elements [147, 83].

Modules are supported in MyHDL and play the same role as Verilog modules and VHDL entities. A FHDL module is a Python object that derives from the Module class, which implements special attributes used by child classes to describe their hardware logic. Module hierarchy is supported through submodules, specials and instances. Legacy or third-party HDL code can be reused within Migen's FHDL design through instantiation. MyHDL specials – predefined common blocks – allow reuse of pre-defined Migen modules. Migen's current specials library contains memory and tri-state input/output (I/O) objects. There is no support for SDR primitives, including FFT, modulators, demodulators and filters.

Listing 5.3 shows a high-level model of a FIR filter description written in Migen. As can be seen from the listing, Migen allows the use of Python's rich and simple software programming constructs, such as classes, constructors and loops, together with hardware-specific elements, such as combinatorial (self.comb) and synchronous (self.sync) statements defined in Migen to specify hardware. Unlike in MyHDL, where synchronous components should be explicitly clocked, in Migen, low-level details such as clock management and sensitivity lists are abstracted away from the designer by default and handled automatically.

```
''' A synthesizable FIR filter '''
class FIR(Module):
    def __init__(self, coef, wsize=16):
        self.coef = coef
  # FIR coefficients
        self.wsize = wsize
        self.i = Signal((self.wsize, True))
  # signal for input samples
        self.o = Signal((self.wsize, True))
  # signal for output samples

        ###
        muls = []
        src = self.i
        for c in self.coef:
            sreg = Signal((self.wsize, True))
            self.sync += sreg.eq(src)
      # copy input samples into the FIR register, synchronously
            src = sreg
            coef_fp = int(c*2**(self.wsize - 1))
      # convert coefficient to fixed point
            muls.append(coef_fp*sreg)
      # compute FIR products
        sum_full = Signal((2*self.wsize-1, True))
        self.sync += sum_full.eq(reduce(add, muls))
  # sum the products, synchronously
        self.comb += self.o.eq(sum_full >> self.wsize-1)
  # write output to output port, combinatorially
```

Listing 5.3: Migen FIR filter code example

Verification Verification of high-level Migen models is supported through integration with external simulators. Unlike MyHDL, Migen does not provide an in-built simulator [83].

To verify functional correctness of a given user FHDL model, Migen first interprets it by converting it into RTL (Verilog) and then performs a cycle-accurate simulation using Icarus Verilog. Similarly as in MyHDL, waveform viewing is supported through tracing simulation signals on a VCD file. The rich Python libraries for random test vector generation, unit-testing and waveform plotting provide excellent facilities that enable the easy creation of powerful test-benches. Reuse of the high-level Python test-bench is supported, and communication between Python and the low-level RTL simulators is handled through procedural interfaces, such as Verilog procedural interface (VPI).

Code Generation Migen includes a homogenous converter that automatically generates generic, synthesisable Verilog code from high-level input Python models. VHDL code generation is not supported. The converter accepts the FHDL structure as input. Migen's code-generation engine is basic, and has no support for design-space exploration. Through the Mibuild package, Migen employs scripting to automate seamless integration and interoperation with downstream FPGA implementation tools [147].

Listing 5.4 shows a snippet of the synthesisable Verilog FIR filter code generated automatically from the high-level FHDL description. The generated code describes a circuit of a 16-bit word FIR filter of 30 taps. The complete generated code can be seen in Appendix B.2.3. The code is highly readable and consists of 206 SLOC, which is significantly higher than the 120 SLOC for the hand-crafted Verilog code. Like MyHDL, Migen's code generator ensures a tight correlation between different identifiers in the original Migen design and the generated Verilog code, thus resulting in good traceability.

```
/* Machine-generated using Migen */
module top(
  input signed [15:0] i,
  output signed [15:0] o,
  input sys_clk,
  input sys_rst
);

reg signed [15:0] sreg0 = 1'd0;
reg signed [15:0] coefficient0 = 1'd0;
reg signed [15:0] sreg1 = 1'd0;
reg signed [15:0] coefficient1 = 1'd0;
reg signed [15:0] sreg2 = 1'd0;
reg signed [15:0] coefficient2 = 1'd0;
reg signed [15:0] sreg3 = 1'd0;
reg signed [15:0] coefficient3 = 1'd0;
reg signed [15:0] sreg4 = 1'd0;
reg signed [15:0] coefficient4 = 1'd0;
reg signed [15:0] sreg5 = 1'd0;
reg signed [15:0] coefficient5 = 1'd0;
reg signed [15:0] sreg6 = 1'd0;
reg signed [15:0] coefficient6 = 1'd0;
reg signed [15:0] sreg7 = 1'd0;
reg signed [15:0] coefficient7 = 1'd0;
reg signed [15:0] sreg8 = 1'd0;
reg signed [15:0] coefficient8 = 1'd0;
reg signed [15:0] sreg9 = 1'd0;
reg signed [15:0] coefficient9 = 1'd0;
reg signed [15:0] sreg10 = 1'd0;
reg signed [15:0] coefficient10 = 1'd0;
reg signed [15:0] sreg11 = 1'd0;
reg signed [15:0] coefficient11 = 1'd0;
reg signed [15:0] sreg12 = 1'd0;
reg signed [15:0] coefficient12 = 1'd0;
reg signed [15:0] sreg13 = 1'd0;
reg signed [15:0] coefficient13 = 1'd0;
reg signed [15:0] sreg14 = 1'd0;
reg signed [15:0] coefficient14 = 1'd0;
reg signed [15:0] sreg15 = 1'd0;
reg signed [15:0] coefficient15 = 1'd0;
reg signed [15:0] sreg16 = 1'd0;
reg signed [15:0] coefficient16 = 1'd0;
reg signed [15:0] sreg17 = 1'd0;
reg signed [15:0] coefficient17 = 1'd0;
reg signed [15:0] sreg18 = 1'd0;
reg signed [15:0] coefficient18 = 1'd0;
reg signed [15:0] sreg19 = 1'd0;
reg signed [15:0] coefficient19 = 1'd0;
reg signed [15:0] sreg20 = 1'd0;
reg signed [15:0] coefficient20 = 1'd0;
reg signed [15:0] sreg21 = 1'd0;
reg signed [15:0] coefficient21 = 1'd0;
reg signed [15:0] sreg22 = 1'd0;
reg signed [15:0] coefficient22 = 1'd0;
```

```
55 reg signed [15:0] sreg23 = 1'd0;
56 reg signed [15:0] coefficient23 = 1'd0;
57 reg signed [15:0] sreg24 = 1'd0;
58 reg signed [15:0] coefficient24 = 1'd0;
59 reg signed [15:0] sreg25 = 1'd0;
60 reg signed [15:0] coefficient25 = 1'd0;
61 reg signed [15:0] sreg26 = 1'd0;
62 reg signed [15:0] coefficient26 = 1'd0;
63 reg signed [15:0] sreg27 = 1'd0;
64 reg signed [15:0] coefficient27 = 1'd0;
65 reg signed [15:0] sreg28 = 1'd0;
66 reg signed [15:0] coefficient28 = 1'd0;
67 reg signed [15:0] sreg29 = 1'd0;
68 reg signed [15:0] coefficient29 = 1'd0;
69 reg signed [15:0] sreg30 = 1'd0;
70 reg signed [15:0] coefficient30 = 1'd0;
71 reg signed [15:0] sreg31 = 1'd0;
72 reg signed [15:0] coefficient31 = 1'd0;
73 reg signed [15:0] sreg32 = 1'd0;
74 reg signed [15:0] coefficient32 = 1'd0;
75 reg signed [15:0] sreg33 = 1'd0;
76 reg signed [15:0] coefficient33 = 1'd0;
77 reg signed [15:0] sreg34 = 1'd0;
78 reg signed [15:0] coefficient34 = 1'd0;
79 reg signed [15:0] sreg35 = 1'd0;
80 reg signed [15:0] coefficient35 = 1'd0;
81 reg signed [15:0] sreg36 = 1'd0;
82 reg signed [15:0] coefficient36 = 1'd0;
83 reg signed [15:0] sreg37 = 1'd0;
84 reg signed [15:0] coefficient37 = 1'd0;
85 reg signed [15:0] sreg38 = 1'd0;
86 reg signed [15:0] coefficient38 = 1'd0;
87 reg signed [15:0] sreg39 = 1'd0;
88 reg signed [15:0] coefficient39 = 1'd0;
89 reg signed [15:0] sreg40 = 1'd0;
90 reg signed [15:0] coefficient40 = 1'd0;
91 reg signed [15:0] sreg41 = 1'd0;
92 reg signed [15:0] coefficient41 = 1'd0;
93 reg signed [15:0] sreg42 = 1'd0;
94 reg signed [15:0] coefficient42 = 1'd0;
95 reg signed [15:0] sreg43 = 1'd0;
96 reg signed [15:0] coefficient43 = 1'd0;
97 reg signed [15:0] sreg44 = 1'd0;
98 reg signed [15:0] coefficient44 = 1'd0;
99 reg signed [15:0] sreg45 = 1'd0;
100 reg signed [15:0] coefficient45 = 1'd0;
101 reg signed [15:0] sreg46 = 1'd0;
102 reg signed [15:0] coefficient46 = 1'd0;
103 reg signed [15:0] sreg47 = 1'd0;
104 reg signed [15:0] coefficient47 = 1'd0;
105 reg signed [15:0] sreg48 = 1'd0;
106 reg signed [15:0] coefficient48 = 1'd0;
107 reg signed [15:0] sreg49 = 1'd0;
108 reg signed [15:0] coefficient49 = 1'd0;
109 reg signed [15:0] sreg50 = 1'd0;
110 reg signed [15:0] coefficient50 = 1'd0;
111 reg signed [15:0] sreg51 = 1'd0;
112 reg signed [15:0] coefficient51 = 1'd0;
113 reg signed [15:0] sreg52 = 1'd0;
114 reg signed [15:0] coefficient52 = 1'd0;
115 reg signed [15:0] sreg53 = 1'd0;
116 reg signed [15:0] coefficient53 = 1'd0;
117 reg signed [15:0] sreg54 = 1'd0;
118 reg signed [15:0] coefficient54 = 1'd0;
119 reg signed [15:0] sreg55 = 1'd0;
120 reg signed [15:0] coefficient55 = 1'd0;
121 reg signed [15:0] sreg56 = 1'd0;
122 reg signed [15:0] coefficient56 = 1'd0;
123 reg signed [15:0] sreg57 = 1'd0;
```

```
124 reg signed [15:0] coefficient57 = 1'd0;
125 reg signed [15:0] sreg58 = 1'd0;
126 reg signed [15:0] coefficient58 = 1'd0;
127 reg signed [15:0] sreg59 = 1'd0;
128 reg signed [15:0] coefficient59 = 1'd0;
129 reg signed [15:0] sreg60 = 1'd0;
130 reg signed [15:0] coefficient60 = 1'd0;
131 reg signed [15:0] sreg61 = 1'd0;
132 reg signed [15:0] coefficient61 = 1'd0;
133 reg signed [15:0] sreg62 = 1'd0;
134 reg signed [15:0] coefficient62 = 1'd0;
135 reg signed [30:0] sum_full = 1'd0;
136
137 // synthesis translate_off
138 reg dummy_s;
139 initial dummy_s <= 1'd0;
140 // synthesis translate_on
141 assign o = (sum_full >>> 4'd15);
142
143 always @(posedge sys_clk) begin
144   if (sys_rst) begin
145     sreg0 <= 1'd0;
146     coefficient0 <= 1'd0;
147     sreg1 <= 1'd0;
```

Listing 5.4: Verilog FIR filter code generated using Migen.

.

Table 5.3: Evaluation of high-level design tools for FPGA-based SDR design.

Design Stage	System Requirements	Migen	MyHDL
High-Level Modeling	Algorithmic-level design entry	✓	✓
	Hierarchical design entry	✓	✓
	SDR primitives library	✗	✗
	SDR MoCs	✓	✗
	Automatic floating-fixed-point conversion	✗	✗
Design-Verification	High-level model simulation?	✓	✓
	High-level formal verification	✗	✗
	HW/SW co-verification	✗	✗
	Automatic TB generation	✗	✓
	Intelligent TB	✗	✗
	HDL simulation	✗	✓
	Virtual platforms	✗	✗
	External verification tools integration	✓	✓
Code-Generation	SW code generation	✗	✗
	HW code generation	✓	✓
	Logic synthesis	✗	✗
	Domain design optimisations	✗	✗
	Generic design optimisations	✗	✗
Design-Implementation	FPGA bitstream generation	✗	✗
	External FPGA implementation tools integration	✓	✗

5.3 DESIGN PRODUCTIVITY AND PERFORMANCE STUDY

This section presents a comparative study of the two tool-flows in terms of design productivity, and the performance of the designs they produce. The goal of high-level design methodologies is to increase design productivity, without significant degradation in design performance.

5.3.1 Performance Results

This subsection presents the performance results of MyHDL and Migen tool-flows. The performance results provide a measure of the quality of the HDL designs generated from the two tools. For both MyHDL and Migen, the generated Verilog FIR code was synthesised and implemented on Xilinx's xc7vx330t- 3ffg1157 FPGA device using ISE 14.7, and the performance results in terms of logic utilisation and the maximum achievable design speed were recorded. Logic utilisation is measured under the number of slice registers, slice LUTs, LUT flip flop pairs, IOBs, and DSP cores. Power consumption was estimated using Xilinx Power Estimator (XPE).

5.3.1.1 MyHDL

Table 5.4 shows the performance of MyHDL relative to that of a hand-crafted FIR Verilog design.

The results show that the hand-crafted design is significantly better than the MyHDL design under all the metrics. For example, the number of slice registers, LUTs and LUT flip flops is almost twice that of the hand-crafted design. The total number of IOBs used by both designs is 34. The MyHDL design has a maximum clock frequency of 14.6MHz, which is significantly lower than the maximum of 904.2MHz for the hand-crafted design.

Table 5.4: Performance of FIR filter VHDL code generated using MyHDL against that of a hand-crafted design

	Hand-crafted FIR	**MyHDL FIR**
SLOC	120	208
Slice registers	150 (1%)	136 (1%)
Slice LUTs	141 (1%)	124 (1%)
IOBs	34 (5%)	34 (5%)
DSP48E1s	1 (1%)	63 (5%)
Minimum period	1.106ns	68.295ns
Maximum frequency	904.2MHz	14.6MHz
Power consumption	0.177W	0.177W

*5.3.1.2 **Migen***

Performance results of the generated Migen design against the hand-crafted FIR HDL are shown in Table 5.5.

The results show that the Migen design logic utilisation is almost equal to that of the hand-crafted design. The Migen design has a maximum clock frequency of 15.2MHz which is significantly lower than the maximum of 904.2MHz for the hand-crafted design.

Table 5.5: Performance of FIR filter Verilog code generated using Migen.

	Hand-crafted FIR	**Migen FIR**
SLOC	120	206
Slice registers	150	168
Slice LUTs	141	147
IOBs	34	34
DSP48E1s	1	63
Minimum period	1.106ns	65.979ns
Maximum frequency	904.2MHz	15.2MHz
Power consumption	0.177W	0.177W

5.3.1.3 Comparison

The comparative performance results, in terms of target device utilization, between Verilog code generated automatically using MyHDL and Migen tools, are summarised in Figure 5.4.

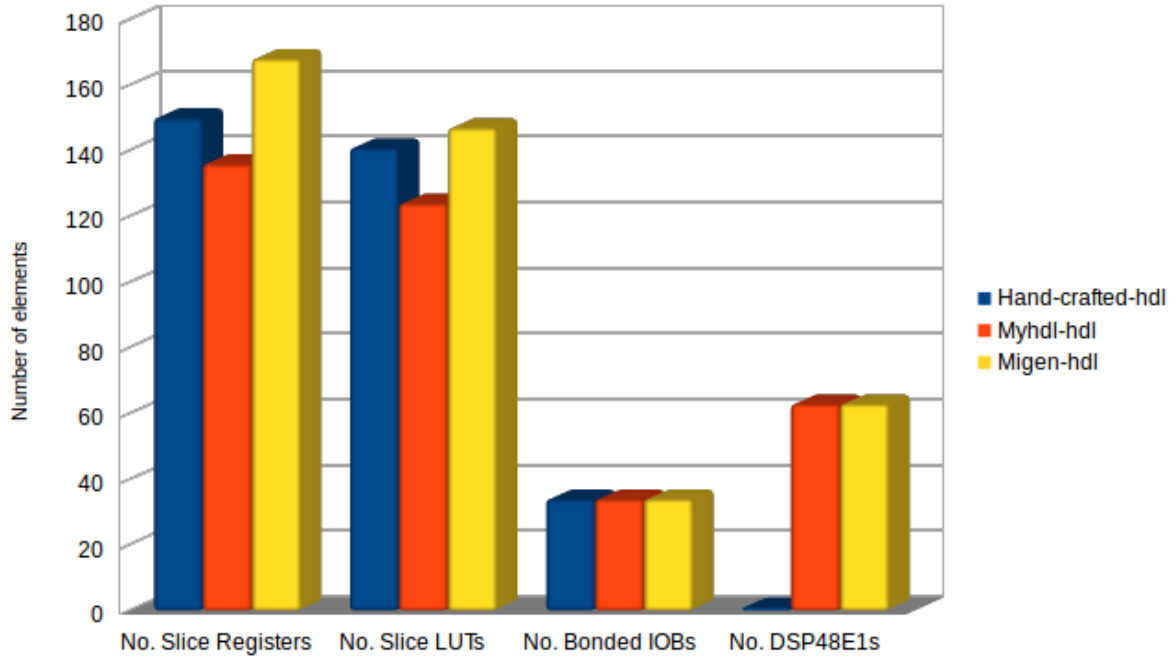

Figure 5.4: FIR filter FPGA utilization results for migen and MyHDL designs

The results indicate that MyHDL-generated design device utilisation is comparable to and somewhat better than that of the hand-crafted Verilog design under all metrics, except under DSP slices. The design produced using Migen has the highest device utilisation under all the metrics. Therefore, these results suggests that in terms of design are efficiency, MyHDL is slightly better than Migen. According to Tables 5.4 and 5.5, the final MyHDL and Migen designs have speeds of 14.6MHz and 15.2MHz respectively. Migen is thus slightly better than MyHDL in terms of speed.

5.3.2 Design Productivity Results

The results with regard to the effort of developing Migen and MyHDL FIR waveforms and their associated test-benches are shown in Table 5.7.. The results are estimates, obtained by logging the time taken to code various parts of the case studies. The time required to describe the FIR filter in Migen is split into 4 for design, and 18 for developing the test-bench and debugging the filter. The time required to describe the FIR filter in MyHDL is split into 8 for design, and 20 for developing the test-bench and debugging the filter. The results show that Migen has an approximate SLOC rate of 4, whereas MyHDL has 3. This suggests that Migen is more suited to fast prototyping than MyHDL.

Table 5.6: Summary of coding effort for developing and testing the FIR filter waveform in Migen and MyHDL.

Design component	**Coding effort**	
	Migen	***MyHDL***
FIR Module	21SLOC (4h)	26SLOC (8h)
Testbench	59SLOC (18h)	50 (20h)
Total	80SLOC (22h)	76SLOC(28h)

Table 5.8 shows the respective design productivity for Migen and MyHDL with respect to manual design. Section 3.2.4.1 describes the method and the tools used to determine design productivity. A design productivity of 1.6x is measured for Migen and 1.4x for MyHDL. This suggests that Migen is slightly more suited for quick prototyping of SDR DSP than MyHDL. Conversely, the MyHDL based design suffers from 1.1 in quality loss compared to 1.3 for the Migen design. Therefore, although slightly less productive than Migen, MyHDL is slightly better than Migen in a case where design quality and not productivity is the design goal.

Table 5.7: Summary of coding effort and performance results for the FIR filter waveform in Migen and MyHDL.

	Migen	***MyHDL***
NRE_{dt}	4	8
NRE_{vt}	18	20
SLOCs	80	76
Chars	2650	6263
No. LUTs	147	124
No. Registers	168	136
No. DSP	63	63
Period (ns)	65.979	68.295
Frequency (MHz)	15.2	14.6

Table 5.8: Gain in NRE design time G_{NRE}, quality loss L_Q and design productivity P_D of Migen vs manual HDL and MyHDL vs manual HDL

	Migen	***MyHDL***
G_{NRE}	2	1.6
L_Q	1.3	1.1
P_D	1.6x	1.4x

5.4 CHAPTER SUMMARY

In this chapter, two high-level tool-flows for rapid prototyping of SDR DSP operations on FPGA platforms were evaluated. The features of Migen and MyHDL tool-flows were compared and discussed against the proposed ideal methodology. The results reveals that Migen out-performs MyHDL in terms of high modelling features, whereas MyHDL is better than Migen when it comes to verification facilities. Both tools are lacking in design implementation facilities.

The tool-flows were further assessed and compared in terms of design productivity and design performance. A FIR filter case study was used to assess these. MyHDL implementation was found to be more efficient than the Migen implementation in terms of logic utilization performance. Migen, however, slightly out-performed MyHDL in terms of design speed. Design productivity of 1.6x was computed for Migen and 1.4x for MyHDL. The scheme used to measure design productivity is based on a trade-off between design efficiency and quality. Therefore, the study suggests that Migen is generally slightly better than MyHDL in terms of providing a balance between fast prototyping and producing quality designs.

CHAPTER 6

EVALUATION OF TOOL-FLOWS FOR RAPID PROTOTYPING SDR DSP ON GPUS

This chapter presents a study of high-level tool-flows for rapid prototyping of SDR DSP operations on GPUs. Qualitative and quantitative methodologies are used to study and compare the tool-flows, which are evaluated in terms of their features, against the ideal high-level SDR DSP tool-flow specification developed in Chapter 4. The aim is to determine the extent to which each of the tools supports the key modelling, verification and implementation features necessary for SDR DSP rapid prototyping on GPUs. In addition, a FIR filter case study is developed, using each of the tools, and used to evaluate and compare the tools in terms of productivity and quality of designs produced.

Section 6.1 describes the evaluation criteria, consisting of both the ideal high-level SDR DSP tool- flow specification and the FIR filter case study. Qualitative evaluations of the tool-flows are presented in Section 6.2. Sections 6.3 and 6.4 evaluates the tools in terms of their respective coding effort and performance of the application. Lastly, a summary of the findings of the study is given in Section 6.5..

6.1 EVALUATION CRITERIA

The criteria used to study the tool-flows in this chapter comprise two main categories, viz. qualitative and quantitative. The qualitative criteria are used to determine the extent to which the tools support the desirable features specified in the ideal tool-flow specification. The quantitative study, which is based on a FIR filter case study, is used to evaluate the actual functionalities of the tools.

6.1.1 Proposed Ideal High-Level GPU SDR Design Flow

The proposed ideal high-level SDR design methodology for GPU targets, shown in Figure 6.1 below, provides baseline criteria for evaluating selected GPU programming models in this chapter. The methodology captures clearly, yet simply, the key design goals that should be incorporated in standard SDR GPU design tools: high-level modeling, design implementation and design testing and verification. The surveys assess the design properties of each of the selected GPU design tools against the proposed ideal methodology.

