SdrLift: A Domain-Specific Intermediate Hardware Synthesis Framework for Prototyping Software-Defined Radios

Lekhobola Joachim Tsoeunyane
MSc(Eng) UCT

Thesis Presented for the Degree of
DOCTOR OF PHILOSOPHY
in the Department of Electrical Engineering
UNIVERSITY OF CAPE TOWN

Rondebosch, July 13, 2020

We must ask for the love of crosses, and then they become sweet. I experienced this myself. I have been slandered. I have had crosses. I have had almost more than I could bear. I began to ask for the love of crosses; then I was happy. I said to myself, "Therein alone is found peace, happiness", *St. John Vianney.*

In memory of my grandfather

Thato Lawrence "Leronti" Tsoeunyane

1942–2007

Declaration

I declare that this thesis is my own, unaided work. It is being submitted for the degree of **Doctor of Philosophy in Electrical Engineering** in the University of Cape Town. It has not been submitted before for any degree or examination in any other university.

Signature of Author Signed by candidate .....

Cape Town
Rondebosch, February 10, 2020

Executive Summary

Modern design of Software-Defined Radio (SDR) applications is based on Field Programmable Gate Arrays (FPGA) due to their ability to be configured into solution architectures that are well suited to domain-specific problems while achieving the best trade-off between performance, power, area, and flexibility. FPGAs are well known for rich computational resources, which traditionally include logic, register, and routing resources. The increased technological advances have seen FPGAs incorporating more complex components that comprise sophisticated memory blocks, Digital Signal Processing (DSP) blocks, and high-speed interfacing to Gigabit Ethernet (GbE) and Peripheral Component Interconnect Express (PCIe) bus. Gateware for programming FPGAs is described at a low-level of design abstraction using Register Transfer Language (RTL), typically using either VHSIC-HDL (VHDL) or Verilog code. In practice, the low-level description languages have a very steep learning curve, provide low productivity for hardware designers and lack readily available open-source library support for fundamental designs, and consequently limit the design to only hardware experts. These limitations have led to the adoption of High-Level Synthesis (HLS) tools that raise design abstraction using syntax, semantics, and software development notations that are well-known to most software developers. However, while HLS has made programming of FPGAs more accessible and can increase the productivity of design, they are still not widely adopted in the design community due to the low-level skills that are still required to produce efficient designs. Additionally, the resultant RTL code from HLS tools is often difficult to decipher, modify and optimize due to the functionality and micro-architecture that are coupled together in a single High-Level Language (HLL). In order to alleviate

these problems, Domain-Specific Languages (DSL) have been introduced to capture algorithms at a high level of abstraction with more expressive power and providing domain-specific optimizations that factor in new transformations and the trade-off between resource utilization and system performance. The problem of existing DSLs is that they are designed around imperative languages with an instruction sequence that does not match the hardware structure and intrinsics, leading to hardware designs with system properties that are unconformable to the high-level specifications and constraints.

The aim of this thesis is, therefore, to design and implement an intermediate-level framework namely SdrLift for use in high-level rapid prototyping of SDR applications that are based on an FPGA. The SdrLift input is a HLL developed using functional language constructs and design patterns that specify the structural behavior of the application design. The functionality of the SdrLift language is two-fold, first, it can be used directly by a designer to develop the SDR applications, secondly, it can be used as the Intermediate Representation (IR) step that is generated by a higher-level language or a DSL. The SdrLift compiler uses the dataflow graph as an IR to structurally represent the accelerator micro-architecture in which the components correspond to the fine-level and coarse-level Hardware blocks (HW Block) which are either auto-synthesized or integrated from existing reusable Intellectual Property (IP) core libraries. Another IR is in the form of a dataflow model and it is used for composition and global interconnection of the HW Blocks while making efficient interfacing decisions in an attempt to satisfy speed and resource usage objectives. Moreover, the dataflow model provides rules and properties that will be used to provide a theoretical framework that formally analyzes the characteristics of SDR applications (i.e. the throughput, sample rate, latency, and buffer size among other factors). Using both the directed graph flow (DFG) and the dataflow model in the SdrLift compiler provides two benefits: an abstraction of the microarchitecture from the high-level algorithm specifications and also decoupling of the microarchitecture from the low-level RTL implementation. Following the IR creation and model analyses is the VHDL code generation which employs the low-level optimizations that ensure optimal hardware design results. The code generation process per-

forms analysis to ensure the resultant hardware system conforms to the high-level design specifications and constraints. SdrLift is evaluated by developing representative SDR case studies, in which the VHDL code for eight different SDR applications is generated. The experimental results show that SdrLift achieves the desired performance and flexibility, while also conserving the hardware resources utilized.

Acknowledgements

I give a multitude of gratitude to God, the Almighty Father the Creator of Heaven and Earth, and of all Things, to His Only Begotten Son Jesus Christ Our Saviour Born of the Virgin Mary, and to the Holy Ghost, for giving me Faith, Hope and Charity.

Thanks to my project supervisor Dr. Simon Winberg for his unconditional support, encouragement, and for creating an excellent work environment for the success of this project. His extensive knowledge and experience has helped me to explore ideas in this research project. I am grateful to my Co-Supervisor Professor Michael Inggs for his guidance and priceless advice that kept this project ticking.

Many thanks to Dr. Lerato Mohapi for his assistance and support in the early phases of the research project. I am very grateful to Dr. Makhamisa Senekane, my former senior schoolmate at Christ The King High School for his constructive feedback on the review of this research project.

Special thanks to my late grandfather Thato Lawrence “Leronti” Tsoeunyane for his parental love and guidance. May his soul, and the souls of all the faithful departed, through the mercy of God, rest in peace.

I thank my wife ’Mathato Alphoncina Tsoeunyane for her true LOVE and for being there for me through thick and thin. And without these two lovely angels, my daughter Puleng Tsoeunyane and my son Thato Lawrence “Leronti Junior” Tsoeunyane, I wouldn’t have finished this project.

This research was supported by the South African Radio Astronomy Observatory (SARAO), which is a facility of the National Research Foundation, an agency of the Department of Science and Technology. The support and financial assistance provided by SARAO are highly acknowledged.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

16-QAM Quadrature Amplitude Modulation 133, 134, 136

ADC Analog-to-Digital Converter 3, 19, 21–23, 28, 29, 153–155

AOCL Altera SDK for OpenCL 40

API Application Programming Interface 24

ASIC Application-Specific Integrated Circuit 4, 18, 19, 23, 25, 26

BEE Berkeley Emulation Engine 27

BF I Butterfly I 91

BF II Butterfly II 91, 92

BORPH Berkeley Operating system for Re-Programmable Hardware 28, 29

BPDF Boolean Parametric Dataflow 37

BPF Bandpass Filter 20, 21, 153

BRAM Block Random Access Memory 26

CAL Cal Actor language 47

CASPER Collaboration for Astronomy Signal Processing and Electronics Research 27, 28, 42

CFIR Compensation FIR filter xv, 138, 139, 157–159

CIC Cascaded Integrator Comb xv, 138, 139, 154, 156, 157

CLB Configurable Logic Block 26

COM Computer-On-Module 28

CP-I Cyclic-prefix Insert 81, 133, 135

CP-R Cyclic-prefix Remove 81, 134, 136

CPU Central Processing Unit 23, 28

CSDF Cyclo-Static Dataflow 37–39, 44

CSE Sub-Expression Elimination 78, 164

DAC Digital-to-Analog Converter 3, 20–23, 29, 140, 153

DC Direct Current 21, 153, 156, 157

DCE Dead-Code Elimination 78, 164

DDC Digital Down Conversion 22, 131

DDR3 Double Data Rate 3 31

DFG directed graph flow iv, xiii, xv, 7, 15, 16, 75, 78, 81–85, 89–95, 102, 104, 106, 107, 109, 162, 166, 167, 169, 173

DFT Discrete Fourier Transform 23

DHDL Delite Hardware Definition Language 42

DNL Differential Non-Linearity 21

DPN Dataflow Process Network 37, 38

DSE Design Space Exploration 165, 166

DSL Domain-Specific Language iv, 8, 10, 13, 36–38, 42, 45, 50, 51, 160, 162

DSP Digital Signal Processor 4, 18

DSP Digital Signal Processing iii, 3, 4, 7, 11–13, 19, 22, 23, 25, 26, 33, 35, 37, 39, 42, 43, 51, 70, 71, 82, 94, 129

EDA Electronic Design Automation 32, 35

EDK Embedded Development Kit 31, 42

ENOB Effective Number of Bits 21

FDF Frequency Domain Filtering 22

FFT Fast Fourier Transform xiii, xvi, 22, 23, 90–93, 95, 107, 109, 110, 134, 136

FG Firing Gap 120–122

FHDL Fragmented Hardware Description Language 43

FIFO Finite In First Out 37, 54–58, 64, 65, 75, 80, 85, 96, 98, 101, 102, 111, 113, 114, 116, 118, 124, 126, 132, 141, 152, 155

FIR Finite Impulse Response xiii, xv, xviii, xxiv, 23, 87–90, 106–108, 138

FM Frequency Modulation xv, 16, 131, 153–159

FM-DDC Frequency Modulation Digital Down Converter xiv, 131, 132, 139, 140, 145, 148, 154

FMC FPGA Mezzanine Card 29, 31

FPGA Field Programmable Gate Array iii, iv, xv, 3–6, 8, 10, 12–19, 23, 25–33, 36, 38, 40–47, 50, 51, 53, 70, 80, 98, 102, 108, 109, 111, 119, 126, 127, 140, 141, 144, 148, 150, 153, 154, 157, 158, 160, 161, 163, 164, 166

FSM Finite-State Machine xiii, xiv, 16, 37, 109, 112, 113, 115–129, 141–144, 163

GbE Gigabit Ethernet iii, 4, 26, 30, 154, 157

GMII Gigabit Media-Independent Interface 158

GPP General-Purpose Processor 4, 18, 19, 23, 25, 164

GPU Graphics Processing Unit 4, 18, 19, 23, 25

GSM Global System for Mobile Communications 131

GSM-DDC Global System for Mobile Communications Modulation Digital Down Converter xiv, 131, 132, 138–140, 145, 148

HDL Hardware Description Language 4, 7, 8, 13, 25, 30, 31, 42–44, 75, 80

HLL High-Level Language iii, iv, 4, 12, 33, 34, 40

HLS High-Level Synthesis iii, 3–7, 10, 12, 15, 17, 33–38, 40, 41, 44–48, 50–52, 80, 160, 162

HMC Hybrid Memory Cube 28

HSDF Homogeneous Synchronous Dataflow 44

HW Block Hardware block iv, 6, 8, 11, 15, 16, 26, 30, 72, 74–76, 80, 86, 96, 100, 101, 111–114, 116, 118, 129, 132, 161, 162

I/O Input/Output xv, 4, 24, 26, 28, 31–33, 153, 155, 158

I/Q In-phase and Quadrature 133, 134, 136, 140

iBOB Interconnect Break-out Board 28

IEEE Institute of Electrical and Electronic Engineers xiv, 18, 131–138, 140, 141, 145, 148, 150, 176

IF Intermediate Frequency 19–22

IFFT Inverse Fast Fourier Transform 23, 91, 133, 135

IG Iteration Gap 120–122, 126

IIR Infinite Impulse Response 23

IL Iteration Latency xxvii, xxix, 58–61, 63, 64, 118

ILP Integer Linear Programming 58

INL Integral Non-Linearity 21

IP Intellectual Property iv, xv, 5–8, 10–14, 16, 26, 27, 33, 38, 39, 44–48, 50, 70, 71, 75, 78, 80, 82, 86, 90, 94, 99, 101, 105, 108, 111, 113, 127, 129, 131, 132, 153–155, 157, 158, 161, 162, 164

IR Intermediate Representation iv, 8, 10, 16, 41, 51, 72, 78, 80–82, 96, 100, 101, 130, 161

IR_{CG} Coarse Grained Intermediate Representation 8, 78, 80–83, 85–88, 94, 96, 101, 104–106, 129

IR_{DF} Dataflow Intermediate Representation 8, 79–82, 94, 98, 99, 101, 108

IR_{FG} Fine Grained Intermediate Representation 8, 78, 80–83, 85, 101–104, 106, 129

ISE Integrated Synthesis Environment 10, 32, 80, 119, 141, 142, 144, 148, 150

JVM Java Virtual Machine 72, 145, 146

KPN Kahn Process Network 37, 38

KSPS Kilo Samples Per Second 138, 139, 157

LHS Left Hand Side 104

LLVM Low Level Virtual Machine 41

LMS Lightweight Modular Staging 8, 42

LNA Low Noise Amplifier 21, 153

LO Local Oscillator 19–21, 155

LUT Lookup Table 8, 26, 32, 85, 112, 124–126, 130, 150, 151

LVDS Low-voltage Differential Signaling 153

MAC Media Access Control 30, 157, 158

Mbps Mega Bits Per Second 140

MDIO Management Data Input/Output 158

MIMO Multiple Input Multiple Output xiv, 131, 132, 136–138, 140, 145, 148

min Minutes xvi, 146, 148

MoC Model of Computation 37–39, 51, 54, 96

ms Milliseconds xvi, 145, 146, 148

MSB Most Significant Bit 74

MSPS Mega Samples Per Second 138, 139, 153, 155, 156

MSSGE Matlab/Simulink/System Generator/EDK 42

MUX Multiplexer 26

NCO Numerically Controlled Oscillator 138, 154–157

OCP Open Core Protocol 46

OFDM Orthogonal Frequency Division Multiplexing 45, 131, 132, 134–136, 150

OFDM-RX Orthogonal Frequency Division Multiplexing Receiver xiv, 131, 132, 134, 136–138, 140, 141, 145, 148

OFDM-TX Orthogonal Frequency Division Multiplexing Transmitter xiv, 131–133, 135–137, 140, 141, 145, 148, 176

OS Operating System 146

P/S Parallel to Serial 137

PA Power Amplifier 20, 21

PASS Periodic Admissible Schedules 57, 58

PC Personal Computer 46, 153, 157, 158, 164

PCIe Peripheral Component Interconnect Express iii, 4, 28

PCSDF Parameterized Cyclo-Static Dataflow 39, 44

PD Path Delay 86–88

PFIR Programmable FIR filter 138

PHY Physical Layer xv, 3, 30, 157, 158

PLL Phase Locked Loop 26, 153

PPL Stanford University's Pervasive Parallelism Laboratory 42

PSDF Parameterized Synchronous Dataflow 37, 39, 44

QoR Quality of Result 130

R-2 Radix-2 90

R-2^2 Radix-2^2 xiii, 90–93, 95, 107, 109, 110

R-4 Radix-4 90

RAM Random Access Memory 102

RBDS Radio Broadcast Data System 158

REG Register 81, 82, 86, 87

RF Radio Frequency 18–23, 31, 153, 156

RHINO Reconfigurable Hardware Interface for ComputatioN and RadiO xii, xv, 29, 153, 154, 157, 158

RHS Right Hand Side 104

ROACH Reconfigurable Open Architecture Computing Hardware 28, 29

ROM Read-Only Memory 92–94, 171

RTL Register Transfer Language iii, iv, 4, 31–35, 41, 42, 46, 50, 51, 72, 130, 131, 144

S/P Serial to Parallel 136, 137

SARAO South African Radio Astronomy Observatory vii, 28

SDF Single-Path Delay-Feedback xiii, xvi, 90–93, 95, 107, 109, 110

SDF Synchronous Dataflow 37–39, 44, 45, 47, 54, 56, 96

SDF-AP Static Dataflow with Access Patterns xii–xiv, xxvii, xxix, 7, 8, 11, 13, 15, 16, 39, 45, 50, 52–58, 60–62, 64, 66, 68, 69, 71, 75, 79–82, 86, 94, 96, 98–102, 108–113, 123, 127–129, 132, 134–136, 141, 142, 144–146, 154, 161–164, 166

SDR Software-Defined Radio iii–v, xiv, xv, 2–8, 10–23, 25–27, 29–31, 33, 37, 38, 43–47, 50–53, 70–72, 75, 78, 81, 86, 98, 100, 101, 109, 129–131, 139, 141–146, 148, 150–152, 158–165

SFDR Spurious Free Dynamic Range 22

SG Startup Gap 120, 121

SINAD Signal-to-Noise and Distortion 21

SKARAB Square Kilometre Array Reconfigurable Application Board 28

SNR Signal-to-Noise Ratio 21

SoC System on Chip 31, 44, 46

SPC Samples Per Cycle 118, 140, 153

SPMD Single Program Multiple Data 41

SPS Samples Per Second 140

SRC Sample Rate Conversion 22

TCL Tool Command Language 80

TCP Transmission Control Protocol 30

THD Total Harmonic Distortion 21

THD+N Total Harmonic Distortion plus Noise 20

TM A topology matrix 57, 58

UDP User Datagram Protocol 30, 157

USB Universal Serial Bus 4, 31

USRP Universal Software Radio Peripheral 30

VCI Virtual Component Interface 46

VHDL VHSIC-HDL iii–v, xiv, 4, 8, 10, 12, 13, 15, 16, 24–26, 31, 35, 42–44, 46, 48, 74, 80, 83, 84, 86, 98, 99, 101–103, 105–108, 110, 111, 113, 114, 118, 122, 124, 129, 131, 132, 141, 144, 145, 154, 158, 161, 162, 165

VLSI Very Large Scale Integration 113

WARP Wireless Open-Access Research Platform 30, 31

WCET Worst-Case Execution Time 38

WIF Wireless Innovation Forum 18

WOLA Weight Overlap Add 22

ZP-I Zeropad Insert 81, 133, 135

ZP-R Zeropad Removal 81, 134, 136

Symbols

a An SDF-AP actor. xxvii–xxix, 54, 55, 58, 60, 62, 63

$\mathcal{A}$ A set of actors in $\mathcal{G}$. 54, 55, 58, 62

AP Access pattern which can either be CP or PP. xxvii, 58, 81, 86, 88

$\mathbb{B}^{ET}$ A set of sequences of binary numbers with length ET. 55

c An SDF-AP channel. xxvii, xxix, 55, 59, 60, 62, 124

θ A vector containing the buffer sizes at time t over one iteration period $\forall t \in \{0...IterationLatency(IL)\}$. 63, 64

$\mathcal{C}$ A set of channels in $\mathcal{G}$. 54, 55, 60

CP A consumption pattern for input port q. xxvii, 55, 58, 81, 88, 121, 124

CR A consumption rate denoting the number of tokens consumed from channel c. 55, 62, 63, 81, 87, 88

dly A delay that denotes initial number of tokens in channel c. 55, 63

EP Execution pattern which is sequence of binary elements of an access pattern AP on the port of actor a where it is active and idle for a duration of IL. 58, 59, 63, 114, 116, 118

ET Execution time of an actor a in clock cycles. xxvii, 54–56, 60, 62, 63, 66, 67, 81, 120–122, 124, 126, 133, 138

M A finite state machine. xiii, xiv, xxviii, 115–126

I The finite input space for M. 115

O The finite output space for M. 115

λ An output function for M. 115

s A state for M. 115–117, 121, 122, 124

S A set of finite states for M. 115

s_0 An initial state for M. 115–118, 121, 122

δ The next state or transition function for M. 115

$\tau_{\mathcal{G}}$ A throughput constraint for $\mathcal{G}$. 60, 62

II Initiation Interval defined as as the minimum interval between two successive firings of actor a. 54

$INPUT$ An input direction of a port p^t. 81

IN A set of input ports for an actor a. 54, 55

T Iteration period or schedule initiation interval. 58, 61, 62, 117, 120

n A node that associates with the template. xxix, 81

$OUTPUT$ An output direction of a port p^t. 81

OUT A set of output ports for an actor a. 54, 55

σ A 1-periodic schedule of an actor a. 58, 60, 62, 63, 120

i An interface. 81

I^t A set of interfaces applicable to ports. 81

$\mathcal{P}$ A a set of ports in $\mathcal{G}$. 54

PP A production pattern for output port p. xxvii, 55, 58, 81, 88

PR A production rate denoting the number of tokens produced on channel c. 55, 62, 81, 87, 88

RV A repetition vector denoting the number of firings for an actor a. 55–58, 60, 62, 120–122, 124, 126

μ Scheduling period. 58, 60, 61, 63, 120

$\mathcal{G}$ A graph that represents SDF-AP model. xxvii–xxix, 54, 60

v A sink actor for channel c. xxix, 55, 63, 64, 120–122, 124

$\mathcal{A}_{snk}$ A set of sink actors for $\mathcal{G}$. 55, 60, 62, 64

$\mathcal{C}_{snk}$ A set of sink channels for $\mathcal{G}$. 55, 62

q A input port for v in channel c. xxvii, 55, 56, 59

u A source actor for channel c. xxix, 55, 62–64

$\mathcal{A}_{src}$ A set of source actors for $\mathcal{G}$. 54, 55, 60, 63, 64

$\mathcal{C}_{src}$ A set of source channels for $\mathcal{G}$. 55

p A output port for u in channel c. xxviii, 55, 56, 59

TC A token count is the sequence of length IL which represents the total number of tokens that are produced (resp. consumed) (i.e. $TC_{i,p}$ (resp. $TC_{i,q}$)) to (resp. from) the channel up to the i-th clock cycle. 59, 63, 64

t A template. 81, 88

P_i^t A set of template input ports. 81

t_n A template that belongs to node n. 81

P_o^t A set of template input ports. 81

p^t A template port. xxviii, 81

d A template port direction. 81

P^t A set of template ports. 81

$\mathcal{T}$ A set of templates. 81

τ A throughput. 60, 62, 64

Chapter 1

Introduction

The ever increasing popularity and evolution of wireless communication technologies and standards are changing the manner in which wireless services and applications are used [8]. The demand and usage of these services by users are growing rapidly and is constantly pushing designs to their limits. Wireless devices are becoming more common and users are demanding the convergence of multiple services and technologies in a single device [9]. These lead to potential challenges in areas of equipment design, wireless service provision, security, and regulation [10]. The apparent outgrowth of hardware capacity and complexity over the hardware design productivity is known as the hardware design-productivity gap [11] - which is to say technology advancements have grown faster than the capabilities of tools and design methodologies to support the complexity of these designs. Most recently, the emergence of big data and streaming applications in the field of telecommunications have significantly brought about a need to design systems that can process big volumes of data at very high speed with minimal energy consumption needs. Streaming applications rely on real-time systems to operate on continuous streams of data where stringent deadlines must be met leading to high demands for increased computing power.

Being intrinsically streaming in nature, SDR is no exemption to applications that are highly compute-intensive [12]. Indeed SDR relies on configurable technologies which are a solution to today's increasing user needs for wireless services and ap-

plications. These types of technologies are upgradable, reconfigurable and adaptable to changes in technology standards and need [13]. SDR is very instrumental in developing wireless standards due to its flexibility and programmability. Such characteristics require the underlying hardware architecture to process tasks in a digital domain at very high-speed with reduced power consumption. FPGAs strike an effective balance between performance and flexibility, essentially trading allowing sacrifices in performance (compared to more rigid application-specific platforms) for significantly greater flexibility. The downside is the difficulty of developing HLS code for FPGAs at the necessary low-level of design abstraction: this issue has become a major design bottleneck especially for domain experts with limited knowledge of hardware design and programming. Consequently, there is a need for "raising the level of abstraction" [14] to allow designers to focus on a high-level of design aspects, and thereby facilitate the exploration of potential novel applications and design solutions, without the distractions of having to deal with the complex low-level details.

1.1 Software-defined radio

SDR system implements some or all of its PHY functionality in software [15]. This makes it more flexible than the rigid traditional radio architecture that relies on analog hardware components to perform radio signal processing functions. A typical SDR is a form of a radio transceiver where a receiver consists of an analog front-end with an antenna and signal conditioning circuitry that is connected to an Analog-to-Digital Converter (ADC) followed by a DSP system to extract a signal of interest. The SDR transmitter, on the other hand, consists of a digital signal processing that feeds information to the Digital-to-Analog Converter (DAC) which connects to the analog front-end that has the analog circuitry and the antenna. Nowadays, many SDR systems need to support a diverse range of adjustable operations and operating modes, such as support for multiple bands and carriers, multiple standards, and enabling a variety of services [15, 16].

High-performance SDR platforms allow for the implementation of the diversity

of operations through the use of multiple types of parallel processing resources including FPGA, Digital Signal Processor (DSP), General-Purpose Processor (GPP), Application-Specific Integrated Circuit (ASIC) and Graphics Processing Unit (GPU) [12]. Although ASICs are faster than the rest of the other devices and more efficient, they are rarely used in these applications, particularly in the case of experimental SDR prototyped systems, for which low-volume bespoke solutions are often used due to their complexity and need for flexibility and customizability [17]. FPGAs have become integral components of many SDR platforms, with their configurable structures allowing processing topologies of varying degrees of parallelism with highly changeable communication schemes. Modern components found in FPGA do not only include fine-grained elements such as logic, register, and routing elements, but they also include coarse-grained integrated blocks [18]. These integrated blocks may include memory blocks, DSP blocks, and physical I/O interface (e.g. GbE, PCIe, Universal Serial Bus (USB), etc). The challenges that arise comprise the integration of these parts, keeping track of how the different parts have been connected, and managing the timing so that they all meet the deadlines, all which are typically defined using low-level Hardware Description Languages (HDL). The description of these applications typically comprises a significant portion of RTL design work using either VHDL or Verilog, but working at this low-level of design abstraction tends to need a thorough understanding of the physical characteristics of the processing resources which make this type of work largely restricted to hardware experts or lead to the developers engaging in lengthy learning curves to acquire the necessary low-level details of the processing resources [19].

As a response to the difficulty of developing hardware applications at the low-level of design abstraction, significant time and effort have in the past three decades been devoted to developing HLLs and tools that are generally known as HLS [14]. HLS tools target non-hardware experts and aim to use familiar HLLs to describe algorithms and to automatically translates them into a gateware. With the hardware capacity and complexity increasing at a more rapid rate than the hardware design productivity [11], advanced and generic HLS techniques have also been implemented. While many of HLS tools have been used for wide

range of applications such as image and video signal processing, security, defense systems, etc some of the most prominent HLS tools in SDR have been Matlab / Simulink [20], Vivado HLS [21], LabViewer [22], and GNU radio [23].

1.2 Problem Statement

The existing tools suffer several limitations which make it difficult for the system designers to design and create FPGA-based SDR applications, particularly.

1.2.1 Complexity of Design

The development of SDR systems using FPGAs compels designers to design new IP cores as well as reusing pre-existing IP cores to meet time-to-market and design efficiency requirements. However, the low-level development difficulties associated with FPGAs hinder productivity, even when the designer is experienced with hardware design. These low-level difficulties include non-standard interfacing methods, component communication and synchronization challenges, complicated timing constraints and processing blocks that need to be customized through time-consuming design tweaks. HLS tools try to alleviate most of these problems by adding an abstraction layer to the development so that designers can spend time on high-level properties that matter most rather than working on the complex low-level design. While this approach has proven to speed up the development time in some applications, the resulting hardware systems often suffer from inefficiency and poor quality of performance. Furthermore, most of these HLS tools use imperative languages (such as C, C++, SystemC, etc) which are inherently low-level and sequential as a result there still exists a gap between a general-purpose, Turing-complete HLS and the resultant hardware design that can efficiently run on an FPGA [24, 25].

1.2.2 Lack of Design Constraints Specification

In general, the existing HLS tools do not allow critical design constraints such as throughput, latency and buffer size requirements to be captured at the high-level of design abstraction [26]. Neither do they consider the physical characteristics of the selected FPGA target in a way that allows capturing critical application timing constraints nor provide mechanisms to automatically adjust integrating logic to maintain these constraints. Consequently, developing new IP cores and reusing existing ones tends to involve many manual and time-consuming design tasks, often at a low level of abstraction. Moreover, these tasks may need to be repeated during the development process.

1.2.3 Lack of support for IP integration

While there exist alternatives for prototyping FPGA-based SDR applications using high-level synthesis tools [12] and overlay architecture frameworks [27], these solutions generally emphasize flexibility and productivity, rather than the performance of the resultant hardware design. To achieve optimal design results, prototyping SDR systems with FPGAs still forces designers to reuse existing hardware processing blocks which are also known as IP cores or simply HW Blocks. A number of these IP cores are provided by the mainstream vendors and are also available as an open-source community contributed libraries. In the practical context of SDR, it is often difficult and tedious to integrate these IP cores into a design, as this usually requires detailed knowledge of the cores [28]. Further challenges that developers encounter include developing designs that provide sufficiently high-speed data exchange, synchronization and correct implementation of communication protocols between the components, interface synthesis to resolve protocol mismatches, the difficulty of component composition [29] - all of these potentially lengthy development activities usually depend on specialized hardware design skills.

1.2.4 Lack of Correctness Verification

One of the common flaws in HLS tools is that they do not check whether the results are generated correctly by the synthesis flow and rather rely on the traditional *correct-by-design* design approach which is error-prone. While these types of HLS tools may result in hardware designs that are correct and conform to the high-level description, they fail to formally prove that the generated hardware design faithfully captures the high-level descriptions hence the design correctness is not always guaranteed. SdrLift therefore addresses the problem of correctness verification in addressed in Section 5.4.

1.3 Contributions of this work

In order to address these challenges, in particular, to move away from the need for low-level implementation know-how and to reduce the time in either describing the new DSP cores or integrating the existing IP cores, SdrLift intermediate compiler framework is presented. This framework boosts productivity by using a domain-specific intermediate-level language that is embedded in Scala to describe accelerators' new cores in a structural manner. It further allows reuse of existing, well-optimized HDL processing codes and by automating the integration of these modules in a best-effort attempt to satisfy performance requirements (the best effort depending on the provided optimization parameters and optimization parameters). The behaviour of custom DSP cores is described as the hierarchical and modular connection of logic components using a DFG. This design approach has proven to be effective in designing DSPs in which explicit parallelism needs to be exposed [30, 24]. The newly defined cores together with pre-defined IP cores are composed into a complete SDR application using a dataflow model of computation known as a SDF-AP model [26]. The SDF-AP model directs interfacing decisions (e.g. buffering sizes) in an attempt to satisfy speed and resource usage objectives. The overall system performance is constrained by throughput that specified in the SDR program description.

In order to automate hardware generation methods discussed in this work, a compiler framework as depicted in Figure 1.1 is developed and it serves as a low-level IR to generate efficient hardware from a DSL for SDR or it can be used directly by domain experts to develop SDR applications. This framework, namely SdrLift, leverages Scala's functional language constructs for embedding DSLs. The input to the SdrLift is the DSL for specifying the behavior of the SDR application. For instance, OptiSDR [31] which facilitates development of SDR on heterogeneous architectures can be used. In the case of OptiSDR, a Lightweight Modular Staging (LMS) and Delite are required to translate the input programs into staged IR which represents SdrLift language. The first step in SdrLift compilation flow is a light-weight intermediate language namely SdrLift language that accepts the descriptions of applications before undergoing a series of compiler intermediary steps. These steps include transformation into three compiler IRs namely Fine Grained Intermediate Representation (IR_{FG}), Coarse Grained Intermediate Representation (IR_{CG}) and Dataflow Intermediate Representation (IR_{DF}). First, the application description is converted into a IR_{FG} that represents a micro-architectural graph of fine hardware elements such as logic, arithmetic, and Lookup Table (LUT). IR_{FG} represents basic components which become nodes that build the IR_{CG}. IR_{CG} results in the larger HW Blocks which are referred to as modules in SdrLift and they are an equivalent of the IP cores. In both IR_{FG} and IR_{CG}, the HDL design elements found in the primitive libraries of the target FPGA are integrated as templates to create more efficient hardware system.

Moreover, the modules that are created in IR_{CG} are integrated alongside the existing IP cores in IR_{DF} using the SDF-AP model. This SDF-AP model is implemented in *scala-graph* (i.e. graph library for Scala) [32] library that is incorporated in a compiler. Before HDL generation starts, the SDF-AP model undergoes analysis and scheduling which are key to validating the system and in determining the system properties; these properties include buffer size, latency and component compatibility from given throughput constraint. The HDL generator then generates the VHDL from the SDF-AP model using the *vMagic* library [33]. vMagic library is used by SdrLift compiler framework to read the VHDL

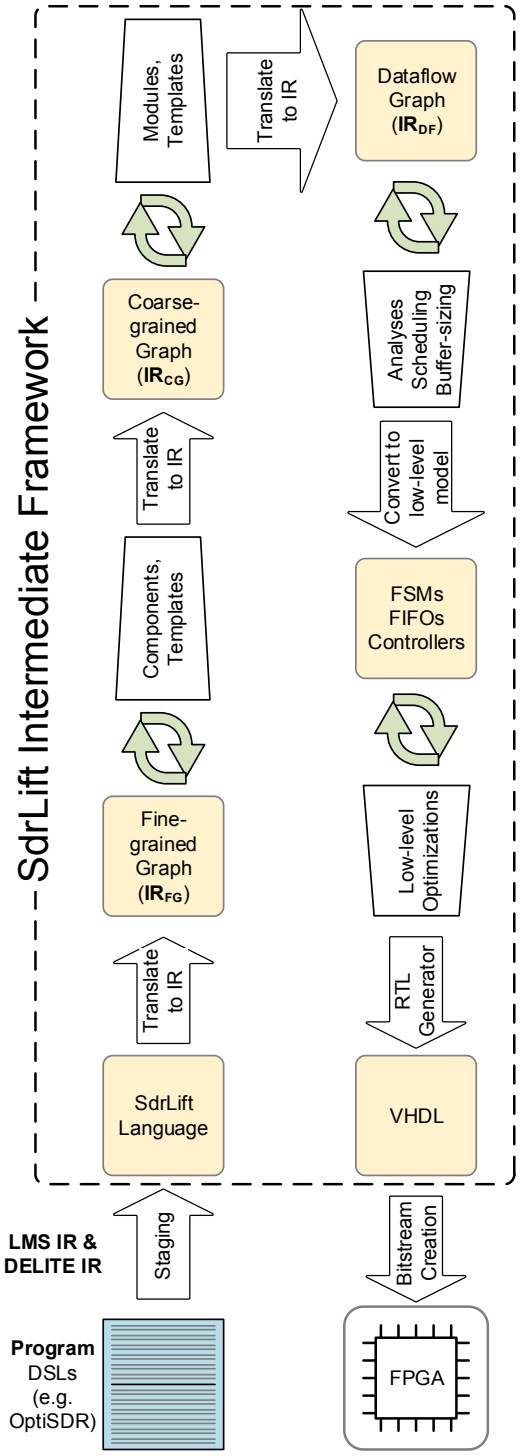


Figure 1.1: Overview of SdrLift Intermediate Compiler Framework.

code for newly synthesized and existing IP cores, to stitch these cores together in VHDL, and to write out the final top-level design in VHDL. Moreover, the optimizations are applied during code generation to enable efficient hardware design results. After code generation, the framework invokes the compilation functions of the *Xilinx Integrated Synthesis Environment (ISE) 14.7* tool-chain. These compilation functions include synthesis, build, map, place and route and finally the binary file creation to target the FPGA device.

1.4 Research Hypothesis

The hypothesis in this thesis is centered around the productivity and performance of the system under development using SdrLift. The productivity refers to shorter time-to-market and reduced application development effort by the user. The user can either be the SDR developer who uses SdrLift for rapid prototyping of SDR applications or a compiler developer who uses the SdrLift as the compiler IR step by integrating it in a HLS/DSL compiler flow. Furthermore, the performance comprises throughput (over millions of samples per second) and latency of the application and this is achieved through efficient hardware design that conserves hardware resources and power consumption. The research hypothesis for this thesis is, therefore:

It is possible to develop an intermediate framework that uses high-level structural and functional language constructs to generate the efficient FPGA code for prototyping SDR applications under the user-specified throughput constraint.

The research questions associated with this thesis relate to the outcomes of previous works on the development of high-level prototyping tools for SDR and compiler automation standards which are of utmost importance to developing SdrLift.

- *What are the traits of the intermediate-language to describe the FPGA-based SDR applications at the high-level of design abstraction?* It is im-

portant to develop a framework with an entry-point that incorporates a language that can capture the SDR specifications in a manner that will reduce the development effort while ensuring the quality hardware results. In addition, the intermediate language should intuitively describe the behavior of the SDR application such that the algorithmic meaning is retained all through to the generated hardware system. The answer to this question will be the development of intermediate language constructs that will structurally describe applications in the domain of SDR using data and topological design patterns that raise the expressive power.

- *What is an appropriate design approach for developing hardware cores that will result in SDR applications that conform to the specified throughput constraint?* In order to build an application that is constrained by a user-specified throughput, the precise times at which the building blocks consume and produce data samples should be known. In the case of existing HW Blocks, the timing information is typically obtained through low-level simulation and reading behavioral operation from the data-sheets. However, building an intelligent compiler that automates the computation of data consumption and production patterns along with timing information for a HW Block is very complex. The solution to automate the computation of HW Block access patterns is proposed in this thesis using a template-based design approach for the synthesis of DSP IP cores in prototyping SDR.

- *What dataflow model is needed to effectively analyse and compose the developed HW Blocks for implementation of a system that conforms to a throughput constraint?* The computation of data access patterns for HW Blocks is vital for gateware generation for applications that are constrained by a throughput. More importantly, the dataflow model is required to determine the interfacing decisions and buffer size requirements in composing HW Blocks into pipelined hardware architectures. In this thesis, an SDF-AP model is adopted for integration and composition of newly developed HW Blocks and existing IP cores to automate the gateware generation for SDR applications.

- *Can the compiler flow be exploited to generate the optimal hardware design?* The main challenge with conventional HLS tools is that they serve as a great productivity tool to users who are highly equipped with hardware design skillset and deep understanding of the compiler intrinsics. It is therefore imperative to design a high-level prototyping framework for ease of use by a domain user while generating low-level gateware using automated optimization techniques that help to obtain high-quality results. This project aims to keep automatic compiler optimizations oblivious to the domain user and extend this by ensuring the correctness of the designed systems and verify that the generated hardware system faithfully conforms to the high-level design constraints.

1.4.1 Research Objectives

The main goal of this thesis is the implementation of an intermediate-level toolflow, namely SdrLift, to be used for prototyping of FPGA-based SDR applications. The SdrLift entry point is a HLL designed around functional programming language constructs and notations which capture algorithmic specifications of the SDR domain. The output of SdrLift is an efficient, synthesizable VHDL code ready to be deployed on the FPGA hardware. SdrLift relies upon the exploitation of commonalities of optimized SDR-based DSP algorithms as implemented on the FPGA and inclusion of efficient IP libraries during the high-level synthesis of SDR accelerators. Therefore the following main objectives were set for this project:

1. To develop a high-level methodology that allows the generation of VHDL code for SDR applications in-order to separate the high-level programming model from low-level system implementation.

2. To investigate strategies and related algorithms that enable analysis, scheduling and computation of buffer size and latency model graphs for systems as well as providing additional semantics that facilitate the description of model graphs of systems.

3. To automate low-level gateware generation from high-level specifications.

4. To demonstrate the capability of SdrLift by generating multiple DSP IP cores in VHDL code.

5. To evaluate the hardware synthesis approach using typical SDR applications to generate synthesizable code that is human-readable to allows manual alterations during simulation and synthesis optimization.

1.5 List of Publications

The work presented in this thesis was subject to different publications that have contributed entirely to its completion. The first paper [2] presents the design and implementation of an open-source library of parameterizable and reusable HDL IP cores designed around the development of FPGA-based SDR applications. Furthermore, the paper demonstrated the importance of these IP cores for use in the compiler design process that aims to automate HDL generation from high-level of design abstraction using a DSL. These IP cores prove to be instrumental in this thesis in that they are integrated into SdrLift as macros to generate a more optimal code of tried and tested HDL designs. The second paper [28] is centered around a high-level tool-flow that automates the stitching together of the existing IP cores using SDF-AP model for prototyping SDR applications. The model will prove to be highly instrumental in SdrLift in connecting both the pre-designed IP cores and the newly compiler-created hardware kernels to maintain particular application timing constraints.

Furthermore, the third paper [34] is an extension to the second paper [28] and in addition to reporting on the automated and behavioral integration of dedicated IP cores for rapid prototyping of multiple SDR applications using SDF-AP model, it further shows how the model components are mapped onto the low-level model of hardware by efficiently applying low-level hardware generation optimizations. This paper also presents the formal analysis technique that guarantees the correctness of the generated low-level solutions. Lastly, the paper [35]

presents SdrLift which consolidates the work in the first three papers and further automates hardware synthesis through both algorithmic-level descriptions and integration of pre-defined IP cores. The paper also introduces the language for SdrLift and the formerly presents an intermediate compiler for SdrLift which is used to accept the high-level SDR application specifications and generates the low-level hardware designs target towards execution on the FPGA platform.

The published research outputs from this thesis are listed below, these have been divided between peer-reviewed journal outputs and conference papers that have been published in proceedings.

1.5.1 Journal Papers

1. L. Tsoeunyane, S. Winberg, and M. Inggs, "Automatic configurable hardware code generation for software-defined radios," *Computers*, vol. 7, no. 4, 2018.

2. L. Tsoeunyane, S. Winberg, and M. Inggs, "Software-defined radio FPGA cores: Building towards a domain-specific language," *International Journal of Reconfigurable Computing*, vol. 2017, 2017.

1.5.2 Conference Papers

1. L. Tsoeunyane, S. Winberg, and M. Inggs, "Sdrlift: An intermediate-level framework for synthesis of software-defined radio accelerators," in *2019 IEEE 10th International Conference on Mechanical and Intelligent Manufacturing Technologies (ICMIMT)*, pp. 166–173, Feb 2019.

2. L. J. Tsoeunyane, S. Winberg, and M. Inggs, "An IP core integration toolflow for prototyping software-defined radios using static dataflow with access patterns," in *2017 International Conference on Field Programmable Technology (ICFPT)*, pp. 88–95, Dec 2017.

1.6 Thesis Overview

The structure of this thesis is organized in chapters as described below:

Chapter 1 outlines the background of SDR study and how the FPGAs have become the best choice in realizing the SDR waveforms over it's counterparts processor technologies. The chapter then discusses the instrumental role that the HLS tools play in the prototyping of FPGA-based SDR applications and the limitations these tools have in the implementation of SDR. It is further discussed in this chapter how the new intermediate level tool presented in this thesis, namely SdrLift alleviates the current HLS limitations. Lastly the objectives and the structure of the thesis are outlined.

Chapter 2 discusses the review of the SDR principles along with the underlying technologies and the FPGA platforms commonly used by experts in both the academic research and the industry. Furthermore, the chapter presents the contemporary HLS tools and how they compare with SdrLift presented in this thesis.

Chapter 3 gives the detailed definition and properties of the SDF-AP model which is integral to a design of complete SDR systems developed in SdrLift. The chapter then continues with definition of the computation methods for buffer size allocation and the latency computation under specified throughput constraints. Lastly, the operational analysis of the SDF-AP model are presented and the additional semantics that facilitate the description of SDF-AP model analyses are provided.

Chapter 4 presents the SdrLift language constructs and how they play an important part in expressing the structural behavior of the SDR applications. The chapter continues with how the SdrLift compiler creates the new HW Blocks using the template-based design and the DFG. This approach proves to be instrumental in the computation of the model properties of the SDF-AP while also decoupling the micro-architecture of the system from entry-point SDR program.

Chapter 5 presents the VHDL code generation in the form HW Blocks that

are also called IP cores. The HW Blocks a generated from the IR represented in DFG and are used to compose SDF-AP model that represents the actual SDR system. The existing IP cores obtained from the IP libraries are integrated and mixed with the newly synthesized blocks to create the SDF-AP model. The hardware implementation aims to reduce the behavioural gap between the SDF-AP model and the hardware model by using FSMs. The code generation incorporates the four optimization techniques for low-level hardware design synthesis the ensure optimized systems. Lastly, the process of code generation ensures that the generated hardware system correctly conforms to the high-level application descriptions using SDF-AP model.

Chapter 6 presents experimental results, using a case study approach in which the VHDL codes for eight representative SDR applications are developed. A more comprehensive case study of the FM receiver is also discussed where the generated hardware system is deployed on the FPGA platform. The results show that an efficient hardware system can be developed in SdrLift while also achieving the high performance required in SDR.

Chapter 7 presents conclusions of this research project, drawing on the experiment-based design and implementation of SdrLift and also discusses the extent to which the research questions have been answered. Lastly, the chapter discusses the possible future work in SdrLift.

Chapter 2

Literature Review[1]

This chapter presents a review of the state-of-the-art literature in the study of SDR and high-level tools that automate generation of hardware accelerators for SDR applications. Section 2.1 starts with an overview SDR and basic components followed by the common hardware architectures that realize the digital signal processing system and the reputable FPGA platforms which are widely used in both the industry and academic research. The chapter then continues with the study of HLS in Section 2.2, describing its key components and providing a context to compare and contrast the contemporary HLS tools with the approach proposed by SdrLift presented in work.

[1]This chapter is based in part upon the following publications:
L. Tsoeunyane, S. Winberg, and M. Inggs, "Software-defined radio FPGA cores: Building towards a domain-specific language," *International Journal of Reconfigurable Computing*, vol. 2017, 2017.
L. J. Tsoeunyane, S. Winberg, and M. Inggs, "An IP core integration tool-flow for prototyping software-defined radios using static dataflow with access patterns," in *2017 International Conference on Field Programmable Technology (ICFPT)*, pp. 88–95, Dec 2017.
L. Tsoeunyane, S. Winberg, and M. Inggs, "Automatic configurable hardware code generation for software-defined radios," *Computers*, vol. 7, no. 4, 2018.
L. Tsoeunyane, S. Winberg, and M. Inggs, "Sdrlift: An intermediate-level framework for synthesis of software-defined radio accelerators," in *2019 IEEE 10th International Conference on Mechanical and Intelligent Manufacturing Technologies (ICMIMT)*, pp. 166–173, Feb 2019.

2.1 Software Defined Radio

The Wireless Innovation Forum (WIF) has collaborated with the IEEE P1900.1 group to establish an accurate definition for SDR, a definition that has become widely used in the field. This definition for SDR is put simply as: "*Radio in which some or all of the physical layer functions are software defined*" [36, 16]. SDR can be viewed as a real-time communication system with high flexibility and ease of programmability that enable adaptation to various air interfaces. It uses software and reconfigurable digital subsystem to perform signal processing functions as opposed to a traditional hardware radio architecture which relies on analogue hardware to perform radio signal processing functions [1].

Since the introduction of SDR more than two decades ago by Mitola [37], it has been a driving force behind evolution of radio communication systems. The reason being it can be configured into one or more of the following operating modes: multi-band, multi-standard, multi-mode, multi-service and multi-carrier [15, 16, 38]. All these capabilities demand the underlying hardware to be flexible, easy to use, cost-effective (i.e. low cost tools and equipment) and able to provide high-performance under limited power budgets. More often, one or a combination of the following computing devices is used to realize the SDR systems: GPP, DSP, GPU, ASIC and FPGA which all lay good platform for implementation of SDR. Each of these devices presents to the designer the challenges associated with performance, power, cost and flexibility [39, 40].

The traditional radio systems are implemented in rigid hardware components which require physical intervention for adaptation to new waveform standards as shown in Figure 2.1. This design approach is inefficient, inflexible, costly and inhibits productivity. By contrast, SDR alleviates all the drawbacks associated with the traditional radio design through software upgrades that take place without a need to modify the underlying hardware platform. A typical SDR design is shown Figure 2.2 and it is capable of operating in a receiver and transmitter mode (i.e. transceiver configuration). For the receiver configuration, the antenna receives the Radio Frequency (RF) signal and relays it to the analog RF front-end where it undergoes amplification and filtering to remove noise and spurious

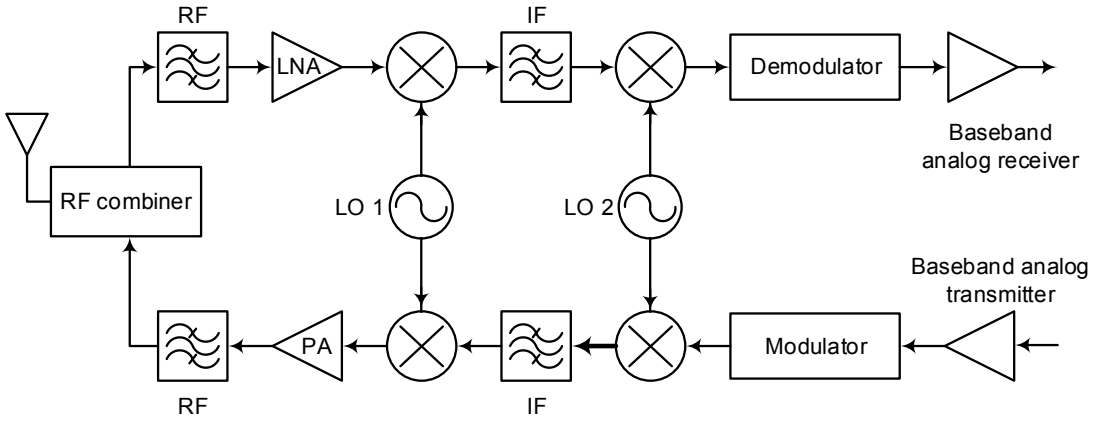


Figure 2.1: A traditional hardware radio architecture (based on [1])

signals. The analog Local Oscillator (LO) may optionally be used to convert the amplified, filtered and down-mixed RF signal to an analog Intermediate Frequency (IF) signal. Then the ADC samples the analog IF signal into digital IF signal samples. Following this is the IF front-end which converts the Digital IF signal into a baseband signal. The DSP section implements DSP operations on the baseband signal. Both the digital stages of the SDR design, namely the IF front-end and the DSP are typically implemented using a dedicated compute devices such as GPP, GPU, FPGA and ASIC. A transmitter is a reverse implementation of the receiver. In general, a transceiver as depicted in Figure 2.2 is composed of the antenna, analog RF front-end, digital front-end and a DSP stage all which are described below.

2.1.1 Antenna

The antenna is used to transmit (resp. receive) radio signals from (resp. to) the SDR system. Due to frequency-agile capability by most of the SDR systems in wide-band operation, multiple antennas are employed to cover wide range of frequency bands. For portable SDR systems, the size often limits the operational bandwidth, gain (or range), and radiation pattern. For some SDR systems, the SDR antenna is required to be capable of tuning to multiple bands, be able to perform beam forming and to be capable of rejecting interference [41, 42].

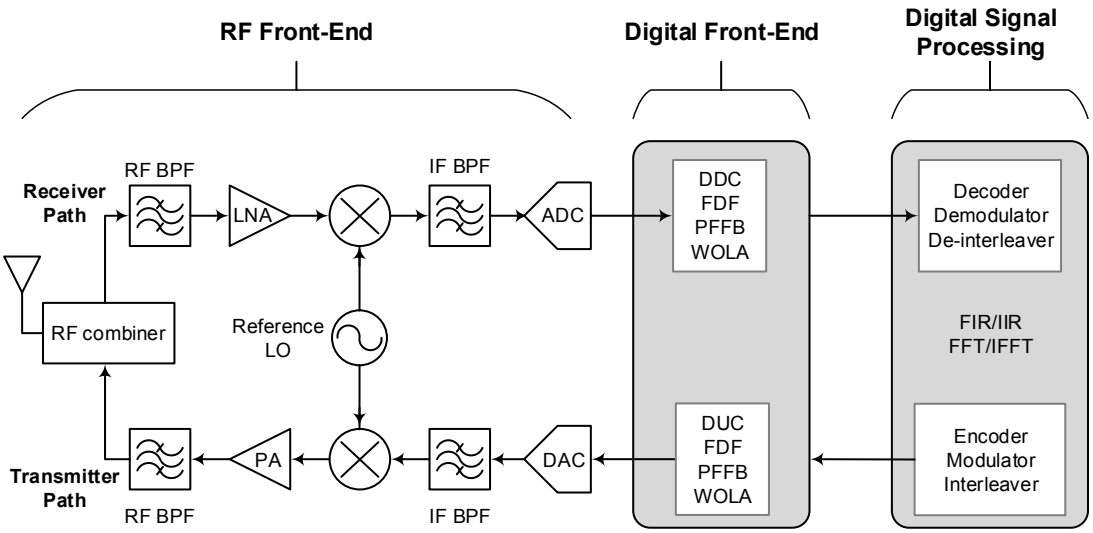


Figure 2.2: A software-defined radio architecture (based on [1])

2.1.2 Analog RF Front-End

The analog RF front-end transmits and receives analog signals at different operating frequencies. The RF front-end is closest to the antenna and its primary function is to convert RF signals to (resp. from) the IF signals using the transmitter (resp. receiver) chain of the SDR. It can therefore be configured to operate in the transmitter mode and/or receiver mode.

The transmit signal path performs the post-conditioning of the analog signal using various components such as DAC, Bandpass Filter (BPF), LO, Power Amplifier (PA) etc many of which are configurable via software. The DAC converts the digital samples into an analog signal at IF in the RF front-end. The BPF limits the bandwidth of the transmitted signal to the desired bandwidth and removes the quantization noise introduced by the signal reconstruction from the sampled data using the DAC [43]. The operation of the DAC is characterized by the (i) resolution: the number of possible output levels the DAC is designed to reproduce and it is quantified by the number of bits per sample, (ii) sampling frequency: maximum speed at which the DAC can operate to reconstruct the signal from digital samples, (iii) monotonicity: the ability of the DAC analog output to increase or remain constant as the digital input increases, (iv) Total

Harmonic Distortion plus Noise (THD+N): measures the extend to which the DAC output signal is distorted and the noise introduced to the signal by the DAC and (v) dynamic range: measures the difference between the noise floor and the maximum output signal level and it is measured in decibels [44]. The LO mixes the DAC output signal with the carrier signal and up-converts the IF signal to the RF signal that is transmitted over the air at the carrier frequency. Lastly, the PA is used to increase the magnitude of power of a transmit signal such that it is high enough to be detected at the receiver.

