

FPGA Programming for Beginners

Bring your ideas to life by creating hardware designs
and electronic circuits with SystemVerilog



Frank Bruno



FPGA Programming for Beginners

Bring your ideas to life by creating hardware designs
and electronic circuits with SystemVerilog

Frank Bruno

Packt

BIRMINGHAM—MUMBAI

FPGA Programming for Beginners

Copyright © 2021 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Wilson D'souza

Associate Publishing Product Manager: Sankalp Khattri

Senior Editor: Rahul Dsouza

Content Development Editor: Nihar Kapadia

Technical Editor: Nithik Cheruvakodan

Copy Editor: Safis Editing

Project Coordinator: Neil D'mello

Proofreader: Safis Editing

Indexer: Tejal Daruwale Soni

Production Designer: Roshan Kawale

First published: March 2021

Production reference: 1050221

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78980-541-3

www.packtpub.com

Contributors

About the author

Frank Bruno is an experienced high-performance design engineer specializing in FPGAs with some ASIC experience. He has experience working for companies such as Cruise, SpaceX, Allston Trading, and Number Nine. He is currently working as an FPGA engineer for Cruise.

About the reviewer

George Kaldis has a bachelor's degree in electrical engineering from Northeastern University and has over 30 years of experience working with FPGAs. He is president of GK-Digital LLC, an FPGA design consulting company. He has implemented many FPGA designs for applications ranging from wireless/wired networking to high-frequency trading and test equipment.

Table of Contents

Preface

Section 1: Introduction to FPGAs and Xilinx Architectures

1

Introduction to FPGA Architectures and Xilinx Vivado

Technical requirements	4	DSP48E1	17
Hardware	4	ASMBL architecture	18
Software	4	Introduction to the Vivado toolset and evaluation boards	19
What is an ASIC?	5	Evaluation boards	19
Why an ASIC or FPGA?	5	Nexys A7 100T (or 50T)	20
How does a company create a programmable device using an ASIC process?	8	Basys 3	22
Fundamental logic gates	8	Introducing Vivado	24
More complex operations	12	Vivado installation	24
Introducing FPGAs	13	Directory structure	25
Exploring the Xilinx Artix-7 and 7 series devices	14	Running the example	29
Combinational logic blocks	14	Summary	42
Storage	16	Questions	42
Clocking	17	Challenge	43
I/Os	17	Further reading	43

Section 2: Introduction to Verilog RTL Design, Simulation, and Implementation

2

Combinational Logic

Technical requirements	47	Creating combinational logic	58
Creating SystemVerilog modules	48	Using custom data types	64
How to create reusable code – parameters	49	Project 1 – creating combinational logic	65
		Testbench	66
Introducing data types	50	Implementing a leading-one detector using the case statement	69
Introducing built-in data types	50	Designing a reusable leading-one detector using a for loop	73
Creating arrays	51	Counting the number of ones	74
Handling multiple-driven nets	54	Implementing an adder/subtractor	74
Handling signed and unsigned numbers	55	Multiplier	76
Adding bits to a signal by concatenating	55	Bringing it all together	78
Casting signed and unsigned numbers	56		
Creating user-defined types	56	Summary	80
Accessing signals using values with enumerated types	57	Questions	80
		Challenge	81
Packaging up code using functions	57	Further reading	81

3

Counting Button Presses

Technical requirements	83	Project 2 – Counting button presses	93
What is a sequential element?	84	Introducing the seven-segment display	93
Clocking your design	84	Detecting button presses	96
Looking at a basic register	85	What about simulation?	108
Registers in the Artix 7	90		

Deep dive on synchronization	109	Summary	112
Why use multiple clocks?	109	Questions	113
Two-stage synchronizer	109	Challenge	113
Synchronizing control signals	109		
Passing data	111	Further reading	114

4

Let's Build a Calculator

Technical requirements	116	Packaging for reuse	124
Implementing our first state machine	116	Coding the top level	126
Writing a purely sequential state machine	116	Investigating the divider	132
Splitting combination and sequential logic in a state machine	118		
Designing a calculator interface	119	Project 4 – Keeping cars in line	138
Designing a Moore state machine	120	Defining the state diagram	139
Implementing a Mealy state machine	122	Displaying our traffic lights	139
Practical state machine design	123		
Project 3 – Building a simple calculator	123	Summary	142
		Questions	142
		Challenge	143
		Extra challenge	143
		Further reading	143

5

FPGA Resources and How to Use Them

Technical requirements	146	Project 6 – Using the temperature sensor	161
Project 5 – Listening and learning	146	Handling the data	163
What is a PDM microphone?	146	Smoothing out the data	164
Simulating the microphone	150		
Introducing storage	152	Summary	171
Capturing audio data	158	Questions	172
		Further reading	173

6

Math, Parallelism, and Pipelined Design

Technical requirements	176	to a pipelined floating-point implementation	188
Introduction to fixed-point numbers	176	Fix to floating point conversion	189
Project 7 – Using fixed-point arithmetic in our temperature sensor	178	Floating-point math operations	191
		Float to fixed point conversion	193
		Simulation	194
Using fixed-point arithmetic to clean up the bring-up time	178	Parallel designs	195
Temperature conversion using fixed-point arithmetic	181	ML and AI and massive parallelism	195
What about floating-point numbers?	184	Parallel design – a quick example	196
A quick look at the AXI streaming interface	187	Summary	197
Project 8 – Updating the temperature sensor project		Questions	197
		Challenge	199
		Further reading	199

Section 3: Interfacing with External Components

7

Introduction to AXI

Technical requirements	204	AXI4 interfaces (full and AXI-Lite)	222
AXI streaming	204	Developing IPs – AXI-Lite, full, and streaming	225
Project 9 – creating IPs for Vivado using AXI streaming interfaces	205	Adding an unpackaged IP to the IP integrator	228
Seven-segment display streaming interface	205	Summary	230
Developing the ADT7420 IP	212	Questions	230
Understanding the <code>flt_temp</code> core IP integrator	212	Further reading	231

8

Lots of Data? MIG and DDR2

Technical requirements	234	Quad Data Rate (QDR) SRAM	257
Project 10 – introducing external memory	234	HyperRAM	257
Introduction to DDR2	236	SPI RAM	257
Generating a DDR2 controller using the Xilinx MIG	237	Summary	258
Modifying the design for use on the board	252	Questions	258
Other external memory types	257	Challenge	259
		Further reading	259

9

A Better Way to Display – VGA

Technical requirements	262	Testing the VGA controller	282
Project 11 – Introducing the VGA	262	Examining the constraints	283
Defining registers	266	Summary	285
Generating timing for the VGA	269	Questions	285
Displaying text	275	Challenge	286
		Further reading	286

10

Bringing It All Together

Technical requirements	288	Displaying the temperature sensor data	305
Investigating the keyboard interface	288	Displaying audio data	307
Project 12 – keyboard handling	294	Summary	311
Testing the PS/2	298	Questions	311
Project 13 – bringing it all together	301	Challenge	312
Displaying PS/2 keycodes on the VGA screen	302	Further reading	312

11

Advanced Topics

Technical requirements	313	Display enhancements	324
Exploring more advanced SystemVerilog constructs	314	A quick introduction to assertions	326
Interfacing components using the interface construct	314	Using \$error or \$fatal in synthesis	326
Using structures	317	Other gotchas and how to avoid them	327
Block labels	318	Inferring single bit wires	327
Looping using for loops	319	Bit width mismatches	328
Looping using do...while	320	Upgrading or downgrading	328
Exiting a loop using disable	321	Vivado messages	328
Skiping code using continue	321	Handling timing closure	330
Using constants	322	Summary	336
Exploring some more advanced verification constructs	322	Questions	337
Introducing SystemVerilog queues	322	Further reading	338
		Why subscribe?	339

Other Books You May Enjoy

Index

Preface

Prepare yourself for some fun. I have been designing ASICs and FPGAs for 30 years and every day brings new challenges and excitement as I push technology to develop new applications. Over the course of my career, I've developed ASICs that powered military aircraft, graphics that ran on high-end workstations and mainstream PCs, technology to power the next generation of software-defined radios, and supplied space-based internet to the globe. Now, I want to give some of that experience back to you.

Who this book is for

This book is for someone interested in learning about FPGA technology and how you might use it in your own projects. We assume you know nothing about digital logic and start by introducing basic gates and their functions and eventually develop full systems. A little programming or hardware knowledge is helpful but not necessary. If you can install software, plug in a USB cable, and follow the projects you will learn a lot.

What this book covers

Chapter 1, Introduction to FPGA Architectures and Xilinx Vivado, explains what an ASIC and an FPGA is, and how to install Xilinx Vivado and create a small design.

Chapter 2, Combinational Logic, looks at writing a complete SystemVerilog module from scratch to perform some basic operations to show how to use combinational logic in your own designs. We'll also introduce testbenches and how to write one that self-checks.

Chapter 3, Counting Button Presses, builds upon the previous chapter's combination logic, adding storage—sequential elements. We'll learn about the capabilities of the Artix-7 and other FPGA devices to store data and design a simple project to count button presses. We'll also take a look at using clocks and synchronization, one of the few things that can break a design completely if not done correctly.

Chapter 4, Let's Build a Calculator, looks at how, to create more complex designs, inevitably you need to keep track of the design state. In this chapter, we'll learn about state machines and use a classic staple of engineering, the traffic light controller. We'll also enhance our calculator and show how we can design a divider using a state-based design.

Chapter 5, FPGA Resources and How to Use Them, takes a step back after having quickly dived into FPGA designs, examining some of the FPGA resources in more detail. To use these resources, we'll introduce some of the board resources, the PDM microphone and i2c temperature sensor attached to the FPGA, and use them in projects.

Chapter 6, Math, Parallelism, and Pipelined Design, takes a deeper dive into fixed-point and floating-point numbers. We'll also look at pipelined designs and parallelism for performance.

Chapter 7, Introduction to AXI, covers how Xilinx has adopted the AXI standard to interface its IP and has developed a tool, IP integrator, to easily connect the IP graphically. In this chapter, we'll look at AXI by taking our temperature sensor and using the IP integrator to integrate the design.

Chapter 8, Lots of Data? MIG and DDR2, looks at how the Artix-7 provides a good amount of memory, but what happens if we need access to megabytes or gigabytes of temporary storage? Our board has DDR2 on it and in anticipation of implementing a display controller, we'll look at the Xilinx Memory Interface Generator to implement the DDR2 interface and test it in simulation and on the board.

Chapter 9, A Better Way to Display – VGA, looks at implementing a VGA and an easy way to display text. We've used LEDs and a seven-segment display to output information from our projects. This does limit us to what can be shown; for example, we can't display our captured audio data and text.

Chapter 10, Bringing It All Together, covers adding to our inputs. We've covered the output with VGA, but we'll add to our inputs by interfacing to the keyboard using PS/2. We'll take our temperature sensor and PDM microphone and create a project that uses the VGA to display this data.

Chapter 11, Advanced Topics, wraps things up by looking at some SystemVerilog concepts that I skipped over but you may still find useful. We'll look at some more advanced verification constructs and finally look at some other gotchas and how to avoid them.

To get the most out of this book

This book assumes no existing knowledge of FPGAs, logic design, or programming. You'll need a computer with Windows or Linux. You'll be guided to install the necessary software in the first chapter.

Software/Hardware covered in the book	OS Requirements
Xilinx Vivado 2020.1	Windows 10 or Linux (Centos 7.4-7.7 or Ubuntu 18.04 or 20.04)
Nexys A7 board	Windows 10 or Linux (Centos 7.4-7.7 or Ubuntu 18.04 or 20.04)

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Learn-FPGA-Programming>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: http://www.packtpub.com/sites/default/files/downloads/9781789805413_ColorImages.pdf

Conventions used

There are a number of text conventions used throughout this book.

Code in text: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "adt7420_i2c_bd.v provides the Verilog wrapper."

A block of code is set as follows:

```
always @(posedge CK) begin
    stage = D;
    Q      = stage;
end
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
module dff (input wire D, CK, output logic Q);
    initial Q = 1;
    always_ff @(posedge CK) Q <= D;
endmodule
```

Any command-line input or output is written as follows:

```
`timescale 1ps/100fs
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "In the block design, right-click and select **Add Module**."

Tips or important notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

Section 1: Introduction to FPGAs and Xilinx Architectures

In this section, you will get an understanding of what a **Field Programmable Gate Array (FPGA)** is, what the underlying technology is, and an introduction to the Artix-7 architecture.

This part of the book comprises the following chapter:

- *Chapter 1, Introduction to FPGA Architectures and Xilinx Vivado*

1

Introduction to FPGA Architectures and Xilinx Vivado

In the following chapter, we will be exploring **Field Programmable Gate Arrays (FPGAs)** and the underlying technology that creates them. This underlying technology allows companies such as Xilinx to produce a reprogrammable chip from an **Application Specific Integrated Circuit (ASIC)** process. We'll then learn how to use an FPGA for a simple task. Whether you want to accelerate mathematically complex operations such as machine learning or artificial intelligence, or simply want to do some projects for fun, such as retro computing or reproducing obsolete video game machines (https://github.com/MiSTER-devel/Main_MiSTER/wiki), this book will jumpstart your journey. There couldn't be a better time to get into this field than the present, even if only as a hobby. Development boards are cheap and plentiful, and vendors have started making their tools available for free for their low cost, smaller parts.

In this book, we are going to build some example designs to introduce you to FPGA development, culminating in a project that can drive a VGA monitor.

By the end of this chapter, you should have a good understanding of an FPGA and its components.

In this chapter, we are going to cover the following main topics:

- What is an ASIC?
- How does a company create an FPGA?
- What makes up an FPGA?
- How can we use the Xilinx Vivado tools to design, test, and implement an FPGA

Technical requirements

To follow along with the examples in this chapter, you need the following hardware and software.:

Hardware

Unlike programming languages, SystemVerilog is a hardware description language, and to really see the fruits of your work in this book, you will need an FPGA board to load your designs into. For the purposes of this book, I am suggesting one of two development boards, which are readily available. It is possible to target another board if you already have one. However, some of the resources may not be identical or you may need to change the constraints file (xdc) to access the resources that another board has:

- Information on the Nexys A7: <https://store.digilentinc.com/nexys-a7-fpga-trainer-board-recommended-for-ece-curriculum/>.
- Information on the Basys 3 Artix-7 FPGA trainer board: <https://store.digilentinc.com/basys-3-artix-7-fpga-trainer-board-recommended-for-introductory-users/>.

The Nexys A7 is preferable if possible because it has external interfaces that will be discussed in later chapters and will give you experience interfacing to external components. I would recommend the 100T in case you get ambitious and would like to explore more as the price difference is relatively small and it has twice the resources. With the exception of the DDR memory, the Basys 3 board can do most of the projects, although some may require the purchase of PMOD interface boards.

Software

You need the following software to follow along:

- <https://www.xilinx.com/products/design-tools/vivado.html>
- Code files for all the examples in this chapter can be found in this book's GitHub repository at <https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH1>.

What is an ASIC?

ASICs are the fundamental building blocks of modern electronics – your laptop or PC, TV, cell phone, digital watch, almost everything you use on a day-to-day basis. It is also the fundamental building block upon which the FPGA we will be looking at is built from. In short, an ASIC is a custom-built chip designed using the same language and methods we will be introducing in this book.

FPGAs came about as the technology to create ASICs followed Moore's law (*Gordon E. Moore, Cramming more components onto integrated circuits, Electronics, Volume 38, Number 8* (<https://newsroom.intel.com/wp-content/uploads/sites/11/2018/05/moores-law-electronics.pdf>) – the idea that the number of transistors on a chip doubles every 2 years. This has allowed for both very cheap electronics in the case of mass-produced items containing ASICs and led to the proliferation of lower cost FPGAs.

Why an ASIC or FPGA?

ASICs can be an inexpensive part when manufactured in high volumes. You can buy a disposable calculator, a flash drive for pennies per gigabyte, an inexpensive cell phone; they are all powered by at least one ASIC. ASICs are also sometimes a necessity when speed is of the utmost importance, or the amount of logic that is needed is very large. However, in these cases, they are typically only used when cost is not a factor.

We can break down the costs of developing a product based on an ASIC or FPGA into **Non-Recurring Engineering (NRE)**, the one-time cost to develop a chip, and the piece price for every chip, excluding NRE. Ed Sperling states the following in *CEO Outlook: It Gets Much Harder From Here, Semiconductor Engineering, June 3, 2019, https://semiengineering.com/ceo-outlook-the-easy-stuff-is-over/*:

"The NRE for a 7nm chip is \$25 million to \$30 million, including mask set and labor."

These costs include more than just the mask sets, the blueprint for the ASIC if you will, that is used to deposit the materials on the silicon wafers that build the chip. It's also the teams of design, implementation, and verification engineers that can number into the hundreds. Usually factored into ASIC costs are re-spins, which are bug fixes. These are a factor because large, complex devices struggle with first-time success.

Compare this to an FPGA. Fairly complex chips can be developed by a single person, or small teams. Most of the NRE has been shouldered by the FPGA vendor in the design of the FPGA chips, which are known good quantities. What little NRE is left is for tools and engineering. Re-spins are free, except for time, since the chip can be reprogrammed without million-dollar mask sets.

The trade-off is the per part cost. High volume ASICs with low complexity, like the one inside a pocket calculator or a digital watch, can cost pennies. CPUs can run into the hundreds or thousands of dollars. FPGAs, even the most inexpensive Spartan-7, start at a few dollars and the largest and fastest can stretch into the tens of thousands of dollars.

Another factor is tool costs. As we will see later in this chapter, Xilinx provides the Vivado tool suite for free in the form of a webpack for the smaller parts. This speeds adoption, where the barrier to entry is now a computer and a development board. Even the cost of developing more expensive parts is only a few thousand dollars if you need to purchase a professional copy of Vivado. ASIC tools can run into the millions of dollars and require years of training since the risk of failure is so high. As we will see in our projects, we'll make mistakes, sometimes to demonstrate a concept, but the cost to fix it will only be a few minutes of time, mostly to understand why it failed:



Figure 1.1 – Simple ASIC versus FPGA flow

The flow for an ASIC or FPGA is essentially the same. ASIC flows tend to be more linear, in that you have one chance to make a working part. With an FPGA, things such as simulation can become an option, although strongly suggested for complex designs. One difference is that the lab debug stage can also act as a form of simulation by using ChipScope, or similar on-chip debugging techniques, to monitor internal signals for debugging. The main difference is that each iteration through the steps costs only time in an FPGA flow. In this situation, any changes to a fabricated ASIC design requires some number of new mask sets that can run into the millions of dollars.

We've briefly looked at what an ASIC is and why we might choose an ASIC or an FPGA for a given application. Now, let's take a look at how an FPGA is created using an ASIC process.

How does a company create a programmable device using an ASIC process?

The basis of any ASIC technology is the transistor, with the largest devices holding up to a billion. There are multiple ASIC processes that have been developed over the years, and they all rely on 0s and 1s, in other words, on or off transistors. These on or off transistors can be thought of as Booleans, true or false values.

The basis of Boolean algebra was developed by George Bool in 1847. The fundamentals of Boolean algebra make up the basis of the logic gates upon which all digital logic is formed. The code that we will be writing will be at a fairly high level, but it is important to understand the basics and it will give us a good springboard for a first project.

Fundamental logic gates

There are four basic logic gates. We typically write the truth tables for the gates to understand their functionality. A truth table shows us what the output is for every set of inputs to a circuit. Refer to the following example involving a NOT gate.

Important note

In this section, we are primarily discussing only the logical functions. There are equivalent bitwise functions that will be introduced. Logical functions are typically used in `if` statements, bitwise functions in logic operations.

Assign statement

In SystemVerilog, we can use an `assign` statement to take whatever value is on the right-hand side of the equal sign to the left-hand side. Its usage is as follows:

```
assign out = in;
```

`in` can be another signal, function, or operation on multiple signals. `out` can be any valid signal declared prior to the `assign` statement.

Comments

SystemVerilog provides two ways of commenting. The first is using a double slash, `//`. This type of comment runs until the end of the line it's located on. The second type of comment is a block comment. Both are shown here:

```
// Everything here is a comment.  
/* I can span
```

```
Multiple
Lines */

```

if statement

SystemVerilog provides a way of testing conditions via the `if` statement. The basic syntax is as follows:

```
if (condition) event
```

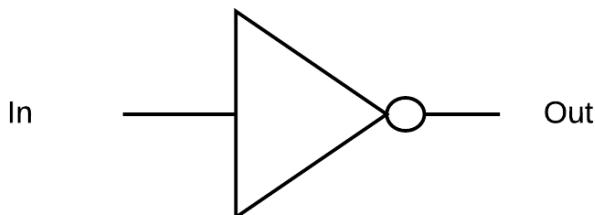
We will discuss `if` statements in more detail in *Chapter 2, Combinational Logic*.

Logical NOT (!)

The output of the NOT gate produces the inverse of the signal going in. The function in SystemVerilog can be written as follows:

```
assign out = !in; // logical or boolean operator
```

The associated truth table is as follows:



Graphical Representation

In	Out
0	1
1	0

Truth Table

Figure 1.2 – NOT gate representation

The NOT gate is one of the most common operators we will be using:

```
if (!empty) ...
```

Often, we need to test a signal before performing an operation. For example, if we are using a **First in First Out (FIFO)** storage to smooth out sporadic data or for crossing clock domains, we'll need to test whether there is data available before *popping* it out for use. FIFOs have flags used for flow control, the two most common being full and empty. We can do this by testing the empty flag, as shown previously.

We will go into greater depth in later chapters on how to design a FIFO as well as use one.

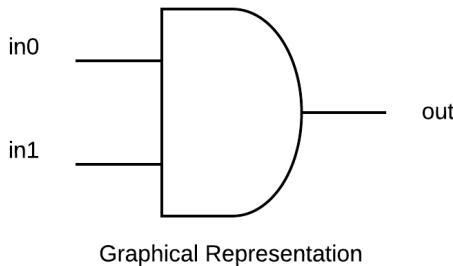
Logical AND (`&&`), bitwise AND (`&`)

Often, we will want to test whether one or more conditions are active at the same time. To do this, we will be using the AND gate.

The function in SystemVerilog can be written as follows:

```
assign out = in1 && in0; // logical or boolean operator
```

The associated truth table is as follows:



in0	in1	out
0	0	0
0	1	0
1	0	0
1	1	1

Truth Table

Figure 1.3 – AND gate representation

Continuing our FIFO example, you might be popping from one FIFO and pushing into another:

```
if (!src_fifo_empty && !dst_fifo_full) ...
```

In this case, you want to make sure that both the source FIFO has data (is not empty) and that the destination is not full. We can accomplish this by testing it via the `if` statement.

Logical OR (`||`), bitwise OR (`|`)

Another common occurrence is to check whether any one signal out of a group is set to perform an operation.

The function in SystemVerilog can be written as follows:

```
assign out = in1 || and in0; // logical or boolean operator
```

The associated truth table is as follows:

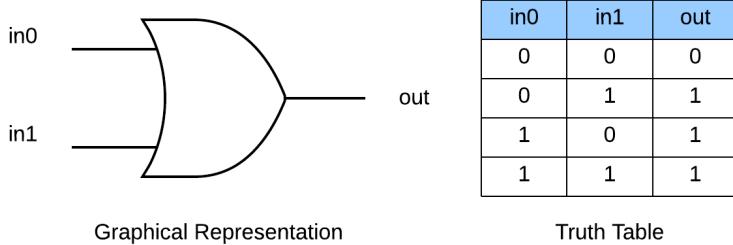


Figure 1.4 – OR gate representation

Next, we will look at the exclusive OR function.

XOR (^)

The exclusive OR function checks whether either one of two inputs is set, but not both. The function in SystemVerilog can be written as follows:

```
assign out = in1 ^ in0; // logical or boolean operator
```

The associated truth table is as follows:

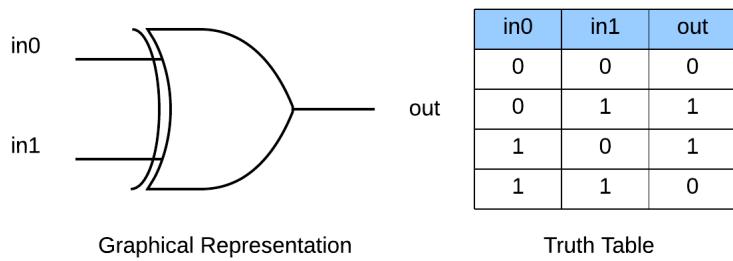


Figure 1.5 – XOR gate representation

This function is used in building adders, parity, and error correcting and checking codes. In the next section, we'll look at how an adder is built using the preceding gates.

More complex operations

We've looked at the basic components in the previous sections that make up every digital design. Here we'll look at an example of how we can put together multiple logic gates to perform work. For this we will introduce the concept of a full adder. A full adder takes three inputs, A, B, and carry in, and produces two outputs, sum and carry out. Let's look at the truth table:

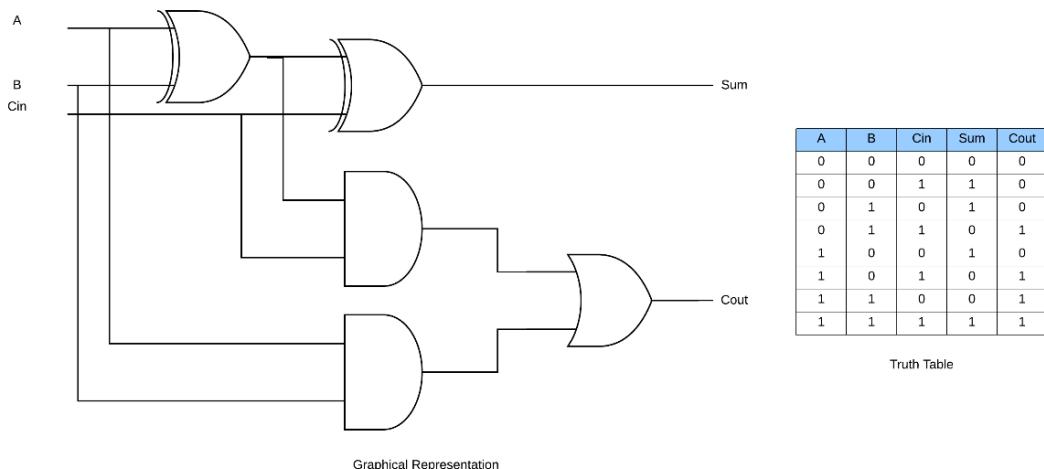


Figure 1.6 – Full adder

The SystemVerilog code for the full adder written as Boolean logic would be as follows:

```
assign Sum = A ^ B ^ Cin;
assign Cout = A & B | (A^B) & Cin;
```

You'll notice that we are using the bitwise operators for AND (`&`) and OR (`|`) since we are operating on bits. From this straightforward, yet important example, you can see that real-world functionality can be built from these basic building blocks. In fact, all the circuits in an ASIC or FPGA are built this way, but luckily you don't need to dive into this level of detail unless you really want to thanks to the proliferation of **High-Level Design Languages (HDLs)**, such as SystemVerilog.

Introducing FPGAs

A gate array in ASIC terms is a sea of gates with some number of mask steps that can be configured for a given application. This allows for a more inexpensive product since the company designing the ASIC only needs to pay for the masks necessary for configuring. The FPGA takes this one step further by providing the programmability of the fabric as part of the device. This does result in an increased cost as you are paying for interconnect you are not using and the storage devices necessary to configure the FPGA fabric, but allows for some cost reductions as these parts become standard devices that can be mass produced.

If we look at the functions in the previous section through the adder example, we can see one commonality; they can all be produced using a truth table. This becomes key in FPGA development. We can regard these truth tables as **Read Only Memory (ROM)** representations of the functions. In fact, we can regard them as **Programmable ROMs (PROMs)** in the case of building up our FPGA.

Let's take the example of the fundamental logic functions. We can reproduce any of them by utilizing a 2-input lookup table, which could look something like this:

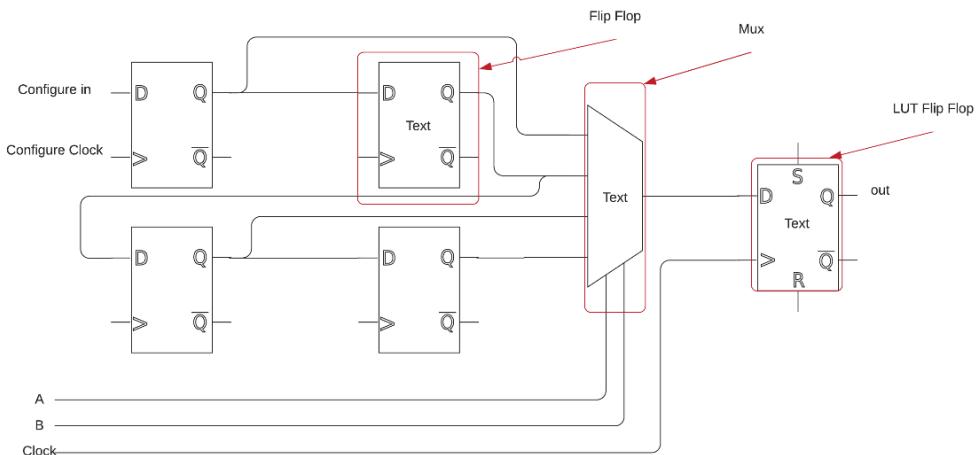


Figure 1.7 – Two input LUT examples

This is an oversimplified example, but what we have are four storage elements, in this case flip-flops, but in the case of an actual FPGA, more likely a much simpler structure utilizing far fewer transistors. The storage elements are connected to one another such that their configuration can be loaded. By attaching other **Lookup Tables (LUTs)** to the chain, multiple LUTs can be configured at startup, or in the case of partial reconfiguration, during normal operation. By adding a flip-flop, we can see the final structure of the LUT take shape.

The power in the simplicity of the structure is the ability to replicate this design many times over. In the case of modern FPGAs, they are built of many tiles or columns of logic such as this, allowing a much simpler piece to be designed, implemented, and verified, and then replicated to produce the large gate count devices available. This allows for a range of lower cost devices with fewer columns of resources to larger devices with many more, some even using **Stacked Silicon Interconnects (SSI)**, which allows multiple ASIC dies to be attached together via an interconnect substrate.

In 1985, Xilinx introduced the XC2064, what we would consider the first FPGA utilizing an array of 64 3-input LUTs with one flip-flop. The breakthrough with this design was that it was modular and had good interconnect resources. This entire part would be approximately equivalent to 1 **Combination Logic Block (CLB)** in the Artix-7 we will be targeting.

At the heart of an FPGA is the programmable fabric. The fabric consists of LUTs with associated flip-flops making up slices and ultimately CLBs. These blocks are all connected using a rich topology of routing channels, allowing for almost limitless configuration. FPGAs also contain many other resources that we will explore over the course of this book, block RAMs, **Serial-Deserial (SERDES)** cores, DSP elements, and many types of programmable I/O.

Exploring the Xilinx Artix-7 and 7 series devices

The FPGAs we will be looking at in this book are the Artix-7 series of devices. These devices are the highest performance per watt of the Xilinx 7 series devices. For a reasonable price, they feature a large amount of relatively high-performance logic to implement your designs. The FPGA components we will introduce here are common in the Spartan (low end), Kintex (mid-range) and Virtex (high end) parts in the 7 series.

Combinational logic blocks

ASICs are made up of logic gates based upon libraries provided by ASIC foundries, such as TSMC or Tower. These libraries can contain everything from AND, OR, and NOT gates to more complicated math cells and storage elements. When developing an FPGA, you will be targeting the same Boolean logic equations as you would in an ASIC. We will be using a very similar flow. However, the synthesis process will target the CLBs of the FPGA:



Figure 1.8 – Xilinx UG474 7 series FPGAs CLB users' guide figure 1-1 (used with permission)

A CLB consists of a pair of slices, each of which contains four 6-input LUTs and their eight flip-flops. Vivado (or optionally a third-party synthesis tool such as Synopsys Synplify) compiles the SystemVerilog code and maps it to these CLB elements. To fully explore the details of the CLB, I would suggest reading Xilinx UG474, 7 Series FPGAs CLB users' guide (https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf). At a high level, each LUT allows a degree of flexibility such that any Boolean function with 6 inputs can be implemented or two arbitrarily defined 5-input functions if they share common inputs. There is also dedicated high speed carry logic for arithmetic functions, which will be discussed in later chapters.

The slices come in two formats, SLICEL (logic) and SLICEM (memory). SLICEM is a superset of SLICEL. SLICEM adds the ability to configure the SLICE into a distributed RAM or shift register. There are approximately three times the number of SLICELs as SLICEMs. The following table for the two suggested development boards for this book shows the breakdown:

Board	Device	Slices	SLICEL	SLICEM	6-input LUTs	Distributed RAM (Kb)	Shift Register (Kb)	Flip-Flops
Basys 3	7A35T	5,200	3,600	1,600	20,800	400	200	41,600
Nexys A7	7A100T	15,850	11,100	4,750	63,400	1,188	594	126,800

Although it is theoretically possible to instantiate and force the functionality of lower-level components, such as slices or LUTs, this is beyond the scope of this book, and a feature not widely used. We will be targeting CLB usage through Vivado synthesis of the HDL that we write.

Storage

Aside from the SLICEMs that make up the CLBs that can be used as memories or shift registers, FPGAs contain **Block RAMs (BRAM)** that are larger storage elements. The 7 series parts all have 36 Kb BRAM that can be split into two 18 Kb BRAMs. The following table shows the BRAM available in the parts on the recommended development boards:

Board	Device	Total 36 Kb BRAM
Basys 3	7A35T	50
Nexys 7 A7	7A100T	135

BRAMs can be configured as follows:

- True dual port memories – Two read/write ports.
- Simple dual port memories – 1 read/1 write. In this case, a 36 Kb BRAM can be up to 72 bits wide and an 18 Kb BRAM up to 36 bits wide.
- A single port.

Contents of BRAMs can be loaded at initialization and configured via a file or initial block in the code. This can be useful for implementing ROMs or start up conditions.

BRAMs in 7 series devices also contain logic to implement FIFOs. This saves CLB resources and reduces synthesis overhead and potential timing problems in a design. We will go over FIFOs in a later chapter.

All 36 Kb BRAMs have dedicated **Error Correction Code (ECC)** functions. As this is something more related to high reliability applications, such as medical-, automotive-, or space-based, something we will not go into detail on in this book.

Clocking

7 series devices implement a rich clocking methodology, which can be explored in detail in UG472 7 Series FPGAs clocking resources user guide (https://www.xilinx.com/support/documentation/user_guides/ug472_7Series_Clocking.pdf). For most purposes, our discussion in the PLL section will give you everything you need to know; however, the referenced document will delve into far more detail.

I/Os

For the most part, we will limit ourselves to the I/Os supported by the two targeted development boards. In general, the 7 series devices handle a variety of interfaces from 3.3v CMOS/TTL to LVDS and memory interface types. The boards we are using will dictate the I/Os defined in our project files. For more information on all the supported types, you can reference the UG471 7 Series FPGAs SelectIO resources user guide.

DSP48E1

FPGAs have a large footprint in **Digital Signal Processing (DSP)** applications that use a lot of multipliers and, more specifically, **Multiply Accumulate (MAC)** functions. One of the first innovations in FPGAs was to include hard multipliers followed by DSP blocks that could implement MAC functions:

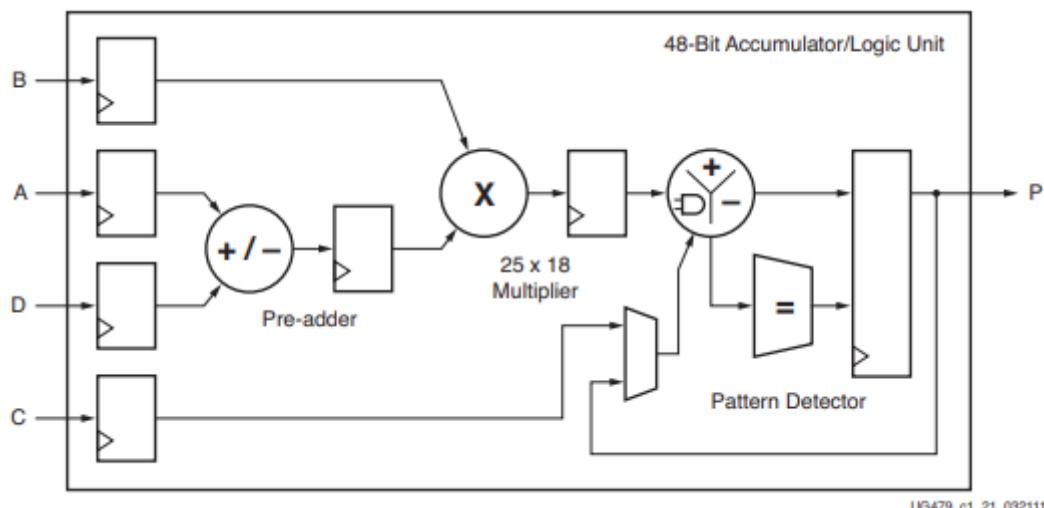


Figure 1.9 – Xilinx UG479 7 series DSP48E1 users' guide figure 1-1 (used with permission)

One of the most expensive operations in an FPGA is arithmetic. In an ASIC, the largest and slowest operation is typically a multiplication operation, and the smaller or faster operation is an add operation. For this reason, for many years, FPGA manufacturers have been implementing hard arithmetic cores in their fabric. This makes the opposite true in an FPGA, where the slower operation is typically an adder, especially as the widths get larger. The reason for this is that the multiply has been hardened into a complex, pipelined operation. We will explore the DSP operator more in later chapters. The UG479 7 Series DSP48E1 user guide (https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf) is a good reference if you are interested in delving into the details.

ASMBL architecture

The 7 series devices are the fourth generation where Xilinx has used the **Advanced Silicon Modular Block (ASMBL)** architecture for implementation purposes. The idea behind this is to enable FPGA platforms optimized for different target applications. Looking at the 7 series families, we can see how different configurations of slices are brought together to achieve these goals. We can see how the pieces we covered in this chapter are arranged as columns to give us the resources we will be using for our example projects ahead:



Figure 1.10 – Xilinx UG474 7 series FPGAs CLB users' guide figure 2-1 (used with permission)

Now that we have looked at what makes up the Artix-7 and other 7 series, we need to get the Xilinx tools installed so that we can get to our first project.

Introduction to the Vivado toolset and evaluation boards

In this section, we will explore the evaluation boards recommended for the projects in this book. We will walk through a very simple design using Vivado to introduce the tool and show how to program the board and demonstrate the functionality of the FPGA.

Evaluation boards

There is no shortage of FPGA evaluation boards available for us to purchase. One company that makes very affordable boards is Digilent. There are several nice features that their boards tend to include, but one of the best is that they have a USB to UART controller built in that Xilinx Vivado recognizes as a programming cable. This makes configuring the device painless. The recommended boards also have the added advantage of being powered over this same USB cable.

Nexys A7 100T (or 50T)

The Nexys A7 is the recommended board for this book. It has all the devices we'll target over the course of the book:

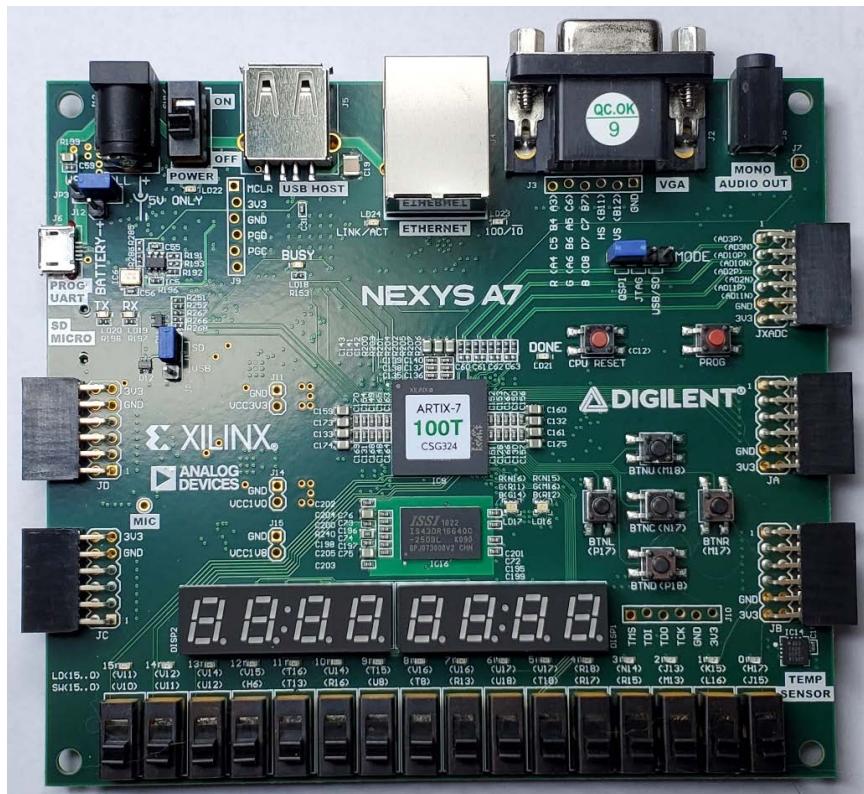


Figure 1.11 – Digilent Nexys A7 board

The board features are as follows:

- Artix-7 XC7A100T or 50T
- 450+MHz operation
- 128 MB DDR2
- Serial Flash
- Built-in USB UART for downloading images and ChipScope debugging
- MicroSD card reader
- 10/100 Ethernet PHY

- PWM audio output/microphone input
- Temperature sensor
- 3 axis accelerometer
- 16 switches
- 16 LEDs
- 5 pushbuttons
- Two 3 color LED
- Two 4-digit 7-segment displays
- USB host device support
- Five PMOD (one XADC)

Let's take a look at the breakdown of the two devices the Nexys board can be ordered with:

Device	XC7A100T-1CSG324C	XC7A50T-1CSG324C
Logic Slices	15,850	8,150
BRAM (Kbits)	4,860	2,700
Clock Management tiles	6	5
DSP	240	120

One benefit to choosing the XC7A100T is the additional RAM. Especially when starting out you may find yourself relying on chip debugging using ChipScope and the additional RAM will allow for additional storage for wider busses or longer capture times. We'll discuss ChipScope in a later chapter.

Basys 3

An alternative evaluation board is the Basys 3:



Figure 1.12 – Digilent Basys 3 board

This board has the same pushbuttons, LEDs, and switches, but only half the number of seven segment displays. We'll be developing code that can run on either board using these features. It does lack the DDR2 RAM, so it will limit using this for a framebuffer as we will introduce in a later chapter. It is also missing the temperature sensor, microphone, and audio, which we'll look at regarding serial interfaces. PMOD boards can be purchased that have this functionality, however, to overcome this limitation.

The board features are as follows:

- Artix-7 XC7A35T
- 450+Mhz operation
- 128MB DDR2
- Serial Flash

- Built-in USB UART for downloading images and ChipScope debugging
- MicroSD card reader
- 10/100 Ethernet PHY
- PWM audio output/microphone input
- 16 switches
- 16 LEDs
- 5 pushbuttons
- Two 3-color LEDs
- Single 4-digit, 7-segment displays
- USB host device support
- Four PMODs (one dual purpose supporting XADC)

Let's now take a look at the breakdown of the Basys 3 board:

Device	XC7A35T-1CSG324C
Logic slices	5,200
BRAM (Kbits)	1,800
Clock management tiles	5
DSP	90

Important note

The Basys 3 board lacks the DDR 2 memory, accelerometers, and audio capabilities, which will be addressed in later chapters. PMODs are available for everything apart from the DDR2. I would recommend the Nexys A7 over the Basys if possible.

We've just taken a look at the boards we are planning on using for the book. Now we need to take a look at the Xilinx tool, Vivado, which will be what we use to design, simulate, implement, and debug our FPGA designs.

Introducing Vivado

Once you have selected a board, the best way to get to know it is to work through an example design.

Vivado is the Xilinx tool we will be using to implement, test, download, and debug our designs. It can be run as a command-line tool in non-project mode, or in project mode using the GUI. For our purposes, we will be using project mode via the GUI; however, we will go through non-project mode as an introduction in the Appendix.

Vivado installation

Xilinx makes Vivado freely available in the form of a webpack for smaller devices. The webpack contains all the features of the full version with a limitation based on device support. It is available for either Windows or Linux. This book will show screenshots for the Linux version; however, everything is tested on both, so you will be fine with using either.

Important note

The Xilinx webpack forces tool feedback information to Xilinx. The paid version allows this to be disabled.

Perform the following steps to effect installation:

1. Create an account at <https://www.xilinx.com/>.
2. Visit <https://www.xilinx.com/support/download.html>.
3. Download the Xilinx Unified Installer. For this book, we'll be using version 2020.1.
4. On Windows, run the .exe file.

On Linux, use the following commands:

```
chmod +x Xilinx_Unified_2020.1_0602_1208_Lin64.bin; ./  
Xilinx_Unified_2020.1_0602_1208_Lin64.bin
```

5. Enter your account information for the installation.
6. When prompted, you can install either Vitis or Vivado. We will not be using Vitis, but it includes Vivado, so if you are **adventurous** and want to try Vitis out, feel free to install this as well.

7. When prompted for the devices, you only need **the 7 Series**.
8. Pick an installation location or use the default option.

Once you've completed these steps, get a cup of coffee... take a nap... write a book. It is going to take a while.

Directory structure

With Vivado installed, we can now walk through a very simple project to introduce you to Vivado and to make sure everything is set up correctly. The directory structure I like to use looks like the following:

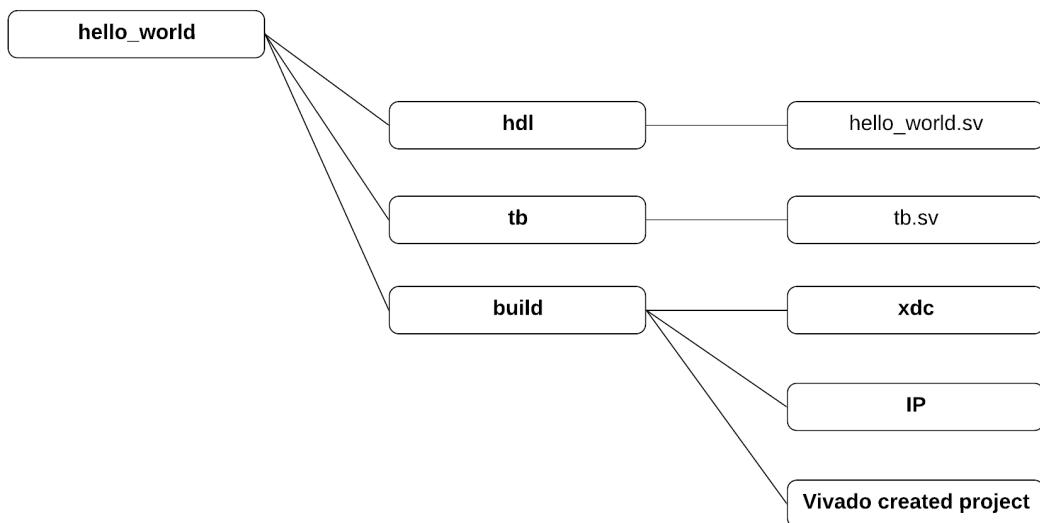


Figure 1.13 – Directory structure

Items in bold are directories. For our first example design, we do not have a lot of code. We will end up creating only three files: the HDL source code, the testbench, and a constraints file.

Inside the **hdl** directory, we'll create a simple design, **logic_ex.sv**, to run through Vivado:

Logic_ex.sv

```

`timescale 1ns/10ps
module logic_ex
(

```

```
input wire [1:0]      SW,
output logic [3:0]    LED
);
assign LED[0]  = !SW[0];
assign LED[1]  = SW[1] && SW[0];
assign LED[2]  = SW[1] || SW[0];
assign LED[3]  = SW[1] ^ SW[0];
endmodule // logic_ex
```

First, we'll define the timescale that we will be operating at in the simulator. 1ns/10ps was pretty standard years ago and for what we'll be doing, it will work fine. If you get involved using high-speed transceivers, you may encounter even smaller timescales, such as 1ps/1fs.

Tip

Each module should reside in its own file and the file should be named the same as the module. This can make life easier when using some tools, such as commercial simulators or even custom scripting.

The syntax for defining the timescale is as follows:

```
`timescale <time unit>/<time precision>
```

time unit defines the value and unit of delays. time precision specifies the rounding precision. This value can usually be overridden in the simulator and these settings have no effect on synthesis. When using `timescale, it is best to set it in all files:

String	Unit of time
s	Seconds
ms	Milliseconds
us	Microseconds
ns	Nanoseconds
ps	Picoseconds
fs	Femtoseconds

We define a port list with one input, SW, which is a 2-bit value that we will connect to the two right-most switches on the board. We also define one output named LED, which are four bits that represent the four LEDs above the four right-most switches:

tb.sv

```
`timescale 1ns/ 100ps;
module tb;
    logic [1:0] SW;
    logic [3:0] LED;
    logic_ex u_logic_ex (*.*);
    //logic_ex u_logic_ex (.SW, .LED);
    //logic_ex u_logic_ex (.SW(switch_sig), .LED(led_sig));
    //logic_ex u_logic_ex (*.*, .LED(led_sig));
```

Here we declare a top-level module called tb. Note that the top-level testbench module should not have any ports. We also declare two logic types that we will hook up to the hello world module.

Here, we instantiate `logic_ex` as an instance, `u_logic_ex`. There are multiple ways of connecting ports. In the uncommented example, we are using `.*`, which will connect all ports with the same name as a defined signal in the instantiating module.

The second example (commented out) uses `.<name>` of the port you wish to connect. It requires the port name to already be defined.

Finally, if there is a signal with a different named signal, we could use the third example, which allows port renaming. It is possible to mix `.*` with renamed ports, as shown in the final example.

A testbench typically has two distinct parts, the stimulus generator and stimulus checker:

```
// Stimulus
initial begin
    $printtimescale(tb);
    SW = '0;
    for (int i = 0; i < 4; i++) begin
        $display("Setting switches to %2b", i[1:0]);
        SW = i[1:0];
        #100;
    end
```

```
$display("PASS: logic_ex test PASSED!");  
$stop;  
end
```

The stimulus block is simple because the design we are testing is simple. We can nest it completely in an initial block. When the simulator starts up, the `initial` block runs serially. First it will print the timescale used in `tb.sv`. Then, SW input into the `logic_ex` module is set to 0. Using a '`0`' in the assignment to `SW` tells the tool to set all bits to 0. There is also an equivalent '`1`', which sets all bits to 1 or '`z`', which would set all bits to `z`. Verilog sizing rules say that assigning `SW = 0` is equivalent to `SW = 32'b0`, which would result in a sizing warning. To limit warnings, using '`0`', '`1`', or '`z`' is preferable.

Important note

`SystemVerilog` is an HDL and this is an important distinction. An HDL must be able to model parallel operations since many or all the slices in an FPGA will be running in parallel all the time. `SW = '0;` is a blocking assignment. So, the assignment is made before moving on. We will discuss blocking versus non-blocking when we discuss clocked processes.

The stimulus block then loops four times via the `for` loop. `SystemVerilog` has the capability of declaring the loop variable within the `for` loop, in this case `i`. It is highly advisable to declare it this way to avoid multiple driven net warnings if you are using the same signal in multiple `for` loops.

Within the `for` loop, we print out the current setting of the switches using the system task `$display`. Since we want to display only the 2 bits we are incrementing without leading 0s, we specify `2%b`. We then set the value of `SW` to the lower two bits of `i`. Although we don't need to, we add in a delay of 100ns by using `#100`.

We are also using `$stop`, which will terminate the simulation run when reached.

Important note

We know that the delay is in ns because of the timescale we define in the test.

We also declare a checker block. In any good testbench, the checker block should be self-checking. This means that at the end of the test, we should be able to print whether the test passed or failed, and, if it failed, why. This also means that writing a testbench can often be as involved or even more involved than writing the code for the FPGA implementation. This is beyond the scope of this book, however. All commercial simulators, including the Vivado simulator, also support Universal Verification Methodology, which is a set of `SystemVerilog` classes and functions specifically for testing HDL designs:

```

always @(SW, LED) begin
    if (!SW[0] != LED[0]) begin
        $display("FAIL: NOT Gate mismatch");
        $stop;
    end
    if (&SW[1:0] != LED[1]) begin
        $display("FAIL: AND Gate mismatch");
        $stop;
    end
    if (|SW[1:0] != LED[2]) begin
        $display("FAIL: OR Gate mismatch");
        $stop;
    end
    if (^SW[1:0] != LED[3]) begin
        $display("FAIL: XOR Gate mismatch");
        $stop;
    end
end
endmodule

```

Conversely to the stimulus generation, we want this block to react to events from our design. We accomplish this by using an `always` block, which is sensitive just to changes on the SW inputs and LED outputs of the design. This is a simple case where we are matching each LED to the corresponding SW values run through their respective expected logic gate. We do this by using `!=`, which is not equals, but takes x's into account in case there is a bug in the design. We will see more complex testbenches in later chapters.

We are also using the reduction operators, `&`, `|`, and `^`, which are applied to the two bits of SW. `&SW[1:0]` is equivalent to `SW[0] & SW[1]`.

Running the example

You will want to copy the files for this book from GitHub at this point or clone the repository.

Loading the design

Let's load the design into Vivado:

1. Under Windows, locate the Vivado installation and double-click on the Vivado icon. Under Linux, the procedure is as follows:

```
Source <Vivado Install>/settings.sh (or.csh)  
Vivado
```

2. Perform *steps 2 and 3* the first time you run Vivado.
3. Open **Xhub Stores**:



Figure 1.14 – Xhub Stores

The Xilinx Xhub Stores are a convenient way of adding scripts, board files, and example designs to your Vivado installation.

4. Install the board files for the example projects.

Select the **Boards** tab, and then navigate to the Digilent Artix A7 100T or 35T and the Basys 3. You'll notice that there are quite a few commercial boards that easily make their files available for installation:



Figure 1.15 – Adding the Digilent boards

5. Select the open project and navigate to CH1/build/logic_ex/logic_ex.prj for the Nexys A7 board, or CH1/build/logic_ex/logic_ex_basys3.prj for the Basys 3 board, as shown in the following screenshot:

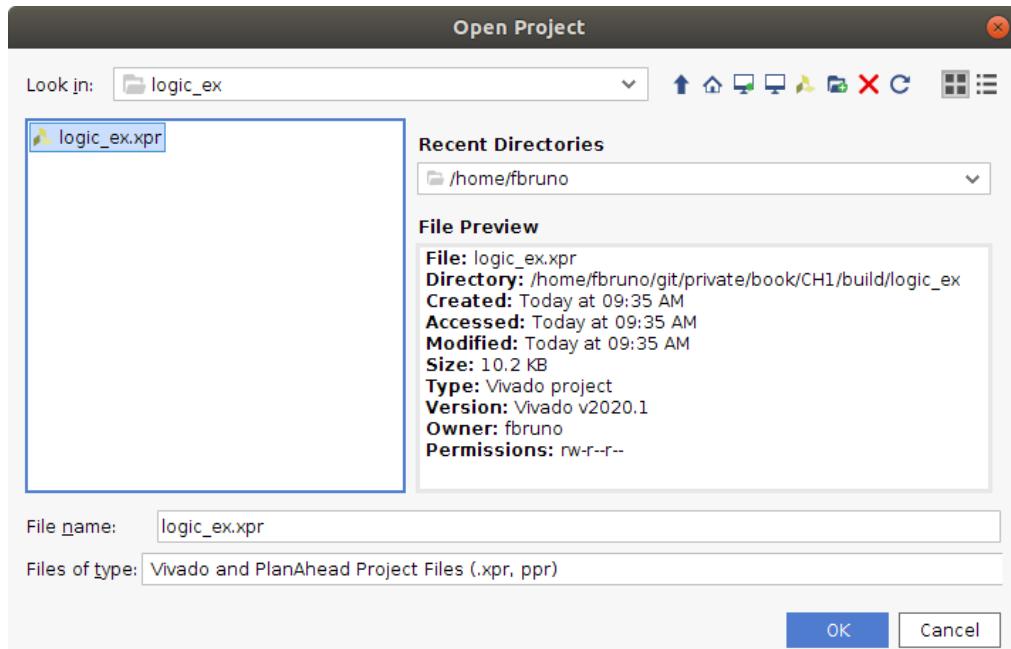


Figure 1.16 – Open Project window

Once open, you'll see the following:



Figure 1.17 – Vivado main screen for the logic_ex project

The Vivado project window gives us easy access to the design flow and all the information relating to the design. On the left-hand side, we see **Flow Navigator**. This gives us all the steps we will use to test and build our FPGA image. Currently, **PROJECT MANAGER** is highlighted. This gives us easy access to the sources in the design and the project summary. The project summary should be empty since we have loaded the project for the first time. On future loads of the project, it will display the information from the previous run.

Important note

To give you a jumpstart, all the projects in this book come complete with pre-set-up project files. Please see the appendix for instructions on setting up the first project in both project mode and non-project mode. This will guide you for setting up your own projects in the future.

Let's explore the sources in the design:



Figure 1.18 – Design sources

Here we can see our design, `logic_ex.sv`. We also have a set of constraints and we can see the testbench, `tb.sv`, instantiating `logic_ex.sv` under simulation sources. You can double-click on any of the files and explore them in the context-sensitive editor built into Vivado. The project is currently set up to reference the files in their current location within the directory structure, so the file can be edited with whatever your favorite editor is.

Looking at **Project Summary**, we can see the project is currently targeting the Arty A7-100 board.

Running a simulation

First, let's run the Vivado simulator to check the validity of our design.

To do this, click **Run Simulation | Run Behavioral Simulation** under **PROJECT MANAGER**. You will see that there are some other options available that are grayed out. These options allow you to run post synthesis or post implementation with or without timing. Behavioral simulation is relatively quick and will accurately represent the function of your design if the code is written properly. I would recommend not running post synthesis or implementation simulation unless you are debugging a board failure and need to accurately test the implemented version of the design as you'll find that the simulations will slow down dramatically.

Running the behavioral simulation will elaborate the design, the first step in the overall flow. The simulation view will take over the Vivado main screen:

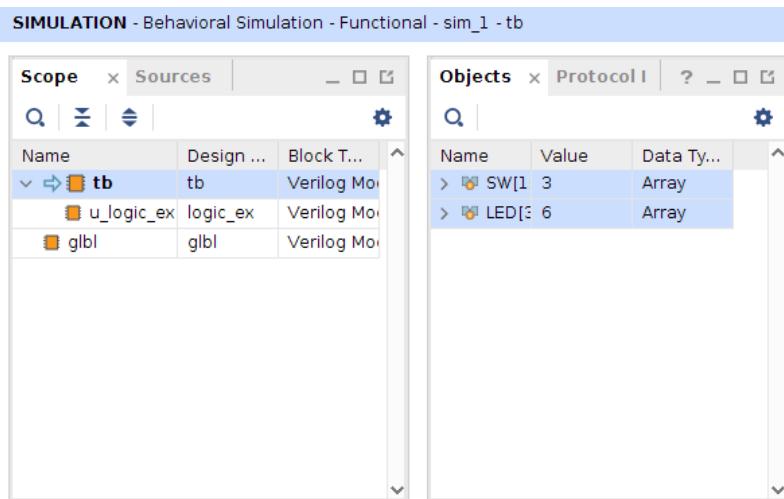


Figure 1.19 – Simulation view

The **Scope** screen gives us access to the objects within a given module. In this case, within the testbench (`tb`), we can see two signals, `SW[1:0]` and `LED[3:0]`. I've added them to the waves and expanded the view:



Figure 1.20 – Wave view

The wave view allows us to look at the signals in the design and how they are behaving as the simulation progresses. This will be the most widely used feature of the simulator when debugging problems. We can see the `SW` signal incrementing due to the `for` loop in the testbench. Correspondingly, we see the `LED` values change. The current display is in hex, but it is possible to change it to binary or, by clicking on the `>` symbol to the right of the signal, to display the individual bits of the signal. Also notice that each change in the signals corresponds to a 100ns time advance. This is due to the `#100` we are using to advance time and the timescale setting.

The final window is the most important for a self-checking testbench:

```

Tcl Console  x Messages  Log
[Search] [New] [Close] [Minimize] [Maximize] [Normal] [Zoom In] [Zoom Out] [Reset]
Time resolution is 1 ps
source tb.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#     send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a waveform"
#   }
# }
# run 1000ns
Timescale of (tb) is 1ns/100ps.
Setting switches to 00
Setting switches to 01
Setting switches to 01
Setting switches to 10
Setting switches to 11
PASS: logic_ex test PASSED!
$stop called at time : 400 ns : File "/home/fbruno/git/private/book/CH1/tb/tb.sv" Line 20
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:06 ; elapsed = 00:00:06 . Memory (MB): peak = 8121.176 ; g
current_wave_config {}
WARNING: [Wavedata 42-16] Error Unable to get wave configuration ''.
Untitled 4
add_wave {{/tb/SW}} {{/tb/LED}}

```

Type a Tcl command here

Figure 1.21 – Tcl Console

The Tcl console will display all the outputs from \$display, or assertions in the design. In this case, we can see the output from our \$printtimescale(tb) function as 1ns / 100ps. We also see the values that the switches are set to and can see within the waves the same values. Finally, we see **PASS: logic_ex test PASSED!**, giving us the result of the test. I would encourage you to experiment with the testbench. Change the operators or invert them to verify that the test fails if you do. This exercise will give you confidence that the testbench is functioning correctly.

The goal of verification is not to ensure the design passes; it is to try to make it fail. This is a simple case, so it is not really possible, but make sure that you test unexpected situations to make sure your design is robust.

Tip

It is advisable to adopt a convention in how you indicate tests passing and failing. This test is simple. However, a much more robust test suite for an actual design may have random stimulus and many targeted tests. Adopting a convention such as displaying the words **PASS** and **FAIL** allows for easy post-processing of test results.

Implementation

Now that we have confidence that the design works as intended, it is time to build it and test it on the board.

First, let's look at the .xdc file. Click back on **Project Manager** in **Flow Navigator**, and then expand the constraints and double-click on the xdc file.

The following lines should be uncommented out for the Arty A7-100T to set the configuration voltages:

```
set_property CFGBVS VCCO [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]
```

`set_property` is the `tcl` command, which will set a given design property used by Vivado. In the preceding command, we are setting `CFGBVS` and `CONFIG_VOLTAGE` to the values required by the Artix-7 FPGA.

The following code block sets up the switch and LED locations (placed together for convenience):

```
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 }
[get_ports { SW[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 }
[get_ports { SW[1] }]; #IO_L3N_T0_DQS_EMCLLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 }
[get_ports { LED[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 }
[get_ports { LED[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 }
[get_ports { LED[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 }
[get_ports { LED[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
```

The `set_property` commands create a `tcl` dictionary (`-dict`) containing `PACKAGE_PIN` and `IOSTANDARD` for each port on the design. We use the `get_port` TCL command to return a port on the design. `#` is a comment in the `TCL`.

The pin locations and I/O standards are defined by the board manufacturer. They have used 3.3 V I/Os and the pins are as specified.

The steps to generate a bitstream are as follows:

1. **Synthesis:** Map SystemVerilog to an intermediate logic format for optimizing.

2. **Implementation:** Place the design, optimize the place results, and route the design.
3. **Generate bitstream:** Generate the physical file to download to the board.

These can be run individually. You might take this route if you need to look at the intermediate results to see how the area or timing is coming out, or if you are designing a custom board and need to do pin planning. In our case, we can click directly on **Generate Bitstream** and allow it to run all the steps automatically for us. Allow it to use the defaults. When complete, open the implemented results:

The screenshot shows the Vivado Project Summary window for a project named "tb.sv" using "Nexys-A7-100T-Master.xdc".

Project Summary

- Overview** | Dashboard
- Project part: Arty A7-100 (xc7a100tcsg324-1)
- Top module name: logic_ex
- Target language: Verilog
- Simulator language: Mixed

Board Part

- Display name: Arty A7-100
- Board part name: [digilentinc.com:arty-a7-100:part0:1.0](#)
- Board revision: E.0
- Connectors: No connections
- Repository path: [/home/fbruno/Xilinx/Vivado/2020.1/xhub/board_store/xilinx_board_store](#)
- URL: [www.digilentinc.com/Arty-A7-100](#)
- Board overview: Arty A7-100

Synthesis

Status: ✓ Complete	Messages: No errors or warnings	Status: ✓ Complete	Messages: ! 3 warnings
Part: xc7a100tcsg324-1	Strategy: Vivado Synthesis Defaults	Part: xc7a100tcsg324-1	Strategy: Vivado Implementation Defaults
Report Strategy: Vivado Synthesis Default Reports	Incremental synthesis: None	Report Strategy: Vivado Implementation Default Reports	Incremental implementation: None

Implementation

Status: ✓ Complete	Messages: ! 3 warnings
Part: xc7a100tcsg324-1	Strategy: Vivado Implementation Defaults
Report Strategy: Vivado Implementation Default Reports	Incremental implementation: None

DRC Violations

No DRC violations were found.

[Implemented DRC Report](#)

Timing

Worst Negative Slack (WNS): NA
Total Negative Slack (TNS): NA
Number of Failing Endpoints: NA
Total Number of Endpoints: NA

[Implemented Timing Report](#)

Utilization

Post-Synthesis Post-Implementation			
Graph Table			
Resource	Utilization	Available	Utilization %
LUT	2	63400	0.01
IO	6	210	2.86

Power

Total On-Chip Power: 3.52 W
Junction Temperature: 41.1 °C
Thermal Margin: 43.9 °C (9.5 W)
Effective θJA: 4.6 °C/W
Power supplied to off-chip devices: 0 W
Confidence level: Low

[Implemented Power Report](#)

Figure 1.22 – Project Summary

Here we can see the summary of our implementation. We are using 2 LUTs and 6 I/Os (SW + LED). There is no timing since this design is purely combinational, otherwise we'd see more information regarding timing numbers.

If we click the **Device** tab, we can get a picture of how the device is being used:



Figure 1.23 – Device view

Here we can see the little white dot midway down the left-hand side. This represents where the LUTs are being placed.

Program the board

You have made it to the end of the chapter and now it's time to see the board in action:

1. Make sure it is plugged in and turned on.
2. Now, click on **Open hardware manager**, the last option under **Flow Navigator**. The hardware manager view will open in the main window.
3. Click **Open target | Autoconnect**.
4. Now, select the program device. The bitstream should be selected automatically. The lights will go out on the board for a few seconds and then, if the left two switches are down, you will be greeted with this:



Figure 1.24 – Board bringup

5. Flip the switches, and go through 00, 01, 10, 11, where 0 is down, 1 is up. Do the lights match the simulation? Do they match what you think they should be? Do you occasionally see one flicker as the switches are flipped? The last question will be answered in *Chapter 3, Counting Button Presses*.

Congratulations! You've completed your first project on an FPGA board. You've taken the first step on this journey and reconfigured the hardware in the FPGA to do some simple tasks. As we go through the book, the tasks will become more complex and more interesting and soon you'll be able to build upon this to create your own projects.

Summary

In this chapter, we've learned the basics of ASICs and FPGAs, how they are built, and when they make monetary sense. We've learned to use an FPGA board and program it. This sets us up for the rest of the book, where we will use this board and our programming skills in a variety of tasks and projects. Ultimately, these skills will be the foundation for developing your own designs, be they for work or for play.

The next chapter will build upon our example design as we delve further into combinational logic design.

Questions

1. When might you use an FPGA?
 - a) You are prototyping an application that may eventually be an ASIC.
 - b) You will only have very small volumes.
 - c) You need something that you can easily change the algorithms on in the future.
 - d) All of the above.
2. When would you use an ASIC?
 - a) You are developing a very specialized application, with just a small number to be built and the budget is tight.
 - b) You've been asked to design a calculator that will be mass produced and that requires a custom processor.
 - c) You need something extremely low power and cost is not a consideration.
 - d) You are developing an imaging satellite and want the ability to update the algorithms over the lifetime of the satellite.
 - e) a and b.
3. We have seen a full adder in the chapter. A half adder is a circuit that can add two inputs, in other words, no carry in. Can you write the truth table for the sum and carry for a half adder?

A	B	Sum	Carry
0	0		
0	1		
1	0		
1	1		

4. Modify the code and testbench to test the following gates: NAND (not AND), NOR (not OR), and XNOR (not XOR). Hint: You can invert a unary operator by adding a ~ operator in front of it, in other words, NAND is `~&`. Try it on the board.

Challenge

1. Open CH1/build/challenge.prj.
2. Modify the lines in challenge.sv to implement a full adder:

```
assign LED[0] = ; // Write the code for the Sum
assign LED[1] = ; // Write the code for the Carry
```

3. Modify tb_challenge.sv to test it:

```
if () begin // Modify for checking
```

Hint: You may want to jump ahead in the book to look at addition or do a quick web search.

Further reading

Please refer to the following links for more information:

- 7 Series FPGAs Configurable Logic Block: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf
- 7 Series FPGAs Clocking Resources: https://www.xilinx.com/support/documentation/user_guides/ug472_7Series_Clocking.pdf
- 7 Series DSP48E1 Slice: https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf
- Nexys A7 Reference Manual: <https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/reference-manual>
- Basys 3 Reference Manual: <https://reference.digilentinc.com/reference/programmable-logic/basys-3/reference-manual>

Section 2: Introduction to Verilog RTL Design, Simulation, and Implementation

In this section, you will learn to program SystemVerilog up to an intermediate level. You will design, test, and implement multiple designs from specification through to implementation on an actual board (if you desire).

This part of the book comprises the following chapters:

- *Chapter 2, Combinational Logic*
- *Chapter 3, Counting Button Presses*
- *Chapter 4, Let's Build a Calculator*
- *Chapter 5, FPGA Resources and How to Use Them*
- *Chapter 6, Math, Parallelism, and Pipelined Design*

2

Combinational Logic

Designs are typically composed of combinational and sequential logic. Combinational logic is made up simply of gates, as we saw in *Chapter 1, Introduction to FPGA Architectures and Xilinx Vivado*. Sequential logic maintains state, usually based on a clock edge, but it can be level-based as well, as we will discuss when we learn what not to do when inferring sequential logic.

In this chapter, we are going to explore writing a complete SystemVerilog module from scratch that can perform some basic real-world operations that you may use one day in your actual designs.

In this chapter, we are going to cover the following main topics:

- Creating SystemVerilog modules
- Introducing data types
- Packaging up code using functions
- Project – creating combinational logic

Technical requirements

The technical requirements for this chapter are the same as those for *Chapter 1, Introduction to FPGA Architectures and Xilinx Vivado*.

To follow along with the examples and the project, please take a look at the code files for this chapter at the following GitHub repository: <https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH2>.

Creating SystemVerilog modules

At the heart of every design are the modules that compose it. From the testbench that's used to verify the design to any instantiated components, they are all declared somewhere as a module. For the example design we'll be covering in this chapter, we'll be creating a set of modules representing the functions that we can access via the buttons and switches on the evaluation board. We'll use these switches to set values, and we'll use five buttons to perform operations.

Let's take a look at the components of a module declaration:

```
module project_2
#(parameter SELECTOR,
    Parameter BITS = 16)
(input wire [BITS-1:0]           SW,
 input wire                      BTNC,
 input wire                      BTNU,
 input wire                      BTNL,
 input wire                      BTNR,
 input wire                      BTND,
 output logic signed [BITS-1:0]);
```

We are creating a module called `project_2`, which will be the top level of our design. The first section within `#()` is the parameter list, which allows us to define parameters that we can use within the port list or module. We can also define parameters anywhere within the module, and they can also be overridden during instantiation. However, parameters must be defined prior to use.

How to create reusable code – parameters

Parameters can be used to override information in a module's instantiation. This information can be used within the module to control the size of the data, as is the case with BITS, which has a default value of 16 if it's not overridden. Parameters can also control the instantiation of logic or modules, as we'll see when we explore the case statement. We can also create a parameter, SELECTOR, which has no default. This is a good way to make sure that something is set in the instantiation since there is no default. If it is not overridden, it will result in an error.

Parameters can be integers, strings, or even types:

```
#(parameter type SW_T = logic unsigned [15:0], ...
  (input   SW_T      SW, ...)
```

Here, we created a type, SW_T, that defaults to logic unsigned [15:0] and creates a port using this type, SW. When the module is instantiated, a new type can be passed, thus overriding the default and allowing for greater design reuse.

Tip

It is good practice to keep parameters intended to be overridden within the parameter list and use localparams, which cannot be overridden, within the module itself. Parameters provide us with a great way to express design intent. When you return to a design after a long period of time, *magic numbers* such as 3.14 have much less meaning than pi.

Let's take a look at the data types we'll be using in SystemVerilog for data movement.

Introducing data types

All computer programming languages need variables. These are places in memory or registers that store values that the program that's running can access. **Hardware Design Languages (HDLs)** are a little different in that you are building hardware. There are variable equivalents in terms of storage/sequential logic, which we'll discuss in the next chapter, but we also need *wires* to move data around the hardware we're building using the FPGA routing resources, even if they are never stored:

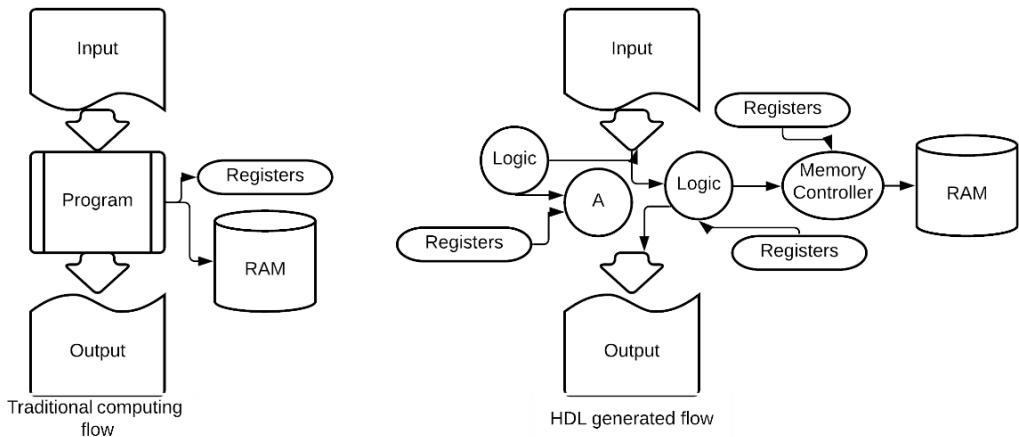


Figure 2.1 – Program flow versus HDL flow

As we can see, in a traditional flow, you have a computer that has a processor and memory. The program flows linearly; however, with modern machines, there are increasing levels of parallelism. When you write SystemVerilog, you are using data types to create hardware that will store or move data around physically from **Lookup Tables (LUTs)** to LUTs. If you want to use external memory, which is something we will introduce in *Chapter 8, Lots of Data? MIG and DDR2*, you need to implement the hardware to communicate with the memory.

Introducing built-in data types

SystemVerilog has multiple built-in types, but the most interesting ones for design are the logic and bit type:

- **logic:** We used this type in the previous chapter. The logic type can represent a 0, 1, x (undefined or treated as don't care, as we'll see shortly), or z (tri-state or also a don't care).

Important note

If you've ever used verilog, you will know of the `reg` type. This was a confusing type to new HDL designers as they would see `reg` and think it's short for register. In fact, a `reg` type was any signal originating from an `always` block, even though `always` blocks could be used to generate combinational logic, as we'll see shortly. Although `reg` can still be used for backward compatibility, you would be better off using `logic` or `bit`, which can be used in both `assign` statements and `always` blocks. The `logic` type also allows for the propagation of `x` through a design. This can be helpful for debugging startup conditions.

- `bit`: The `bit` type uses less memory to store than `logic`, but it can only store a 0 or 1. This allows for lower memory usage and potentially faster simulation at the expense of tracking undefined values.

There are also four other, lesser used two state types:

- `byte`: 8 bits
- `shortint`: 16 bits
- `int`: 32 bits
- `longint`: 64 bits

Important note

The differences between `bit` and `logic` are purely related to how they behave in simulation. Both types will generate the same logic and storage elements in hardware. All the other types only differ in size or default sign representation.

With that, we've looked at the basic types. But what if we need to deal with different sizes of data or more data than the types can handle?

Creating arrays

The reason that `byte`, `shortint`, `int`, and `longint` are not used as much is because typically, you will size your signals as needed; for example:

```
bit [7:0] my_byte; // define an 8 bit value
```

Here, `my_byte` is defined as a packed 8-bit value. It's possible to also create an unpacked version:

```
bit my_byte[8]; // define an 8 bit value
```

Packed versions have the advantage of slicing into arrays, while unpacked versions have the advantage of inferring memories, as we'll discuss in *Chapter 5, FPGA Resources and How to Use Them*.

Arrays can also have multiple dimensions:

```
bit [2:0][7:0] my_byte[1024][768]; // define an 8 bit value  
//   3     4           1     2           Array ordering
```

The ordering of the array is defined in the preceding code. The following are valid ways to access the array:

```
my_array[0][0] Returns a value of [2:0][7:0]  
my_array[1023][767][2] Returns an 8 bit value
```

Defining an array can be done using a range, such as `[7:0]`, or a number of elements, such as `[1024]`.

Querying arrays

SystemVerilog provides system functions for accessing array information. As we'll see in this project, this allows for reusable code.

Important note

The dimension parameter is optional and defaults to 1.

This becomes even more important when we want to implement type parameters:

<code>\$dimensions(my_array)</code>	4.
<code>\$left(my_array, [dimension])</code>	<code>[1]=1023,[2]=767,[3]=2,[4]=7.</code>
<code>\$right(my_array, [dimension])</code>	0 for all the dimensions.
<code>\$high(my_array, [dimension])</code>	Largest value in the dimension's range.
<code>\$low(my_array, [dimension])</code>	Smallest value in the dimension's range.
<code>\$size(my_array, [dimension])</code>	<code>= \$high(my_array, [dimension]) - \$low(my_array, [dimension]) + 1.</code>
<code>\$increment(my_array, [dimension])</code>	Returns 1 if <code>\$left >= \$right</code> ; otherwise, -1. Useful for for loops.
<code>\$bits()</code>	Returns the number of bits used by a particular variable or expression. This is useful for passing size information to instantiations.
<code>\$clog2()</code>	Returns the size of an array that can hold that number of items, not the value that was passed. For example, <code>\$clog(4)</code> returns 2, which can store four values from 0 to 3.

These system functions allow us to query an array to get its parameters.

Assigning to arrays

When we want to assign a value to a signal defined as an array, we should size it properly to avoid warnings. If we don't specify a size, then the size defaults to 32 bits, which was part of the Verilog Language Reference Manual (LRM).

There are three ways we can assign without providing a sign: '`1`' assigns all bits to 1, '`0`' assigns all bits to 0, and '`z`' assigns all bits to z. If we have a single packed dimension, we can use `n'b` to specify a binary value of n bits, `n'd` to specify a decimal value of n bits, or `n'h` to specify a hex value of n bits:

```
logic [63:0] data;
assign data = '1; // same as data = 64'hFFFFFFFFFFFFFFF;
assign data = '0; // same as data = 64'd0;
assign data = 'z; // same as data = 64'hzzzzzzzzzzzzzz;
assign data = 0; // data[31:0] = 0, data[63:32] untouched.
```

It's important to remember that n in these cases is the number of bits, not the number of digits.

Handling multiple-driven nets

There is one other type that deserves to be mentioned, although we will not be using it for a while. This is a wire. The wire type represents 120 different possible values; that is, the four basic values – 0, 1, x, and z – and drive strengths. The wire type has what is known as a resolution function. Wire types are the only signals that can be connected to multiple drivers. We will see this when we introduce the Serial Peripheral Interface (SPI) protocol and access the DDR2 memory on the Nexys A7 board:

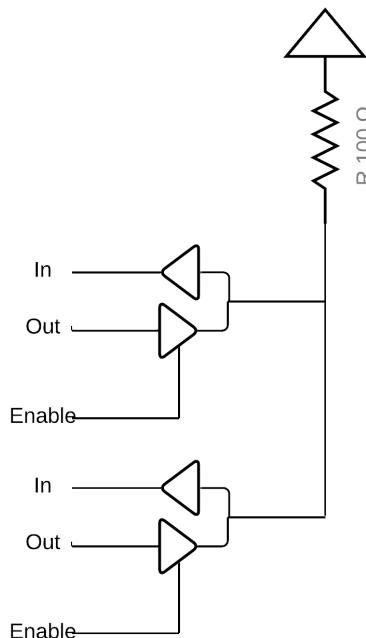


Figure 2.2 – Tri state example

FPGAs, in general, do not have internal tri-state capabilities. The preceding example shows two devices each with tri-state **Input/Output (I/O)** buffers connected:

```

logic [1:0] in;
logic [1:0] out;
logic [1:0] enable;
tril R_in;
assign R_in = (enable[0]) ? out[0] : 'z;
```

```

assign R_in = (enable[1]) ? out[1] : 'z;
assign in[0] = R_in;
assign in[1] = R_in;

```

The preceding code demonstrates how the two tri-state buffers are constructed. `tri1` is a testbench construct where a signal is declared as a tri-state with a weak pullup to 1.

Handling signed and unsigned numbers

Verilog had just one signed signal type, `integer`. SystemVerilog allows us to define both unsigned and signed numbers explicitly for any built-in type:

```

bit signed [31:0] signed_vect; // Create a 32 bit signed value
bit unsigned [31:0] unsigned_vect; // create a 32 bit unsigned
value

```

When performing signed arithmetic, it's important to make sure the sizing is correct. Also, when computing with signed numbers, you should make sure all the signals involved are signed so that the correct result is obtained.

Important note

Digital logic, such as computer processors or FPGA implementations, use 2's complement to represent signed numbers. What this means is that to negate a number, you simply invert it and add 1. For example, to get -1 in 2's complement, assuming there's 4 bits for representation, we would take 4 '`b0001`', invert it to get 4 '`b1110`', and add 1, resulting in 4 '`b1111`'. Bit 3 is the sign bit, so if it's 0, the number is positive and 1 if it's negative. This also means that the maximum number of signed values that we can represent by using 4 bits is 4 '`b0111`' or +7 and 4 '`b1000`' or -8.

Adding bits to a signal by concatenating

SystemVerilog provides a powerful concatenation function, { }, for adding bits or signals to create larger vectors or replication. When casting an unsigned integer to a signed integer, typically, you'll want to use the concatenation operator, { }, to prepend 1 '`b0`' into the sign bit so that the resulting signal remains unsigned. The concatenation operator can be used to merge multiple signals together, such as {1'`b0`, `unsigned_vect`}. It can also be used to replicate signals. For example, {2{`unsigned_vect`}} would be equivalent to {`unsigned_vect`, `unsigned_vect`}.

Casting signed and unsigned numbers

You can cast an unsigned number to a signed number by using the `signed'` keyword, and cast a signed number to an unsigned number using the `unsigned'` keyword:

```
logic unsigned [15:0] unsigned_vect = 16'hFFFF;
logic unsigned [15:0] final_vect;
logic signed [16:0] signed_vect;
logic signed [15:0] signed_vect_small;
assign signed_vect = signed'({1'b0, unsigned_vect}); // +65535
assign signed_vect_small = signed'(unsigned_vect); // -1
assign unsigned_vect = unsigned'(signed_vect);
assign final_vect = unsigned'(signed_vect_small); // 65535
```

Here, you can see that an unsigned 16-bit number can go from 0 to 65535. A 16-bit signed number can go from -32768 to 32767, so if we assign a number larger than 32767, it would have its sign bit set in the same-sized signed number, causing it to become negative.

These are equivalent to the verilog system functions; that is, `$signed()` and `$unsigned()`. However, it's preferable to use the casting operators.

Important note

When casting signed to unsigned or unsigned to signed, pay attention to sizing. For example, to maintain the positive nature of unsigned, typically, you'll use the concatenation operator, `{ }`, as in `signed'({1'b0, unsigned_vect})`; which means the resulting signal will be 1 bit larger. When going from signed to unsigned, care must be taken to ensure that the number is positive; otherwise, the resulting assignment will not be correct. You can see an example of mismatched assignments in the preceding code, where `signed_vect_small` becomes -1 rather than 65535 and `final_vect` becomes 65535, even though `signed_vect_small` is -1.

Creating user-defined types

We can create our own types using `typedef`. A common example that's used in SystemVerilog is to create a user-defined type for speeding up simulations. This can be done by using a define:

```
`ifndef FAST_SIM
  typedef bit bit_t
```

```

`else
  typedef logic bit_t
`endif

```

If FAST_SIM is defined, then any time we use bit_t, the simulator will use bit; otherwise, it will use logic. This will speed up simulations.

Tip

It is a good idea to adopt a naming convention when creating types – in this case, _t. This helps you identify user-defined types and avoid confusion when using the type within your design.

Accessing signals using values with enumerated types

When it comes to readability, it's often preferable to use variables with values that make more sense and are self-documenting. We can use enumerated types to accomplish this, like so:

```
enum bit [1:0] {RED, GREEN, BLUE} color;
```

In this case, we are creating a variable, color, made up of the values RED, GREEN, and BLUE. Simulators will display these values in their waveforms. We'll discuss enumerated types in more detail in *Chapter 3, Counting Button Presses*.

Packaging up code using functions

Often, we'll have code that we will be reusing within the same module or that's common to a group of modules. We can package this code up in a function:

```

function [4:0] func_addr_decode(input [31:0] addr);
  func_addr_decode = '0;
  for (int i = 0; i < 32; i++) begin
    if (addr[i]) begin
      return(i);
    end
  end
endfunction

```

Here, we created a function called `func_addr_decode` that returns a 5-bit value. The function takes a 32-bit input called `address`. Functions can have multiple outputs, but we will not be using this feature. To return the function's value, you can assign the result to the function name or use the `return` statement.

Creating combinational logic

The two main ways of creating logic are via `assign` statements and `always` blocks. `assign` statements are convenient when creating purely combinational logic with only a few terms. This is not to say the resulting logic will necessarily be small. For instance, you could create a large multiply accumulator using a single line of code, or large combinational structures by utilizing an `assign` statement and calling a function:

```
assign mac = (a * b) + old_mac;  
assign addr_decoder = func_addr_decode(current_address);
```

An `always` block allows for more complex functionality to be defined in a single process. We looked at `always` blocks in the previous chapter. There, we were using a sensitivity list in the context of a testbench. Sensitivity lists allow an `always` block to only be triggered when a signal in the list changes. Let's look back at the testbench that was provided in *Chapter 1, Introduction to FPGA Architectures and Xilinx Vivado*:

```
always @(SW, LED) begin
```

In this example, the `always` block would only be triggered when `SW` or `LED` transitions from one state to another.

Important note

Sensitivity lists are not synthesizable and are only useful in testing.
`always_comb` is recommended when describing synthesizable code in an `always` block.

When we write synthesizable code using an `always` block, we use the `always_comb` structure. This type of code is synthesizable and recommended for combinational logic. The reason is that `always_comb` will create a warning or error if we inadvertently create a latch.

Important note

A note about latches: They are a type of storage element. They are level-sensitive, meaning that they are *transparent* when the gating signal is high, but when the gating signal transitions to low, the value is held. Latches do have their uses, particularly in the ASIC world, but they should be avoided at all costs in an FPGA as they almost always lead to timing problems and random failures. That being said, we will demonstrate how a latch works and why it can be bad as part of this chapter's project.

There are a few different operations that can go within an `always` block. Since we are generating combinational logic, we must make sure that all the possible paths through any of these commands are covered. We will discuss this later.

Handling assignment operators

There are two basic types of assignments in SystemVerilog: blocking and non-blocking. Because we are writing in an HDL, we need to be able to model the hardware we are creating. All the hardware you design will be effectively running in parallel inside the FPGA.

Creating multiple assignments using non-blocking assignments

In hardware, whenever you create multiple `always` blocks, they are all executing at the same time. Since this is effectively impossible on a normal computer running linearly or, at best, a few threads in parallel, we need a way to model parallel behavior. Simulators accomplish this by using a scheduler that splits up simulation time into delta cycles. This way, if multiple assignments are scheduled to happen, there is still a linear flow to them. This makes handling blocking and non-blocking assignments critical.

A non-blocking assignment is something that is scheduled to occur within a delta when the simulator's time advances. We will discuss non-blocking in more detail in *Chapter 3, Counting Button Presses*.

Using blocking assignments

Blocking assignments occur immediately. With rare exception, usually only with regards to testbenches, all assignments within an `always_comb` block will be blocking.

There are several blocking assignments in SystemVerilog:

=	Assign the RHS to the LHS.
+=	Increment by value on RHS and assign.
-=	Decrement by value on RHS and assign.
*=	Multiply by value on RHS and assign.
/=	Divide by RHS value and assign.
%=	Divide by RHS value and assign modulus.
&=	Logical and by RHS and assign.
=	Logic or with RHS and assign.
^=	Logical XOR with RHS and assign.
<<=	Bitwise left shift and assign (equivalent to multiplying by 2^{RHS}).
>>=	Bitwise right shift and assign (equivalent to dividing by 2^{RHS}).
<<<=	Arithmetic left shift and assign (equivalent to multiplying by 2^{RHS}); the sign bit is preserved.
>>>=	Arithmetic right shift and assign (equivalent to dividing by 2^{RHS}); the sign bit is preserved.

There are also some shortcuts for incrementing or decrementing signals.

Incrementing signals

Here's a list of the shortcuts for incrementing:

- Pre-increment, `++i`, increments the value of `i` before using it
- Post-increment, `i++`, increments `i` after using it
- Pre-decrement, `--i`, increments the value of `i` before using it
- Post-decrement, `i--`, increments `i` after using it

Now that we've learned how to manipulate values, let's learn how to use these variables to make decisions.

Making decisions – if-then-else

One of the basics of any programming language is to control the flow through any operation. In the case of an HDL, this is generating the actual logic that will be implemented in the FPGA fabric. We can view an if-then-else statement as a multiplexor, the conditional expression of the `if` statement the select lines. Let's take a look at it in its simplest form:

```
if (add == 1) sum = a + b;  
else           sum = a - b;
```

This will essentially select whether b will be added or subtracted from a based on whether the add signal is high. A simplified view of what could be generated is shown in the following diagram:



Figure 2.3 – An if-then-else representation

In all likelihood, the logic will be implemented in a much less expensive way. It's worth looking at the results of your designs as they are built to understand the kind of optimizations that occur.

Comparing values

SystemVerilog supports normal equality operations such as `==` and `!=`. These operators check if two sides of a comparison are equal or not equal, respectively. Since we are dealing with hardware and there is the possibility of us having undefined values, there is a disadvantage to these operators in that `x`'s can cause a match, even if it's not intended, by falling through to the `else` clause. This is usually more of an issue in testbenches. There are versions of these operators that are resistant to `x`'s; that is, `====` and `!==`. In a testbench, it is advised to use these operators to avoid unanticipated matches.

Comparing wildcard equality operators

It is also possible to match against ranges of values. This is possible using the `=?=?` and `!?=` operators. These allow us to use wildcards in the match condition. For example, say you had a 32-bit bus, but needed to handle odd aligned addressing:

```

if (address[3:0] =?= 4'b00zz)      slot = 0;
else if (address[3:0] =?= 4'b01zz)  slot = 1;
  
```

The wildcard operators allow you to do this. The preceding examples would ignore the lower two bits.

Qualifying if statements with unique or priority

Normally, when thinking of an `if` statement, you think of each `if` evaluation as a separate comparison relying on the previous `ifs` that came before it. This type of `if` statement is a priority, meaning that the first `if` that matches will evaluate to true. In the simple example shown previously, we can see that we are looking at the same address and masking out the lowest two bits. Often, during optimization, the tool will realize that the `if` statements cannot overlap and will optimize the logic accordingly. However, if we know this to be the case, we can use the `unique` keyword to tell Vivado that each `if` doesn't overlap with any that come before or after. This allows the tool to better optimize the resulting logic. Care must be taken, however. Let's see what would happen if we tried to do the following:

```
unique if (address[3:0] == 4'b00zz) slot = 0;  
else if (address[3:0] == 4'b01zz) slot = 1;  
else if (address[3:0] == 4'b1000) slot = 2;  
else if (address[3:0] == 4'b1zzz) slot = 3;
```

Here, we can see that the last two `else if` statements overlap. If we specify `unique` in this case, we are likely to get a mismatch between simulation and synthesis. If `address[3:0]` was equal to `4'b1000` during the simulation, the simulator would issue a warning that the `unique` condition had been violated. Synthesis would optimize incorrectly, and the logic wouldn't work as intended. We'll see this when we violate `unique` on a `case` statement, when we work on this chapter's project.

This type of `if` is actually a priority, and if we wanted to, we could direct the tool, like so:

```
priority if (address[3:0] == 4'b00zz) slot = 0;
```

Priority is not really required except to provide clarity of intent. This is because the tool will usually be able to figure out if an `if` can be optimized as `unique`. If not, it will be treated as priority.

Introducing the case statement

A `case` statement is typically used for making a larger number of comparisons. There are three versions of the `case` statement you might use: `case`, `casex`, and `casez`. The `case` statement is used when wildcards are not necessary. If you want to use wildcards, as we saw previously, `casez` is recommended. There are two ways `case` statements are usually used. The first is more traditional:

```
casez (address[3:0])  
 4'b00zz: slot = 0;  
 4'b01zz: slot = 1;
```

```
4'b1000: slot = 2;
4'b1zzz: slot = 3;
endcase
```

Just like in the case of the `if` statement, `unique` or `priority` can be used to guide the tool. Also, we can have a default fall-through case that can be defined. This must be defined if `unique` is used.

Important note

`unique` and `priority` are powerful tools in that they can greatly reduce the final logic's area and timing. However, care must be taken as incorrectly specifying them can cause logic errors. Simulation will check that the conditions are not violated, but it will only detect cases that occur during simulation.

There is another way of writing a `case` statement that can be especially useful:

```
priority case (1'b1)
  address[3]: slot = 0;
  address[2]: slot = 1;
  address[1]: slot = 2;
  address[0]: slot = 3;
endcase
```

In this particular case, we have created a leading-one detector. Since we may have multiple bits set, specifying a `unique` modifier could cause optimization problems. If the design had one-hot encoding on `address`, then specifying `unique` would create a more optimized solution.

Important note

There are different ways to encode data. Binary encoding can set multiple bits at the same time and is typically an incrementing value. One-hot encoding has one bit set at a time. This makes decoding simpler. There is also something we'll explore when we discuss **First-In-First-Out (FIFOs)**, called gray coding, which is a manner of encoding that is impervious to synchronization problems when properly constrained.

For more simple selections, SystemVerilog supplies a simple way of handling this.

Using the conditional operator to select data

SystemVerilog provides a shortcut for conditionally selecting a result in the following form:

```
Out = (sel) ? ina : inb;
```

When `sel` is high, `ina` will be assigned to `out`; otherwise, `inb` will be assigned to `out`.

Tip

Writing `sel ? ...` is a shortcut for `sel == 1'b1 ?`

In this section, we've looked at basic data types and arrays and how to use them. In the next section, we'll learn how to use custom data types more tailored to our designs.

Using custom data types

SystemVerilog provides us with a variety of ways to create user-defined types. User-defined types can also be stored in arrays.

Creating structures

Structures allow us to group signals that belong together. For example, if we wanted to create a 16-bit value composed of two 8-bit values, `h` and `l`, we could do something like this:

```
typedef struct packed {bit [7:0] h; bit [7:0] l;} reg_t;
reg_t cpu_reg;
assign cpu_reg.h = 8'hFE;
```

Here's what the keywords signify:

- `typedef` signifies we are creating a user-defined type.
- `struct` means we are creating a structure.
- `packed` signifies the structure is to be packed in-memory.

Tip

Structures and unions can be packed or unpacked, but as packed tends to make more sense in the context of hardware, it's what we'll use here.

We access parts of a structure by using the created signal by appending the part of the structure – in this case, h – separated with a period.

Creating unions

A union allows us to create a variable with multiple representations. This is useful if you need multiple methods for accessing the same data. For instance, as microprocessors advanced from 8 bits to 16 bits, there needed to be ways of accessing parts of the register for older operations:

```
union packed {bit [15:0] x; cpu_reg cr;} a_reg;
always_comb begin
    a_reg.x = 16'hFFFF;
    a_reg.cr.h = '0;
end
```

In the preceding example, we created a union of a 16-bit register and a structure composed of two 8-bit values. After the first blocking assignment, a_reg sets all bits to 1. After the second assignment, the upper 8 bits were set to 0, meaning a_reg is 16'h00FF.

Project 1 – creating combinational logic

In this chapter, we've discussed signal types and how to create combinational logic. This project will contain multiple components that allow us to come up with a small calculator. It will be a rather simple one and will have the following capabilities:

- Find the leading-one position of a vector's input via switches
- Add, subtract, or multiply two numbers
- Count the number of switches that have been set

The following diagram shows what the Nexys A7 board looks like:



Figure 2.4 – Nexys A7 board I/O

In the previous chapter's project, we learn how to use switches for input and LEDs for output. In this project, we'll be using all the switches in the preceding diagram for the number of ones calculator and the leading-one detector. For the leading-one detector, we'll detect the position of the left-most switch that's been set out of the 16 positions.

For the arithmetic operations, we'll divide the switches into two groups. Switches 7 : 0 will be for input B, while switches 15 : 8 will be for input A. The output will be displayed as a 2's complement number using all the 16 LEDs above the switches, as shown in the preceding diagram. This means that -1 would mean all the LEDs are lit, while 0 would mean that all the LEDs are off.

Testbench

Since we will be building up individual components, we'll want a versatile testbench that will allow us to test each component individually and then all together. We'll accomplish this by using parameters. In this testbench, there are three parameters:

- SELECTOR is used for the leading-one module to determine one of four ways of finding the leading-one. It's also used to select between addition or subtraction for the `add_sub` module.

- UNIQUE_CASE determines whether we are going to generate unique case values or purely random numbers that can have multiple bits set.
- TESTCASE allows us to test individual components (LEADING_ONES, NUM_ONES, ADD, SUB, and MULT) or all of them (ALL).

To change these parameters in the testbench, select **Settings | Simulation | Generics/Parameters:**

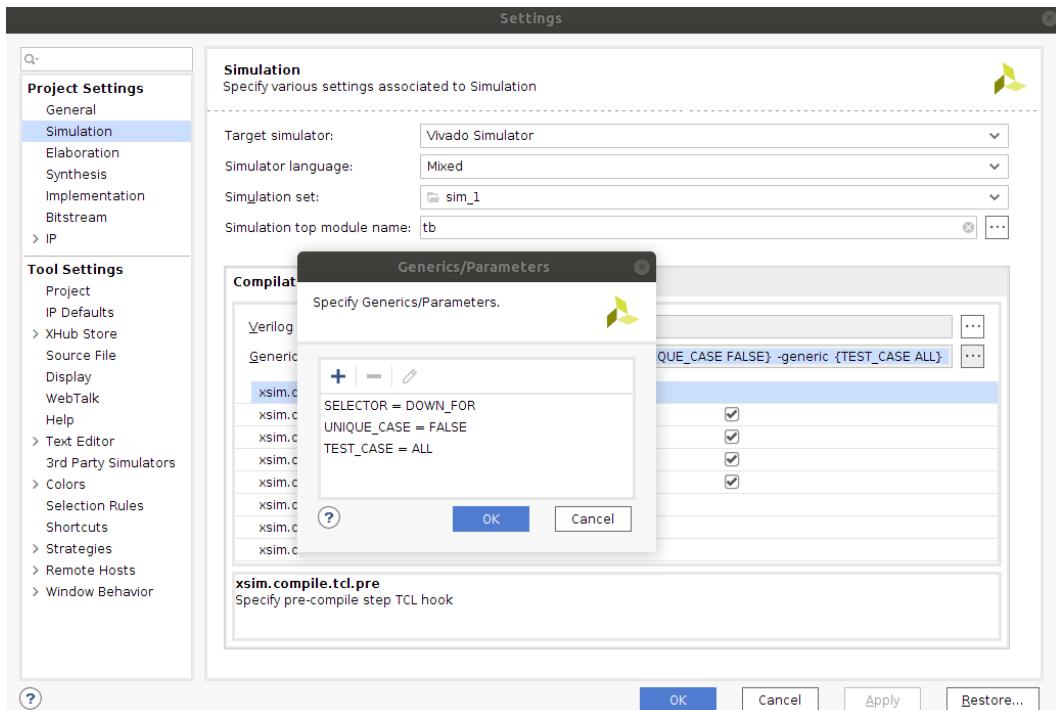


Figure 2.5 – Specifying simulation parameters

Similarly, to change the parameters for the implementation, select **Settings | General | Generics/Parameters**:



Figure 2.6 – Specifying implementation parameters

There are many ways to write testbenches. In the past, I've used separate include files for individual tests and used a shell script to invoke the simulator multiple times. If you are interested in exploring this type of testbench, please check out my open source graphics accelerator GPLGPU on GitHub: <https://github.com/asicguy/gplgpu>. What we will be using for our project is something simpler: using parameters to select test cases.

In general, there are three ways of testing your design.

Simulating using targeted testing

This type of test is used when you have a specific test case you want to make sure is hit. An example of this would be to see what happens when no bits are set in the leading-one detector, all bits are set in the number of ones, or the largest and smallest numbers in the case of mathematical operations. They can also be used to round out randomized testing.

Simulating using randomized testing

We are using this mostly in the self-checking testbenches that we'll be creating. To accomplish this, we'll use two system functions:

- `$random()`, which returns a 32-bit random number. It returns a new random number every time it's invoked.
- `$urandom_range(a, b)`, which returns a number inclusively between `a` and `b`. In our case, we are using `$urandom_range(0, 4)` to set one of the four buttons.

Next, we'll learn how to simulate using constrained randomization.

Simulating using constrained randomization

`SystemVerilog` has a very robust set of testing capabilities built into it. You can imagine this type of testing being used if you have a CPU with a number of valid instructions, and you want to randomize the testbench so that it uses these instructions and makes sure they are all used at some point. This is beyond the scope of this book, but I'll provide links in the *Further reading* section.

Implementing a leading-one detector using the case statement

Our first module will be a leading-one detector. We'll implement it in a few different ways and take a look at the advantages, disadvantages, and potential problems.

The first thing we need to decide is if the incoming signal is one-hot. If it is one-hot, we can get an optimized result by using the `unique` keyword:



Figure 2.7 – Testing leading-one using a case statement

Verify that your simulation parameters are set as shown in the preceding screenshot.

Controlling implementation using generate

Take a moment and examine the `leading_ones.sv` file. Here, you'll see how a `generate` statement can be used to selectively create code. The format of a `generate` statement is `generate <condition>`, as follows:

```
generate
  if (SELECTOR == "UNIQUE_CASE") begin : g_UNIQUE_CASE
```

In this case, the condition is an `if` statement, and is used to selectively instantiate one of four `always` blocks. Case statements and `for` loops are also valid conditions that we'll explore as we progress. This is where parameters are especially useful for controlling what gets created.

Tip

It is a good idea to use labels inside `generate` blocks. In future versions of SystemVerilog, this will be a requirement.

Notice that the `case` statement's default is commented out. Leave it as-is for now and run the test:

```
WARNING: 100000ns : none of the conditions were true for unique
case from File:/home/fbruno/git/books/Learn-FPGA-Programming/
CH2/hdl/leading_ones.sv:17
```

Why are we getting a warning? When we create a unique case, we must ensure that not only do we ever only match once, but we also match one. We want to make LED = 0 when no SW is set, so we uncomment the default. Now, we can run it again and the test will pass.

Important note

Parameters can control how logic is implemented or how testbench code is executed. In the testbench, you will see `if (UNIQUE_CASE == "TRUE") begin`, which controls how the code is executed to limit the number of ones being set.

Now, let's allow non-unique values to see how the simulator handles them. Change UNIQUE_CASE to "FALSE":

```
Setting switches to 0011010100100100
WARNING: 0ns : Multiple conditions true
          condition at line:21 conflicts with condition at line:20
          for unique case from File:/home/fbruno/git/books/Learn-
          FPGA-Programming/CH2/hdl/leading_ones.sv:17
```

This is only the first one that I saw, but you will see many. If our testbench hits cases that violate our unique assumption, we will see warnings that let us know the design may have problems.

So, let's see what happens when we take the design through to a bitstream by itself. Make sure that **Settings | General | Top Module Name** is set to `leading_ones` and that **SELECTOR**, under **Generics/Parameters**, is set to `UNIQUE_CASE`. Then, click on **Generate Bitstream**.

Important note

Generics/Parameters are set in two places in Vivado. General settings apply to building the design. Simulation applies only to simulation.

Take a look at the project summary. In the lower left of the window, look at the post-implementation utilization. By default, it comes up with a graph, but you can click on the table option to get hard numbers. In my build, this is what I got:



Resource	Utilization	Available	Utilization %
LUT	7	63400	0.01
IO	21	210	10.00

Figure 2.8 – Post-implementation utilization

We used 7 LUTs for this implementation. But what happens when we try this on the board? Open the hardware manager and the target, and then choose **Program device**.

We are expecting one-hot values, so try setting one bit at a time, starting from 0, so that only one switch is up at a time, one-hot encoded. Do you see the LEDs light up properly? You should see the binary value for the switch you have set plus one, so SW0 will show 5 'b00001, SW1 will show 5 'b00010, and SW15 will show 5 'b10000. Now, try to set multiple switches, such as 15 and 0. What did you get? In my case, I saw 5 'b10001. Now, try some others. You'll notice that some combinations still give the correct value by chance. There must be something to those warnings!

Now, let's try rebuilding without the `unique` keyword. Set `SELECTOR` to "CASE" and then generate the bitstream.

By looking at the summary of this build, we can see that handling priority cost us almost 2x the number of LUTs. My build took 13. Let's try it on the board.

Try combining multiple switches. Do you always get the switch position +1 for the uppermost switch?

In this section, you saw that `unique` allows optimization. The `unique case` statement was almost half the size of the `case` without `unique`. The `case` statement does have the disadvantage of us having to specify all possible cases, so it's not really reusable for an arbitrary number of cases. Let's explore another, more scalable way of handling a leading-one detector: using a `for` loop.

Designing a reusable leading-one detector using a for loop

The `for` loop allows us to quickly create replicated logic. In the case of a leading-one detector, it is also easy to imagine how we can do this using a `for` loop. There are two ways to accomplish this, both of which we'll look at in this section.

Setting SELECTOR = DOWN_FOR

The first is straightforward and follows along the lines of how the `case` statement accomplishes this task:

```
always_comb begin
    LED = '0;
    for (int i = $high(SW); i >= $low(SW); i--) begin
        if (SW[i]) begin
            LED = i + 1;
            break;
        end
    end
end
```

We use the `$high` and `$low` system tasks for reusability. The loop breaks when a 1 is detected for the first time.

Tip

A `break` in a `for` loop is synthesizable. The important thing to consider is whether you can unroll the loop or if there is a way to write the loop in a way that the `break` isn't necessary. If you can think of a relatively easy way this can be done, then you probably won't have an issue synthesizing it.

For example, we could unroll the loop by writing it as follows:

```
Logic [3:0] SW;
always_comb begin
    LED = '0;
    if      (SW[3]) LED = 4;
    else if (SW[2]) LED = 3;
    else if (SW[1]) LED = 2;
    else if (SW[0]) LED = 1;
```

```
    else          LED = 0;  
end
```

We can now look at another way of writing the `for` loop that satisfies our unrolling requirement.

Setting SELECTOR = UP_FOR

By progressing from the lowest bit to the highest bit while searching for a 1, we are guaranteed to find the highest bit as the last 1 that's found. This is also how you know that the break can be synthesized, since we have found a way to rewrite the `for` loop so that it's not necessary.

Counting the number of ones

Related to finding the leading-one is counting the number of ones in a vector. We can do this easily using a `for` loop:

```
always_comb begin  
    LED = '0;  
    for (int i = $low(SW); i <= $high(SW); i++) begin  
        LED += SW[i];  
    end  
end
```

Set `SELECTOR` to `NUM_ONES` and `TEST_CASE` to `NUM_ONES` and run the simulation to verify it works. Verify that `SELECTOR` is set to `NUM_ONES` under the **General** tab and that the top module's name is set to `num_ones`. Then, generate the bitstream and run it on the board.

Verify the design on the board by flipping the switches one by one in any order. You should see the LEDs light up in the pattern of a binary count; that is, `16'b0`, `16'b1`, `16'b10`, `16'b11`, and so on.

Implementing an adder/subtractor

Let's take a look at the `add_sub` module. There are many ways to implement an adder or subtractor in math in general. Many companies sell tools for high performance or low gate count designs. For FPGAs, 99% of the time, you are better off letting the tools optimize your designs. Because of this, you'll see that the module itself is fairly small. We choose whether we are adding or subtracting based on the `SELECTOR` parameter.

Add

Set SELECTOR to ADD and TEST_CASE to ADD and run the simulation to verify it works. Verify that SELECTOR is set to ADD under the **General** tab and that the top module's name is set to add_sub. Then, generate the bitstream and run it on the board:

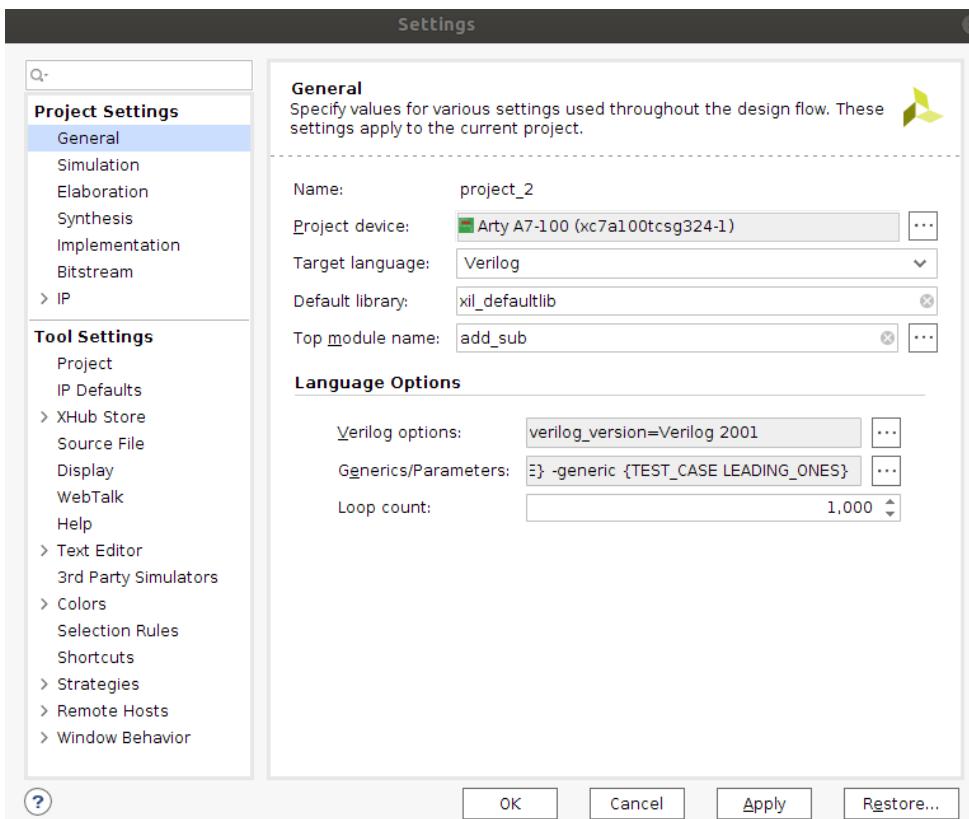


Figure 2.9 – Top module set to add_sub

Once you've downloaded the bitstream on the board, try some combinations of bits on the lower 8 and upper 8 bits. In particular, if you set bit 0 and bit 8 both to 1, you should see bit 1 set on the LED; that is, a value of $16'h2$. Now, try setting bit 0 and bit 15 – what do you get?

It may look a bit weird seeing so many LEDs lit, but you'll notice that only the upper bits are lit. This is because we have specified $8'h80 + 8'h1$. Since we are specifying two's complement numbers, in decimal, this would be $-128 + 1$ or -127 , which in hex would be $16'hFF81$.

Subtractor

Set SELECTOR to SUB and TEST_CASE to SUB and run the simulation to verify it works. Verify that SELECTOR is set to SUB under the **General** tab and that the top module's name is set to add_sub. Then, generate the bitstream and run it on the board.

Now, we are subtracting the lower 8 bits from the upper 8 bits. Try setting bit to 0. All the LEDs should be lit, or -1.

Important note

Remember, to get -1 in binary, we invert and add 1; for example,

```
-16'b0000000000000001 = 16'b1111111111111110 + 1 =
16'b1111111111111111.
```

Note that for the adder and subtractor, no matter what you add with signed numbers, the upper 8 bits will always be either all 0s or all 1s.

Multiplier

The final module we will look at is the multiplier. HDL is the simplest out of all of them, and since the multiplier is only 8*8, by default, it is implemented in LUTs.

Set SELECTOR to MULT and TEST_CASE to MULT and run the simulation to verify it works.

This simulation is automated. However, we can also use the `add_force` command in the simulator. An example of this is shown in the following screenshot. A force will override a value on a signal in the simulator. When the simulation ended, I forced a value of `0x1234` onto the `SW` input of the multiplier. Since I've done this, I need to advance simulation time, which I can do with `run 10ns`.

The `force` command is good for when you are trying to isolate a particular scenario or experiment with a *what if* scenario during a run. In general, you will not want to simulate solely this way as you'll want to have a way of reproducing your results, so putting your tests in a SystemVerilog testbench is a better long-term solution.

If and when you are done with a scenario, you can use the `remove_forces` command on a signal to return control to the testbench:

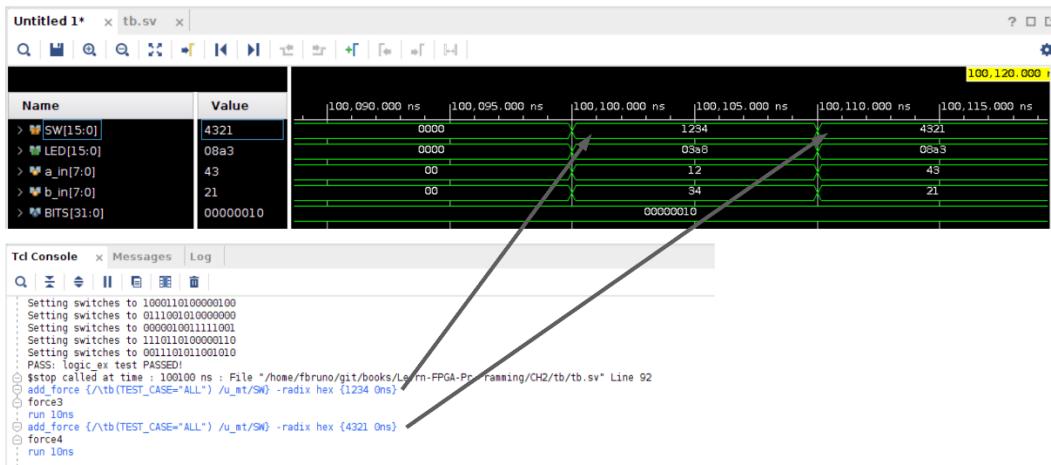


Figure 2.10 – Force statement in a simulation

Verify that SELECTOR is set to MULT under the **General** tab and the top module name is set to `mult` then generate the bitstream and run it on the board:

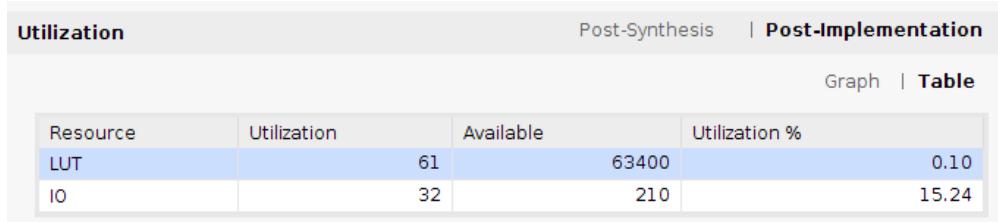


Figure 2.11 – Multiplier utilization

The preceding snippet shows our utilization from building the multiplier.

Tip

Adding two signed numbers of size n will result in a value of size n .

Adding two unsigned numbers of size n will result in a value of size $n+1$.

Multiplying two numbers of size n will result in a value of size $2 \cdot n$.

Bringing it all together

Now, we'll create a simple ALU top level so that we can bring everything together. Take a look at `project_2`. There are five buttons on the board. We'll use these to control the output:

Button	Operation
Center	Multiply
Up	Leading-one position
Down	Number of switches set to one
Left	$SW[15:8] + SW[7:0]$
Right	$SW[15:8] - SW[7:0]$

Instantiate the submodules. We'll need to use `add_sub` twice and use `SELECTOR` so that it's hardcoded to select the one we want. We'll still pass the selector to the leading-one calculator in case we want to play around with it:

```
leading_ones #( .SELECTOR (SELECTOR) , .BITS (BITS) )
  u_lo (.*, .LED (LO_LED));
add_sub    #( .SELECTOR ("ADD") ,      .BITS (BITS) )
  u_ad (.*, .LED (AD_LED));
add_sub    #( .SELECTOR ("SUB") ,      .BITS (BITS) )
  u_sb (.*, .LED (SB_LED));
num_ones   #( (                           .BITS (BITS) )
  u_no (.*, .LED (NO_LED));
mult       #( (                           .BITS (BITS) )
  u_mt (.*, .LED (MULT_LED));
```

Now that we have overridden the names of the LED outputs of the submodules, we can mux them to the LEDs:

```
always_comb begin
  LED = '0;
  case (1'b1)
    BTNC: LED = MULT_LED;
    BTNU: LED = LO_LED;
    BTND: LED = NO_LED;
    BTNL: LED = AD_LED;
```

```

BTNR: LED = SB_LED;
endcase
end

```

Set TEST_CASE to ALL and run the simulation to verify it works. Verify that SELECTOR is set to UNIQUE_CASE, CASE, UP_FOR, or DOWN_FOR under the **General** tab and that the top module's name is set to project_2. Then, generate the bitstream and run it on the board:



Figure 2.12 – Complete project_2 utilization

When the image finishes downloading, the LEDs will be off. Flip some switches and select a function by pushing a button. Congratulations – your simple calculator is complete! Notice that when you release the button, the LEDs go dark.

Adding a latch

Since we are not using any clocks yet, let's add a latch. In this particular case, the switches are static, so using a latch shouldn't cause us any problems:

```

always_latch begin
//always_comb begin
    /LED = '0;

```

Change always_comb to always_latch and comment out the LED = '0; default. Then, rerun it. What happens when you download and try to select an operation? If your build is like mine, then this operation will not be what you expected and the LEDs will seem to behave in an almost random fashion. This is the reason that I have stressed not to use latches. If you encounter a situation where your circuit doesn't behave as intended, search the compile logs and make sure no latch is inferred.

Summary

In this chapter, we learned how to create combinational logic, how to create different modules, and how to test them as utilize self-checking testbenches. We also explored different optimizations we can perform on the case statement and showed you how to get substantial area savings in some cases, but also how we may have problems if our design assumptions are incorrect. We then mentioned latches and the problems they cause, even when they should be safe.

At this point, hopefully, you have some confidence in how to create logic and test it. In the next chapter, we'll introduce sequential logic; that is, using registers to store values and perform operations. We'll expand upon our simple calculator and see how we can improve it now that we have some storage elements.

Questions

1. A packed array is used to infer memories. True or false?
2. A break statement can be used in a `for` loop when?
 - a) Any time.
 - b) If it's possible to rewrite the `for` loop in such a way as to not need the break.
 - c) Only if you can reverse the direction of the loop; that is, go from low to high instead of high to low.
3. Size the `add_unsigned`, `add_signed`, and `mult` signals:

```
Logic unsigned [7:0] a_unsigned;
logic unsigned [7:0] b_unsigned;
logic signed [7:0] a_signed;
logic signed [7:0] b_signed;
assign add_unsigned = a_unsigned + b_unsigned;
assign add_signed = a_signed + b_signed;
assign mult = a_unsigned * b_unsigned;
```

4. Division is a very costly operation. Look at the supported Vivado constructs in the Vivado Synthesis manual (*Further reading*). Can you easily replace the multiply operation with a division operation? What is possible without custom code?

Challenge

Look at the following add_sub module:

```
logic signed [BITS/2-1:0]      a_in;
logic signed [BITS/2-1:0]      b_in;
...
{a_in, b_in} = SW;
```

If you were to replace a_in and b_in with a custom type that encapsulates both, would you use a structure or a union? Modify the code so that it uses your custom type, and then simulate and try it on the board.

Further reading

Please refer to the following links for more information regarding what was covered in this chapter:

- UVM information: <https://www.accellera.org/downloads/standards/uvm>
- Vivado Synthesis manual: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug901-vivado-synthesis.pdf

3

Counting Button Presses

In this chapter, we'll learn how to maintain the state of a design by adding sequential elements. Limited to combinational logic with no way to store information, we can't actually accomplish very much. In order to have a useful CPU, you need a program counter, registers, and long-term storage. What would your cell phone be without the capability to store numbers, emails, or pictures?

In this chapter, we are going to cover the following main topics:

- Learn what sequential elements are and how to use them
- Project – Counting button presses
- Looking at synchronization in detail

Technical requirements

The technical requirements for this chapter are the same as those for *Chapter 1, Introduction to FPGA Architectures and Xilinx Vivado*.

To follow along with the examples and the project, you can find the code files for this chapter at the following repository on GitHub: <https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH3>.

What is a sequential element?

We looked at the latch in *Chapter 1, Introduction to FPGA Architectures and Xilinx Vivado*, and we saw that it's not something we really want to be using. What FPGA designers use to store information is a register, or flip flop. Before we create our first flip flop, we need a quick introduction to clocks.

Clocking your design

In the realm of digital logic, we usually need at least one source of timing in our design and often several. We call this source of timing a clock, which is usually generated by an external crystal oscillator that vibrates at a certain frequency and generates a string of 0s and 1s in our design. Sometimes we'll use the clock input directly, but if we need a specific frequency faster or slower than our input, we have other options such as **Phase Locked Loops (PLLs)** and **Mixed Mode Clock Managers (MMCMs)**, which we'll discuss in *Chapter 5, FPGA Resources and How to Use Them*.

When we draw timing diagrams, we typically draw our clocks and the inputs and outputs as square waves, as shown in *Figure 3.1*:



Figure 3.1 – Simple clock

We also need to tell Vivado about the clock we have created so it can properly time our designs. Up until now, we've ignored timing since we haven't had a reference to measure time against. We'll be adding the following to our XDC file:

```
## Clock signal
set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMS33 } \
[get_ports { clk }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name clk -period 10.00 -waveform {0 5} \
[get_ports {clk}];
```

To create a clock in a design, we use the `create_clock` **Tool Command Language (TCL)** command. We need to specify the period of the clock in nanoseconds and, optionally, we can specify what the waveform looks like. We apply this to a port on the design using `get_ports` and give the clock a name using `-name`. In more complex designs, it's possible to define multiple clocks on a given pin; for example, you might have a fast clock for performance and a slow clock for saving power and use a **Phase Lock Loop (PLL)** to generate the clock. By applying multiple clocks, the timing analyzer can make sure that your design meets timing and has safe clock domain crossings.

Tip

In most cases, you don't need to worry about specifying the waveform. If you have multiple clocks of the same frequency shifted in phase, then you would want to specify the waveform option to ensure the timing analysis is performed properly.

For now, we'll keep things simple and generate a single clock.

Looking at a basic register

If you were designing an ASIC, you'd likely have a few different register types in your library because they may be optimized for area based on functionality, such as toggle **Flip Flops (FF)**, devices that toggle when the control signal is high and they are clocked. Since Xilinx is targeting all possible design types, the registers are based on what are known as **D Flip Flops (DFFs)**:

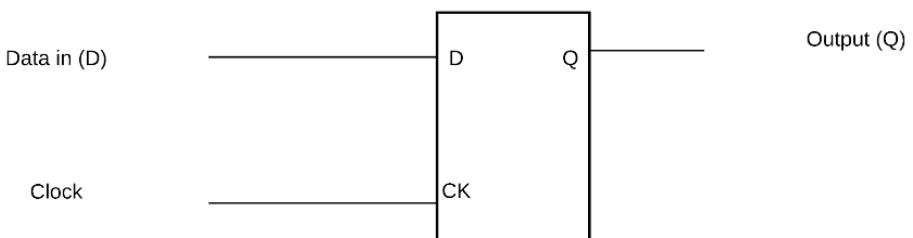


Figure 3.2 – Simple DFF

The simple DFF in *Figure 3.2* accepts data in the D input and stores it every clock cycle, presenting it in the Q output. This type of storage element must be continuously fed because every change in the input is mirrored in the output.

Creating a flip flop

In SystemVerilog, we can create a flip flop in one of two ways: using `always_ff @ (edge sensitivity list)` or `always @(edge sensitivity list)`.

Two SystemVerilog keywords that convey an event occurring on the edge of a signal are `posedge`, the rising edge of the clock, and `negedge`, the falling edge. In general, we'll only be using the rising edge, but in some special circumstances you may need the falling edge:



Figure 3.3 – Posedge DFF timing

The preceding diagram shows the clock edges labeled as `posedge` and `negedge`.

Tip

In general, stick to one edge of the clock and use it consistently. In our designs, we'll be strictly using the positive edge. This will help prevent timing problems in your design.

Let's look at how we construct a DFF in SystemVerilog:

CH3/simple_ff/hdl/simple_ff.sv

```
module dff (input wire D, CK, output logic Q);
  always_ff @(posedge CK) Q <= D;
endmodule
```

`always_ff` has the advantage of conveying design intent. Vivado accepts the construct; however, it doesn't generate an error if an FF is not inferred. Other tools will generate an error, so it's advisable to still use `always_ff` in most instances. If you'd like to try simulating this or running on the Nexys A7 board, the project is located at https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH3/simple_ff. Follow the procedures you learned in *Chapter 2, Combinational Logic*, to simulate, build, and test:



Figure 3.4 – Simple FF timing

From the preceding waveform, we see that `Q` is changing along with every `D` input. It allows us to store data, but if we were just using a DFF we would need to add recirculating logic ourselves to hold the data.

When to use `always @()` for FF generation

There is a limitation to `always_ff` in that any signal generated from it cannot be driven by any other construct. FPGAs support using an initial statement to determine the FF startup value. In our `simple_ff.sv` example, the startup value for `Q` is not defined by the code. This can be seen in *Figure 3.4*, where the startup condition is '`x`'. This means that the synthesis tool can use either 0 or 1. We can, however, create an initial value for the FF:

CH3/simple_init_ff/hdl/simple_init_ff.sv

```
module dff (input wire D, CK, output logic Q);
    initial Q = 1;
    always_ff @(posedge CK) Q <= D;
endmodule
```

However, if we try to run the simulator, we'll get the following in the messages pane:

```
✓ Simulation (2 errors)
  ✓ sim_1 (2 errors)
    ! [VRFC 10-3818] variable 'Q' is driven by invalid combination of procedural drivers [simple_init_ff.sv:6]
    ! [XSIM 43-3322] Static elaboration of top level Verilog design unit(s) in library work failed.
```

Figure 3.5 – Simulation failure using `always_ff` with an initial value

The fix is to change `always_ff` to `always`, and the design will work correctly. Make the change and run the simulation again:



Figure 3.6 – Initial value of Q

`Q` has an initial value of 1 in the preceding simulation; however, the first clock cycle immediately loads `D`, which is 0, into `Q`.

Using non-blocking assignments

You'll notice that for the first time we've performed an assignment by using `<=`, or the non-blocking assignment. The reason for using a non-blocking assignment is that now that we have introduced sequential elements, we need to address the inherent parallelism in **Hardware Design Languages (HDLs)** and look at scheduling as a way of modeling it.

Up until now, everything we've done has occurred as any conventional program would, in a series of steps performed sequentially. Let's take a look at how this would work if we applied the same structure to a block of code inferring a register:

CH3/blocking/hdl/blocking.sv

```
always @ (posedge CK) begin
    stage = D;
    Q      = stage;
end
```

Now we've introduced an intermediate storage element called `stage`. What happens if we run a simulation on the preceding code?



Figure 3.7 – Simulation of blocking assignments in a clocked always block

The important thing to note is that `stage` is immediately assigned the value of `D`, then `Q` is immediately the value of `stage`. `stage` in effect becomes a wire in the final implementation.

What happens if we try changing `BLOCK` to "FALSE" in the testbench?

```
always @(posedge CK) begin
    stage  <= D;
    Q      <= stage;
End
```

If we run the simulation, we now see `stage` behaving as a pipeline stage, as intended:



Every clock cycle, `Q` gets the previous value of `stage` and `stage` gets the previous value of `D`

Figure 3.8 – Simulation of non-blocking assignments in a clocked always block

Notice the difference. When we look at the preceding code using non-blocking assignments, it reads as follows:

- Schedule the assignment of `D` to the next value of `stage`, but the current value of `stage` remains the same.
- Schedule the assignment of the current value of `stage` to the next value of `Q`.

In fact, the simulator will go through the entire code of the design scheduling the assignments. These scheduled events are referred to as delta cycles. Once that has been completed, then time will advance. The assignments will then take effect and it will do it again. This is how we want our designs to behave.

Tip

All combinational blocks (`always_comb`) in a design should use blocking assignments. All sequential blocks (`always @(posedge)` or `always_ff`) should use non-blocking assignments. Failure to adhere to this can cause simulation/synthesis mismatches.

The project to demonstrate blocking versus non-blocking can be found in CH3 / blocking/build/blocking.xpr.

Registers in the Artix 7

We've looked at the simplest version of a register. These map fine into the Artix 7. However, the Artix 7 registers offer a lot more functionalities, which we'll examine here.

There are two FFs for every LUT, one dedicated FF and one that can be configured as an FF or Latch. As we saw in *Chapter 2, Combinational Logic*, latches are unreliable at best, so we won't be going over them here. Also, if latches are selected, the other four FFs cannot be used, further limiting the number of resources available. Here is an example of **Combinational Logic Block (CLB) registers**:

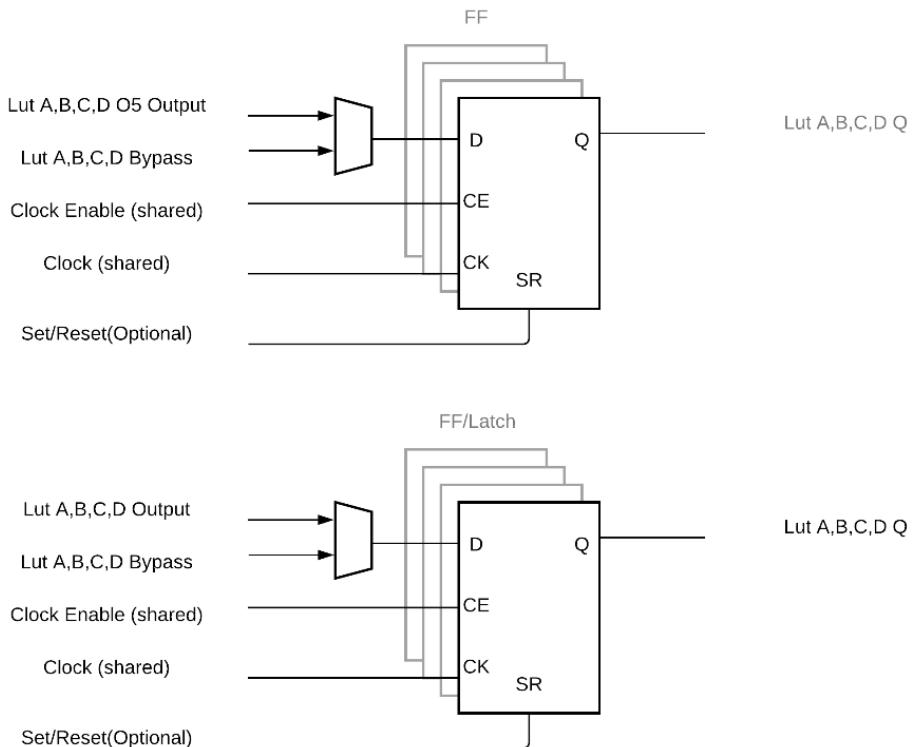


Figure 3.9 – Artix 7 CLB registers

For each group of FFs, we can see we now have a shared clock enable, a shared clock, and a shared set or reset line. The D inputs are all individually selectable from within the **Look Up Tables (LUTs)** or the LUT can be bypassed.

How to hold onto state using clock enables

The simple DFFs we've looked at change output whenever the input changes. Clock enables allow an FF to hold onto its value whenever we are not actively changing the data. Let's look at how we can use this in practice:

```
module dff (input wire D, CK, CE, output logic Q);
    initial Q = 1;
    always @ (posedge CK) if (CE) Q <= D;
endmodule
```

We can look at the waveforms and see how using the CE affects the Q output of the FF:

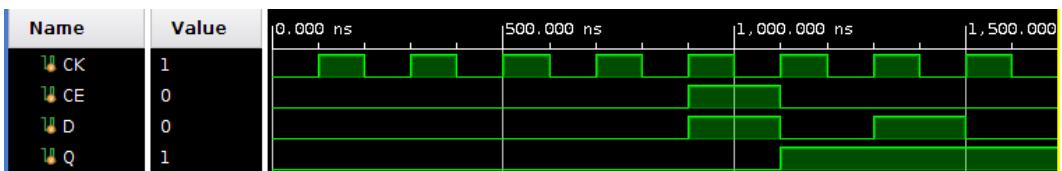


Figure 3.10 – Clock enabled FF

Notice how the first time D goes high along with the CE, Q goes high also. Once CE goes low, the value Q remains high regardless of the D input value.

Resetting the FF

We have seen how we can use an initial value to have the device come up in a known state. This works great after the initial download, but what if we need to operate in a system that gets reset on occasion? Xilinx designed the Artix 7 **Field Programmable Gate Array (FPGA)** with a configurable set/reset system. The LUT register's **Set/Reset (SR)** input can be configured as set or reset and synchronous or asynchronous.

The first choice is whether you need to set/reset your FF asynchronously or synchronously. Synchronously means the reset signal is generated by the clock driving the FF so it will be properly timed in the design. The limitation is that a clock is required to be running when you use synchronous resets. If the clock is not guaranteed to be running, then you need an asynchronous reset. Asynchronous resets should be designed such that they will assert asynchronously and release synchronously to the FF clock. This eliminates potential timing problems in the design.

Xilinx recommends limiting resets to essential signals to speed up timing analysis and save routing resources by limiting high fanout nets:

CH3/simple_ff_async/hdl/simple_ff_async.sv, ASYNC = "TRUE"

```
always @ (posedge CK, posedge SR) begin
    if (SR) Q <= '0;
    else if (CE) Q <= D;
end
```

In the preceding code, we see an example of inferring an FF with an asynchronous reset. It's a reset because Q gets a value of '0 when SR goes high. It would be a set if it went to a value of '1. We can see from the waveforms that the reset immediately affects the Q output:

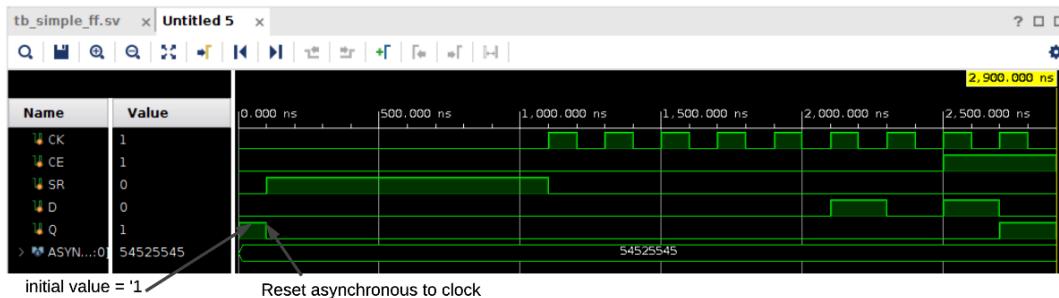


Figure 3.11 – Asynchronous reset simulation

Here, we can see an example of an FF with a synchronous reset:

CH3/simple_ff_async/hdl/simple_ff_async.sv, ASYNC != "TRUE"

```
always_ff @ (posedge CK) begin
    if (SR) Q <= '0;
    else if (CE) Q <= D;
end
```

Looking at the asynchronous version, we can see that `posedge SR` is in the sensitivity list. This is what allows the reset to be effective without the clock. We can see from the simulation waveforms that the reset is not effective until the clock starts running, unlike the asynchronous reset:



Figure 3.12 – Synchronous reset simulation

Tip

I would recommend synchronous resets when a reset is necessary unless you are not guaranteed to have a running clock. Limit the number of signals to be reset and the tool will take care of timing analysis.

Now that we have the basics of registers behind us, let's tackle this chapter's project.

Project 2 – Counting button presses

The project in this chapter will count button presses and display the count in a human-readable form using the seven-segment display.

Introducing the seven-segment display

In the previous chapters, we displayed binary numbers by using the LEDs on the board. You might have wondered why we weren't using the row of unlit 8s. The reason is that there is timing associated with the display that we need registers to accomplish.

Let's take a look at how we light up the seven segments. The following diagram shows which segment is controlled by which cathode:

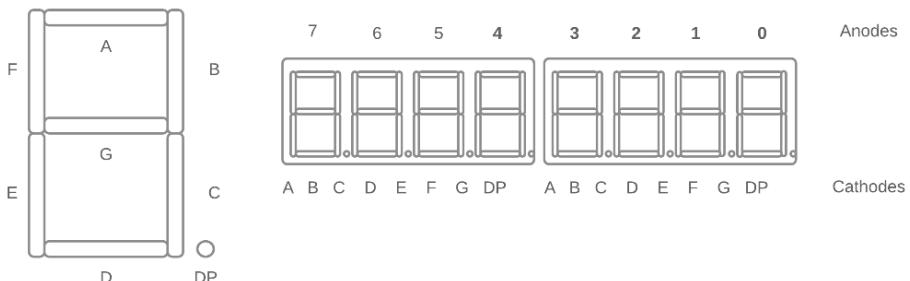


Figure 3.13 – Seven segment display

Looking at the preceding diagram, we can see there are eight signals that define whether a given LED is lit or not. To compose an image, we simply need to come up with a module that takes in a **Binary Coded Decimal (BCD)** or hexadecimal number and converts it to a format that the display can handle. We have a few choices of how to implement this. We can create one converter that can handle the whole display or we can have one converter for each digit. Either way, we need a design.

Important note

We've been using binary up until now, displaying one LED per bit.

Hexadecimal numbers represent 4-bit binary numbers from $4'h0$ to $4'hF$. BCD numbers are a way of representing decimal numbers in computer storage by using the values $4'b0000$ = $4'd0$ to $4'b1001$ = $4'd9$; values above 9 are not used.

Regardless of whether we are displaying BCD or hexadecimal, we can create a module that takes in a 4-bit number and encodes the bits we want to display, in this case onto an 8-bit cathode bus. We take the **digit point (DP)** in as a separate signal to keep it aligned with the other data:

CH3/counting_buttons/hdl/cathode_top.sv