6.1.2 Design Productivity and Performance

Apart from the qualitative study, which entails discussing each tool against the ideal tool-flow reference, quantitative metrics are used to study the design productivity of the tool-flows and the performance of the designs they generate. Performance is measured in terms of kernel execution time, memory transfer time, throughput, and power consumption of the produced GPU design. A FIR filter of the specification shown in Table 6.1 is used as a case

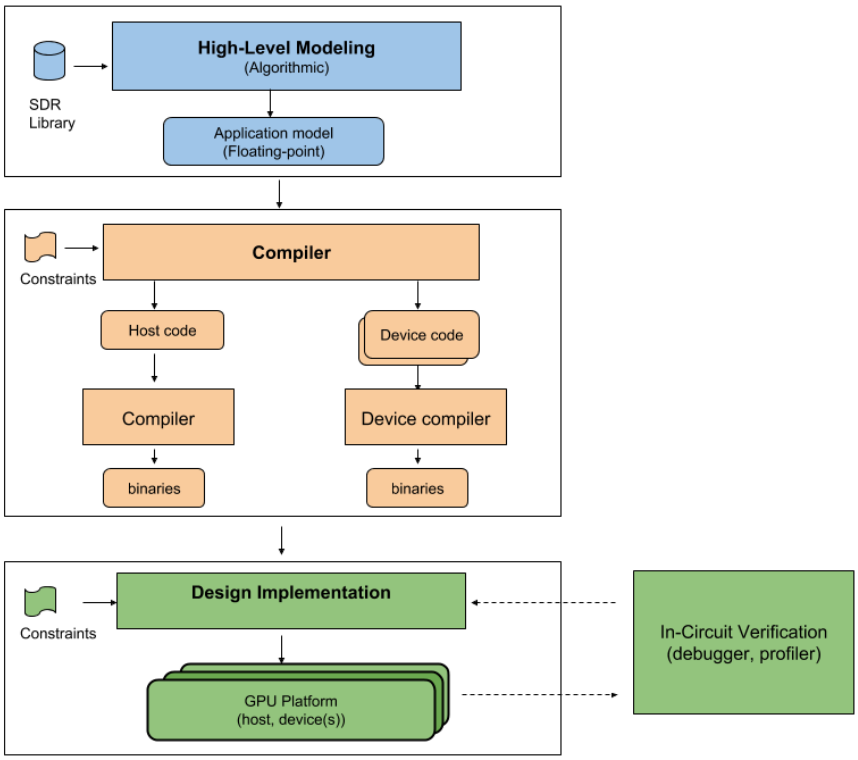

Figure 6.1: Logical architecture of the ideal high-level tool-flow for SDR DSP development on GPUs.

study SDR DSP application. Section 3.3 of the methodology provides a detailed description of the quantitative evaluation process and criteria for the GPU tool-flows.

Table 6.1: FIR filter specification.

Design criteria	**Specification**
Type	Lowpass
Passband	0 - 8MHz
Stopband	8 - 10MHz
Order	63
Passband ripple	2.3%
Stopband attenuation	40dB

6.2 QUALITATIVE STUDY

The proposed ideal high-level SDR design methodology for GPU targets provides the baseline criteria for evaluating selected GPU programming models in the subsequent subsections. The methodology captures, in detail, key design goals that should be incorporated in standard SDR GPU design tools under four categories: high-level modeling, design implementation and design testing and verification. These tool-flow evaluations are carried out by assessing the design properties of each of the selected GPU design tools against the proposed ideal methodology.

6.2.1 CUDA

High-Level Modeling CUDA provides a software layer that enables developers to write C/C++ high-level application software that transparently scales its parallelism to leverage existing number of processing cores on a GPU device. The layer provides three key abstractions – a hierarchy of thread groups, shared memories and barrier synchronization. These are exposed to the programmer as a minimal set of language extensions that grant direct access to the GPU's virtual instruction set and parallel computation elements for the execution of compute kernels.

CUDA contains over 5000 GPU-accelerated primitives that provide highly optimised functions in various domains including linear algebra, image and signal processing (SDR).

The CUDA API provides keywords that let kernels get indices of the running threads: threadIdx.x and blockDim.x. Specifically, threadIdx.x gives the index of the current thread within its block and blockDim.x stores the number of threads in the block. As shown in the code listing , these keywords are used to slice the incoming block of data samples across multiple FIR processing threads. The C++'s 64 bit double precision floating point type was used to store the filter coefficients, input and output filter data.

Listing 6.1 shows a code snippet of a CUDA implementation of a FIR filter.

```
/*
   CUDA FIR filter kernel
   d_data: input sample on device pinned memory
   d_ffCoeff: FIR coefficients on device pinned memory
   d_filteredData: FIR output on pinned device
*/

__global__ void filterData(double *d_data,
                           double *d_ffCoeff,
                           double *d_filteredData,
                           const int ffLength,
                           const int filteredDataLength){
    double sum = 0.00;
    double *d_ffCoeffp=d_ffCoeff;
    double *d_datap=d_data;

    int index=threadIdx.x;
    // set index of current thread in the block
```

```
    int stride=blockDim.x;
    // set threads to process data in equal chucks according to their number
    for (int  n = index; n < filteredDataLength; n+=stride ) {
      // let this thread begin at it's index and compute FIR output
      d_ffCoeffp=d_ffCoeff;
      double * activeinsamp=&d_datap[ffLength-1+n];
      sum=0.0f;
      for(int i=0;i<ffLength;i++){
  // compute FIR sum of products
  sum+=(*d_ffCoeffp++)*(*activeinsamp--);
      }
      d_filteredData[n]=sum;
      // store result of this thread
    }
}
```

Listing 6.1: CUDA FIR filter implementation.

Implementation Implementation of an application from a high-level CUDA C/C++ specification is a straightforward process, which is achieved through NVIDIA's CUDA Compiler (NVCC). NVCC is based on the widely used LLVM open source compiler infrastructure, and translates application specifications written in CUDA C/C++ into parallel thread execution (PTX) code, which the GPU driver compiles into a suitable binary that can be run on the processing cores.

Verification CUDA supports an executable specification. A CUDA application is written in C/C++ language, and augmented with several functions and types and other constructs to support GPU programming. Functional validation of the application can be easily done by compiling and executing the program and analysing the output.

6.2.2 OpenCL

High-level Modeling OpenCL is a framework for developing applications that execute across a range of device types made by different vendors. It supports a wide range of levels of parallelism and efficiently maps to homogeneous or heterogeneous, single- or multiple-device systems, consisting of CPUs, GPUs, and other types of devices, which are limited only by the imagination of vendors. The OpenCL definition offers both a device-side language and a host management layer for the devices in a system.

The OpenCL version of the FIR waveform, shown in Listing 6.2, was coded in C++. Similar to the CUDA API, OpenCL provides methods that enable software developers to exploit data-level parallelism in algorithms: get_local_id, get_global_id. Specifically, get_local_id(a), returns the unique work-item ID within a specific work-group for dimension a and get_global_id(b) gives the unique global work-item ID value for dimension b.

```
/*
   OpenCL FIR filter kernel
   input: input samples
   coeff: FIR coefficients
   output: FIR output samples
   lengths: threads block dimensions
*/
__kernel void fir(__global double* input,
          __global double* output,
    __global  double* coeff,
     __global int* lengths) {

                int idx, acc;
    acc = 0;
    idx = get_global_id(0);
        // set index of this thread in the block
                double sumprod=0;
    /* compute FIR sum of products for this thread */
    for(int j = 0; j<lengths[1]; j++){
```

```
20        sumprod += input[idx-j] * coeff[j];
21      }
22      // store result in output buffer
23      output[idx] = sumprod;
24  }
```

Listing 6.2: OpenCL FIR filter implementation

Verification Functional validation of OpenCL's high-level waveform specifications can be simply performed by compiling and executing the C/C++ specification. In addition, OpenCL provides various tools for application debugging and profiling.

An OpenCL SDR DSP application can include kernels and a large amount of IO between the host and the device. Profiling such an application can help to improve performance by helping to identify bottlenecks and parallelisable parts of the application. The OpenCL API provides some basic features for application profiling, and also provides facilities for using the operating system's APIs to time sections of code. For example, AMD Accelerated Parallel Processing (APP) Profiler [148] is a performance analysis tool that gathers data from the OpenCL runtime and AMD Radeon GPUs during the execution of an OpenCL application. AMD APP KernelAnalyzer [149] is a static analysis tool for compiling, analyzing and disassembling an OpenCL kernel for AMD Radeon GPUs.

Implementation OpenCL currently supports CPUs that include x86, ARM, and PowerPC, and it has been adopted into graphics card drivers by both AMD (where it is called the APP SDK) and NVIDIA. AMD APP SDK is a software development kit by AMD for accelerated parallel processing (APP) providing support for not only GPUs, but also heterogeneous platforms. Support for OpenCL is rapidly expanding, as a wide range of platform vendors have adopted OpenCL, and support or plan to support it for their hardware platforms. These vendors fall within a wide range of market segments, from the embedded vendors (ARM and Imagination Technologies) to the high performance computing (HPC) vendors (AMD, Intel, NVIDIA, and IBM). The architectures supported include multi-core CPUs, throughput and vector processors, such as GPUs, and fine-grained parallel devices, such as FPGAs.

6.3 Coding Effort Results

A comparison of the coding effort required to develop both the kernel and the host test bench parts of the FIR filter in CUDA and OpenCL is given in Table 6.2. The results are approximates obtained by logging the amount of development time spent on each tool. Time spent by the author researching and getting to know the three tools was not included.

Table 6.2: Summary of coding effort for developing and testing the FIR filter waveform in CUDA and OpenCL.

Design component	**Coding effort**	
	CUDA	***OpenCL***
Kernel	13SLOC (0.5h)	8SLOC (0.5h)
Host test bench	200SLOC (28h)	330 SLOC (96h)
Total	213SLOC (28.5h)	338SLOC (96.5h)

6.4 Performance Results

This section presents the performance results of the CUDA and OpenCL tool-flows. The performance of the tools is measured in terms of kernel and memory transfer times, throughput and power consumption.

6.4.1 Execution Times

Table 6.3 shows the times taken by both the host CPU and GPU to process a given number of test input samples for CUDA and OpenCL programming methodologies under the paged host memory and paged device memory transfer configuration. From the results, it can be seen that the CUDA-based FIR application is always better than the OpenCL counterparts in both kernel processing and memory transfer efficiency. In particular, the results depict that CUDA is approximately six times better than OpenCL in terms of kernel execution and two times in terms of data transfer performance. CUDA's higher kernel execution rate shows that CUDA, compared to OpenCL, is able to parallelise the kernel more efficiently on the NVIDIA device.

CUDA's out-performance of OpenCL in both kernel processing and transfer efficiency can be explained in terms of CUDA's alignment and specialisation to the test platform architecture, NVIDIA GPU. CUDA is designed for NVIDIA's GPUs whereas OpenCL is more generic and flexible, not tied to any GPU architecture, but targets heterogeneous platforms comprising of a combination of GPPs, DSPs, FPGAs and GPUs. Therefore, the CUDA-based FIR kernel, in comparison to the OpenCL-based one, is able to extract more performance out of the GPU by taking advantage of the uniform and well-known NVIDIA GPU architecture.

Table 6.3: FIR filter kernel and memory transfer times for CUDA and OpenCL under paged host to device and device to host memory transfer.

Input samples	**CUDA GPU (ms)**	**CUDA Mem (ms)**	**OpenCL GPU (ms)**	**OpenCL Mem (ms)**
8k	0.088482	0.023522	0.482	0.033
16k	0.149581	0.047149	0.899	0.072
32k	0.269409	0.087457	1.694	0.132
64k	0.478915	0.168387	3.326	0.257
128k	0.935561	0.33057	6.555	0.505

Figure 6.2 shows the percentage time spent executing the FIR filter kernel on the GPU versus that taken transferring data to and from between the GPU and the host CPU for the CUDA tool-flow. A similar graph is shown in Figure 6.3 for the OpenCL tool-flow. For CUDA, the results show that the percentage amount of time spent on memory transfer operations increases with an increase in input data size. With OpenCL on ther other hand, the percentage amount of time spent either processing a kernel or transferring memory to or from the host remains generally constant throughout different input sizes. This implies that, of the three main parallel speed-up responses, namely super-linear, linear and sub-linear, both OpenCL and CUDA are sub-linear and thus scale well with an increase in problem size. However, the results further depict that CUDA's computational ability wears off faster than OpenCL's with an increase in input size, suggesting that OpenCL, though slower than CUDA, is more robust.

Table 6.4 shows the times taken by both the host CPU and GPU to process a given number of test input samples for CUDA and OpenCL programming methodologies under the pinned host memory and pinned device memory transfer configuration. According to the results, the CUDA-based application still outperforms its OpenCL counterpart under both kernel processing and memory transfer for all input sizes. The results further show that the kernel performance of the CUDA-based application has approximately doubled, with a change from the paged to the pinned memory transfer scheme. CUDA is now atleast ten times faster than OpenCL with pinned memory. OpenCL's kernel performance on the other hand, is generally not benefited by the change from paged to pinned memory transfer scheme. This implies that the CUDA programming interface, because of being tailored for the NVIDIA devices only, is able to detect pinned memory configuration on such device and use it to optimise kernel execution.

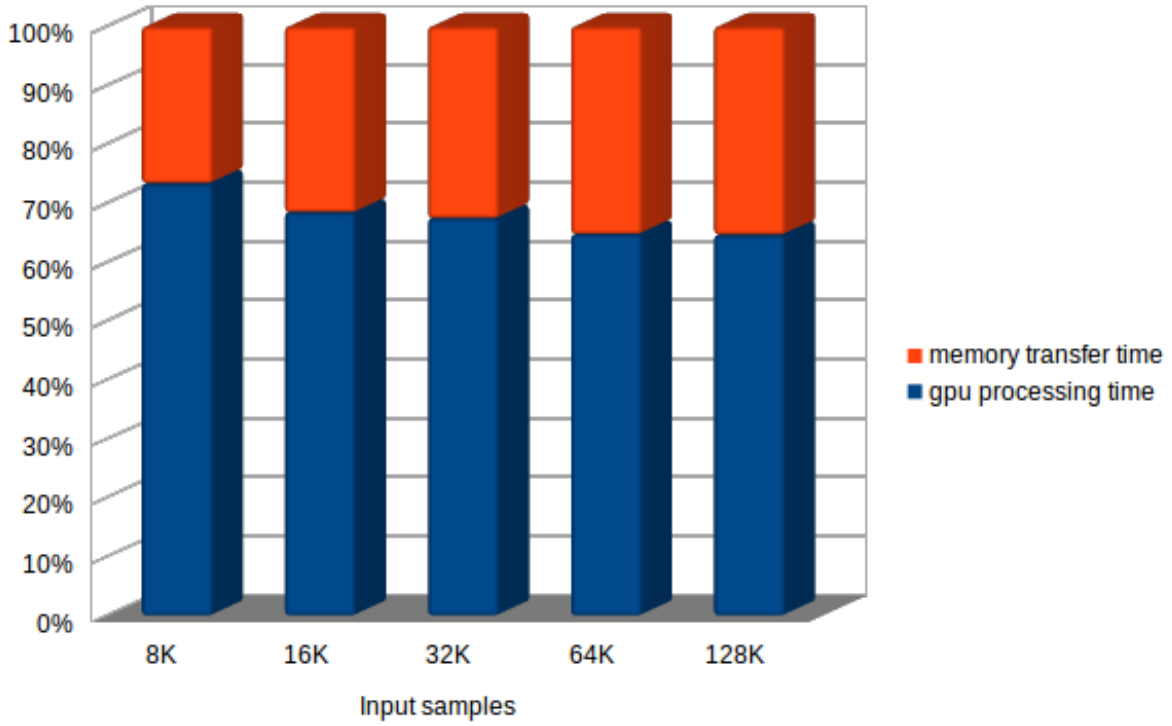

Figure 6.2: Percentage of time taken by the GPU processing the FIR filter kernel vs time taken performing memory transfers between host and device for increasing input sizes under CUDA.

Table 6.4: FIR filter kernel and memory transfer times for CUDA and OpenCL under pinned host to device and device to host memory transfer.

Input samples	**CUDA GPU (ms)**	**CUDA Mem (ms)**	**OpenCL GPU (ms)**	**OpenCL Mem (ms)**
8k	0.04272	0.027232	0.482	0.033
16k	0.081474	0.04752	0.899	0.072
32k	0.15799	0.081938	1.694	0.132
64k	0.2891	0.16889	3.326	0.257
128k	0.58281	0.33051	6.555	0.505

6.4.2 Throughput

The throughputs of CUDA and OpenCL are compared in Figure 6.4. Throughput gives a measure of the rate of processing given input data. Given that the input samples were stored as 8 bits integers, throughput in kilobits per second was computed.

The results show that, for each problem size, the CUDA FIR waveform processes more bits per second than the OpenCL waveform. In fact, the throughput of the CUDA version increases proportionately to the problem size, until the largest problem size of 128k samples, where it begins to diminish. Conversely, the throughput of the OpenCL version remains virtually constant for all the problem sizes.

6.4.3 Power Consumption

Table 6.5 shows the GPU power consumption results for CUDA and OpenCL for an increasing number of input samples. Each measurement was taken ten times and the average recorded in order to increase the precision of the results. K20Power only measures the time taken by the GPU to process the FIR kernel for given input samples and thus the results do not include the power drawn by the CPU in transferring input data to and output data from the device. The results suggests that CUDA is slightly more power efficient that OpenCL.

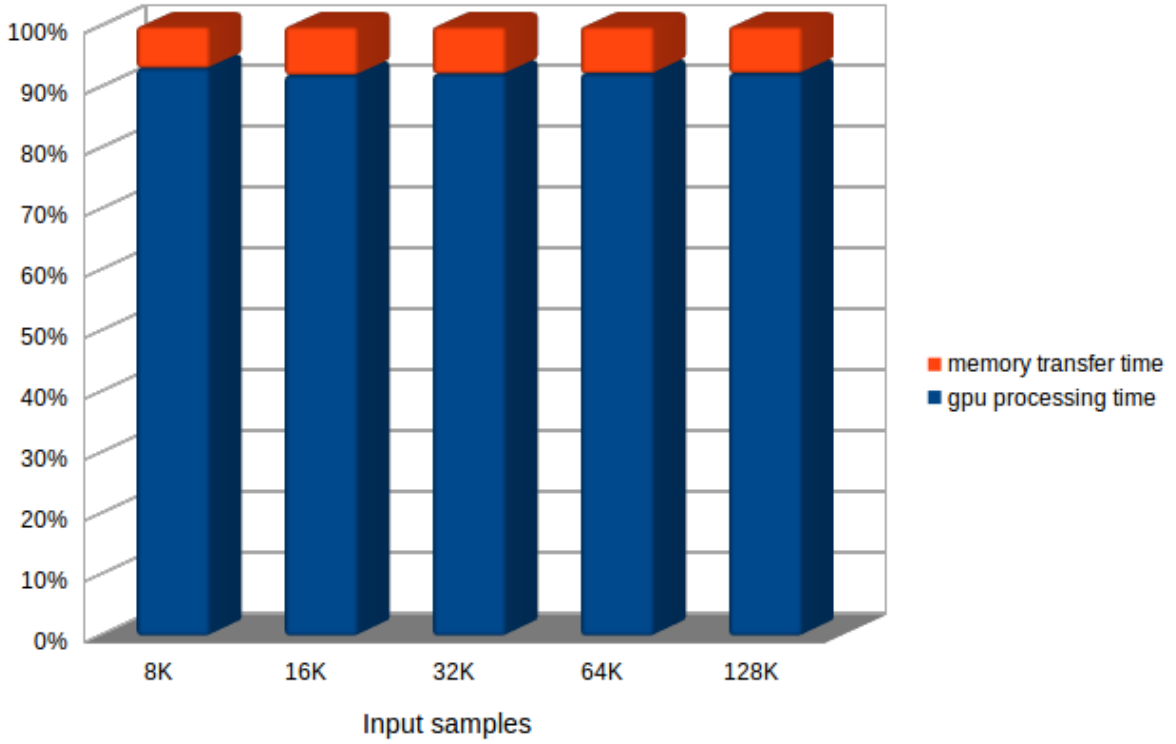

Figure 6.3: Percentage of time taken by the GPU processing the FIR filter kernel vs time taken performing memory transfers between host and device for increasing input sizes under OpenCL.

Table 6.5: Power consumption results for CUDA and OpenCL

Tool	Inputs	Power (W)
CUDA	8k	72.7265265419
	16k	72.8264906761
	32k	72.549415315
	64k	73.236578341
	128k	71.4304684381
OpenCL	8k	73.3289380378
	16k	71.867180597
	32k	72.5935341095
	64k	71.7220846855
	128k	72.9625904528

6.5 CHAPTER SUMMARY

In this chapter, a comparative study of two high-level tool-flows, namely, CUDA and OpenCL, for rapid prototyping of SDR DSP functions on GPU platforms has been presented. The study first involved a qualitative analysis of the features of each tool, comparing them against baseline criteria. Furthermore, a FIR filter case study was implemented, using each of the tools; this was used to evaluate the performance and programming effort associated with each tool.

The study has revealed that, in terms of fast prototyping, CUDA out-performs OpenCL by enabling the designer to quickly write a bit more SLOC than OpenCL. CUDA is more high-level than OpenCL and provides more abstractions of the underlying low-level architectural details. In addition, the performance of the CUDA-based application is better than its OpenCL equivalent. CUDA is optimised for specific GPU devices (NVIDIA) and therefore in comparison to OpenCL, which is designed for generic heterogeneous hardware platforms, it is able to extract higher degrees of computational and transfer performance on an NVIDIA test platform as is the case in this study.