The receive signal path selects the desired signals from a wide range of RF spectrum and performs the pre-conditioning of the RF signal using the hardware components such as a Low Noise Amplifier (LNA), BPF, LO and ADC. The LNA amplifies the amplitude of the weak signals and minimizes the level of noise in the signal to allow signal detection by the receiver. The BPF selects the desired signals in the specific bandwidth and attenuates the unwanted signals outside the desired frequency range. The LO down-converts the amplified RF signal to the IF signal. The ADC converts received analogue signals from the analog domain to digital domain. The ADC often introduces in the output signal the noise and distortion the degrades its quality [45]. This effect can be measured with specific parameters that provide designers with fairly accurate correlation of the the performance expectations of a particular ADC. These parameters are divided into static and dynamic parameters. The static parameters are applicable in low frequency applications with frequency that is way below the operating frequency range of the SDR. They include offset error, gain error, Differential Non-Linearity (DNL) and Integral Non-Linearity (INL). The dynamic parameters are very relevant to SDR as they apply in high frequency applications [46] and they include (i) Signal-to-Noise Ratio (SNR): measures the ratio of the fundamental signal to the noise spectrum, (ii) Total Harmonic Distortion (THD): characterizes the ratio of the sum of power of the first six harmonics to the fundamental signal power, (iii) Signal-to-Noise and Distortion (SINAD): is the combination of SNR and THD and is further described as the sum of all spectral components except Direct Current (DC) and fundamental relative to the signal power measure, (iv) Effective Number of Bits (ENOB): is a figure of merit which tells how close the

ADC is near to the theoretical mathematical model and (v) Spurious Free Dynamic Range (SFDR): is the ratio of the level of the input signal to the level of the largest distortion components or spur.

The improved and higher sampling ADCs and DACs are pushing the tasks traditionally performed in analog closer towards the antenna, hence allowing them to be processed digitally using processors or reconfigurable devices [47]. However, a drawback is that the ADCs and DACs are usually costly, and achieving high sampling rates (over millions of samples per second) remain a limitation in SDR [8]; this is a motivating factor for reusable SDR platforms for prototyping to share the cost of the same platform across multiple projects.

2.1.3 Digital Front-End

The digital front-end stage converts the baseband signal to (resp. from) digital IF samples in transmit signal path (resp. receive signal path). The main functions performed by the digital front-end include Sample Rate Conversion (SRC) and channelization [48]. SRC converts sampling from one rate to another. In the transmit path, the sampling is converted from the lower rate to the higher rate while in the receive path the sampling is converted from higher rate to the lower rate. Channelization is considered the most computationally intensive part of the DSP as the processing is performed at very high sampling rates [49]. Its main functions include up (resp. down) conversion in the transmitter (resp. receiver) path and filtering to insert channels into RF band for transmission or to extract channels of interest from the RF band in the receiver side [50]. The most popular channelization techniques for SDR include a Digital Down Conversion (DDC), Frequency Domain Filtering (FDF), Polyphase FFT Filter Banks, and Weight Overlap Add (WOLA) FFT. [51, 50, 52].

2.1.4 Digital Signal Processing

The premise of SDR is to process all the tasks in a digital domain. While this may be practically inviable due to complexity of RF signal processing, researchers still work tirelessly to move the ADC/DAC and DSP closest to the antenna. The DSP is the continuous mathematical operations performed in real-time, and often this occurs quickly and repeatedly on large sets of data [53]. The DSP performance largely depends on the digital computing hardware device used. For compute-intensive DSP operations, GPP, GPU, FPGA and ASIC are commonly used. Further DSP operations are performed in software hosted on single/multi core Central Processing Unit (CPU) to provide functional and high-level abstractions needed by the end-users who in many most cases are non-hardware experts. Popular DSP operations adapted in SDR include (i) digital filtering implemented in finite FIR, Infinite Impulse Response (IIR) and virtibi decoder, (ii) convolution, (iii) Discrete Fourier Transform (DFT) using FFT and Inverse Fast Fourier Transform (IFFT), (iv) encoding/decoding, (v) interleaving/deinterleaving, (vi) modulation/demodulation, and (vii) scrambling/descrambling etc.

2.1.5 Overview of Hardware Architectures

This section discusses different hardware architectures used to implement the digital subsystem of the SDR. These architectures namely GPP, DSP, GPU, ASIC and FPGA are analyzed and compared as per a set of performance metrics such as processing power, programming models, strengths and weaknesses as summarized in Table 2.1.

Table 2.1: Comparison of GPP, GPU, FPGA and ASIC.

	GPP	**GPU**	**FPGA**	**ASIC**
Overview	Market agnostic	Very popular, especially in graphics processing applications	Restricted market	Market-Specific
Processing	Sequential, single and multi-core processors, wide choice of analog and digital peripherals	parallel, thousands of identical small cores, very limited peripherals which includes cache memory	massively parallel, thousands of hard or soft IP cores configured for application end, multiple peripherals can be interfaces via configurable I/O banks	massively parallel, includes thousands of IP cores tailored to a specific application, peripherals a hard-wired and application-specific
Programming	Very easy to program with a wide range of high-level programming languages such as C/C++, python, java.	Easy to program using OpenCL and Nvidia's CUDA Application Programming Interface (API) which allow general-purpose programming in C/C++, Python, Java, Fortran	Harder to program with Verilog and VHDL	Rigid, less programmable, few programmers using Verilog or VHDL
Strengths	Very versatile, easy to program and capability to run multiple tasks simultaneously	Very high processing power for massively parallel operations, exellent in floating point applications	Flexible and configuration can be changed, massively parallel, power efficient, very efficient for parallel and fixed-point operations	Optimized for a specific application, very high performance will very low power consumption
Weaknesses	Sequential processing, low performance, more overhead, not power efficient	very high power consumption, extra effort required to reformulate algorithms to exploit parallelism, may not suit some applications	difficult to program, relatively costly, poor performance for sequential operations and floating-point operations	Very costly, very few programmers, complex and takes long to develop, configuration cannot change once programmed

GPP

A GPP is optimized for sequential processing of tasks and it is one of the first computing platforms to be used in SDR [54]. GPPs are very popular and have for many years been used for unlimited number of applications without a need to build application-specific circuits. They are very cost effective and offer a flexible and easy way to program the SDR platforms. However, all these benefits come at the cost of abundant power consumption, large form factor and low performance.

DSP

A DSP is a special case of a GPP and it is specifically optimized for processing of DSP tasks. Because they are dedicated for a special purpose of processing digital signals, they have higher performance and consume lower power in comparison to the GPPs. Nevertheless, they too suffer from increased power consumption and limited performance when compared with FPGAs.

GPU

GPUs are designed for graphics and signal processing algorithms, more particularly where large blocks of streaming data and multiple tasks need be handled simultaneously. Most GPUs are coupled together with the GPPs to compensate for GPPs limitations in massively parallel processing (MPP). However, the high performance of GPUs is achieved through significant consumption of power.

ASIC

ASICs are permanently programmed for one sole purpose and their function never changes throughout the operating lifetime. Like FPGAs, the logic of ASIC is specified with HDLs such as VHDL and Verilog. ASICs can run faster than the FPGAs and are much more power efficient than the FPGAs. However, they are very difficult to program, very rigid and cost millions of dollars.

FPGA

The FPGA is composed of thousands of Configurable Logic Blocks (CLB) (i.e. LUTs, Multiplexers (MUX) and Flip Flops) and a sea of interconnects in which their behaviour is programmed in Verilog or VHDL to implement complex logic functions. In addition to CLBs and routing interconnects, FPGAs may optionally contain dedicated HW Blocks to perform specific functions and these blocks are Block Random Access Memory (BRAM), DSP blocks, Phase Locked Loops (PLL), Multi-Gigabit Transceivers, GbE, External Memory Controllers, a set of I/O cells etc. Unlike ASICs which are programmed once, the FPGAs are more flexible and they can be programmed multiple times [55, 56]. In SDR, FPGAs are the best computing architecture as they are capable of offering the best balance between performance, power, cost and flexibility [57].

FPGAs have led to the concept of design for reuse which is a driving factor in enhancing the productivity and improving the system-level design of SDR applications. The library parameterizable IP cores play an instrumental role in the effectiveness of the design for reuse [58]. Various configurations which include timing, area, and power allow for mix-and-match of the IP cores and allowing the designer to make choices based on the trade-offs that best suit the system under design [59].

The continuous design and implementation of a library of FPGA functions, called IP cores in this thesis, is increasingly driven by the desire to meet shortest possible time-to-market. This has led to greater demands of minimal development and debugging time [60, 58]. Many of the IP libraries have one or more of the characteristics listed such as [61, 62, 63, 60, 58]: modularity, parameterizability, portability, reusability, upgradability, specific technology independency, and the ability to consume fewer FPGA resources.

Hardware designers are relying on pre-designed IP cores from the IP libraries to increase productivity and reduce design time. However, many of the FPGA vendors and third-party IP libraries are static [62]. A static IP core does not allow high performance to be achieved even when hardware resources or power

budget is available nor achieve better performance to save both size and power consumption [62]. Integrating the third-party IP cores can also be a challenge. It is often time-consuming and error-prone [63]. The IP libraries developed by private vendors are expensive and prohibitive to low-cost prototyping [64].

All the above shortcomings of private vendor IP libraries have led to new open source hardware development models where reusable IP cores are developed and made freely available to the public. Two examples of communities supporting open IP cores are OpenCores and GRLIB. OpenCores has the considerable number of IP as well as Wishbone bus and its cores are accessible for free, however, OpenCores IPs are not parameterizable [64]. Likewise, GRLIB has many IP cores and are interconnected by AMBA-2.0 AHB/APB bus on a SoC design. But a drawback of using GRLIB is that not all the IP cores are free [63].

2.1.6 SDR Platforms

This section presents an overview of different types of SDR platforms. The study includes analysis of different different features that each platform and the role it plays in the SDR development community.

CASPER Hardware

The Collaboration for Astronomy Signal Processing and Electronics Research (CASPER) has over the past decade been working on the collection of FPGA platforms used to implement the digital radio-astronomy instruments hence reducing the time and cost of designing, building and deploying such instruments [65]. A brief overview of the CASPER FPGA platforms is given below:

- *Berkeley Emulation Engine (BEE) 2*: BEE2 was first developed as a processor emulation to speed up the development of new processor architectures. It was developed by University of California, Berkeley, but the latest iterations (BEE3 and BEE4) have been developed by BEEcube. It is described as a platform where "researchers can rapidly prototype a variety of

architectures in a relatively short amount of time by using a repository of low-level component design" [66].

- *Interconnect Break-out Board (iBOB)*: iBOB is the collaboration work with the Berkeley Wireless Research Center and the UC Berkeley SETI group. glsibob is based on the Xilinx Virtex II FPGA and interfaces with the ADC cards over a 10 Gb/s Ethernet network [67].

- *Reconfigurable Open Architecture Computing Hardware (ROACH)*: ROACH is a Xilinx Virtex 5 FPGA based platform that is designed in collaboration with a SARAO primarily for radio astronomy applications [68]. ROACH uses a single FPGA architecture that connects to on-board processor running Berkeley Operating system for Re-Programmable Hardware (BORPH) [69] and has enhanced memory and connectivity options. This provides a user with a simple interface to monitor and control the hardware design running on the FPGA. There is no need to use special JTAG programmers. An upgrade to ROACH is ROACH2 featuring a Xilinx Virtex 6 FPGA that provides with increased processing performance and I/O options.

- *Square Kilometre Array Reconfigurable Application Board (SKARAB)*: The SKARAB is the latest generation of the CASPER FPGA hardware and it succeeeds the ROACH2 plaform which to this far has been the most prolific of the CASPER platform generations [70]. Unlike other CASPER platforms, SKARAB is the product of Peralex Electronics (Pty) Ltd with its design specifications provided by the SARAO. SKARAB features a Xilinx Virtex 7 FPGA and it has four mezzanine card sites where each site is capable of providing an interface to 16 high-speed (10 Gb/s) serial transceivers. SKARAB currently support two mezzanine cards: a QSFP + Mezzanine Module which connects with four 40Gb Ethernet interfaces, and a Hybrid Memory Cube (HMC) module that provides additional memory capacity. Rather than to have the on-board CPU, the SKARAB provides with a the Computer-On-Module (COM) Express mezzanine site which intefaces with an external processor via single lane PCIe.

RHINO

RHINO is a standalone FPGA processing board and has commonalities with the better known ROACH, however, it is a significantly cut-down and lower-cost alternative which has similarities in the interfacing and FPGA or processor interconnects of ROACH. RHINO was designed at the University Of Cape Town and is largely aimed around a lower cost, totally open source FPGA board which provides a good platform for the development of SDR applications [71]. The RHINO platform was designed to be a combination of an education and training platform for learning about reconfigurable computing, and as a research and prototyping platform for studies related to SDR [72, 71].

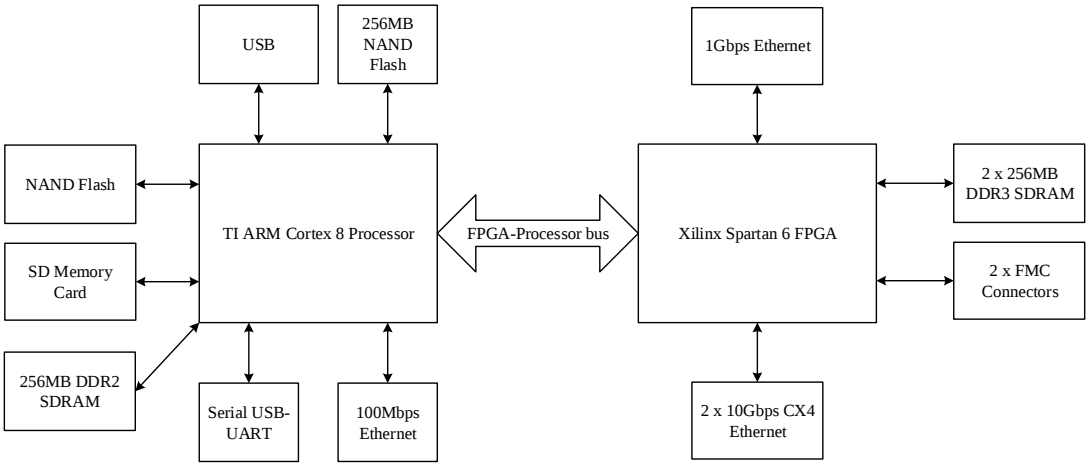


Figure 2.3: An architecture of RHINO platform building blocks [2]

.

The two main processing elements of RHINO include ARM processor and spartan-6 FPGA as shown in Figure 2.3. The computationally intensive functions are processed by the FPGA while the ARM processor provides configuration, control and interface function with FPGA through BORPH [71, 69]. BORPH is an extended Linux kernel that allows control of FPGA resources as if they were native computational resource [69]. This, as a result, allows users to program the FPGA with a given design or configuration and run it as software process within Linux. Other building blocks of RHINO include FPGA Mezzanine Card (FMC) connectors which enable interface with ADC, DAC, and mixed signal

daughter cards, supporting sample rates over 1GS/s [66]. The 1/10 GbE connectors provide a high-speed network connection between the FPGA and remote devices using standard Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) transport layer protocols to convey packets of data.

USRP N200 and N210

The Universal Software Radio Peripheral (USRP) N200 and N210 are FPGA boards designed by Ettus Research, specifically for SDR applications. The SDR supported applications include broadcast TV, mobile telephone network base-stations and satellite navigation, in both academic and industrial sectors.

Airblue

This is an FPGA-based SDR platform designed for cross-layer experimentation between PHY and Media Access Control (MAC) layers by connecting them with streaming interfaces [73]. This flexibiliy enables high system configurability to be achieved. The modular HW Blocks of Airblue are implemented in a low-level HDL language called Bluespec. Airblue offers modular refinements which enable atomic changes to be made to individual HW Blocks while imposing to the configurable radio system the two important design proterties namely latency-insensitivity and data-driven control. Furthermore, the HW Blocks are highly parameterized leading to increased reusability of the designs. In order to demonstrate flexibility and high performance, Airblue has been evaluated on the IEEE 802.11 using a set of complex protocol changes. The results show higher cross-layer communication speed can be achieved in comparison to Sora [17] and that it meets performance requirement of wireless prototocols [73].

WARP

The Wireless Open-Access Research Platform (WARP) is a highly capable, scalable and extensible platform designed for software/hardware codesign and pro-

totyping of wireless protocols [74]. WARP applications are targeted on high-performance programmable hardware, for instance, WARP v3 uses a Xilinx Virtex-6 FPGA. The SDRs are designed with Xilinx Embedded Development Kit (EDK) with the help of open-source repository of reference designs and support materials made avaiable to the research community.The WARP project has gained popularity with users around the world contributing to its self-sustained growth.

Zynq-based SDR

The Zynq System on Chip (SoC) is combination of a dual ARM core A9, and FPGA and additional hardware components for different I/O (i.e. Double Data Rate 3 (DDR3) memory interface, USB, etc) making it a high-performance hardware platform on which to deploy SDR. Drozdenko et al. [75] demonstrate the capability of Xilinx Zynq ZC706 by implementing the IEEE 802.11a and compare the performance with the Zedboard SoCs. The RF-frontend is based on Analog Devices AD-FMCOMMS2-EBZ [76] high-speed analog FMC module with integrated AD9361 RF agile transceiver module [77]. The gateware generation is automated via Model-based Design using MathWorks Simulink and HDL Coder [78].

2.2 High Level Synthesis

FPGAs are notably the best choice when it comes to designing the SDR applications [79]. This is because FPGAs provide high performance with reduced power requirements while also enabling flexibility and reprogrammability of the hardware design [57]. A typical design of an FPGA follows the RTL design process as depicted in Figure 2.4. The design flow starts with the algorithm that is implemented in RTL using a HDL like VHDL and Verilog. Both VHDL and Verilog have been widely used since the 1980s and they allow FPGA designers to describe logic using low-level components such as adders, multipliers, gates,

registers, LUTs. When the designers have completed the RTL description, they use Electronic Design Automation (EDA) tools like Xilinx ISE, Altera Quartus to carry out the physical layout steps verification, compilation, logic syntheis, translation, mapping, placing and routing. The end-product of this is the bit-stream that is programmed on the FPGA for system execution. The results for RTL design flow have in many cases demonstrated an improved performance, however, the effectiveness of the compiler heavily depends on the low-level EDA tool [80]. In addition, the designer needs to be equipped with rare skills that are typically found among hardware experts [58, 3] in order to fully harness the performance of the FPGA. Despite the high performance that can be achieved using the inherently low-level tools, it is very complex and time-consuming even for hardware designers with a great deal of experience.

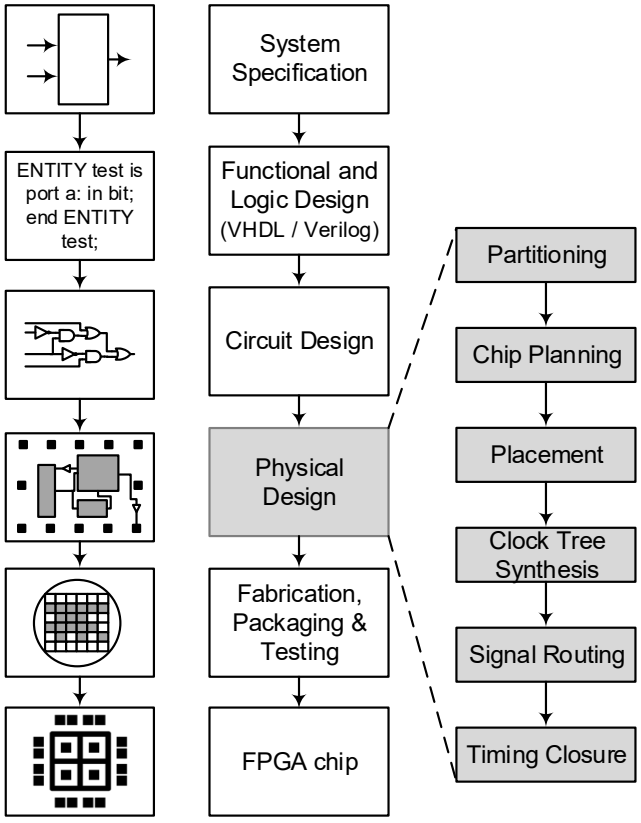


Figure 2.4: Register Transfer Level design flow (based on [3, 4])

FPGAs possess an abundance of logic, memory, register and I/O resources which

are a much more natural fit for implementation of DSP algorithms commonly used in high-performance SDR systems. Even so, great effort is required to manage these resources carefully in order to avoid over-utilization and failure to achieve optimal results while keeping power consumption to a minimum. In SDR, the parallel and pipelined DSP structures are often used to process data in real-time (i.e. tasks processing must occur within tight time constraints [81]) resulting in minimized hardware resources and increased performance. Moreover, the study shows the hardware capacity and complexity outgrows the productivity of hardware design [11, 5] resulting in what is known as hardware design-productivity gap as shown in Figure 2.5. In FPGAs, this problem was alleviated by design for reuse which concept inspired from developing FPGAs with pre-defined IP cores. These tried and tested IP cores largely improve hardware design productivity but extra-ordinary hardware skills are still needed in order to achieve optimal designs [58].

The complex tasks often performed at the low-level of design abstraction include (i) creating pipeline control, (ii) scheduling of operations, (iii) allocation of hardware resources, (iv) synchronizing of tasks, (v) managing both local and off-chip memory, (vi) co-ordination of fine-grained and coarse-grained parallel processes, (vii) third-party IP integration, (viii) off-chip I/O communications, (ix) and prolonged process of simulation and verification. While all these processes apply efficiently at low level design approach, there is a need to automate such processes in order to leverage parallelism and pipelining; and finding the appropriate computational model that encapsulates SDR system characteristics (throughput, sample rate, clock rate) and design quality attributes (area, speed, power, accuracy) all at high level of design abstraction.

FPGA end-users which include domain experts, scientists and software developers etc., and they often do not possess the low level skills and they seek tools usable at high level of programming abstraction [82]. In order to make the FPGAs more accessible to the end-users, the HLS tools were developed. HLS is a process of using a HLLs like C, C++, Java, Scala, Python to write algorithm specification and automatically convert it to a low level RTL language that is optimized for performance, area and power requirements [4]. Recent surveys

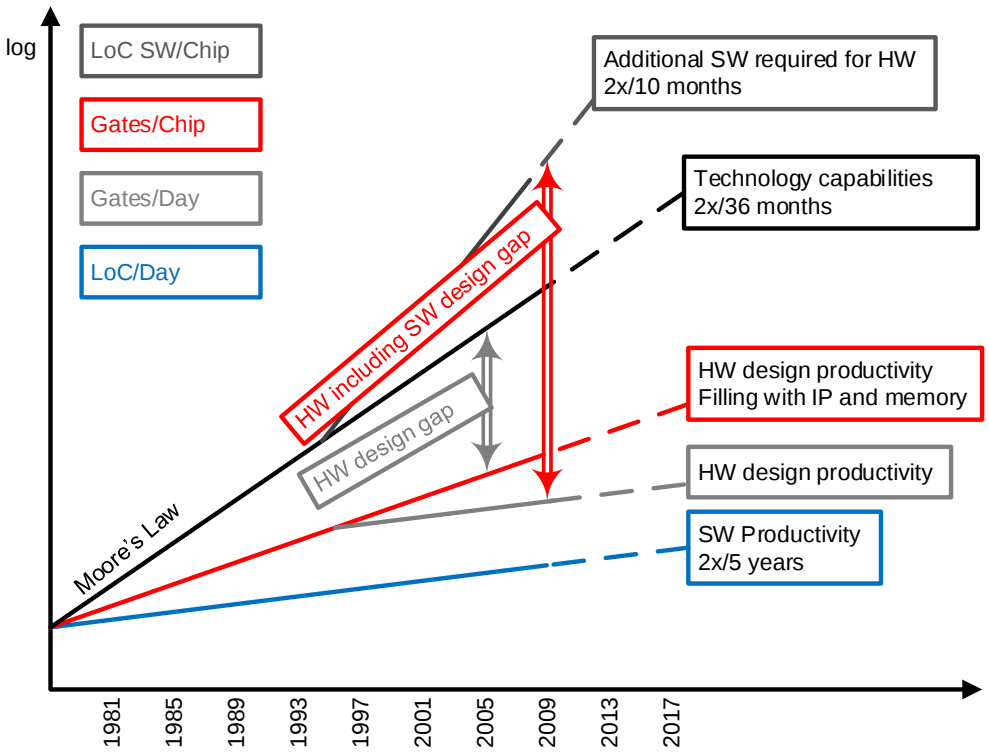


Figure 2.5: Hardware, software and design-productivity gap [5]

and evaluations show that HLS tools can effectively produce competitive designs with hand-crafted designs using RTL languages [40]. Furthermore, the open-source HLS tools which formerly fell short of adaptation in industry, academia and research community have in recent years shown capable of competing with commercial HLS tools [83, 84, 85].

A typical HLS design flow consists of an algorithm specification using a HLL as shown in Figure 2.6. The specification is compiled into a computational model that can easily enable back-end HLS tasks to be performed. These tasks include (i) allocation of hardware resources needed to satisfy design constraints, (ii) scheduling of the operations with respect to clock cycles, (iv) binding the HLS descriptions to the hardware resources, e.g. operations map to the functional units, variables map to the storage elements, data transfers map to the buses,

etc. Following this is the generation of RTL description in either VHDL or Verilog. Using the EDA tools, the RTL is then converted into gate-level netlist which is in turn transformed into physical layout. The tasks that take place to create a physical layout are depicted and described in Figure 2.4.

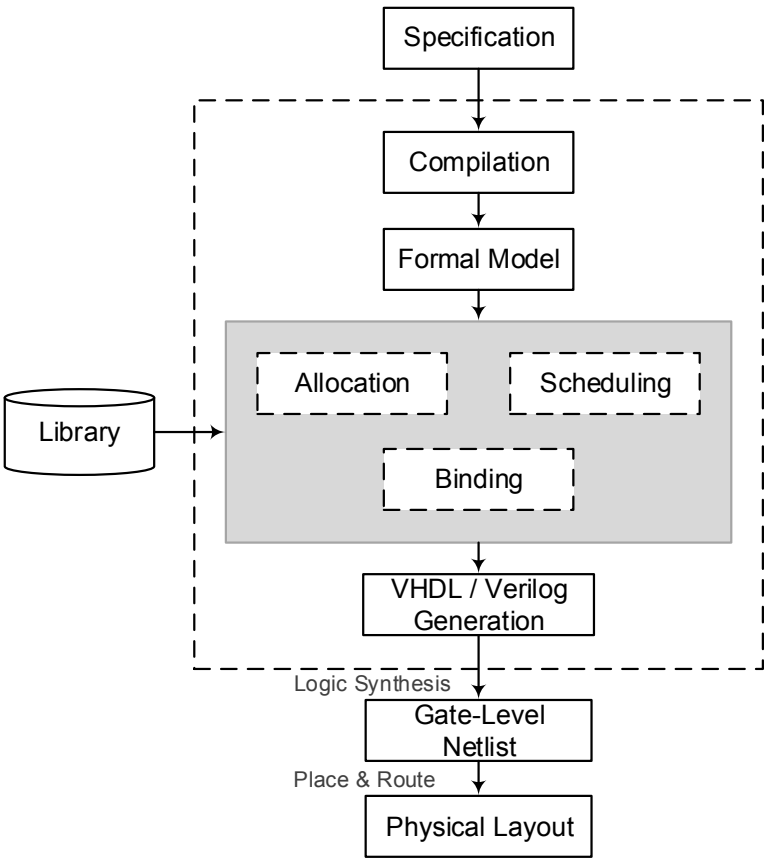


Figure 2.6: High Level Synthesis design flow (based on [3, 4, 6])

2.2.1 Synthesis from General Purpose Languages

The contemporary HLS design often involves synthesizing hardware from input specifications using general purpose languages like C/C++, OpenCL, Java, Scala. This traditional approach is typically planned around being generic and comes with a benefit that HLS solution is provided for wide range of problem domains like Video and Image Processing, DSP, machine learning, high perfor-

mance computing, communications, radar, radio astronomy. However, there are several limitations of using this approach such as (i) many HLS design tools struggle to find adoption by wide design community due to poor quality of synthesis results [86], (ii) users are still required to apply low-level skills involving using of pragmas and code refactoring in order to get better results, (iii) the HLS tools are not optimized for particular designs making it difficult to expose domain-specific operations and optimizations [14], (iv) and the imperative languages such C, C++, SystemC. are intrinsically low-level and require code restructuring to obtain optimized results [87].

2.2.2 Synthesis from Domain Specific Languages

Although the HLS tools are meant to improve productivity, enhance design performance and also to reduce difficulties of FPGA development, the final design often suffers tremendous inefficiency [14]. Besides, their wide adoption by design community is still not satisfactory due to the steep learning curve. To mitigate these problems, the domain knowledge of non-hardware experts is exploited by developing DSL [88, 89, 90, 82, 91]. Unlike general-purpose programming languages, the purpose of DSLs is to provide solutions in a particular problem domain to subject-matter experts. This results in easily understandable, reusable, maintainable designs that are easy to reason about. However, the successful development of DSLs is often inhibited by the cost of overhead in DSL infrastructure and a enormous effort required to build them [91].

There are two variants of DSLs namely internal (embedded) and external DSL. Internal DSLs are embedded in another language and their syntax and optimizations derive from the compiling framework and other features provided by the host language. Whereas the external DSLs require the development of the new host language including the compiling framework [39]. Using internal or embedded DSLs reduces many of the costs involved in developing external DSLs. Moreover, recent work in DSL development reveals that functional languages are best suited to raising a level of programming abstraction for generation of reconfigurable accelerator hardware [25, 24]. When compared to imperative languages,

they provide the cost-effective way of coupling DSL with HLS while increasing productivity through exploitation of rich syntax to build optimized, expressive, concise and scalable applications [88, 7].

2.2.3 Hardware Synthesis from Computing Models

Another body of work in HLS explores transformation from high-level programs to the underlying Model of Computation (MoC) which in turn translates to the low-level hardware accelerators. MoCs are actively used in DSP systems to specify, simulate and/or execute algorithms [92] and Sangiovanni-Vincentelli [93] advocates for its use in system-level designs such as HLS. There are various reasons why MoCs suit the HLS are (i) they are ideal for design, programming and validation of concurrent and stream-based DSP applications, (ii) and they properly describe parallelism and data dependencies in a manner that is natural to how software and hardware operates [94, 95]. In hardware-oriented SDR, it is necessary to implement systems that have high decidability and predictability while also meeting the requirements of SDR dynamism and the scarcity of underlying hardware architecture resources [95]. Therefore designers often face a challenge of choosing the best MoC that have both expressivity and analysability properties. MoCs are classified into two: (i) Process-based models include Kahn Process Network (KPN) [96], Dataflow Process Network (DPN) [97], Synchronous Dataflow (SDF) [98], Cyclo-Static Dataflow (CSDF) [99], Parameterized Synchronous Dataflow (PSDF) [100], and Boolean Parametric Dataflow (BPDF) [101]. (ii) State-based models include FSMs [102] and Petri Nets [103].

A process network or dataflow model is a directed graph that is composed of vertices (actors) interconnected to one another via edges (channels). The actors represent computational blocks while the channels are Finite In First Out (FIFO) queues which facilitate communication between the actors. At each execution (firing), an actor consumes data from one or more input channels or produces data onto one or more output channels. The data is referred to as tokens. One of the earliest and popular models namely the KPN uses a network of processes that communicate via unbounded FIFO queues with blocking read and non-blocking

write semantics [96]. For instance, MAPS [104] implements a compiler flow of SDR applications on heterogeneous platforms using KPN. The problem with MAPS is that KPN deadlock detection is undecidable. An extension to KPN is the DPN which divides the behaviour of each actor into a sequence of execution steps known as firing [97]. Ptolemy [105] is one of many projects which makes use of DPN.

For an SDF model, an actor consumes a fixed number of tokens from each input channel and produces a fixed number of tokens on each output channel [98]. This use of constant consumption and production rates in SDF plays a fundamental role in guaranteeing decidability and predictibilty of key model properties at compile-time. These properties include: boundedness, deadlock-free model, liveness, throughput, execution schedule and latency. An SDF is one of the most popular MoCs and as result it is used in various HLS tools such as LabView [22], frame-based DSL [39], CAPH [106]. However, SDF models are not capable of specifying and capturing critical information about data access with respect to time. As a result, they can be too defensive by allocating buffers that are too big or can be incorrect by allocating buffers that are too small [26]. To work around this problem, a common practice is to use Worst-Case Execution Time (WCET) models to capture timing information [107, 108, 109, 110]. The problem with WCET is that it does not analyze timing behavior of accessed tokens at precise specified clock cycles from start of information as required by IP blocks in FPGA-based systems. The result of this is sub-optimal and conservatively estimated hardware resources [111].

A special case of SDF namely CSDF strives to address the above problems. CSDF is a generalization of SDF model in which consumption and production rates are allowed to vary dynamically according to a fixed pattern [99]. At each firing there exists a phase of the cyclic pattern. Although it may seem dynamic due to varying rates during firing, CSDF model is classified as static due to its compile-time predictability. Like SDF, CSDF still relies on the hypothesis that an actor starts firing only when there is enough tokens in its input channel. This hypothesis is not desired when analyzing precise timing information of token access.

Both SDF and CSDF are static models, however, many DSP applications require that the consumption and production rates vary at run-time. In order to model the dynamic behavior of many DSP applications, a convenient dataflow MoC is essential for analysis and specification that will offer a good trade-off between expressivity and analysability. Recent studies show that Parameterized dataflows such as PSDF as well as Parameterized Cyclo-Static Dataflow (PCSDF) use setting of parameters to enable run-time changes in model rates. This has proven to significantly extend expressive power of SDF and CSDF without excessively compromising the analysability of the system properties [94, 95, 112, 113, 114, 115]. However, it is extremely difficult to implement the analysis techniques of such models. It is even more challenging to realize them on hardware due to resource-intensive control logic which also results in undesirable latency. Furthermore, analysing timing behavior of token access with respect to precise clock cycles is even more difficult to achieve.

The MoCs described above exhibit the deficiency of inability to capture the timing information about token consumption and production resulting in defensive and incorrect DSP designs. To remedy this problem, SDF-AP model was introduced in [26]. Other formalisms regarding its syntax, semantics and analysis algorithms are presented in [111, 116]. SDF-AP is an augmented version of SDF where each actor fires for specified number of clock cycles called execution time. An actor further associates each channel with a binary pattern word with a length equal to the actor's execution time. The binary value 1 denotes a single token read or token write from or on input channel or output channel respectively in a single clock cycle. The binary value 0 denotes there is no token read or token write operation. The information about execution time and access patterns is obtained from the IP core documentation or manual IP simulation using low level simulation tools. In this thesis, SdrLift is designed around modeling the integration of the IP cores using SDF-AP, and most importantly, to use it to direct interfacing decisions (e.g. buffering sizes) in an attempt to satisfy speed and resource usage objectives.

2.2.4 High Level Synthesis Tools

The summary of popular state-of-the-art HLS tools in Figure 2.7 is adopted from Shao [6] et al. approach that categorizes the HLS tools into HLLs, C-based languages and hardware description languages.

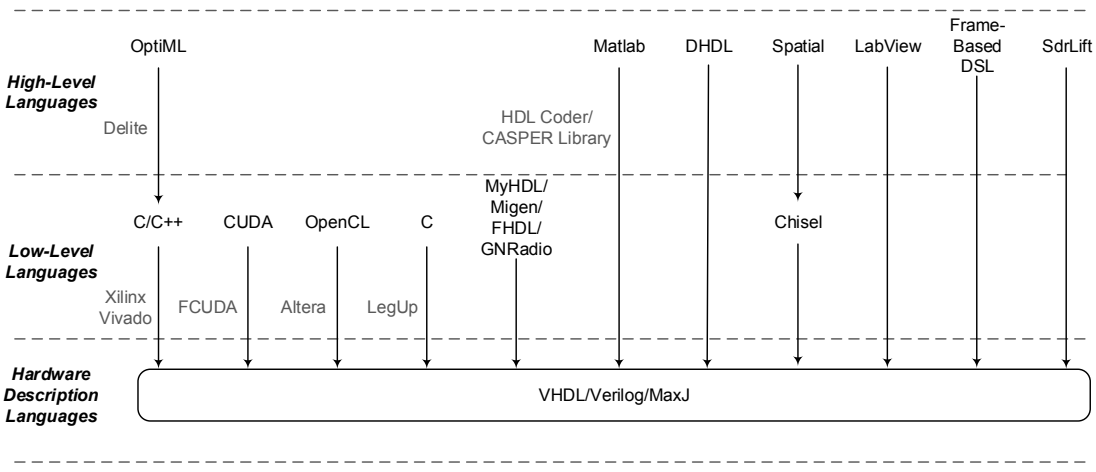


Figure 2.7: High-level synthesis landscape (based on [6])

The HLS tools which are shown in Table 2.2 are commonly used in both academia and industry and are further described below.

Altera SDK for OpenCL

OpenCL is a C-based open industry standard that automates code generation targeted at heterogeneous computing platforms. Altera SDK for OpenCL (AOCL) exploits parallelism and pipelining to generate FPGA circuits using OpenCL C. Each stage of a pipeline represents a different thread, as a result the generated hardware code is efficient and most suited for dataflow and streaming applications [117, 40].

Chisel

Chisel is embedded in Scala and it is characterised by programming concepts such as object-orientation, functional programming, parameterized and type inference. The aim of Chisel is to provide high productivity by using Scala applications to generate C++ simulators and Verilog RTL description. The elements of a Chisel are interconnected using the RTL approach [118, 40].

FCUDA

FCUDA implements a high level FPGA programming flow that maps course and fine grained parallelism on to the FPGA using CUDA. The input instructions of the flow are written in CUDA and FCUDA compiles Single Program Multiple Data (SPMD) CUDA code into coarse grained parallel code in C. The Autopilot [119] then extracts the fine grained parallelism from FCUDA generated code to produce final RTL design [120].

LegUp

LegUp is C-based HLS tool that compiles a C input program to map on a hybrid architecture that is composed of 32-bit FPGA-based MIPS soft processor and custom hardware accelerators which are interconnected to each other by Altera Avalon bus interface. The LegUp's C compiler compiles C code into binary executable that run on MIPS processor. Furthermore, the LegUp's HLS uses LLVM [121] compiler framework to convert input C into IR which undergoes analyses and a series of optimization passes. Legup HLS interacts directly with Low Level Virtual Machine (LLVM) IR and converts it into Verilog that is synthesised into bitstream using third party tools [122].

CASPER Library

The CASPER Group at the University of California Berkely has developed an open-source DSP library and glue logic functional blocks that enable radio astronomers to rapidly prototype real-time systems that run on reconfigurable, modular instruments [123]. The CASPER library adopts a tool-flow namely Matlab/Simulink/System Generator/EDK (MSSGE) where Matlab implements the script-based backend for Simulink. Simulink provides schematics and system modeling to capture the design user design specifications. System generator is used for translation of Simulink blocks into VHDL or Verilog and also provides with design simulation. Lastly the Xilinx EDK synthesises the HDL design into bitstream ready to be downloaded into a target FPGA device [124, 125].

Delite

Delite is a compiler framework and runtime developed by Stanford University's Pervasive Parallelism Laboratory (PPL) [7]. It builds on top of another compiler framework called LMS [126] whose primary purpose is to enable development of embedded DSLs in Scala. As an extension of LMS, Delite provides developers with reusable components like, parallel patterns, optimizations and automatic code generators to facilitate construction of embedded DSLs. The current support of resultant languages includes C++, CUDA, OpenCL and Scala. C-code generated by DSLs such as OptiML [127], OptiSDR [31, 89] can be converted into RTL implementation using Xilinx Vivado. Delite Hardware Definition Language (DHDL) [25] is another DSL embedded in Delite and it generates MaxJ code that is ready to be converted into RTL by Maxeler compiler and be executed on Maxeler platform.

The Delite Architecture as shown in Figure 2.8 starts with the DSL progam that borrows from Delite the Scala-based data structures and datatypes: array, vector, struct, hashmap etc. that facilitate the specification of the dataflow model. Moreover, the dataflow actors leverage the domain-specific computation patterns map, reduce, zipwith, sort, foreach, group-by, filter and serial which

are used to define the domain constructs. These actors are instantiated at the top-level design of the application and they represent the existing library cores with ports and generic parameters.

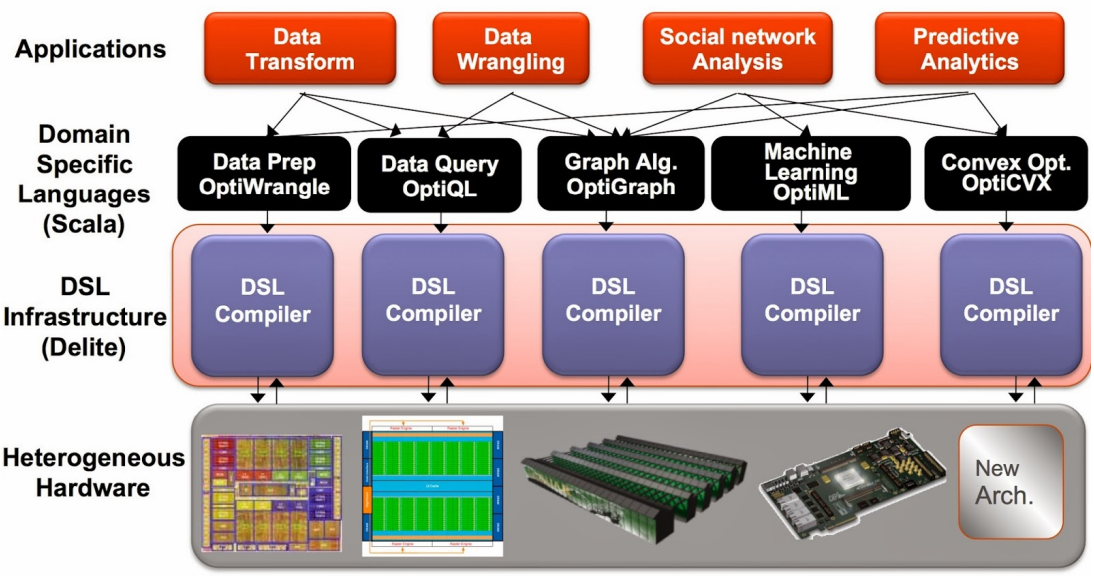


Figure 2.8: Overview of Delite Compiler Architecture (adopted from [7])

Another python-based package is Migen [128]. Migen is a tool that uses python to automate the FPGA design process. It takes advantage of the modern software programming concepts such as object-oriented programming and meta-programming to describe hardware design at the higher level. At its lowest layer lies a Fragmented Hardware Description Language (FHDL) which defines the digital hardware at the register-transfer level. The FHDL is then converted to Verilog as the final output of the Migen tool flow [128].

Matlab and Simulink

Simulink [78] combines textual and graphical programming to model, simulate and analyze multi-domain systems. The SDR applications can be developed using a wide range of built-in functions/tools and components, in particular, the ones that support DSP and telecommunication systems [12]. Developers use HDL coder to generate Verilog or VHDL and various optimizations are supported

at high-level and low-level of optimizations. Incorporating the external IP cores requires the creation of an interface (called Black-Box) that models the existing subsystem in VHDL or Verilog. Models can be stitched together by connecting the ports across different models using an interactive GUI. While the predefined models can easily be used, the code generation method does not support dataflow models and the FPGA programming only targets the Xilinx and Intel SoC device, therefore limiting generated code portability. Furthermore, HDL coder requires a license to use and the source code is not open for the public to access.

Vivado HLS

Xilinx Vivado IP Integrator [21] uses a combination of IP Integrator and Xilinx System Generator [129], that provides support for several languages (e.g. C, C++, SystemC, and OpenCL), for facilitating the process of generating VHDL or Verilog for multi-domain FPGA applications. These offer developers with several manual optimization rules through the use of directives and pragmas which require extensive knowledge of low-level hardware design. Hence the quality of results relies upon the hardware skills that the designer has. The generated IP modules can be reused in a Vivado Design Suite and they can only be programmed on Xilinx FPGA devices. Vivado HLS does not support dataflow models and its a commercial tool that requires expensive license to use.

LabView

LabView [22] offers a graphical programming approach that helps to easily develop and visualize the multi-domain applications such as data acquisition, instrument control, industrial automation, SDR. IP integration requires the IP synthesis files, such as .vhd files, Xilinx IP configuration files, or netlist files to meet the requirements for predefined rules for integrating the hardware models. LabView supports FPGA code generation from SDF models and other SDF-extended models which include Homogeneous Synchronous Dataflow (HSDF), CSDF, PSDF and PCSDF [113]. Furthermore, LabView supports FPGA de-

sign using an SDF model with access patterns SDF-AP. The LabView hardware implementation prototypes using the SDF-AP model are discussed in [26, 116, 111, 130] where the experimental results indicate that SDF-AP model can reduce the buffer size requirements by about 63%. However, generating the FPGA code from these supported dataflow models relies on *correct-by-design* approach which does not provide conformance analysis of the hardware design. Furthermore, using LabView requires a licensing (i.e. uses closed source-code that is not shared with the public), and the FPGA design caters well for National Instruments and compatible systems. As a result, these limitations make it very difficult for experimentation by the open-source community.

Frame-based DSL for SDR

Ouedraogo et al. [39] present a frame-based DSL for prototyping SDR applications on FPGAs and the DSL is available as open-source to enable further research in the SDR community. The DSL design flow uses the SDF model to connect the IP components and to communicate frame information among the network components. Furthermore, it supports high-level HLS optimizations and enables the re-use of defined components. The disadvantages of these DSL are that it only supports frame-based applications such Orthogonal Frequency Division Multiplexing (OFDM) systems, it does not formally bridge the semantic between the SDF model and the generated hardware designs, and lastly the SDF model does not use the access patterns which are necessary for correct and optimized designs.

Python-based HLS tools

There are other FPGA design packages which do not offer sophisticated capabilities such as the ones in HLS tools, however, they can still simplify FGPA development and improve productivity. The open-source Python open-source community has a few packages which can be used to automate FPGA design using python. MyHDL is an open-source package that uses python to describe and

verify FPGA digital circuits at the register-transfer level of abstraction. It retains many of the RTL modeling concepts used in VHDL and Verilog while leveraging the syntactic and semantic power of python. MyHDL is not considered to be a HLS, however, the design specifications in MyHDL can be converted to user-readable VHDL and Verilog [131]. GREasy is another good example which extends GNU Radio such that the resulting SDR application can execute both on the Personal Computer (PC) and the FPGA [132]. It relies on flow graphs to automate the assembly of software and hardware modules of the model-based application.

Other Tools

The readily available and verified IP cores have proven to be fast and efficient means for gateware development [59]. Consequently, several attempts to integrate them using high-level design tools have been made in which the goal is to increase design productivity and abstract the complexities of manually integrating the cores. Some of these tools are provided by mainstream IP vendors, such as Altera SOPC builder [133] and Xilinx Vivado IP Integrator [21], which both give the user a freedom to define interfaces for the existing cores and to manually connect up the IP cores using C/C++. However, they do not automatically integrate the IP cores, leaving the user with the time-consuming tasks of fine-tuning the low-level C/C++ code to enable IP integration at the system level. Other tools such as VSIA [134] and Open Core Protocol (OCP)-IP [135] strive to specify open interface standards such as the Virtual Component Interface (VCI) and the OCP which simplify IP integration on the SoC. IP-XACT is an IEEE standardized (IEEE 1685-2009 [136]) XML schema for IP integration and it is provided by Accellera Initiative [137]. This standard has been adopted by tools such GRIP [138], etc to ensure interoperability and reusability of a wide range of IP cores through parameterization, good documentation and sample code to explain the IP core interface. Recently, Yang et al. [139] presented a generalized behavioral IP integration framework using a HLS for both synthesizable and non-synthesizable IP cores. Their tool enables support for two effective interfaces for

fixed and variable-latency IP cores which can be instantiated internally (i.e. the IP cores instantiated within the HLS-generated top module for the application) or at the behavioral level. The drawback of these tools is that they are typically designed around being more application-independent (as opposed to being design with SDR stream-processing uses in mind), which leads to a relatively higher overhead in terms of accommodating a range of possible interfaces and protocols [140]. Amongst the FPGA generators that support dataflow models, Ptolemy [141] is the leading open source framework used predominantly in the academic research. However, it supports restricted dataflow and focuses more on proof-of-concept prototyping, rather than generating efficient solutions. Commercial tools that support dataflow models include LabView [22] and Simulink [20] among others; but these tools, along with their support libraries, are expensive to license, and they use stateflow and dataflow models which are inefficient for hardware targets. OpenDF is another dataflow programming framework that is based on the Cal Actor language (CAL) to generate FPGA designs [142]. However, CAL builds on a dynamic dataflow model which is undecidable and suffers from the inability to capture timing information for data access. Another recent initiative is the CAPH language [106]. This is a domain-specific language that is used to describe and implement stream-processing applications on the reconfigurable hardware. SDF is used in this language to connect the model actors while the behavior of the actors is defined as a set of transition rules using pattern matching. The disadvantage of CAPH is a deep learning curve, in particular, defining rules of actor behavior and it also uses an inefficient SDF model which does not capture timing information about token production and consumption.

Comparison of HLS tools

Table 2.2 shows the HLS tools and provides a comparison of the features which are contributed to SdrLift framework. The last five tools closely relate to SdrLift in that they support prototyping for SDR applications. The first desirable feature determines whether the tool is domain-specific to SDR which helps to increase design productivity using high-level constructs and language idioms familiar to

the domain expert. This is followed by a feature which determines whether a tool generates VHDL or Verilog which are popular and standard hardware description both in academic research and industry. Using a functional programming approach enables programs that are free from side-effects, inherently concurrent, race-free and the immutable data structures results in programs that are easy to reason to about. In order for the HLS to be Turing-Complete, it must support both the algorithmic specifications and IP core integration at the high-level of design abstraction. A dataflow model of computation is another feature which hides the low-level behaviour by performing analysis of timing and performance properties at the high-level of design abstraction. Furthermore, the dataflow model is required to have access patterns, a desirable feature that moves a dataflow model closer to a hardware by determining exact clock cycles at which the data tokens are produced and consumed. The high-level design constraints should be correctly mapped to the low-level to ensure the conformance of the low-level design to the high-level design. In addition, the predefined hardware descriptions using the prototyping tool should easily be re-used to speed-up the hardware design. In order to produce quality results, the tool must automate optimizations at the high and low levels of a hardware generation. Lastly, the tool has to be open to the public to enable modification and improvements.

Table 2.2: Comparison of features for different SDR prototyping tools.

Features	***Altera SDK***	***Chisel***	***FCUDA***	***LegUp***	***CASPER Library***	***Delite***	***Simulink***	***Vivado***	***LabView***	***Frame-based DSL***	***SdrLift***
Domain-Specific to SDR	✗	✗	✗	✗	✓	✗	✓	✗	✓	✓	✓
Generates VHDL/Verilog	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
Based on a functional language	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✓
Synthesis from algorithmic specification	✓	✗	✓	✗	✗	✓	✓	✗	✓	✓	✓
Support IP core integration	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Uses a MoC	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓
MoC supports access patterns	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓
Low-level to high-level conformance analyses	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Supports design re-use	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
Automates optimizations	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
Open-source and available to public	✗	✓	✓	✓	✓	✓	✗	✗	✗	✓	✓

The summary of modern HLS tools as tabularly depicted in Table 2.2 clearly show that most of the HLS tools are built upon imperative languages whose major disadvantage is lack of expressive power and inability to naturally represent high level models for hardware design. Furthermore, a number of HLS tools strives to provide solutions in all problem domains, these limits aggressive optimizations resulting in lesser performance and sub-optimal FPGA designs. Many of the popular HLS tools are commercial but they are very expensive. In this project, a SdrLift shown in the last last column of Table 2.2 is presented.

In this thesis, a methodology which automates the seamless integration of IP cores for SDR applications that runs on the FPGA is developed. The framework namely SdrLift is based on the proposed methodology in this work relies on the SDF-AP model which aids in the system analysis and optimization. The framework also leverages the embedded DSLs features of a functional programming language namely Scala which include highly expressive and modular constructs for expressing functions quickly. The conformance analysis ensures the system correctness and adds confidence to the results that satisfy the specified design constraints.

2.3 Chapter Summary

This chapter has shown that there is a need for a drastic shift in a programming paradigm for the high-level design of FPGA architectures. Many HLS tools still hold on to sequential models using imperative languages which are not favourable for hardware design. While the imperative languages impose overhead on the synthesis to discover concurrency and pipelining, the users also find them difficult and time-consuming. This is because they describe hardware at a low-level and require the use of directives or pragmas in the source code to generate efficient hardware designs [25]. Furthermore, meeting FPGA design timing requirements using a HLS tool can be extremely challenging. The challenges appear when third party IP cores that interface with external memory or mixed-signal devices are integrated into the design. Human interaction with RTL code is typically

required whenever the timing requirements are not accurately met, however, human experts often find machine-generated RTL code complex to read. This is because the HLS-generated RTL contains components, wires and registers' names that do not correlate with the input specification as a result of preceding code optimizations and transformations. Consequently, it can be very time-consuming and sometimes impossible to perform debugging and verification using third-party RTL simulation tools [143]. In this project, an effort is made to use variable, signal and component naming conventions that will result in easy code reading.

The functional programming languages represent a better model for the realization of FPGA hardware architectures at a high level of design abstraction. This is because the order of their operations is not sequential, rather, they encapsulate a data-dependency model that neatly matches the intrinsics of a hardware [24]. Implementing DSL languages that couple with HLS tools using the functional programming language model is, therefore, a promising path to hardware design. The SDR architectures are marked by parallel, real-time, multi-rate and streaming characteristics [112] which make a DSL an ideal tool to capture SDR algorithmic specifications. DSLs have more benefits over general-purpose languages because they offer a higher level of design abstraction using rich types, data structures and parallel design patterns that allow users to express and confidently reason about the system behaviour [7][144]. Furthermore, they support aggressive optimizations which in turn result in high performance and optimal designs. In a nutshell, the toolflow proposed in this project expects expressive and modular DSL defined using functional language semantics.