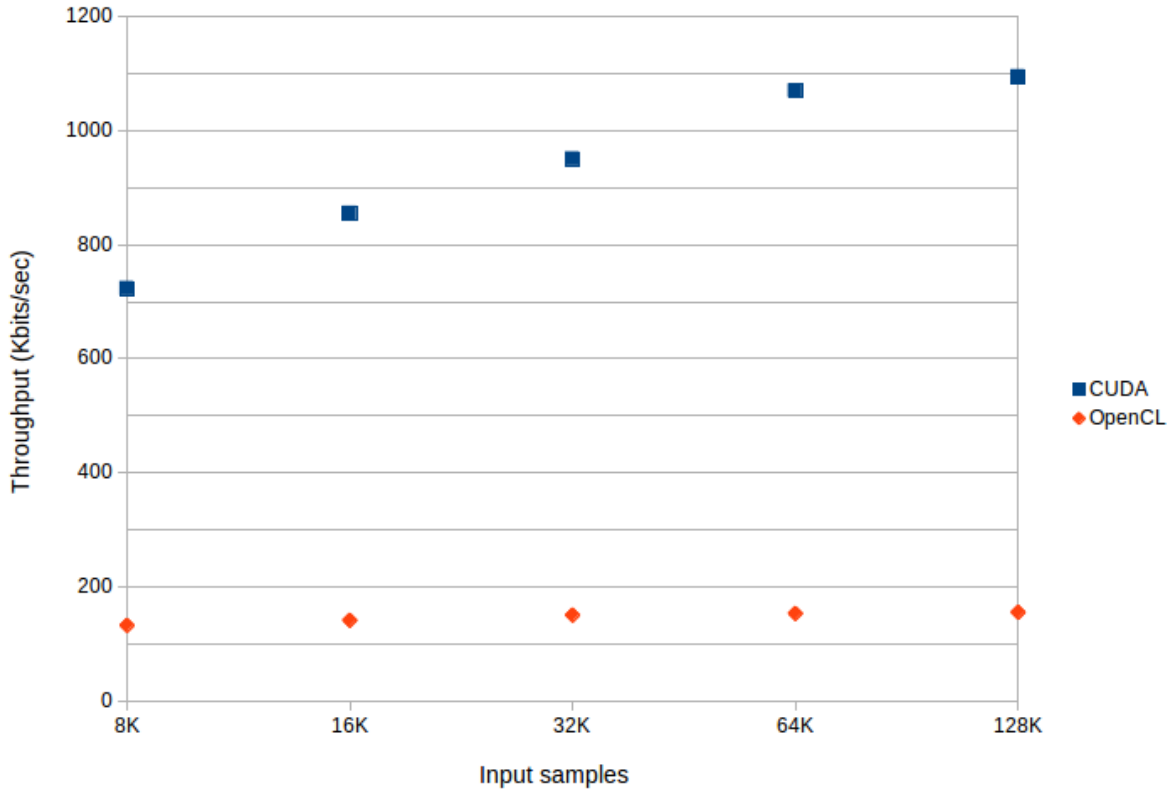

Figure 6.4: FIR filter throughputs for CUDA and OpenCL.

CHAPTER 7

CONCLUSION AND FUTURE WORK

High-level design tool-flows are one of the main solutions used in both industry and academia to reduce the current design productivity challenges affecting modern high-performance platforms. Multicore GPPs, FPGAs and GPUs are some of the main modern hardware platforms that possess high computational density. There is a large number of these tool-flows in the literature, and many of them are based on different techniques. In order to help application designers to make good decisions with regard to these tool-flows, a comparative study of the existing solutions is necessary. SDR is one of the emerging application domains that are characterised by high processing demands, and thus it is one of the main users of modern high-performance hardware platforms. In fact, FPGAs and GPUs are two main platforms that are commonly used to implement high-speed SDR waveforms.

In this dissertation, a comparative study of high-level tool-flows for the rapid prototyping of SDR DSP operations on FPGAs and GPUs has been presented. Migen and MyHDL were studied under FPGA-based tool-flows and OpenCL and CUDA were studied under GPU-based tool-flows. A systematised specification of an ideal high-level SDR DSP tool-flow has been developed and used as the criteria to evaluate the features of the tools qualitatively. In addition, a FIR filter case study was implemented in each of the tools and used to study and compare the tools quantitatively, in terms of design productivity and design performance.

7.1 IDEAL HIGH-LEVEL TOOL-FLOW SPECIFICATION

One of the primary objectives of this dissertation was to conceptualise a systematised ideal tool-flow for developing SDR applications on heterogeneous GPP+FPGA and GPP+GPU accelerator platforms.

7.1.1 FPGA Tool-Flow

In Chapter 4, a systematised ideal tool-flow for developing SDR DSP applications on GPP+FPGA based platforms was conceptualised. Figure 7.1 shows the logical architecture of the proposed tool-flow system. It consists of a set of requirements grouped into four main categories: high-level modeling, code generation, design implementation and design verification. The complete system engineering process followed to specify the proposed tool-flow is documented in Appendix A.

7.1.2 GPU Tool-Flow

In Chapter 4, a systematised ideal tool-flow for developing SDR DSP applications on GPP+GPU based platforms was conceptualized. Figure 7.2 shows the logical architecture of the proposed tool-flow system. It is characterised by a set of requirements, grouped into four main categories: high-level modeling, code generation, design implementation and design verification. Appendix A documents the complete system engineering process followed to conceptualise the tool-flow.

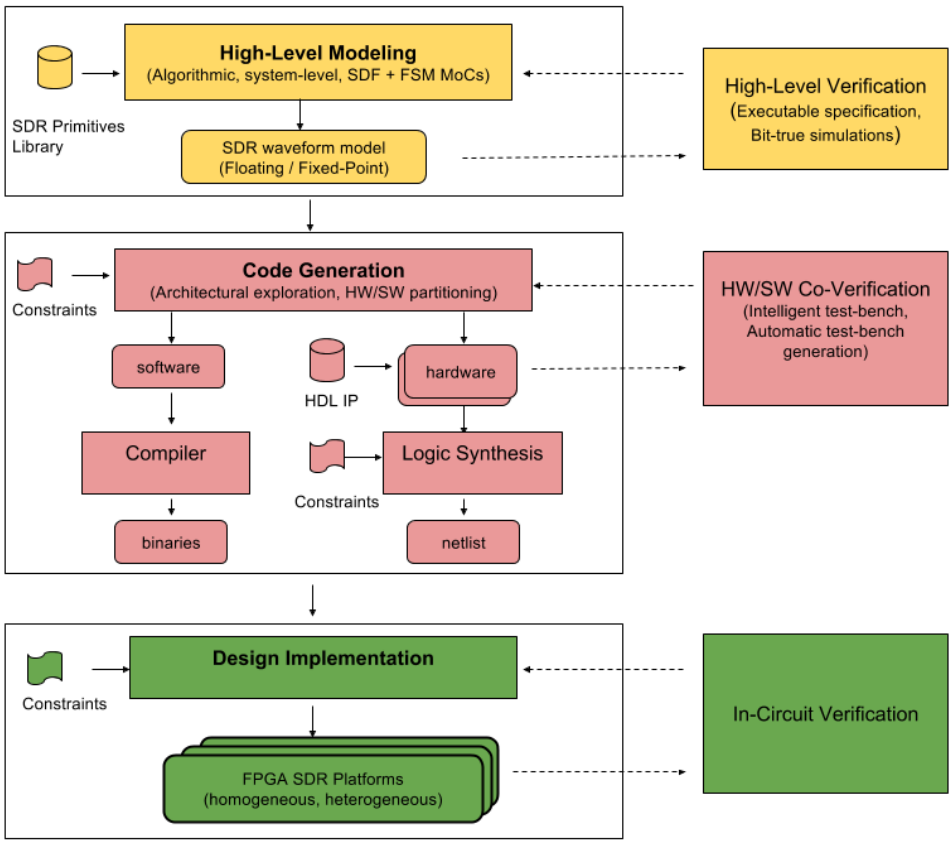

Figure 7.1: The logical architecture of the proposed ideal high-level tool-flow for developing SDR DSP applications on FPGA platforms.

7.2 Evaluations of Existing SDR DSP Tool-Flows

In addition to developing systematised specifications for high-level tool-flows for mapping SDR waveforms on FPGA and GPU platforms, another main objective of this dissertation was to survey the selected existing FPGA and GPU tools.

7.2.1 FPGA SDR DSP Tools

A wide variety of high-level tools targeted at FPGA platforms was presented in Chapter 2 of this dissertation. Two of these, viz. Migen and MyHDL, were selected and surveyed in Chapter 5 to evaluate their capabilities to streamline the process of developing SDR waveforms on FPGA platforms. Detailed results of the survey can be seen in sections 5.2 and 5.3.

Both Migen and MyHDL support 8 out of the 22 required ideal SDR DSP tool-flow features for FPGA targets. A close analysis of the results revealed that Migen out-performs MyHDL with regard to high-level modeling features. Firstly, Migen's FHDL input language is relatively high-level and also easier to learn and to use than MyHDL's design entry syntax, which closely resembles the standard VHDL and Verilog HDLs. Further, although it was still in the developmental stage at the time of writing this dissertation, Migen provides support for the dataflow model of computation, which is an integral feature of ideal SDR DSP design tools. Conversely, MyHDL was found to be slightly better than Migen with regard to design verification. Although they both support simulation of the high-level SDR waveform models and waveform viewing through Python plots and the VCD standard, MyHDL further supports HDL co-simulation and automatic generation of the HDL testbench.

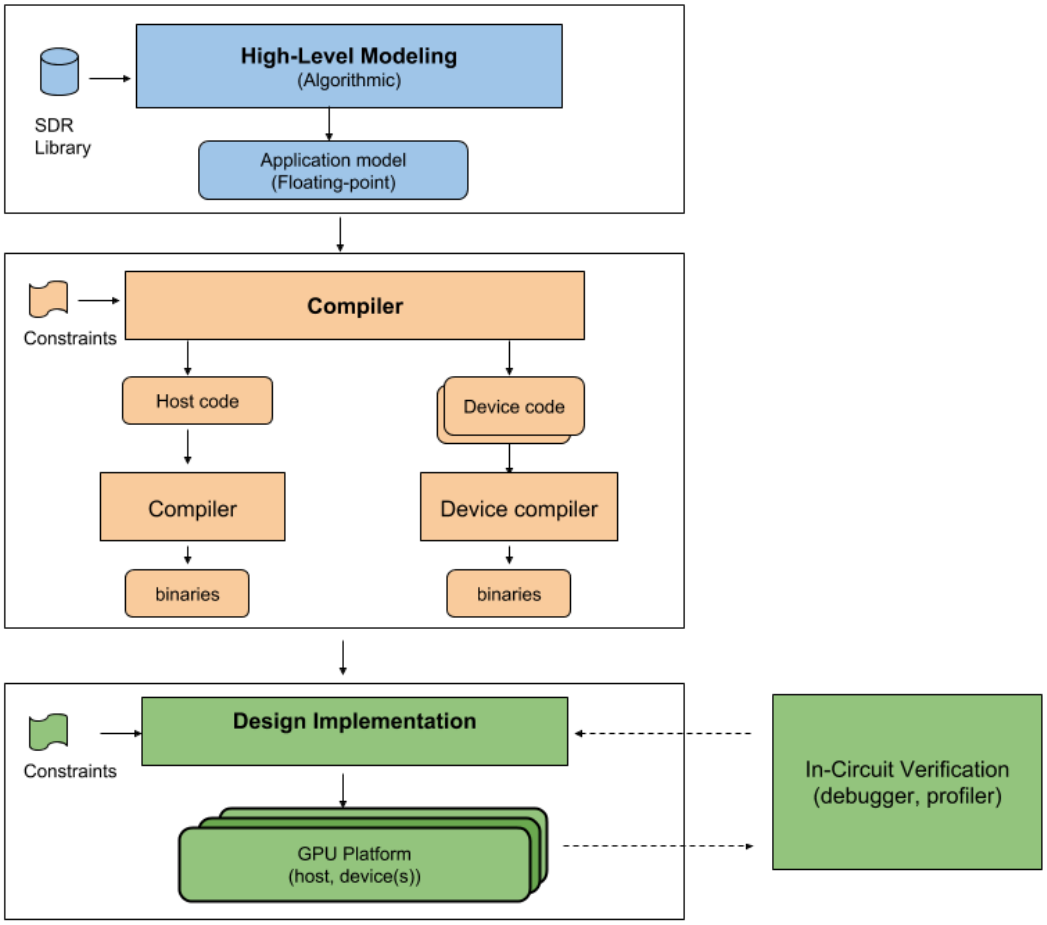

Figure 7.2: The logical architecture of the proposed ideal high-level tool-flow for developing SDR DSP applications on GPU platforms.

The performance of the FIR filter waveform firmware produced by MyHDL was found to be slightly better than that obtained from Migen in terms of logic resources utilisation. Migen, in contrast, slightly out-performed MyHDL in terms of design speed. Design productivity of 1.6x was computed for Migen and 1.4x for MyHDL. The scheme used for design productivity measurement is based on a trade-off between design efficiency and quality. Therefore, the study suggests that Migen is generally slightly better than MyHDL in terms of providing a balance between fast prototyping and producing quality designs.

7.2.2 GPU SDR DSP Tools

CUDA and OpenCL were selected and surveyed in Chapter 6 to assess their capabilities to streamline development of SDR waveforms on GPU platforms. Detailed results of the evaluations can be seen in sections 6.2, 6.3 and 6.4.

The overall performance of the FIR filter waveform developed using CUDA was better than the OpenCL implementation under an equal number of processing cores. In terms of the total running time, including both host and GPU work, CUDA was about 7 times faster than OpenCL for the maximum number of input samples. CUDA kernels were executed 10 times faster than OpenCL for the maximum number of input samples. Data transfer to and from the GPU was found to be 1.5 times faster in CUDA than in OpenCL. However, in terms of power consumption, no difference was observed between CUDA and OpenCL: an average active power of 72W was measured for both.

7.3 CONCLUSION

The overarching objective of this dissertation was to evaluate and compare the capabilities of a selection of high-level design tools to streamline development of SDR DSP waveforms on FPGA and GPU platforms. Our approach involved, firstly, defining the evaluation criteria by specifying the ideal SDR DSP tool-flow systems for FPGA targets as well as GPU targets, and then using the specifications along with the FIR filter case study to perform the evaluations.

For GPU targets, the study has revealed that in terms of fast prototyping SDR DSP algorithms for GPU platforms, CUDA out-performs OpenCL on NVIDIA test platform. CUDA is more high-level than OpenCL and provides more abstractions for the underlying low-level hardware. In addition, the performance of the CUDA-based FIR implementation is better than its OpenCL counter-part.

For FPGA targets, our results show that, although still lacking in several essential element such as co-simulation and intelligent test-bench, Migen is generally more viable than MyHDL for high-level development of SDR waveforms.

7.4 FUTURE WORK

7.4.1 Increase number of case study programmers for statistical significance

Due to time and human resource limitations, the author was the sole programmer of the case study applications used in this dissertation. In order to improve the statistical significance of the tool-flow studies, however, it would be better that the design productivity and performance evaluations of the tool-flows be based on multiple case study implementations by multiple programmers.

7.4.2 Use more and larger case study application

Due to time limitations, the study focused on using only the FIR filter as a representative SDR DSP application for the case studies. Although widely used in similar studies, the FIR filter alone is not enough. There is a need to use the FIR filter alongside more and larger operations, such as the Viterbi decoder.

7.4.3 VHDL code generation capability for Migen

Migen is more suited to high-level FPGA design than MyHDL. Currently, however, it only provides support for Verilog code generation from high-level FHDL descriptions. In order to make it available to a wider community of hardware designers, it is necessary or VHDL code generation capability to be implemented.

7.4.4 SDR DSP primitives for Migen

Migen currently provides a limited set of SDR DSP primitives. In order to facilitate faster prototyping of FPGA-based SDR waveforms using Migen, new DSP primitives must be added to the library.

BIBLIOGRAPHY

[1] "Gnu radio." https://www.gnuradio.org/. [Accessed: 18-08-2018].

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

[3] E. Grayver, *Implementing software defined radio*. Springer Science & Business Media, 2012.

[4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, *et al.*, "The landscape of parallel computing research: A view from berkeley," tech. rep., Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[5] D. F. Bacon, R. Rabbah, and S. Shukla, "Fpga programming for the masses," *Communications of the ACM*, vol. 56, no. 4, pp. 56–63, 2013.

[6] V. Bhatnagar, G. S. Ouedraogo, M. Gautier, A. Carer, and O. Sentieys, "An fpga software defined radio platform with a high-level synthesis design flow," in *Vehicular Technology Conference (VTC Spring), 2013 IEEE 77th*, pp. 1–5, IEEE, 2013.

[7] B. Drozdenko, M. Zimmermann, T. Dao, M. Leeser, and K. Chowdhury, "High-level hardware-software co-design of an 802.11 a transceiver system using zynq soc," in *Computer Communications Workshops (INFOCOM WKSHPS), 2016 IEEE Conference on*, pp. 682–683, IEEE, 2016.

[8] C. Ribeiro and A. Gameiro, "A software-defined radio fpga implementation of ofdm-based phy transceiver for 5g," *Analog Integrated Circuits and Signal Processing*, vol. 91, no. 2, pp. 343–351, 2017.

[9] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," *Design Automation for Embedded Systems*, vol. 16, no. 3, pp. 31–51, 2012.

[10] A. Agarwal, M. C. Ng, and Arvind, "A comparative evaluation of high-level hardware synthesis using reed-solomon decoder," *Embedded Systems Letters, IEEE*, vol. 2, pp. 72–76, Sept 2010.

[11] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, *et al.*, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016.

[12] S. Memeti, L. Li, S. Pllana, J. Kołodziej, and C. Kessler, "Benchmarking opencl, openacc, openmp, and cuda: programming productivity, performance, and energy consumption," in *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, pp. 1–6, ACM, 2017.

[13] I. Mitola, J., "Software radios-survey, critical evaluation and future directions," in *Telesystems Conference, 1992. NTC-92., National*, pp. 13/15–13/23, 1992.

[14] W. Esterhuyse, "Concept description: Meerkat receptor for consideration by ska," tech. rep., Square Kilometre Array, June 2011.

[15] C. Maxfield, *The Design Warrior's Guide to FPGAs*. Orlando, FL, USA: Academic Press, Inc., 2004.

[16] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[17] Xilinx, "Fpga design flow overview." https://www.xilinx.com, 2008. [Accessed: 2018-10-31].

[18] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski, and J. Teich, "Electronic system-level synthesis methodologies," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1517–1530, 2009.

[19] P. R. Panda, "Systemc: A modeling platform supporting multiple design abstractions," in *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, (New York, NY, USA), pp. 75–80, ACM, 2001.

[20] "Myhdl." `http://www.myhdl.org/doku.php`. [Accessed: 2014-02-12].

[21] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: Constructing hardware in a scala embedded language," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pp. 1212–1221, June 2012.

[22] S. Windh, X. Ma, R. J. Halstead, P. Budhkar, Z. Luna, O. Hussaini, and W. A. Najjar, "High-level language tools for reconfigurable computing," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 390–408, 2015.

[23] D. Singh and C. K. Reddy, "A survey on platforms for big data analytics," *Journal of Big Data*, vol. 2, p. 8, Oct 2014.

[24] D. Singh and C. K. Reddy, "A survey on platforms for big data analytics," *Journal of Big Data*, vol. 2, no. 1, p. 8, 2015.

[25] R. Smith, "The nvidia geforce gtx titan x review." https://www.anandtech.com/show/9059/the-nvidia-geforce-gtx-titan-x-review, 2015. [Accessed: 7-02-2019].

[26] O. Maitre, "Understanding nvidia gpgpu hardware," in *Massively Parallel Evolutionary Computation on GPGPUs*, pp. 15–34, Springer, 2013.

[27] "Cuda zone." https://developer.nvidia.com/cuda-zone. [Accessed: 18 August 2018].

[28] T. Ni, "Direct compute: Bring gpu computing to the mainstream," in *GPU Technology Conference*, p. 23, 2009.

[29] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[30] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*. Springer, 2007.

[31] X. Wang, T. Ge, J. Li, C. Su, and F. Hong, "Efficient multi-rate encoder of qc-ldpc codes based on fpga for wimax standard," *Chinese Journal of Electronics*, vol. 26, no. 2, pp. 250–255, 2017.

[32] K. Kapinchev, A. Bradu, F. Barnes, and A. Podoleanu, "Gpu implementation of cross-correlation for image generation in real time," in *Signal Processing and Communication Systems (ICSPCS), 2015 9th International Conference on*, pp. 1–6, IEEE, 2015.

[33] R. H. Hosking, *Software Defined Radio Handbook*. Pentek Inc, 2012.

[34] G. Stitt, "Are field-programmable gate arrays ready for the mainstream?," *Micro, IEEE*, vol. 31, pp. 58–63, Nov 2011.

[35] "The international technology roadmap for semiconductors," tech. rep., Edition 2001.

[36] E. El-Araby, M. Taher, M. Abouellail, T. El-Ghazawi, and G. Newby, "Comparative analysis of high level programming for reconfigurable computers: Methodology and empirical study," in *Programmable Logic, 2007. SPL '07. 2007 3rd Southern Conference on*, pp. 99–106, Feb 2007.

[37] D. J. Warne, N. A. Kelson, and R. F. Hayward, "Comparison of high level {FPGA} hardware design for solving tri-diagonal linear systems," *Procedia Computer Science*, vol. 29, no. 0, pp. 95 – 101, 2014. 2014 International Conference on Computational Science.

[38] S. Qin and M. Berekovic, "A comparison of high-level design tools for soc-fpga on disparity map calculation example," *CoRR*, vol. abs/1509.00036, 2015.

[39] G. Bosman, "A survey of co-design ideas and methodologies," Master's thesis, Chess-iT, August 2003.

[40] M. . i. e. j. Hofstra, "Comparing hardware description languages," 2012.

[41] K. Setetemela, S. Winberg, and M. Inggs, "Evaluation of high-level open-source tool-flows for rapid prototyping of software defined radios," in *Radio and Antenna Days of the Indian Ocean (RADIO), 2015*, pp. 1–2, IEEE, 2015.

[42] K. Setetemela and S. Winberg, "Systematic design of an ideal toolflow for accelerating big data applications on fpga platforms," in *Mechanical and Intelligent Manufacturing Technologies (ICMIMT), 2018 IEEE 9th International Conference on*, pp. 202–206, IEEE, 2018.

[43] "Wireless innovation forum top ten most wanted innovations." https://www.wirelessinnovation.org/top-ten-innovation-1. [Accessed: 10-08-2018].

[44] S. K. Jayaweera, *Introduction*, pp. 1–26. John Wiley & Sons, Inc, 2014.

[45] P. G. Cook and W. Bonser, "Architectural overview of the speakeasy system," *IEEE Journal on Selected Areas in Communications*, vol. 17, pp. 650–661, Apr 1999.

[46] F.-L. Luo, ed., *Digital Front-End in Wireless Communications and Broadcasting: Circuits and Signal Processing*. Cambridge University Press, 2011.

[47] G. Sklivanitis, A. Gannon, S. N. Batalama, and D. A. Pados, "Addressing next-generation wireless challenges with commercial software-defined radio platforms," *IEEE Communications Magazine*, vol. 54, no. 1, pp. 59–67, 2016.

[48] K. Motoi, N. Oshima, M. Kitsunezuka, and K. Kunihiro, "A band-switchable and tunable nested bandpass filter with continuous 0.4–3ghz coverage," in *Microwave Integrated Circuits Conference (EuMIC), 2016 11th European*, pp. 492–495, IEEE, 2016.

[49] R. C. Johnson and H. Jasik, "Antenna engineering handbook," *New York, McGraw-Hill Book Company, 1984, 1356 p. No individual items are abstracted in this volume.*, 1984.