The DSL could directly be translated into gateware. However, such hardware design approach would be very unpredictable, complex and results would be highly inefficient, inaccurate and poor. Rather, most HLS tools convert the DSL code into IR which undergoes various analyses, optimizations and transformation passes without deviating the original algorithmic information [90]. The final step is emitting RTL code using a code generator. Stream-based, real-time and parallel DSP algorithms are often mapped into a MoC before RTL code generation [90][18]. Dataflows are classified into static and dynamic data. Dur-

ing design, one has to trade-off between expressiveness and analytical properties [114] which correspond to dynamic and static dataflows respectively. There are several dataflow-based HLS tools for fast prototyping of SDR such as Ptolemy [141], LabView [22] and Simulink [145]. Other tools combine static and dynamic dataflow properties in order to accommodate multiple modes of operations of SDR system. These previous attempts to exploit both properties [113][144][114] have revealed challenges faced which include minimizing buffer size while keeping the throughput high and reducing the glue logic required to realize schedulers. SdrLift seeks to adopt the SDF-AP model which results in the efficient use of hardware resources with a premise of an accurate analysis of data access patterns with regard to the clock cycles.

Chapter 3

Dataflow Model[1]

This chapter presents a dataflow model of computation used as an integral component for implementation of SdrLift back-end. This model of computation namely, SDF-AP [26] is employed for computation of timing and performance properties of the hardware system thereby raising the level of design abstraction. The SdrLift approach targets the problem domain of SDR in which the resultant solutions run on the FPGA platform, more specifically, the contributions are as follows:

- Present the operational analysis of the SDF-AP model and provide additional semantics that facilitate the description of SDF-AP model analyses.
- Define the computation methods for buffer size allocation and the latency under 1-periodic scheduling [146] and throughput constraints.

[1]This chapter is based in part upon the following publications:
L. J. Tsoeunyane, S. Winberg, and M. Inggs, "An IP core integration tool-flow for prototyping software-defined radios using static dataflow with access patterns," in *2017 International Conference on Field Programmable Technology (ICFPT)*, pp. 88–95, Dec 2017.
L. Tsoeunyane, S. Winberg, and M. Inggs, "Automatic configurable hardware code generation for software-defined radios," *Computers*, vol. 7, no. 4, 2018.

3.1 The SDF-AP Model

A dataflow program is represented as a collection of computational elements (called actors) exchanging data objects (called tokens) through unidirectional channels (FIFO). Execution (called firing) of actors is governed by a set of rules called the MoC. These rules specify when the actor can be activated based on a number of tokens available on its input channels and when the tokens can be written on the output channels. A commonly used dataflow model, called SDF [98], guarantees decidability and predictability of key model properties at compile-time. However, it does not specify how tokens are consumed and produced with respect to precise times. This leads to a defensive and potentially ineffective design where buffer sizes allocated for channels may be too big or too small [26]. To alleviate this problem, the SDF-AP model was proposed by [26] and formally presented in [111, 116]. SDF-AP model is a big step towards improving the SDF model [98] by incorporating *access patterns* which describe the precise clock cycles at which data production/consumption occurs, as a result SDF-AP is considered to have moved SDF closer to the hardware. Recently, Du *et al.* [147, 148] proposed a solution named "stretchable patterns" which modify the characteristics of the access patterns of the SDF-AP model actors resulting in a model that operates with no buffers. While this solution significantly optimizes the generated hardware system, it is not applicable to the integration of the off-the-shelf IP cores that are typically characterized by fixed access patterns.

An SDF-AP model $\mathcal{G} = (\mathcal{A}, \mathcal{C})$ is described as a directed graph with a set of vertices (actors) $\mathcal{A}$ interconnected to one another by a set of edges (channels) $\mathcal{C}$. At each execution (firing), an actor consumes data (represented by tokens) from one or more input channels or produces tokens onto one or more output channels. Each actor $a \in \mathcal{A}$ is a tuple (IN, OUT, ET, II), where $IN(a) \subseteq \mathcal{P}$ is a set of input ports, $OUT(a) \subseteq \mathcal{P}$ is a set of output ports with $IN(a) \cap OUT(a) = \emptyset$; $ET(a) \in \mathbb{N}$ represents *execution time* which is the time in clock cycles for an actor to complete one firing, and II is the *initiation interval* of an actor defined as the minimum interval between two successive firings of an actor a. For a set of model source actors $\mathcal{A}_{src} \subseteq \mathcal{A}$, $IN(a) = \emptyset$; and for a set of model sink actors

$\mathcal{A}_{snk} \subseteq \mathcal{A}$, $OUT(a) = \emptyset$. A channel $c = (u, v, p, q, dly) \in \mathcal{C}$ represents a FIFO buffer from source actor u (via source port $p \in OUT(u)$) to a destination actor v (via destination port $q \in IN(v)$) and dly is a *delay* and denotes the initial tokens in channel c. A port $p \in OUT(u)$ is a tuple (PR, PP) where $PR(c)$ is the *production rate* (i.e. number of tokens produced on channel c) and $PP(c)$ is the *production pattern* for output port p. In addition, port $q \in IN(v)$ is a pair (CR, CP) where $CR(c)$ is the *consumption rate* (i.e. number of tokens consumed from channel c) and $CP(c)$ is the *consumption pattern* for input port q. The access patterns $PP = \mathbb{B}^{ET}$ and $CP = \mathbb{B}^{ET}$ are a set of sequences of binary numbers with length $ET(a)$. Their function is to determine when the actor reads or writes tokens at a particular clock cycle during firing. The *i-th* element of the access pattern is denoted as $PP_{i,c}$ (resp. $CP_{i,c}$) where $i = \{0...ET(u) - 1$ (resp. $ET(v) - 1)\}$. For a given clock cycle, the element with value 1 denotes a single token read (resp. write) from (resp. to) the input (resp. output) channel. The element with value 0 represents the fact that there is no token read (resp. write) from (resp. to) the input (resp. output) channel. The number of 1's in PP and CP equal the value of PR and CR respectively. The information about $ET(a)$ and access patterns (i.e PP, CP) is obtained from a vendor-supplied IP core documentation or manual IP timing simulation results using the low level simulation tools. For a set of model source channels $\mathcal{C}_{src} \subseteq \mathcal{C}$, each channel must have $u \in \mathcal{A}_{src}$; and for a set of model sink channels $\mathcal{C}_{snk} \subseteq \mathcal{C}$, each channel must have $v \in \mathcal{A}_{snk}$.

A bounded schedule of each actor can statically be determined at compile time if one exists. Such a schedule ensures that each actor is eventually executed in order to ensure *liveness* and that the model execution is infinite using finite buffers to ensure *boundedness* of FIFOs . An *iteration*, which is a sequence with a minimum number of firings of each actor, is used to validate the above properties. It can be solved with a system of balance equations $RV(u) \times PR(u) = RV(v) \times CR(v)$ where $RV(v)$ is the number of firings for a source actor u and $RV(v)$ denotes the number of firings for a destination actor v. For a graph to be *consistent*, all the entries of a *repetition vector* (RV) must be non-zero. An example of an SDF-AP model consisting of two actors (i.e. x and y) and one channel (i.e. $c1$) is shown

in Figure 3.1. An actor x fires three times (i.e. $RV(x) = 3$) per *iteration* and executes for three clock cycles (i.e. $ET(x) = 3$). It produces two tokens at output port (p_1) with access pattern [011] which can also be represented as $[(011)^1]$ or as $[(0)^1(1)^2]$. This pattern denotes that x produces nothing on the first cycle and produces two tokens on the last two clock cycles. Generally, the access patterns specify groups with *parenthesis* and repetitions with *superscript*. The sub-pattern $(b)^n$ means the binary sequence b is replicated n times (e.g. $[1(01)^2$=10101]). In the example below, an actor y executes for five clock cycles (i.e. $ET(y) = 5$) and fires twice (i.e. $RV(y) = 2$) per *iteration*. It consumes three tokens at input port (q_1) with access pattern [10101] which can also be represented as $[(10101)^1]$ or as $[1(01)^2$=10101]. This pattern denotes the consumption of three tokens on the the first clock cycle, the the third clock cycle and the fifth cycle respectively while nothing is consumed on the second and fourth clock cycles.

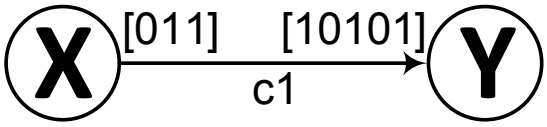


Figure 3.1: An SDF-AP model example.

3.2 Analysis of SDF-AP Model

The key properties of the model are analysed at compile time. This analysis includes checking the boundedness, the ability to avoid the deadlock, finding the schedule and computing the buffer size. A model is bounded if it can be executed infinitely using finite FIFO buffers. Like in the SDF model, the SDF-AP boundedness exists if there is a finite non-zero number of firings for each actor such that executing the model the number of times as specified in the repetition vector (i.e. RV) takes it back to its original state. The SDF-AP model is deadlock-free if each actor can fire without interruption for the number of times specified in the repetition vector. However, the deadlock-free property for an SDF-AP model is sufficient but not a necessary condition as often times the actor is fired before all the tokens are available in the input FIFO buffer.

A bounded schedule which is statically determined at compile time ensures that each actor is eventually executed (ensuring liveness) and that the execution is infinite using finite buffers (ensuring boundedness of FIFOs). To ensure that the SDF-AP graph is free from deadlock and that it has unbounded execution using a bounded buffer, the so-called Periodic Admissible Schedules (PASS) is used. The PASS defines a schedule as the sequence in which the actors must fire. An admissible schedule is the firing order that avoids a deadlock and ensures a bounded storage allocation while a periodic schedule denotes the sequence of firing repeats after every iteration [149]. The following steps outline how the PASS is created using the SDF-AP model in Figure 3.1 as an example:

- **Step 1.** First, a A topology matrix (TM) of the SDF-AP graph is created using Equation 3.1. The number of TM rows equals the graph edges (FIFO channels) while the number TM columns equal the graph nodes (actors). The entry (i, j) at *i-th* row and *j-th* column of the TM is positive if the node j produces tokens into channel i. Furthermore, the entry is negative if node j consumes tokens from channel i and the rest of the entries are filled with a value 0 to denote the absence of the edge.

$$TM = \begin{matrix} \text{x} & \text{y} \\ \begin{bmatrix} 2 & -3 \end{bmatrix} & \leftarrow \text{edge(x,y)} \end{matrix} \tag{3.1}$$

- **Step 2.** The existence of PASS is checked by determining the rank of TM which must be one less than the graph order (also known as the number model actors or graph vertices) and the proof of this theorem is provided in [98]. The rank is the number of linearly independent vectors in TM which in this case is 1.

- **Step 3.** Since the rank (i.e. =1) of TM is valid in that it is one less than the order of the graph (i.e. 2), the system has an infinite number of solutions for a firing vector RV. The simplest solution is determined with the algorithm by Bhattacharyya et al. [150] and the results are shown in

Equation 3.2.

$$RV = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \end{bmatrix} \tag{3.2}$$

In order to ensure finite buffer allocation and infinite execution, the product of TM and RV must be zero as shown in Equation 3.3.

$$TM \times RV = \begin{bmatrix} 2 & -3 \end{bmatrix} \times \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{3.3}$$

- **Step 4.** Each actor in model is then fired the number of times as specified in RV. If all the firings for each is successful, the the system is deemed deadlock-free.

The PASS schedule is followed by another schedule of a bounded SDF-AP model execution which is referred to as a *1-periodic schedule* [146]. The 1-periodic schedule is defined as $\sigma(i, j, a) = \sigma(0, 0, a) + i \cdot T + j \cdot \mu(a)$ where $\sigma(i, j, a)$ is the actor schedule at iteration index $i \in \mathbb{N}$ and actor instance index $j \in \{0...RV(a) - 1\}$, $\sigma(0, 0, a)$ is a start time (scheduling offset) of the first actor instance, iteration period (or iteration/schedule initiation interval) $T \geq RV(a) \cdot \mu(a)$, and $\mu(a)$ is the actor scheduling period (i.e. interval between successive actor instances in one iteration). The buffer computation for SDF-AP is briefly explained in [111, 116] whereby the constraint formulation is used to iteratively explore the buffer sizes for FIFO channels to the specified throughput. Wang et al. [146] generalize this approach and introduce an optimization technique that is based on Integer Linear Programming (ILP) to minimize communication buffers. In this work, it is presented in Section 3.2.2, a method to formally compute a buffer size using a 1-periodic schedule which can easily be automated in high-level synthesis tool.

Moreover, an actor is associated with the *execution pattern* (EP) which is a sequence of binary elements of an *access pattern* ($AP = \{PP, CP\}$) on the port of actor $a \in \mathcal{A}$ where it is active (i.e. firing state) and idle (i.e. non-firing state) for a duration of IL which will be explained in Section 3.2.1. The order of the EP elements is determined by a 1-periodic schedule where the actor idleness (i.e. $\sigma(i, j, a = \emptyset)$ in the schedule is denoted by 0's. The $EP_{i,p}$ (resp. $EP_{i,q}$) is used to

access the *i-th* element of EP on source (resp. sink) port q (resp. p) of channel c. For example, in Figure 3.2 the execution pattern of a source port is

$$EP_{*,p} = [\ 0 \quad 1 \quad 1 \quad \mathbf{0} \quad 0 \quad 1 \quad 1 \quad \mathbf{0} \quad 0 \quad 1 \quad 1 \quad \mathbf{0} \quad \mathbf{0}\ ]$$

and that of the sink port is

$$EP_{*,q} = [\ \mathbf{0} \quad \mathbf{0} \quad \mathbf{0} \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1\ ].$$

The asterisk ($*$) represents the whole vector EP where the individual elements are for example accessed as follows, the element of $EP_{*,p}$ at *i=1* is $EP_{1,p} = 1$ and the element of $EP_{*,q}$ at *i=1* is $EP_{1,q} = 0$. Note that the elements in bold represent locations where an actor is idle hence its neither producing nor consuming a token. A *token counter* (TC) is also defined as the sequence of length IL which represents the total number of tokens that are produced (resp. consumed) (i.e. $TC_{i,p}$ (resp. $TC_{i,q}$)) to (resp. from) the channel up to the *i-th* clock cycle. The TC computation is a trivial cumulative sum of EP and using the same example of EP vectors above, the token counters for a channel can be determined as

$$TC_{*,p} = [\ 0 \quad 1 \quad 2 \quad 2 \quad 2 \quad 3 \quad 4 \quad 4 \quad 4 \quad 5 \quad 6 \quad 6 \quad 6\ ]$$

and

$$TC_{*,q} = [\ 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 2 \quad 2 \quad 3 \quad 4 \quad 4 \quad 5 \quad 5 \quad 6\ ].$$

The elements of a TC that preceded the last element (i.e. locations with indexes less than $IL-1$ increase incrementally while the last elements (i.e. at $IL-1 = 12$) of both TC's are the equal (i.e. $EP_{IL-1,p} = EP_{IL-1,q} = EP_{12,p} = EP_{12,q} = 6$) which implies the same number of tokens produced and consumed on the channel in one iteration as determined by the system of balance equations presented in Section 3.1.

3.2.1 Iteration Latency Computation

The IL is defined as the time delay between the start of firing of the model root actor $a_r \in \mathcal{A}_{src}$ and the end of firing of the model sink actor $a_e \in \mathcal{A}_{snk}$ in one schedule iteration of the SDF-AP model. IL can be determined by Algorithm 1 whereby its procedure ComputeIterationLatency accepts graph $\mathcal{G}$ and the throughput $\tau_{\mathcal{G}}$ as its parameters. First the temporary sum is initialized to 0, followed by a traversal of all the channels ($c \in \mathcal{C}$) for a model $\mathcal{G}$. For each iteration, the cumulative sum of the sink actor initial schedule (i.e. $\sigma(0, 0, v)$) based on throughput $\tau_{\mathcal{G}}$ constraint. Finally, the IL is computed by adding $csum$, the product of repetition vector for a model sink actor less by 1 ($(RV(a_e)-1)$ and model sink actor scheduling period $\mu(a_e)$), and the execution time for a model sink actor (i.e. $ET(a_e)$). Using the example in Figure 3.1, $csum$ and $\mu(a_e)$ are computed as

$$csum = \sigma(0, 0, y) = 3, \mu(a_e) = \mu(y) = 5$$

and IL becomes

$$\begin{aligned} IL &= csum + ((RV(a_e) - 1) \times \mu(a_e)) + ET(a_e) - 1 \\ &= csum + ((RV(y) - 1) \times \mu(y)) + ET(y) - 1 \\ &= 4 + ((2 - 1) \times 5) + 5 - 1 \\ &= 13 \end{aligned}$$

3.2.2 Buffer Size Computation

In order to compute the minimal buffer size from a given throughput constraint, the 1-periodic schedule is determined as in Figure 3.2 under the throughput constraint of 6 samples per 12 cycles (i.e. $\tau = 0.5$) where $RV(x) = 3$ and $RV(y) = 2$. The rectangles are used to represent actor firings and the holes inside the rectangles are access patterns. A black hole denotes a single token consumption or production by an actor while the white whole indicates that

Algorithm 1 Compute the iteration latency (IL) for SDF-AP

Input: An SDF-AP graph $\mathcal{G}$

Input: A throughput $\tau_{\mathcal{G}}$

Result: An iteration latency IL

procedure ComputeIterationLatency($\mathcal{G}, \tau_{\mathcal{G}}$)
  $csum \leftarrow 0$
  for each channel c in $\mathcal{C}$ **do** ▷ traverse channels
    Find $\sigma(0, 0, v)$ based on $\tau_{\mathcal{G}}$
    $csum \leftarrow csum + \sigma(0, 0, v) + 1$
  end for
  Find $\mu(a_e)$ based on $\tau_{\mathcal{G}}$
  $IL \leftarrow csum + ((RV(a_e) - 1) \times \mu(a_e)) + ET(a_e) - 1$
  return IL
end procedure

token consumption or production does not occur. Each SDF-AP model iteration is represented by a sequence of actor firing with similar filled colour, hence the alternating white and shaded firing sequences correspond to individual iterations. The schedule has actor x which executes once every four clock cycles (i.e. $\mu(x) = 4$) whereas actor y executes once in five or more clock cycles ($\mu(y) \geq 5$). The initiation interval of the model execution is 12 cycles (i.e. $T = 12$), and the iteration latency is 13 cycles (i.e. $IL = 13$).

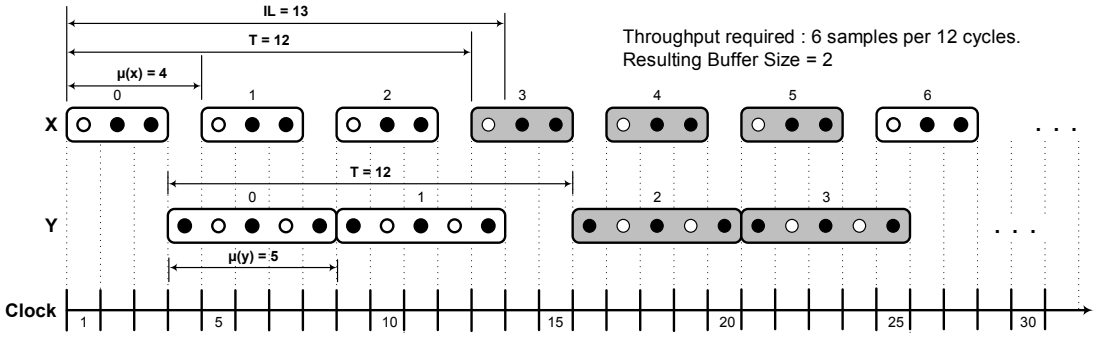


Figure 3.2: The 1-periodic scheduling of SDF-AP for example in Figure 3.1.

Given a throughput constraint, a valid buffer-size for each model channel can be computed from the 1-periodic schedule [146] model provided that the system is bounded and deadlock-free. The throughput $\tau(a)$ of an actor a is defined as the average number of firings per unit time and is determined using $\tau(a) = (RV(a) \cdot PR(c)$ or $CR(c))/T$. It can also be defined as how often the schedule σ executes, in this case, the throughput formula $\tau(\sigma) = 1/T$ is used where T is an iteration period. The maximum throughput of the SDF-AP model is only bounded by an actor with the longest execution time (ET) and calculated as

$$\tau_{max} = \frac{RV(a_e) \times CR(c_e)}{RV(a_e) \times ET_{max}(a)} = \frac{CR(c_e)}{ET_{max}(a)}$$

where a_e is a model sink actor (i.e. $a_e \in \mathcal{A}_{snk}$), c_e is a model sink channel (i.e. $c_e \in \mathcal{C}_{snk}$) and $ET_{max}(a)$ denotes the maximum execution time of the model actor (i.e. actor $a \in \mathcal{A}$ with the longest $ET(a)$). For example, the maximum throughput of SDF-AP model in Figure 3.1 is

$$\frac{CR(c1)}{ET(y)} = \frac{3}{5} = 0.6$$

The problem of buffer size computation of a bounded SDF-AP model execution under a 1-periodic schedule is addressed and a throughput constraint by implementing a buffer sizing algorithm shown in Algorithm 2. To explain this algorithm, the model example in Figure 3.1 is used. Generally, the algorithm accepts the throughput constraint $\tau_{\mathcal{G}} = 6/12 = 0.5$ of the model and returns a set $D = \{(c, 2)\}$ of channel-buffer size pairs. First the iteration period T is determined in line 2 as

$$T = \textsc{Ceil}\left(\frac{RV(a_e) \times CR(c_e)}{\tau_{\mathcal{G}}}\right) = \textsc{Ceil}\left(\frac{RV(y) \times CR(c1)}{\tau_{\mathcal{G}}}\right) = \textsc{Ceil}\left(\frac{2 \times 3}{0.5}\right) = 12$$

with respect to model sink actor $a_e \in \mathcal{A}_{snk}$ and model sink channel $c_e \in \mathcal{C}_{snk}$ (where $RV(y) = 2$ and $CR(y) = 3$). The algorithm continues iteratively (line 3) to find the channel buffer size of each channel of the SDF-AP model $\tau_{\mathcal{G}}$ where there is only one channel (i.e. $c1$) in this example. To compute the buffer size for each channel, the initial source actor scheduling period $\sigma(0, 0, u)$ is initialized

to 0 (line 4). The source actor scheduling period $\mu(u)$ remains set (line 5) to the scheduling period $\mu(v_p)$ of a sink actor from the predecessor channel if the two conditions (lines 6 and 8) of Algorithm 2 do not hold. The first condition ensures that $\mu(u)$ does not fall below the $ET(u)$ while the second one applies when the source actor of the channel (i.e. u) is also a root actor in a model (i.e. $a_r = u \in \mathcal{A}_{src}$).

Next, the algorithm determines the initial schedule of the first sink actor instance $\sigma(0, 0, v)$, calculated in line 11 as

$$\sigma(0, 0, y) = TC_{CR(c1),p} - CR(c1) + 1 = 3 - 3 + 1 = 1,$$

and this value remains unchanged as the conditions in lines 12 and 14 do not hold. The initial schedule of the second sink actor instance $\sigma(0, 1, v)$ is calculated in line 17 as

$$\sigma(0, 1, y) = TC_{(2\times CR(c1)),p} - CR(c1) + 1 = 5 - 3 + 1 = 3,$$

and a sink actor scheduling period $\mu(v)$ is computed in line 18 as

$$\mu(y) = \sigma(0, 1, y) - \sigma(0, 0, y) = 3 - 1 = 2.$$

To allocate the buffer size $\theta_{t,c}$ in a channel c at clock cycle $t \in \{0...IL\}$, the number of tokens consumed prior to t is subtracted from the sum of number of produced tokens up to t and initial delay dly. Given TC that is computed from EP as explained in Section 3.2, the vector $TC_{*,p}$ is extended to length $IL + 1$ by appending the last element in line 17 as

$$TC_{*,p} \cup TC_{IL-1,p} = [\ 0\ \ 1\ \ 2\ \ 2\ \ 2\ \ 3\ \ 4\ \ 4\ \ 4\ \ 5\ \ 6\ \ 6\ \ 6\ \ 6\ ]$$

and the vector $TC_{*,q}$ is extended to $IL + 1$ by prepending 0 in line 17 as

$$0 \cup TC_{*,q} = [\ 0\ \ 0\ \ 0\ \ 0\ \ 1\ \ 1\ \ 2\ \ 2\ \ 3\ \ 4\ \ 4\ \ 5\ \ 5\ \ 6\ ].$$

The element-wise difference (line 17) of the two vectors above becomes

$$\theta_{t,c} = [\ 0 \quad 1 \quad 2 \quad 2 \quad 1 \quad 2 \quad 2 \quad 2 \quad 1 \quad 1 \quad 2 \quad 1 \quad 1 \quad 0\ ].$$

This resulting vector $\theta_{t,c}$ contains the buffer sizes at time t over one iteration period $\forall t \in \{0...IL\}$. The optimal buffer size of a channel c is then determined by finding the maximum element of $\theta_{t,c}$ which in this case is 2. Generally, the variation in buffer size between successive throughput values largely depends on the structure of the access pattern and the average number of tokens produced and consumed over a period of iteration latency IL. The are three possibilities regarding the buffer size results of the FIFO channel/s as the throughput τ_i ($0 < \tau_i \leq \tau_{max} | i \leq N$ samples) increases; the buffer size either remains constant, increases or decreases with the increased throughput. Given the two throughput values τ_1, τ_n such that $\tau_n > \tau_1$ and their respective computed buffer sizes $\theta_{1,c}$, $\theta_{n,c}$ for channel c, the buffer size values $\theta_{k,c}$ ($0 < i \leq n-1$) computed in the range $\tau_1 \leq \tau_k \leq \tau_n$ are constant if ratio of the last elements of *token counters* (i.e. $TC_{(IL-1),p}/TC_{(IL-1),q}$) at τ_1 is respectively equal to the ratio of last elements of *token counters* at τ_n, otherwise the buffer size from τ_1 to τ_n increases or decreases. The reason for why there is an increase or decrease when throughput goes high is attributable to the access patterns as well as the source and sink scheduling periods. Calculating the impact on buffering is not a straightforward operation due to needing to know these implementation-dependent aspects on these parameters. This aspect is out of the scope of this thesis, but there is a plan to take this study on buffer size further in the future research.

Furthermore, the SDF-AP model example illustrated in Figure 3.1 is based on a simple acyclic graph whose analysis, scheduling and buffer computation are straight-forward. For a model with a cyclic graph, the same methodology for analysis, scheduling, and buffer computation can be used in the same way as for a model in Figure 3.1. However, this can only be possible on condition that the model source ($u \in \mathcal{A}_{src}$) and sink actors ($v \in \mathcal{A}_{snk}$) are not a subset of cyclic sub-graphs of the model graph. The limitation of a modeled source and sink actors that are not part of a cycle can be lifted by connecting these actors to

Algorithm 2 Compute the buffer size for SDF-AP channels

Input: An SDF-AP graph $\mathcal{G}$

Input: A throughput $\tau_{\mathcal{G}}$

Result: A set D of pairs (channel c, buffer size $\theta(c)$)

procedure ComputeBufferSize($\mathcal{G}, \tau_{\mathcal{G}}$)
  $T \leftarrow \lceil (RV(a_e) \times CR(c_e)) \div \tau_{\mathcal{G}} \rceil$ ▷ iteration period T
  for each channel c in $\mathcal{C}$ **do** ▷ traverse a set of channels $\mathcal{C}$ of graph $\mathcal{G}$
    $\sigma(0,0,u) \leftarrow 0$ ▷ source actor initial schedule
    $\mu(u) \leftarrow \mu(v_p)$ ▷ source scheduling period
    if $(T \div RV(u)) < ET(u)$ **then**
      $\mu(u) \leftarrow ET(u)$
    else if $u \in \mathcal{A}_{src}$ **then**
      $\mu(u) \leftarrow T \div RV(u)$ ▷ source scheduling period
    end if
    $\sigma(0,0,v) \leftarrow TC_{CR(c),p} - CR(c) + 1$ ▷ sink actor initial schedule
    if $\sigma(0,0,v) < 0$ **then**
      $\sigma(0,0,v) \leftarrow ET(v) - 1$
    else if $\sigma(0,0,v) = 1$ and $(\sigma(0,0,v) + RV(v)) < (\mu(u) \times RV(u))$ **then**
      $\sigma(0,0,v) \leftarrow (\mu(u) \times RV(u)) - (CR(c) \times RV(v))$
    end if
    $\sigma(0,1,v) \leftarrow TC_{(2\times CR(c)),p} - CR(c) + 1$ ▷ sink second schedule
    $\mu(v) \leftarrow \sigma(0,1,v) - \sigma(0,0,v)$ ▷ Sink scheduling period
    if $\sigma(0,1,v)) < \sigma(0,0,v)$ or $(\sigma(0,1,v) - \sigma(0,0,v)) < ET(v)$ **then**
      $\mu(v) \leftarrow ET(v)$ ▷ Sink scheduling period
    end if
    $\theta_{t,c} \leftarrow$ an element-wise difference between $(TC_{*,p} \cup TC_{IL-1,p})$ and $(0 \cup TC_{*,q})$
    $D \leftarrow D \cup (c,$ maximum element of set $\theta_{t,c})$
  end for
  return D
end procedure

virtual actors with infinite FIFOs. While this limitation aspect is out of the scope of this paper it will, however, be considered in the more complex examples

that will be presented in the future work.

3.3 Timed SDF-AP Semantics

The operational semantics of an SDF-AP model is defined by a labelled transition system N [111]. This transition system represents a model for SDF-AP model and its behaviour is easy to analyse and compare with the model for hardware. A state $s = (g, h)$ of the system is a tuple containing a vector g that describes the number of tokens in every channel and h associates to each actor a a multi-set $h(a)$ of tuples of the form $(\eta, \kappa) \in \mathbb{N}_0 \times \{w, r, \perp\}$. Each tuple $(\eta, \kappa) \in h(a)$ denotes an active actor instance where η is the number of clock cycles since the start of the execution in one iteration, and κ marks the stage of an active actor instance within the clock cycle. The stages are divided into three namely idle $\perp$, reading r and writing w. When $h(a) = \emptyset$, an actor is considered to be inactive.

A transition (s, ℓ, s') is denoted as $s \xrightarrow{\ell} s'$ where s' is a successor state of s and $\ell \in L$ is an action label which belongs to a set of labels $L = \{begin(a), end(a), tick, get(a), put(a)\}$. A transition with label $begin(a)$ denotes the beginning of firing of a newly added instance of actor a to a list of active actor instances. The removal of an instance from a list of active actor instances is marked by $end(a)$ transition when the clock counter has reached $ET(a)$. A transition $tick$ denotes one time unit lapse of the clock where the clock counter for each actor is increased by 1. The clock counter is paired with a stage (i.e. (η, κ)) allowing an actor to undergo the respective order of stages $\perp$, r , w and back to $\perp$ at the end of firing. The transition labels $get(a)$ (resp. $put(a)$) correspond a reading from (resp. writing to) input channels (resp. output channels). The preconditions for each of the labels are fully explained in [111].

A transition system for SDF-AP model in Figure 3.1 is shown in Figure 3.3 whereby the throughput constraint is set to $6/12 = 0.5$ SPC. The system starts with a *begin(x)* transition which adds an actor x instance to active instances in s_1. Note that in s_1 actor x is in idle stage as this is the beginning of firing. A tick transition updates the tick count and a read stage in s_2. This is followed by *get(x)*

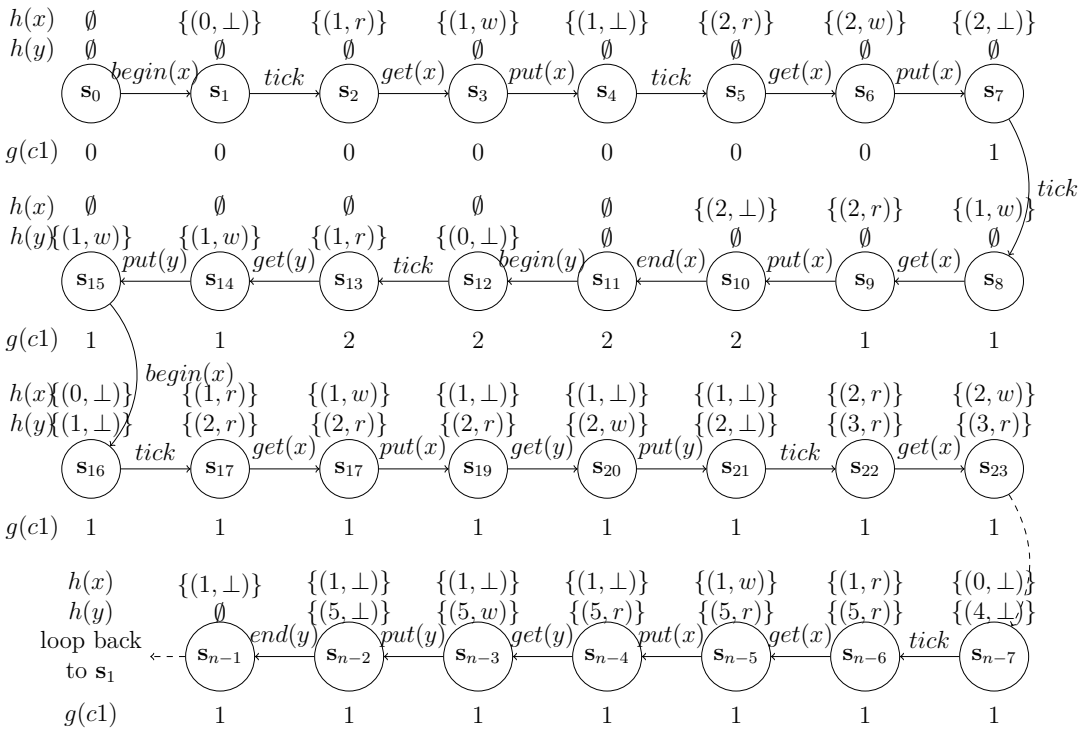


Figure 3.3: The transition system of the SDF-AP model example in Figure 3.2.

transition which leads to token count that remains unchanged as the actor x is a source actor and consumes no tokens. A *put(x)* transition does not produce a token because the output pattern *011* begins with *0*, as a result the token count does not change. The second tick transition from s_4 to s_5 increments actor x clock count to 2. The *get(x)* transition leads to no change in token count while the *put(x)* transition increases the token count to 1. An actor execution continues until it reaches *end(x)* transition which takes place when the clock transition is 2 (i.e. $ET(x) = 2$). Note the actor y only begins after 3 tick transitions and its *put(x)* transition does not change the token count as it is a sink actor, however, when it consumes a token, its reduces the current token count in channel $c1$ by 1. The broken lines between s_{23} between s_{n-7} represent the intermediate states and transitions up to the last tick (i.e. $(n-1)^{th}$) transition. s_{n-1} marks an end of execution iteration and the transition that follow leads back to s_1, most notably, s_0 only occurs once while the rest of the states are repeated infinitely.

Moreover, every channel of an SDF-AP model is constrained as a distinct *closed dataflow network* in which every sink input port is connected to a source input port [151]. This implies that the source actor only has the output ports that connect with the sink actor input ports, likewise, the sink actor only has the input ports that connect with the source actor output ports. The notion of a closed transition system is further used to simplify the analysis of the transition in that the *get(a)* (resp. *put(a)*) transitions of the source (resp. sink) actor can be dropped together with *begin(x)* and *end(x)* transitions. This only leaves the *tick*, *put* and *get* transitions where *put* and *get* transitions define token production and consumption by both source and sink respectively. The *put* and *get* transitions only occur when the corresponding access pattern of the predecessor tick transition is 1 otherwise it is not shown in the system. A succession of tick transitions with no intermediate *put* and *get* denotes idleness of both source and sink actors. Each state is labeled using a three-element vector $[s, \eta, g(c)]$ where s denotes a state number, η is the count of clock transitions during actor firing and $g(c)$ is the current token count in channel c. An example of a simplified version of a transition system in Figure 3.3 is shown in Figure 3.4.

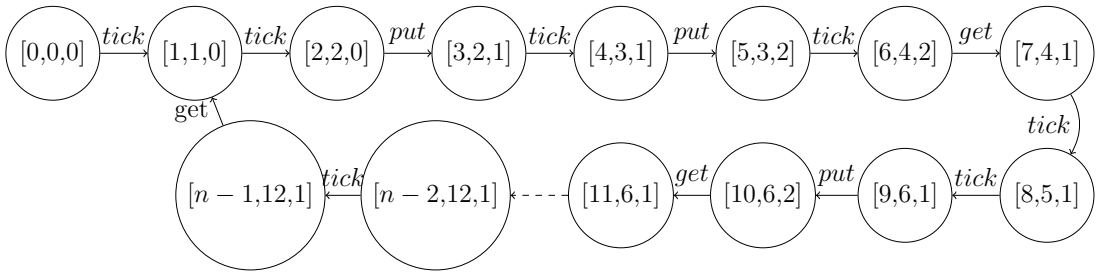


Figure 3.4: The simplified transition system of the SDF-AP model example in Figure 3.2.

The concept of the *observable behaviour* [151] of the transition system is adopted. Denoted by ρ, the observable behaviour groups a set of labels (i.e. $L = \{tick, put, get\}$) as a sequence $\alpha_0\alpha_1\alpha_2...$ such that α_i is *tick* and either or none of *put* and *get* actions. The corresponding observable behaviour for the simplified transition system in Figure 3.4 is shown below

$$\omega_{\mathsf{N}} = \mathsf{tick} \cdot \Big(\mathsf{tick} \cdot \{\mathsf{put}\} \cdot \mathsf{tick} \cdot \{\mathsf{put}\} \cdot \mathsf{tick} \cdot \{\mathsf{get}\} \cdot \mathsf{tick} \cdot \mathsf{tick} \cdot \{\mathsf{put}, \mathsf{get}\} \ \ldots \ \mathsf{tick} \cdot \{\mathsf{get}\}\Big)^{\infty}$$

where ρ^{∞} represent the infinite repetition of a sequence ρ.

3.4 Chapter Summary

This chapter has presented the preliminary concepts and semantics that are necessary for describing the operational behavior of SDF-AP which serves as the top-level system model in SdrLift. SDF-AP can capture a precise time when tokens are read and written to actor ports. This provides a benefit over other dataflow based tools which are limited in specifying how data is accessed in time and often leading to sub-optimal designs. In conclusion, the chapter is summarized as the analysis of the SDF-AP, system validation and the computation of key system properties that include execution schedule, buffer size and latency size given a throughput constraint.

Chapter 4

SdrLift: A Compiler Framework for Software-Defined Radios[1]

This chapter presents SdrLift, a domain-specific intermediate-level compiler framework for prototyping SDR applications on an FPGA. SdrLift facilitates development by automating the synthesis of FPGA data-paths from structural descriptions using the template-based design approach and through the integration of pre-defined IP cores. Applications are the composition of actors that exchange data through unidirectional channels in a dataflow network. The behavior of each actor is described in an intermediate-level language that is designed around topological patterns and computational data patterns, resulting in a high-level of design abstraction. This chapter focuses on the constructs of the intermediate language namely SdrLift language and the construction of dataflow actors which represent the custom DSP IP cores in resultant hardware design. Additionally,

[1]This chapter is based in part upon the following publications:
L. J. Tsoeunyane, S. Winberg, and M. Inggs, "An IP core integration tool-flow for prototyping software-defined radios using static dataflow with access patterns," in *2017 International Conference on Field Programmable Technology (ICFPT)*, pp. 88–95, Dec 2017.
L. Tsoeunyane, S. Winberg, and M. Inggs, "Automatic configurable hardware code generation for software-defined radios," *Computers*, vol. 7, no. 4, 2018.
L. Tsoeunyane, S. Winberg, and M. Inggs, "Sdrlift: An intermediate-level framework for synthesis of software-defined radio accelerators," in *2019 IEEE 10th International Conference on Mechanical and Intelligent Manufacturing Technologies (ICMIMT)*, pp. 166–173, Feb 2019.

the methodology in this work allows the integration of existing IP cores which are typically created and added to the reusable IP library of cores by hardware experts. To maintain the high performance of the SDR designs, both the new DSP cores and integrated library IP cores are stitched together to create new SDR applications using characteristics of the SDF-AP model. Throughout this chapter, each step of the compilation flow is illustrated with examples. The remainder of the chapter proceeds with the description of the SdrLift language in Section 4.1. Following this is the detail on the design and implementation of the SdrLift compiler framework in Section 4.2.

4.1 SdrLift language

This section presents SdrLift language, a domain-specific intermediate-level language that serves as an entry-point to the SdrLift compiler framework. Instead of building a new language from scratch, SdrLift language constructs are embedded within Scala programming language [152]. The name Scala stands for "scalable languageâĂİ because it was built to easily grow in proportion to the increasing demands of its users and it is used widely in writing small scripts to build large systems [153, 154, 155]. Technically, Scala combines the concepts of object-oriented and functional programming in a statically typed language. The object-oriented approach is suitable for building larger systems and makes software reuse very easy while the functional programming builds programs using pure functions, that is, functions without any side effects. This combination of object-oriented and functional programming in Scala results in a really powerful programming methodology with constructs that can be used to express new kinds of programming patterns and component abstractions with a concise programming style [153, 154, 155]. The Scala language is used to implement SdrLift for the following several reasons:

- Scala is a novel programming language that incorporates functional and object-oriented programming concepts.
- It is designed to form a base for the generation of new domain-specific

languages.

- It enables higher levels of abstraction through high-order functions, polymorphism, paremeterizations of system designs and helps the compiler to capture and translate the high-level constructs into rich IR with inherent parallelism.
- It compiles into Java Virtual Machine (JVM), hence it is compatible with existing Java frameworks.

The SdrLift language is designed around the structural description of new HW Blocks and the integration of existing HW Blocks for prototyping SDR applications. The SDR application under design is converted into IR that represents a structural graph in which individual nodes are hardware components. This IR has the following benefits (i) it provides an abstraction between the compiler model and the high-level program descriptions, (ii) it decouples the compiler model representing the program from the low-level RTL generation [156]. The SdrLift language syntax is presented in Figure 4.1 and is made up of the high-level functional constructs. More importantly, these specialized constructs provide a language that SDR subject-matter experts can use to more easily describe and maintain designs and to reason about solutions, as opposed to using the standard imperative language code that is more convoluted and difficult to maintain. The SdrLift program comprises functions (*func-def*), components (*comp-def*), templates (*temp-def*), modules (*mod-def*) and macros (*macro-def*) definitions. Other constructs of the program include expressions, types, and patterns. Patterns are rich constructs which are classified into topological data patterns and design data pattern are generally used to build the IR graph of the compiler with very high expressive power. Below are the structures that compose the SdrLift language.

4.1.1 Expressions

The most basic expression in SdrLift language is a variable which can be in various forms: a function parameter, a component parameter, a template pa-

program ::=	*func-def*	
|	*comp-def*	
|	*temp-def*	
|	*mod-def*	
|	*mac-def*	
func-def ::=	**def** *fid* vid^* = *expr*	
comp-def ::=	**case class** *cid* vid^* **extends** Component = *expr*	
temp-def ::=	**case class** *tid* vid^* **extends** Template = *expr*	
mod-def ::=	**case class** *mid* vid^* **extends** Module = *expr*	
mac-def ::=	**case class** *kid* vid^* **extends** Macro = *expr*	
expr ::=	*vid*	▷ Variable
|	*fid* vid^*	▷ Function call
|	*cid* vid^*	▷ Component call
|	*tid* vid^*	▷ Template call
|	*mid* vid^*	▷ Module call
|	*kid* vid^*	▷ Macro call
|	*tp* cid^* tid^* mid^* kid^*	▷ Topological pattern call
|	*dp* cid^* tid^* mid^* kid^*	▷ Design pattern call
|	**val** *vid* = *expr*	▷ Variable binding
type ::=	**Int**	▷ Finite integer type
|	**String**	▷ String type
|	*Streamer*	▷ Streamer type
|	*Constant*	▷ Constant type
|	*Component*	▷ Component type
|	*Template*	▷ Template type
|	*Module*	▷ Module type
|	*Macro*	▷ Macro type
tp ::=	*Chain*	▷ Chain topological pattern
|	*Merge*	▷ Merge topological pattern
|	*Broadcast*	▷ Broadcast topological pattern
dp ::=	*Reduce*	▷ Reduce data pattern
|	*ZipWith*	▷ Zip-with data pattern
|	*FoldR*	▷ Fold-right data pattern
vid ::=	Variable identifier	
fid ::=	Function identifier	
cid ::=	Component identifier	
tid ::=	Template identifier	
mid ::=	Module identifier	
kid ::=	Macro identifier	

Figure 4.1: The syntax of SdrLift intermediate language

rameter, a module parameter, a macro parameter, and a local name bound by *val* construct. All the calls of the following types: function call, component call, template call, and macro call require one or more arguments (represented as vid^*) which must all be variables. The arity, which refers to the number of arguments to a function/component/template/macro must correspond to the function type as inferred from its definition. The *val* construct binds one or more local variables to expressions. The expressions represent the structural descriptions of graphs in which the vertices are HW Blocks and the edges orchestrate the data movement between these blocks. For this reason, *val* construct is an immutable reference to the expressions that can easily be converted into models that better represent the hardware.

4.1.2 Types

The values of SdrLift language are of type finite *integer*, *string*, *streamer*, *constant*, *component*, *template*, *module*, and *macro*. The primitive types namely *string* and *integer* are used to define the fixed parameters of the HW Blocks defined as components or templates. The *streamer* is used to define a vector of bits which is kept in registers or flows in wires of connected logic elements. This *streamer* type is a fixed-point number which can either be signed or unsigned and is associated with std_logic_vector type of VHDL. The bits of a *streamer* can be accessed as a slice of bits or as individual bits as shown in Listing 1. Line 1 declares a data streamer with width 8, Line 2 accesses the second Most Significant Bit (MSB) of data and Line 3 extracts the lower nibble of data.

```
1 val data = Streamer("data", 8)
2 val bit_1 = data.bitAt(1)
3 val r_nible = data.slice(3, 0)
```

Listing 1: An example showing how to declare a streamer, how its bit 1 is accessed and how a lower nible of a streamer is sliced.

Moreover, the *constant* type implicitly defines the literal expressions of integers or strings which are represented in hardware as the register that keeps constant

values. The *component* type defines a basic, fine-grained block HW Block using a graph of primitive elements, hence it forms a basis for the hierarchical construction of other complex and coarse-grained blocks. The *template* is similar to the *component* but serves as a placeholder for pre-defined template HDL components. To build complex HW Blocks which are also known as IP cores, the *module* type is used. The *macro* type facilitates the integration of existing IP blocks.

4.1.3 Topological Patterns

SdrLift adopts a concept of the topological pattern (*tp*) [157] for concise specification of regular functional SDR structures that can easily be converted to the compiler model. The topological patterns not only enable the scalability of the model substructures but also explicitly expose the inherent graph to the underlying intermediate representation. In this context, the topological patterns are applied in building the graphs for DFG and SDF-AP model. Note that the vertices and edges are used to generalizes the vertices of both graphs in the description provided in this thesis. For a DFG model, a topological pattern is a mapping that, when applied to a finite sequence of ordered nodes (vertices) it forms a relation of two nodes and in turn builds a structure of inter-node relationships via data-dependencies (edges). One or more patterns can also relate to form complex structures. Similarly, for an SDF-AP model, the relationships are FIFO channels (edges) that connect the actors (vertices) to one another. The topological patterns help to exploit the parallelism of different computing problems by using multiple vertices with one or more stream sources. These vertices are interconnected to perform continuous processing on the input stream(s) resulting in one or more output streams. Note that each vertex in the topological pattern is regarded as a pure function, which implies that the parallel functions can be computed without any side effects. The three topological patterns used in SdrLift language: *Chain*, *Merge* and *Broadcast* are summarized in Tables 4.1 and are further described below.

- **Chain**(verts)

The *Chain* pattern processes a stream of data using a cascade of vertices in n stages. The first stage processes the input stream, and for the succeeding stages, each stage processes the stream results of the previous stage.
verts: a sequence of n vertices.

- **Merge**(verts)(snk)
  The *Merge* pattern processes $n-1$ parallel streams of data which are respectively received by $n-1$ source vertices. The output results of each of the source vertices are input to a sink vertex which outputs one or more data streams.
  verts: a sequence of $n-1$ source vertices.
  snk: a sink vertex.

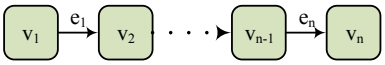
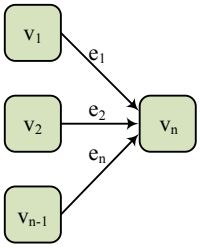
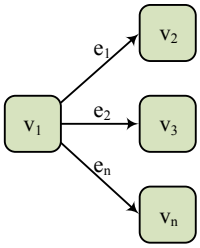
- **Broadcast**(src)(verts)
  The *Broadcast* pattern processes a stream of data which is received by a single source component. The source stream outputs $n-1$ parallel output streams which are respectively drained to $n-1$ sink vertices. Each of the sink vertices outputs one or more streams of data. **src:** a source vertex.

4.1.4 Data Patterns

The data pattern (*tp*) type performs operations using arithmetic or logical elements on one or more stream sources which result in one continuous stream of data. The input streams are sourced from HW Blocks of types *streamer*, *constant*, *component* and *template*. The data patterns used in SdrLift language are *Reduce*, *ZipWith*, and *FoldR* as described below:

- **Reduce**(seq)(op)
  The *Reduce* pattern uses a associative arithmetic/logical block tree to combine multiple stream sources into a single stream.
  seq is a sequence of stream sources
  op is the arithmetic vertex.

Table 4.1: The topological patterns

Topological Pattern	Name	Description
	chain	The *chain* $TP_C = (V, E)$ pattern connects an ordered sequence of $n \in \mathbb{Z}_{\geq 2}$ vertices. It relates a vertex v_i to vertex v_{i+1} using edge $e_j \in E$ where $1 \leq i \leq n-1$ and $1 \leq j \leq n-1$.
	merge	The *merge* $TP_M = (V, E)$ pattern connects the first $n-1$ vertices to the last component in an ordered sequence of vertexs. It therefore relates each vertex v_i to vertex v_n using $e_j \in E$ where $1 \leq i \leq n-1$ and $1 \leq j \leq n-1$.
	broadcast	The *broadcast* $TP_B = (V, E)$ pattern connects the first vertex to every other vertex in an ordered sequence of vertices. It relates the first vertex v_1 to every component v_i using $e_j \in E$ where $2 \leq i \leq n$ and $1 \leq j \leq n-1$.

- **ZipWith**(seq)(seq)
  The *ZipWith* pattern works by applying the same arithmetic operation or function to multiple streams which results in a new set of streams.
  seq is a sequence of stream sources.
  op is the arithmetic vertex.

- **FoldR**(seq)(op)
  The *FoldR* pattern uses a reduction tree to aggregate multiple stream sources into a single stream using associative arithmetic operation.
  seq is a sequence of stream sources
  op is the arithmetic vertex.

4.2 Compiler Framework

The SdrLift compiler infrastructure is implemented as depicted in Fig. 4.2 by embedding it in Scala thereby leveraging its type system. An entry-point to the compiler framework is a Scala-based intermediate-level program written in SdrLift language presented in Section 4.1. SdrLift language captures the system specifications using high-level constructs, topological patterns and parallel data patterns with high expressive power. It is an intermediate language that is designed to be used by domain experts to prototype SDR applications and it can serve as a back-end IR to generate efficient hardware accelerators. It is assumed that before the intermediate program specification, the higher-level optimizations have been applied. These optimizations include Sub-Expression Elimination (CSE), Dead-Code Elimination (DCE) and code motion.

The compilation flow entails transformations into multiple IR levels. Using the multiple IR-formats to define the back-end of the compiler leads to a cost-effective compiler development and reduced compiler design effort. George et al. [90] has further demonstrated the effectiveness of this approach by organizing the IR-formats into standalone optimization modules which can easily be analyzed and optimized individually. For this design, the intermediate program is first converted into a IR_{FG} which is based on a DFG. The DFG is divided into multiple DFGs where each DFG implements a single computational element which in this work is called a *Component*. A *Component* implements the basic computation and is built from custom program specifications and parameterizable templates.

The IR_{FG} results in the fine-grained components, followed by transformation into IR_{CG} which is also implemented around a DFG. IR_{CG} connects up the components according to specifications of topological and data patterns to form coarse-grained computational elements which in this project is referred to to as *kernels*. These kernels are classified into two, *custom* and *black-box* kernels. A custom kernel is an equivalent of the IP core and is composed of components. The black-box kernel allows the integration of an existing IP core into the compilation flow by simply specifying the interface.

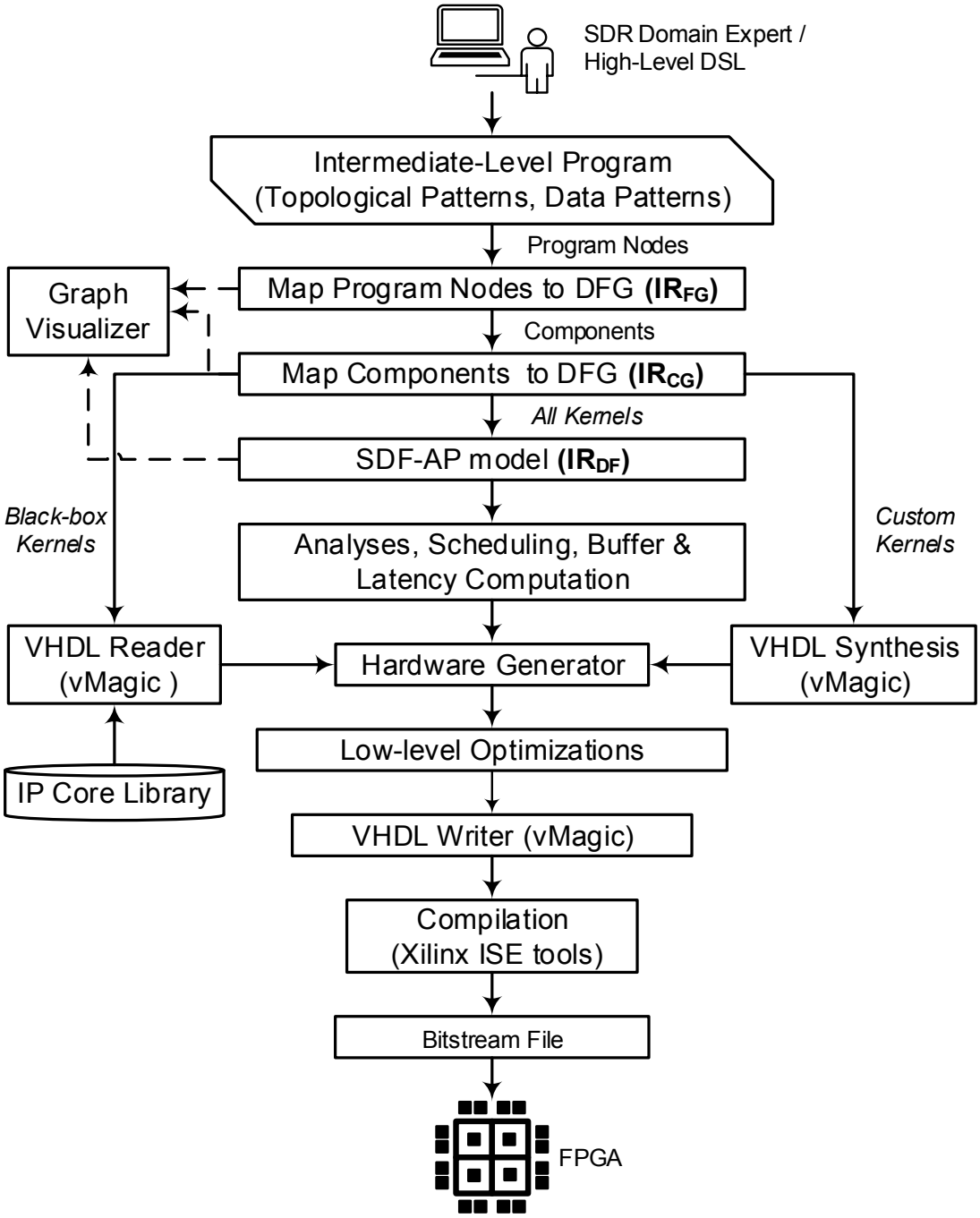


Figure 4.2: The SdrLift compiler infrastructure.

The IR_{DF} comprises the kernels which compose a dataflow network that is based on the SDF-AP model. The SDF-AP model provides automated analyses of

the kernels and is also key to validating the application using dataflow model semantics. Following the SDF-AP analyses is the computation of the system properties that include throughput, FIFO channel(s) buffer size and component compatibility.

Moreover, the **graph visualizer** produces visual representations of the application structure at different levels of IR (i.e. IR_{FG}, IR_{CG} and IR_{DF}) in *Dot* language that belongs to *Graphviz* suite of tools [158] as illustrated in Sections 4.2.2, 4.2.3, 4.2.4. To implement the graphs for IR_{FG}, IR_{CG} and IR_{DF}, the *Graph for Scala* (scala-graph) library [32] is used. Scala-graph provides a basis for graph applications and fits seamlessly into the Scala standard collections library.

After successful IR transformation passes, the **VHDL Generator** automatically generates VHDL code. The vMagic [33] library is used to implement the VHDL code for custom kernels built by IR_{DF}, to read the VHDL code for the existing IP cores, and to stitch-up the coarse-grained kernels in VHDL at the top-level of the application design. This is followed by HDL compilation and FPGA bitstream file creation using Xilinx ISE tools. This part of the compilation is automated by a series of commands in a Tool Command Language (TCL) script called by the SdrLift compiler.

4.2.1 Template-based Design

In order to separate the domain knowledge from the complex low-level design, parameterizable templates are used to implement the efficient hardware design from algorithmic-level specifications. These templates are primarily developed by designers who are well equipped with skills in domain design, hardware design, and HLS design. The domain designers can select from a library of templates that are later translated to corresponding HW Blocks by HLS to suit the designer's needs. Hence the parameterizable template is used as the basic building block from which new templates can be implemented. The design approach in this work takes inspiration from template-based design from Matai et al.[159], defines how

hardware designs can be parameterized and combined into more complex ones.

A template $t \in \mathcal{T}$ is described as a tuple $t = (P^t, ET, n)$, where $\mathcal{T}$ is the set of templates, P^t is the set of template ports, ET is the template execution time, and n being a node associated with the template. A template of node n is denoted by t_n. $P^t = P^t_i \cup P^t_o$ and $P^t_i \cap P^t_o = \emptyset$, P^t_i are a set of template input ports and P^t_o a set of template output ports. A port $p^t = (d, i, AP) \in P^t$ is a tuple with a direction $d \in \{INPUT, OUTPUT\}$ and an interface $i \in I^t$, I^t being a set of interfaces applicable to ports. $AP = (CP, PP)$ is the access pattern that is associated with the port and like in the SDF-AP model, the access patterns play an instrumental role of specifying the exact times (clock cycles) at which the data samples are produced (resp. consumed) by the templates to (resp. from) the output (resp. input) channels via the output (resp. input) ports. More details on access patterns are discussed in Section 3.1.

The templates used during the hardware synthesis are associated with different levels of compiler IRs. The IRs are used by the compiler framework to represent the SdrLift language in multi-level formats that are easy to validate, analyze and optimize. Table 4.2 summarizes the templates as found at various levels of the IR formats. These IR formats, also known as SdrLift IR, are classified into IR_{FG}, IR_{CG}, and IR_{DF}. IR_{FG} and IR_{CG} are based on a DFG while IR_{DF} is built around the SDF-AP model. An IR node represents a template with design and model parameters that enable different automated optimizations and analyses. The model parameters include the ET, the model rates PR and CR, and the access patterns PP and CP.

The IR_{FG} comprises primitive templates that perform single-cycle basic operations such as arithmetic, logic, and multiplexers. Consequently, the model parameters for these templates are all 1. IR_{CG} consists of templates that perform complex multi-cycle operations such as Register (REG), Zeropad Insert (ZP-I), Zeropad Removal (ZP-R), Cyclic-prefix Insert (CP-I), and Cyclic-prefix Remove (CP-R) which are all very common in many SDR applications. The model parameters for IR_{CG} are computed from the design parameters as shown in Table 4.2. An artefact of IR_{CG} is module for which the model parameters

can be computed from the combination of template model parameter in SdrLift compiler as detailed in Section 4.2.3. Interesting is how the model parameters of the REG template largely depend on the delay that is measured in cycles that are introduced in the data-path from input ports to the output ports. Hence this characteristic of the REG template enables computation of more complex designs using total delay found in the data-path. Furthermore, IR_{FG} contains component templates that correspond to *CompNode* and are custom-made as per designer needs.

IR_{DF} builds on the previous work on the SDF-AP model which captures the key application properties as well as facilitating the glue design of the newly developed DSP cores and the existing IP cores to generate efficient implementations [26, 111, 116, 146, 28, 34]. The integration of the IP cores is automated using the SDF-AP model where the model parameters are derived through simulation of the functional behaviour of each IP core and reading relating data-sheets. The simulation is often performed manually using low-level third-party tool-flows such as Xilinx ISim. The IP cores are regarded by SdrLift as Macros and are represented as nodes of type *MacroNode*. Manually performing complex low-level simulations can itself be tedious and time-consuming. This delay can be avoided by using the module templates which are the custom IP cores as viewed by the SdrLift compiler.

The IR-formats for SdrLift (i.e. IR_{FG}, IR_{CG} and IR_{DF}) are detailed below whereby their graphs can be built using the topological and design patterns described in Section 4.1.3 and Section 4.1.4 respectively.

4.2.2 Fine Grained IR

The most basic view of the IR nodes are used to build the IR_{FG}. $IR_{FG} = (N, E)$ is centered around a DFG where N represents IR nodes that are connected to one another by their data dependencies E. The node types that compose IR_{FG} include *StrmNode*, *ConstNode*, *ArithNode*, *LogiNode* and are described in Table 4.3. Moreover, IR_{FG} can be viewed as a component that will be used as

Table 4.2: Description of templates in SdrLift and access pattern computation for each template. (Design Parameters: W=Data Width, S=Select Width, D=Register Depth, L_i=Input sample length, L_o=Output sample length, L_{pad}=Pad length, L_{pref}=Prefix length, L_{par}=Parallel input/output lines)

IR Type	IR Node	Template	Description	Design Parameters	Model Parameters
IR_{FG}	ArithNode	+, −, ∗, /	Arithmetic operations	W	$ET = CP = PP = 1$
		MUX	Multiplexer	W, S	$ET = CP = PP = 1$
	LogiNode	<, >, <=, >=, !, ==, ! =	Logic operations	W	$ET = CP = PP = 1$
IR_{CG}	TmplNode	REG	A pipeline register or a shift register.	W, D, L_i, L_o	$ET = L_i + D$ $PR = CR = L_i = L_o$ $CP = (1)^{L_i}(0)^{D}$ $PP = (0)^{D}(1)^{L_i}$
		ZP	Zero-pad	W, L_i, L_{pad}	$ET = L_i + L_{pad} + 1$ $CR = L_i$ $PR = L_i + L_{pad}$ $CP = (1)^{L_i}(0)^{L_{pad}+1}$ $PP = (0)^{1}(1)^{L_i+L_{pad}}$
		ZT	Zero-truncate	W, L_o, L_{pad}	$ET = L_o + L_{pad} + 1$ $CR = L_o + L_{pad}$ $PR = L_o$ $CP = (1)^{L_o+L_{pad}}(0)^{1}$ $PP = (0)^{1}(1)^{L_o}(0)^{L_{pad}}$
		CP-A	Cyclic-prefix add	W, L_i, L_{pref}	$ET = 2 \times L_i + 1$ $CR = L_i$ $PR = L_i + L_{pref}$ $CP = (1)^{L_i}(0)^{L_i+1}$ $PP = (0)^{(L_i+1)-L_{pref}}(1)^{L_i+L_{pref}}$
		CP-R	Cyclic-prefix remove	W, L_o, L_{pref}	$ET = L_o + L_{pref} + 1$ $CR = L_o + L_{pref}$ $PR = L_o$ $CP = (1)^{L_o+L_{pref}}(0)^{1}$ $PP = (0)(L_{pref} + 1)(1)^{L_o}$
		S-P	Serial-to-parallel	W, L_i, L_{par}	$ET = (L_i \div L_{par}) \times (L_{par} + 1)$ $CR = L_i$ $PR = L_i \div L_{par}$ $CP = ((1)^{L_{par}}(0)^{1})^{L_i \div L_{par}}$ $PP = ((0)^{L_{par}}(1)^{1})^{L_i \div L_{par}}$
		P-R	Parallel-to-serial	W, L_i, L_{par}	$ET = (L_i \div L_{par}) \times (L_{par} + 1)$ $CR = L_i$ $PR = L_i \div L_{par}$ $CP = ((1)^{1}(0)^{L_{par}})^{L_i \div L_{par}}$ $PP = ((0)^{1}(1)^{L_{par}})^{L_i \div L_{par}}$

node in a IR_{CG} thereby enabling hierarchical composition of DFGs. The ultimate representation of IR_{FG} in hardware design is a behavioural entity described in VHDL.

Fugure 4.4 shows the IR_{FG} DFG of a complex multiplier which accepts four input

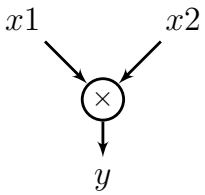


Figure 4.3: A DFG of a mixer.

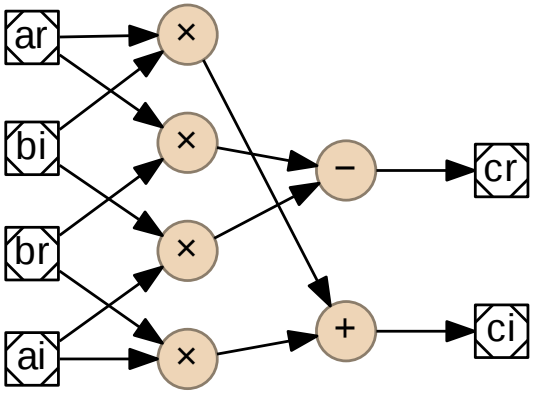


Figure 4.4: A DFG of a complex multiplier.

nodes (*ar*, *ai*, *br* and *bi*) of *StrmNode* type and performs the operations using three arithmetic nodes ("×", "+" and "−") to produce output nodes (*cr* and *ci*). Listing 2 is an example description in SdrLift for the DFG of Fig 4.4. It defines a complex multiplier as a basic logic unit, called *ComplexMult* which inherits from a *Component* class. The *ComplexMult* passes an instance (*inst*) parameter and the data width (*a_width*, *b_width* and *c_width*) parameters for complex ports (*ai/ar*, *bi/br* and *ci/cr*) respectively. Line 2 declares input nodes and Line 3 declares the output node. The *Combinational* directive encloses an arithmetic nodes which combinationally describe a complex multiplier of *ai/ar* and *bi/br* to produce outputs *cr/ci* in Lines 9–14. Following this is the listing of arithmetic expressions (mapped to arithmetic nodes) using ":=" operator and the links that connect the arithmetic nodes to the output nodes using "∼>" operator in Line 16. The *name* in Line 19 specifies the name of the VHDL entity after code generation while the *dfg* in Line 21 implements the actual DFG of the complex multiplier using the nodes found in the *Combinational* directive.

```
case class ComplexMult(inst: String, a_width: Int, b_width: Int, c_width: Int)
  ↪  extends Component {
  //inputs nodes
  val (ar, ai, br, bi) = (Streamer("ar", a_width), Streamer("ai", a_width),
   ↪  Streamer("br", b_width), Streamer("bi", b_width))
  // output nodes
  val (cr, ci) = (Streamer("cr", c_width), Streamer("ci", c_width))
  // combinational logic
  val cmb = Combinational {
        // arithmetic nodes
        val ar_br = ar * br
        val ai_bi = ai * bi
        val ai_br = ai * br
        val ar_bi = ar * bi
        val ar_br_ai_bi_sub = ar_br - ai_bi
        val ai_br_ar_bi_sum = ai_br + ar_bi
        // return registers and output ports
        :=(ar_br_ai_bi_sub, ai_br_ar_bi_sum, cr, ci, ar_br_ai_bi_sub ~> cr,
         ↪  ai_br_ar_bi_sum ~> ci)
  }
  // component name
  override val name = "complexMult"
  // directed flow graph (DFG)
  override def dfg = model(Seq(cmb))
}
```

Listing 2: SdrLift code for a complex multiplier component represented as DFG in Fugure 4.4

4.2.3 Coarse Grained IR

The IR_{CG} is implemented with IR_{FG} which instead of being viewed as a DFG, is now considered as a coarse-grained node. $IR_{CG} = (N, E)$ is a DFG where N represents both the basic nodes and coarse-grained nodes that are connected to one another by their data dependencies E. The coarse-grained nodes are of type *CompNode* and *TmplNode* a summarized in Table 4.3. *CompNode* represents a component build as IR_{FG} and the *TmplNode* represents the parameterizable template which is pre-defined in SdrLift. The templates (e.g. Delay, FIFO, Multiplexer, LUT, Counter, etc.) are used to capture different types of memory access patterns and to naturally express hardware parallelism which may be too complex to describe in SdrLift language. These templates are highly optimized

Table 4.3: Description of the IR nodes.

IR Type	IR Node and Description	Node Parameters
IR_{FG}	**StrmNode(stm):** The *StrmNode* node internally represents the *Streamer*.	**stm:** the *Streamer* object which corresponds to the register or the port of two's complement fixed-point format.
	ConstNode(value): The *ConstNode* represents the constant integer value that is implicitly defined in the component.	**vl:** the *Const* object with an integer value.
	ArithNode(name, lhs, rhs, width, opr): This is used for the following primitive arithmetic operations: $+$, $-$, $*$, $/$.	**name:** the name of the node which is generated by the compiler based on *lhs* and *rhs*. **lhs:** the first operand. **rhs:** the second operand. **width:** the width as computer by the compiler. **opr:** an arithmetic operator.
	LogiNode(name, lhs, rhs, width, opr): This is used for the following primitive logical operations: $<$, $>$, $<=$, $>=$, !, $==$, $!=$.	**name:** the name of the node which is generated by the compiler based on *lhs* and *rhs*. **lhs:** the first operand. **rhs:** the second operand. **width:** the width as computer by the compiler. **opr:** an logical operator. **comp:** the *Component* object
IR_{CG}	**CompNode(cmp):** The *CompNode* represents the custom kernels which are defined as components in SdrLift.	**comp:** the *Component* object
	TmplNode(cmp): The *TmplNode* represents the predefined parameterizable templates which are easily called in SdrLift Language.	**cmp:** the *Component* object
IR_{DF}	**ModNode(cmp):** describes the custom kernel which is mapped to a custom IP core in VHDL.	**cmp:** the *Component* object
	MacroNode(cmp): is a black-box representation of the existing IP core.	**cmp:** the *Component* object

and they can be easily expressed with the high-level patterns and have parameters that can easily be varied. The IR_{CG} is regarded to be the module that is translated into synthetic IP core (i.e. HW Block) in VHDL.

The implementation of a module is accompanied by a computation of the access patterns which are instrumental in building the SDF-AP model that represents the SDR system. These patterns are determined from the templates of nodes that make up a structure of the module. The Algorithm 3 shows how the ComputePatterns function that determines the access patterns of a module given the parameters of IR_{CG} graph, the $AP = (1, 1)$, and Path Delay (PD) which is the delay incurred in a path from the start node to the current node. This delay can is obtained by adding up the *depth* (D) of REG nodes in the path being

traversed.

The ComputePatterns is a recursive function that traverses the nodes of IR_{CG} and determines the access patterns based on the node type. For a node type of *TmplNode*, the algorithm continues to check the template type. If the template type is REG, the access patterns are computed as in *line 9* and using the rates and the PD. Note that the CR is obtained from the L_i of REG while PR is obtained from the L_o of REG. The D becomes is the sum of the previous PD and the D of the current REG node. For any other template that not of type REG, the access patterns are computed using the corresponding formula provided in Table 4.2. Moreover, the node of type *CompNode* triggers a recursive operation of the function based on whether the current node is part of the cycle. If the cycle exists, the access patterns for all cycle nodes are computed recursively or else the computation occurs on the nodes that form the node. If the nodes are neither of type *TmplNode* nor *CompNode*, the access patterns remain unchanged.

Consider the mathematical definition of a FIR filter of rank 4 in Equation 4.1:

$$y[n] = \sum_{i=0}^{3} h[i] \cdot x[n-i] \tag{4.1}$$

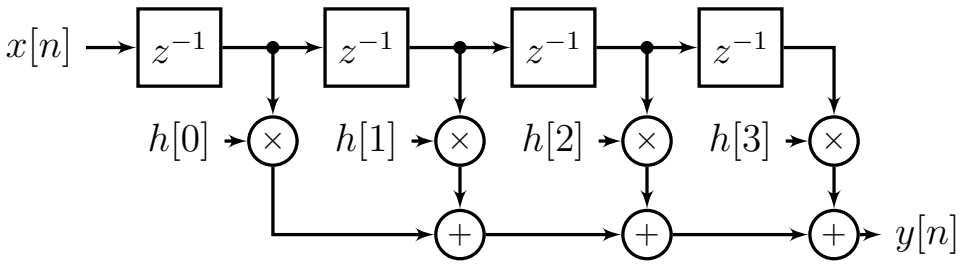


Figure 4.5: The hardware architecture of the direct form FIR filter.

This description can easily be implemented in software, however, it is inconvenient for hardware realization. The direct form implementation in hardware for the FIR filter can be directly obtained from Equation 4.1. The hardware architecture as shown in Figure 4.5, is obtained by decomposing the direct form formula into delayed inputs which are scaled by factors producing partial products that are in turn summed into the output. The SdrLift implementation of the same

Algorithm 3 Computation of Access Patterns for a Module

Input: The IR_{CG} graph
Result: A tuple (CP, PP)

```
1: procedure ComputePatterns(IR_CG, AP, PD)
2:     for each node v in IR_CG do
3:         if v is type TmplNode then
4:             Set t_v = a template of v
5:             if t_v is REG then
6:                 CR = REG.L_i
7:                 PR = REG.L_o
8:                 D = PD + REG.D
9:                 return ([(1)^CR (0)^D], [(0)^D (1)^PR])
10:            else
11:                return (CP of t_v, PP of t_v))
12:            end if
13:        else if v is type CompNode then
14:            G_cycle = (V_c, E_c) = FindCycleContaining(v)
15:            if v_c ≠ ∅ then
16:                return ComputePatterns(G_cycle, AP)
17:            else
18:                return ComputePatterns(G_comp, AP)
19:            end if
20:        else
21:            return (CP(AP), PP(AP))
22:        end if
23:    end for
24: end procedure
```

FIR hardware architecture entails the exploitation of the four replicated structure portions each representing a tap that is composed of three nodes namely the unit delay, the multiplier, and the adder. The FIR filter is represented as the IR_{CG} in Figure 4.6 and is composed of four delay nodes of *TmplNode*, four coefficient constants of *ConstNode* type, seven arithmetic nodes (4 multiplier nodes

and 3 adder nodes), and two-port nodes of *StrmNode* type (i.e. x and y). The FIR DFG as generated in SdrLift in Figure 4.6 is similar to hardware structure in Figure 4.5.

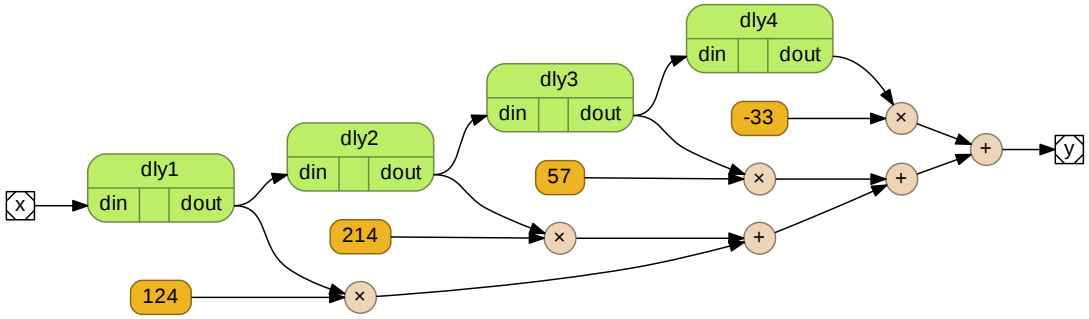


Figure 4.6: A DFG of the direct form FIR filter.

Listing 3 is an example description in SdrLift for DFG of Fugure 4.6. It implements a *custom kernel* (i.e. *ModNode* type) called a *DFIR* which passes instance (*inst*) and data width (w) parameters. Line 3 declares the input *Streamer* x and output *Streamer* y where the width of each is set to 8 bits in the example (i.e. $w = 8$). Line 5 declares the four delay templates where each has the width of 8 and the depth of 1. Note that a delay template corresponds to predefined parameterizable component in SdrLift with an input data port *din* and an output data port *dout* where the width of both ports can be varied. A cascade of four connected delays is formed in Line 9 using a *Chain* topological pattern. The *Chain* pattern is able to automate the links between components if each component has exactly one input port *DIN* and/or one output port *DOUT*. Alternatively, the *outLinks* function can be used to specify which data port(s) connects to the data port(s) of the successive component. This method applies to a case of creating links that connect to/from multiple ports of either *DIN* or *DOUT* type. The multiplication of delay outputs (with coefficients obtained from [56]) is performed with a *ZipWith* data pattern in Line 15. The filter coefficients $H[z] = \{0.48301, 0.8365, 0.2241, -0.1294\}$ are quantized into nine bits (including a sign bit) of two's complement precision as $H[z] = \{124, 214, 57, -33\}/256$ which are used in the code. The summation of *ZipWith* outputs is performed with *Reduce* in Line 21 and the links to input and output nodes are defined with *otherLinks* function in Line 25.

```
case class DFIR(inst: String, w: Int) extends Module {
  // input node x and output node y
  val (x, y) = (Streamer("x", w), Streamer("y", 2 * w + 1))
  // declare delay templates
  val (dly1, dly2, dly3, dly4) = (Delay("dly1", w, 1), Delay("dly2", w, 1),
   ↪  Delay("dly3", w, 1), Delay("dly4", w, 1))
  // stitch up the delay template nodes
  //val chn = Chain(dly1, dly2, dly3, dly4)
  val chn = Chain(dly1 outLinks (dly1.dout ~> dly2.din), dly2 outLinks
   ↪  (dly2.dout ~> dly3.din), dly3 outLinks (dly3.dout ~> dly4.din), dly4)
  // coefficients with data width set to 9
  val coeffs = Constants(9)(124, 214, 57, -33)
  // zipwith for chain outputs and constant coefficients
  val zw = ZipWith(chn.comps)(coeffs)(_ * _)
  // apply a sum fold-right to zipwidth outputs
  val fld = FoldR(zw.comps)(_ + _)
  // x ~> (dly1, dly1.din; connects x to dly1, fld.out ~> y;  connects fld
   ↪  output to y
  override val dfg = model(Seq(x ~> (dly1, dly1.din), chn, zw, fld, fld.out ~>
   ↪  y))
  // filter module name
  override val name: String = "nfir"
}
```

Listing 3: SdrLift code for FIR filter module represented as DFG in Fugure 4.6

Moreover, consider a comprehensive example of the R-2^2 SDF FFT algorithm that was introduced in [160, 161, 162]. The benefits of using R-2^2 to design the FFT core is that its FFT architecture has a simple pipeline control and reduced multipliers by a factor of $(N-1)/2$ compared to Radix-2 (R-2) and Radix-4 (R-4) which are used to design the FFT for Xilinx IP Cores Library [163]. By using a change variable technique $n =< \frac{N}{2}n_1 + \frac{N}{4}n_2 + n_3 > N$ and $k =< k_1 + 2k_2 + 4k_3 > N$, the relation the transform was reached as follows:

$$X(k_1 + 2k_2 + 4k_3) = \sum_{n_3=0}^{\frac{N}{4}-1} \left[H(k_1, k_2, n_3) W_N^{n_3(k_1+2k_2)}\right] W_{\frac{N}{4}}^{n_3 k_3},$$

where,

$$H(k_1 + k_2 + n_3) = \underbrace{\overbrace{\left[x(n_3) + (-1)^{k_1} x(n_3 + \frac{N}{2})\right]}^{BFI} + (-j)^{k_1+2k_2} \overbrace{\left[x(n_3 + \frac{N}{4}) + (-1)^{k_1} x(n_3 + \frac{3}{4}N)\right]}^{BFI}}_{BFII}. \tag{4.2}$$

As seen from Equation 4.2 above, a complete stage consists of two butterflies namely Butterfly I (BF I) and Butterfly II (BF II), delay feedback shift register and a twiddle factor complex multiplier. Furthermore, half a stage only has a single butterfly BF I and the control logic for the stages is facilitated by a simple counter. The complete pipeline 64-point R-2^2 SDF FFT processor is shown in Figure 4.7 and the generic formulas for determining SDF FFT parameters are described in Table 4.4. Note that this same FFT structure can be converted into the IFFT architecture using a straightforward procedure of conjugating the twiddle factors of the corresponding forward FFT output [164].

Table 4.4: The description of SDF FFT parameters

Formula	**Description**
N	Number of FFT points
$N-1$	Number of registers found in feedback registers
$log_4(N)$	Number of stages
$log_2(N)$	Number of butterflies and shift registers
$2log_2(N)$	Number of adders
$\frac{log_2(N)-1}{2}$	Number of complex multipliers

The structure of the FFT in Figure 4.7 can be exploited in SdrLift to generated the hardware code from a high-level code. The corresponding DFG is illustrated

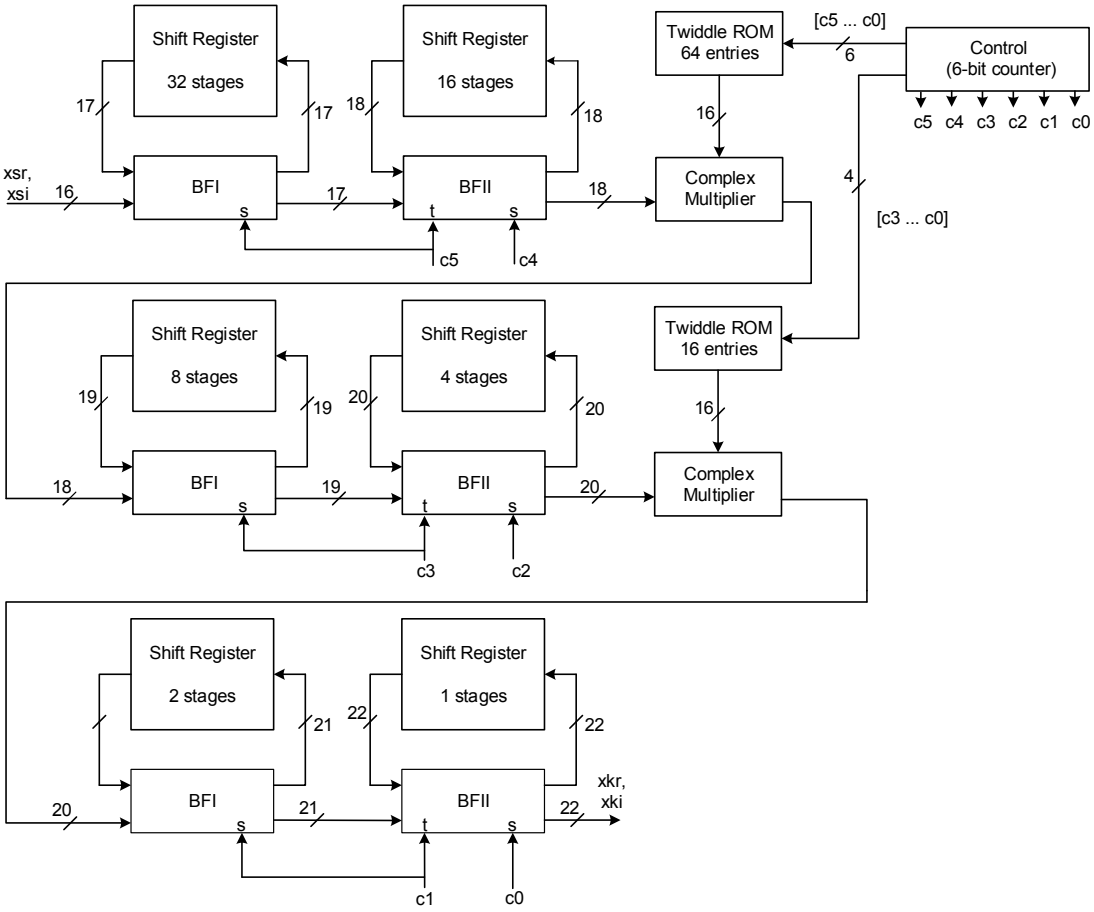


Figure 4.7: A 64-point R-2^2 SDF FFT hardware architecture.

in in Figure 4.8 and it is composed of four Read-Only Memory (ROM) nodes of type *TmplNode* which are used to store the twiddle factors, a 6-bit counter node of type *TmplNode* is used for control logic at each pipeline stage, and the three full stage nodes of *CompNode* type. The *CompNode* nodes are composed of other nodes which are illustrated in Appendix A. For example, each stage is the composite of other nodes where the first two nodes consist of the complex multiplier as illustrated in Figure A.1 and the last stage in Figure A.2 has no complex multiplier. Lastly, the DFG in Figure 4.8 also has five port nodes of *StrmNode* type (i.e. *en*, *xnr*, *xni*, *xkr* and *xki*) which are used for interfacing with other modules.

A full stage which is not the last stage in the R-2^2 SDF FFT chain has the BF II

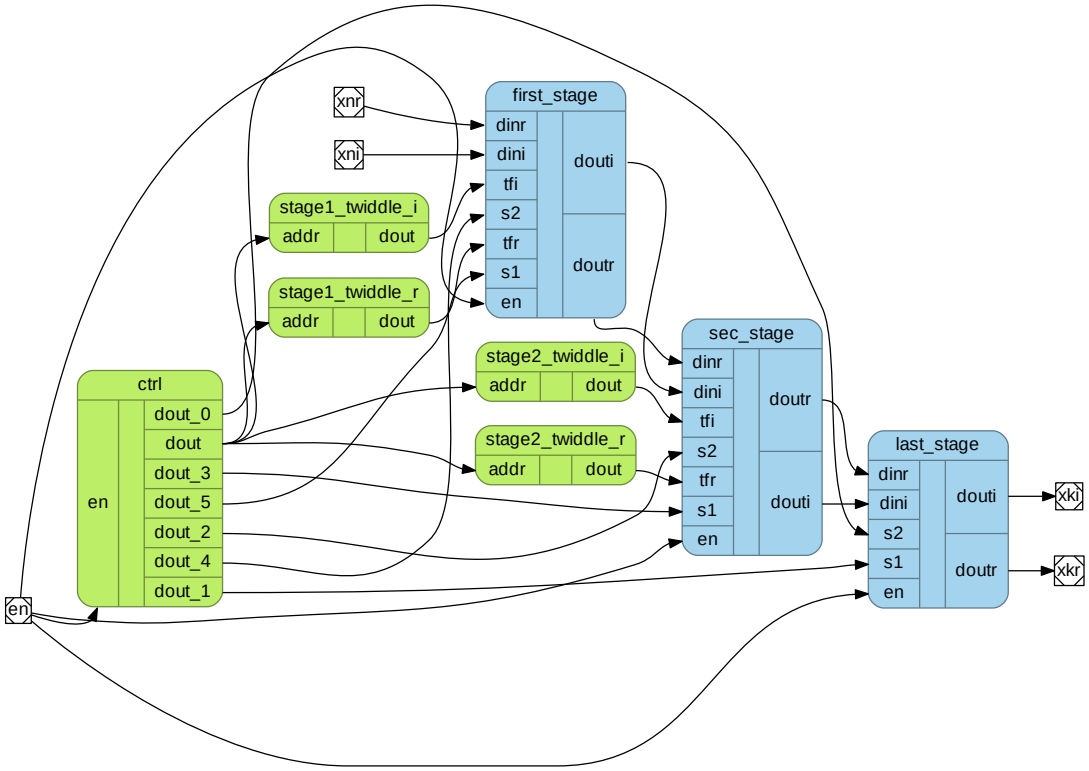


Figure 4.8: A DFG of a 64-point R-2^2 SDF FFT.

output that multiplied by that a twiddle factor. The twiddle factors are constant complex values stored in a ROM with a depth calculated as $N/2^{2i}$ at each i^{th} stage. The twiddle factors are computed using Equation 4.3 [165] and each value in ROM is selected by a counter output value. For example, in Figure 4.7 each ROM is addressed by a sliced vector of bits from a counter. The vector slice is in a form of $log_2\left(\frac{N}{2^{2i}}\right)$. The twiddle factor at i^{th}-stage, with $i = 0, 1, ..., (log_4 N) - 2$ is given by $W_i = \{u_x\}; x = 0, 1, ..., \frac{N}{2^{2i}}$ with $u_x = e^{\frac{-j2\pi v}{N}}$

$$v = \begin{cases} 0, & 0 \leqslant x < a \\ 2^{2i+1} \times (x-a), & a \leqslant x < 2a \\ 2^{2i} \times (x-2a), & 2a \leqslant x < 3a \\ 3 \times 2^{2i} \times (x-3a), & 3a \leqslant x < 4a \end{cases} \tag{4.3}$$

where,

$$a = \frac{N}{2^{2+2i}}$$

Listing 4 shows the high-level code for a DFG in Figure 4.8 using SdrLift Language. This results in an implementation of a *custom kernel* which is a node of type *ModNode*. The first two lines (Lines 2 and 3) declare the inputs and outputs ports respectively followed by input-output port mappings. For instance, a pair of port tuples indicates that *xnr* input port data will be processed and its results will be output on port *xkr*. Line 7 declares a counter which is a controller that exists as a template and it is instantiated as shown in Listing 20 of Appendix A. This is followed by input port interfacing in Line 8 which uses operator $\sim>$ to indicate that input port *en* declared in Line 2 connects to input port *en* of a counter. Lines 10 – 18 initialize the ROM components with twiddle factors while also creating the input interfaces with the controller. The ROM component is instantiated in Listing 21. Furthermore, Lines 20 – 26 declare the stages and their input interfaces. The first two stages are defined in Listing 18 and the last one is defined in Listing 19.

4.2.4 Dataflow IR

The IR_{DF} is implemented in the SDF-AP model using nodes produced by the IR_{CG} (known as modules) and pre-existing IP cores. A dataflow model allows for precise specification, simulation, and execution of complex DSP systems designs [92]. The features of a dataflow model enable bridging the gap between the high-level programming model and the low-level hardware. More notably, a dataflow model is implemented in an intuitive form of a dataflow graph that is familiar to most programmers. It allows concurrency in streaming applications where

```
case class FFT64(inst: String, w: Int) extends Module {
  val (en, xnr, xni) = (Streamer("en", 1), Streamer("xnr", w), Streamer("xni", w)) //inputs
  val (xkr, xki) = (Streamer("xkr", w + 6), Streamer("xki", w + 6)) // outputs
  override val iopaths = List((xnr, xkr), (xni, xki))
  val cmb = Combinational {
    // counter - fft controller
    val ctrl = Counter("ctrl", 6)
    val ctrl_comm = ctrl inLinks (en ~> (ctrl, ctrl.en))
    // Twiddle Factor ROM 1
    val rom1_r = Rom("stage1_twiddle_r", 16, Seq(16384, 16069, 15137, 13623, 11585, 9102,
     ↪  6270, 3196, 0, -3196, -6270, -9102, -11585, -13623, -15137, -16069, 16384, 16305,
     ↪  16069, 15679, 15137, 14449, 13623, 12665, 11585, 10394, 9102, 7723, 6270, 4756,
     ↪  3196, 1606, 16384, 15679, 13623, 10394, 6270, 1606, -3196, -7723, -11585, -14449,
     ↪  -16069, -16305, -15137, -12665, -9102, -4756, 16384, 16384, 16384, 16384, 16384,
     ↪  16384, 16384, 16384, 16384, 16384, 16384, 16384, 16384, 16384, 16384, 16384))
    val rom1_r_comm = rom1_r inLinks ((ctrl, ctrl.dout) ~> (rom1_r, rom1_r.addr))
    val rom1_i = Rom("stage1_twiddle_i", 16, Seq(0, -3196, -6270, -9102, -11585, -13623,
     ↪  -15137, -16069, -16384, -16069, -15137, -13623, -11585, -9102, -6270, -3196, 0,
     ↪  -1606, -3196, -4756, -6270, -7723, -9102, -10394, -11585, -12665, -13623, -14449,
     ↪  -15137, -15679, -16069, -16305, 0, -4756, -9102, -12665, -15137, -16305, -16069,
     ↪  -14449, -11585, -7723, -3196, 1606, 6270, 10394, 13623, 15679, 0, 0, 0, 0, 0, 0, 0,
     ↪  0, 0, 0, 0, 0, 0, 0, 0, 0, 0))
    val rom1_i_comm = rom1_i inLinks ((ctrl, ctrl.dout) ~> (rom1_i, rom1_i.addr))
    // Twiddle Factor ROM 1
    val rom2_r = Rom("stage2_twiddle_r", 16, Seq(16384, 11585, 0, -11585, 16384, 15137, 11585,
     ↪  6270, 16384, 6270, -11585, -15137, 16384, 16384, 16384, 16384))
    val rom2_r_comm = rom2_r inLinks ((ctrl, ctrl.dout, 3, 0) ~> (rom2_r, rom2_r.addr))
    val rom2_i = Rom("stage2_twiddle_i", 16, Seq(0, -11585, -16384, -11585, 0, -6270, -11585,
     ↪  -15137, 0, -15137, -11585, 6270, 0, 0, 0, 0))
    val rom2_i_comm = rom2_i inLinks ((ctrl, ctrl.dout, 3, 0) ~> (rom2_i, rom2_i.addr))
    // -- stages
    val fs1 = Stage("first_stage", w, 32, 16) // stage 1
    val fs1_comm = fs1 inLinks(en ~> (fs1, fs1.en), (ctrl, ctrl.dout, 5) ~> (fs1, fs1.s1),
     ↪  (ctrl, ctrl.dout, 4) ~> (fs1, fs1.s2), (rom1_r, rom1_r.dout) ~> (fs1, fs1.tfr),
     ↪  (rom1_i, rom1_i.dout) ~> (fs1, fs1.tfi), xnr ~> (fs1, fs1.dinr), xni ~> (fs1,
     ↪  fs1.dini))
    val fs2 = Stage("sec_stage", w + 2, 8, 4) // stage 1
    val fs2_comm = fs2 inLinks(en ~> (fs2, fs2.en), (ctrl, ctrl.dout, 3) ~> (fs2, fs2.s1),
     ↪  (ctrl, ctrl.dout, 2) ~> (fs2, fs2.s2), (rom2_r, rom2_r.dout) ~> (fs2, fs2.tfr),
     ↪  (rom2_i, rom2_i.dout) ~> (fs2, fs2.tfi), (fs1, fs1.doutr) ~> (fs2, fs2.dinr), (fs1,
     ↪  fs1.douti) ~> (fs2, fs2.dini))
    val ls = LastEvenStage("last_stage", w + 4, 2, 1)  // last stage
    val ls_comm = ls inLinks(en ~> (ls, ls.en), (ctrl, ctrl.dout, 1) ~> (ls, ls.s1), (ctrl,
     ↪  ctrl.dout, 0) ~> (ls, ls.s2), (fs2, fs2.doutr) ~> (ls, ls.dinr), (fs2, fs2.douti) ~>
     ↪  (ls, ls.dini)) outLinks((ls, ls.doutr) ~> xkr, (ls, ls.douti) ~> xki)
    :=(ctrl_comm, rom1_r_comm, rom1_i_comm, rom2_r_comm, rom2_i_comm, fs1_comm, fs2_comm,
     ↪  ls_comm)
  }
  override val name: String = "fft_n64"
}
```

Listing 4: SdrLift code for 64-point R-2^2 SDF FFT module represented as DFG in Fugure 4.8

actors encapsulate their own state that cannot be shared among other actors of the graph. Lastly, it describes the parallelism and data dependencies in a manner

that is natural to how the hardware operates [94, 95].

In a dataflow model, computational units known as 'actors' exchange data objects known as 'tokens' via unidirectional channels that are based on FIFO, as result it enables execution pipelining at the coarse-grained granularity level. The execution of the computational blocks, also known as the 'firing' of actors is defined by a set of rules called the dataflow MoC. These rules specify the firing rules, determine when the tokens can be consumed on the input FIFO channels, and also determines when the tokens can be produced on the output channels. In this work, the SDF [98] that is extended with access patterns (SDF-AP) [26, 111, 116, 146] is used. The access patterns move the SDF closer to hardware by specifying precise clock cycles when the token(s) can be read on the input channel(s) and written on the output channel(s). The description of the main classes that are used to implement the dataflow IR are described below:

Actor Creation

An actor maps the functional block to hardware and corresponds to a node in a directed graph. It is created as shown in Listing 5. Its attributes include a unique identifier *id*, instantiation label *inst*, execution time *execTime* and generic parameters *params* which are configured to define the behaviour of the underlying functional HW Block. Lastly, the *kernel* is the kernel object that represents either the IR_{CG} implementation if the kernel is a module or the place holder in a case of a black-box integration.

```
1 case class Actor(val id: String,val inst: String,val params: HashMap[String,
  ↪  Any],val execTime: Int, val kern: Kernel) {
2         ... // methods
3 }
```

Listing 5: An actor definition

Port Creation

A port provides an interface through which an actor can communicate with the adjacent actors. A source port and sink port are associated with a channel respectively. The attributes of a port include a rate, an access pattern and custom labels for port wires as defined in Listing 6.

```
1 case class KernelPort(val rate: Int, val ap: List[(Int, String)], val labels :
  ↪  java.util.HashMap[String,String])
```

Listing 6: A port definition

Channel Creation

A channel connects a source port to a sink port and it corresponds to an edge in a directed graph. It is implemented by inheriting the base object *DiEdge* of the *scala-graph* that defines a directed edge as shown in Line 0 of Listing 7. For all edge implementations, the *ExtendedKey*, *EdgeCopy* and *OuterEdge* are inherited. The *Seq* designates all the key attributes to the directed edge while the abstract method *copy* of *EdgeCopy* is transparently called by the graph to create an edge and it must return an instance of the edge class. The *delay* attribute denotes the number of initial tokens available in the channel while the *portMap* parameter specifies a mapping of the source actor output port to the sink actor input port. Lastly, the established channel edge factory shortcut *##* in Line 17 propagates a directed edge to a channel.

Dataflow Creation

The dataflow corresponds to a graph and is composed of *Actor* and *Channel* objects defined in Listing 5 and 7 respectively. Its definition of *SdfApGraph* object is shown in Listing 8 where *N* represents the type of nodes of a graph instance and *E[X]* is the kind of type of edges of a graph instance [32]. To demonstrate the

```
1  case class Channel[+N](srcActor: N, snkActor: N, id: String, srcPort:
   ↪  KernelPort, snkPort: KernelPort, dly: Int, portMap : (String,String))
   ↪  extends DiEdge[N](NodeProduct(srcActor, snkActor)) with ExtendedKey[N]
   ↪  with EdgeCopy[Channel] with OuterEdge[N, Channel] {
2   private def this(nodes: Product, id: String, srcPort: KernelPort, snkPort:
    ↪  KernelPort, dly: Int, portMap : (String,String)) {
3     this(nodes.productElement(0).asInstanceOf[N],
      ↪  nodes.productElement(1).asInstanceOf[N], id, srcPort, snkPort, dly,
      ↪  portMap)
4   }
5   def keyAttributes = Seq(id)
6   override def copy[NN](newNodes: Product) = new Channel[NN](newNodes, id,
    ↪  srcPort, snkPort, dly, portMap)
7   override protected def attributesToString = s" ($id)"
8   override def equals(other: Any) = other match {
9     case that: Channel[N] => that.id == this.id
10    case _                => false
11  }
12
13  /* other methods omitted */
14 }
15 object Channel {
16  implicit final class ImplicitEdge[A <: Actor](val e: DiEdge[A]) extends
    ↪  AnyVal {
17    def ##(id: String, srcPort: KernelPort, snkPort: KernelPort, dly: Int,
      ↪  portMap : (String,String)) = new Channel[A](e.source, e.target, id,
      ↪  srcPort, snkPort, dly, portMap)
18  }
19 }
```

Listing 7: A channel definition

usage of a IR_{DF}, the Listing 9 implements the graph in Figure 3.1 using *Actor*, *Channel* and *SdfApGraph* objects. Lines 0 to 3 declare the generic parameters, these are followed by Line 4 which declares the two actors, then Line 6 creates a dataflow graph and Line 8 invokes the *toplevel* method that generates the VHDL.

The IR_{DF} represents a complete SDR application which is ready for execution on the FPGA. It is based on the SDF-AP model where the actors represent the custom and blackbox kernels which respectively correspond to *Module* and *Macro* nodes below. The channels that connect the actors are created with the FIFO template nodes and allow coarse-grained pipelining needed for high performance.

```
object SdfApGraph {
  implicit class ExtGraph[N,E[X] <: EdgeLikeIn[X]](g: Graph[Actor,Channel]) {
         /* graph analysis methods */
         ...
  }
  implicit class ExtGraphNode[N,E[X] <: EdgeLikeIn[X]](node_ :
   ↪  Graph[Actor,Channel]#NodeT) {
    type NodeT = graph.NodeT
    val graph = node_.containingGraph
    val node = node_.asInstanceOf[NodeT]
    /* other methods */
    ...
  }
}
```

Listing 8: An SDF-AP dataflow definition.

```
val xParams = new java.util.HashMap[String, Any]()
xParams.put("dWidth", 8)
val yParams = new java.util.HashMap[String, Any]()
     yParams.put("dWidth", 8)
val (x,y)  =  (Actor("x", "xInst", xParams, 3), Actor("y", "yInst", yParams,
 ↪  5))

val sdfap  =  Graph[Actor, Channel](x ~> y ## ("ch1", Port(2, List((1,
 ↪  "011")), null), Port(3, List((1, "10101"), (1, "0")), null), 0, null))

sdfap.toplevel("exampleTop", 0.5)
```

Listing 9: An IR_{DF} definition based on SDF-AP.

- **Module**(name){body}: The *Module* describes the custom kernel which is mapped to a custom IP core in VHDL. **name:** the *Module* name; **body:** an arbitrary block of code executed by the *Module*.
- **Macro**(name){ifc}: The *Macro* is a blackbox representation of the existing IP core. **name:** the *Module* name; **ifc:** a block of code that specifies the interface for the existing IP core.

4.3 Chapter Summary

This chapter presents SdrLift, an intermediate compile framework for rapid prototyping of SDR applications. SdrLift consists of SdrLift language and the SdrLift compiler. The SdrLift language captures a structural behavior of the SDR application using functional constructs, topological patterns and data patterns that raise the level of design abstraction allowing the domain experts to focus only on the system design aspects that matter. The SdrLift compiler adopts a multifaceted IR transformations that decouple the micro-architecture from the entry-level program as well as bridging the gap between the SDF-AP model and the low-level implementation. This chapter has also extended the capability of SDF-AP by automating the computation of data access patterns for newly synthesized HW Blocks which are required to create SDF-AP model.

Chapter 5

Code Generation[1]

This chapter presents the hardware code (i.e. gateware) generation from three different levels of SdrLift IR namely IR_{FG}, IR_{CG} and IR_{DF} which were discussed in Sections 4.2.2, 4.2.3 and 4.2.4. First, the IR_{FG} graph is translated into small HW Blocks in VHDL which in this project are referred to as components. Then followed by mapping IR_{CG} graph to large HW Blocks in VHDL which in this project are called modules. These modules which correspond to the IP cores become the actors that form an SDF-AP model. The gateware generation from IR_{DF} begins quickly after successful analysis, validation, buffer sizing and schedule computation of the SDF-AP model. This gateware generation process in IR_{DF} entails the mix-and-match of the developed modules and the integration of existing VHDL blocks from the IP core library. This results in the VHDL generation for a complete SDR application which comprises the composition of the VHDL actors using the FIFO channels to represent the SDF-AP model. Furthermore, it is shown in this chapter how the model is mapped onto the low-level model of

[1]This chapter is based in part upon the following publications:
L. Tsoeunyane, S. Winberg, and M. Inggs, "Software-defined radio FPGA cores: Building towards a domain-specific language," *International Journal of Reconfigurable Computing*, vol. 2017, 2017.
L. Tsoeunyane, S. Winberg, and M. Inggs, "Automatic configurable hardware code generation for software-defined radios," *Computers*, vol. 7, no. 4, 2018.
L. Tsoeunyane, S. Winberg, and M. Inggs, "Sdrlift: An intermediate-level framework for synthesis of software-defined radio accelerators," in *2019 IEEE 10th International Conference on Mechanical and Intelligent Manufacturing Technologies (ICMIMT)*, pp. 166–173, Feb 2019.

hardware by efficiently applying low-level based optimizations and using a formal analysis technique that guarantees the correctness of the generated solutions. The actors in the SDF-AP model are mapped to the logic resources of the FPGA while the channels are mapped to the physical memory of an FPGA, typically the localized distributed Random Access Memory (RAM) for small FIFO and block RAM for large FIFO.

5.1 Code Generation from Fine Grained IR

The actual hardware implementation of the IR_{FG} nodes aims to build the components. These components are in the form of VHDL design units or entities with generic parameters. The algorithm for generating the code of a component is shown in Algorithm 4. The VHDL synthesis occurs through traversal of a DFG of the IR_{FG} and making various code generator decisions for each node visited in the process. For a node of type *ConstNode*, its identifier is declared as an integer constant. The *StrmNode* node is generally used for interface port in the entity. Its identifier is declared as an input port if the node is a subset of root nodes of IR_{FG}. Conversely, the identifier of *StrmNode* node is declared as an output port if the node belongs to a set of leaf nodes of IR_{FG}. Moreover, the arithmetic or logical operation is performed in the VHDL design architecture if the visited node is either of type *ArithNode* or *LogiNode*. This operation is based on the first direct predecessor node (*DirPredNode1st*) and the second direct predecessor node (*DirPredNode2nd*). SdrLift is able to determine the type of arithmetic/logical operation by checking the *prefix* of the current arithmetic/logical node.