[50] P. Liu, S. Yang, X. Wang, M. Yang, J. Song, and L. Dong, "Directivity-reconfigurable wideband two-arm spiral antenna," *IEEE Antennas and Wireless Propagation Letters*, vol. 16, pp. 66–69, 2017.

[51] A. Haghighat, "A review on essentials and technical challenges of software defined radio," in *MILCOM 2002. Proceedings*, vol. 1, pp. 377–382, IEEE, 2002.

[52] E. Grayver, "State-of-the-art sdr components," in *Implementing Software Defined Radio*, pp. 159–181, Springer, 2013.

[53] H. Rajagopalan, J. M. Kovitz, and Y. Rahmat-Samii, "Mems reconfigurable optimized e-shaped patch antenna design for cognitive radio," *IEEE Transactions on Antennas and Propagation*, vol. 62, pp. 1056–1064, March 2014.

[54] B. Le, T. W. Rondeau, J. H. Reed, and C. W. Bostian, "Analog-to-digital converters," *IEEE Signal Processing Magazine*, vol. 22, no. 6, pp. 69–77, 2005.

[55] R. H. Walden, "Analog-to-digital conversion in the early twenty-first century," *Wiley Encyclopedia of Computer Science and Engineering*, pp. 1–14, 2007.

[56] B. Murmann, "Adc performance survey 1997-2018." [Online] Available: http://web.stanford.edu/ murmann/adcsurvey.html.

[57] S. Creaney and I. Kostarnov, "Designing efficient digital up and down converters for narrowband systems," *XILINX, XAPP1113 (v1. 0)*, 2008.

[58] O. Anjum, T. Ahonen, F. Garzia, J. Nurmi, C. Brunelli, and H. Berg, "State of the art baseband dsp platforms for software defined radio: A survey," *EURASIP Journal on wireless communications and networking*, vol. 2011, no. 1, p. 5, 2011.

[59] J. Chacko, C. Sahin, D. Nguyen, D. Pfeil, N. Kandasamy, and K. Dandekar, "Fpga-based latency-insensitive ofdm pipeline for wireless research," in *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pp. 1–6, IEEE, 2014.

[60] N. Ali and B. Garg, "New energy efficient reconfigurable fir filter architecture and its vlsi implementation," in *International Symposium on VLSI Design and Test*, pp. 519–532, Springer, 2017.

[61] G. Wang, G. Stitt, H. Lam, and A. D. George, "A framework for core-level modeling and design of reconfigurable computing algorithms," in *Proceedings of the Third International Workshop on High-Performance Reconfigurable Computing Technology and Applications*, pp. 29–38, ACM, 2009.

[62] K. A. Hoque, O. A. Mohamed, Y. Savaria, and C. Thibeault, "Early analysis of soft error effects for aerospace applications using probabilistic model checking," in *International Workshop on Formal Techniques for Safety-Critical Systems*, pp. 54–70, Springer, 2013.

[63] S. Winberg, L. Mohapi, and M. Inggs, "Optisdr—a domain specific language to improve developer productivity for software defined radio," in *Emerging Technologies and Innovative Business Practices for the Transformation of Societies (EmergiTech), IEEE International Conference on*, pp. 20–26, IEEE, 2016.

[64] K. A. Hoque, O. A. Mohamed, and Y. Savaria, "Formal analysis of seu mitigation for early dependability and performability analysis of fpga-based space applications," *Journal of Applied Logic*, vol. 25, pp. 47–68, 2017.

[65] R. Woods, J. McAllister, Y. Yi, and G. Lightbody, *DSP Basics*, pp. 11–40. John Wiley & Sons, Ltd, 2017.

[66] R. Lyons, *Understanding Digital Signal Processing*. Pearson Education, 2010.

[67] R. Chand, P. Tripathi, A. Mathur, and K. Ray, "Fpga implementation of fast fir low pass filter for emg removal from ecg signal," in *Power, Control and Embedded Systems (ICPCES), 2010 International Conference on*, pp. 1–5, IEEE, 2010.

[68] I. Kuon, R. Tessier, J. Rose, *et al.*, "Fpga architecture: Survey and challenges," *Foundations and Trends® in Electronic Design Automation*, vol. 2, no. 2, pp. 135–253, 2008.

[69] U. Farooq, Z. Marrakchi, and H. Mehrez, "Fpga architectures: An overview," in *Tree-based heterogeneous FPGA architectures*, pp. 7–48, Springer, 2012.

[70] S. Scott, *Rhino: Reconfigurable hardware interface for computation and radio*. PhD thesis, University of Cape Town, 2011.

[71] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra, "Graphics processing unit (gpu) programming strategies and trends in gpu computing," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 4–13, 2013.

[72] K. Li, M. Wu, G. Wang, and J. R. Cavallaro, "A high performance gpu-based software-defined basestation," in *2014 48th Asilomar Conference on Signals, Systems and Computers*, pp. 2060–2064, Nov 2014.

[73] "Flex 10k." https://www.altera.com/en_US/pdfs/literature/ds/archives/dsf10k.pdf. [Accessed: 18 August 2018].

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

[75] rhino, "Rhino wiki." https://www.ohwr.org/projects/rhino-hardware-01/wiki. [Accessed: 19-07-2018].

[76] R. A. Primiani, K. H. Young, A. Young, N. Patel, R. W. Wilson, L. Vertatschitsch, B. B. Chitwood, R. Srinivasan, D. MacMahon, and J. Weintroub, "Swarm: A 32 ghz correlator and vlbi beamformer for the submillimeter array," *Journal of Astronomical Instrumentation*, vol. 5, no. 04, p. 1641006, 2016.

[77] W. New, "Python based fpga design-flow," Master's thesis, University of Cape Town, 2016.

[78] CASPER, "Roach." https://casper.berkeley.edu/wiki/ROACH. [Accessed: 22-07-2018].

[79] Ettus Research, *USRP N200/N210 Networked Series*, September 2012.

[80] "Usrp hardware driver (uhd) software." https://github.com/EttusResearch/uhd. [Accessed: 24 August 2018].

[81] "Migen." `http://milkymist.org/3/migen.html`. Accessed: 2014-02-12.

[82] J. Decaluwe, "Myhdl: a python-based hardware description language.," *Linux journal*, no. 127, pp. 84–87, 2004.

[83] S. Bourdeauducq, *Migen Manual*. M-Labs, 0.5 ed., October 2017.

[84] K. Li, B. Yin, M. Wu, J. R. Cavallaro, and C. Studer, "Accelerating massive mimo uplink detection on gpu for sdr systems," in *2015 IEEE Dallas Circuits and Systems Conference (DCAS)*, pp. 1–4, Oct 2015.

[85] V. Jalili-Marandi, Z. Zhou, and V. Dinavahi, "Large-scale transient stability simulation of electrical power systems on parallel gpus," in *2012 IEEE Power and Energy Society General Meeting*, pp. 1–11, July 2012.

[86] C. S. Lin, W. L. Liu, W. T. Yeh, L. W. Chang, W. M. W. Hwu, S. J. Chen, and P. A. Hsiung, "A tiling-scheme viterbi decoder in software defined radio for gpus," in *2011 7th International Conference on Wireless Communications, Networking and Mobile Computing*, pp. 1–4, Sept 2011.

[87] I. Alyafawi, A. Durand, and T. Braun, "High-performance wideband sdr channelizers," in *Wired/Wireless Internet Communications* (L. Mamatas, I. Matta, P. Papadimitriou, and Y. Koucheryavy, eds.), (Cham), pp. 3–14, Springer International Publishing, 2016.

[88] NVIDIA, *CUDA C PROGRAMMING GUIDE*, March 2018.

[89] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2011.

[90] X. Cai, M. Zhou, and X. Huang, "Model-based design for software defined radio on an fpga," *IEEE Access*, vol. 5, pp. 8276–8283, 2017.

[91] M. C. McFarland, A. C. Parker, and R. Camposano, "The high-level synthesis of digital systems," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 301–318, 1990.

[92] R. A. Rutenbar, "Dac at 50: The second 25 years," *IEEE Design & Test*, vol. 31, no. 2, pp. 32–39, 2014.

[93] J. M. Cardoso and M. Weinhardt, "High-level synthesis," in *FPGAs for Software Programmers*, pp. 23–47, Springer, 2016.

[94] P. Coussy, M. Meredith, A. Takach, and D. D. Gajski, "An introduction to high-level synthesis," *IEEE Design & Test of Computers*, vol. 26, pp. 8–17, 08 2009.

[95] "What is gnu radio?." https://www.nutaq.com/blog/what-gnu-radio. [Accessed: 18 August 2018].

[96] K. Karimi, N. G. Dickson, and F. Hamze, "A performance comparison of cuda and opencl," *arXiv preprint arXiv:1005.2581*, 2010.

[97] M. Pelcat, C. Bourrasset, L. Maggiani, and F. Berry, "Design productivity of a high level synthesis compiler versus hdl," in *Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), 2016 International Conference on*, pp. 140–147, IEEE, 2016.

[98] V. R. Basili and K. Freburger, "Programming measurement and estimation in the software engineering laboratory.," *Journal of Systems and Software*, vol. 2, no. 1, pp. 47–57, 1981.

[99] K. Beeck, F. Heylen, J. Meel, and T. Goedemé, "Comparative study of model-based hardware design tools," in *Proc. of the European Conference on the Use of Modern Information and Communications Technologies (ECUMICT), Ghent, Belgium*, pp. 295–306, 2010.

[100] V. K. Pallipuram, M. Bhuiyan, and M. C. Smith, "A comparative study of gpu programming models and architectures using neural networks," *The Journal of Supercomputing*, vol. 61, no. 3, pp. 673–718, 2012.

[101] S. Ravi and M. Joseph, "Open source hls tools: a stepping stone for modern electronic cad," in *Computational Intelligence and Computing Research (ICCIC), 2016 IEEE International Conference on*, pp. 1–8, IEEE, 2016.

[102] J. Bard and V. J. Kovarik Jr, *Software defined radio: the software communications architecture*, vol. 6. John Wiley & Sons, 2007.

[103] C. Belisle, V. Kovarik, L. Pucker, and M. Turner, "The software communications architecture: two decades of software radio technology innovation," *IEEE Communications Magazine*, vol. 53, pp. 31–37, September 2015.

[104] R. C. Reinhart, S. K. Johnson, T. J. Kacpura, C. S. Hall, C. R. Smith, and J. Liebetreu, "Open architecture standard for nasa's software-defined space telecommunications radio systems," *Proceedings of the IEEE*, vol. 95, no. 10, pp. 1986–1993, 2007.

[105] "Strs - general overview." https://spaceflightsystems.grc.nasa.gov/sopo/scsmo/advanced-communications-systems/technology-studies/strs/. [Accessed: 24 August 2018].

[106] SEBok, "Systems engineering overview." https://www.sebokwiki.org/wiki/Systems_Engineering_Overview. [Accessed: 18 August 2018].

[107] U. DoD, "Systems engineering fundamentals," *DoD, Department of Defense (USA)*, 2001.

[108] C. S. Wasson, *System engineering analysis, design, and development: Concepts, principles, and practices*. John Wiley & Sons, 2015.

[109] J. Hickish, Z. Abdurashidova, Z. Ali, K. D. Buch, S. C. Chaudhari, H. Chen, M. Dexter, R. S. Domagalski, J. Ford, G. Foster, *et al.*, "A decade of developing radio-astronomy instrumentation using casper open-source technology," *Journal of Astronomical Instrumentation*, vol. 5, no. 04, p. 1641001, 2016.

[110] I. Dogaru and R. Dogaru, "Designing low complexity computational intelligence modules in fpga using high level synthesis tools," in *Electrical and Electronics Engineering (ISEEE), 2017 5th International Symposium on*, pp. 1–5, IEEE, 2017.

[111] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 33–36, ACM, 2011.

[112] E. Rebello, L. Vanfretti, and M. S. Almas, "Pmu-based real-time damping control system software and hardware architecture synthesis and evaluation," in *Power & Energy Society General Meeting, 2015 IEEE*, pp. 1–5, IEEE, 2015.

[113] H. M. Borgund, "Spam filtering with approximate search in fpga hardware," Master's thesis, NTNU, 2018.

[114] J. Kasser *et al.*, *The cataract methodology for systems and software acquisition*. PhD thesis, Citeseer, 2002.

[115] P. Isza, "Tfilter." http://t-filter.engineerjs.com/. [Accessed on: 17-08-2018].

[116] S. Mesnier, "Measuring the effectiveness of fpga programming languages," 2008.

[117] Xilinx, "Xilinx power estimator (xpe)." Accessed: 5-02-19.

[118] D. R. N. Yoge and N. Chandrachoodan, "Gpu implementation of a programmable turbo decoder for software defined radio applications," in *VLSI Design (VLSID), 2012 25th International Conference on*, pp. 149–154, IEEE, 2012.

[119] G. Wang, M. Wu, B. Yin, and J. R. Cavallaro, "High throughput low latency ldpc decoding on gpu for sdr systems," in *Global Conference on Signal and Information Processing (GlobalSIP), 2013 IEEE*, pp. 1258–1261, IEEE, 2013.

[120] T. Nyländen, J. Janhunen, O. Silvén, and M. Juntti, "A gpu implementation for two mimo-ofdm detectors," in *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pp. 293–300, IEEE, 2010.

[121] M. Wu, Y. Sun, S. Gupta, and J. R. Cavallaro, "Implementation of a high throughput soft mimo detector on gpu," *Journal of Signal Processing Systems*, vol. 64, no. 1, pp. 123–136, 2011.

[122] C. Ahn, S. Bang, H. Kim, S. Lee, J. Kim, S. Choi, and J. Glossner, "Implementation of an sdr system using an mpi-based gpu cluster for wimax and lte," *Analog Integrated Circuits and Signal Processing*, vol. 73, no. 2, pp. 569–582, 2012.

[123] S. C. Kim and S. S. Bhattacharyya, "Implementation of a high-throughput low-latency polyphase channelizer on gpus," *EURASIP Journal on advances in signal processing*, vol. 2014, no. 1, p. 141, 2014.

[124] M. Harris, "How to optimize data transfers in cuda c/c++." https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/, December 2012. [Accessed: 24 August 2018].

[125] "nvprof." https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview. [Accessed: 30-08-2018].

[126] M. Burtscher, I. Zecena, and Z. Zong, "Measuring gpu power with the k20 built-in sensor," in *Proceedings of Workshop on General Purpose Processing Using GPUs*, p. 28, ACM, 2014.

[127] "Square kilometre array project - meerkat radio telescope." http://www.ska.ac.za/gallery/meerkat/. Accessed: 2017-10-10.

[128] "Wireless innovation forum." http://wirelessinnovation.org/. [Accessed: 07-02-2019].

[129] S. Smith, *Digital signal processing: a practical guide for engineers and scientists*. Elsevier, 2013.

[130] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajdax, "Feldspar: A domain specific language for digital signal processing algorithms," in *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pp. 169–178, IEEE, 2010.

[131] T. Hill, "Floating- to fixed-point matlab algorithm conversion for fpgas." https://www.eetimes.com/document.asp?doc_id=1275410, April 2007. [Accessed: 24-08-2018].

[132] H. D. Foster, "Trends in functional verification: a 2014 industry study," in *Proceedings of the 52nd Annual Design Automation Conference*, p. 48, ACM, 2015.

[133] H. D. Foster, "Why the design productivity gap never happened," in *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '13, (Piscataway, NJ, USA), pp. 581–584, IEEE Press, 2013.

[134] S. Catenacci, "Advanced verification for fpgas," tech. rep., Mentor Graphics, 2018.

[135] B. So, M. W. Hall, and P. C. Diniz, "A compiler approach to fast hardware design space exploration in fpga-based systems," in *ACM SIGPLAN Notices*, vol. 37, pp. 165–176, ACM, 2002.

[136] "Project icestorm." http://www.clifford.at/icestorm/. [Accessed: 19-11-2017].

[137] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity-the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[138] I. Roadmap, "International technology roadmap for semiconductors, 2009 edn," *Executive Summary. Semiconductor Industry Association*, 2009.

[139] K. Skey, J. Bradley, and K. Wagner, "A reuse approach for fpga-based sdr waveforms," in *Military Communications Conference, 2006. MILCOM 2006. IEEE*, pp. 1–7, IEEE, 2006.

[140] Y. Jararweh, A. Doulat, A. Darabseh, M. Alsmirat, M. Al-Ayyoub, and E. Benkhelifa, "Sdmec: Software defined system for mobile edge computing," in *Cloud Engineering Workshop (IC2EW), 2016 IEEE International Conference on*, pp. 88–93, IEEE, 2016.

[141] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4s, p. 134, 2014.

[142] T. Ulversoy, "Software defined radio: Challenges and opportunities," *Communications Surveys Tutorials, IEEE*, vol. 12, pp. 531–550, Fourth 2010.

[143] T. Feist, "Vivado design suite," *White Paper*, vol. 5, 2012.

[144] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "Gpu cluster for high performance computing," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, p. 47, IEEE Computer Society, 2004.

[145] L. J. Mohapi, *A domain specific language for facilitating automatic parallelization and placement of SDR patterns into heterogeneous computing architectures*. PhD thesis, University of Cape Town, 2017.

[146] D. S. Gianmarco Baldini, "Software defined radio certification in europe: Challenges and processes," *nternational Journal o n Advances in Telecommunications*, vol. 3, 2010.

[147] S. Bourdeauducq, "Migen." https://m-labs.hk/migen/slides.pdf, 2013. [Accessed: 31-10-2018].

[148] "Amd app profiler." https://developer.amd.com/tools-and-sdks/amdappprofiler/. [Accessed: 18-08-2018].

[149] "App kernel analyzer." https://gpuopen.com/archive/app-kernel-analyzer/. [Accessed: 18-08-2018].

APPENDIX A

SYSTEMATIC DESIGN OF AN IDEAL HIGH-LEVEL SDR TOOL-FLOW

A.1 STAKEHOLDERS IDENTIFICATION

A system may have different stakeholders throughout its life cycle. Therefore, in order to develop a more comprehensive set of requirements, it is necessary to consider all stages of the life cycle model when identifying the stakeholders of a system [106].

Every system has its own stages of life, which typically include stages such as concept, development, production, operations, sustainment, and retirement. For each stage, a list of all stakeholders having an interest in the future system must be identified. The goal is to get every stakeholder's point of view for every stage of the system life in order to consolidate a complete set of stakeholder needs that can be prioritized and transformed into the set of stakeholder requirements as exhaustively as possible [106]. For the Ideal High-Level SDR Tool-Flow system, the considered life cycle stages are: engineering, development, operation and maintenance.

Stakeholder classes for the ideal high-level SDR tool-flow system were identified by considering each stage of the system life cycle and identifying a list of main stakeholders who have an interest in the system. See Table A.1 for a list of identified stakeholders. The list is not exhaustive, but it is large enough to aid in developing a comprehensive requirements basis for the ideal tool-flow.

Table A.1: Identification of Stakeholders for the Ideal High-Level SDR Design Flow.

Life Cycle Stage	Stakeholders
Engineering	EDA tools engineers
	EDA standards bodies
	SDR waveform developers
	FPGA-based SDR platform makers
Development	EDA tools developers
	EDA standards bodies
Operation	EDA SDR waveform developers
	EDA Researchers
Maintenance	EDA tools developers

The stakeholders are further organised into four main classes which are described below.

- **End-Users** - this group consists of end-users of the ideal high-level SDR tool-flow. Members are: DSP gateware engineers (MeerKAT [127] digital-backend (DBE) team); SDR researchers (Software-Defined Radio Group at the University of Cape Town).
- **Research & Development** - this group comprises scientists and engineers who are involved in the design and construction of the ideal high-level SDR tool-flow. Members are: High-level hardware design technology researchers; SDR researchers; Software engineers; Gateware engineers.
- **Hardware Manufacturers** - this group consists of manufacturers of chips and boards that are used to implement SDR applications. Members are: SDR FPGA board makers (UCT RHINO team, ROACH team); manufacturers of FPGA devices (Xilinx, Altera).
- **Standardisation Bodies** - this group comprises relevant standardisation bodies who direct the design and development of compatible, interoperable and good quality design tools and applications. Members are: SDR standardisation bodies (Wireless Innovation Forum [128]); FPGA EDA standardisation bodies.

A.2 ORIGINAL STAKEHOLDER REQUIREMENTS

The purpose of the Stakeholder Needs and Requirements Definition process is to define the stakeholder requirements for a system that can provide the capabilities needed by users and other stakeholders in a defined environment [106]. For this project, requirements interviews and workshops were held with representative members from the various stakeholder groups to elicit requirements for the Ideal High-Level SDR Tool-Flow system. Additional requirements were obtained through a review of the literature. Figures A.10, A.3, A.4 and A.1 show the original tool-flow requirements from the end-users, research and development, hardware manufactures and standardisation bodies respectively.

A.2.1 Standardisation Bodies

SDR has evolved and matured significantly to the point where it is gaining more application in a wider variety of practical systems [8]. It is expected that adoption and deployment of SDR technology in place of the traditional radio architecture will continue to increase [142]. However, lack of standards slows down faster and wider adoption of SDR [3]. For example, in the absence of well defined standards, selecting and adopting a specific SDR hardware platform is risky because it might be difficulty to impossible to port software written for that platform to another SDR platform. Therefore, standardisation in SDR should address the following three main different parts of an SDR waveform life cycle [3]:

1. Describe the waveform. A waveform description should be entirely hardware independent but contain all the necessary information to implement the waveform on appropriate hardware.
2. Implement the waveform. Waveform implementation should be done in a relatively hardware-independent environment to facilitate portability to differ- ent hardware platforms.

3. Control the waveform. Once the waveform is operating, its features may be modified at runtime.

According to the international technology roadmap for semiconductors [35], the current design technology challenges include design productivity, power consumption, manufacturability, reliability,and interference. The ITRS emphasises a need for new design techniques and tools capable of supporting system-level specification of designs, new efficient verification techniques including executable specification, ESL formal verification and intelligent testbench. In addition, the roadmap motivates that design reuse and ability to automatically generate implementation designs from system-level specifications are among key requirements for future design technologies aimed at addressing the current design technology challenges.

The Wireless Innovation Forum (WinnForum) [128] states that one of the top ten most wanted innovations for SDR technology involves a set of techniques and tools for efficient porting of waveform applications between embedded heterogeneous platforms [43]. Software architecture and design paradigms must evolve to integrate the software design model with the physical radio architecture to address platform-specific requirements and differences in the current versus the target radios that impact software porting. Further, according to WinnForum, in order to reduce development time and cost, it is important that the software written be "easily" portable from platform to platform. In order to achieve this objective, software architecture and design paradigms must evolve to encompass multiple programming languages such as C, C++ and HDL. In addition, design paradigms should allow for multiple design approaches such as multi-threaded applications on GPP or concurrent state machine designs for an FPGA. Design paradigms should also integrate the software model with a systems model of the physical radio architecture to address platform-specific requirements and differences in the current versus the target radios that impact software porting.