For example, consider a DFG of a complex multiplier in Figure 4.4. This DFG can be represented in another DFG in Figure 5.1 which exposes more node attributes such as identifier (*id*), port type (*type*), data width (*width*) and arithmetic/logical operator (*op*). The code segment that is generated from the DFG is shown in Listing 10. The root nodes of the DFG in Figure 4.4 become the input ports of the entity (lines 6 – 9) while the leaf nodes are converted into output ports of the entity (lines 10 – 11). The rest of the nodes are used to implement the arithmetic

Algorithm 4 Code Generation Algorithm for a Component

Input: The IR_{FG} graph

Result: A *result* as VHDL design file

1: Set $Temp$ = Implementation component
2: **for each** *node* in IR_{FG} **do**
3: Set $v = node$
4: **if** v is type $ConstNode$ **then**
5: Declare a v identifier as a constant in $Temp$
6: **else if** v is type $StrmNode$ **then**
7: **if** v is a root of IR_{FG} **then**
8: Declare a v $indentifier$ as an entity input port of $Temp$
9: **else if** v is a leaf of IR_{FG} **then**
10: Declare a node $indentifier$ as an entity output port of $Temp$
11: **end if**
12: **else if** v is type $ArithNode$ OR $LogiNode$ **then**
13: Set $DiPredNode1st$ = First direct predecessor node of v in $Temp$
14: Set $DiPredNode2nd$ = Second direct predecessor node of v in $Temp$
15: $arithWith$ = compute integer width given two predecessor nodes
16: Declare a node $indentifier$ as an signal with width $arithWith$ in $Temp$
17: Set $arithOp$ = arithmetic or logical operation on $DiPredNode1st$ and $DiPredNode2nd$ in $Temp$
18: Add Signal Assignment of $arithOp$ to v signal in the architecture of $Temp$
19: **end if**
20: **end for**
21: Write $Temp$ to *result*

operations defined in lines 22 – 27. Unlike most code generators that generate VHDL code with random alphanumeric variables that make the code difficult to decipher, SdrLift uses the *{Prefix}_{DirPredNode1st}_{DirPredNode2nd}* convention which allows the code to be human-readable. For instance, the signal **mul_ai_bi** denotes the multiplier arithmetic node with prefix *mul* operates on

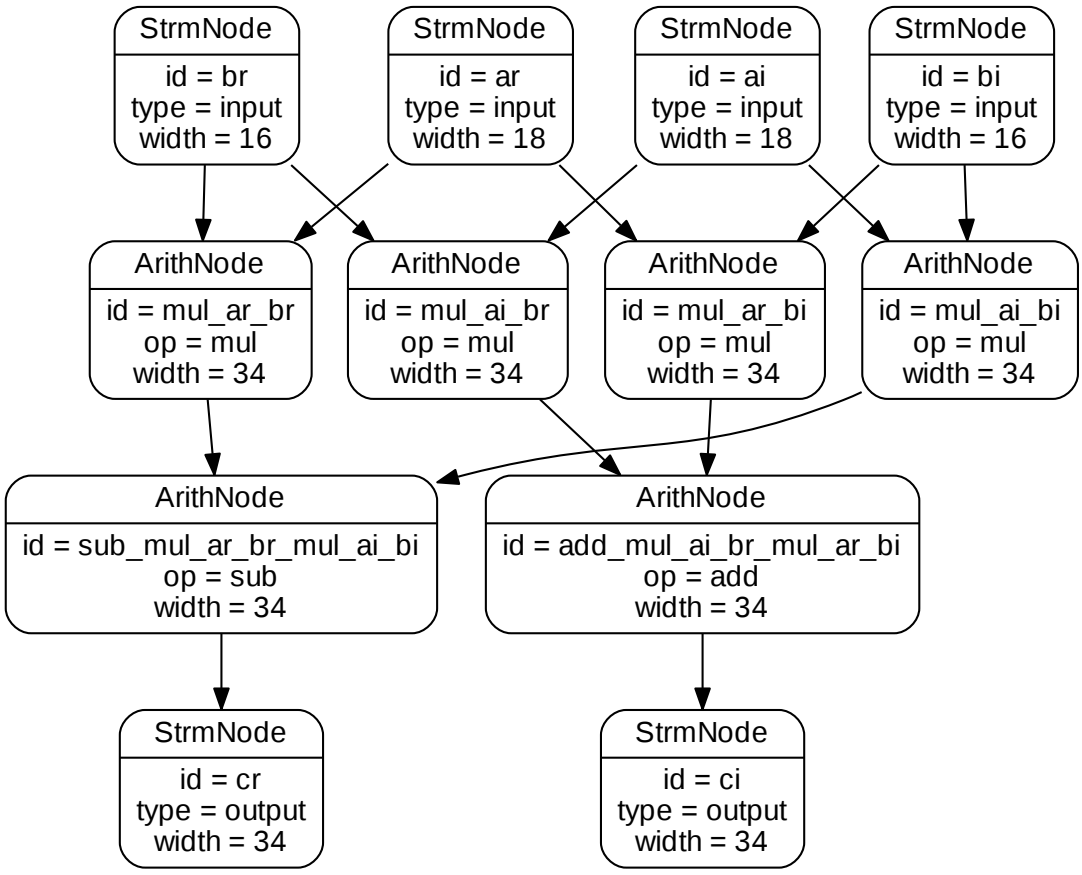


Figure 5.1: A detailed DFG of the direct complex multiplier.

Left Hand Side (LHS) node with identifier **ai** and the Right Hand Side (RHS) node with identifier **bi**. However, for long DFG paths with multiple successive arithmetic nodes, the signal identifier length grows exponentially leading to very long variables which are hard to read. This undesirable effect is alleviated by converting the last two parts of the variable naming format (i.e. *{DirPredNode2nd}* into short variable built of alphanumeric characters.

5.2 Code Generation from Coarse Grained IR

Gateware generation from the IR_{CG} nodes aims to generate ultra coarse-grained modules (i.e. custom kernels). Notice that the IR_{FG} nodes and the predefined

```vhdl
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
entity complexMult is
    port (
        ar : in std_logic_vector(17 downto 0);
        bi : in std_logic_vector(15 downto 0);
        ai : in std_logic_vector(17 downto 0);
        br : in std_logic_vector(15 downto 0);
        ci : out std_logic_vector(33 downto 0);
        cr : out std_logic_vector(33 downto 0)
    );
end;
architecture rtl of complexMult is
    signal mul_ai_bi : std_logic_vector(33 downto 0);
    signal mul_ar_br : std_logic_vector(33 downto 0);
    signal sub_mul_ar_br_mul_ai_bi : std_logic_vector(33 downto 0);
    signal mul_ai_br : std_logic_vector(33 downto 0);
    signal mul_ar_bi : std_logic_vector(33 downto 0);
    signal add_mul_ai_br_mul_ar_bi : std_logic_vector(33 downto 0);
begin
    mul_ai_bi <= ai * bi;
    mul_ar_br <= ar * br;
    sub_mul_ar_br_mul_ai_bi <= mul_ar_br - mul_ai_bi;
    mul_ai_br <= ai * br;
    mul_ar_bi <= ar * bi;
    add_mul_ai_br_mul_ar_bi <= mul_ai_br + mul_ar_bi;
    ci <= add_mul_ai_br_mul_ar_bi;
    cr <= sub_mul_ar_br_mul_ai_bi;
end;
```

Listing 10: The component VHDL code of a complex multiplier based on example Figure 5.1.

templates can be mixed with the IR_{CG} node to create the modules. These modules are viewed as IP cores in VHDL which will eventually be composed to build a top-level VHDL design of the desired application. In order to create the VHDL code for a module, the IR_{CG} is traversed where each node has a representative component (i.e. VHDL entity). Algorithm 5 shows the algorithm for generating the code of a module. For each node visited, the algorithm creates a unique instance of the component depending on whether the type of a node is *CompNode* or *TmplNode*. Multiple instances of a component may share the same VHDL

entity but each component instance is unique and has its own generic parameters and mapped signals. The input ports for each component instance of a current node are connected to the source node through the incoming edge. Likewise, the output ports of the component instance of a current node are connected to the sink node through the outgoing edge.

Algorithm 5 Code Generation Algorithm for a Module

Input: A IR_{CG} graph
Result: A *result* as VHDL design file

```
1:  Set Temp = Implementation module
2:  for each node in IR_CG do
3:      Set v = node
4:      if v VHDL entity is NOT declared then
5:          Declare VHDL entity in Temp
6:      end if
7:      Add VHDL instance to architecture of Temp
8:      if v is type CompNode then
9:          Generate the VHDL design file of v from IR_FG
10:     else if v is type TmplNode then
11:         Generate the VHDL design file of v from predefined Template
12:     else
13:         Apply Algorithm 4 and append created code to Temp
14:     end if
15: end for
16: Write Temp to result
```

For example, consider a DFG of the direct form FIR filter in Figure 4.6. The DFG can be drawn in another DFG form that exposes more parameters used by the compiler as shown in Figure 5.2. Translating this DFG into VHDL results in a code segment that is depicted in Listing 11. For the sake of brevity, the header with library/package clauses is omitted and the declaration of signals and components are left hidden. In addition, the input signals (*clk*, *rst* and *en*) and output signal (*vld*) that belong to the entity and the nodes of type *TmplNode* have also been removed in the illustration in Listing 11. The *ConstNode* nodes

are directly converted into binary constants that become operands to *ArithNode* nodes. The *TmplNode* nodes represent the delays that are implemented as shift registers in hardware. They are instantiated by mapping the generic parameters and the input/output port signals.

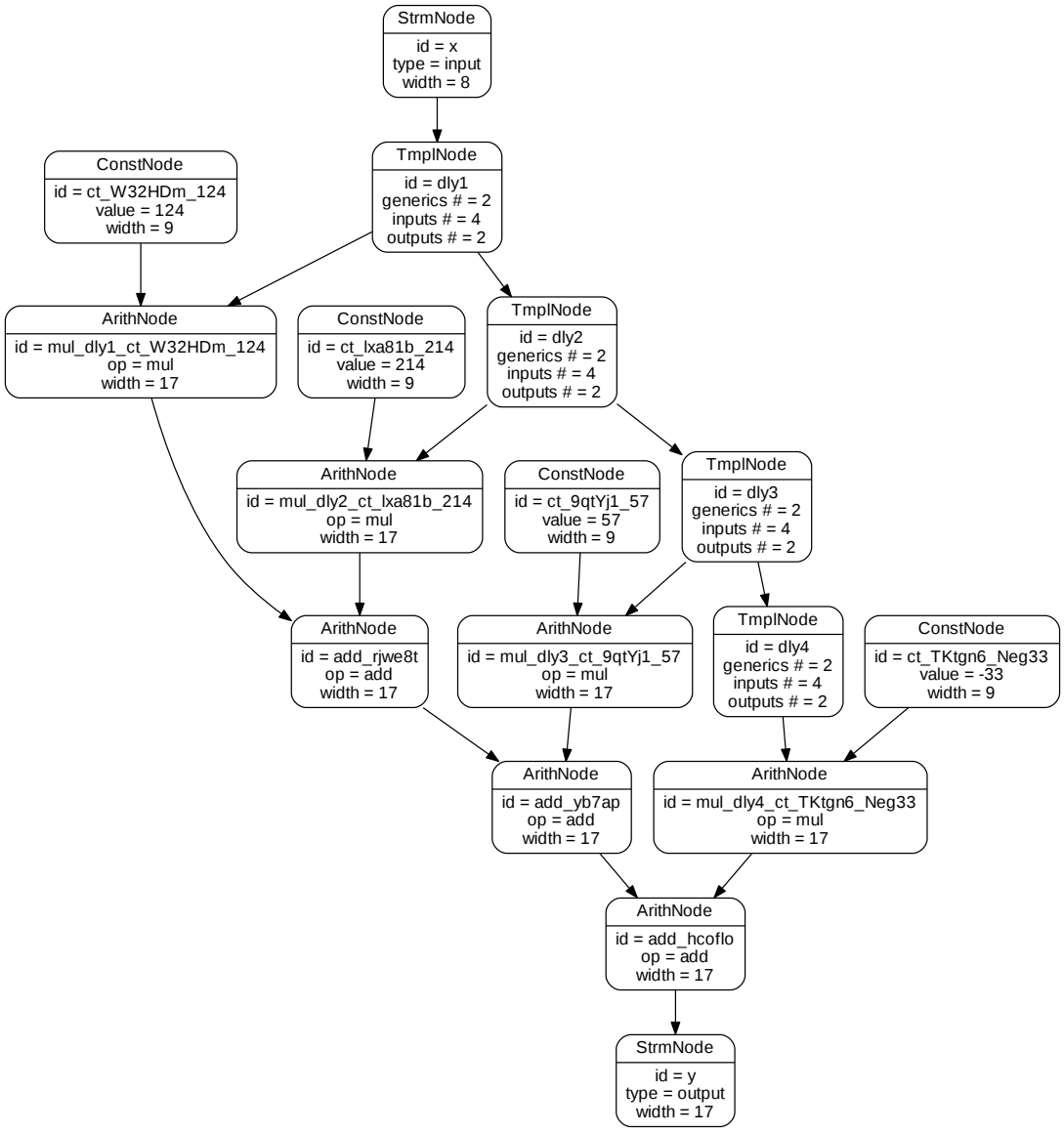


Figure 5.2: A detailed DFG of the direct form FIR filter.

Furthermore, the DFG of the R-2^2 SDF FFT with more compiler parameters is shown in Figure 5.3. The compiler translates this DFG into the VHDL code that

```vhdl
-- omitted library clauses ...
entity nfir is
    port (
        x : in std_logic_vector(7 downto 0);
        y : out std_logic_vector(16 downto 0)
    );
end;
architecture rtl of nfir is
    -- omitted signal and component declarations ...
begin
    mul_dly4_ct_TKtgn6_Neg33 <= dly4_dout * b"111011111";
    mul_dly2_ct_lxa81b_214 <= dly2_dout * b"011010110";
    mul_dly3_ct_9qtYj1_57 <= dly3_dout * b"000111001";
    mul_dly1_ct_W32HDm_124 <= dly1_dout * b"001111100";
    dly1 : delay
        generic map (WIDTH => 8, DEPTH => 1)
        port map (din => x, dout => dly1_dout);
    add_hcoflo <= add_yb7ap + mul_dly4_ct_TKtgn6_Neg33;
    add_rjwe8t <= mul_dly1_ct_W32HDm_124 + mul_dly2_ct_lxa81b_214;
    add_yb7ap <= add_rjwe8t + mul_dly3_ct_9qtYj1_57;
    y <= add_hcoflo;
    dly2 : delay
        generic map (WIDTH => 8, DEPTH => 1)
        port map (din => dly1_dout, dout => dly2_dout);
    dly3 : delay
        generic map (WIDTH => 8, DEPTH => 1)
        port map (din => dly2_dout, dout => dly3_dout);
    dly4 : delay
        generic map (WIDTH => 8, DEPTH => 1)
        port map (din => dly3_dout, dout => dly4_dout);
end;
```

Listing 11: The component VHDL code of a direct form FIR filter based on example Figure 5.2.

is shown in Listing 12 where port, signal and component declarations have been omitted.

5.3 Code Generation from Dataflow IR

This section presents the composition of IP blocks from IR_{DF} and generation of hardware code that runs on the FPGA. IR_{DF} employs an SDF-AP model

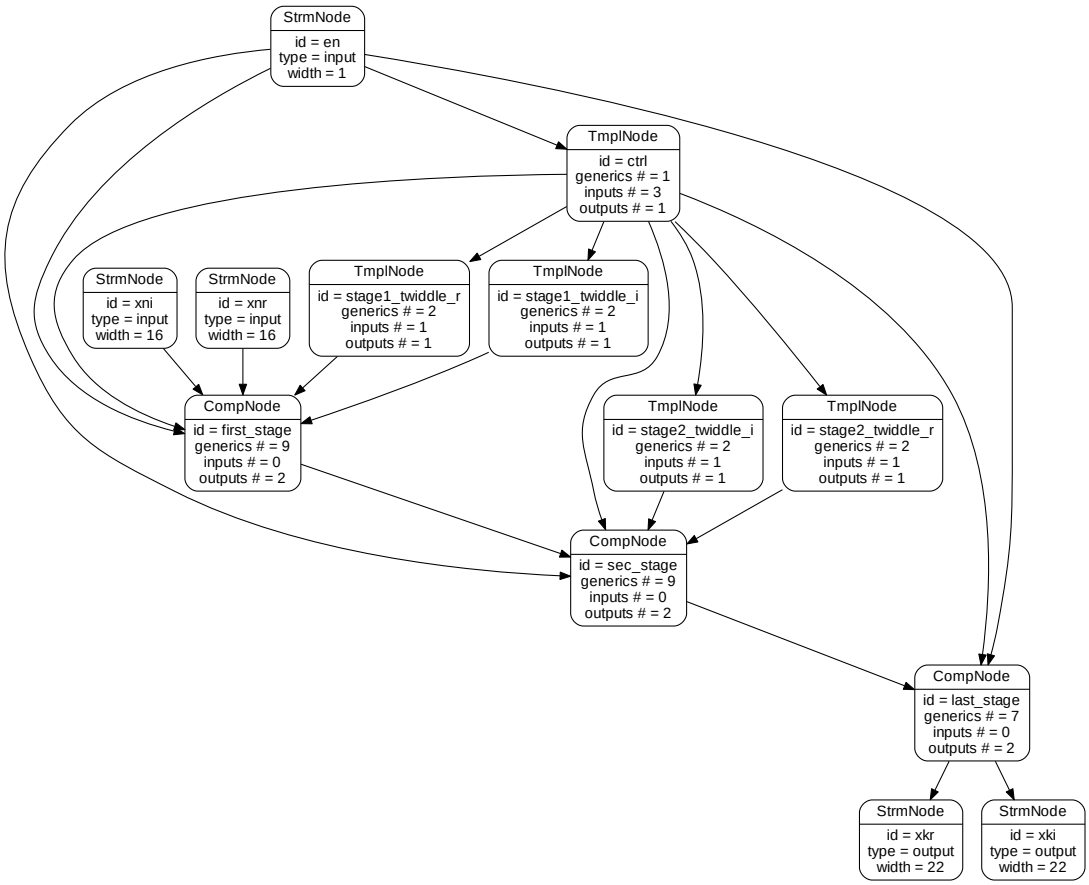


Figure 5.3: A detailed DFG of a 64-point R-2^2 SDF FFT.

for computation of timing and performance properties of the hardware system thereby raising the level of design abstraction. The aim is to bridge the semantic gap between the high-level model using a dataflow model and the low-level model of hardware which is largely described using FSMs. The approach in this work targets the problem domain of SDR in which the resultant solutions run on the FPGA platform. The hardware implementation begins right after the analyses, validation, and scheduling of the SDF-AP model. The four optimization techniques for low-level hardware design synthesis that result efficient hardware design results are defined. Instead of using the traditional hardware generation approach which relies upon *correct-by-construction* methods, the approach in this work guarantees the efficiency of the results which conform to the original application specifications. The design conformance is ensured through

```
entity fft_n64 is
      -- omitted en,rst,clk,xnr,xni,xki, and xkr port declaration
end;
architecture rtl of fft_n64 is
  -- omitted signal and component declarations
begin
    ctrl_dout_5 <= ctrl_dout(5); ctrl_dout_4 <= ctrl_dout(4);
    first_stage : first_stage_stage port map(en=>en, s1=>ctrl_dout_5,
     ↪  dinr=>xnr, tfi=>stage1_twiddle_i_dout, dini=>xni, s2=>ctrl_dout_4,
     ↪  tfr=>stage1_twiddle_r_dout, douti=>first_stage_douti,
     ↪  doutr=>first_stage_doutr, clk=>clk, rst=>rst);
    ctrl_dout_1 <= ctrl_dout(1); ctrl_dout_0 <= ctrl_dout(0);
    last_stage : last_stage_evenstage port map(s1=>ctrl_dout_1,
     ↪  dinr=>sec_stage_doutr, dini=>sec_stage_douti, s2=>ctrl_dout_0,
     ↪  en=>en, douti=>last_stage_douti, doutr=>last_stage_doutr, clk=>clk,
     ↪  rst=>rst);
    ctrl_dout_3 <= ctrl_dout(3); ctrl_dout_2 <= ctrl_dout(2);
    sec_stage : sec_stage_stage port map(dini=>first_stage_douti, en=>en,
     ↪  s1=>ctrl_dout_3, tfi=>stage2_twiddle_i_dout, dinr=>first_stage_doutr,
     ↪  tfr=>stage2_twiddle_r_dout, s2=>ctrl_dout_2, doutr=>sec_stage_doutr,
     ↪  douti=>sec_stage_douti, clk=>clk, rst=>rst);
    ctrl : ctrl_counter generic map(WIDTH=>6) port map(en=>en,
     ↪  dout=>ctrl_dout, clk=>clk, rst=>rst);
    xki <= last_stage_douti; xkr <= last_stage_doutr;
    stage1_twiddle_i : stage1_twiddle_i_rom generic map(ADDR_WIDTH=>6,
     ↪  DATA_WIDTH=>16) port map(addr=>ctrl_dout,
     ↪  dout=>stage1_twiddle_i_dout);
    ctrl_dout_slice_5_downto_2 <= ctrl_dout(5 downto 2);
    stage2_twiddle_i : stage2_twiddle_i_rom generic map(ADDR_WIDTH=>4,
     ↪  DATA_WIDTH=>16) port map(addr=>ctrl_dout_slice_5_downto_2,
     ↪  dout=>stage2_twiddle_i_dout);
    ctrl_dout_slice_5_downto_2 <= ctrl_dout(5 downto 2);
    stage2_twiddle_r : stage2_twiddle_r_rom generic map(ADDR_WIDTH=>4,
     ↪  DATA_WIDTH=>16) port map(addr=>ctrl_dout_slice_5_downto_2,
     ↪  dout=>stage2_twiddle_r_dout);
    stage1_twiddle_r : stage1_twiddle_r_rom generic map(ADDR_WIDTH=>6,
     ↪  DATA_WIDTH=>16) port map(addr=>ctrl_dout,
     ↪  dout=>stage1_twiddle_r_dout);
end;
```

Listing 12: The component VHDL code of a 64-point R-2^2 SDF FFT.

the formal analysis of how a generated hardware model faithfully implements its specification as captured in the SDF-AP model is discussed in Section 5.4.

5.3.1 Hardware Dataflow Actors

To implement the SDF-AP model in hardware, each SDF-AP actor becomes a block of logic which encapsulates its own state that cannot be shared among other blocks in the network. The block of logic is also known as an IP core (or HW Block) and it has handshaking communication ports both on the input and output interface. For each HW Block to execute, it must obey all the firing rules of an actor as specified by the SDF-AP model. All the HW Blocks are expected to be synchronous to a fundamental clock (*clk*) input port and can be reset asynchronously via a *reset* input port. The input data is received on data-in (*din*) input port when the value of valid-in (*en*) input port is set high. Furthermore, an output data is sent through data-out (*dout*) output port when the value of the valid-out (*vld*) output port is set high to denote a valid output data.

5.3.2 Hardware Dataflow Channels

The actors of the SDF-AP model use unidirectional channels to communicate tokens to each other. The channel is mapped to a physical FIFO buffer that is typically implemented as distributed or block RAM in FPGA. The allocated buffer size for each FIFO is determined using the Algorithm 2. The FIFO is also regarded as a fixed HW Block with the generic parameters (such as a customizable data width and storage depth) and their values can be changed during synthesis of the VHDL code. In addition to *clk*, *rst*, *din*, *vld*, and *dout* ports, the FIFO has input and output handshaking ports namely write-enable (*we*) input port which is set by a source HW Block to enable the FIFO write operation and the read-enable (*re*) input port which is set by a sink HW Block to request the read of data sample from a FIFO. There are also status ports which include the fifo-empty (*em*) and a fifo-full (*fl*). *em* indicates that there are no stored data samples in the FIFO and *fl* asserts when the FIFO buffer is full. An empty FIFO will not output a valid data when *vld* port is set high, similarly, the FIFO will not allow write operation when *we* port is set high.

5.3.3 Hardware Design

The SDF-AP model may be closest to the hardware in contrast to other SDF-based models but its implementation on hardware is not as trivial as it may seem. Like most dataflow models, SDF-AP model is asynchronous and abstracts most of the hardware behaviour, therefore, making it suitable for high-level application description. It performs analysis of timing (i.e. token consumptions and productions) and performance (i.e. such throughput, latency and buffer sizes) properties which are often difficult to analyse at the low-level of hardware description. However, it has no prior knowledge of the low-level models of hardware implementation such as the finite state machines, datapath components, multiplexers, LUTs, pipeline registers. In this work, more emphasis is put on the synchronous FSM as it is the most dominant model in the generated hardware design. It is evident that there is a huge semantic gap between a dataflow model and hardware and this complicates the correct implementation of the hardware.

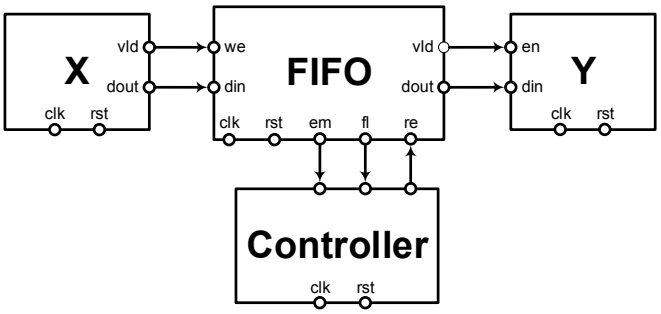


Figure 5.4: The hardware design of SDF-AP (based on Figure 3.1).

The first step towards hardware design is by an illustration of the expected hardware design in Figure 5.4 which is implemented from the SDF-AP example in Figure 3.1. Each HW Block in Figure 5.4 has the input and output ports which connect to other blocks using signals or wires. The signals in Figure 5.4 are of output type which makes the system compliant with a Moore machine. The ports which are not shown in HW Blocks X and Y interface with the external systems. These ports are either system input ports or system output ports which

form a top-level entity of the VHDL design as shown in Listing 13.

```
1  library IEEE;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  entity sdfapv0 is
5      port (
6          clk  : in std_logic;
7          rst  : in std_logic;
8          en   : in std_logic;
9          din  : in std_logic_vector(7 downto 0);
10         vld  : out std_logic;
11         dout : out std_logic_vector(7 downto 0)
12     );
13 end;
```

Listing 13: A top-level entity of the hardware design in Figure 5.4

The input signals *en* and *din* of the HW Block X allow input data to be received by the system while signals *vld* and *dout* of block Y send the data out of the system. All the blocks connect to the fundamental system signals *rst* and *clk*. The top-level entity description is followed by the behavioural description which composes of two the HW Blocks (i.e. X and Y) using a FIFO buffer of size 2. In this example, the HW Blocks have been described in VHDL "by hand". In real-world applications, some blocks may be acquired from a library of IP cores which are provided by mainstream commercial Very Large Scale Integration (VLSI) vendors such Xilinx, Altera etc or the open-source development communities such as OpenCores [166], GRLIB [63, 167]. The VHDL code in Listing 14 is an extract from the system architecture description in Figure 5.4.

First, the input registers of the source block X are connected to top-level entity ports and then followed by the instantiation of the HW Blocks and FIFO channel. The interfacing of the HW Blocks with the FIFO is a combinational assignment of output signals (lines 11 – 14). The *process* implements the FSM that controls the flow of data to or from the FIFO and the details of how it is built are presented in Section 5.3.4. The last two lines route data samples to the external environment of the system. The FIFO buffer stores and stalls the samples such that the strict pattern matching of SDF-AP can be achieved under the 1-periodic schedule and

```
1  architecture rtl of sdfapv0 is
2  ... -- hidden register and component declaration
3  begin
4      xInst_en_i <= en;
5      xInst_din_i <= din;
6      xInst : x port map ( ... );
7      yInst : y port map ( ... );
8      xInst_yInst_ch1 : fifo
9          generic map (DATA_WIDTH => 8, FIFO_DEPTH => 2)
10         port map ( ... );
11     xInst_yInst_ch1_we <= xInst_vld_o;
12     xInst_yInst_ch1_din <= xInst_dout_o;
13     yInst_en_i <= xInst_yInst_ch1_vld;
14     yInst_din_i <= xInst_yInst_ch1_dout;
15     ... -- process definition hidden
16     vld <= yInst_vld_o;
17     dout <= yInst_dout_o;
18 end;
```

Listing 14: The architecture description of the hardware design in Figure 5.4

throughput constraints. This pattern matching is relative to specific triggering of actor firings at specific clock cycles and this is facilitated by the FIFO controller in Figure 5.4 that is realized using a VHDL *process*.

The correct functional operation of the implemented hardware design in accordance with the schedule in Figure 3.2 is described by the timing diagram in Figure 5.5. For the sake of brevity, *din* and *dout* buses of the HW Blocks are excluded, and instead use the status and control signals. The signals are labelled according to the HW Blocks (i.e. X = source HW Block, FF = FIFO buffer, Y = sink HW Block) to which they belong. For example, X_{vld} refers to *vld* signal of the HW Block X. From the timing diagram shown in Figure 5.5 it is clear that the X_{vld} and Y_{en} signals correspond to the execution patterns of the source port and the sink port (i.e. $EP_{*,p}$ and $EP_{*,q}$) as previously defined in Section 3.2. It is noteworthy to observe that X_{vld} and FF_{we} are similar as they connect to each other directly, hence forming a single signal which in call w. Similarly, the FF_{vld} and Y_{en} are the same as they connect to each other directly and they are both a 180° phase shifted versions of FF_{re}. The direct connection of FF_{vld} and Y_{en} ports is called the output signal e.

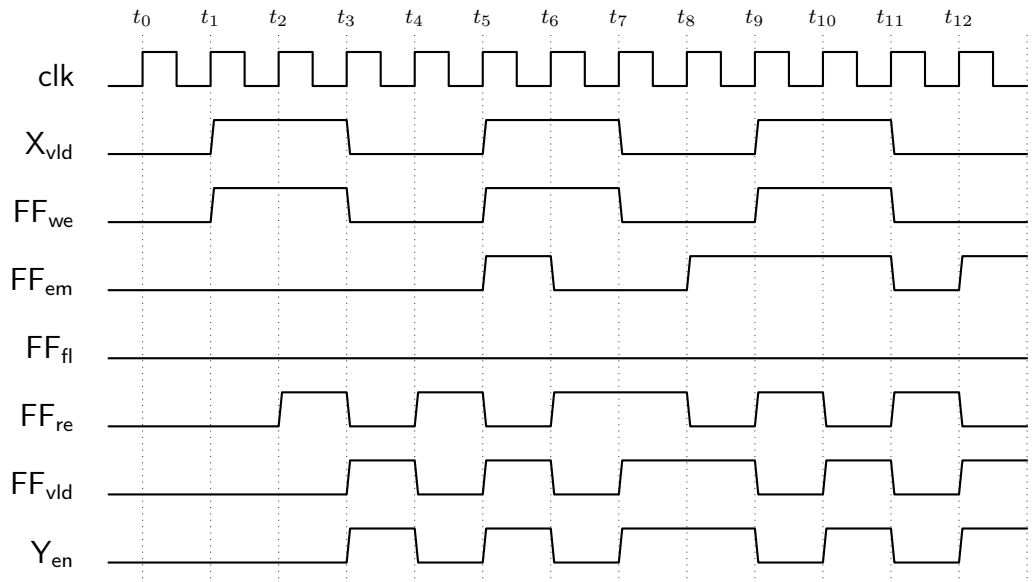


Figure 5.5: Timing diagram of a source actor, the FIFO channel and a sink actor (based on Figure 5.4).

5.3.4 Hardware Model Using FSM

The low-level of hardware design abstraction is modeled as FSMs that are based on a *Moore machine*. An FSM is a 6-tuple $M = (I, O, S, s_0, \delta, \lambda)$ where I and O represent the finite input and output space respectively (i.e. boolean input/output signals of M). S is a set of finite states where $s_0 \in S$ is the initial state; $\delta : S \times I \to S$ is the next state (transition) function and $\lambda : S \times I \to O$ is the output function. An FSM M is a Moore machine if the all the output signals depend on the present state and not the values of its inputs, hence the output function becomes $\lambda : S \to O = \{0, 1\}$. This type of FSM is also said to be *closed* [151] due to the fact that the set of its input signals is empty (i.e. $I = \emptyset$). On the other hand, M is *open* if $I \neq \emptyset$.

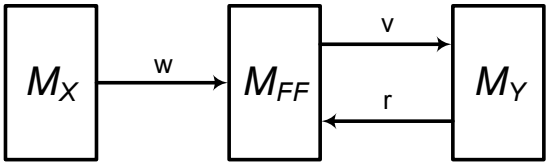
A set of behaviours as defined by a closed FSM M are of the form $s_0 \xrightarrow{a_0/b_0} M_1 \xrightarrow{a_1/b_1} M_2 \xrightarrow{a_2/b_2} \cdots$ where $s_i \in S$ denotes the current state, $a_i \in I$ is the current state input assignment, $s_{i+1} = \delta(a_i, b_i)$ is the next state, $b_i = \lambda(s_i, a_i) \in O$ is the the current state output assignment. The states occur at synchronous clock cycle i and the *observable behaviour* of M is defined as $(a_0, b_0)(a_1, b_1)(a_2, b_2) \cdots$

[151]. Clearly, the *closed* FSM defines a set of behaviours in a form of $s_0 \xrightarrow{b_0} M_1 \xrightarrow{b_1} M_2 \xrightarrow{b_2} \cdots$ and the *closed observable behaviour* becomes $(b_0)(b_1)(b_2)\cdots$.

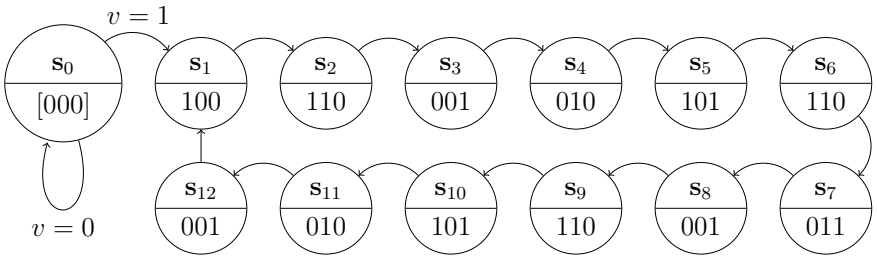
5.3.5 FSM Composition

The composition of FSMs leads to a single FSM with a set of states which is the product of the set of states of FSMs of the HW Blocks in the system. For a closed FSM which is used in this work, each composite state has output signals with propagation that is instantaneous. The transition from the present state to the next takes place on every rising edge of the system clock. An example of a composite FSM $M = M_X \times M_{FF} \times M_Y$ is shown in Figure 5.6b. Figure 5.6a is the black-box representation of M which is composed of the source HW Block FSM (i.e. M_X), FIFO buffer FSM (i.e. M_{FF}) and a sink HW Block FSM (i.e. M_Y). Since M is a Moore machine, it only has outputs signals namely write-enable w, data-valid v and read-enable r as defined in Section 5.3.3. The upper half of each state labels the state s_i where $i = \{0...N-1\}$ and N is the total number of states. The lower half of each state is either a single-dimensional or a two-dimensional vector of the output signals w, r, and v. A format wrv is used to represent a single-dimensional vector and a two-dimensional vector of a single state is represented in a form of $[w_i r_i v_i \ w_1 r_1 v_1 \ w_2 r_2 v_2 \ ... \ w_{N-1} r_{N-1} v_{N-1}]$ that has a sequential order. This two-dimensional vector is therefore associated with a state that has a *loop* transition and the vector length equals the number of times iterated by the *loop* transition in synchronous to the system clock. The values of all the three output signals are obtained using the 1-periodic schedule as explained in Section 5.3.3. Generally, for every valid execution schedule that leads to a finite buffer size such as the one in Figure 3.2, the composite FSM can be correctly constructed by matching the source actor execution pattern $EP*, p$ to the output signal w, and the sink actor execution pattern $EP*, q$ to the output signal v. Note that the output signal r is the $-180°$ phase-shifted version of output signal v.

The FSM M example in Figure 5.6b has features which are key to understanding execution properties of a generated hardware system. First, it is important to



(a) Composite FSM structure using three FSMs: M_X, M_{FF} and M_Y.



(b) Behavior of a Composite FSM $M = M_X \times M_{FF} \times M_Y$

Figure 5.6: The composite FSM M is constructed using FSMs M_X, M_{FF} and M_Y by using the computed 1-periodic schedule in Figure 3.2. The vectors in lower half of each state represent the respective output signals w, r and v in that state.

note that the order of transitions in M is sequential where each transition is synchronous to a fundamental system clock. The initial finite sequence of states and transitions is called the *startup phase*, and followed by this is another sequence which repeats infinitely and is referred to as a *periodic phase*. The output signals of the states in the startup phase are in the form of a two-dimensional vector where all their values are set to 0 (i.e. $w = r = v = 0$). In the given example, there is only one state (i.e. s_0) and one transition in this phase. The periodic phase has 12 states and transitions starting from M_1 to s_{12}.

The T is determined by counting the number of transitions in the period phase of FSM M phase in Figure 5.6b. Furthermore, the number of data samples produced (resp. consumed) to (resp. from) the output (resp. input) channel correspond to number of output signal w (resp. v) where their values is set high (i.e. $w = v = 1$). The produced samples are always equal to the consumed samples and in the given example 12 is obtained. The throughput is determined by dividing the count of one of the output signals (i.e. w or v or r where its value is set to 1) by number of transitions in the periodic phase where in the example

the throughput value of $6/12 = 0.5$ measured in Samples Per Cycle (SPC) is obtained. The IL is the sum of all the the transitions in the startup phase and periodic phase and in the example of Figure 5.6 IL is 13. An observable behaviour of the closed M therefore becomes

$$\begin{aligned}\omega_M &= (\bar{\mathsf{w}}\bar{\mathsf{r}}\bar{\mathsf{v}})\Big((\mathsf{w}\bar{\mathsf{r}}\bar{\mathsf{v}})(\mathsf{w}\mathsf{r}\bar{\mathsf{v}})(\bar{\mathsf{w}}\bar{\mathsf{r}}\mathsf{v})(\bar{\mathsf{w}}\mathsf{r}\bar{\mathsf{v}})(\mathsf{w}\bar{\mathsf{r}}\mathsf{v})(\mathsf{w}\mathsf{r}\bar{\mathsf{v}})(\bar{\mathsf{w}}\mathsf{r}\mathsf{v})(\bar{\mathsf{w}}\bar{\mathsf{r}}\mathsf{v})(\mathsf{w}\mathsf{r}\bar{\mathsf{v}})(\mathsf{w}\bar{\mathsf{r}}\mathsf{v})(\bar{\mathsf{w}}\mathsf{r}\bar{\mathsf{v}})\bar{\mathsf{w}}\bar{\mathsf{r}}\mathsf{v})\Big)^{\infty} \\ &= (000)\Big((100)(110)(001)(010)(101)(110)(011)(001)(110)(101)(010)(001)\Big)^{\infty}\end{aligned}$$

where $()^{\infty}$ denotes the periodic phase of M.

The FSM M in Figure 5.6 implements the control logic that enables the sink HW Block to read data from the FIFO. Since the buffer holds data for a finite period of time until the sink block is ready to read it, the controller determines the exact clock times when this should happen as specified in the consumption execution pattern ($EP_{*,q}$). Writing data to the FIFO block does not require the FSM control logic as the allocated buffer size of the FIFO buffer is sufficient to store the received data from source HW Block. Therefore a direct asynchronous connection of X_{vld} to FF_{we} is sufficient to write the data samples into a FIFO buffer. Implementing a controller involves a sequential description of the FSM M inside the *process* in VHDL as shown in Listing 15. As mentioned above, a startup phase only consists of the first state (i.e. s_0) of M which corresponds to the state *"0000"* in VHDL. A transition from the startup phase to the periodic phase takes place when the HW Block X produces the first sample. This is detected by M when the value of the output signal X_{vld} is set high thereby moving the M to the second state where the periodic phase starts. The register *state* (i.e. **xInst_yInst_ch1_state** in VHDL) keeps track of the next state of FSM M.

5.3.6 FSM Optimizations

In very large and complex systems, the composition of states often has an exponential growth of the size of the system state space leading to a problem known

```
xInst_yInst_ch1_proc : process(rst, clk)
begin
   if rst = '1' then
      ... --reset process registers
   elsif clk'EVENT and clk = '1' then
      ... -- hidden code
      case xInst_yInst_ch1_state is
         when b"0000" =>
            xInst_yInst_ch1_re <= '0';
            xInst_yInst_ch1_state <= b"0000";
            if xInst_vld_o = '1' then
               xInst_yInst_ch1_re <= '1';
               xInst_yInst_ch1_state <= b"0010";
            end if;
         when b"0001" =>
            xInst_yInst_ch1_re <= '1';
            xInst_yInst_ch1_state <= b"0010";
         when b"0010" =>
            xInst_yInst_ch1_re <= '0';
            xInst_yInst_ch1_state <= b"0011";
         when b"0011" =>
            xInst_yInst_ch1_re <= '1';
            xInst_yInst_ch1_state <= b"0100";
         when b"0100" =>
            xInst_yInst_ch1_re <= '0';
            xInst_yInst_ch1_state <= b"0101";
         when b"0101" =>
            xInst_yInst_ch1_re <= '1';
            xInst_yInst_ch1_state <= b"0110";
         when b"0110" =>
            xInst_yInst_ch1_re <= '1';
            xInst_yInst_ch1_state <= b"0111";
         when b"0111" =>
            xInst_yInst_ch1_re <= '0';
            xInst_yInst_ch1_state <= b"1000";
         when b"1000" =>
            xInst_yInst_ch1_re <= '1';
            xInst_yInst_ch1_state <= b"1001";
         when b"1001" =>
            xInst_yInst_ch1_re <= '0';
            xInst_yInst_ch1_state <= b"1010";
         when b"1010" =>
            xInst_yInst_ch1_re <= '1';
            xInst_yInst_ch1_state <= b"1011";
         when b"1011" =>
            xInst_yInst_ch1_re <= '0';
            xInst_yInst_ch1_state <= b"1100";
         when b"1100" =>
            xInst_yInst_ch1_re <= '0';
            xInst_yInst_ch1_state <= b"0001";
         when others => null;
      end case;
   end if;
end process;
```

Listing 15: The process implementation of the FSM M for hardware design in Figure 5.4.

as *state explosion*. The drawback of this problem is a significant waste of hardware resources which eventually lead to hardware system failure. For example, in Xilinx ISE, this common error *"ERROR: Portability:3 - This Xilinx application has run out of memory or has encountered a memory conflict..."* is reported after FPGA synthesis failure as a result of FSMs that are too big. In order to

avoid this problem, some characteristics of the FSM M are exploited in order to reduce too many state variables while also optimizing for significant cut-down of utilized hardware resources. Below the four types of optimizations are proposed. These optimizations provide trade-off between resource utilization and performance that can be used by the designer to explore the best solution space that meets the application requirements.

First Optimization

The first optimization (*opt1*) aims to reduce the number of FSM states by exploiting what in this work is referred to as *gaps* and the periodicity of the schedule in Figure 3.2. A gap occurs where there are stalls in the execution of a sink actor, more specifically, this refers to where the output signal $v = 0$ and there is no sink actor schedule (i.e. $\sigma(i, j, v) = \emptyset$). The gap can be of the three types namely a Startup Gap (SG), a Firing Gap (FG) and an Iteration Gap (IG). The SG occurs during the startup phase of the system and is computed as $SG = \sigma(0, 0, v)$ where $\sigma(0, 0, v)$ is the initial schedule of the sink actor. For example, in Figure 3.2 the $SG = 3$ and this occurs from clock cycle 1 to 3. FG is the time delay between two consecutive sink actor firings in one iteration and is determined as $FG = \mu(v) - ET(v)$ where $\mu(v)$ is the sink actor scheduling period and $ET(v)$ represents the *execution time* of the sink actor. FG is 0 in Figure 3.2 as $\mu(v)$ is equal to $ET(v)$. Moreover, IG refers to the time delay between two consecutive sink schedule iterations and can be computed using $FG = T - (\mu(v) \times (RV(v) - 1)) + ET(v)$ where T is the schedule period and $RV(v)$ is the repetition vector of the sink actor. In Figure 3.2, IG is 2 and this occurs at clock cycles 14 and 15. All the three gaps SG, FG and IG use respective counters sgc, fgc and igc to create a timing delay.

Another feature of the schedule to exploit is the periodicity of the output signal r. The periodic sequence of r has the length that is equivalent to $ET(v)$ and is repeated $RV(v)$ times per iteration. Instead of creating $ET(v) \times RV(v)$ states needed where the sink actor does not stall (i.e. where it executes), this number of states is reduced to $ET(v)$ states by using a counter jc for sink actor firings

Table 5.1: Types of states used for iterative optimization

State Type	Time Delay	Multiplicity	Has Transition Loop?	Condition for Next State Transition
M_0	$(i \mid CP_{i,c} = 1) + 1$	[1..*]	Yes	$v = 1$
M_{sg}	SG	[0..*]	Yes	$sgc = SG$
M_k	$ET(v) \times RV(v)$	1..1	No	$jc < RV(v)$ or $jc = RV(v)$
M_{fg}	FG	[0..*]	Yes	$fgc = FG$
M_{ig}	IG	[0..*]	Yes	$igc = IG$

in a schedule iteration. jc is incremented at the end of each firing, therefore, enabling firing states (i.e. where there are no gaps) to be iterated $RV(v)$ times. The five types of states which are used to implement the optimized version of the FSM M_{opt1} in this section are tabulated in Table 5.1. s_0 is the initial state, s_{sg} is the state type that is used for startup gap, s_k realizes periodic firing sequences as defined using a 1-periodic schedule, s_{fg} state type is used for firing gap and lastly the s_{ig} implements the iteration gap. Note that s_0 and s_{sg} occur in the startup phase while s_k, s_{fg} and s_{sg} constitute the periodic phase of the FSM. Each state type is associated with four properties namely time delay, multiplicity, whether it has a loop transition or not and the condition for the next state transition. The time delay specifies the number of clock ticks it takes for the present state of the particular state type to execute before the transition to the next state. In order to describe the types of vectors (i.e. single or two-dimensional vectors containing values of the output signals) corresponding to states per state type, the multiplicity that takes three forms is used. 1..1 specifies a one-dimensional vector of exactly one-clock delay. [0..∗] denotes a two-dimensional vector of output signals which may either be zero or many in a single state type. The last multiplicity of the form [1..∗] specifies a two-dimensional vector of output signals with at least one value.

Figure 5.7 shows a generalized and optimized FSM M_{opt1} which optimizes FSM M example in Figure 5.6. The optimized version M_{opt1} begins with the state type s_0 where the output signals r and v of its sub-states are deasserted. In the provided example, the loop transition occurs once when $v = 0$ resulting in a two-dimensional vector (i.e. [000]). When $v = 1$, the FSM changes the state type to s_{sg} which creates a time delay of SG clock cycles. The sub-states of this state type have the output signals r and v all set to 0. In the example, $SG = 0$

therefore the FSM does not have s_{sg} and it will transition directly from s_0 to M_k state type. The M_k has $ET(v)$ sub-states (i.e. denoted in a dotted transition line between M_k and s_{et-1} where $k = \{0...ET(v) - 1\}$) which repeat $RV(v)$ times. The last sub-state s_{et-1} of state type s_k can use one of the two transitions, the first one leads to state type s_{fg} that creates time delay of FG and the second transition directs the FSM to s_{ig} where it delays execution for IG clock cycles. In the example provided, the s_k type undergoes $ET(y) \times RV(y) = 5 \times 2 = 10$ sub-states without any firing gaps (i.e. $FG = 0$) while only experiencing time delays created by iteration gaps (i.e. $IG = 2$).

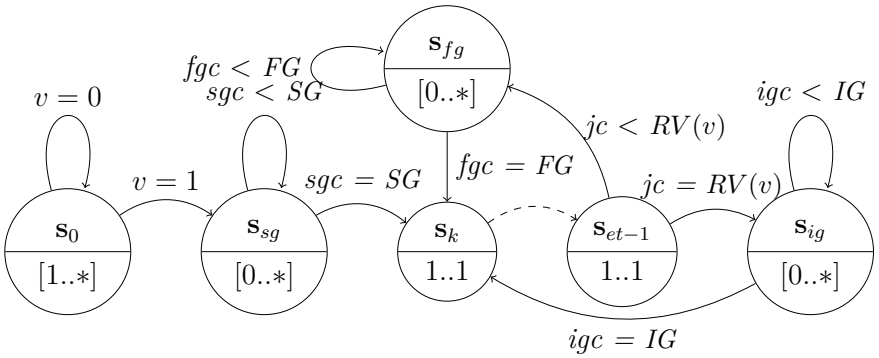


Figure 5.7: The generic optimized composite FSM M_{opt1} as constructed using FSMs M_X, M_{FF} and M_Y by using the computed 1-periodic schedule in Figure 3.2. Each node is the state type which may consists of sub-states which are defined by a sequence of one or two-dimensional vectors in lower half of the state type.

When described in hardware as shown in the VHDL process in Listing 16, the optimized FSM M_{opt1} is reduced to seven states in comparison to a classical FSM M in Figure 5.6. The startup phase only has a single state (i.e "000") followed by the periodic phase with the six states where the first four states are of type s_{sk} (i.e. "001", "010", "011", "100", "101"). The count register **xInst_yInst_ch1_jc** which is initially set to zero, increments until it reaches $RV(y) = 2$ where it moves the FSM from s_{sk} state type to s_{ig} state type. The iteration gap is realized in state type s_{ig} (i.e. "110") with the count register **xInst_yInst_ch1_igc** that has a threshold of 2.

```
xInst_yInst_ch1_proc : process(rst, clk)
begin
   if rst = '1' then
       ... --reset process registers
   elsif clk'EVENT and clk = '1' then
      ... -- hidden code
      case xInst_yInst_ch1_state is
         when b"000" =>
            xInst_yInst_ch1_re <= '0';
            xInst_yInst_ch1_state <= b"000";
            if xInst_vld_o = '1' then
               xInst_yInst_ch1_re <= '1';
               xInst_yInst_ch1_state <= b"010";
            end if;
         when b"001" =>
            xInst_yInst_ch1_re <= '1';
            xInst_yInst_ch1_state <= b"010";
         when b"010" =>
            xInst_yInst_ch1_re <= '0';
            xInst_yInst_ch1_state <= b"011";
         when b"011" =>
            xInst_yInst_ch1_re <= '1';
            xInst_yInst_ch1_state <= b"100";
         when b"100" =>
            xInst_yInst_ch1_re <= '0';
            xInst_yInst_ch1_state <= b"101";
         when b"101" =>
            xInst_yInst_ch1_re <= '1';
            xInst_yInst_ch1_state <= b"001";
            xInst_yInst_ch1_jc <= xInst_yInst_ch1_jc + 1;
            if xInst_yInst_ch1_jc = 1 then
               xInst_yInst_ch1_jc <= 0;
               xInst_yInst_ch1_state <= b"110";
            end if;
         when b"110" =>
            xInst_yInst_ch1_re <= '0';
            xInst_yInst_ch1_state <= b"110";
            xInst_yInst_ch1_igc <= xInst_yInst_ch1_igc + 1;
            if xInst_yInst_ch1_igc = 1 then
               xInst_yInst_ch1_igc <= 0;
               xInst_yInst_ch1_state <= b"001";
            end if;
         when others => null;
      end case;
   end if;
end process;
```

Listing 16: The process implementation of the second optimization FSM M_{opt1} for hardware design in Figure 5.4.

Second Optimization

The first optimization technique in Section 5.3.6 works effectively in systems where access patterns are short by exploiting the gaps and periodicity of the SDF-AP schedule. However, most real-world applications are often characterized by very long access patterns, this implies multiple sub-states of state type M_k which

lead to a *state explosion problem*. The second optimization (*opt2*) alleviates this problem by grouping all the chained sub-states of M_k into one state. The corresponding hardware implementation involves the LUT of the consumption pattern ($CP(c)$) which is indexed with a digital counter where each element of the LUT is assigned to the read-enable output signal r in the same state type but at different clock cycles. As depicted in Figure 5.8, the optimized FSM M_{opt2} has a state type M_k that now uses multiplicity of $[1..*]$ and that has loop transition to enable the traversing of $ET(v)$ elements of a $CP(c)$. Two counters are used to control data access on the FIFO. The first counter ic counts the number of states in a single firing period of $ET(v)$ after which the transition from M_k state type to s_{fg} state type takes place. The second counter jc counts the total number of sub-states passed by the M_k state type in one schedule iteration. When its threshold (i.e. $jc = ET(v) \times RV(v)$) is reached, an M_{opt2} moves from M_k state type to s_{ig} state type.

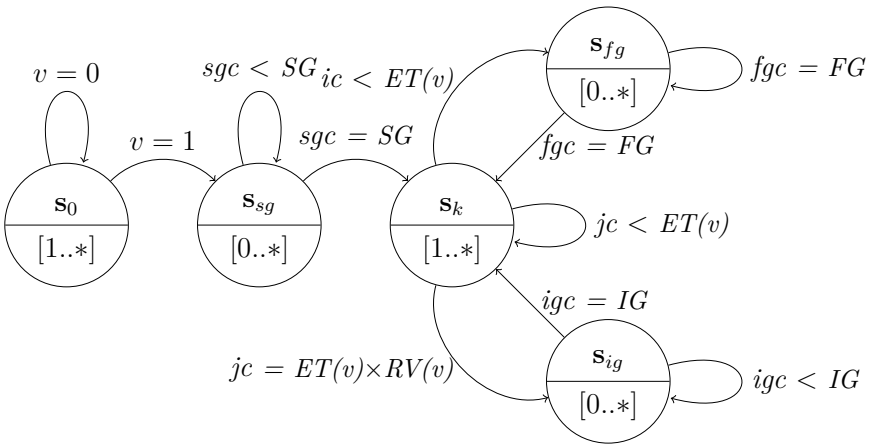


Figure 5.8: The generic optimized composite M M_{opt2} as constructed using Ms M_X, M_{FF} and M_Y by using the computed 1-periodic schedule in Figure 3.2. Each node is the state type which may consists of sub-states which are defined by a sequence of one or two-dimensional vectors in lower half of the state type.

Converting an optimized FSM M_{opt2} (shown in Figure 5.8) into hardware design results in VHDL process code that is shown in Listing 17. In comparison to the M_{opt1} VHDL process in Section 5.3.6, the number of states for the M_{opt2} are reduced from seven to three. This reduction is facilitated by the **xInst_yInst_ch1_cp** $CP(y)$ that maps a reversed $CP(y)$ (i.e. "10101") in

```
... -- hidden architecture code
constant xInst_yInst_ch1_cp :
  ↪  std_logic_vector(4 downto 0) :=
  ↪  b"10101";
... -- hidden code
xInst_yInst_ch1_proc : process(rst,
  ↪  clk)
begin
   if rst = '1' then
      --reset process registers
   elsif clk'EVENT and clk = '1' then
      -- hidden code
      case xInst_yInst_ch1_state is
         when b"00" =>
            xInst_yInst_ch1_re <= '0';
            xInst_yInst_ch1_state <=
             ↪  b"00";
            if xInst_vld_o = '1' then
               xInst_yInst_ch1_re <=
                ↪  xInst_yInst_ch1_cp(xInst_yInst_ch1_ic);
               xInst_yInst_ch1_ic <=
                ↪  xInst_yInst_ch1_ic
                ↪  + 1;
               xInst_yInst_ch1_jc <=
                ↪  xInst_yInst_ch1_jc
                ↪  + 1;
               xInst_yInst_ch1_state
                ↪  <= b"01";
            end if;
         when b"01" =>
            xInst_yInst_ch1_state <=
             ↪  b"01";
            xInst_yInst_ch1_ic <=
             ↪  xInst_yInst_ch1_ic +
             ↪  1;
            xInst_yInst_ch1_re <=
             ↪  xInst_yInst_ch1_cp(xInst_yInst_ch1_ic);
            if xInst_yInst_ch1_ic = 4
             ↪  then
               xInst_yInst_ch1_ic <=
                ↪  0;
               xInst_yInst_ch1_state
                ↪  <= b"01";
            end if;
            xInst_yInst_ch1_jc <=
             ↪  xInst_yInst_ch1_jc +
             ↪  1;
            if xInst_yInst_ch1_jc = 9
             ↪  then
               xInst_yInst_ch1_jc <=
                ↪  0;
               xInst_yInst_ch1_state
                ↪  <= b"10";
            end if;
         when b"10" =>
            xInst_yInst_ch1_re <= '0';
            xInst_yInst_ch1_state <=
             ↪  b"10";
            xInst_yInst_ch1_igc <=
             ↪  xInst_yInst_ch1_igc +
             ↪  1;
            if xInst_yInst_ch1_igc = 1
             ↪  then
               xInst_yInst_ch1_igc <=
                ↪  0;
               xInst_yInst_ch1_state
                ↪  <= b"01";
            end if;
         when others => null;
      end case;
   end if;
end process;
```

Listing 17: The process implementation of the second optimization FSM M_{opt2} for hardware design in Figure 5.4.

LUT using a big-endian style. At the transition from a periodic phase (i.e. comprises a single sub-state “00") to the periodic phase (i.e. comprises two states “01" and “10"), the least significant bit (LSB) of **xInst__yInst__ch1__cp** is assigned to read-enable signal **xInst__yInst__ch1__re** together with the activation

of the ic, jc counters. The second state (i.e "01") creates the M_k state type of the FSM with the aid of the two count registers namely **xInst_yInst_ch1_ic** and **xInst_yInst_ch1_jc**. The indexing of individual bits of LUT is made possible by using the register **xInst_yInst_ch1_ic** which counts from 0 to $ET(y) - 1 = 5 - 1 = 4$ where the actor firing terminates. Another count register **xInst_yInst_ch1_jc** detects the end of iteration when the threshold of $(ET(y) \times RV(y)) - 1 = (5 \times 2) - 1 = 10 - 1 = 9$ is reached. The second state transitions to the iteration gap state (i.e. "10") which introduces a time delay of $(IG = 2)$ at the end of every iteration.

Third and Fourth Optimizations

Implementing FIFO buffers along with their controllers have both the advantages and disadvantages in the designed hardware. The advantages are that the buffers break long paths, enable pipeline, avoid deadlocks and allow throughput-constraint scheduling to be achieved via pipeline stalls. The disadvantages are that they result in a waste of both memory and logic resources on the FPGA with increased latency. On the contrary, the buffer-free designs are fast and conservative in utilizing hardware resources, however, they often lead to combinational data-paths that limit the clock speed [168]. To strike the balance of buffer-based and buffer-free data-paths, the hardware design is optimized further by removing buffers where the buffer size is 1 while the logic controllers remain unchanged. The buffers are simply replaced by registers along with the assertion of v whenever the single data sample is available on the input port. The third (*opt3*) and fourth (*opt4*) optimizations in this section are buffer-free versions of the first optimization *opt1* in Section 5.3.6 and second optimization *opt2* in Section 5.3.6 respectively.

5.4 Conformance Analysis

This section presents a formal analysis of the proposed hardware implementation method in Section 5.3 proving that it accurately produces correct systems according to specifications using SDF-AP model. The approach in this work has similarities to conformance analysis technique proposed in [151] which targets generic dataflow models, however, ours is different in that it focuses only on SDF-AP model. It is noteworthy that this conformance analysis is based on the correct behavior of the predefined IP cores and the correctness of their extracted access patterns as required by the SDF-AP model. The aim is to bridge the gap between model for a dataflow (represented as a *closed transition system*) in Section 3.3 and a model for hardware (represented as a *closed finite-state machine* in Section 5.3.4). Due to semantic differences of both models, the execution properties are identified and they should remain unchanged during conversion from a dataflow model to the hardware model. These properties are statically determined by the SDF-AP model at compile time and include a buffer size, throughput and latency and they all expected to be correctly converted into a hardware model.

The buffer size as computed in Section 3.2.2, is allocated to channels by the SDF-AP model and it matches the physical memory size of the corresponding hardware. Given sufficient memory resources on the target FPGA, the infinite buffer size ensures the non-overflow buffers and a deadlock-free hardware system. Computing the throughput using both the dataflow and hardware model is performed with respect to the sink actor. For the SDF-AP model, the transition model is used to compute throughput by counting the number of *gets* per number of ticks in a periodic phase. For example, using Figure 3.4, the number of *gets* is 6 while the *tick* count is 12 resulting in throughput 6/12. For the FSM, the throughput is determined by counting the number of states with asserted *valid* output signal value (i.e. $v = 1$) per total number of transitions in a periodic phase. For example, in Figure 5.6, the number states with $v = 1$ are 6 and the number of transitions is 12, as a result the throughput becomes 6/12. Furthermore, the latency of a transitions system is obtained by counting all the *tick*

transitions and the latency of the FSM equals the number of all transitions. For example, the number of *ticks* of a transition system in Figure 3.4 is 13, likewise, the transition count for FSM in Figure 5.6 is 13.

While the access patterns may arguably move the SDF-AP model closer to a hardware by describing at which clock cycles the tokens are produced and consumed, the behaviour of an SDF-AP model is asynchronous making it difficult to directly compare with a synchronous FSM. On the other hand, it is not easy to observe where token productions and consumptions take place by merely looking at the state transitions. One common approach to defining conformance is using containment of set of behaviours of the two disparate models. This principle is simply applied in the SdrLift setting as in [151] by proposing FSM model N conforms to dataflow model ω_M if the set of behaviours of N is a subset of the set of behaviours of M. However, due to the deterministic behaviour of the SDF-AP model (under a 1-periodic schedule and throughput constraints), there is exactly one observable behaviour for FSM M (i.e. ω_M) and exactly one observable behaviour for a dataflow model N (i.e. ω_N). This leads to the conformance formulation Ω which maps an observable behaviour of FSM ω_M to an observable behaviour of a dataflow model ω_N as

$$\Omega : \omega_M \mapsto \omega_N = \rho_0 \cdot \rho_1 \cdot \rho_1 \cdot \cdot\cdot$$

where $\rho_i = \mathsf{tick} \cdot \alpha_i$ and $\alpha_i := \{\ell \in \{\mathsf{put}, \mathsf{get}\}\}$ and the mapping $\psi : \{\mathsf{put}, \mathsf{get}\} \mapsto \{w, v\} \mid w = 1, v = 1$ maps the dataflow actions $\ell = \{\mathsf{put}, \mathsf{get}\}$ to respective FSM output signals output signals $O = \{\mathsf{w}, \mathsf{v}\}$ where their values are set to 1 (i.e. therefore resulting in $\psi = \{\mathsf{put} \mapsto w, \mathsf{get} \mapsto v\}$).

For example, consider the FSM observable behaviour as defined in Section 5.3.5

$$\omega_M = (\bar{\mathsf{w}}\mathsf{r}\bar{\mathsf{v}})\Big((\mathsf{w}\mathsf{r}\bar{\mathsf{v}})(\mathsf{w}\mathsf{r}\bar{\mathsf{v}})(\bar{\mathsf{w}}\mathsf{r}\bar{\mathsf{v}})(\bar{\mathsf{w}}\mathsf{r}\bar{\mathsf{v}})(\mathsf{w}\mathsf{r}\bar{\mathsf{v}})(\mathsf{w}\mathsf{r}\bar{\mathsf{v}})(\bar{\mathsf{w}}\mathsf{r}\mathsf{v})(\bar{\mathsf{w}}\mathsf{r}\bar{\mathsf{v}})(\mathsf{w}\mathsf{r}\bar{\mathsf{v}})(\mathsf{w}\mathsf{r}\bar{\mathsf{v}})(\bar{\mathsf{w}}\mathsf{r}\bar{\mathsf{v}})\bar{\mathsf{w}}\mathsf{r}\mathsf{v})\Big)^{\infty}.$$

This ω_M is mapped using Ω to

$$\Omega(\omega_M) = \mathsf{tick} \cdot \Big(\mathsf{tick} \cdot \{\mathsf{put}\} \cdot \mathsf{tick} \cdot \{\mathsf{put}\} \cdot \mathsf{tick} \cdot \{\mathsf{get}\} \cdot \mathsf{tick} \cdot \mathsf{tick} \cdot \{\mathsf{put}, \mathsf{get}\} \ \ldots \ \mathsf{tick} \cdot \{\mathsf{get}\}\Big)^{\infty}$$

which is equivalent to dataflow network observable behaviour ω_N that is defined in Section 3.3.

5.5 Chapter Summary

This chapter elaborates gateware generation in SdrLift Compiler that results in VHDL code. First the HW Blocks are generated from the IR_{FG} and IR_{CG}, followed by top-level design of the SDR application once the SDF-AP model analyses have been successful. The low-level model implementation is realized in FSMs and the four optimizations are applied to obtain efficient results in the code generation process. The chapter concludes with the design process that verifies the conformation of the created system with high-level program specifications. In Chapter 6 the capability of SdrLift is demonstrated by developing the DSP IP cores for which SdrLift generates a synthesizable VHDL code. The results show that good performance and design flexibility can be achieved using SdrLift, while also conserving hardware resources in the subsequent implementation.

Chapter 6

Experimental Results[1]

This chapter presents the experimental results of applying the SdrLift design flow to a list of small to large scale representative SDR applications that are constrained by throughput values specified in the SdrLift program. These experimental results lead to the objective analysis of the main features of SdrLift which include, application description in SdrLift Language, model analyses, IR creation and transformations and code generation using SdrLift compiler. The measurement metrics used in the experimental evaluation are efficiency and Quality of Result (QoR). The efficiency entails the number of code lines and the measured design times comprising application modeling time, code generation time, and RTL synthesis time. The QoR measures the resource usage including the number of registers, the number of LUTs, and the number of logic slices. The QoR also includes power consumption and the maximum frequency of the design.

[1]This chapter is based in part upon the following publications:
L. Tsoeunyane, S. Winberg, and M. Inggs, "Software-defined radio FPGA cores: Building towards a domain-specific language," *International Journal of Reconfigurable Computing*, vol. 2017, 2017.
L. Tsoeunyane, S. Winberg, and M. Inggs, "Automatic configurable hardware code generation for software-defined radios," *Computers*, vol. 7, no. 4, 2018.
L. Tsoeunyane, S. Winberg, and M. Inggs, "Sdrlift: An intermediate-level framework for synthesis of software-defined radio accelerators," in *2019 IEEE 10th International Conference on Mechanical and Intelligent Manufacturing Technologies (ICMIMT)*, pp. 166–173, Feb 2019.

6.1 Experimental Results

This section presents experimental results of the design and implementation of eight representative SDR applications using SdrLift intermediate compiler framework. These applications comprise the two OFDM-TXs which are both based on a modified IEEE 802.11a standard [169] and IEEE 802.22 standard [170] respectively and the two OFDM-RXs which are also based on IEEE 802.11a standard and IEEE 802.22 standard respectively. Additionally, the MIMO system is implemented in combination with OFDM which is based on IEEE 802.11a standard, for which the complete system is referred to as MIMO OFDM in the results [171]. The MIMO OFDM system is composed of two typical SDR subsystems namely the MIMO OFDM transmitter (MIMO OFDM-TX) and receiver (MIMO OFDM-RX) where each of these have four output ports and four input ports respectively. The last two applications derive from a DDC whereby the first one implements FM (i.e. the FM-DDC design) and the second one realizes a Global System for Mobile Communications (GSM) design (i.e. GSM-DDC).

The IP cores that compose the applications are synthesized with SdrLift, some were previously coded by hand in VHDL and others are obtained from the Xilinx core library. The model parameters (i.e. access patterns, execution time, rates) for SdrLift-developed cores are computed at compilation. For existing IP cores, the relative parameters are determined through low-level simulations using the Xilinx ISim Simulator. In some cases, where the third-party IP cores are incorporated into the implementation, the data-sheets documentation of the relative cores are used to determine the appropriate model parameters.

6.1.1 Applications

The eight SDR applications, namely OFDM-TX (IEEE 802.11a), OFDM-RX (IEEE 802.11a), OFDM-TX (IEEE 802.22), OFDM-RX (IEEE 802.22), MIMO OFDM-TX (IEEE 802.11a), MIMO OFDM-RX (IEEE 802.11a), GSM-DDC and FM-DDC are depicted in Figures 6.1– 6.8. Note that for each figure, the blue sharp rectangles represent the pre-coded IP cores in RTL code (i.e. VHDL or

Table 6.1: The execution properties of different SDR applications

Application	Number of Actors	# FIFO Channels	# FIFO Channels with Buffer Size = 1
OFDM-TX(IEEE 802.11a)	6	15	7
OFDM-RX(IEEE 802.11a)	6	15	5
OFDM-TX(IEEE 802.22)	6	15	7
OFDM-RX(IEEE 802.22)	6	15	5
MIMO OFDM-TX(IEEE 802.11a)	22	59	28
MIMO OFDM-RX(IEEE 802.11a)	22	59	16
GSM-DDC	11	23	10
FM-DDC	13	27	10

Verilog) while the red rounded rectangle denote the IP cores in witch their VHDL is automatically created in SdrLift. The SDF-AP properties of each application are summarized in Table 6.1 and are also displayed in each figure using the two rows each with various property labels next the corresponding HW Block. The first row shows the output model proprieties (**O**) of the blocks and the second row represents the input model properties (**I**) of the blocks. These properties include the number of Actors, the total number of FIFO channels and the number of FIFO channels which are allocated the buffer size of 1. The applications are described in the subsections that follow.

OFDM Transmitter (IEEE 802.11a)

This application implements the physical layer of the OFDM transmitter for the IEEE 802.11a standard. The SDF-AP graph is shown in Figure 6.1. The application code for this OFDM-TX is shown in Listing 26 of Appendix A. As shown in Listing 26, the application stitches together the modules and macros declared in Lines 2 – 5 using the *Chain* topological pattern in Lines 7 – 54.

The OFDM-TX (IEEE 802.11a) SDF-AP actors are described below:

- **SRC:** The source actor produces a frame of 48 real-valued data samples

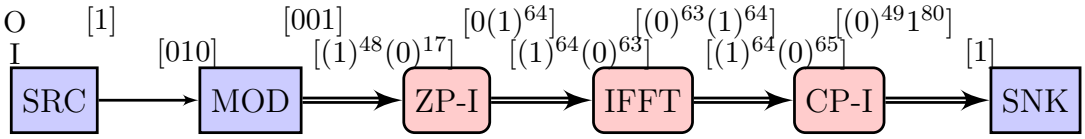


Figure 6.1: OFDM-TX (IEEE 802.11a).

at the rate of one sample on every cycle of 48 successive clock cycles using the output pattern [1].

- **Mod:** This is a Quadrature Amplitude Modulation (16-QAM) block that uses pattern [010] to receive the frame where a single data sample is consumed on every second cycle. The *Mod* modulates the 48 data samples from the source block into 48 In-phase and Quadrature (I/Q) samples in a frequency domain and outputs them on every third clock cycle using the pattern [001].

- **ZP-I:** This is a ZP-I actor that appends 16 zeros to the 48 samples which are consumed with pattern $[(1)^{48}(0)^{17}]$, whereafter the ZP-I produces 64 samples with pattern $[0(1)^{64}]$.

- **IFFT:** A 64-point IFFT block receives 64 sample frame with pattern $[(1)^{64}(0)^{64}]$ (i.e. 64 samples are consumed in the first 64 cycles of ET=128). The IFFT transforms the 64 samples from the frequency domain to the time domain and sends out the output samples with pattern $[(0)^{64}(1)^{64}]$ (i.e. 64 samples are produced in the last 64 cycles of ET=128).

- **CP-I:** The CP-I block prepends the cyclic prefix (last 16 IFFT samples) to the 64 IFFT samples received with pattern $[(1)^{64}(0)^{65}]$. The 80 samples are then produced by the CP-I using pattern $[(0)^{49}(1)^{80}]$.

- **Sink:** The sink actor which receives the 80 sample frame at the rate of one sample per cycle using an input access pattern of [1].

OFDM Transmitter (IEEE 802.11a)

This application implements the physical layer of the OFDM receiver for the IEEE 802.11a standard. The corresponding SDF-AP receiver system is shown in Figure 6.2 as designed using SdrLift.

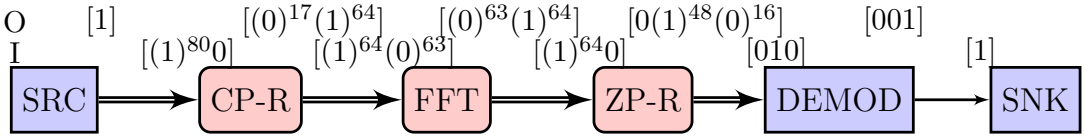


Figure 6.2: OFDM-RX (IEEE 802.11a).

The OFDM-RX (IEEE 802.11a) SDF-AP actors are described below:

- **SRC:** The source block produces 80 sample OFDM frame at the rate of one sample on every cycle of 80 successive clock cycles using the output pattern [1].
- **CP-R:** The CP-R receives the samples with pattern $[(1)^{80}0]$. For each 80 sample OFDM frame, the CP-R removes the 16 samples of a cyclic prefix to produce 64 samples using pattern $[(0)^{17}(1)^{64}]$.
- **FFT:** The 64-point FFT block consumes 64 samples from CP-R using pattern $[(1)^{64}(0)^{64}]$. It then transforms the samples from the time-domain back to 64 frequency domain samples which are output with pattern $[(0)^{64}(1)^{64}]$.
- **ZP-R:** The zeropad removal (**ZP-R**) accepts 64 FFT samples with pattern $[(1)^{64}0]$, and then detaches the last 16 zero samples from the FFT samples to produce 48 data samples with pattern $[0(1)^{48}(0)^{16}]$.
- **Demod:** This is a 16-QAM demodulation (**Demod**) which demodulates the incoming 48 frequency-domain I/Q samples (on input port with pattern [010]) back to real-valued 48 samples (sent via the output port with pattern [001]) before feeding them into a sink block.
- **SNK:** The sink actor receives the 48 sample demodulated frame at the rate of one sample per cycle using an input access pattern of [1].

OFDM-TX (IEEE 802.22)

This application implements the physical layer of the OFDM transmitter for the IEEE 802.22 standard. As shown in Figure 6.3, the structure of IEEE 802.22 standard transmitter system is similar to the IEEE 802.11a transmitter in Figure 6.1 with different model parameter.

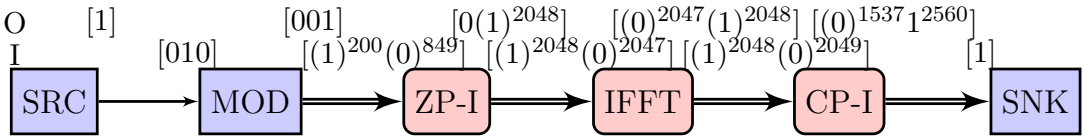


Figure 6.3: OFDM-TX (IEEE 802.22).

The SDF-AP model actors and the corresponding model parameters are described below:

- **Mod:** This block receives 1200 samples from the source block with pattern [010] and modulates the samples to produce 1200 frame with pattern [001].
- **ZP-I:** This actor accepts the 1200 sample frame with the input pattern $[(1)^{1200}(0)^{849}]$, then appends 848 zero samples to the modulated samples to produce 2048 sample with output pattern $[0(1)^{2048}]$). which input into a 2048-point IFFT.
- **IFFT:** This is 2048-point IFFT which receives 2048 zero-padded samples using input pattern $[(1)^{2048}(0)^{2048}]$, then transforms the samples into 2048 sample frame with output pattern $[(0)^{2048}(1)^{2048}]$.
- **CP-I:** The CP-I prepends a 512 sample cyclic prefix to the IFFT frame consumed with input pattern $[(1)^{2048}(0)^{2049}]$ produces a 2560 sample OFDM frame with output pattern $[(0)^{1537}(1)^{2560}]$.

OFDM-RX (IEEE 802.22)

The IEEE 802.22 standard receiver system is shown in Figure 6.4 and is similar to IEEE 802.11a receiver in Figure 6.2 except that it has different configurations

for the blocks.

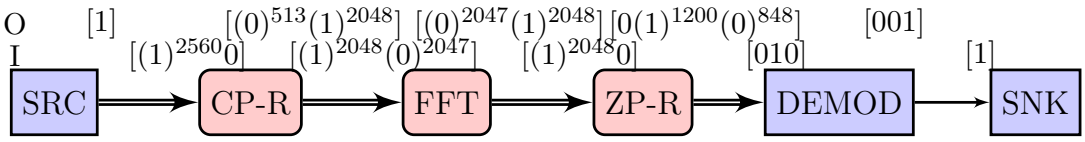


Figure 6.4: OFDM-RX (IEEE 802.22).

In the IEEE 802.22 receiver configuration, the SDF-AP actors are described as follows:

- **CP-R:** This actor with input pattern $[(1)^{2560}0]$ and output pattern $[(0)^{513}(1)^{2048}]$ is used to remove 512 samples of a cyclic prefix from the 2560 sample OFDM resulting in 2048 samples.
- **FFT:** The 2048-point FFT block uses input pattern $[(1)^{2048}(0)^{2048}]$ to receive the time-domain samples and transforms them into frequency-domain using output pattern $[(0)^{2048}(1)^{2048}]$.
- **ZP-R:** The 848 zeros of the FFT output are removed by the ZP-R to produce 1200 samples where ZP-R uses the input pattern $[(1)^{2048}0]$ and the output pattern $[0(1)^{1200}(0)^{848}]$.
- **Demod:** This is a 16-QAM demodulation (**Demod**) which demodulates the incoming 1200 frequency-domain I/Q samples (on input port with pattern [010]) back to real-valued 1200 samples (sent via the output port with pattern [001]) before feeding them into a sink block.

MIMO OFDM-RX (IEEE 802.11a)

The MIMO OFDM-TX system is shown in Figure 6.5 and the building blocks for its four transmit paths operate in a similar manner as the corresponding blocks for the IEEE 802.11a transmitter in Figure 6.1. The only new block in this system is a Serial to Parallel (S/P) block which converts the 192 sample serial stream into four parallel 48 sample streams for the transmit paths. This

S/P uses the access pattern $[(11110)^{48}]$ on its input port (i.e. consumes four samples every five cycles) and on each of its four output ports it produces data samples with pattern $[(00001)^{48}]$ (i.e. produces one data sample on every fifth clock cycle).

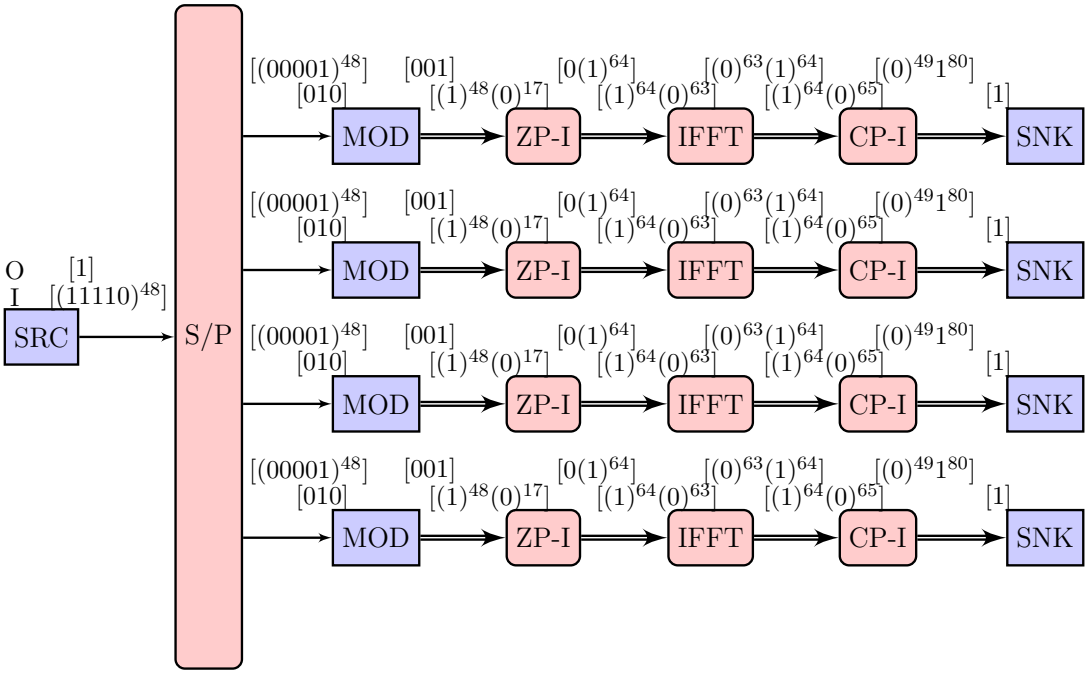


Figure 6.5: MIMO OFDM-TX (IEEE 802.11a).

MIMO OFDM-RX (IEEE 802.11a)

The MIMO OFDM-RX system is illustrated in Figure 6.6 and the blocks for each of the four receive paths operate in the same way as the corresponding blocks for IEEE 802.11a standard receiver in Figure 6.2. The only exception is the newly added Parallel to Serial (P/S) block which serializes the four parallel 48 sample data streams to a single stream of 192 samples. On each port of the four input ports, the P/S consumes the data samples with the access pattern $[(10000)^{48}]$ (i.e. one data sample is consumed every fifth cycle starting from the first cycle) and it uses pattern $[(01111)^{48}]$ on its output port (i.e. i.e. produces four samples every five cycles).

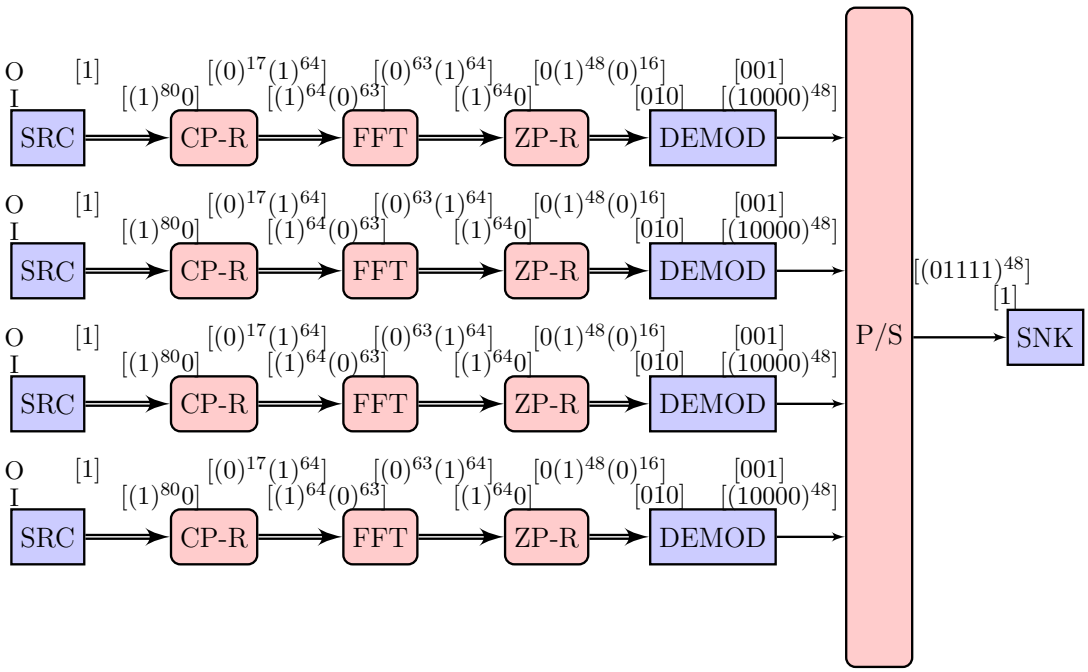


Figure 6.6: MIMO OFDM-RX (IEEE 802.11a).

GSM-DDC

The GSM-DDC as shown in Figure 6.7 accepts a high sample-rate (69.33 Mega Samples Per Second (MSPS)) bandpass signal from the source block which produces one sample every five cycles using pattern [00001]. The data produced by the Numerically Controlled Oscillator (NCO) with pattern [00001] is mixed with bandpass data to produce a low sample-rate (270.83 Kilo Samples Per Second (KSPS)) data stream. Note that the mixing process is performed by a digital mixer block with the input access pattern [10] and the output access pattern [01]. The CIC block performs a decimation of factor 256 by receiving the 256 samples with pattern $[(1)^{256}(0)^2]$ (i.e. consumes 256 samples at the rate of one data sample on every cycle of $ET = 256$) and produces one sample every 256 cycles using pattern $[(0)^{257}1]$. The rest of the blocks use pattern [1] for both input and output ports. Lastly, the compensation of the CIC signal is performed by a CFIR which is followed by a Programmable FIR filter (PFIR) that finalizes the filtering process.

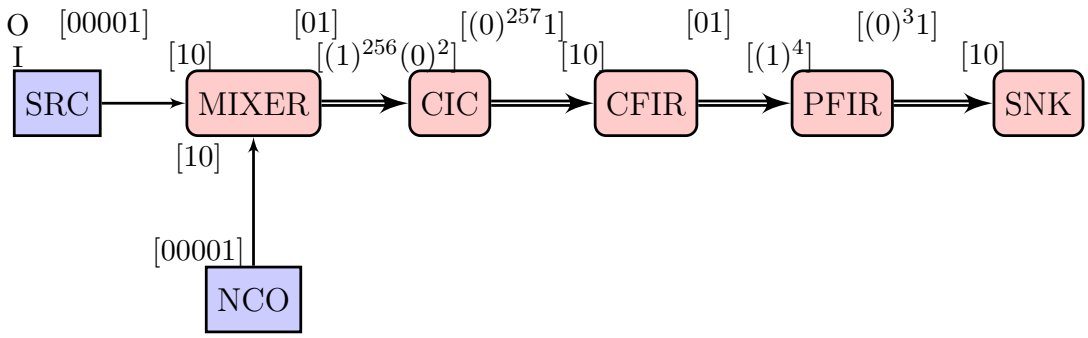


Figure 6.7: GSM-DDC.

FM-DDC

The FM-DDC in Figure 6.8 accepts the high sample-rate (81.92 MSPS) bandpass signal and produces a low sample-rate (160 KSPS) signal. The decimation factor of 512 is facilitated by the two CIC filters CIC1 = input patttern $[(1)^{128}(0)^2]$ and output pattern $[(0)^{129}1]$, and glscic2 = input patttern $[(1)^4]$ and output pattern $[(0)^3 1]$) with respective decimation factors of 128 and 4. Each CIC filter is followed by a CFIR1 and CFIR2 respectively) which improves the corresponding CIC output signal.

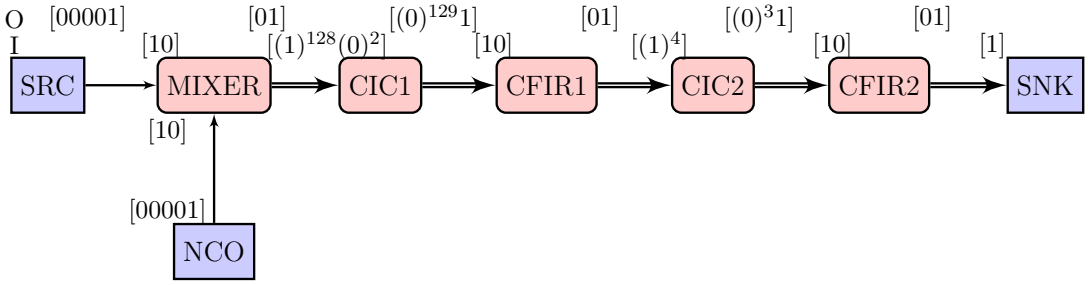


Figure 6.8: FM-DDC

6.1.2 Experimental Procedure

Each of the eight SDR applications is associated with ten design variants (range: **V1** – **V10**) which are generated under ten throughput constraints which are 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% and 100% of the maximum

throughput for each application as shown in Table 6.2. For each application, the throughput is relative to a sink actor where the maximum throughput τ_{max} is determined as described in Section 3.2.2. The results of different throughput constraints may be similar, in which case the **All** column (range: **T1** – **T8**) of Table 6.2 is used to to group all the design variants for each application. The throughput is presented to be measured in SPC . SPC can clearly be translated into the more standard Samples Per Second (SPS) units by multiplying the value by the clock frequency. However, the SPC is chosen to be used as it is a more FPGA independent measure – for instance, an FPGA that can support a higher clock rate will correspondingly achieve a higher throughput. A real example is using the maximum throughput value of IEEE 802.11a OFDM-TX which is 0.007752 SPC together with DAC (i.e. 16-bit I/Q sink actor) driven by the FPGA at the clock speed that meets standard transmission data rate of 54 Mega Bits Per Second (Mbps). By choosing the DAC clock speed of 218 MHz, the practical data rate can be computed as 0.007752 SPC × 218 MHz × (2×16-bit I/Q sample) = 54.078 Mbps that equals the standard data rate.

Table 6.2: The throughput constraints for SDR applications.

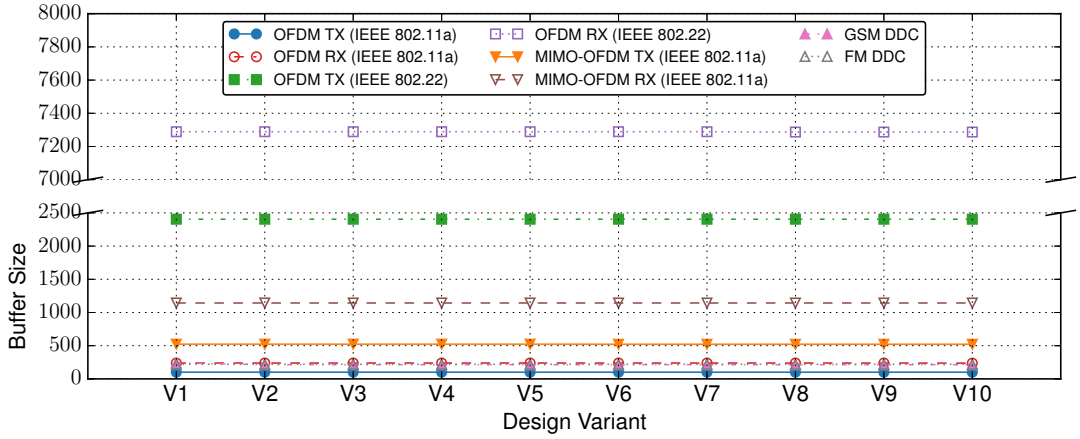
		Design Variants									
Application	**All**	**V1**	**V2**	**V3**	**V4**	**V5**	**V6**	**V7**	**V8**	**V9**	**V10**
OFDM-TX (IEEE 802.11a)	**T1**	0.000775 (10%)	0.001550 (20%)	0.0023260 (30%)	0.003101 (40%)	0.003876 (50%)	0.004651 (60%)	0.005426 (70%)	0.006202 (80%)	0.006977 (90%)	0.007752 (100%)
OFDM-RX (IEEE 802.11a)	**T2**	0.000787 (10%)	0.001575 (20%)	0.002362 (30%)	0.003150 (40%)	0.003937 (50%)	0.004724 (60%)	0.005512 (70%)	0.006299 (80%)	0.007087 (90%)	0.007874 (100%)
OFDM-TX (IEEE 802.22)	**T3**	0.000024 (10%)	0.000049 (20%)	0.000073 (30%)	0.000098 (40%)	0.000122 (50%)	0.000146 (60%)	0.000171 (70%)	0.000195 (80%)	0.000220 (90%)	0.000244 (100%)
OFDM-RX (IEEE 802.22)	**T4**	0.000024 (10%)	0.000049 (20%)	0.000073 (30%)	0.000098 (40%)	0.000122 (50%)	0.000146 (60%)	0.000171 (70%)	0.000195 (80%)	0.000220 (90%)	0.000244 (100%)
MIMO OFDM-TX (IEEE 802.11a)	**T5**	0.000417 (10%)	0.000833 (20%)	0.001250 (30%)	0.001667 (40%)	0.002083 (50%)	0.002500 (60%)	0.002917 (70%)	0.003333 (80%)	0.003750 (90%)	0.004167 (100%)
MIMO OFDM-RX (IEEE 802.11a)	**T6**	0.000417 (10%)	0.000833 (20%)	0.001250 (30%)	0.001667 (40%)	0.002083 (50%)	0.002500 (60%)	0.002917 (70%)	0.003333 (80%)	0.003750 (90%)	0.004167 (100%)
GSM-DDC	**T7**	0.000769 (10%)	0.001538 (20%)	0.002308 (30%)	0.003077 (40%)	0.003846 (50%)	0.004615 (60%)	0.005385 (70%)	0.006154 (80%)	0.006923 (90%)	0.007692 (100%)
FM-DDC	**T8**	0.000769 (10%)	0.001538 (20%)	0.002308 (30%)	0.003077 (40%)	0.003846 (50%)	0.004615 (60%)	0.005385 (70%)	0.006154 (80%)	0.006923 (90%)	0.007692 (100%)

Buffer Size and Latency Computation

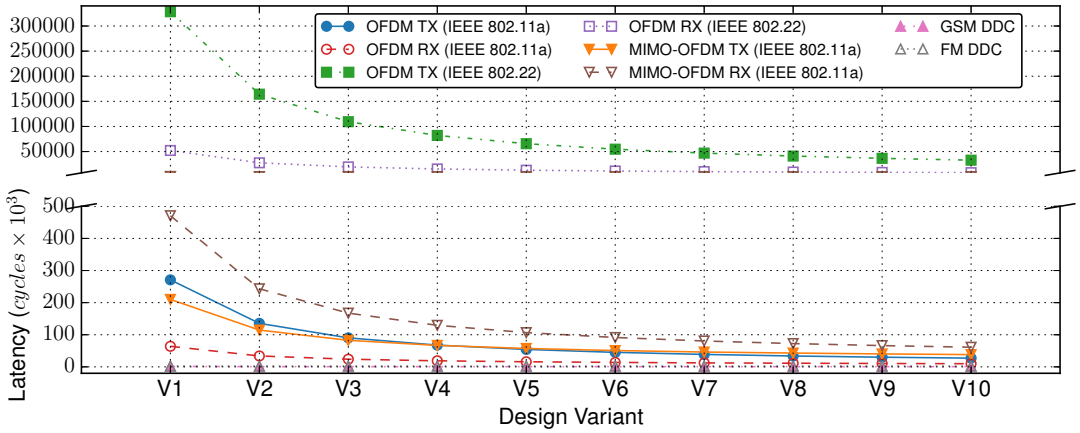
The system properties of the design variants as per application which is computed during SDF-AP analysis include the buffer size and latency as depicted in Figure 6.9. The computed buffer size is the sum of the allocated buffer sizes for all FIFO channels in each application and this sum corresponds with a single throughput constraint as shown in Figure 6.9a. For the most part, the total buffer size allocated to the FIFO channels of each application remains constant and relatively decreases with the increased throughput. For example, OFDM-TX and OFDM-RX of IEEE 802.11a have the highest throughput constraint values which result in the smallest buffer sizes. Similarly, the OFDM-TX and OFDM-RX of IEEE 802.22 have the lowest throughput constraint values which lead to the largest buffer size allocation. The reason for the increase of computed buffer size under low throughput constraint is explained in Section 3.2.2. Furthermore, the latency results are obtained as shown in Figure 6.9b. For all SDR applications, the latency decreases exponentially with increasing throughput.

FSM States

The FSM states are instrumental in realizing the low-level model that enables hardware code generation. The generated VHDL code exhibits a number of characteristics which include the number of FSM states, the number of code lines and the total length of time it takes to translate the SDF-AP model into VHDL code and build the code with the ISE tool-flow. Figure 6.10 depicts the results of the total number of FSM states for each SDR application using the non-optimized approach as explained in Section 5.3.5 and comparing it with the optimized versions *opt1*, *opt2*, *opt3*, and *opt4* which are discussed in Sections 5.3.6, 5.3.6 and 5.3.6 respectively. The number of FSM states (measured in millions of FSM states) for non-optimized applications as shown in Figure 6.10a is too large to be correctly implemented on the target FPGA. Although it seemingly decreases exponentially with increasing throughput values, it is still considered sub-optimal. The significant number of FSM states are undesirable in the SDR applications as they lead to state explosion problem and in this setting, all the non-optimized



(a) Buffer size

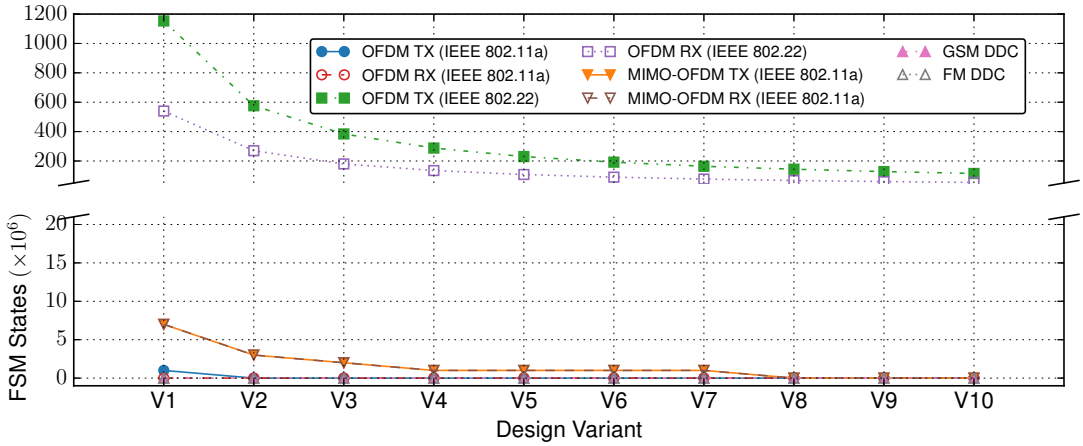


(b) Latency

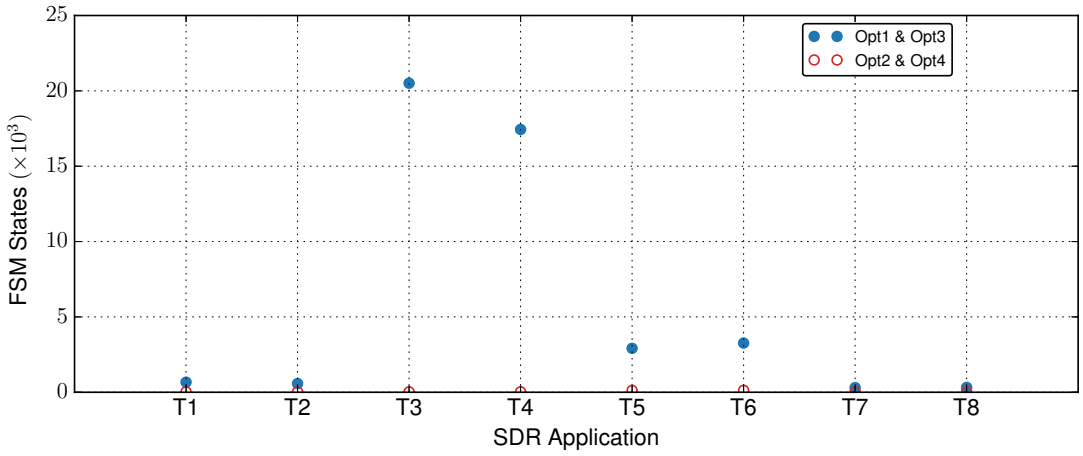
Figure 6.9: The results of SDF-AP analysis for the SDR applications showing the computed buffer size and latency of each application.

SDR applications could not synthesize successfully using the ISE tool-flow. A workaround to the above problems is applying the four optimizations to the applications which result in the reduced number of FSM states (measured in thousands of FSM states) in Figure 6.10b. It is important to note that the optimized versions exhibit the same results under all throughput constraints, therefore each point in the graph serves to summarize all the design variants by using the range of **T1** – **T8**. For each application, *opt1* and *opt3* have the same number of states and also *opt2* and *opt4* have similar number of states. The fewest number of

FSM states are obtained when optimizations *opt2* and *opt4* are applied which reduce the non-optimized number of states by the factor of 1078455 while the optimizations *opt1* and *opt3* reduce the non-optimized number of states by the factor of 10892.



(a) Non-optimized



(b) Optimized

Figure 6.10: The total number of FSM states for each SDR application.

Code Lines

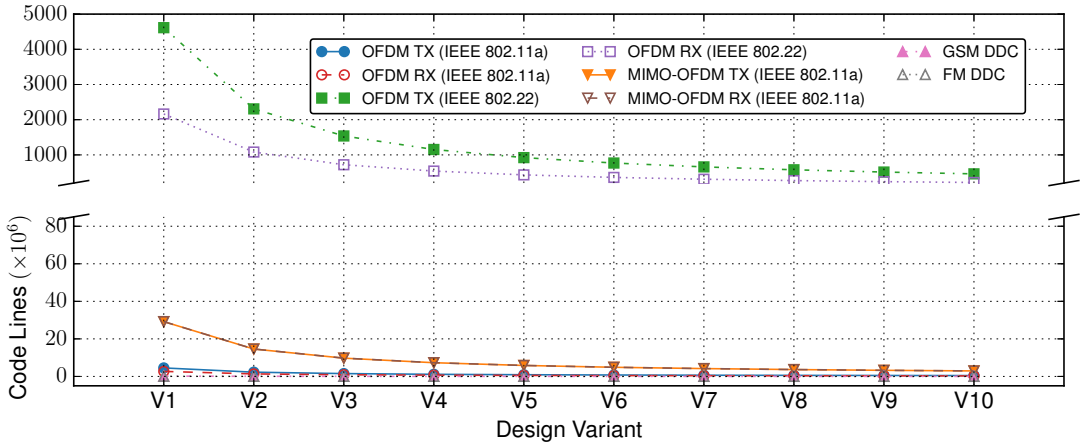
The simplicity and readability of most of the HLS-generated gateware is relatively low making it difficult to read and sometimes almost impossible to understand

in comparison to good hand-written code. The SdrLift code generator results in VHDL code that is concise, often using only a few lines of code as a result of applying the design optimizations. It is also ensured that SdrLift generated VHDL is well-structured and indented and that the interconnections are generally kept simple and concise where possible, all of which helps to make the code more readable.

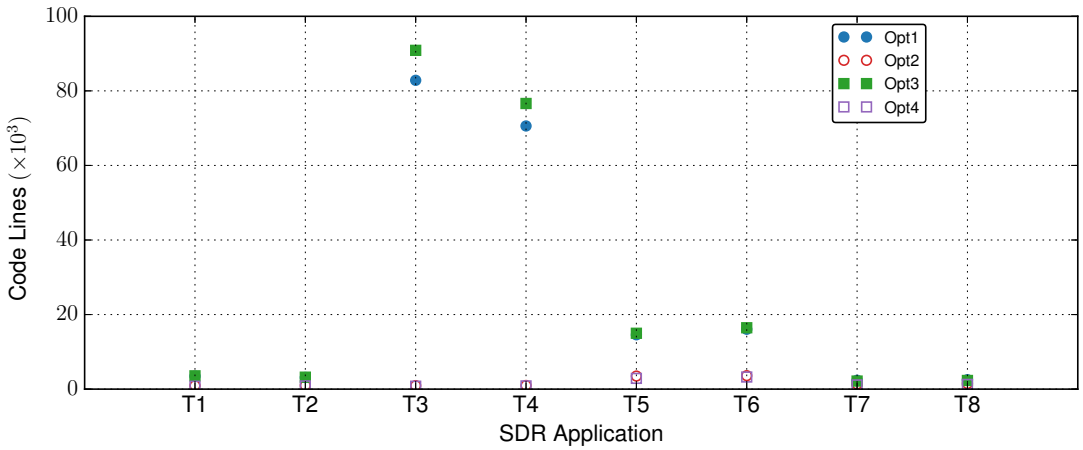
Figure 6.11 shows the total number of VHDL code lines for every SDR application when the application is not optimized and when the optimizations are applied. The results showing the code length of the non-optimized approach are depicted in Figure 6.11a. The resulting number of code lines is very huge (measured in millions of code lines) due to a large number of FSM states as discussed previously in Figure 6.10a. This number of code lines is drastically reduced (measured in thousands of code lines) when the optimizations are applied to the applications in Figure 6.11b. The results of the number of code lines are constant under different design variants hence the generic range (**T1** – **T8**) is used. The number of code lines varies with the application and the optimization type used. *Opt4* yields the fewest number of code lines by reducing the non-optimized number of code lines by a factor of 170188. This is followed by *opt2* which shortens the code length by the factor of 146207 and finally optimizations *opt1* and *opt3* which reduce the number of code lines by factors of 10252 and 9538 respectively. Generally, the number of code lines is directly proportional to the number of FSM states shown in Figure 6.10.

Compilation Duration

One of the benefits of using SdrLift design approach is to reduce developer time and improve designer productivity. SdrLift framework allows benchmarking of the execution time that elapses from the design description using SDF-AP model to the FPGA bitstream creation. This time combines the SDF-AP analysis, VHDL generation and the RTL synthesis using the ISE. The total time duration taken to perform the SDF-AP model analysis of each application given a throughput constraint is plotted in 6.12. The average time duration for each



(a) Non-optimized



(b) Optimized

Figure 6.11: The total number of VHDL code lines for each SDR application.

application is measure in ms. The OFDM-TX (IEEE 802.22) takes the longest time that averages 5967 ms, then OFDM-RX (IEEE 802.22) which takes 2148 ms, followed by MIMO OFDM-RX, OFDM-TX (IEEE 802.11a), MIMO OFDM-TX, OFDM-RX (IEEE 802.11a), FM-DDC, and GSM-DDC with respective 81, 31, 20, 11, 11, 8 ms.

In analysing the SDF-AP model, arrays are required to keep the model access patterns. For the applications with long access patterns, the on-heap memory of JVM does not handle the caching of gigabytes of data. A workaround to this

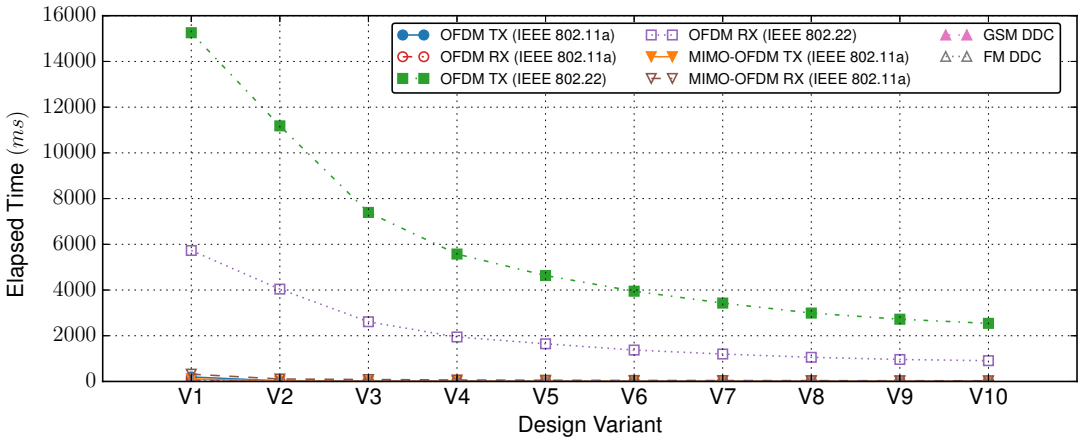
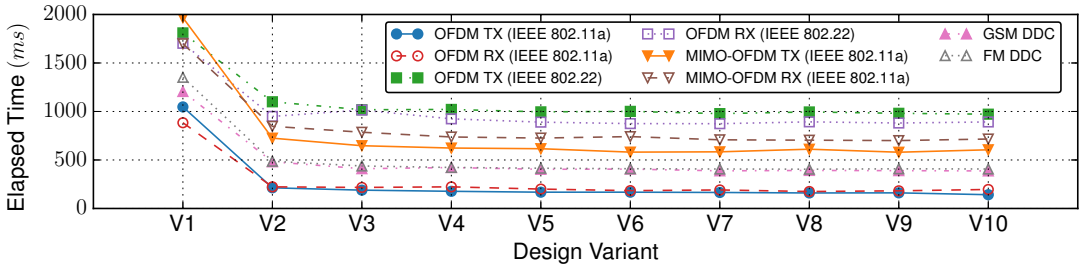


Figure 6.12: The total time elapsed for SDF-AP analysis of each application as per the throughput constraint.

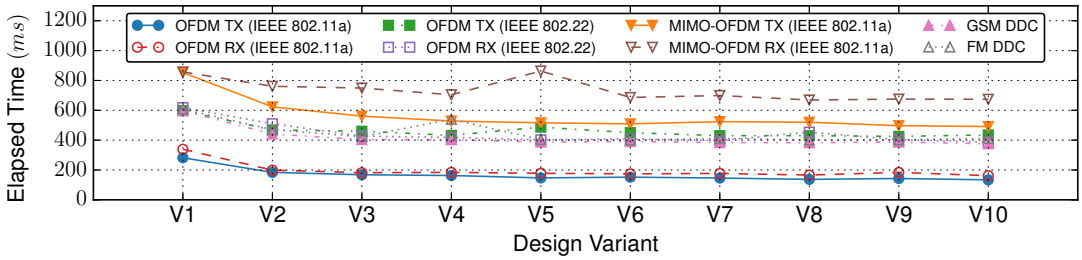
problem is using the off-heap memory which enables storing data outside the heap in the Operating System (OS) memory part. Because there is no JVM involved, the off-heap introduces the overhead of serializing and deserializing the long arrays to corresponding objects. There is an additional cost of dealing with native memory which does not exist in on-heap memory leads to the delayed analysis of SDF-AP model when the application access patterns are long.

Moreover, the time elapsed (in ms) it takes for SdrLift to generate the gateware for each application with respect to the throughput constraint is plotted in Figure 6.13. This figure contains the elapsed time graphs for optimizations *Opt1*, *Opt2*, *Opt3* and *Opt4* as illustrated in Figures 6.13a, 6.13b, 6.13c and 6.13d respectively. The average the times for each application when a relative optimization is used are summarized in Table 6.3. It can be observed that *Opt4* results in the shortest time taken for gateware generation, the followed by *Opt2*, *Opt3*, and *Opt4* respectively. As depicted in the results in Table 6.3, the bigger the application, the longer it takes to generate it's gateware in SdrLift.