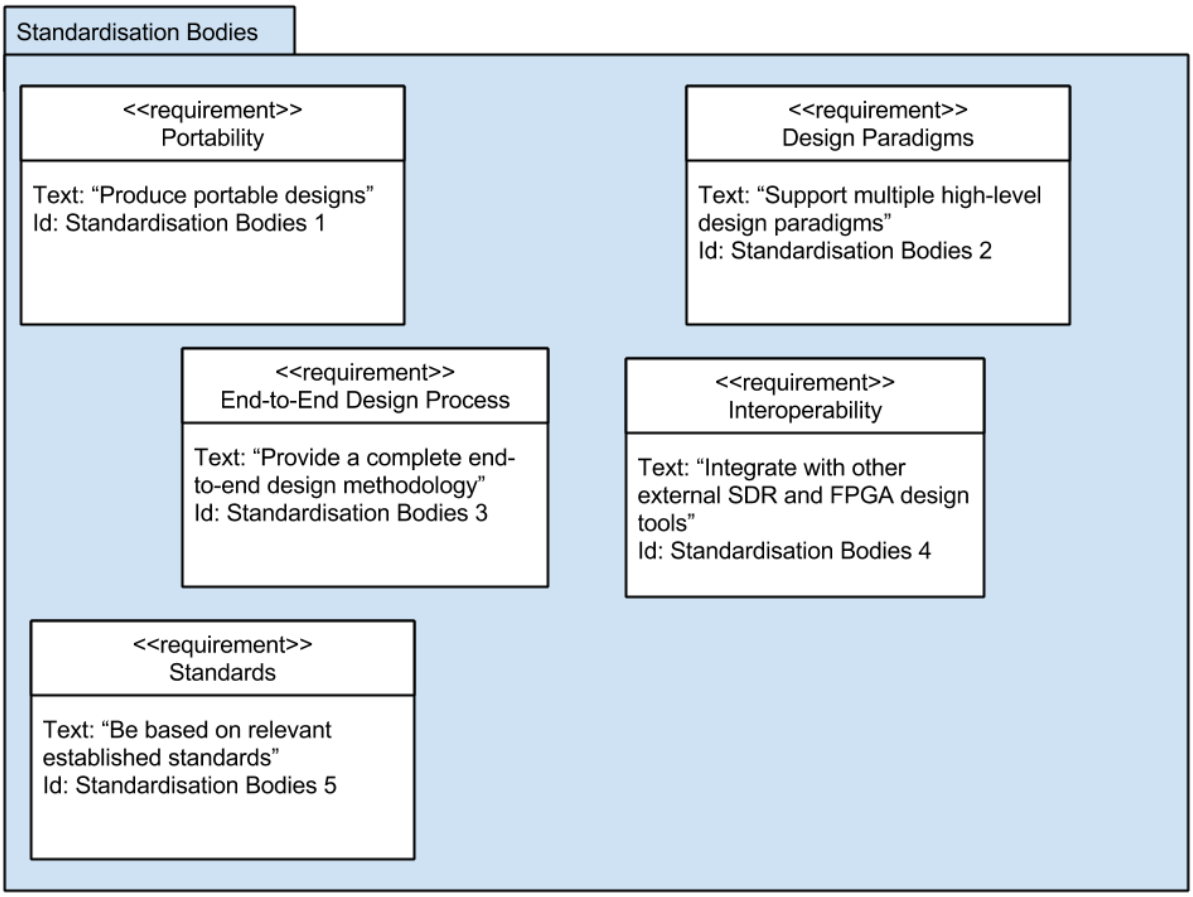

Figure A.1: Original Standardisation Bodies stakeholder group requirements for the Ideal High-level SDR Tool-Flow.

A.2.2 Users

Some of the key current challenges associated with SDR DSP development tools and flows are from the user's perspective. Users range from SDR DSP application designers without solid expertise in low level hardware details yet conversant in software programming languages to those with low-level hardware skills and are comfortable with low-level development languages such as VHDL for logic devices, C and assembly and for processor architectures.

Traditional development tool-flows typically support development at lower levels of abstraction and provide minimal support for portability. They often have a steep learning curve particularly to users lacking in understanding of lower level details of implementation hardware platform. For example, it is very difficult to use VHDL/Verilog to develop a simple low-pass filter and deploy it on an FPGA-based SDR platform without prior knowledge of digital design concepts such as logic gates, clock management and synchronous and asynchronous operations.

Therefore, the ideal high-level SDR DSP tool-flow should provide ability to abstract the designer from the low-level hardware-specific issues and allow the focus to be on fast development of waveforms. In order to achieve this, the tool-flow should be easy to use, support design reuse, and support design entry at higher levels of abstraction yet without resulting in significant losses in the quality of designs.

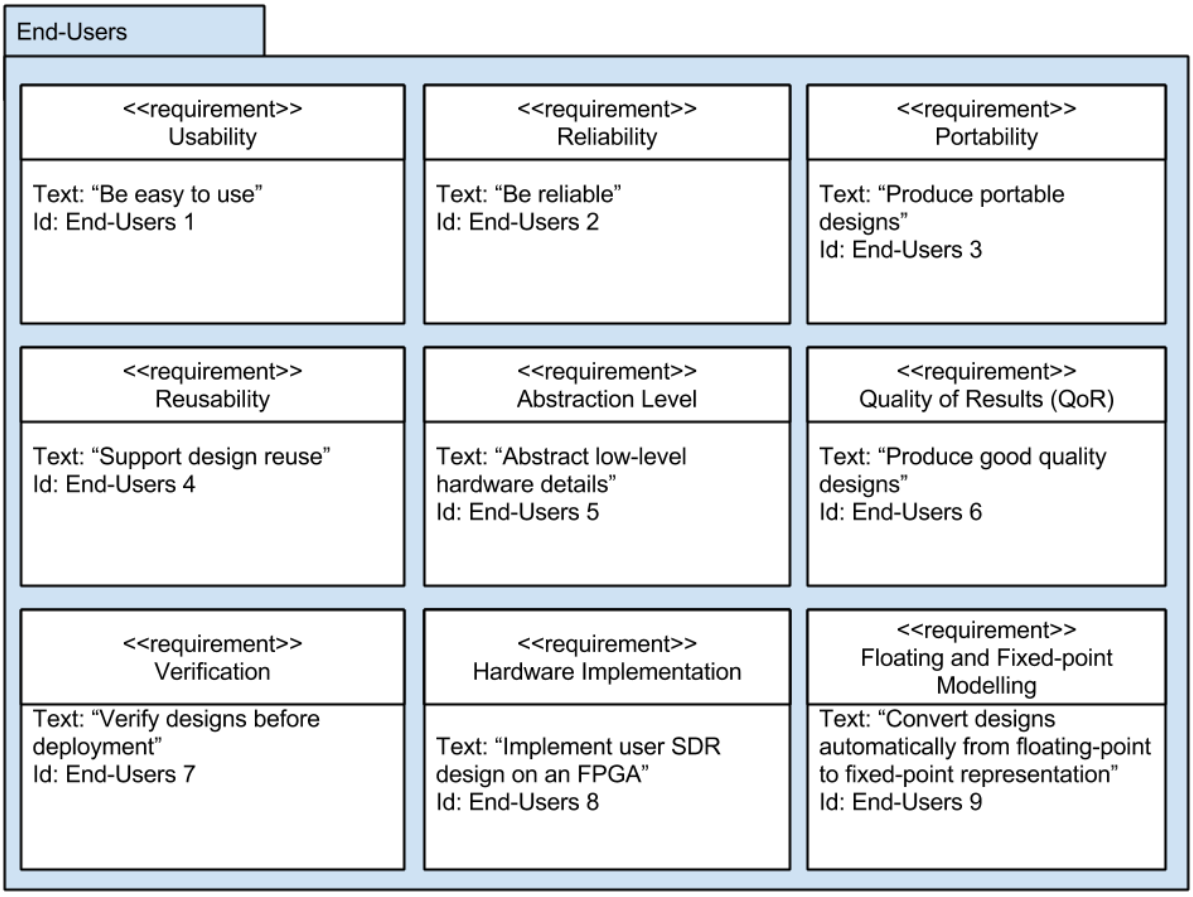

Figure A.2: Original End-Users stakeholder group requirements for the Ideal High-level SDR Tool-Flow.

A.2.3 Research and Development

In a work [77] that concern development of a high-level development framework for SDR DSP, it is proposed that the framework should:

- be able to target different hardware platforms. Particularly, for FPGA platforms both Altera and Xilinx FPGAs should be supported.

- provide a one-click solution to compile a design. The aim is to abstract the designer from the complex underlying intermediary building tools and allow the focus to be directed on development of the SDR DSP application.
- support ability to model, simulate and verify designs using a high-level language. This is a key requirement given that part of the productivity gap challenge is due to the increasingly high verification effort which is difficult to reduce at lower design abstraction levels due to long simulations times.
- provide a set of parameterised libraries. The libraries are required to promote design reuse and thus aid in the rapid implementation of systems.
- provide a streamlined solution from design to implementation. Given that there are typically many steps in the process of transforming a set of SDR DSP application requirements to a working software implementation on a hardware platform, the framework should streamline the process and provide the designer with an integrated complete and easy to use environment.

[3] proposes an ideal tool-flow for SDR development comprising of requirements definition, requirements flowdown, architecture specification, system simulation, firmware development and system verification.

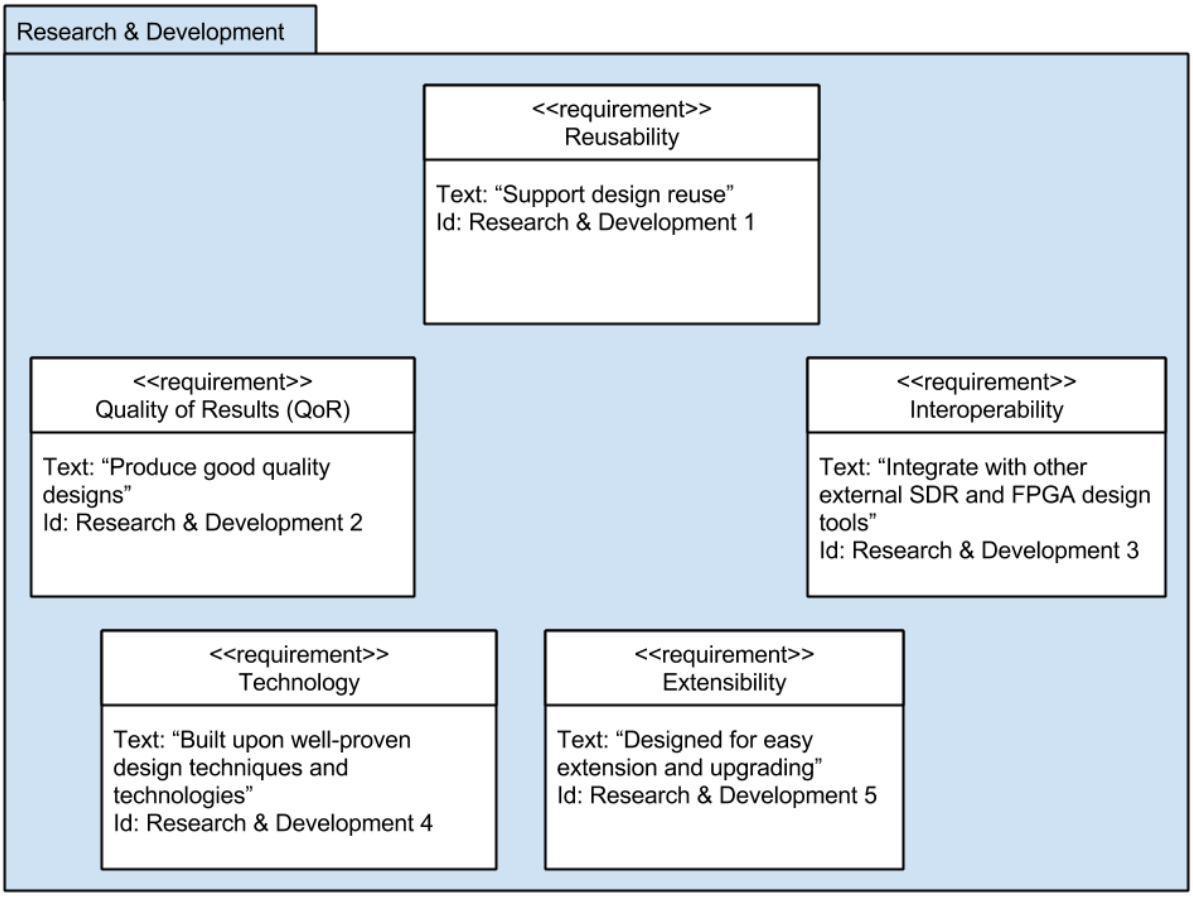

Figure A.3: Original Research & Development Team stakeholder group requirements for the Ideal High-level SDR Tool-Flow.

A.2.4 **Hardware Manufacturers**

SDR DSP hardware manufacturers comprise of all who make processors and platforms targeted at SDR DSP applications. Considering the RHINO SDR platform as an example, the following requirements for the high-level tool-flow system were proposed [74]: high-level formal design specification, ability to implement the design on an actual hardware platform, ability to port the design across multiple platforms.

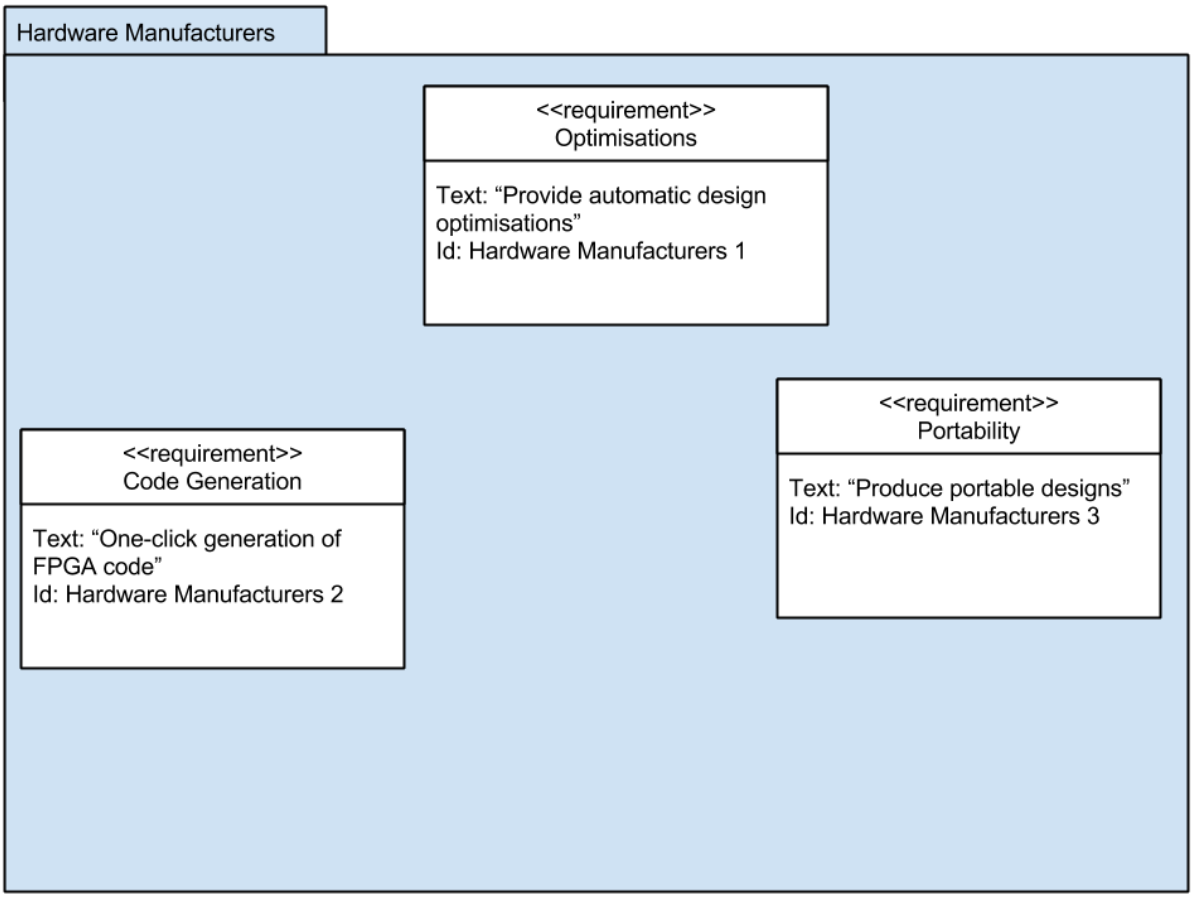

Figure A.4: Original Hardware Manufacturers stakeholder group requirements for the Ideal High-level SDR Tool-Flow.

A.3 Functional Analysis

The purpose of the functional analysis process is to define what a system or it's element does. Functional analysis serves as the basis for the technical and functional and operational requirements of the system. It entails defining the system's functions using an active verb and measurable noun. Several techniques can be used to enhance functional analysis including organising the functions in a work break-down structure (WBS), function analysis system technique (FAST) and a functional architecture diagram. A functional architecture diagram is used in this project. The diagram presents, from a behavioural point of view, all the key tool-flow system functions and interfaces in a hierarchical way. It further shows parent and child relationships between functions.

An analysis of the original stakeholder tool-flow requirements reveals that the requirements can be organised into five main functional categories namely, hardware abstraction, floating to fixed-point conversion, design verification, code generation and design optimisations. Figure A.5 shows this high-level functional decomposition of the tool-flow system.

Hardware abstraction refers to the requirement for the tool-flow system to abstract as much of the underlying low-level hardware details from the SDR designer as possible. For example, the tool-flow should enable the designer to describe an SDR application without having to explicitly handle low-level particulars such as clock management for FPGA designs and synchronisation between GPU cores.

A.4 System Requirements

This section presents the system requirements of the Ideal High-Level FPGA SDR Toolflow system. The requirements describe the complete system-level functions that the Toolflow as a whole should fulfil in order to meet the stakeholder needs and requirements. The system requirements are expressed in technical

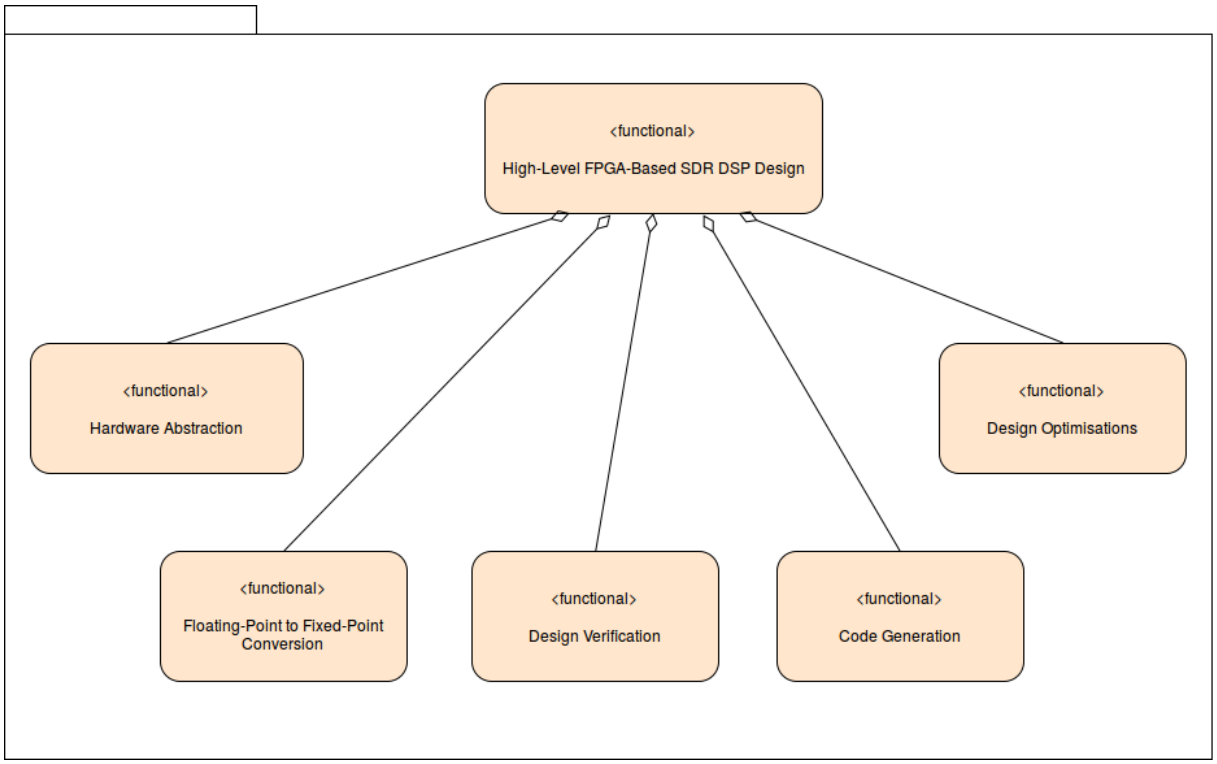

Figure A.5: High-Level FPGA-Based SDR DSP Toolflow Functional Requirements

language that is useful for architectural design and are derived from the originating stakeholder requirements and captures both the functional and non-functional needs of the system.

Table A.2: Functional system requirements for the high-level FPGA-based SDR design flow.

ID	Stakeholder Requirement	System Requirement	Explanation
EU-R5	Abstract low-level hardware details	Specify SDR design functionality at algorithmic level of abstraction	Algorithmic design entry higher than RTL on the design abstraction hierarchy.
		Specify SDR design functionality at system level of abstraction	[128, 3]
EU-R6	Detect SDR design errors early in the development cycle	Simulate high-level input specification	[128]
		High level SDR specification formal verification	[137, 138]
		Automatic test bench generation	To speedup the verification process [138]
		Intelligent test bench	To speed up and streamline the verification task [138]
		Virtual platforms	[139, 140]
EU-R9	Floating and fixed point analysis and design	Floating-point to fixed-point conversion	F-point design implementation on an FPGA is difficult and costly. Fix-point models may suffice for some SDR applications.
		Bit-accurate high-level input specification simulation	[47]
EU-R8	Optimise SDR designs automatically	Domain-specific (SDR) design optimisations	[141]
		Generic design optimisations	[141]
EU-R7	Deploy SDR waveform onto FPGA platform	Generate HDL code	Must generate VHDL/Verilog for mapping with the traditional flow
		Generate FPGA bitstream	[3]
		Program FPGA	[3]
RD-3	Integrate with third-party SDR and FPGA design tools	Integrate with third-party verification tools	[3]
		Integrate with third-party FPGA bitstream generation tools	FPGA bitstream generation largely proprietary except a few [3]
		Integrate with third-party FPGA programming tools	[3]

Table A.3: System requirements for the high-level FPGA-based SDR design flow.

Design Stage	**System Requirements**
High-Level Modeling	Algorithmic-level design entry?
	System-level design entry?
	SDR primitives library?
	SDR MoCs?
	Floating to fixed-point conversion?
Design Verification	High-level model simulation?
	High-level model formal verification?
	HW/SW co-verification?
	Automatic test-bench generation?
	Intelligent test-bench?
	Virtual platforms?
	Third-party verification tools integration?
Code Generation	Software code generation?
	Hardware code generation?
	Logic synthesis?
	FPGA bitstream generation?
	Domain-specific design optimisations?
	Generic design optimisations?
Design Implementation	FPGA bitstream generation?
	Third-party FPGA implementation tools integration?

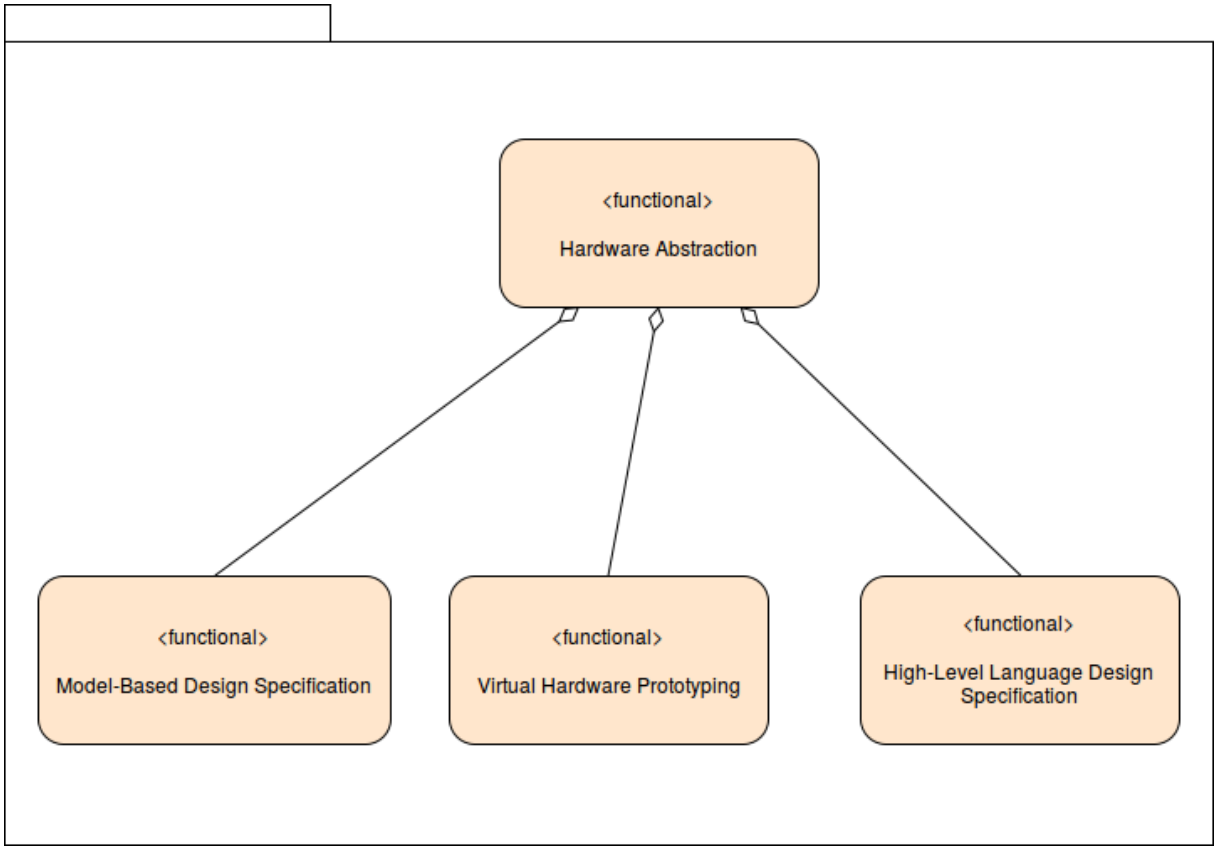

Figure A.6: Hardware abstraction.

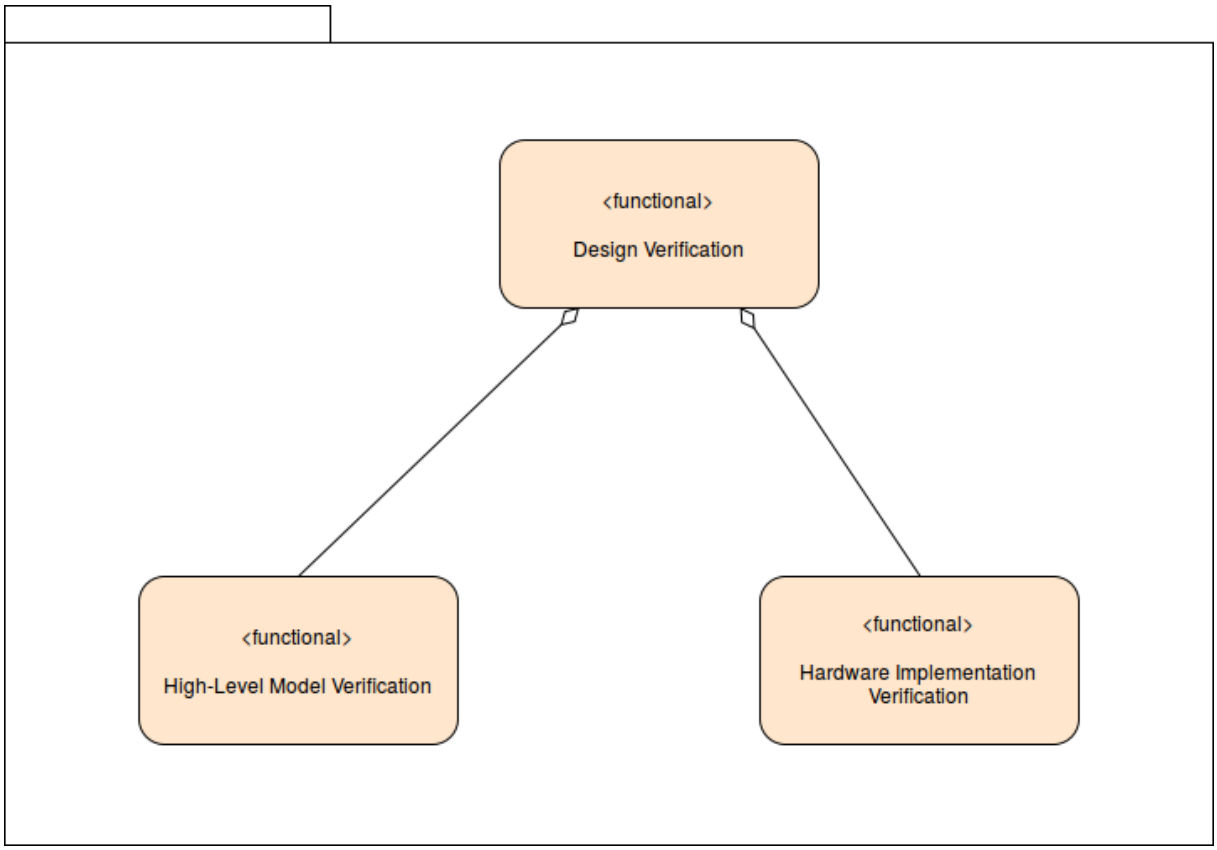

Figure A.7: Design verification.

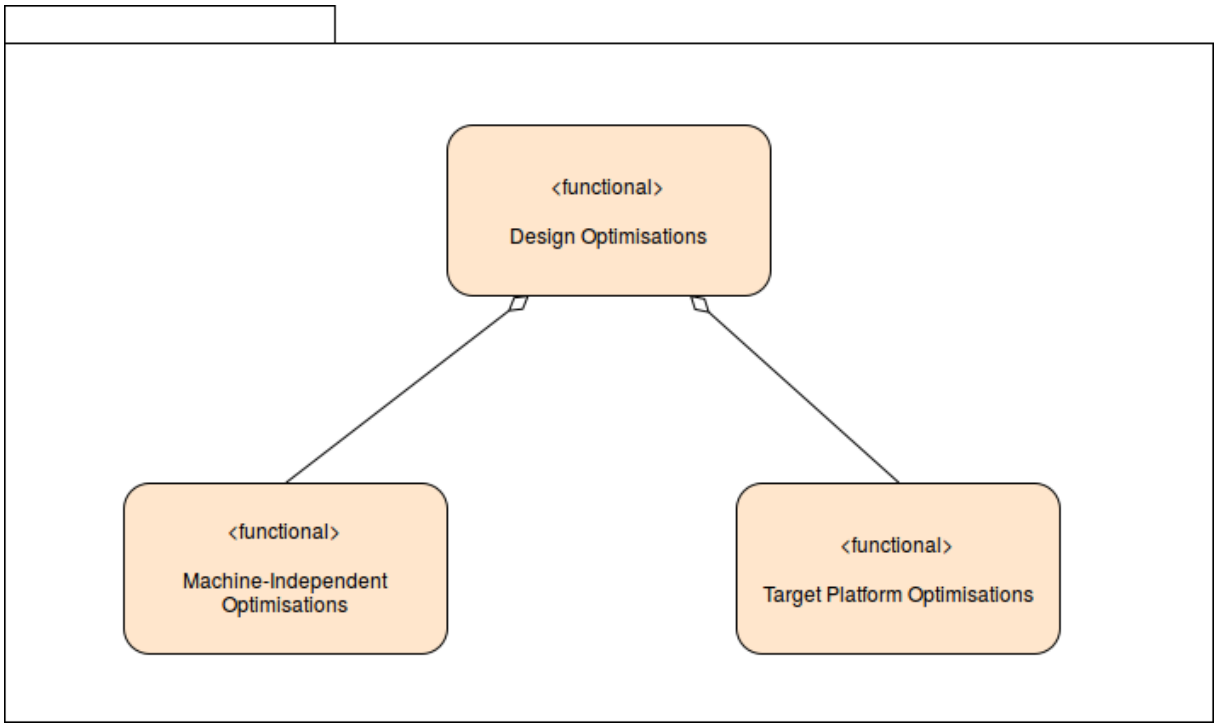

Figure A.8: Design optimisations.

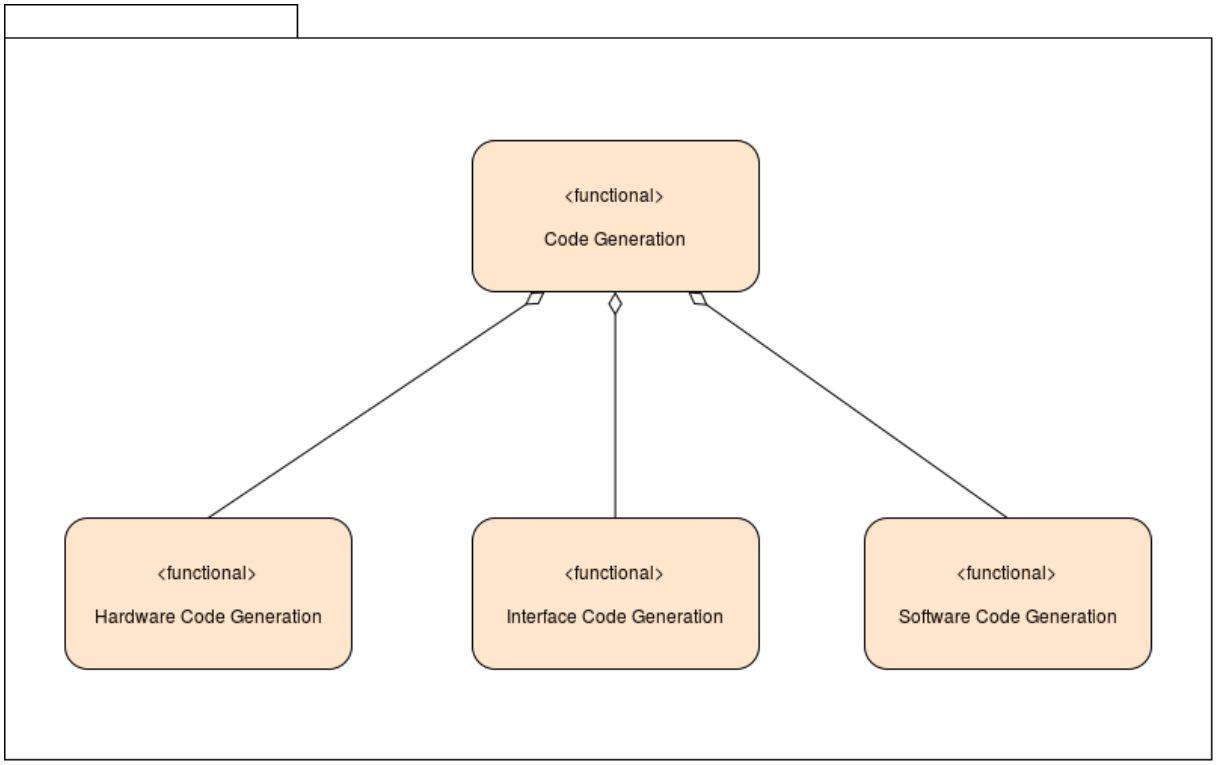

Figure A.9: Code Generation.

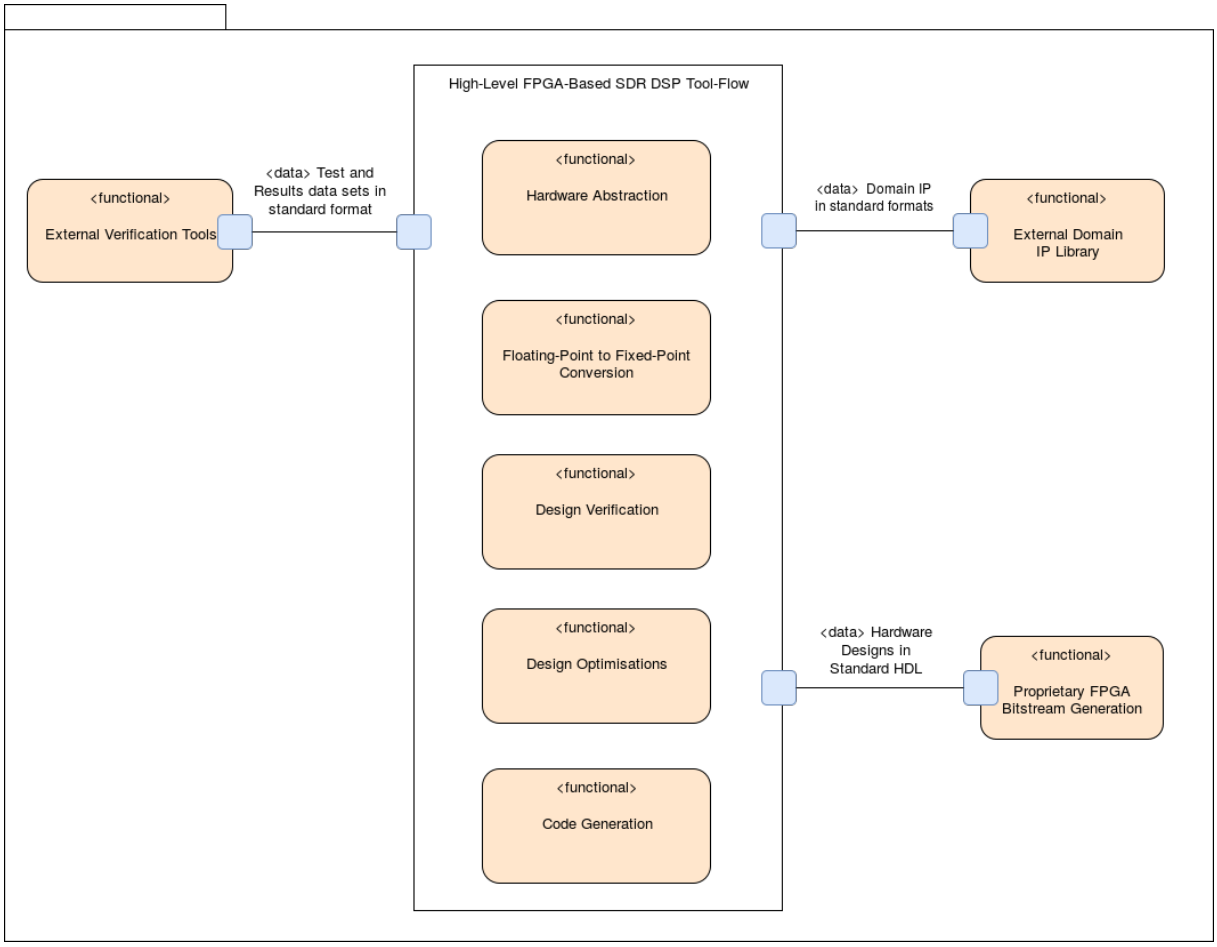

Figure A.10: High-level FPGA-based SDR DSP Toolflow functional structure and interface diagram.

APPENDIX B

FPGA TOOLS - SOURCE CODE

Source code for the FIR filter case studies conducted across each of the six high-level design tools being evaluated in this dissertation is presented in detail in this appendix. The appendix includes code for the high-level FIR filter design specification and associated high-level test-bench and the generated hardware (VHDL/Verilog) or software (for tools lacking hardware targeting) and test-bench.

B.1 MyHDL

B.1.1 FIR Filter High-Level Specification

```
from myhdl import *

def FIR(clock, reset, sig_in, sig_out, coef):
    taps = [Signal(intbv(0, min=sig_in.min, max=sig_in.max))
            for ii in range(len(coef))]
    # FIR taps
    coef = tuple(coef)
    mshift = len(sig_in)-1

    @always(clock.posedge)
    def rtl_sop():
        if reset:
            for ii in range(len(coef)):
                taps[ii].next = 0
            sig_out.next = 0
        else:
            sop = 0
            for ii in range(len(coef)):
                if ii == 0:
                    taps[ii].next = sig_in
                else:
                    taps[ii].next = taps[ii-1]
                c = coef[ii]
                sop = sop + (taps[ii] * c)
            sig_out.next = (sop >> mshift)
    return rtl_sop
```

Listing B.1: MyHDL FIR filter code example

B.1.2 FIR Filter High-Level Test Bench

```
import sys
import os
import argparse
from argparse import Namespace

from myhdl import *
from rhea import *

import numpy as np
from numpy import pi, log10
```

```
from scipy import signal
from matplotlib import pyplot as plt
from matplotlib import mlab

# various forms of a FIR filter
from firfilt import m_firfilt

class SignalStream(object):
    def __init__(self, stype):
        self.val = Signal(stype)
        self.dv = Signal(bool(0))

def test_firfilt(args):
    # what word size do we want?
    sMax = 2**15; sMin=-1*sMax

    ''' Get the filter coefficients and plot the response '''
    coef = signal.fir_filter_design.firwin(args.Nflt, args.Fc)
    icoef = map(int, coef*sMax)
    w,H = signal.freqz(coef)
    fig1,ax1 = plt.subplots(1)
    ax1.plot(w, 20*log10(abs(H)))
    ax1.grid(True)

    sig_in = Signal(intbv(0, min=sMin, max=sMax))
    sig_out = Signal(intbv(0, min=sMin, max=sMax))
    clock = Clock(0, frequency=args.Fs)
    reset = Reset(False, active=1, async=False)

    ''' Print FIR coefficients -- debugging '''
    def print_coef():
            for c in range(len(coef)):
                print(coef[c])
    #yield clock.posedge

    ''' Test FIR filter '''
    def _test_firfilt():
        x,y = (sig_in,sig_out,)
        if args.trace:
            tb_dut = traceSignals(m_firfilt,clock,reset,x,y,icoef)
        else:
            tb_dut = m_firfilt(clock,reset,x,y,icoef)
        tb_clk = clock.gen(hticks=5)

        @instance
        def tb_stimulus():
            # pulse the reset
            yield reset.pulse(100)
            for ii in xrange(2):
                yield clock.posedge

            # chirp 1 (time response pictoral)
            print("   chirp 1 ...")
            samp_in = signal.chirp(np.arange(args.Nsamps/2)*1/args.Fs,
                                   8, .64, 480,
                                   method=u'logarithmic')*.94
            samp_in = np.concatenate(
                (samp_in,
                 np.array([ss for ss in reversed(samp_in[:-1])] )))
            samp_out = []

            # input samples, save the output
            for ii in xrange(args.Nsamps-1):
                sig_in.next = int(np.floor(samp_in[ii]*(sMax)))
                yield clock.posedge
                samp_out.append(float(sig_out)/float(sMax))

            samp_out = np.array(samp_out)
            c = signal.lfilter(coef, 1, samp_in)
```

```
            sdiff = np.abs(c[:-2] - samp_out[2:])
            plt.figure(3); plt.plot(sdiff)
            assert np.max(sdiff) < 1e-3, "error too large"
            ia = np.concatenate((np.ones(args.Nflt/2)*.98, samp_in))
            fig,ax = plt.subplots(1)
            ax.plot(ia, 'b');
            ax.plot(samp_out[1:], 'r');
            ax.plot(c, 'y--')
            fig.savefig('__plot2.png')

            # chirp 2 (frequency response, more points)
            print("   chrip 2 ...")
            Nfft = 8*args.Nsamps
            samp_in = signal.chirp(np.arange(Nfft)*1/args.Fs,
                                   0.1, 1, 500)*.98
            samp_out = []
            for ii in xrange(Nfft):
                sig_in.next = int(np.floor(samp_in[ii]*(sMax)))
                yield clock.posedge
                samp_out.append(float(sig_out)/float(sMax))
            samp_out = np.array(samp_out)
            Pi,fi = mlab.psd(samp_in)
            Po,fo = mlab.psd(samp_out)
            ax1.plot(pi*fi, 10*log10(abs(Po/Pi)), 'r')
            ax1.grid(True)
            fig1.savefig('__plot1.png')

            raise StopSimulation

        g = (tb_dut,tb_clk,tb_stimulus,)
        return g

    fig1.savefig('fir__plot1.png')

    ''' convert FIR module to VHDL and to Verilog '''
    toVHDL(m_firfilt,clock, reset,sig_in, sig_out, icoef)
    toVerilog(m_firfilt,clock, reset,sig_in, sig_out, icoef)

    # run simulation
    Simulation(_test_firfilt()).run()

if __name__ == '__main__':
    args = Namespace(Nsamps=1024,
                     Fs=170e6,
                     Nflt=63,
                     Fc=8e9,
                     trace=True)
    test_firfilt(args)
```