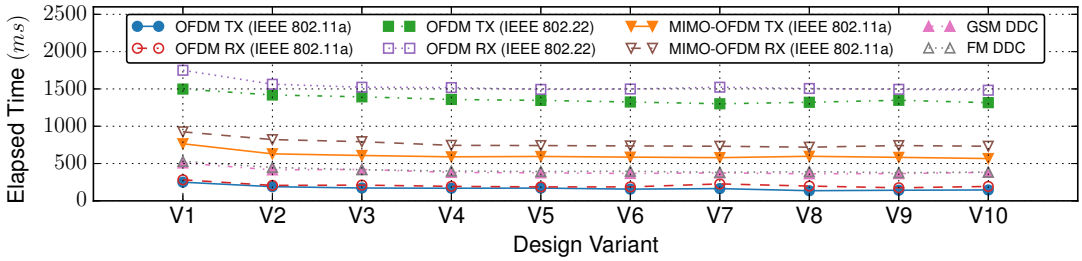
Figure 6.14 shows the elapsed time (measured in min) taken to synthesize each SDR application. For each application, the four optimizations techniques namely *Opt1*, *Opt2*, *Opt3* and *Opt4* as illustrated in Figures 6.14a, 6.14b, 6.14c and 6.14d are applied respectively. The average the times for each application when a



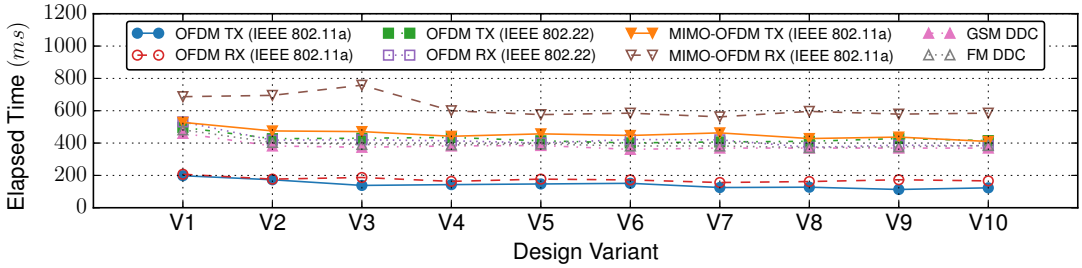
(a) Opt1



(b) Opt2



(c) Opt3



(d) Opt4

Figure 6.13: The time elapsed for application gateware generation using various optimizations as per throughput constraint.

Table 6.3: The average time elapsed (in ms) for application gateware generation using various optimizations.

Application	Opt1	Opt2	Opt3	Opt4
OFDM-TX (IEEE 802.11a)	259	165	169	143
OFDM-RX (IEEE 802.11a)	267	194	206	174
OFDM-TX (IEEE 802.22)	1087	461	1362	426
OFDM-RX (IEEE 802.22)	989	442	1535	417
MIMO OFDM-TX (IEEE 802.11a)	753	562	610	455
MIMO OFDM-RX (IEEE 802.11a)	836	733	768	622
GSM-DDC	488	415	395	382
FM-DDC	515	447	412	395

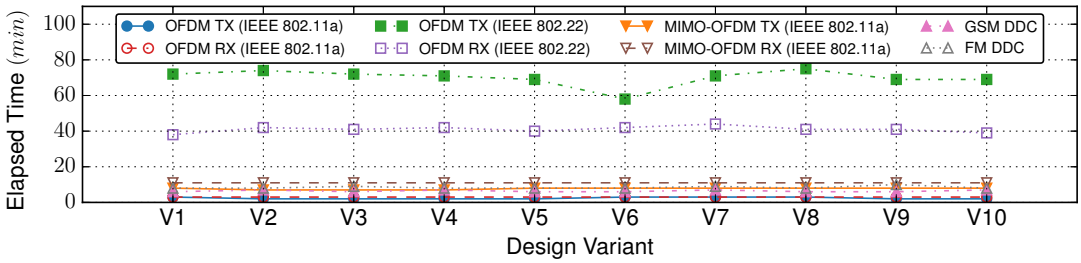
relative optimization is used are summarized in Table 6.4. These optimizations take various time delays to synthesis the gateware, with *Opt1* taking an average of 6 min, then followed by *Opt3*, *Opt2*, and *Opt4* which take 9, 13 and 15 min respectively. Furthermore, the biggest applications take longest to synthesize the gateware using Xilinx ISE.

Table 6.4: The average time elapsed (in min) for application synthesis using various optimizations.

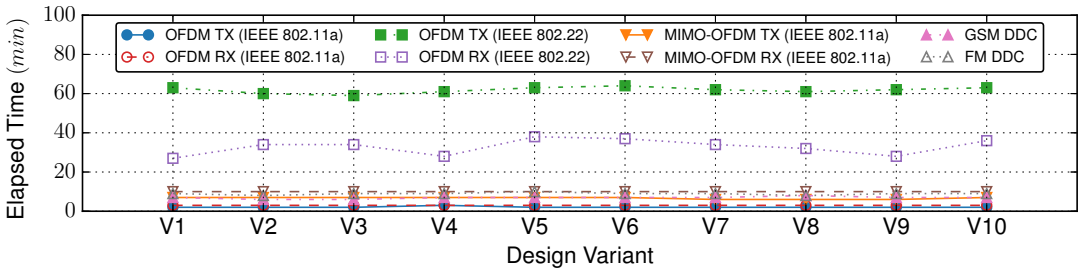
Application	Opt1	Opt2	Opt3	Opt4
OFDM-TX(IEEE 802.11a)	2	2	2	2
OFDM-RX(IEEE 802.11a)	3	3	2	2
OFDM-TX(IEEE 802.22)	70	61	27	11
OFDM-RX(IEEE 802.22)	41	32	20	11
MIMO OFDM-TX(IEEE 802.11a)	7	6	7	6
MIMO OFDM-RX(IEEE 802.11a)	11	10	7	6
GSM-DDC	6	6	4	5
FM-DDC	8	8	5	5

Area and Performance Benchmarks

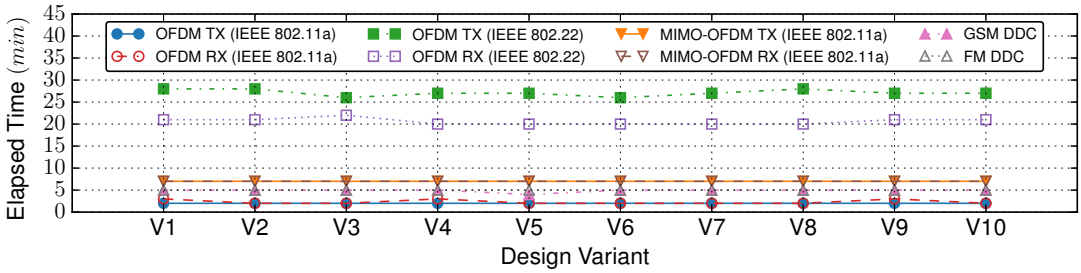
The benchmarks of the optimized versions of SDR applications are performed by targeting the Xilinx Spartan-6 xc6slx150t FPGA device. The non-optimized solutions are excluded as they are all not synthesizable on the FPGA. The FPGA



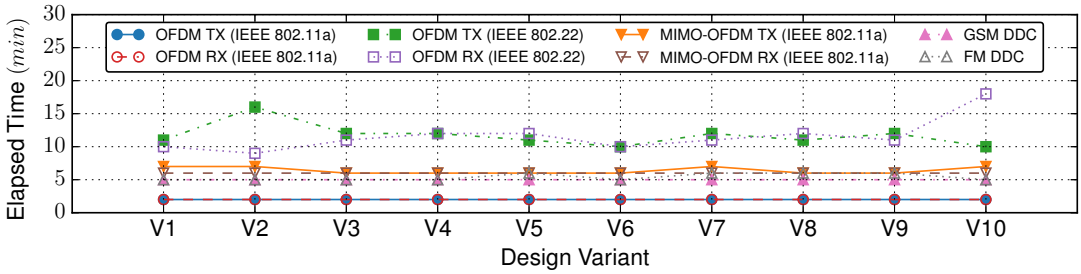
(a) Opt1



(b) Opt2



(c) Opt3

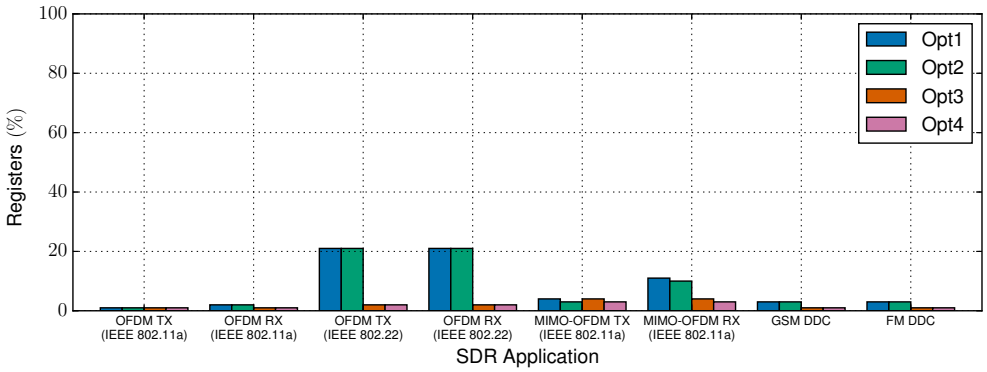


(d) Opt4

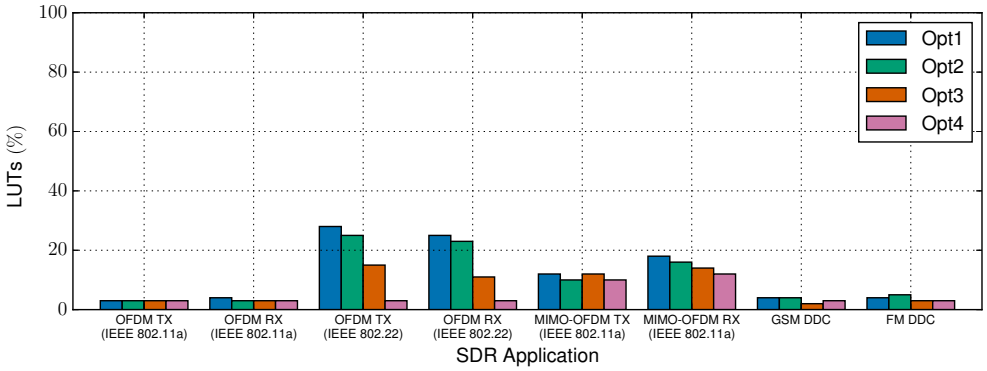
Figure 6.14: The time elapsed for application synthesis using various optimizations as per throughput constraint.

area utilization comprises the number of Registers, LUTs and occupied Slices. The total number of individual resources which are available on the FPGA are as follows, Registers = 184304, LUTs = 92152 and Slices = 23038. It needs to be noted that the percentage of used registers and LUTs in the results are in terms of those available in the occupied slices (i.e., not in terms of the total available registers and LUTs on the device). The total average resource usage for each application using optimizations *opt1*, *opt2*, *opt3* and *opt4* is shown in Figure 6.15 as a the percentage of available FPGA resources. The FPGA uses less registers as shown in Figure 6.15a, followed by the LUTs in Figure 6.15b and the slices in Figure 6.15c. Generally, *opt3* and *opt4* use less resources than *opt1* and *opt2*. Note that the results for non-synthesizable designs are not shown in which case the rectangular bars are skipped and this happens in large OFDM applications that are based on the IEEE 802.22.

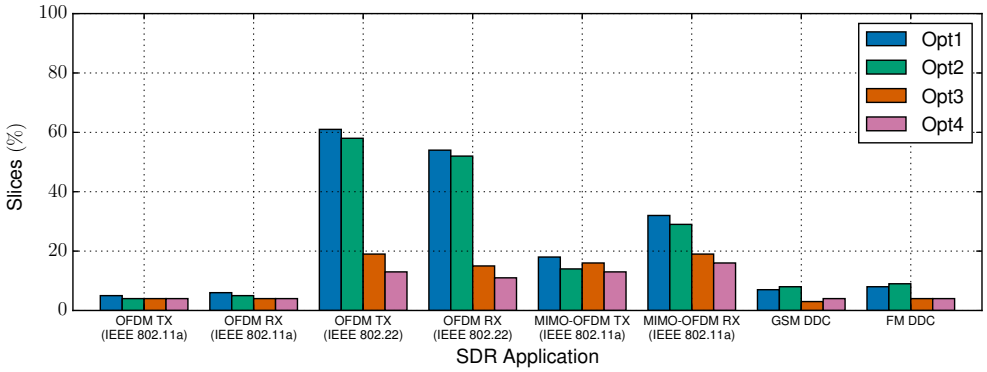
Moreover, the performance benchmark experiments are carried out using the metrics of power consumption and maximum frequency for each SDR application where the results are shown in Figure 6.16. Note that the power consumption values are acquired through running the power analysis and estimation tool that is available as part of the design process in the Xilinx ISE tool-flow. Also, the frequency values are obtained from the Xilinx ISE synthesis report. The results of the average power consumption by each of the SDR applications when the four optimizations are used are as shown in Figure 6.16a. In most cases, the applications with the largest area consume more power than the ones with the smallest area footprint on the FPGA. On average, *Opt1* consumes 0.0123%, 0.296%, and 0.314% less power than *opt2*, *opt3* and *opt4* respectively. Similarly, *Opt2* consumes 0.283%, and 0.302% less power than *opt3* and *opt4* respectively. Lastly, the *opt3* power consumption is 0.0182% more than that of the *opt4*. The final performance benchmark results include the maximum frequency that is achieved for each SDR application as shown in Figure 6.16b. For each application, the frequency results are the same in all design variants. The optimizations *opt1* and *opt2* have the same frequency results in each application, likewise *opt1* and *opt2* also result in similar maximum frequencies for each application. Both optimizations *opt1* and *opt2* have highest maximum achievable frequency by an average



(a) Registers



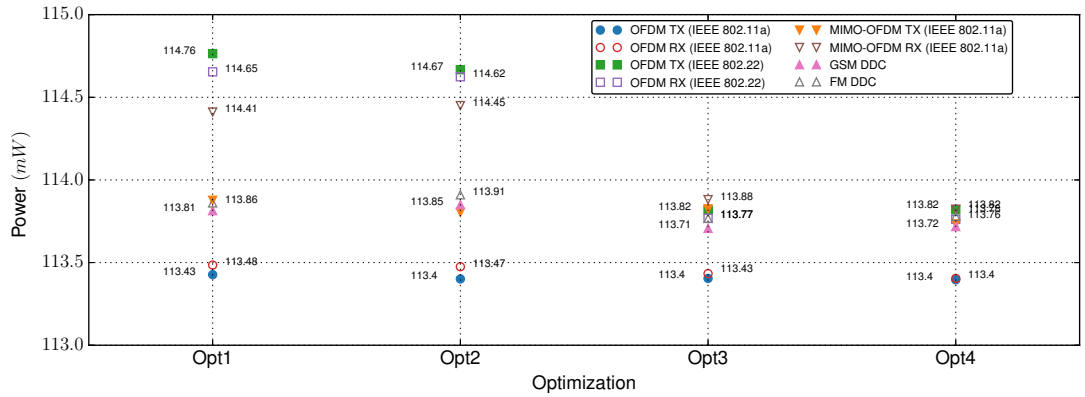
(b) LUTs



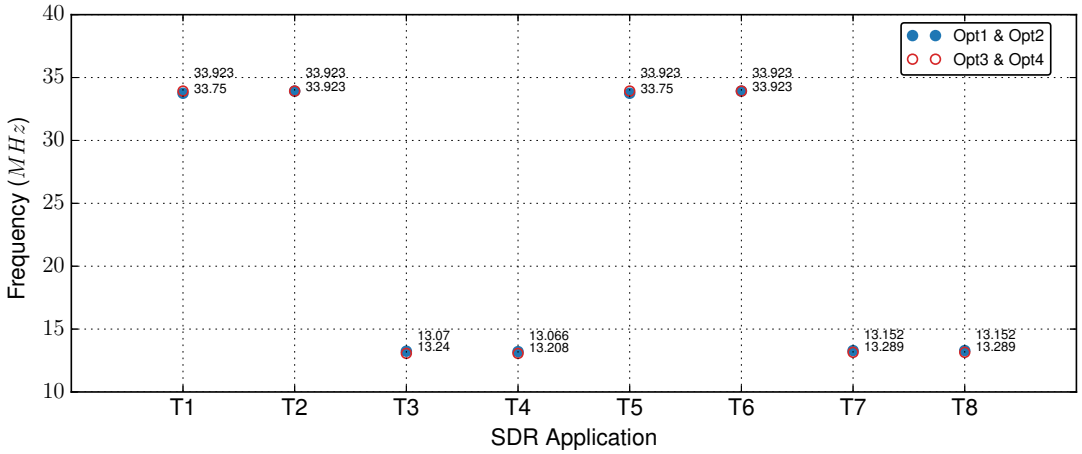
(c) Slices

Figure 6.15: The benchmark results of resource utilization for each SDR application.

factor of 1.00128 in comparison to *opt3* and *opt4*. This happens because *opt3* and *opt4* operate without FIFO channel instances where the allocated buffer size is 1. In general, the clock frequencies in Figure 6.16b are low and the work to increase them will be part of future improvements of SdrLift. Such improvements will entail buffer optimization techniques that will remove complex FIFO interface synchronizations that lead to long critical paths that limit the clock speed at hardware code generation phase.



(a) Average Power



(b) Frequency

Figure 6.16: The frequency and power consumption results for each SDR application.

6.2 FM Receiver

This section presents a comprehensive case study on the design and implementation of the FM receiver application using the SdrLift as shown in Figure 6.17. The throughput constraint of the FM receiver is set to 0.03846 SPC and the optimization is set to *opt4*. The are three core components that build the FM receiver namely RF front-end, the baseband processing on the FPGA, and the Desktop PC.

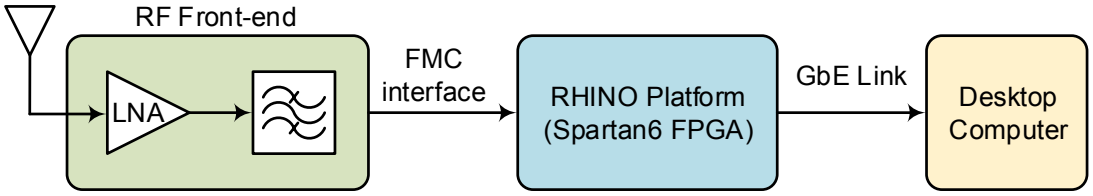


Figure 6.17: The FM Receiver block diagram.

6.2.1 RF Front-end

The RF front-end has the antenna that connects to a LNA, followed by a Mini-Circuits SXBP-100+ BPF [172] which passes frequencies in the range of 87 – 117 MHz. The BPF then links to the ADC of the Abaco Systems FMC150 ADC daughter-board [173]. The FMC150 is designed with TI's ADS62P49/ADS4249 dual-channel 14-bit 250 MSPS ADC and TI's DAC3283 dual channel 16-bit 800 MSPS DAC. The TI's CDCE72010 PLL is the clock distribution device that provides a clock to drive the DAC and ADC. The internal clock source can optionally be locked to on-board 100 MHz or external reference clock. The RHINO platform uses interfaces with the FMC150 through the Low-voltage Differential Signaling (LVDS) which is implemented in FPGA using the I/O IP core with architecture depicted in Figure 6.19. In this setting, only the ADC is used and it is configured to operate at the sampling rate of 7.68 MSPS. The front-end is able to isolate the the one channel of interest at 89 MHz station as depicted in Figure 6.18. The lack of anti-aliasing filter has led to other channels being downshifted closer to the DC as a result of aliasing effect, but they are further

away to keep the 89 MHz in isolation which is downshifted to 3.16 MHz after sub-sampling.

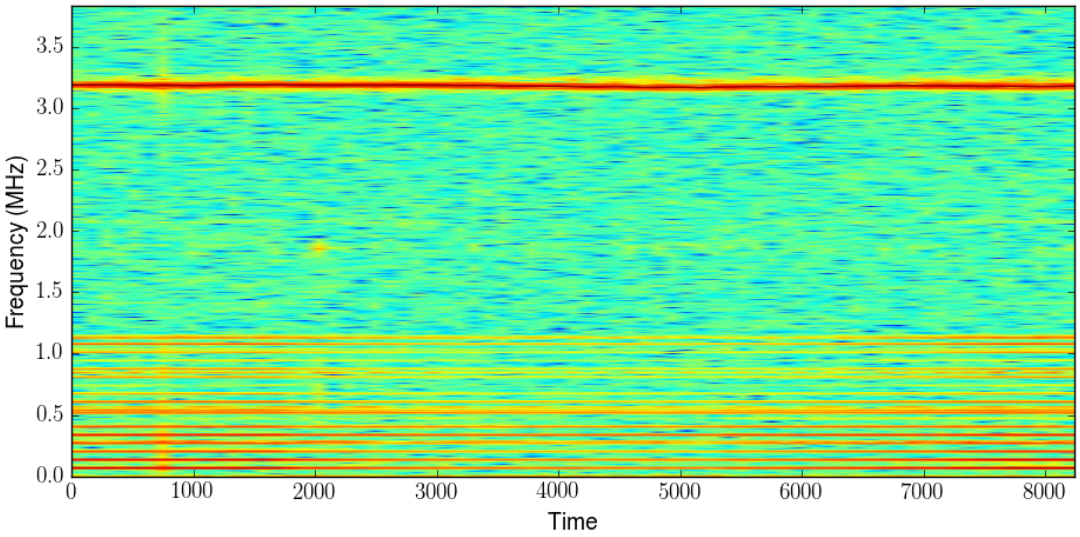


Figure 6.18: The received FM channels before digital down conversion.

6.2.2 Baseband Processing

The baseband infrastructure which is developed in SdrLift, implements an FM-DDC on a Spartan-6 xc6slx150t FPGA device of a RHINO platform [71]. This platform was developed in-house at the University of Cape Town for the intended use of radio and radar system prototyping and for use in embedded systems training. Figure 6.20 depicts the SDF-AP model example of a frequency modulation FM-DDC used for baseband processing of the case study. The SDF-AP model uses consumption and production patterns to accurately capture the timing of data production and consumption. A pattern is a binary word where each letter determines when the actor reads (i.e letter 1 denotes read, letter 0 denotes no data is read) or writes (i.e letter 1 denotes write, letter 0 denotes no data is written) token(s) at a specific clock cycle during firing.

This FM-DDC, as shown in Figure 6.20 whereby IP blocks ADC ,NCO ,CIC and GbE are predefined in VHDL and are therefore integrated into the design

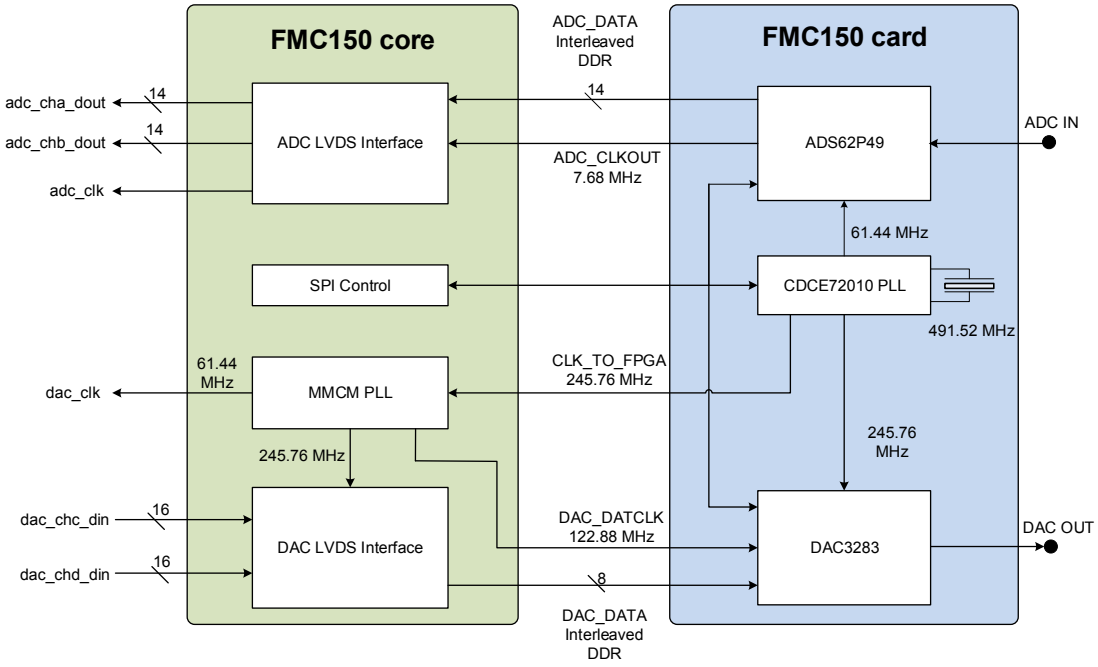


Figure 6.19: The I/O IP core interfacing the Abaco Systems FMC150 card.

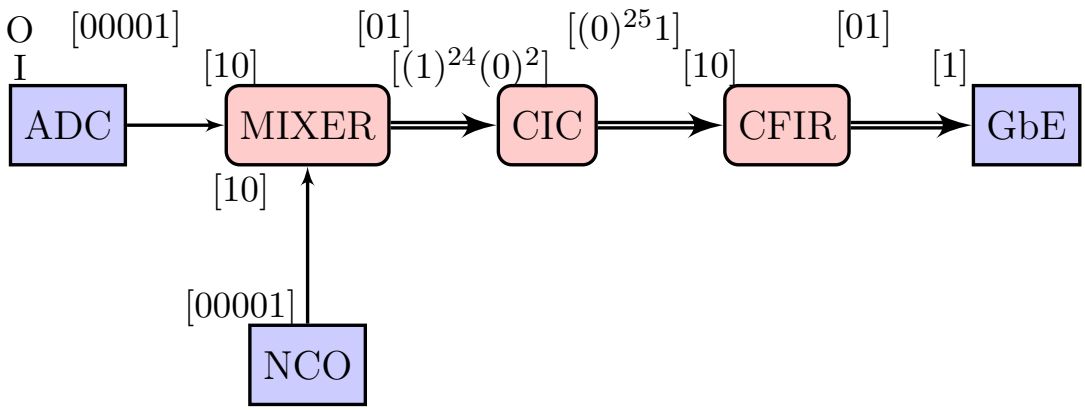


Figure 6.20: An SDF-AP model for FM Digital Down Converter.

as *Macro* (or black-box) kernels. The block diagram in Figure 6.20 starts with the ADC that digitizes a 20 MHz bandwidth FM signal into the 14-bit samples at the band-pass sampling rate of 7.68 MSPS. The ADC produces to its output FIFO buffer/channel one data sample every clock cycle as denoted by the pattern [1]. Similarly, the NCO signal uses pattern [1] to produce the LO (cosine and sine) signals that help to select the desired 200 kHz FM channel. The selection

process is carried out by tuning to 89 MHz station (i.e. SABC 5FM Radio) by setting the frequency of NCO to 3.16 MHz which is the aliased frequency of the radio station.

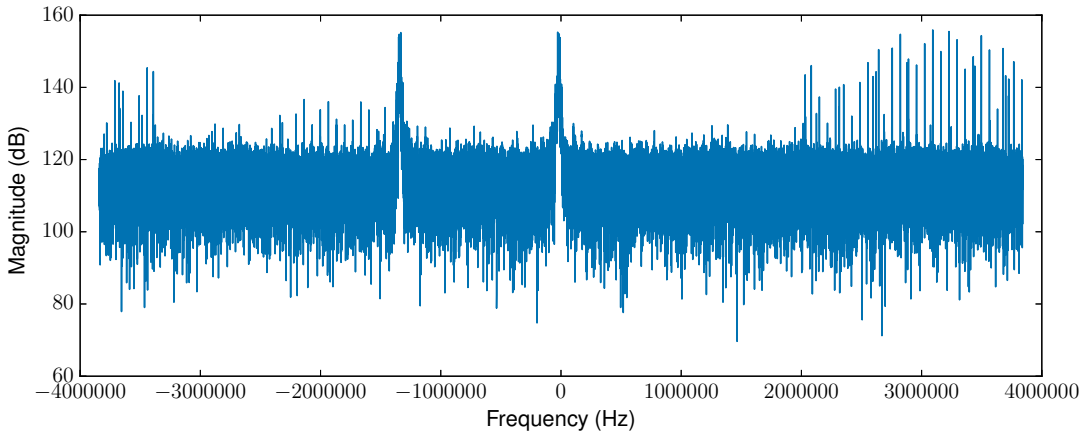


Figure 6.21: The Mixer signal spectrum.

Moreover, the baseband signal is generated by mixing, or multiplying, the received FM RF signal with the NCO signal as shown in Figure 6.21. This mixer results in a 200 kHz channel being shifted to a DC and its replica is moving to 1.58 MHz. Note that mixing is performed by a Mixer which receives data samples with pattern [10] and produces the results with pattern [01].

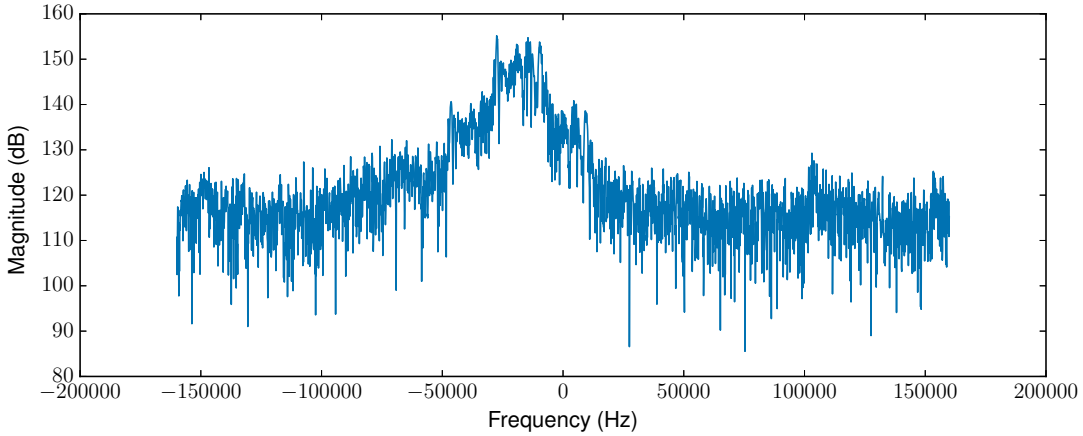


Figure 6.22: The CIC signal spectrum.

The mixer is followed by the CIC filter which decimates the 7.68 MSPS sample

rate to 320 KSPS by a factor of 24. The CIC accepts 24 samples every cycle using pattern $[1^{24}(0)^2]$ and produces one sample every 128 cycle using pattern $[0^{25}1]$ and the CIC filter results are shown in Figure 6.22. The disadvantage of a CIC filter is that its pass-band is not flat, which is the undesirable behaviour in the design. This problem is alleviated by a CFIR [174] which receives data samples with pattern [10], produces the output samples with pattern [01], and the corresponding CFIR results are shown in Figure 6.23. In both cases of CIC and CFIR results, the 89 FM station is not perfectly centered at DC. This is caused by the hardware's local oscillator (NCO) offset, however, this is negligible for this experiment. Lastly, the baseband data samples are received by the GbE with pattern [1] before being transmitted to the Desktop PC in UDP frames of 128 bytes.

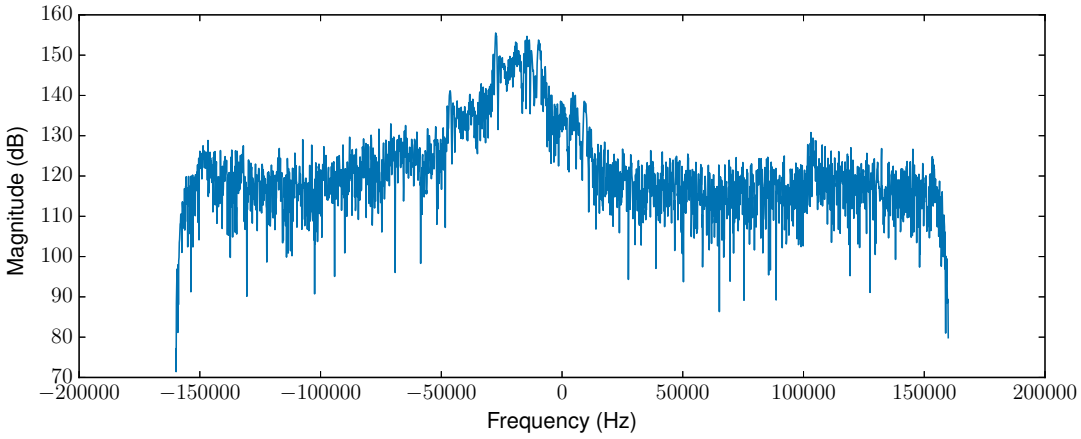


Figure 6.23: The CFIR signal spectrum

6.2.3 Desktop PC

The results of baseband processing are then streamed to a Desktop PC via a GbE where FM demodulation is performed. On the FPGA, the GbE IP core depicted in Figure 6.24 delivers UDP messages between the RHINO FPGA and the PC. Integral to the GbE core is the Open-Cores tri-mode MAC [175] plays a role of transferring data over a shared physical channel. The two user interfaces on the MAC simplify interfacing with the PHY (Marvell 88E111 PHY chip).

The PHY encodes and decodes packet data transferred between a MAC and a PHY using Gigabit Media-Independent Interface (GMII). Furthermore, the Management Data Input/Output (MDIO) bus transfers configuration data to the PHY.

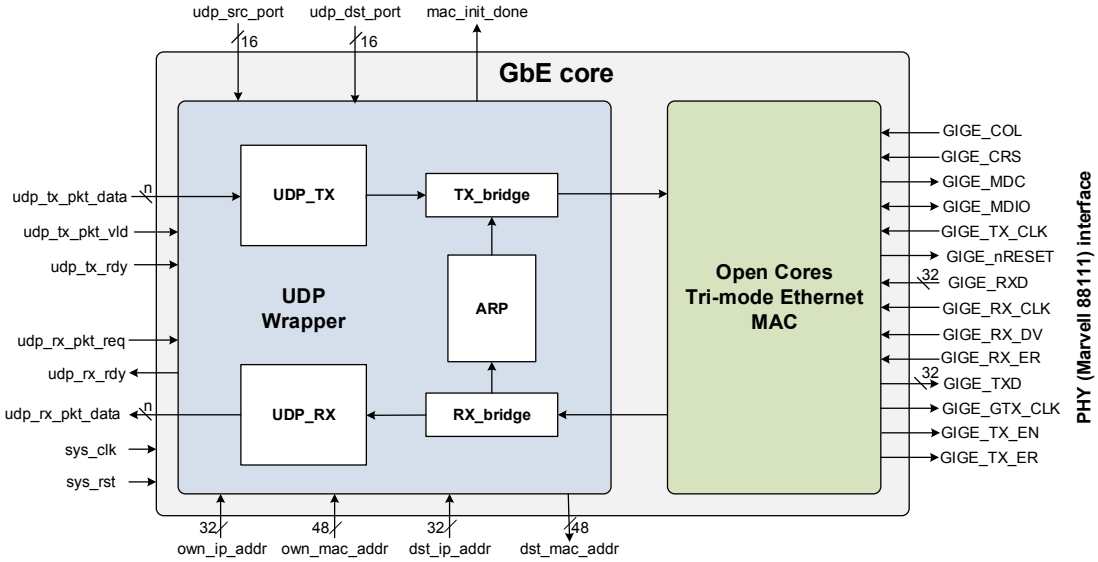


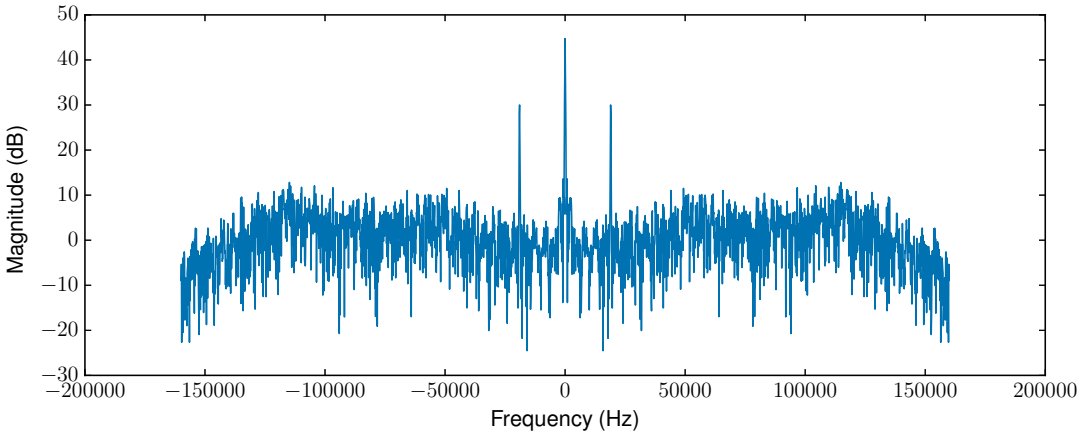
Figure 6.24: The I/O IP core interfacing the RHINO PHY to the FPGA.

Once the CFIR signal is received on the the PC, it is demodulated in Matlab on the Computer and the resulting spectrum is shown in Figure 6.25. The mono audio channel (left+right), 19 kHz stereo pilot, stereo audio channel (left-right), and the Radio Broadcast Data System (RBDS) signal can clearly be seen on the demodulated FM signal.

6.3 Chapter Summary

This chapter has presented the experimental evaluation of SdrLift. The applicability of SdrLift to SDR application prototyping by developing eight representative applications as well a comprehensive digital FM receiver for which SdrLift generated an executable VHDL implementations. The results showed that high-performance can be achieved through the custom design of new IP cores and the

Figure 6.25: The resulting spectrum after FM demodulating the CFIR signal.



integration of pre-existing ones; this confirms that our technique shows promise as a domain-specific prototyping tool for SDR.

Chapter 7

Conclusions and Further Work

This chapter presents the conclusions and future work for SdrLift presented in this thesis.

7.1 Conclusions

This thesis presents SdrLift, an intermediate-level compiler framework for automating the generation of hardware accelerators in the application domain of SDR. It achieves this by incorporating an entry-point point language that captures the structural behavior of the SDR applications at the high-level of design abstraction using functional language constructs and design patterns that raise the expressive power as presented in Chapter 4. The input language alleviates the *complexity of designs* and *lack of design constraints specification* limitations that are outlined in Sections 1.2.1 and 1.2.2 respectively. The benefits of the input language are two-fold; firstly it can be used directly by SDR experts to describe SDR applications that target are executed on the FPGA architectures, hence increasing the productivity of design. Secondly, it can be integrated into a higher-level language or a domain-specific language for implementation of an intermediate-representation compiling stage, thereby eliminating the development strain for new HLS or DSL compilers.

Integral to SdrLift is the directed flow graph and dataflow model which are used to implement the different IR levels of compilation in Chapter 4. First, the directed flow graph is used to represent fine-level components and coarse-level HW Blocks. The coarse-level HW Blocks are either synthesized by SdrLift from the user-specified description or integrated into SdrLift compilation flow from existing IP core libraries. Secondly, the static dataflow model that supports data access patterns (i.e. SDF-AP model) is employed to create complete compositional applications by stitching together the synthesized HW Blocks and pre-defined or hand-coded IP cores as presented in Chapter 5. Therefore Chapter 5 addresses the last two limitations which are *lack of support for IP integration* and *lack of correctness verification* as outlined in Sections 1.2.3 and 1.2.4 respectively.

The applicability of SdrLift to SDR application prototyping is demonstrated by developing eight representative applications in which VHDL code is generated for each application as presented in Chapter 6. Furthermore, a more comprehensive case study of the digital FM receiver is developed for which SdrLift generated an executable VHDL implementation. The results show that high-performance can be achieved through the custom design of new IP cores and the integration of pre-existing ones; this confirms that SdrLift shows promise as a high-level prototyping tool for SDR. SdrLift is available in an open-source license and is be publicly accessible for free at `https://github.com/lekhobola/SdrLift`.

The contribution of the work presented in this thesis are evaluated through the review and examination of the questions asked to validate the hypothesis that provides the guiding principle for this project.

- *What are the traits of the intermediate-language to describe the FPGA-based SDR applications at the high-level of design abstraction?* By focusing on the problem domain of SDR, SdrLift has created an abstraction to the low-level intricacies often experienced by the SDR designers who develop SDR applications that run on the FPGAs. This abstraction has been made possible through the implementation of the SdrLift language that captures the structural behavior of the SDR application using functional constructs, design patterns, and topological patterns. It is demonstrated in Chapter 6

that SdrLift language is capable of describing new SDR applications using SdrLift-defined HW Blocks and existing IP cores developed by hand in VHDL and using some that are obtained from the IP libraries. Not only does SdrLift language describe the SDR specifications concisely which high expressivity, but it also allows the resultant system to be constrained to the desired throughput at the high-level of design abstraction. SdrLift language will play an instrumental role in helping the SDR designers accelerate the development process and will be adopted HLS or DSL as the IR compiler step.

- *What is an appropriate design approach for developing hardware cores that will result in SDR applications that conform to the specified throughput constraint?* Through the exploitation of the template-based design together with the DFG, SdrLift compiler computes the model properties of the new synthesizable HW Blocks as presented in Chapter 6. The benefit of this design approach is that it decouples the high-level specifications from microarchitecture The automated computation of the model properties helps the SdrLift compiler to build an SDF-AP model that represents a complete SDR application to meet the throughput constraint specified by SDR designer using SdrLift language. As for IP cores that are integrated from the IP libraries, their model properties are obtained through timing and behavioral information provided in the datasheets and manual, and some cases, the low-level simulations are performed to acquire the properties.

- *What dataflow model is needed to effectively analyse and compose the developed HW Blocks for implementation of a system that conforms to a throughput constraint?* By adopting the SDF-AP model, the SdrLift compiler stitches together the HW Blocks through the process of composition validation, performing scheduling of the operational blocks, and computing the optimal buffer requirements as per the throughput constraint specified in the SdrLift program. This SdrLift methodology of system generation bridges the gap between the microarchitecture and the low-level model of the hardware. In Chapter 6, the case study was developed whereby the VHDL code for eight representative applications was generated, the results

showed that hardware designs which conform to the high-level specifications are guaranteed by the SdrLift compilation flow.

- *Can the compiler flow be exploited to generate the optimal hardware design?* The low-level implementation of the SDF-AP model is largely realized in FSMs on the FPGA. These FSMs can lead to very poor results if not implemented carefully on the hardware as demonstrated in Chapter 6. In order to obtain quality results, four optimizations have been implemented to provide a solution space that enables the user to choose the best solution that meets the desired performance under throughput and target hardware resource constraints. The applicability of the SdrLift approach was demonstrated through the practical implementation of a selection of eight representative SDR case studies. The results showed that high-performance constraints (i.e. latency, buffer size, maximum frequency, and power) and optimal area utilization can be achieved and can continue to be improved to provide a best-effort throughput performance, within reasonable limits of the target hardware concerned.

7.2 Recommendations for Further Work

In this thesis, various hardware synthesis methods have been analysed for prototyping of SDR applications targeted for deployment on FPGA platform. The further investigation aimed at improving the SdrLift language constructs and the SdrLift compiler capability remains a topic for future research. The future refinements and improvements are therefore described below.

- Thus far the SdrLift language only describes the structural behavior of the underlying SDR system using the design patterns and topological patterns. In order to move closer to the Turing-complete code generation mechanism in SdrLift language, additional constructs such as loops, case statements and if structures are required for the algorithmic description of the system behavior. This will be added as another feature for future improvements.

- The algorithmic specifications in a high-level SDR program imply that high-level code optimizations will be required in the future refinements of SdrLift. Such optimizations include Code hoisting, Loop Fission and Fusion, Loop Unrolling, Loop Tiling, Loop Pipelining, Bitwidth optimization, CSE, DCE and code motion while also applying domain-specific optimizations.

- The continuing work seeks to enrich the SdrLift compiler with more signal processing templates and to expand the library of IP blocks.

- In order to solve a wider range of SDR domain problems, and to make SdrLift-created systems adaptable to a wider range of waveforms, the SDF-AP model needs to be extended further to alleviate its limitations many of which have been identified in [148]. Such improvements will also allow dynamicity and reconfigurability of the SDR systems at run-time.

- SDR applications are nowadays being more often deployed on heterogeneous computing platforms in order to leverage the architectural benefits which such platforms offer. For instance, these SDR platforms often combine the flexibility of a GPP with the performance and parallelism of an FPGA to perform control and compute-intensive functions respectively. However, the low-level development difficulties associated with these heterogeneous platforms reduce productivity, even when the designer is experienced in both hardware and software design. Moreover, productivity may be compromised for SDR experts with little or no low-level hardware design skills. These low-level difficulties include non-standard interfacing methods, PC-FPGA communication and synchronization challenges, complicated timing constraints and processing modules that need to be customized through time-consuming design tweaks. The future enhancements to SdrLift will facilitate development by automating automatically generates PC-based software that performs configuration and setup of the FPGA modules and its peripherals, as opposed to having to incorporate these cumbersome routines into the FPGA that can consume valuable resources. This PC software will also allow SDR data streaming and analysis in both the

upstream and downstream operating modes. This future complete SDR system is depicted in Figure 7.1 in which its architecture is composed of programmable logic and the processing system. The SdrLift will, therefore, support heterogeneous code generation in will VHDL/Verilog will be generated to run on the programmable logic while also generating C++ code that will be deployed for execution on the processing system.

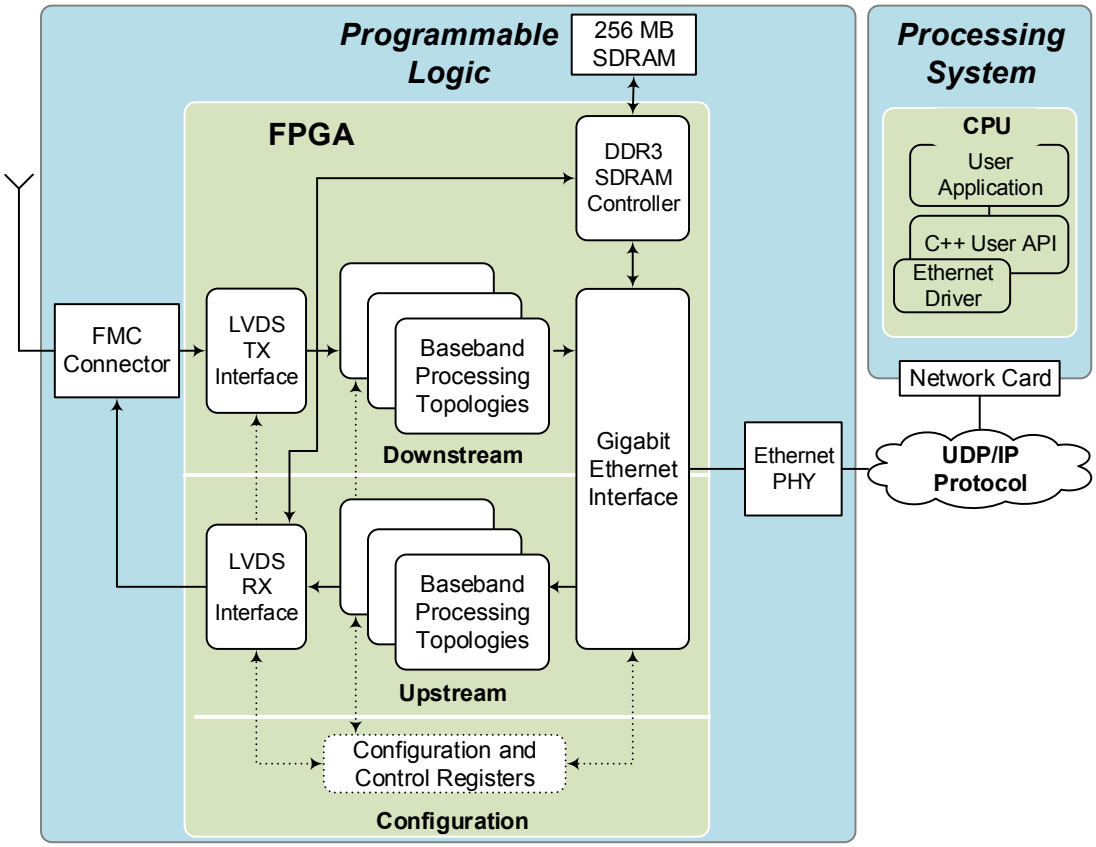


Figure 7.1: Proposed future SDR architecture to be created in SdrLift.

- Introduce a Design Space Exploration (DSE) in SdrLift that will make the SDR application much easier. The DSE will be performed in both the early and late stages of the system compilation and it will consist of various test algorithms on multiple architectures and making appropriate resource usage and performance choices between the gateware generation or programmable logic and the software generation for the processing system.

- Lastly, the future work will exploit the DFG and SDF-AP models in order to estimate the FPGA resource usage for designs expressed in SdrLift. The run-time area estimates will be compared with reports provided by the place-and-route stage of FPGA vendor tools in order to make SdrLift a tool best suited for design DSE.

Appendix A

SdrLift Sample Source Code

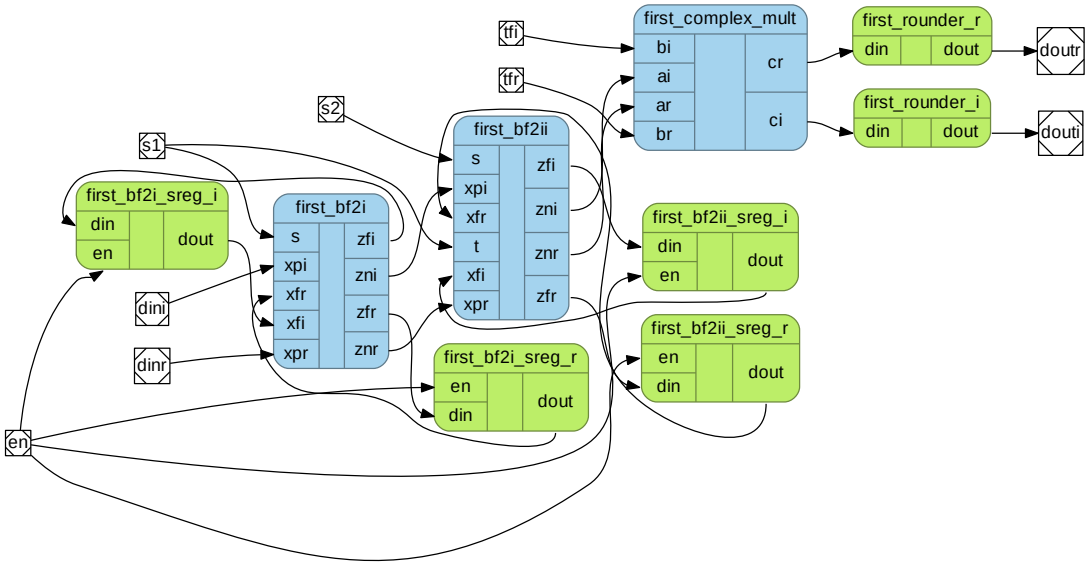


Figure A.1: A DFG of a full stage with a complex multiplier.

```
case class Stage(inst: String, data_width: Int, b1_depth: Int, b2_depth: Int) extends Component {
  //inputs
  val (en, s1, s2, tfr, tfi, dinr, dini) = (Streamer("en", 1), Streamer("s1", 1), Streamer("s2", 1), Streamer("tfr", 16), Streamer("tfi", 16), Streamer("dinr", data_width), Streamer("dini", data_width))
  // outputs
  val (doutr, douti) = (Streamer("doutr", data_width + 2), Streamer("douti", data_width + 2))
  override val iopaths = List((dinr, doutr), (dini, douti))
  val cmb = Combinational {
    // BF2I
    val bf2i = BF2I(inst + "_bf2i", data_width)
    val bf2i_comm = bf2i inLinks(s1 ~> (bf2i, bf2i.s), dinr ~> (bf2i, bf2i.xpr), dini ~> (bf2i, bf2i.xpi))
    // BF2I shift registers
    val bf2i_sreg_r = Delay(inst + "_bf2i_sreg_r", data_width + 1, b1_depth)
    val bf2i_sreg_r_comm = bf2i_sreg_r inLinks(en ~> (bf2i_sreg_r.en), (bf2i, bf2i.zfr) ~> (bf2i_sreg_r, bf2i_sreg_r.din)) outLinks ((bf2i_sreg_r, bf2i_sreg_r.dout) ~> (bf2i, bf2i.xfr))
    val bf2i_sreg_i = Delay(inst + "_bf2i_sreg_i", data_width + 1, b1_depth)
    val bf2i_sreg_i_comm = bf2i_sreg_i inLinks(en ~> (bf2i_sreg_i.en), (bf2i, bf2i.zfi) ~> (bf2i_sreg_i, bf2i_sreg_i.din)) outLinks ((bf2i_sreg_i, bf2i_sreg_i.dout) ~> (bf2i, bf2i.xfi))
    // BF2II
    val bf2ii = BF2II(inst + "_bf2ii", data_width + 1)
    val bf2ii_comm = bf2ii inLinks(s2 ~> (bf2ii, bf2ii.s), s1 ~> (bf2ii, bf2ii.t), (bf2i, bf2i.znr) ~> (bf2ii, bf2ii.xpr), (bf2i, bf2i.zni) ~> (bf2ii, bf2ii.xpi))
    // BF2II shift registers
    val bf2ii_sreg_r = Delay(inst + "_bf2ii_sreg_r", data_width + 2, b2_depth)
    val bf2ii_sreg_r_comm = bf2ii_sreg_r inLinks(en ~> (bf2ii_sreg_r.en), (bf2ii, bf2ii.zfr) ~> (bf2ii_sreg_r, bf2ii_sreg_r.din)) outLinks ((bf2ii_sreg_r, bf2ii_sreg_r.dout) ~> (bf2ii, bf2ii.xfr))
    val bf2ii_sreg_i = Delay(inst + "_bf2ii_sreg_i", data_width + 2, b2_depth)
    val bf2ii_sreg_i_comm = bf2ii_sreg_i inLinks(en ~> (bf2ii_sreg_i.en), (bf2ii, bf2ii.zfi) ~> (bf2ii_sreg_i, bf2ii_sreg_i.din)) outLinks ((bf2ii_sreg_i, bf2ii_sreg_i.dout) ~> (bf2ii, bf2ii.xfi))
    // Multiplier
    val cmult = ComplexMult(inst, data_width + 2, 16, data_width + 18)
    val cmult_comm = cmult inLinks((bf2ii, bf2ii.znr) ~> (cmult, cmult.ar), (bf2ii, bf2ii.zni) ~> (cmult, cmult.ai), tfr ~> (cmult, cmult.br), tfi ~> (cmult, cmult.bi)) //
    outLinks((cmult, cmult.cr) ~> doutr, (cmult, cmult.ci) ~> douti)
    // Rounder
    val rounder_r = Rounder(inst + "_cmultr", data_width + 16, data_width + 2)
    val rounder_r_comm = rounder_r inLinks ((cmult, cmult.cr) ~> (rounder_r, rounder_r.din)) outLinks ((rounder_r, rounder_r.dout) ~> doutr)
    val rounder_i = Rounder(inst + "_cmulti", data_width + 16, data_width + 2)
    val rounder_i_comm = rounder_i inLinks ((cmult, cmult.ci) ~> (rounder_i, rounder_i.din)) outLinks ((rounder_i, rounder_i.dout) ~> douti)
    :=(bf2i_comm, bf2i_sreg_r_comm, bf2i_sreg_i_comm, bf2ii_comm, bf2ii_sreg_r_comm, bf2ii_sreg_i_comm, cmult_comm, rounder_r_comm, rounder_i_comm)
  }
  override val name: String = inst + "_stage"
}
```

Listing 18: SdrLift code for a full stage with a complex multiplier.