Listing B.2: MyHDL FIR filter code example

B.1.3 FIR Filter Generated HDL

```
// File: m_firfilt.v
// Generated by MyHDL 1.0dev
// Date: Tue Aug 28 05:47:08 2018


`timescale 1ns/10ps

module m_firfilt (
    clock,
    reset,
    sig_in,
    sig_out
);
//

input clock;
```

```
17 input reset;
18 input signed [15:0] sig_in;
19 output signed [15:0] sig_out;
20 reg signed [15:0] sig_out;
21
22 reg signed [15:0] taps [0:63-1];
23
24
25
26 always @(posedge clock) begin: M_FIRFILT_RTL_SOP
27     integer ii;
28     integer sop;
29     integer c;
30     if (reset) begin
31         for (ii=0; ii<63; ii=ii+1) begin
32             taps[ii] <= 0;
33         end
34         sig_out <= 0;
35     end
36     else begin
37         sop = 0;
38         for (ii=0; ii<63; ii=ii+1) begin
39             if ((ii == 0)) begin
40                 taps[ii] <= sig_in;
41             end
42             else begin
43                 taps[ii] <= taps[(ii - 1)];
44             end
45             case (ii)
46                 0: c = 37;
47                 1: c = 38;
48                 2: c = 38;
49                 3: c = 40;
50                 4: c = 42;
51                 5: c = 44;
52                 6: c = 47;
53                 7: c = 50;
54                 8: c = 54;
55                 9: c = 58;
56                 10: c = 63;
57                 11: c = 68;
58                 12: c = 74;
59                 13: c = 80;
60                 14: c = 87;
61                 15: c = 94;
62                 16: c = 101;
63                 17: c = 109;
64                 18: c = 117;
65                 19: c = 125;
66                 20: c = 134;
67                 21: c = 143;
68                 22: c = 153;
69                 23: c = 162;
70                 24: c = 172;
71                 25: c = 182;
72                 26: c = 192;
73                 27: c = 202;
74                 28: c = 213;
75                 29: c = 223;
76                 30: c = 234;
77                 31: c = 245;
78                 32: c = 255;
79                 33: c = 266;
80                 34: c = 277;
81                 35: c = 287;
82                 36: c = 298;
83                 37: c = 308;
84                 38: c = 318;
85                 39: c = 329;
```

```
                    40: c = 339;
                    41: c = 348;
                    42: c = 358;
                    43: c = 367;
                    44: c = 376;
                    45: c = 385;
                    46: c = 393;
                    47: c = 401;
                    48: c = 409;
                    49: c = 417;
                    50: c = 424;
                    51: c = 430;
                    52: c = 436;
                    53: c = 442;
                    54: c = 447;
                    55: c = 452;
                    56: c = 456;
                    57: c = 460;
                    58: c = 464;
                    59: c = 466;
                    60: c = 469;
                    61: c = 471;
                    default: c = 37;
                endcase
                sop = ($signed({1'b0, sop}) + (taps[ii] * c));
            end
            sig_out <= $signed(sop >>> 15);
        end
    end

    endmodule
```

Listing B.3: MyHDL generated Verilog FIR filter code

B.1.4 FIR Filter HDL Test Bench

```
module tb_m_firfilt;

reg clock;
reg reset;
reg [15:0] sig_in;
wire [15:0] sig_out;

initial begin
    $from_myhdl(
        clock,
        reset,
        sig_in
    );
    $to_myhdl(
        sig_out
    );
end

m_firfilt dut(
    clock,
    reset,
    sig_in,
    sig_out
);

endmodule
```

Listing B.4: MyHDL generated Verilog FIR filter test-bench

B.2 MIGEN

B.2.1 FIR Filter High-Level Specification

```
''' A synthesizable FIR filter '''
class FIR(Module):
    def __init__(self, coef, wsize=16):
        self.coef = coef
  # FIR coefficients
        self.wsize = wsize
        self.i = Signal((self.wsize, True))
  # signal for input samples
        self.o = Signal((self.wsize, True))
  # signal for output samples

        ###
        muls = []
        src = self.i
        for c in self.coef:
            sreg = Signal((self.wsize, True))
            self.sync += sreg.eq(src)
      # copy input samples into the FIR register, synchronously
            src = sreg
            coef_fp = int(c*2**(self.wsize - 1))
      # convert coefficient to fixed point
            muls.append(coef_fp*sreg)
      # compute FIR products
        sum_full = Signal((2*self.wsize-1, True))
        self.sync += sum_full.eq(reduce(add, muls))
  # sum the products, synchronously
        self.comb += self.o.eq(sum_full >> self.wsize-1)
  # write output to output port, combinatorially
```

Listing B.5: MyHDL FIR filter code example

B.2.2 FIR Filter High-Level Test Bench

```
'''
@name: Migen FIR
@author: K.Setetemela
@date: May 2018
@purpose:
  -simulate FIR filter
  -plot results
  -generate HDL

'''

from functools import reduce
from operator import add

from math import cos, pi
from scipy import signal
import matplotlib.pyplot as plt

from migen import *
from migen.fhdl import verilog
from migen.fhdl.verilog import convert


''' A synthesizable FIR filter module'''
class FIR(Module):
    def __init__(self, coef, wsize=16):
        self.coef = coef
        self.wsize = wsize
        self.i = Signal((self.wsize, True))
        self.o = Signal((self.wsize, True))

        ###

        muls = []#products
        src = self.i
  #16 bits input sample
```

```
        for c in self.coef:
  #for each of the 30 coefficients
            sreg = Signal((self.wsize, True))
      #empty 16 bits sample register
            self.sync += sreg.eq(src)
      #load input sample onto the sample register at next rising clock edge
            src = sreg
      #store sample again on src
            c_fp = int(c*2**(self.wsize - 1))
      #convert float coefficient to fixed point
            coefficient =Signal((self.wsize, True))
            self.sync += coefficient.eq(c_fp)
            muls.append(c_fp*sreg)
      #compute fir product for this coefficient and this input sample and append product
        sum_full = Signal((2*self.wsize-1, True))
  #31 bits signal
        self.sync += sum_full.eq(reduce(add, muls))
        self.comb += self.o.eq(sum_full >> self.wsize-1)

'''
    A test bench for our FIR filter.
    Generates a sine wave at the input and records the output.
'''
def fir_tb(dut, frequency, inputs, outputs):
    f = 2**(dut.wsize - 1)
    for cycle in range(200):
        v = 0.1*cos(2*pi*frequency*cycle)
  # sine sample for n = cycle
        yield dut.i.eq(int(f*v))
  # convert sample to fixed point and write to FIR input port
        inputs.append(v)
  # adds v to the input samples list
        outputs.append((yield dut.o)/f)
  # convert sample back to floating point and write to output samples list
        yield

''' MAIN '''
if __name__ == "__main__":
    Fs=170e6
    cutoff=8e9
    stop=10e9
    Wcutoff=cutoff/(Fs)
    Wstop=stop/(Fs)
    # Compute lowpass filter coefficients with SciPy.
    coef = signal.remez(63, [0, Wcutoff,Wstop, 0.5], [0, 1])

    '''Simulate for different frequencies and concatenate the results.'''
    in_signals = []
    out_signals = []
    for frequency in [0.2, 0.3, 0.9]:
        dut = FIR(coef)
        tb = fir_tb(dut, frequency, in_signals, out_signals)
        run_simulation(dut, tb,vcd_name="fir.vcd")

    ''' Plot data from the input and output waveforms.'''
    plt.plot(in_signals)
    plt.plot(out_signals)
    plt.show()

    # Print the Verilog source for the filter.
    fir = FIR(coef)
    #print(verilog.convert(fir, ios={fir.i, fir.o}))
    verilog.convert(fir, ios={fir.i, fir.o}).write("FIR.v")
```

Listing B.6: Migen FIR filter code example

B.2.3 FIR Filter Generated HDL

```
/* Machine-generated using Migen */
```

```
2  module top(
3    input signed [15:0] i,
4    output signed [15:0] o,
5    input sys_clk,
6    input sys_rst
7  );
8
9  reg signed [15:0] sreg0 = 1'd0;
10 reg signed [15:0] coefficient0 = 1'd0;
11 reg signed [15:0] sreg1 = 1'd0;
12 reg signed [15:0] coefficient1 = 1'd0;
13 reg signed [15:0] sreg2 = 1'd0;
14 reg signed [15:0] coefficient2 = 1'd0;
15 reg signed [15:0] sreg3 = 1'd0;
16 reg signed [15:0] coefficient3 = 1'd0;
17 reg signed [15:0] sreg4 = 1'd0;
18 reg signed [15:0] coefficient4 = 1'd0;
19 reg signed [15:0] sreg5 = 1'd0;
20 reg signed [15:0] coefficient5 = 1'd0;
21 reg signed [15:0] sreg6 = 1'd0;
22 reg signed [15:0] coefficient6 = 1'd0;
23 reg signed [15:0] sreg7 = 1'd0;
24 reg signed [15:0] coefficient7 = 1'd0;
25 reg signed [15:0] sreg8 = 1'd0;
26 reg signed [15:0] coefficient8 = 1'd0;
27 reg signed [15:0] sreg9 = 1'd0;
28 reg signed [15:0] coefficient9 = 1'd0;
29 reg signed [15:0] sreg10 = 1'd0;
30 reg signed [15:0] coefficient10 = 1'd0;
31 reg signed [15:0] sreg11 = 1'd0;
32 reg signed [15:0] coefficient11 = 1'd0;
33 reg signed [15:0] sreg12 = 1'd0;
34 reg signed [15:0] coefficient12 = 1'd0;
35 reg signed [15:0] sreg13 = 1'd0;
36 reg signed [15:0] coefficient13 = 1'd0;
37 reg signed [15:0] sreg14 = 1'd0;
38 reg signed [15:0] coefficient14 = 1'd0;
39 reg signed [15:0] sreg15 = 1'd0;
40 reg signed [15:0] coefficient15 = 1'd0;
41 reg signed [15:0] sreg16 = 1'd0;
42 reg signed [15:0] coefficient16 = 1'd0;
43 reg signed [15:0] sreg17 = 1'd0;
44 reg signed [15:0] coefficient17 = 1'd0;
45 reg signed [15:0] sreg18 = 1'd0;
46 reg signed [15:0] coefficient18 = 1'd0;
47 reg signed [15:0] sreg19 = 1'd0;
48 reg signed [15:0] coefficient19 = 1'd0;
49 reg signed [15:0] sreg20 = 1'd0;
50 reg signed [15:0] coefficient20 = 1'd0;
51 reg signed [15:0] sreg21 = 1'd0;
52 reg signed [15:0] coefficient21 = 1'd0;
53 reg signed [15:0] sreg22 = 1'd0;
54 reg signed [15:0] coefficient22 = 1'd0;
55 reg signed [15:0] sreg23 = 1'd0;
56 reg signed [15:0] coefficient23 = 1'd0;
57 reg signed [15:0] sreg24 = 1'd0;
58 reg signed [15:0] coefficient24 = 1'd0;
59 reg signed [15:0] sreg25 = 1'd0;
60 reg signed [15:0] coefficient25 = 1'd0;
61 reg signed [15:0] sreg26 = 1'd0;
62 reg signed [15:0] coefficient26 = 1'd0;
63 reg signed [15:0] sreg27 = 1'd0;
64 reg signed [15:0] coefficient27 = 1'd0;
65 reg signed [15:0] sreg28 = 1'd0;
66 reg signed [15:0] coefficient28 = 1'd0;
67 reg signed [15:0] sreg29 = 1'd0;
68 reg signed [15:0] coefficient29 = 1'd0;
69 reg signed [15:0] sreg30 = 1'd0;
70 reg signed [15:0] coefficient30 = 1'd0;
```

```
71  reg signed [15:0] sreg31 = 1'd0;
72  reg signed [15:0] coefficient31 = 1'd0;
73  reg signed [15:0] sreg32 = 1'd0;
74  reg signed [15:0] coefficient32 = 1'd0;
75  reg signed [15:0] sreg33 = 1'd0;
76  reg signed [15:0] coefficient33 = 1'd0;
77  reg signed [15:0] sreg34 = 1'd0;
78  reg signed [15:0] coefficient34 = 1'd0;
79  reg signed [15:0] sreg35 = 1'd0;
80  reg signed [15:0] coefficient35 = 1'd0;
81  reg signed [15:0] sreg36 = 1'd0;
82  reg signed [15:0] coefficient36 = 1'd0;
83  reg signed [15:0] sreg37 = 1'd0;
84  reg signed [15:0] coefficient37 = 1'd0;
85  reg signed [15:0] sreg38 = 1'd0;
86  reg signed [15:0] coefficient38 = 1'd0;
87  reg signed [15:0] sreg39 = 1'd0;
88  reg signed [15:0] coefficient39 = 1'd0;
89  reg signed [15:0] sreg40 = 1'd0;
90  reg signed [15:0] coefficient40 = 1'd0;
91  reg signed [15:0] sreg41 = 1'd0;
92  reg signed [15:0] coefficient41 = 1'd0;
93  reg signed [15:0] sreg42 = 1'd0;
94  reg signed [15:0] coefficient42 = 1'd0;
95  reg signed [15:0] sreg43 = 1'd0;
96  reg signed [15:0] coefficient43 = 1'd0;
97  reg signed [15:0] sreg44 = 1'd0;
98  reg signed [15:0] coefficient44 = 1'd0;
99  reg signed [15:0] sreg45 = 1'd0;
100 reg signed [15:0] coefficient45 = 1'd0;
101 reg signed [15:0] sreg46 = 1'd0;
102 reg signed [15:0] coefficient46 = 1'd0;
103 reg signed [15:0] sreg47 = 1'd0;
104 reg signed [15:0] coefficient47 = 1'd0;
105 reg signed [15:0] sreg48 = 1'd0;
106 reg signed [15:0] coefficient48 = 1'd0;
107 reg signed [15:0] sreg49 = 1'd0;
108 reg signed [15:0] coefficient49 = 1'd0;
109 reg signed [15:0] sreg50 = 1'd0;
110 reg signed [15:0] coefficient50 = 1'd0;
111 reg signed [15:0] sreg51 = 1'd0;
112 reg signed [15:0] coefficient51 = 1'd0;
113 reg signed [15:0] sreg52 = 1'd0;
114 reg signed [15:0] coefficient52 = 1'd0;
115 reg signed [15:0] sreg53 = 1'd0;
116 reg signed [15:0] coefficient53 = 1'd0;
117 reg signed [15:0] sreg54 = 1'd0;
118 reg signed [15:0] coefficient54 = 1'd0;
119 reg signed [15:0] sreg55 = 1'd0;
120 reg signed [15:0] coefficient55 = 1'd0;
121 reg signed [15:0] sreg56 = 1'd0;
122 reg signed [15:0] coefficient56 = 1'd0;
123 reg signed [15:0] sreg57 = 1'd0;
124 reg signed [15:0] coefficient57 = 1'd0;
125 reg signed [15:0] sreg58 = 1'd0;
126 reg signed [15:0] coefficient58 = 1'd0;
127 reg signed [15:0] sreg59 = 1'd0;
128 reg signed [15:0] coefficient59 = 1'd0;
129 reg signed [15:0] sreg60 = 1'd0;
130 reg signed [15:0] coefficient60 = 1'd0;
131 reg signed [15:0] sreg61 = 1'd0;
132 reg signed [15:0] coefficient61 = 1'd0;
133 reg signed [15:0] sreg62 = 1'd0;
134 reg signed [15:0] coefficient62 = 1'd0;
135 reg signed [30:0] sum_full = 1'd0;
136
137 // synthesis translate_off
138 reg dummy_s;
139 initial dummy_s <= 1'd0;
```

```
140 // synthesis translate_on
141 assign o = (sum_full >>> 4'd15);
142
143 always @(posedge sys_clk) begin
144   if (sys_rst) begin
145     sreg0 <= 1'd0;
146     coefficient0 <= 1'd0;
147     sreg1 <= 1'd0;
```

Listing B.7: Migen generated Verilog FIR filter code example

B.2.4 **FIR Filter HDL Test Bench**

Migen does not generate an automatic test-bench.

APPENDIX C

GPU TOOLS - SOURCE CODE

C.1 OPENCL

C.1.1 FIR Filter High-Level Specification

```
/*
   OpenCL FIR filter kernel
   input: input samples
   coeff: FIR coefficients
   output: FIR output samples
   lengths: threads block dimensions
*/
__kernel void fir(__global double* input,
          __global double* output,
    __global  double* coeff,
     __global int* lengths) {

                int idx, acc;
    acc = 0;
    idx = get_global_id(0);
        // set index of this thread in the block
                double sumprod=0;
    /* compute FIR sum of products for this thread */
    for(int j = 0; j<lengths[1]; j++){
      sumprod += input[idx-j] * coeff[j];
    }
    // store result in output buffer
    output[idx] = sumprod;
}
```

Listing C.1: OpenCL FIR filter code example

C.1.2 FIR Filter High-Level Test Bench

```
/*
name: OpenCL FIR
author: K. Setetemela
date: August 2018
purpose:
  - reads samples from file
  - applies FIR filtering operation on a GPU
  - writes results to file
  - profiles both the host and device operations
*/


#include <ctime>
#include <iostream>
#include <cstdint>
#include <fstream>
#include <cstring>
#include <math.h>
```

```
19 #include <stdio.h>
20 #include <stdlib.h>
21 #include <string.h>
22 #include <time.h>
23
24 using namespace std;
25
26 #ifdef MAC
27 #include <OpenCL/cl.h>
28 #else
29 #include <CL/cl.h>
30 #endif
31
32 #define PROGRAM_FILE "fir.cl"
33 #define KERNEL_FUNC "fir"
34
35 #define MAX_INPUT_SAMPLES 128000
36 #define FIR_LEN 63
37 #define BUFFER_LEN (MAX_INPUT_SAMPLES+FIR_LEN)
38 size_t NUM_GLOBAL_WITEMS;
39 size_t NUM_LOCAL_WITEMS;
40 double BUFFER[BUFFER_LEN];
41 double coeffs[ FIR_LEN ] =
42 /* lowpass FIR filter:
43    passband: 0 - 8MHz
44    stopband: 8 - 10MHz
45    passband ripple: 2.3%
46    stopband attenuation: 40dB
47 */
48 {
49   -0.0448093,  0.0322875,   0.0181163,   0.0087615,   0.0056797,
50    0.0086685,  0.0148049,   0.0187190,   0.0151019,   0.0027594,
51   -0.0132676, -0.0232561,  -0.0187804,   0.0006382,   0.0250536,
52    0.0387214,  0.0299817,   0.0002609,  -0.0345546,  -0.0525282,
53   -0.0395620,  0.0000246,   0.0440998,   0.0651867,   0.0479110,
54    0.0000135, -0.0508558,  -0.0736313,  -0.0529380,  -0.0000709,
55    0.0540186,  0.0766746,   0.0540186,  -0.0000709,  -0.0529380,
56   -0.0736313, -0.0508558,   0.0000135,   0.0479110,   0.0651867,
57    0.0440998,  0.0000246,  -0.0395620,  -0.0525282,  -0.0345546,
58    0.0002609,  0.0299817,   0.0387214,   0.0250536,   0.0006382,
59   -0.0187804, -0.0232561,  -0.0132676,   0.0027594,   0.0151019,
60    0.0187190,  0.0148049,   0.0086685,   0.0056797,   0.0087615,
61    0.0181163,  0.0322875,  -0.0448093
62 };
63
64 /* initialise FIR buffer with zeros */
65 void initFIRBuffer();
66 /* initialise GPU device */
67 cl_device_id create_device();
68 /* build kernel for device */
69 cl_program build_program(cl_context ctx, cl_device_id dev, const char* filename);
70 /* convert fixed point values to floating point */
71 void intToFloat(int8_t* input, double * output,int length);
72 /* convert floating point values to fixed point*/
73 void floatToInt(double * input, int8_t* output,int length);
74
75
76 int main() {
77
78    /* OpenCL structures */
79    cl_device_id device;
80    cl_context context;
81    cl_program program;
82    cl_kernel kernel;
83    cl_command_queue queue;
84    cl_int i, j, err;
85    size_t local_size, global_size;
86
87    /* Data and buffers */
```

```
88    int8_t file_input_int[MAX_INPUT_SAMPLES];
89    double* dfloatOutput;
90    int8_t* doutput;
91    cl_mem input_buffer, output_buffer, lengths_buffer, coeffs_buffer, fir_length_buffer;
92    cl_int num_groups;
93
94
95    /* Initialize data */
96    ifstream fin;
97    ofstream hfout,dfout;
98    fin.open("../data/11k8bitpcm.wav",ios::binary|ios::in);
99
100   fin.read(reinterpret_cast<char*>(&file_input_int),MAX_INPUT_SAMPLES*sizeof(int8_t));
101   int actual_input_samples=fin.gcount();
102
103   double* floatInput=new double[actual_input_samples];
104
105   doutput=new int8_t[actual_input_samples];
106
107   intToFloat(file_input_int,floatInput,actual_input_samples);
108
109   initFIRBuffer();
110
111   //store the new inputlen samples into the higher end of the fir buffer
112   memcpy(&BUFFER[FIR_LEN-1],floatInput,actual_input_samples*sizeof(double));
113
114   /* Create device and context */
115   device = create_device();
116   context = clCreateContext(NULL, 1, &device, NULL, NULL, &err);
117   if(err < 0) {
118      perror("Couldn't create a context");
119      exit(1);
120   }else{
121     printf("[OK] Created context\n");
122   }
123
124   /* Build program */
125   program = build_program(context, device, PROGRAM_FILE);
126
127   /* Create data buffers and initialise them */
128   NUM_GLOBAL_WITEMS=actual_input_samples;
129   NUM_LOCAL_WITEMS=1;
130   dfloatOutput=new double[actual_input_samples];
131   int * lengths=new int[2];
132   lengths[0]=actual_input_samples;
133   lengths[1]=FIR_LEN;
134
135   input_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
       actual_input_samples* sizeof(double), floatInput, &err);
136   output_buffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
       actual_input_samples * sizeof(double), dfloatOutput, &err);
137   coeffs_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY  | CL_MEM_COPY_HOST_PTR,
       FIR_LEN * sizeof(double), coeffs, &err);
138   lengths_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY  | CL_MEM_COPY_HOST_PTR, 2*
       sizeof(int), lengths, &err);
139
140   if(err < 0) {
141      perror("Couldn't create a buffer");
142      exit(1);
143   }else{
144     printf("[OK] Created input and output data buffers\n");
145  }
146
147   /* Create a command queue */
148   queue = clCreateCommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE, &err);
149   if(err < 0) {
150      perror("Couldn't create a command queue");
151      exit(1);
152   }else{
```

```
153     printf("[OK] Created command queue on device\n");
154   }
155
156   /* Write data onto device buffers and profile each operation*/
157    unsigned long start = 0;
158    unsigned long end = 0;
159
160    cl_event input_buffer_transfer;
161    err =clEnqueueWriteBuffer(queue, input_buffer, CL_TRUE, 0, actual_input_samples*
        sizeof(double), floatInput, 0, nullptr, &input_buffer_transfer);
162    clWaitForEvents(1, &input_buffer_transfer);
163
164    clGetEventProfilingInfo(input_buffer_transfer, CL_PROFILING_COMMAND_START, sizeof (
        cl_ulong), &start, NULL);
165    clGetEventProfilingInfo(input_buffer_transfer, CL_PROFILING_COMMAND_END, sizeof (
        cl_ulong), &end, NULL);
166    unsigned long  input_buffer_transfer_time= end - start;
167
168    cl_event output_buffer_transfer;
169    err |=clEnqueueWriteBuffer(queue, output_buffer, CL_TRUE, 0, actual_input_samples*
        sizeof(double), dfloatOutput, 0, nullptr, &output_buffer_transfer);
170    clWaitForEvents(1, &output_buffer_transfer);
171    clGetEventProfilingInfo(output_buffer_transfer, CL_PROFILING_COMMAND_START, sizeof (
        cl_ulong), &start, NULL);
172    clGetEventProfilingInfo(output_buffer_transfer, CL_PROFILING_COMMAND_END, sizeof (
        cl_ulong), &end, NULL);
173    unsigned long  output_buffer_transfer_time= end - start;
174
175
176    cl_event coeffs_buffer_transfer;
177    err |=clEnqueueWriteBuffer(queue, coeffs_buffer, CL_TRUE, 0, FIR_LEN * sizeof(double),
         coeffs, 0, nullptr, &coeffs_buffer_transfer);
178    clWaitForEvents(1, &coeffs_buffer_transfer);
179    clGetEventProfilingInfo(coeffs_buffer_transfer, CL_PROFILING_COMMAND_START, sizeof (
        cl_ulong), &start, NULL);
180    clGetEventProfilingInfo(coeffs_buffer_transfer, CL_PROFILING_COMMAND_END, sizeof (
        cl_ulong), &end, NULL);
181    unsigned long  coeffs_buffer_transfer_time= end - start;
182
183    cl_event lengths_buffer_transfer;
184    err |=clEnqueueWriteBuffer(queue, lengths_buffer, CL_TRUE, 0, 2*sizeof(int), lengths,
        0, nullptr, &lengths_buffer_transfer);
185    clWaitForEvents(1, &lengths_buffer_transfer);
186    clGetEventProfilingInfo(lengths_buffer_transfer, CL_PROFILING_COMMAND_START, sizeof (
        cl_ulong), &start, NULL);
187    clGetEventProfilingInfo(lengths_buffer_transfer, CL_PROFILING_COMMAND_END, sizeof (
        cl_ulong), &end, NULL);
188    unsigned long  lengths_buffer_transfer_time= end - start;
189    if(err < 0) {
190       perror("Couldn't write data to device buffers");
191       exit(1);
192    }else{
193
194      printf("[OK] Written all data to device buffers\n");
195    }
196
197    /* Create a kernel */
198    kernel = clCreateKernel(program, KERNEL_FUNC, &err);
199    if(err < 0) {
200       perror("Couldn't create a kernel");
201       exit(1);
202    }else{
203
204      printf("[OK] Created kernel: %s\n",KERNEL_FUNC);
205    }
206
207    /* Create kernel arguments */
208    err = clSetKernelArg(kernel,  0, sizeof(cl_mem), &input_buffer);
209    err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &output_buffer);
```

```
210   err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &coeffs_buffer);
211   err |= clSetKernelArg(kernel, 3, sizeof(cl_mem), &lengths_buffer);
212
213   if(err < 0) {
214      perror("Couldn't create a kernel argument");
215      exit(1);
216   }
217   else{
218
219     printf("[OK] Created kernel arguments\n");
220   }
221
222   /* Enqueue kernel */
223   cl_event event;
224   err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &NUM_GLOBAL_WITEMS, &
       NUM_LOCAL_WITEMS, 0, NULL, &event);
225   clWaitForEvents(1, &event);
226   if(err < 0) {
227      perror("Couldn't enqueue the kernel");
228      exit(1);
229   }else{
230
231     printf("[OK] Enqueued the kernel on device\n");
232  }
233
234   /* Read the kernel's output and time the operation */
235   cl_event fir_output_transfer;
236   err = clEnqueueReadBuffer(queue, output_buffer, CL_TRUE, 0, actual_input_samples*
       sizeof(double), dfloatOutput, 0, NULL, &fir_output_transfer);
237   clWaitForEvents(1, &fir_output_transfer);
238   clGetEventProfilingInfo(fir_output_transfer, CL_PROFILING_COMMAND_START, sizeof (
       cl_ulong), &start, NULL);
239   clGetEventProfilingInfo(fir_output_transfer, CL_PROFILING_COMMAND_END, sizeof (
       cl_ulong), &end, NULL);
240   unsigned long  fir_output_transfer_time= end - start;
241   if(err < 0) {
242      perror("Couldn't read the buffer");
243      exit(1);
244   }else{
245
246     printf("[OK] Read output_buffer from device\n");
247  }
248
249
250   /* Check result and display profiling results */
251   clFinish(queue);
252   floatToInt(dfloatOutput,doutput,actual_input_samples);
253
254   cl_ulong time_start=0;
255   cl_ulong time_end=0;
256
257   clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(time_start), &
       time_start, NULL);
258   clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(time_end), &time_end,
       NULL);
259
260   double milliSeconds = (cl_double)(time_end-time_start)*(cl_double)(1e-06);
261   double data_transfer_time = (cl_double)(fir_output_transfer_time+
       lengths_buffer_transfer_time+
262                                coeffs_buffer_transfer_time+input_buffer_transfer_time+
       output_buffer_transfer_time)*(cl_double)(1e-06);
263   printf("Input size: %d \n",MAX_INPUT_SAMPLES);
264   printf("Kernel Execution Time: %0.5f milliseconds \n",milliSeconds);
265   printf("Total Data Transfer Time: %0.5f milliseconds \n",data_transfer_time);
266
267   /* Deallocate resources */
268   clReleaseKernel(kernel);
269   clReleaseMemObject(input_buffer);
270   clReleaseMemObject(output_buffer);
```

```
271     clReleaseMemObject(lengths_buffer);
272     clReleaseMemObject(coeffs_buffer);
273     clReleaseCommandQueue(queue);
274     clReleaseProgram(program);
275     clReleaseContext(context);
276     return 0;
277 }
278
279 /* initialise filter buffer */
280 void initFIRBuffer(){
281
282   memset(BUFFER,0,sizeof(BUFFER));
283 }
284
285
286 /* Find a GPU or CPU associated with the first available platform */
287 cl_device_id create_device() {
288
289     cl_platform_id platform;
290     cl_device_id dev;
291     int err;
292
293     /* Identify a platform */
294     err = clGetPlatformIDs(1, &platform, NULL);
295     if(err < 0) {
296        perror("Couldn't identify a platform");
297        exit(1);
298     }
299
300     /* Access a device */
301     err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &dev, NULL);
302     if(err == CL_DEVICE_NOT_FOUND) {
303        err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, 1, &dev, NULL);
304     }
305     if(err < 0) {
306        perror("Couldn't access any devices");
307        exit(1);
308     }
309
310     printf("[OK] Created device\n");
311
312     return dev;
313 }
314
315 /* Create program from a file and compile it */
316 cl_program build_program(cl_context ctx, cl_device_id dev, const char* filename) {
317
318     cl_program program;
319     FILE *program_handle;
320     char *program_buffer, *program_log;
321     size_t program_size, log_size;
322     int err;
323
324     /* Read program file and place content into buffer */
325     program_handle = fopen(filename, "r");
326     if(program_handle == NULL) {
327        perror("Couldn't find the program file");
328        exit(1);
329     }
330     fseek(program_handle, 0, SEEK_END);
331     program_size = ftell(program_handle);
332     rewind(program_handle);
333     program_buffer = (char*)malloc(program_size + 1);
334     program_buffer[program_size] = '\0';
335     fread(program_buffer, sizeof(char), program_size, program_handle);
336     fclose(program_handle);
337
338     /* Create program from file */
339     program = clCreateProgramWithSource(ctx, 1,
```

```
340         (const char**)&program_buffer, &program_size, &err);
341     if(err < 0) {
342         perror("Couldn't create the program");
343         exit(1);
344     }
345     else{
346         printf("[OK] Created program from kernel source\n");
347     }
348     free(program_buffer);
349 
350     /* Build program */
351     err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
352     if(err < 0) {
353 
354         /* Find size of log and print to std output */
355         clGetProgramBuildInfo(program, dev, CL_PROGRAM_BUILD_LOG,
356               0, NULL, &log_size);
357         program_log = (char*) malloc(log_size + 1);
358         program_log[log_size] = '\0';
359         clGetProgramBuildInfo(program, dev, CL_PROGRAM_BUILD_LOG,
360               log_size + 1, program_log, NULL);
361         printf("%s\n", program_log);
362         free(program_log);
363         exit(1);
364     }
365     else{
366         printf("[OK] Program built\n");
367 
368     }
369     return program;
370 }
371 
372 void intToFloat(int8_t* input, double * output,int length){
373 //convert an 8 bit integer array to a float array
374   for(int i=0;i<length;i++){
375     output[i]=(double)input[i];
376   }
377 }
378 
379 void floatToInt(double * input, int8_t* output,int length){
380 //convert a float array to an 8 bit integer array
381   for(int i=0;i<length;i++){
382     if ( input[i] > 127.0 ) {
383             input[i] = 127.0;
384      } else if ( input[i] < -128.0 ) {
385             input[i] = -128.0;
386      }
387      output[i]=(int8_t)input[i];
388   }
389 }
```

Listing C.2: OpenCL FIR filter test bench code

C.2 CUDA

C.2.1 FIR Filter High-Level Specification

```
1  /*
2     CUDA FIR filter kernel
3     d_data: input sample on device pinned memory
4     d_ffCoeff: FIR coefficients on device pinned memory
5     d_filteredData: FIR output on pinned device
6  */
7 
8  __global__ void filterData(double *d_data,
9                             double *d_ffCoeff,
10                            double *d_filteredData,
11                            const int ffLength,
12                            const int filteredDataLength){
```

```
    double sum = 0.00;
    double *d_ffCoeffp=d_ffCoeff;
    double *d_datap=d_data;

    int index=threadIdx.x;
    // set index of current thread in the block
    int stride=blockDim.x;
    // set threads to process data in equal chucks according to their number
    for (int  n = index; n < filteredDataLength; n+=stride ) {
      // let this thread begin at it's index and compute FIR output
      d_ffCoeffp=d_ffCoeff;
      double * activeinsamp=&d_datap[ffLength-1+n];
      sum=0.0f;
      for(int i=0;i<ffLength;i++){
  // compute FIR sum of products
  sum+=(*d_ffCoeffp++)*(*activeinsamp--);
      }
      d_filteredData[n]=sum;
      // store result of this thread
    }
}
```

Listing C.3: CUDA FIR filter code example

C.2.2 FIR Filter High-Level Test Bench

```
/*
name: CUDA FIR
author: K. Setetemela
date: August 2018
purpose:
  - reads samples from file
  - applies FIR filtering operation on a GPU
  - writes results to file
  - profiles both the host and device operations
*/


#include <ctime>
#include <stdio.h>
#include <cuda.h>
#include <assert.h>

/* Convenience function for checking CUDA runtime API results
can be wrapped around any runtime API call. No-op in release builds. */
inline
cudaError_t checkCuda(cudaError_t result)
{
#if defined(DEBUG) || defined(_DEBUG)
  if (result != cudaSuccess) {
    fprintf(stderr, "CUDA Runtime Error: %s\n",
            cudaGetErrorString(result));
    assert(result == cudaSuccess);
  }
#endif
  return result;
}

#define MAX_INPUT_SAMPLES 128000
// maximum FIR input samples
#define FIR_LEN 63
// FIR length
#define BUFFER_LEN (MAX_INPUT_SAMPLES+FIR_LEN)
// FIR buffer length

double h_filter_coeffs_o[ FIR_LEN ] =
/* Coefficients for low-pass FIR filter with:
   passband: 0 - 8MHz
   stopband: 8 - 10MHz
```

```
   passband ripple: 2.3%
   stopband attenuation: 40dB
*/
{
  -0.0448093,  0.0322875,   0.0181163,   0.0087615,   0.0056797,
   0.0086685,  0.0148049,   0.0187190,   0.0151019,   0.0027594,
  -0.0132676, -0.0232561,  -0.0187804,   0.0006382,   0.0250536,
   0.0387214,  0.0299817,   0.0002609,  -0.0345546,  -0.0525282,
  -0.0395620,  0.0000246,   0.0440998,   0.0651867,   0.0479110,
   0.0000135, -0.0508558,  -0.0736313,  -0.0529380,  -0.0000709,
   0.0540186,  0.0766746,   0.0540186,  -0.0000709,  -0.0529380,
  -0.0736313, -0.0508558,   0.0000135,   0.0479110,   0.0651867,
   0.0440998,  0.0000246,  -0.0395620,  -0.0525282,  -0.0345546,
   0.0002609,  0.0299817,   0.0387214,   0.0250536,   0.0006382,
  -0.0187804, -0.0232561,  -0.0132676,   0.0027594,   0.0151019,
   0.0187190,  0.0148049,   0.0086685,   0.0056797,   0.0087615,
   0.0181163,  0.0322875,  -0.0448093
};


/* CUDA FIR filter kernel declaration */
__global__ void filterData(double *d_data,double *d_ffCoeff,
        double *d_filteredData, const int ffLength,
        const int filteredDataLength);
/* method to convert fixed point values to floating point */
void intToFloat(int8_t* input, double * output,int length);
/* method to convert floating point values to fixed point */
void floatToInt(double * input, int8_t* output,int length);
/* initialise FIR buffer */
void firFloatInit( double* );


int main(void)
{

  double* buffer;
  checkCuda( cudaMallocHost((void**)&buffer,sizeof(double)*BUFFER_LEN) );
  // allocate pinned host memory for the buffer
  double* h_filter_coeffs;
  checkCuda( cudaMallocHost((void**)&h_filter_coeffs,sizeof(double)*FIR_LEN ) );
  // allocate pinned host memory for coefficients

  /* copy coefficients to pinned host memory */
  for(int i=0;i<FIR_LEN;i++){
    h_filter_coeffs[i]=h_filter_coeffs_o[i];
  }



  firFloatInit(buffer);

  /* declare input and output file handlers */
  FILE   *in_fid;
  FILE   *out_fid;

  /* open the input waveform file */
  in_fid = fopen( "../data/11k8bitpcm.wav", "rb" );
  if ( in_fid == 0 ) {
    printf("[ERROR] Couldn't open input file\n");
    return -1;
  }
  else{
    printf("[OK] Input file opened successfully\n");
  }

  /* open the output waveform file */
  out_fid = fopen( "../data/cuda_11k8bitpcm_output_parallel.wav", "wb" );
  if ( out_fid == 0 ) {
    printf("[ERROR] Couldn't open output file\n");
```

```
113     return -1;
114   }
115   else{
116     printf("[OK] Output file opened\n");
117   }
118
119   if(in_fid && out_fid){
120
121     int pcmHeaderLength=44;
122     char * pcmHeader = new char[pcmHeaderLength];
123     // read the pcm file header
124     fread(pcmHeader, sizeof(char),pcmHeaderLength, in_fid );
125     // write it to output file
126     fwrite(pcmHeader, sizeof(char),pcmHeaderLength, out_fid );
127
128     // Create device pointers
129     double *d_data = nullptr;
130     cudaMalloc((void **)&d_data,BUFFER_LEN * sizeof(double));
131     if(d_data){
132       printf("[OK] Allocated %d B on device for input samples\n",BUFFER_LEN);
133     }else{
134       printf("[ERROR] Coud not allocate %d B on device for input samples\n",BUFFER_LEN);
135       return -1;
136     }
137
138     /* allocate pinned device memory for filter coefficients */
139     double *d_numerator = nullptr;
140     cudaMalloc((void **)&d_numerator, FIR_LEN * sizeof(double));
141     if(d_numerator){
142       printf("[OK] Allocated %d B on device for filter coefficients\n",FIR_LEN);
143     }else{
144       printf("[ERROR] Coud not allocate %d B  on device for filter coefficients\n",
    FIR_LEN);
145       return -1;
146
147     }
148
149     /* Copy filter coefficients data to device */
150     cudaError_t err;
151     err= cudaMemcpy(d_numerator, h_filter_coeffs, FIR_LEN * sizeof(double),
    cudaMemcpyHostToDevice);
152     if(err==cudaSuccess){
153       printf("[OK] Copied %d B coefficients from host to device\n",FIR_LEN);
154     }
155     else{
156       printf("[ERROR] Could not copy %d B coefficients from host to device\n",FIR_LEN);
157       return -1;
158     }
159
160     int counts=1;
161     int size=0;
162
163     int8_t input[MAX_INPUT_SAMPLES];
164     int8_t *h_inputPinned;
165     //allocate pinned host memory for samples read from file
166     checkCuda( cudaMallocHost((void**)&h_inputPinned,sizeof(int8_t)*MAX_INPUT_SAMPLES) );
167
168     /* read input samples from file */
169     size = fread( input, sizeof(int8_t), MAX_INPUT_SAMPLES, in_fid );
170     if(size>0){
171       printf("[OK] %d  Input samples read successfully into memory\n",size);
172       counts++;
173     }
174     else{
175       printf("[ERROR] Failed to read input samples from input file\n");
176       return -1;
177     }
178
179     /* convert samples to floats */
```

```
180     double *h_data = new double[size];
181     intToFloat( input, h_data, size );
182 
183     //store the new input samples into the higher end of the buffer
184     memcpy(&buffer[FIR_LEN-1],h_data,size*sizeof(double));
185 
186     /* allocate memory on device for filter output samples */
187     double *d_filteredData = nullptr;
188     cudaMalloc((void **)&d_filteredData, size * sizeof(double));//pinned device memory
189     if(d_filteredData){
190       printf("[OK] Allocated %d B on device for FIR output\n",size);
191     }
192     else{
193       printf("[ERROR] Coud not allocate %d B  on device for FIR output\n",size);
194       return -1;
195     }
196 
197     /* Copy input samples to device */
198     err= cudaMemcpy(d_data, buffer, BUFFER_LEN * sizeof(double), cudaMemcpyHostToDevice);
199     if(err==cudaSuccess){
200       printf("[OK] Copied %d B of input from host to device\n",size);
201     }
202     else{
203       printf("[ERROR] Could not copy %d B of input from host to device\n",size);
204       return -1;
205     }
206     /* Copy filter coefficients to device */
207     err= cudaMemcpy(d_numerator, h_filter_coeffs, FIR_LEN * sizeof(double),
       cudaMemcpyHostToDevice);
208     if(err==cudaSuccess){
209       printf("[OK] Copied %d B coefficients from host to device\n",FIR_LEN);
210     }
211     else{
212       printf("[ERROR] Could not copy %d B coefficients from host to device\n",FIR_LEN);
213       return -1;
214     }
215 
216     /* Allocate pinned host memory to store filter output samples */
217     double *h_filteredData = new double[size];
218     double *h_filteredDataPinned;
219     checkCuda( cudaMallocHost((void**)&h_filteredDataPinned,sizeof(double)*size) );
220 
221 
222     /* Launch the kernel and being profiling */
223     int threadsPerBlock = 256;
224     int blocksPerGrid = (size + threadsPerBlock - 1) / threadsPerBlock;
225     float GPUKernelTime;
226     cudaEvent_t start, stop;
227     cudaEventCreate(&start);
228     cudaEventCreate(&stop);
229     cudaEventRecord(start);
230     filterData<<<blocksPerGrid,threadsPerBlock>>>(d_data, d_numerator, d_filteredData,
       FIR_LEN,size);
231     cudaEventRecord(stop);
232     cudaEventSynchronize(stop);
233 
234     GPUKernelTime=0;
235     cudaEventElapsedTime(&GPUKernelTime,start,stop);
236 
237     /*
238             shift input samples in fir buffer back in time for next time
239       the shift is such that fflen-1 current samples are carried forward into the next
       fir round
240       and merged with new samples
241       */
242     memmove(&buffer[0],&buffer[size],(FIR_LEN-1)*sizeof(double));
243 
244     /* Copy filtered results to host */
```

```
245     err=cudaMemcpy(h_filteredData, d_filteredData, size * sizeof(double),
        cudaMemcpyDeviceToHost);
246     if(err==cudaSuccess){
247       printf("[OK] Copied %d B output from device to host\n",size);
248     }
249     else{
250       printf("[ERROR] Could not copy %d B output from device to host\n",size);
251       return -1;
252     }
253
254     int8_t *output=new int8_t[size];
255     // convert filtered samples to ints
256     floatToInt( h_filteredData, output, size);
257     /* write filtered samples to file */
258     int sizew=fwrite( output, sizeof(int8_t), size, out_fid );
259     if(sizew!=0){
260       printf("[OK] Written %d B FIR output to output file\n",size);
261       printf("GPU Kernel Time (ms): %.8f\n",GPUKernelTime);
262     }
263     else{
264       printf("[ERROR] Could not write %d B FIR output to output file\n",size);
265       return -1;
266     }
267
268     /* close the files */
269     fclose( in_fid );
270     fclose( out_fid );
271
272   }
273
274 }
275
276 /* initialise FIR buffer with zeros */
277 void firFloatInit( double* buffer){
278     memset( buffer, 0, BUFFER_LEN );
279 }
280
281 /*
282    CUDA FIR filter kernel
283    d_data: input sample on device pinned memory
284    d_ffCoeff: FIR coefficients on device pinned memory
285    d_filteredData: FIR output on pinned device
286 */
287 __global__ void filterData(double *d_data,
288                            double *d_ffCoeff,
289                            double *d_filteredData,
290                            const int ffLength,
291                            const int filteredDataLength){
292     double sum = 0.00;
293     double *d_ffCoeffp=d_ffCoeff;
294     double *d_datap=d_data;
295
296     int index=threadIdx.x;
297     // set index of current thread in the block
298     int stride=blockDim.x;
299     // set threads to process data in equal chucks according to their number
300     for (int  n = index; n < filteredDataLength; n+=stride ) {
301       // let this thread begin at it's index and compute FIR output
302       d_ffCoeffp=d_ffCoeff;
303       double * activeinsamp=&d_datap[ffLength-1+n];
304       sum=0.0f;
305       for(int i=0;i<ffLength;i++){
306   // compute FIR sum of products
307   sum+=(*d_ffCoeffp++)*(*activeinsamp--);
308       }
309       d_filteredData[n]=sum;
310       // store result of this thread
311     }
312
```

```
313 }
314 /* convert an 8 bit integer array to a float array */
315 void intToFloat(int8_t* input, double * output,int length){
316
317   for(int i=0;i<length;i++){
318     output[i]=(double)input[i];
319   }
320
321
322 }
323 /* convert a float array to an 8 bit integer array */
324 void floatToInt(double * input, int8_t* output,int length){
325
326   for(int i=0;i<length;i++){
327     if ( input[i] > 127.0 ) {
328             input[i] = 127.0;
329     } else if ( input[i] < -128.0 ) {
330             input[i] = -128.0;
331     }
332     output[i]=(int8_t)input[i];
333   }
334 }
```

Listing C.4: CUDA FIR filter test bench code