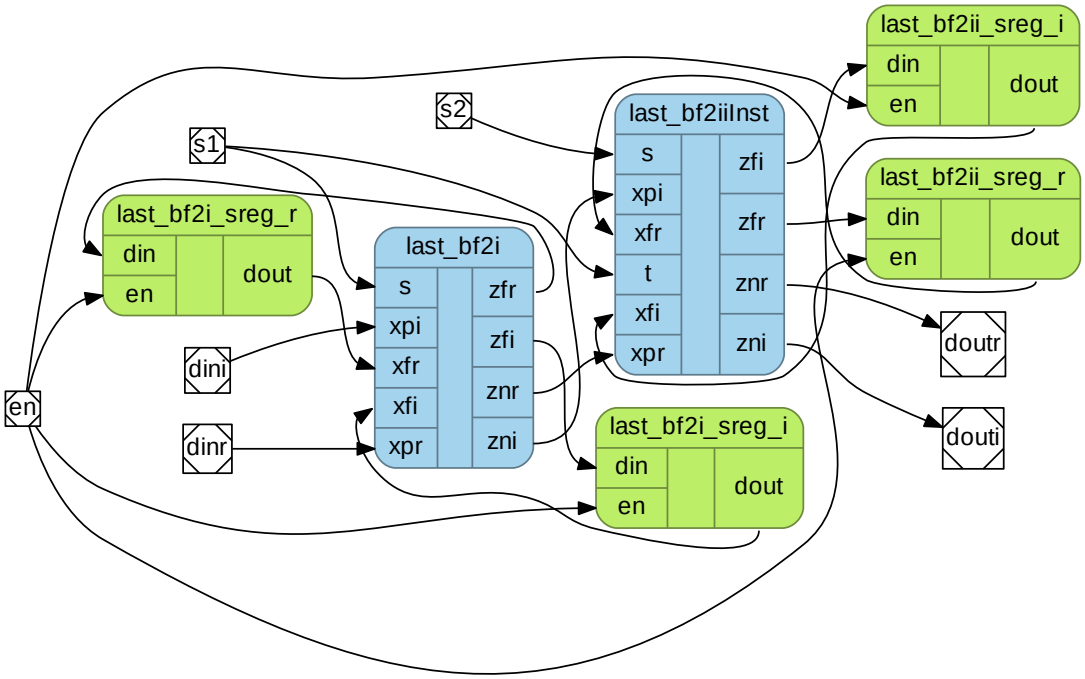


Figure A.2: A DFG of the last full stage with no a complex multiplier.

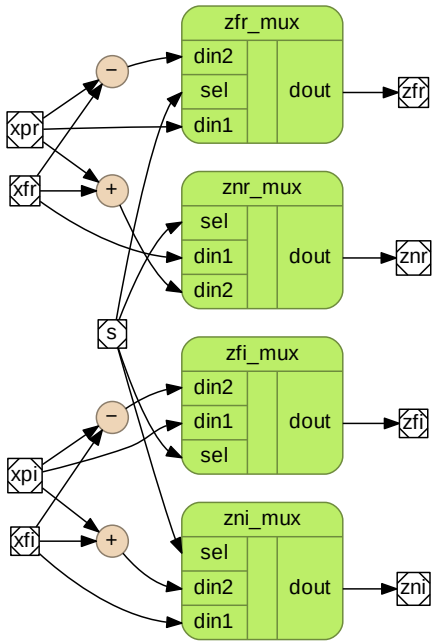


Figure A.3: Butterfly I DFG.

```
case class LastEvenStage(inst: String, data_width: Int, b1_depth: Int, b2_depth: Int) extends Component {

  //inputs
  val (en, s1, s2, dinr, dini) = (Streamer("en", 1), Streamer("s1", 1), Streamer("s2", 1), Streamer("dinr", data_width), Streamer("dini", data_width))
  // outputs
  val (doutr, douti) = (Streamer("doutr", data_width + 2), Streamer("douti", data_width + 2))

  override val iopaths = List((dinr, doutr), (dini, douti))

  val cmb = Combinational {
    // BF2I
    val bf2i = BF2I(inst + "_bf2i", data_width)
    val bf2i_comm = bf2i inLinks(s1 ~> (bf2i, bf2i.s), dinr ~> (bf2i, bf2i.xpr), dini ~> (bf2i, bf2i.xpi))

    // BF2I shift registers
    val bf2i_sreg_r = Delay(inst + "_bf2i_sreg_r", data_width + 1, b1_depth)
    val bf2i_sreg_r_comm = bf2i_sreg_r inLinks(en ~> (bf2i_sreg_r.en), (bf2i, bf2i.zfr) ~> (bf2i_sreg_r, bf2i_sreg_r.din)) outLinks ((bf2i_sreg_r, bf2i_sreg_r.dout) ~> (bf2i, bf2i.xfr))
    val bf2i_sreg_i = Delay(inst + "_bf2i_sreg_i", data_width + 1, b1_depth)
    val bf2i_sreg_i_comm = bf2i_sreg_i inLinks(en ~> (bf2i_sreg_i.en), (bf2i, bf2i.zfi) ~> (bf2i_sreg_i, bf2i_sreg_i.din)) outLinks ((bf2i_sreg_i, bf2i_sreg_i.dout) ~> (bf2i, bf2i.xfi))

    // BF2II
    val bf2ii = BF2II(inst + "_bf2iiInst", data_width + 1)
    val bf2ii_comm = bf2ii inLinks(s2 ~> (bf2ii, bf2ii.s), s1 ~> (bf2ii, bf2ii.t), (bf2i, bf2i.znr) ~> (bf2ii, bf2ii.xpr), (bf2i, bf2i.zni) ~> (bf2ii, bf2ii.xpi)) outLinks ((bf2ii, bf2ii.znr) ~> doutr, (bf2ii, bf2ii.zni) ~> douti)

    // BF2II shift registers
    val bf2ii_sreg_r = Delay(inst + "_bf2ii_sreg_r", data_width + 2, b2_depth)
    val bf2ii_sreg_r_comm = bf2ii_sreg_r inLinks(en ~> (bf2ii_sreg_r.en), (bf2ii, bf2ii.zfr) ~> (bf2ii_sreg_r, bf2ii_sreg_r.din)) outLinks ((bf2ii_sreg_r, bf2ii_sreg_r.dout) ~> (bf2ii, bf2ii.xfr))
    val bf2ii_sreg_i = Delay(inst + "_bf2ii_sreg_i", data_width + 2, b2_depth)
    val bf2ii_sreg_i_comm = bf2ii_sreg_i inLinks(en ~> (bf2ii_sreg_i.en), (bf2ii, bf2ii.zfi) ~> (bf2ii_sreg_i, bf2ii_sreg_i.din)) outLinks ((bf2ii_sreg_i, bf2ii_sreg_i.dout) ~> (bf2ii, bf2ii.xfi))

    :=(bf2i_comm, bf2i_sreg_r_comm, bf2i_sreg_i_comm, bf2ii_comm, bf2ii_sreg_r_comm, bf2ii_sreg_i_comm)
  }
  override val name: String = inst + "_evenstage"
}
```

Listing 19: SdrLift code for the last full stage with no complex multiplier.

```scala
case class Counter(inst: String, w: Int) extends Component {
  override val name = "counter"
  val en = Streamer("en", 1, PortTypeEnum.EN)
  val dout = Streamer("dout", w, PortTypeEnum.DOUT)

  _params = Map("width" -> w, "inst" -> inst)
  _ports = List(en, dout)
}
```

Listing 20: The counter that implements the controller.

```scala
case class Rom(inst: String, data_width: Int, vector: Seq[Int]) extends Component {

  override val name = "rom"

  val addr_width = Math.ceil(Math.log10(vector.length) / Math.log10(2)).toInt
  val addr = Streamer("addr", addr_width, PortTypeEnum.DIN)
  val dout = Streamer("dout", data_width, PortTypeEnum.DOUT)

  _params = Map("data_width" -> data_width, "addr_width" -> addr_width, "vector" -> vector,
    ↪   "inst" -> inst)
  _ports = List(addr, dout)
}
```

Listing 21: The ROM that stores twiddle factors.

```
case class BF2I(inst: String, w: Int) extends Component {
  //inputs
  val (s, xpr, xpi, xfr, xfi) = (Streamer("s", 1), Streamer("xpr", w), Streamer("xpi", w),
   ↪   Streamer("xfr", w + 1), Streamer("xfi", w + 1))
  // outputs
  val (znr, zni, zfr, zfi) = (Streamer("znr", w + 1), Streamer("zni", w + 1), Streamer("zfr", w
   ↪   + 1), Streamer("zfi", w + 1))

  override val iopaths = List((xpr, znr), (xpi, zni))

  val cmb = Combinational {
    val xfr_xpr_sum = xfr + xpr
    val xfi_xpi_sum = xfi + xpi
    val xfr_xpr_diff = xfr - xpr
    val xfi_xpi_diff = xfi - xpi

    val znr_mux = Mux2to1("znr_mux", w + 1)
    val znr_mux_comm = znr_mux inLinks(s ~> znr_mux.sel, xfr ~> znr_mux.din1, xfr_xpr_sum ~>
     ↪   znr_mux.din2) outLinks (znr_mux.dout ~> znr)

    val zni_mux = Mux2to1("zni_mux", w + 1)
    val zni_mux_comm = zni_mux inLinks(s ~> zni_mux.sel, xfi ~> zni_mux.din1, xfi_xpi_sum ~>
     ↪   zni_mux.din2) outLinks (zni_mux.dout ~> zni)

    val zfr_mux = Mux2to1("zfr_mux", w + 1)
    val zfr_mux_comm = zfr_mux inLinks(s ~> zfr_mux.sel, xpr ~> zfr_mux.din1, xfr_xpr_diff ~>
     ↪   zfr_mux.din2) outLinks (zfr_mux.dout ~> zfr)

    val zfi_mux = Mux2to1("zfi_mux", w + 1)
    val zfi_mux_comm = zfi_mux inLinks(s ~> zfi_mux.sel, xpi ~> zfi_mux.din1, xfi_xpi_diff ~>
     ↪   zfi_mux.din2) outLinks (zfi_mux.dout ~> zfi)

    :=(xfr_xpr_sum, xfi_xpi_sum, xfr_xpr_diff, xfi_xpi_diff, znr_mux_comm, zni_mux_comm,
     ↪   zfr_mux_comm, zfi_mux_comm)
  }

  override val name: String = inst + "_bf2i"
  override val width: Int = w + 1
}
```

Listing 22: Butterfly I SdrLift code.

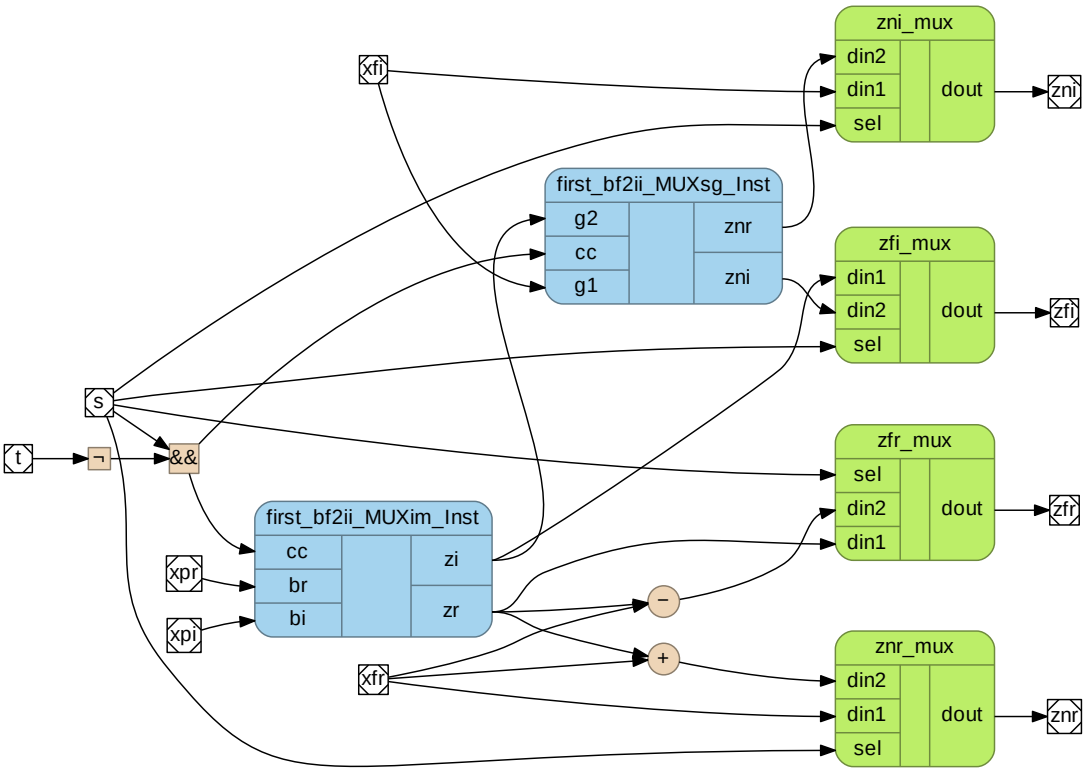


Figure A.4: Butterfly II DFG.

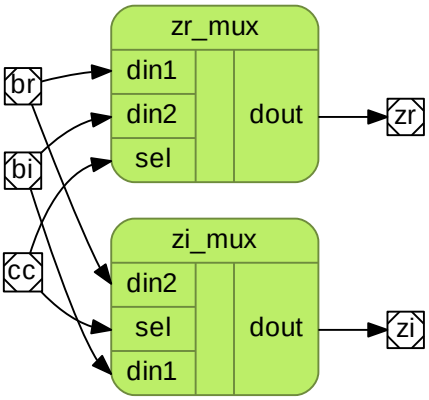


Figure A.5: A MUXim multiplexer.

```scala
case class BF2II(inst: String, w: Int) extends Component {
  //inputs
  val (s, t, xpr, xpi, xfr, xfi) = (Streamer("s", 1), Streamer("t", 1), Streamer("xpr", w),
    ↪  Streamer("xpi", w), Streamer("xfr", w + 1), Streamer("xfi", w + 1))
  // outputs
  val (znr, zni, zfr, zfi) = (Streamer("znr", w + 1), Streamer("zni", w + 1), Streamer("zfr", w
    ↪  + 1), Streamer("zfi", w + 1))

  override val iopaths = List((xpr, znr), (xpi, zni))

  val cmb = Combinational {
    val t_not = t !
    val cc = s && t_not;

    val muxim = MUXim(inst, w)
    val muxim_comm = muxim inLinks(cc ~> muxim.cc, xpr ~> muxim.br, xpi ~> (muxim, muxim.bi))

    val muxsg = MUXsg(inst, w + 1)
    val muxsg_comm = muxsg inLinks(cc ~> muxsg.cc, xfi ~> muxsg.g1, (muxim, muxim.zi) ~>
      ↪  muxsg.g2)

    val xfr_xpr_sum = xfr + (muxim, muxim.zr)
    val xfr_xpr_diff = xfr - (muxim, muxim.zr)

    val znr_mux = Mux2to1("znr_mux", w + 1)
    val znr_mux_comm = znr_mux inLinks(s ~> znr_mux.sel, xfr ~> znr_mux.din1, xfr_xpr_sum ~>
      ↪  znr_mux.din2) outLinks (znr_mux.dout ~> znr)

    val zni_mux = Mux2to1("zni_mux", w + 1)
    val zni_mux_comm = zni_mux inLinks(s ~> zni_mux.sel, xfi ~> zni_mux.din1, (muxsg, muxsg.h1)
      ↪  ~> zni_mux.din2) outLinks (zni_mux.dout ~> zni)

    val zfr_mux = Mux2to1("zfr_mux", w + 1)
    val zfr_mux_comm = zfr_mux inLinks(s ~> zfr_mux.sel, (muxim, muxim.zr) ~> zfr_mux.din1,
      ↪  xfr_xpr_diff ~> zfr_mux.din2) outLinks (zfr_mux.dout ~> zfr)

    val zfi_mux = Mux2to1("zfi_mux", w + 1)
    val zfi_mux_comm = zfi_mux inLinks(s ~> zfi_mux.sel, (muxim, muxim.zi) ~> zfi_mux.din1,
      ↪  (muxsg, muxsg.h2) ~> zfi_mux.din2) outLinks (zfi_mux.dout ~> zfi)

    :=(cc, muxim_comm, muxsg_comm, xfr_xpr_sum, xfr_xpr_diff, znr_mux_comm, zni_mux_comm,
      ↪  zfr_mux_comm, zfi_mux_comm)
  }
  override val name: String = inst + "_bf2ii"
  override val width: Int = w + 1
}
```

Listing 23: Butterfly II SdrLift code.

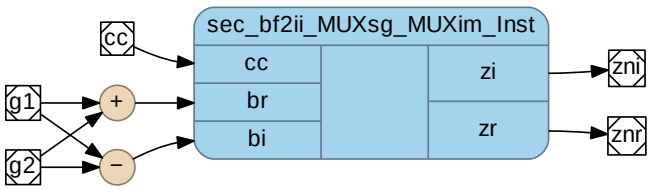


Figure A.6: A MUXsg multiplexer.

```
case class MUXim(id: String, w: Int) extends Component {
  //inputs
  val (cc, br, bi) = (Streamer("cc", 1), Streamer("br", w), Streamer("bi", w))
  // outputs
  val (zr, zi) = (Streamer("zr", w), Streamer("zi", w))
  override val iopaths = List((br, zr), (bi, zi))
  val cmb = Combinational {
    val zr_mux = Mux2to1("zr_mux", w)
    val zr_mux_comm = zr_mux inLinks(cc ~> (zr_mux, zr_mux.sel), br ~> (zr_mux, zr_mux.din1), bi
     ↪   ~> (zr_mux, zr_mux.din2)) outLinks ((zr_mux, zr_mux.dout) ~> zr)
    val zi_mux = Mux2to1("zi_mux", w)
    val zi_mux_comm = zi_mux inLinks(cc ~> (zi_mux, zi_mux.sel), bi ~> (zi_mux, zi_mux.din1), br
     ↪   ~> (zi_mux, zi_mux.din2)) outLinks ((zi_mux, zi_mux.dout) ~> zi)
    :=(zr_mux_comm, zi_mux_comm)
  }
  override val name: String = id + "_MUXim"
  override val inst: String = name + "_Inst"
}
```

Listing 24: SdrLift code for MUXim multiplexer.

```
case class MUXsg(id: String, w: Int) extends Component {
  //inputs
  val (cc, g1, g2) = (Streamer("cc", 1), Streamer("g1", w), Streamer("g2", w))
  // outputs
  val (h1, h2) = (Streamer("znr", w), Streamer("zni", w))

  override val iopaths = List((g1, h1))

  val cmb = Combinational {
    val g1_g2_sum = g1 + g2
    val g1_g2_diff = g1 - g2
    val muxim = MUXim(id + "_MUXsg", w)
    val muxim_comm = muxim inLinks(cc ~> muxim.cc, g1_g2_sum ~> muxim.br, g1_g2_diff ~>
     ↪   muxim.bi) outLinks(muxim.zr ~> h1, muxim.zi ~> h2)
    :=(g1_g2_sum, g1_g2_diff, muxim_comm)
  }
  override val name: String = id + "_MUXsg"
  override val inst: String = name + "_Inst"
  override val width: Int = w

  override def dfg: Graph[DfgNode, DfgEdge] = model(Seq(cmb))
}
```

Listing 25: SdrLift code for MUXsg multiplexer.

```
case class Tx80211a(name: String) extends SdrApp {
  val (zp, cpa) = (ZeroPadMod("zpInst", 16, 48, 16), CycliPrefixAddMod("cpaInst", 16 + 6, 64,
   ↪  16))
  val ifft = IFFT_N64("ifft64", 16)
  val (source, qam, sink) =
    (Src("sourceInst", 4), Modulator("qamInst"), Snk("sinkInst", 22))

  val chn = Chain(
    source outLinks (source.dout ~> (qam, qam.din)),
    qam outLinks(qam.iout ~> (zp, zp.iin), qam.qout ~> (zp, zp.qin)),
    zp outLinks(zp.iout ~> (ifft, ifft.xnr), zp.qout ~> (ifft, ifft.xni)),
    ifft outLinks(ifft.xkr ~> (cpa, cpa.iin), ifft.xki ~> (cpa, cpa.qin)),
    cpa outLinks(cpa.iout ~> (sink, sink.iin), cpa.qout ~> (sink, sink.qin)),
    sink
  )

  //override val name: String = "tx80211a"
  override val sdfap = model(Seq(chn))
}
```

Listing 26: SdrLift code for OFDM-TX (IEEE 802.11a).

References

[1] P. Burns, *Software defined radio for 3G*. Artech house, 2002.

[2] L. Tsoeunyane, S. Winberg, and M. Inggs, "Software-defined radio FPGA cores: Building towards a domain-specific language," *International Journal of Reconfigurable Computing*, vol. 2017, 2017.

[3] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu, *VLSI physical design: from graph partitioning to timing closure*. Springer Science & Business Media, 2011.

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

[5] X. Li, O. Hammami, and E. Paristech, "Fast design productivity for embedded multiprocessor through Multi-FPGA emulation: The case of a 48-way multiprocessor with noc." https://www.design-reuse.com/articles/21324/multi-fpga-emulation-multiprocessor-noc.html, Dec 2008.

[6] Y. S. Shao and D. Brooks, *Research Infrastructures for Hardware Accelerators*. Morgan & Claypool, 2015.

[7] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, pp. 89–100, Oct 2011.

[8] W. H. W. TUTTLEBE, "Software- defined radio: Facets of a developing technology," *IEEE Personal Communications*, vol. 6, pp. 38–44, Apr. 1999.

[9] J. Ghetie, "Fixed wireless and cellular mobile convergence: Technologies, solutions, services," in *Telecommunications, 2007. ConTel 2007. 9th International Conference on*, pp. 343–343, June 2007.

[10] C. E. Caicedo and P. D. Student, "Software defined radio and software radio technology: Concepts and application," *Department of Information Science and Telecommunications University of Pittsburgh*, 2007.

[11] W. Ecker, W. MÃijller, and R. Domer, eds., *Hardware-dependent software : principles and practice*. Berlin: Springer, 2009.

[12] R. Akeela and B. Dezfouli, "Software-defined radios: Architecture, state-of-the-art, and challenges," *Computer Communications*, vol. 128, pp. 106 – 125, 2018.

[13] A. C. Tribble, "The software defined radio: Fact and fiction," in *2008 IEEE Radio and Wireless Symposium*, pp. 5–8, Jan 2008.

[14] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, pp. 473–491, April 2011.

[15] W. H. W. Tuttlebee, "Software-defined radio: facets of a developing technology," *IEEE Personal Communications*, vol. 6, pp. 38–44, April 1999.

[16] T. J. Rouphael, *RF and digital signal processing for software-defined radio: a multi-standard multi-mode approach*. Newnes, 2009.

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

[18] S. Neuendorffer and K. Vissers, "Streaming systems in FPGAs," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pp. 147–156, Springer, 2008.

[19] P. Chu, *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. Wiley - IEEE, Wiley, 2006.

[20] M. Simulink and M. Natick, "Simulink user's guide." https://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf, 2018. Accessed: 2018-10-19.

[21] Xilinx, "Accelerating integration: Block-based IP integration with Vivado IP integrator." https://www.xilinx.com/products/design-tools/vivado/integration.html. Accessed: 2018-02-18.

[22] H. A. Andrade and S. Kovner, "Software synthesis from dataflow models for g and LabVIEW," in *In Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pp. 1705–1709, 1998.

[23] GNU, "Gnu radio: The free & open software radio ecosystem." https://www.gnuradio.org/, 2019. Accessed: 2019-10-07.

[24] R. Prabhakar, D. Koeplinger, K. J. Brown, H. Lee, C. De Sa, C. Kozyrakis, and K. Olukotun, "Generating configurable hardware from parallel patterns," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, (New York, NY, USA), pp. 651–665, ACM, 2016.

[25] D. Koeplinger, C. Delimitrou, R. Prabhakar, C. Kozyrakis, Y. Zhang, and K. Olukotun, "Automatic generation of efficient accelerators for reconfigurable hardware," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, (Piscataway, NJ, USA), pp. 115–127, IEEE Press, 2016.

[26] S. Tripakis, H. Andrade, A. Ghosal, R. Limaye, K. Ravindran, G. Wang, G. Yang, J. Kornerup, and I. Wong, "Correct and non-defensive glue design

using abstract models," in *2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 59–68, Oct 2011.

[27] C. Y. Lin, Z. Jiang, C. Fu, H. K.-H. So, and H. Yang, "FPGA high-level synthesis versus overlay: Comparisons on computation kernels," *SIGARCH Comput. Archit. News*, vol. 44, pp. 92–97, Jan 2017.

[28] L. J. Tsoeunyane, S. Winberg, and M. Inggs, "An IP core integration toolflow for prototyping software-defined radios using static dataflow with access patterns," in *2017 International Conference on Field Programmable Technology (ICFPT)*, pp. 88–95, Dec 2017.

[29] V. D'silva, S. Ramesh, and A. Sowmya, "Synchronous protocol automata: a framework for modelling and verification of SoC communication architectures," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, vol. 1, pp. 390–395 Vol.1, Feb 2004.

[30] K. Sano, "DSL-based design space exploration for temporal and spatial parallelism of custom stream computing," in *Proceedings of the Second International Workshop on FPGAs for Software Programmers*, Sep 2015.

[31] L. J. Mohapi, S. Winberg, and M. Inggs, "A domain-specific language to facilitate software defined radio parallel executable patterns deployment on heterogeneous architectures," in *2014 IEEE 33rd International Performance Computing and Communications Conference (IPCCC)*, pp. 1–8, Dec 2014.

[32] A. Berls, "Graph for Scala: scalax.collection.graph." http://www.scala-graph.org/. Accessed: 2018-08-06.

[33] C. Pohl, C. Paiz, and M. Porrmann, "vMAGIC: automatic code generation for VHDL," *International Journal of Reconfigurable Computing*, vol. 2009, 2009.

[34] L. Tsoeunyane, S. Winberg, and M. Inggs, "Automatic configurable hardware code generation for software-defined radios," *Computers*, vol. 7, no. 4, 2018.

[35] L. Tsoeunyane, S. Winberg, and M. Inggs, "SdrLift: An intermediate-level framework for synthesis of software-defined radio accelerators," in *2019 IEEE 10th International Conference on Mechanical and Intelligent Manufacturing Technologies (ICMIMT)*, pp. 166–173, Feb 2019.

[36] Wireless-Innovation-Forum, "What is software defined radio?." https://www.wirelessinnovation.org/Introduction_to_SDR. Accessed: 2019-12-23.

[37] J. Mitola, "Software radios: Survey, critical evaluation and future directions," *IEEE Aerospace and Electronic Systems Magazine*, vol. 8, pp. 25–36, April 1993.

[38] F. Dowla, *Handbook of RF and wireless technologies*. Newnes, 2003.

[39] G. S. Ouedraogo, M. Gautier, and O. Sentieys, "A frame-based domain-specific language for rapid prototyping of FPGA-based software-defined radios," *EURASIP Journal on Advances in Signal Processing*, vol. 2014, p. 164, Nov 2014.

[40] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, and M. LujÃąn, "An empirical evaluation of high-level synthesis languages and tools for database acceleration," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, Sept 2014.

[41] T. Yang, W. A. Davis, W. L. Stutzman, S. M. Shajedul Hasan, R. Nealy, C. B. Dietrich, and J. H. Reed, "Antenna design strategy and demonstration for software-defined radio (SDR)," *Analog Integrated Circuits and Signal Processing*, vol. 69, p. 161, Sep 2011.

[42] P. S. Hall, P. Gardner, and A. Faraone, "Antenna requirements for software defined and cognitive radios," *Proceedings of the IEEE*, vol. 100, pp. 2262–2270, July 2012.

[43] S. C. Pires, P. M. Cabral, and J. C. Pedro, "A carrier-burst transmitter implementation: Design of bandpass filter and amplifier-BPF connection," in *2012 Workshop on Integrated Nonlinear Microwave and Millimetre-wave Circuits*, pp. 1–3, Sep 2012.

[44] F. Maloberti, *Data converters*. Springer Science & Business Media, 2007.

[45] T. Hofner, "Measuring and evaluating dynamic ADC parameters," *Microwaves and RF*, vol. 39, pp. 78–94, 2000.

[46] S. Bowling, "Understanding A/D converter performance specifications." http://ww1.microchip.com/downloads/en/appnotes/00693a.pdf, 2000.

[47] O. Romain and B. Denby, "Prototype of a software-defined broadcast media indexing engine," in *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, vol. 2, pp. II–813–II–816, April 2007.

[48] T. Hentschel, M. Henker, and G. Fettweis, "The digital front-end of software radio terminals," *IEEE Personal Communications*, vol. 6, pp. 40–46, Aug 1999.

[49] A. Vinod and E. Lai, "Low power and high-speed implementation of fir filters for software defined radio receivers," *Wireless Communications, IEEE Transactions on*, vol. 5, pp. 1669–1675, July 2006.

[50] L. Pucker, "Channelization techniques for software defined radio," in *Proceedings of SDR Forum Conference*, pp. 1–6, 2003.

[51] Jyothi.N, Jayaprakash.S, and S. K. Gowda, "Design and VLSI implementation of high performance DUC and DDC for software defined radio applications," in *2013 International Conference on Emerging Trends in Communi-*

cation, Control, Signal Processing and Computing Applications (C2SPCA), pp. 1–3, Oct 2013.

[52] J. Lillington and S. Matthews, "Flexible architectures for wideband SDR channelisation," in *2005 The 2nd IEE/EURASIP Conference on DSPenabledRadio (Ref. No. 2005/11086)*, pp. 14–14/5, Sep 2005.

[53] S. Knapp, "Using programmable logic to accelerate DSP functions." http://ebook.pldworld.com/_semiconductors/XILINX/AppLINX%20CD-ROM/Rev.7%20(Q3-1998)/docs/wcd0000f/wcd00ff9.pdf, Oct 1995.

[54] Z. Geng, X. Wei, H. Liu, R. Xu, and K. Zheng, "Performance analysis and comparison of GPP-based SDR systems," in *2017 7th IEEE International Symposium on Microwave, Antenna, Propagation, and EMC Technologies (MAPE)*, pp. 124–129, Oct 2017.

[55] A. Azarian and M. Ahmadi, "Reconfigurable computing architecture survey and introduction," in *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*, pp. 269–274, Aug 2009.

[56] M. Baese, *Digital Signal Processing with Field Programmable Gate Arrays*. Berlin: Springer, 3rd ed., 2007.

[57] M. Alawieh, M. Kasparek, N. Franke, and J. Hupfer, "A high performance FPGA-GPU-CPU platform for a real-time locating system," in *2015 23rd European Signal Processing Conference (EUSIPCO)*, pp. 1576–1580, Aug 2015.

[58] R. Woods, J. McAllister, G. Lightboy, and Y. Yi, *FPGA-based Implementation of Complex Signal Processing Systems*. John Wiley and Sons, Ltd, 2008.

[59] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. P. Pande, C. Grecu, and A. Ivanov, "System-on-chip: Reuse and integration," *Proceedings of the IEEE*, vol. 94, pp. 1050–1069, June 2006.

[60] J. Pimentel and H. Le-Huy, "A VHDL library of IP cores for power drive and motion control applications," in *Electrical and Computer Engineering, 2000 Canadian Conference on*, vol. 1, pp. 184–188 vol.1, 2000.

[61] D. Backer, C. Chang, D. Chapman, H. Chen, P. Droz, C. de Jesus, D. MacMahon, A. Siemion, J. Wawrzynek, D. Werthimer, and M. Wright, "A new approach to radio astronomy signal processing: Packet switched, FPGA-based, upgradable, modular hardware and reusable, platform-independent signal processing libraries," *National Radio Science Meeting*, 2006.

[62] F. Fang, J. C. Hoe, M. Püschel, and S. Misra, "Generation of custom DSP transform IP cores: Case study Walsh-Hadamard transform," in *High Performance Embedded Computing (HPEC)*, 2002.

[63] J. Gaisler, "An open-source VHDL IP library with plug&play configuration," in *Building the Information Society*, pp. 711–717, Springer, 2004.

[64] A. Lopez-Parrado and J.-C. Valderrama-Cuervo, "OpnRISC-based system-on-chip for digital signal processing," in *Image, Signal Processing and Artificial Vision (STSIVA), 2014 XIX Symposium on*, pp. 1–5, Sep 2014.

[65] J. Hickish, Z. Abdurashidova, Z. Ali, K. D. Buch, S. C. Chaudhari, H. Chen, M. Dexter, R. S. Domagalski, J. Ford, G. Foster, D. George, J. Greenberg, L. Greenhill, A. Isaacson, H. Jiang, G. Jones, F. Kapp, H. Kriel, R. Lacasse, A. Lutomirski, D. MacMahon, J. Manley, A. Martens, R. McCullough, M. V. Muley, W. New, A. Parsons, D. C. Price, R. A. Primiani, J. Ray, A. Siemion, V. Van Tonder, L. Vertatschitsch, M. Wagner, J. Weintroub, and D. a. Werthimer, "A decade of developing radio-astronomy instrumentation using CASPER open-source technology," *Journal of Astronomical Instrumentation*, vol. 05, no. 04, p. 1641001, 2016.

[66] S. Scott, "RHINO: Reconfigurable hardware interface for computation and radio," Master's thesis, University Of Cape Town, Nov. 2011.

[67] Casper, "IBOB: Interconnect break-out board." https://casper.ssl.berkeley.edu/wiki/IBOB. Accessed: 2020-01-25.

[68] Casper, "Roach." https://casper.ssl.berkeley.edu/wiki/ROACH. Accessed: 2020-01-25.

[69] R. Brodersen, A. Tkachenko, and H. Kwok-Hay So, "A unified hardware/-software runtime environment for FPGA-based reconfigurable computers using BORPH," in *Hardware/Software Codesign and System Synthesis, 2006. CODES+ISSS '06. Proceedings of the 4th International Conference*, pp. 259–264, Oct 2006.

[70] Casper, "SKARAB." https://casper.ssl.berkeley.edu/wiki/SKARAB. Accessed: 2020-01-25.

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

[72] M. Inggs, G. Inggs, A. Langman, and S. Scott, "Growing horns: Applying the rhino software defined radio system to radar," in *2011 IEEE RadarCon (RADAR)*, pp. 951–955, May 2011.

[73] M. C. Ng, K. E. Fleming, M. Vutukuru, S. Gross, Arvind, and H. Balakrishnan, "Airblue: A system for cross-layer wireless protocol development," in *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '10, (New York, NY, USA), pp. 4:1–4:11, ACM, 2010.

[74] P. Murphy, A. Sabharwal, and B. Aazhang, "Design of WARP: A wireless open-access research platform," in *2006 14th European Signal Processing Conference*, pp. 1–5, Sep 2006.

[75] B. Drozdenko, M. Zimmermann, T. Dao, K. Chowdhury, and M. Leeser, "Hardware-software codesign of wireless transceivers on Zynq heterogeneous systems," *IEEE Transactions on Emerging Topics in Computing*, vol. PP, no. 99, pp. 1–1, 2017.

[76] "AD-FMCOMMS3-EBZ user guide." https://wiki.analog.com/resources/eval/user-guides/ad-fmcomms3-ebz, Sep 2019. Accessed: 2019-12-23.

[77] "AD9361, ANALOG DEVICES-RF agile transceiver." https://www.analog.com/media/en/technical-documentation/data-sheets/AD9361.pdf, 11 2016. Accessed: 2019-12-23.

[78] MathWorks, "Simulink user's guide, revised for Simulink 9.3." https://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf, Mar 2019. Accessed: 2019-03-31.

[79] J. Huie, P. D'Antonio, R. Pelt, and B. Jentz, "Synthesizing FPGA cores for software-defined radio," in *SDR Forum, Nov*, 2003.

[80] F. Morgan, T. El-Ghazawi, and H. Amano, *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2010.

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

[82] N. George, H. Lee, D. Novo, T. Rompf, K. J. Brown, A. K. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne, "Hardware system synthesis from domain-specific languages," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, Sept 2014.

[83] 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, Sept 2015.

[84] 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.

[85] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Transactions on*

Computer-Aided Design of Integrated Circuits and Systems, vol. PP, no. 99, pp. 1–14, 2016.

[86] P. Coussy and A. Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Science and Business Media B.V, 2008.

[87] J. Matai, D. Richmond, D. Lee, and R. Kastner, "Enabling FPGAs for the masses," *CoRR*, vol. abs/1408.5870, 2014.

[88] I. Graves, A. Procter, W. L. Harrison, M. Becchi, and G. Allwein, "Hardware synthesis from functional embedded domain-specific languages: A case study in regular expression compilation," in *Applied Reconfigurable Computing*, pp. 41–52, Springer, 2015.

[89] L. J. Mohapi, S. Winberg, and M. Inggs, "Using a domain specific language for SDR to facilitate radar signal processing in heterogeneous computing architectures," in *2015 IEEE Radar Conference*, pp. 306–311, Oct 2015.

[90] N. George, D. Novo, T. Rompf, M. Odersky, and P. Ienne, "Making domain-specific hardware synthesis tools cost-efficient," in *Field-Programmable Technology (FPT), 2013 International Conference on*, pp. 120–127, Dec 2013.

[91] C. Kulkarni, G. Brebner, and G. Schelle, "Mapping a domain specific language to a platform FPGA," in *Design Automation Conference, 2004. Proceedings. 41st*, pp. 924–927, July 2004.

[92] M. Pelcat, S. Aridhi, J. Piat, and J.-F. Nezan, *Dataflow Model of Computation*, pp. 53–75. London: Springer London, 2013.

[93] A. Sangiovanni-Vincentelli, "Quo Vadis, SLD? reasoning about the trends and challenges of system level design," *Proceedings of the IEEE*, vol. 95, pp. 467–506, March 2007.

[94] H. Berg, C. Brunelli, and U. Lucking, "Analyzing models of computation for software defined radio applications," in *System-on-Chip, 2008. SOC 2008. International Symposium on*, pp. 1–4, Nov 2008.

[95] F. Siyoum, M. Geilen, O. Moreira, R. Nas, and H. Corporaal, "Analyzing synchronous dataflow scenarios for dynamic software-defined radio applications," in *System on Chip (SoC), 2011 International Symposium on*, pp. 14–21, Oct 2011.

[96] K. Gilles, "The semantics of a simple language for parallel programming," *Information processing*, vol. 74, pp. 471–475, 1974.

[97] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, pp. 773–801, May 1995.

[98] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[99] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static data flow," in *1995 International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, pp. 3255–3258 vol.5, May 1995.

[100] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling of DSP systems," in *2000 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.00CH37100)*, vol. 6, pp. 3362–3365 vol.6, June 2000.

[101] V. Bebelis, P. Fradet, A. Girault, and B. Lavigueur, "BPDF: A statically analyzable dataflow model with integer and boolean parameters," in *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, pp. 1–10, Sep 2013.

[102] D. D. Gajski and J. Kleinsmith, *Principles of digital design*, vol. 42. Prentice Hall Upper Saddle River, NJ, 1997.

[103] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, pp. 541–580, April 1989.

[104] J. Castrillon, S. Schürmans, A. Stulova, W. Sheng, T. Kempf, R. Leupers, G. Ascheid, and H. Meyr, "Component-based waveform development: the nucleus tool flow for efficient and portable software defined radio," *Analog Integrated Circuits and Signal Processing*, vol. 69, no. 2-3, p. 173, 2011.

[105] C. Brooks, E. A. Lee, X. Liu, Y. Zhao, H. Zheng, S. S. Bhattacharyya, C. Brooks, and E. Cheong, "Ptolemy II - heterogeneous concurrent modeling and design in Java," tech. rep., University of California at Berkeley, 2005.

[106] J. Serot, F. Berry, and S. Ahmed, "Implementing stream-processing applications on FPGAs: A DSL-based approach," in *2011 21st International Conference on Field Programmable Logic and Applications*, pp. 130–137, Sept 2011.

[107] C.-J. Hsu, M.-Y. Ko, and S. S. Bhattacharyya, "Software synthesis from the dataflow interchange format," in *Proceedings of the 2005 workshop on Software and compilers for embedded systems*, pp. 37–49, ACM, 2005.

[108] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. C-36, pp. 24–35, Jan 1987.

[109] O. M. Moreira and M. J. G. Bekooij, "Self-timed scheduling analysis for real-time applications," *EURASIP Journal on Advances in Signal Processing*, vol. 2007, no. 1, p. 083710, 2007.

[110] P. Poplavko, T. Basten, M. Bekooij, J. van Meerbergen, and B. Mesman, "Task-level timing models for guaranteed performance in multiprocessor networks-on-chip," in *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03, (New York, NY, USA), pp. 63–72, ACM, 2003.

[111] A. Ghosal, R. Limaye, K. Ravindran, S. Tripakis, A. Prasad, G. Wang, T. N. Tran, and H. Andrade, "Static dataflow with access patterns: Semantics and analysis," in *DAC Design Automation Conference 2012*, pp. 656–663, June 2012.

[112] H. Andrade, J. Correll, A. Ekbal, A. Ghosal, D. Kim, J. Kornerup, R. Limaye, A. Prasad, K. Ravindran, T. Tran, *et al.*, "From streaming models to FPGA implementations," in *Proc. of the Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2012.

[113] H. Kee, C.-C. Shen, S. S. Bhattacharyya, I. Wong, Y. Rao, and J. Kornerup, "Mapping parameterized cyclo-static dataflow graphs onto configurable hardware," *J. Signal Process. Syst.*, vol. 66, pp. 285–301, Mar 2012.

[114] O. Moreira and H. Corporaal, *Scheduling Real-Time Streaming Applications Onto an Embedded Multiprocessor*. Springer, 2013.

[115] S. Saha, S. Puthenpurayil, and S. S. Bhattacharyya, "Dataflow transformations in high-level DSP system design," in *2006 International Symposium on System-on-Chip*, pp. 1–6, IEEE, 2006.

[116] K. Ravindran, A. Ghosal, R. Limaye, G. Wang, G. Yang, and H. Andrade, "Analysis techniques for static dataflow models with access patterns," in *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing*, pp. 1–8, Oct 2012.

[117] S. O. Settle, "High-performance dynamic programming on FPGAs with OpenCL," in *Proc. IEEE High Perform. Extreme Comput. Conf.(HPEC)*, pp. 1–6, 2013.

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

[119] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, *High-Level Synthesis: From Algorithm to Digital Circuit*, ch. AutoPilot: A Platform-Based ESL Synthesis System, pp. 99–112. Dordrecht: Springer Netherlands, 2008.

[120] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W. M. W. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, pp. 35–42, July 2009.

[121] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pp. 75–86, March 2004.

[122] B. Fort, A. Canis, J. Choi, N. Calagar, R. Lian, S. Hadjis, Y. T. Chen, M. Hall, B. Syrowik, T. Czajkowski, S. Brown, and J. Anderson, "Automating the design of processor/accelerator embedded systems with LegUp high-level synthesis," in *Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on*, pp. 120–129, Aug 2014.

[123] A. Siemion, "Developing radio astronomy instruments with Simulink libraries." https://www.mathworks.com/company/newsletters/articles/developing-radio-astronomy-instruments-with-simulink-libraries.html, April 2011. Accessed: 2020-02-09.

[124] D. Werthimer, "The CASPER collaboration for high-performance open source digital radio astronomy instrumentation," in *General Assembly and Scientific Symposium, 2011 XXXth URSI*, pp. 1–4, Aug 2011.

[125] Casper, "CASPER wiki, getting started with CASPER." https://casper.berkeley.edu/wiki. Accessed: 2016-05-16.

[126] T. Rompf and M. Odersky, "Lightweight Modular Staging: A pragmatic approach to runtime code generation and compiled DSLs," in *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, (New York, NY, USA), pp. 127–136, ACM, 2010.

[127] A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Wu, A. R. Atreya, K. Olukotun, T. Rompf, and M. Odersky, "OptiML: an implicitly parallel domain specific language for machine learning," in *Proceedings of the 28th International Conference on Machine Learning, ser. ICML*, 2011.

[128] S. Bourdeauducq, *Migen annual release X*, July 2013.

[129] S. Mittal, S. Gupta, and S. Dasgupta, "System generator: The state-of-art FPGA design tool for DSP applications," in *Third International Innovative Conference On Embedded Systems, Mobile Communication And Computing (ICEMC2 2008)*, pp. 187–190, 2008.

[130] K. Ravindran, A. Ghosal, R. Limaye, D. Kim, H. Andrade, J. Correll, J. Kornerup, I. Wong, G. Wang, G. Yang, A. Ekbal, M. Trimborn, A. Prasad, and T. N. Tran, *Modeling, Analysis, and Implementation of Streaming Applications for Hardware Targets*, pp. 19–39. New York, NY: Springer New York, 2014.

[131] J. I. Villar, J. Juan, M. J. Bellido, J. Viejo, D. Guerrero, and J. Decaluwe, "Python as a hardware description language: A case study," in *Programmable Logic (SPL), 2011 VII Southern Conference on*, pp. 117–122, April 2011.

[132] R. H. L. Stroop, "Enhancing GNU radio for run-time assembly of FPGA-based accelerators," Master's thesis, Virginia Polytechnic Institute and State University, 8 2012.

[133] Altera, "SOPC builder, Altera Inc.." https://www.altera.com/en_US/pdfs/literature/ug/ug_sopc_builder.pdf, Dec 2010. Accessed: 2018-02-21.

[134] V. S. I. Alliance, "Legacy documents of the VSI alliance." https://vsia.org/. Accessed: 2018-02-18.

[135] OCPIP. http://www.ocpip.org/. Accessed: 2018-02-18.

[136] IEEE, "IEEE standard for IP-XACT, standard structure for packaging, integrating, and reusing ip within tool flows," *IEEE Std 1685-2014 (Revision of IEEE Std 1685-2009)*, pp. 1–510, Sept 2014.

[137] Accellera. http://www.accellera.org/. Accessed: 2018-02-18.

[138] M. Jassi, D. MÃijller-Gritschneder, and U. Schlichtmann, "GRIP: Grammar-based IP integration and packaging for acceleration-rich SoC

designs," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2015.

[139] A. M. Chana and P. Quinton, *Intellectual Property (IP) Integration Approach for Data-Flow Parallel Embedded Systems*, pp. 298–307. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.

[140] H. Nikolov, T. Stefanov, and E. Deprettere, "Automated integration of dedicated hardwired IP cores in heterogeneous MPSoCs designed with ESPAM," *EURASIP Journal on Embedded Systems*, vol. 2008, p. 726096, Apr 2008.

[141] 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, pp. 127–144, Jan 2003.

[142] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet, "OpenDF: A dataflow toolset for reconfigurable hardware and multicore systems," *SIGARCH Comput. Archit. News*, vol. 36, pp. 29–35, Jun 2009.

[143] L. Yang, M. Ikram, S. Gurumani, S. Fahmy, D. Chen, and K. Rupnow, "JIT trace-based verification for high-level synthesis," in *Field Programmable Technology (FPT), 2015 International Conference on*, pp. 228–231, Dec 2015.

[144] G. Stewart, M. Gowda, G. Mainland, B. Radunovic, D. Vytiniotis, and C. L. Agullo, "Ziria: A DSL for wireless systems programming," *SIGPLAN Not.*, vol. 50, pp. 415–428, Mar. 2015.

[145] T. E. Dwan and T. E. Bechert, "Introducing Simulink into a systems engineering curriculum," in *Frontiers in Education Conference, 1993. Twenty-Third Annual Conference. 'Engineering Education: Renewing America's Technology', Proceedings.*, pp. 627–631, Nov 1993.

[146] G. Wang, R. Allen, H. Andrade, and A. Sangiovanni-Vincentelli, "Communication storage optimization for static dataflow with access patterns

under periodic scheduling and throughput constraint," *Computers & Electrical Engineering*, vol. 40, no. 6, pp. 1858 – 1873, 2014.

[147] K. Du, S. Domas, and M. Lenczner, "A solution to overcome some limitations of SDF based models," in *2018 IEEE International Conference on Industrial Technology (ICIT)*, pp. 1395–1400, Feb 2018.

[148] K. Du, S. Domas, and M. Lenczner, "Actors with stretchable access patterns," *Integration*, vol. 66, pp. 44 – 59, 2019.

[149] P. R. Schaumont, *Data Flow Modeling and Transformation*, pp. 31–59. Springer US, 2013.

[150] S. Bhattacharyya, P. Murthy, and E. Lee, *Software Synthesis from Dataflow Graphs*. The Springer International Series in Engineering and Computer Science, Springer US, 2012.

[151] S. Tripakis, R. Limaye, K. Ravindran, and G. Wang, "On tokens and signals: Bridging the semantic gap between dataflow models and hardware implementations," in *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pp. 51–58, July 2014.

[152] EPFL, "The Scala programming language." https://www.scala-lang.org/. Accessed: 2020-07-07.

[153] P. Chiusano and R. Bjarnason, *Functional Programming in Scala*. Manning Publications, 2014.

[154] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*. Artima, 2008.

[155] C. K. Loverdos and A. Syropoulos, *Steps in Scala: an introduction to object-functional programming*. Cambridge University Press, 2010.

[156] A. Sharifian, R. Hojabr, N. Rahimi, S. Liu, A. Guha, T. Nowatzki, and A. Shriraman, "MIR - an intermediate representation for transforming and

optimizing the microarchitecture of application accelerators," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, (New York, NY, USA), pp. 940 – 953, Association for Computing Machinery, 2019.

[157] N. Sane, H. Kee, G. Seetharaman, and S. S. Bhattacharyya, "Topological patterns for scalable representation and analysis of dataflow graphs," *Journal of Signal Processing Systems*, vol. 65, p. 229, Aug 2011.

[158] Graphviz, "Graphviz - graph visualization software." https://www.graphviz.org/. Accessed: 2018-11-05.

[159] J. Matai, D. Lee, A. Althoff, and R. Kastner, "Composable, parameterizable templates for high-level synthesis," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 744–749, March 2016.

[160] S. He and M. Torkelson, "A new approach to pipeline FFT processor," in *Parallel Processing Symposium, 1996., Proceedings of IPPS '96, The 10th International*, pp. 766–770, Apr 1996.

[161] Shousheng He and M. Torkelson, "Design and implementation of a 1024-point pipeline FFT processor," in *Proceedings of the IEEE 1998 Custom Integrated Circuits Conference (Cat. No.98CH36143)*, pp. 131–134, 1998.

[162] Shousheng He and M. Torkelson, "Designing pipeline FFT processor for OFDM (de)modulation," in *1998 URSI International Symposium on Signals, Systems, and Electronics. Conference Proceedings (Cat. No.98EX167)*, pp. 257–262, 1998.

[163] Xilinx, "LogiCORE IP - fast fourier transform v8. 0." https://www.xilinx.com/support/documentation/ip_documentation/ds808_xfft.pdf, 2012.

[164] A. Saeed, M. Elbably, G. Abdelfadeel, and M. I. Eladawy, "Efficient FPGA implementation of FFT/IFFT processor," *International Journal of Circuits, Systems and Signal Processing*, vol. 3, pp. 103–110, 2009.

[165] A. Saeed, M. Elbably, G. Abdelfadeel, and M. I. Eladawy, "Efficient FPGA implementation of FFT/IFFT processor," *International Journal of Circuits, Systems and Signal Processing*, vol. 3, 2009.

[166] OpenCores, "OpenCores projects." http://www.opencores.org/projects/. Accessed: 2018-09-02.

[167] C. Gaisler, "LEON/GRLIB." https://www.gaisler.com/index.php/downloads/leongrlib/. Accessed: 2018-09-02.

[168] S. A. Edwards, R. Townsend, and M. A. Kim, "Compositional dataflow circuits," in *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, MEMOCODE '17, (New York, NY, USA), pp. 175–184, ACM, 2017.

[169] IEEE, "IEEE Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," *IEEE Std 802.11-2007 (Revision of IEEE Std 802.11-1999)*, pp. 1–1076, June 2007.

[170] S. Gupta and V. Malagar, "IEEE 802.22 Standard for Regional Area Networks," in *2017 International Conference on Next Generation Computing and Information Systems (ICNGCIS)*, pp. 126–130, Dec 2017.

[171] Y. G. Li, J. H. Winters, and N. R. Sollenberger, "MIMO-OFDM for wireless communications: signal detection with enhanced channel estimation," *IEEE Transactions on Communications*, vol. 50, pp. 1471–1477, Sep 2002.

[172] Mini-Circuits, "SXBP-100+ surface mount bandpass filter." http://www.4dsp.com/pdf/FMC150_user_manual.pdf. Accessed: 2018-12-05.

[173] A. Systems, "FMC150 user manual." http://www.4dsp.com/pdf/FMC150_user_manual.pdf, Aug 2010.

[174] Altera, "Understanding CIC compensation filters." https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an455.pdf, Apr 2007.

[175] J. Gao, "10_100_1000 Mbps Tri-mode Ethernet MAC specification." https://opencores.org/ocsvn/ethernet_tri_mode/ethernet_tri_mode/trunk/doc/Tri-mode_Ethernet_MAC_Specifications.pdf, Sep 2018.