```
always_ff @(posedge clk) begin
    cathode[7] <= digit_point;
    case (encoded)
        4'h0: cathode[6:0] <= 7'b1000000;
        4'h1: cathode[6:0] <= 7'b1111001;
        ...
        4'hE: cathode[6:0] <= 7'b0000110;
        4'hF: cathode[6:0] <= 7'b0001110;
    endcase
end
```

If we look at *Figure 3.13* we can see how the seven-segment displays are mapped. Each segment will light up when driven to 0. This is called an active low signal. From the code in `cathode_top` you can see, for example, that when `encoded` is $4'h0$, segments A-F are driven low, so they will be lit, and segment G is high, so it will be off. If we look at *Figure 3.13*, you can see this would result in a 0 being displayed.

Let's now take a look at the timing we need to display on the physical display. Note that the Basys 3 version has only 4x7 segments, so we'll need to make our encoding module parameterizable:



Figure 3.14 – Seven-segment display timing

We need to generate the anode signal by cycling a value of 0 through all the anode positions at a refresh rate of 1/8, or $\frac{1}{4}$ if using the Basys 3 board. We'll create two counters:

CH3/counting_buttons/hdl/seven_segment.sv

```
localparam INTERVAL = int'(1000000000 / (CLK_PER * REFR_RATE));
```

First, we'll create a local parameter, `INTERVAL`, that will be used to figure out the interval in which we need to cycle through the anodes. Note that we can calculate parameters. In this case, we are taking 1×10^9 nanoseconds in a second and dividing it by the clock period (100 MHz clock has a period of 10 nanoseconds, the period of the clock fed directly into the Arty A7 100T board) multiplied by the refresh rate. Since this will return a floating point value, we cast it to an integer by using `int'()`:

```
initial begin
    refresh_count = '0;
    anode_count   = '0;
end
always @(posedge clk) begin
    if (refresh_count == INTERVAL) begin
```

```
refresh_count <= '0;
anode_count    <= anode_count + 1'b1;
end else refresh_count <= refresh_count + 1'b1;
anode           <= '1;
anode [anode_count] <= '0;
cathode         <= segments [anode_count];
end
```

Looking at the preceding code, we are generating two counters, the first of which is the refresh counter, which generates the timing for when we will assert each anode. The INTERVAL for this counter is set based on the refresh rate. When the refresh counter reaches the INTERVAL value, we reset it and increment our second counter, anode_count. This specifies which anode will be asserted (the signal is active low, so drive a 0) for updating the seven-segment display. We don't bound the anode_count because it will count to either 4 or 8, so we can simply let it roll over naturally. If the number of displays was not a power of 2, we could bound our count the same way as we do refresh_count.

Detecting button presses

The buttons on the board are wired to deliver a 1 when pushed, meaning they are normally driving a value of 0 into the FPGA. This means to detect a button press we need to look for a rising edge.

Analyzing timing

Let's quickly take a look at clock relationships and how timing is analyzed. There are two main timing constraints between data and a clock:

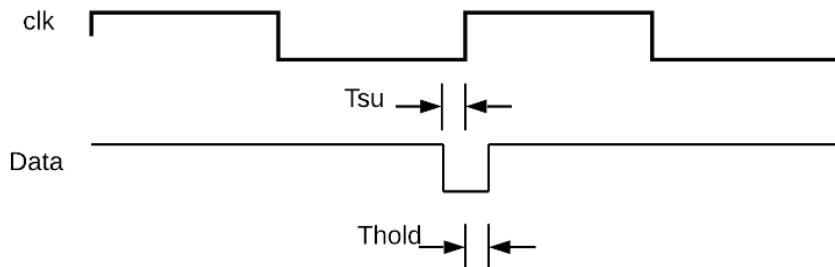


Figure 3.15 – Timing constraints

These constraints must be met for a design's reliable operation. The first constraint is the setup time, or **Tsu** in the timing constraints diagram. This is the amount of time that the signal is required to be stable prior to the clock edge. If the signal transitions within or after the **Tsu**, then the device may not operate correctly. In a synchronous design, **Tsu** is usually only violated if the clock frequency is too fast for the longest clock to clock data paths in the design. The thing about setup time in a synchronous design is that it can be fixed by lowering the clock speed or redesigning it to cut down long timing paths.

The second constraint, hold time or **Thold**, is the window in which a signal must remain stable after the clock. It's normally not a problem in a single **Super Logic Region (SLR)** device such as the one we are using. However, it is often a problem when designs have multiple clocks and care is not taken to properly synchronize signals between the clocks. The issue with hold time is that there is no way to fix it by reducing clock speed—you can only prevent it. When it occurs in a synchronous design, the cause is usually locally routed clocks in designs with major routing congestion.

Important note

Xilinx builds very large devices that consist of multiple FPGA die bonded to a substrate. Each one of these dies is what is called an SLR. This allows for high-density devices that rival ASICs in terms of gate counts.

When we have a purely asynchronous signal, such as BTNC, the center button, we cannot fix the setup and hold as easily without proper synchronization.

Looking at asynchronous issues

Hopefully, in your mind you can picture how a purely random asynchronous signal such as BTNC could cause both setup and hold time violations. To show the problem, I've added a `create_clock` timing constraint on the BTNC input that you can uncomment and run. The frequency doesn't matter, just that we create it so the edges will wander with its relationship to `clk`:

CH3/counting_buttons/build/Nexys-A7-100T-Master.xdc

```
create_clock -add -name BTNC -period 99.99 [get_ports {BTNC}] ;
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets BTNC_IBUF]
```

We'll also need to set `CLOCK_DEDICATED_ROUTE` to FALSE since the BTNC, the center button, pin was not set to a dedicated clock pin. Normally you want an external clock on your FPGA to come in on a pin that connects to the dedicated clock resources of the FPGA. `CLOCK_DEDICATED_ROUTING` relaxes this constraint so it can use internal routing resources. This can add clock skew that could be a problem with related clocks, but in our case it won't. Let's look at the timing summary.

Using the asynchronous signal directly

Set `ASYNC_BUTTON` in counting buttons to `NOCLOCK`. We'll leave the clock constraints to represent an asynchronous signal coming in on the input. We'll use BTNC as an input directly to a register clocked on `clk`.

After building the design, look again at the clock interactions and you'll see the BTNC to the `clk` path is still unsafe:



Figure 3.16 – Asynchronous BTNC input

Investigating further we see that we have a 0 ns timing requirement:

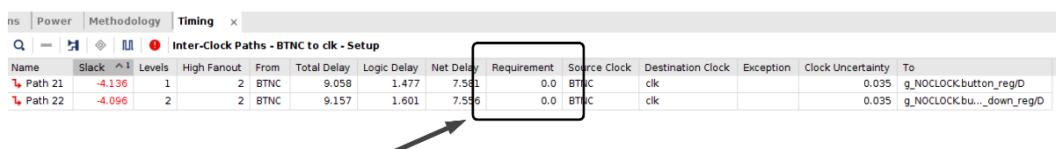


Figure 3.17 – Timing requirements

This is a sign that the signals originate from asynchronous clocks. The way the timing analysis works is that when you have clocks of different frequencies, the tool will walk the edges through each other to find the worst-case alignment for timing. In this case, Vivado has found an alignment where the signal needs to meet a 0-nanosecond requirement.

You can also bring up the clock interaction window:

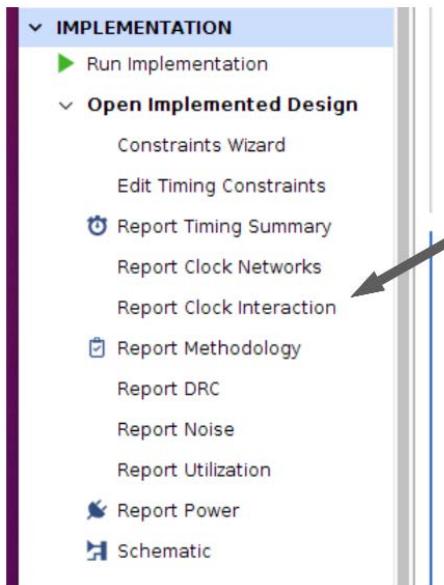


Figure 3.18 – Clock interaction

The clock interaction pane shows us the paths from BTNC to `clk` are timed, but they are unsafe.

If you try this on the board, you'll see that the seven-segment display will increment on every push of the center button. It counts up in hexadecimal. You can also reset the design by using the red reset button on the board.

The problem with push buttons

The way we finally get everything to work is to properly synchronize our BTNC signal. Because the BTNC signal is being pressed on human timescales, we can consider it a static signal except for when the button is depressed and released. When dealing with a signal like this, we can use a two-stage synchronizer plus an additional FF for edge detection:



Figure 3.19 – Button down synchronizer

If we look at *Figure 3.19*, we can see that on clock cycle 0, it's possible we may violate timing on FF0. This is represented by an X in the timing diagram, but it's in fact when FF0 has gone metastable. Metastability is when the output Q of an FF is in an indeterminate state because setup or hold times have been violated. That is, FF1 will recognize it as a 0 or a 1, but it's not guaranteed which one. This is why FF0 drives one and only one FF. If it were to drive two or more FF, it is possible that each FF would see a different value. By the time FF1 has output its Q value, it is safe to use in a design on the clk clock domain, but because we cannot use FF0 and we need to detect an edge, we need FF2 to hold the old value of BTNC:

```
logic [2:0] button_sync;
always @(posedge clk) begin
    button_sync <= button_sync << 1 | BTNC;
```

```

if (button_sync[2:1] == 2'b01) button_down <= '1;
else button_down <= '0;
end

```

If you recall, `<<` is the left shift operator, so we can view `button_sync <= button_sync << 1 | BTNC;` as a shift register where bit 0 is the asynchronous FF, which is why it's not used in the comparison operation.

Important note

Metastability is all about statistics and mean time between failure analysis. It is still possible for FF1 or FF2 to propagate the metastability of FF0; however, it is statistically unlikely. It's important to understand this point as it's often a question asked in interviews.

There is still one problem we need to address.

Designing a safe implementation

Make sure to comment out the BTNC constraints added in the previous section and set `ASYNC_BUTTON = "SAFE"`. After running the design, open the timing summary:

Design Timing Summary			
Setup	Hold	Pulse Width	
Worst Negative Slack (WNS): 6.498 ns	Worst Hold Slack (WHS): 0.169 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints:	0
Total Number of Endpoints: 168	Total Number of Endpoints: 168	Total Number of Endpoints:	125

All user specified timing constraints are met.

Figure 3.20 – Timing for properly synchronized design

If you look at the clock interaction pane, you'll see everything is safe. We can now try it on the board.

One thing you may notice is that you may still occasionally see some counting weirdness. We have forgotten to take one thing into account. We are dealing with mechanical switches and mechanical switches suffer from a phenomenon called bouncing:



Figure 3.21 – Undebounced switch

Figure 3.19 shows a condensed view of an electromechanical switch and how it will oscillate for some period of time before settling. What we need to do is create a circuit that waits for a period of time after it detects the switch being depressed, resetting itself if it detects a bounce:

```
always @(posedge clk) begin
    button_down <= '0;
    button_sync <= button_sync << 1 | BTNC;
    if (button_sync[2:1] == 2'b01) counter_en <= '1;
    else if (~button_sync[1])      counter_en <= '0;
    if (counter_en) begin
        counter <= counter + 1'b1;
        if (&counter) begin
            counter_en <= '0;
            counter     <= '0;
            button_down <= '1;
        end
    end
end
```

The preceding circuit starts a timer when the button is detected being pushed, that is, the falling edge. It then waits 256 clock cycles to make sure the switch doesn't bounce, resetting the counter if it does and waiting to detect the next edge.

Try this on the board and you'll see no more weird counting issues.

The final stats for the design are 68 LUTs, 133 FF, and a **Worst Negative Slack (WNS)** of 6.732. We have a lot of room to increase the clock speed if so desired. We were targeting a 100 MHz clock (with a 10 nanosecond period). With a WNS slack of 6.732, we could decrease our clock period to 10-6.732, which would be about 300 MHz clock.

When analyzing timing in a design, aside from constraints, we have a few other metrics:

- **WNS:** The longest path slack in the design in nanoseconds. If a design is violating timing, it will be negative.
- **Total Negative Slack (TNS):** The sum total of all violating paths in the design (0 if WNS is positive).
- **Worst Hold Slack (WHS):** Must be positive or 0 for a functional design.
- **Total Hold Slack (THS):** Must be positive or 0 for a functional design.

Now, let's see how we can switch to decimal representation.

Switching to decimal representation

The counter up until now counts in hexadecimal. I've been working with binary/hexadecimal numbers for 30 years, so I'm fairly used to it. However, most people are more comfortable with decimal numbers. As is usually the case, there are multiple ways of accomplishing a task. We can count in binary and convert to decimal, or simply count in decimal.

For this design, I decided to count in decimal. If we are representing decimal on eight seven-segment displays, the maximum number we could represent is 99,999,999. If we counted in binary and converted, we'd get $2^{32} = 4,294,967,296$, and that wouldn't fit on the display (not that you would spend the rest of eternity pushing buttons to count that high):

```
// Decimal increment function
function [NUM_SEGMENTS-1:0] [3:0] dec_inc;
    input [NUM_SEGMENTS-1:0] [3:0] din;
    bit [3:0] next_val;
    bit carry_in;
    carry_in = '1;
    for (int i = 0; i < NUM_SEGMENTS; i++) begin
        next_val = din[i] + carry_in;
        if (next_val > 9) begin
            dec_inc[i] = '0;
            carry_in = '1;
        end else begin
            dec_inc[i] = next_val;
            carry_in = '0;
        end
    end // for (int i = 0; i < NUM_SEGMENTS; i++)
endfunction // dec_inc
```

The preceding function accepts **Binary Coded Decimal (BCD)** encoded data. The `for` loop iterates over each digit counting up until it reaches 10, in which case it resets that digit to 0 and feeds a carry into the next digit.

Change the mode from HEX to DEC and build.

The final stats for the design are 97 LUTs, 133 FF, and a WNS of 1.490. There is a cost for a more readable display: approximately 50% more LUTs and about 5 nanoseconds of delay, however, we still easily meet timing at 100 MHz.

Introducing the ILA

Xilinx provides a very capable on-chip debugging solution within Vivado called an **Integrated Logic Analyzer (ILA)**. An ILA gives us the ability to add a logic analyzer that we can insert into our design. The simplest way of approaching an ILA is to start by marking signals for debugging. We can do this by adding the `mark_debug` attribute as follows:

```
(* mark_debug = "true" *) logic button_down;
(* ASYNC_REG = "TRUE", mark_debug = "true" *) logic [2:0]
button_sync;
```

We apply attributes to signals by using the SystemVerilog comment style (* *). You can see how multiple attributes can be applied by looking at `button_sync`.

Both the `SAFE` and `DEBOUNCE` circuits have `mark_debug` already set. Pick `SAFE` and let's look at the steps to set up and run an ILA.

Marking signals for debugging

If you take a look at `CH3/counting_buttons/hdl/counting_buttons.sv`, you'll see a few signals marked for debugging. Don't worry about getting everything, but the more you can add now that are of interest, the better chance of finding them after synthesis:

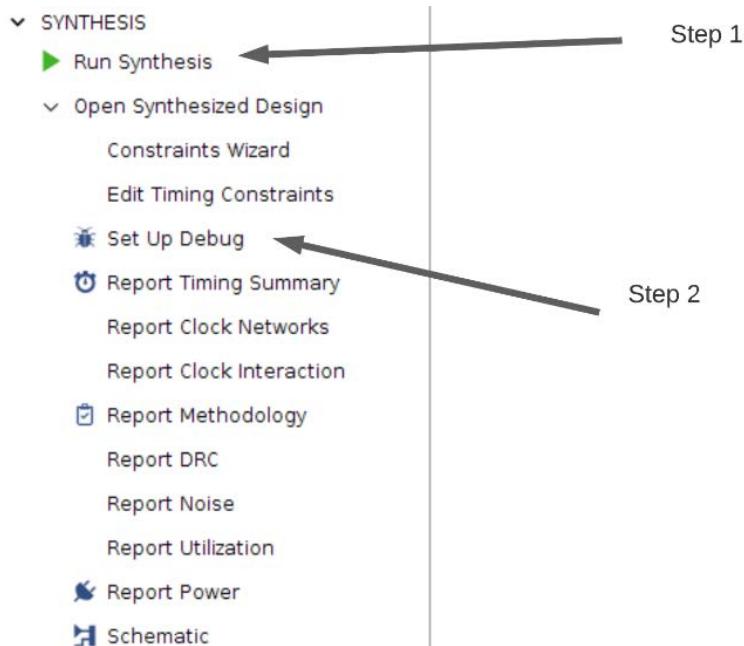


Figure 3.22 – Setting up an ILA

You need to follow these steps:

1. Select **Run Synthesis** and **Open Synthesized Design**. Usually we combine this step with generating a bitstream, but we need to run synthesis by itself so we can set up debugging.
2. Select **Set Up Debug...**:



Figure 3.23 – Set Up Debug

3. A window will pop up. Select **Next**.
4. Select **Continue Debugging** (since we have already added signals via `mark_debug`). I've added the synchronization registers and the button down output. If you look at the `DEBOUNCE` version, you'll see the counter as well.

5. Select the **Find Nets to Add...** button. Click **OK**. This will show you all the nets that you could probe in the design:



Figure 3.24 – Adding nets

6. We have enough to look at, so simply hit **Cancel**.

Feel free to add some more signals if you'd like, or experiment later. The thing to remember is that you have only the block RAM in the FPGA for internal storage of waves, so the more signals you add, potentially the less sample depth is available. We have a large device, so there's nothing to worry about right now.

7. Back on the setup debug screen, hit **Next**.

Leave the ILA core options at the default. A sample depth of 1024 is fine for now. Our clock speed is slow enough that we don't need pipe stages. An ILA has some advanced debugging features, but we won't get into them here.

Programming the device

As is always the case, we need to build the device so we can see our results. Select **Finish** and generate the bitstream:

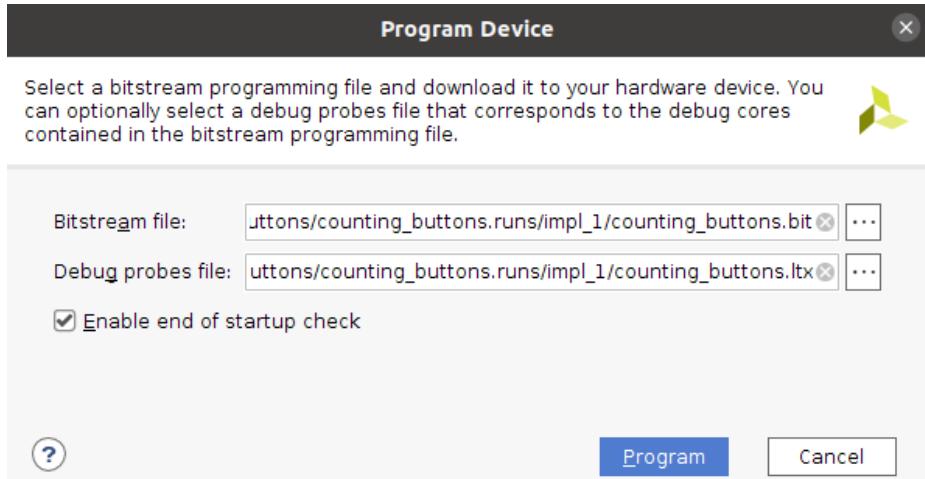


Figure 3.25 – Program Device

You'll notice that the debug probes are now being set up. Select **Program**.

You'll now have the ILA view open:

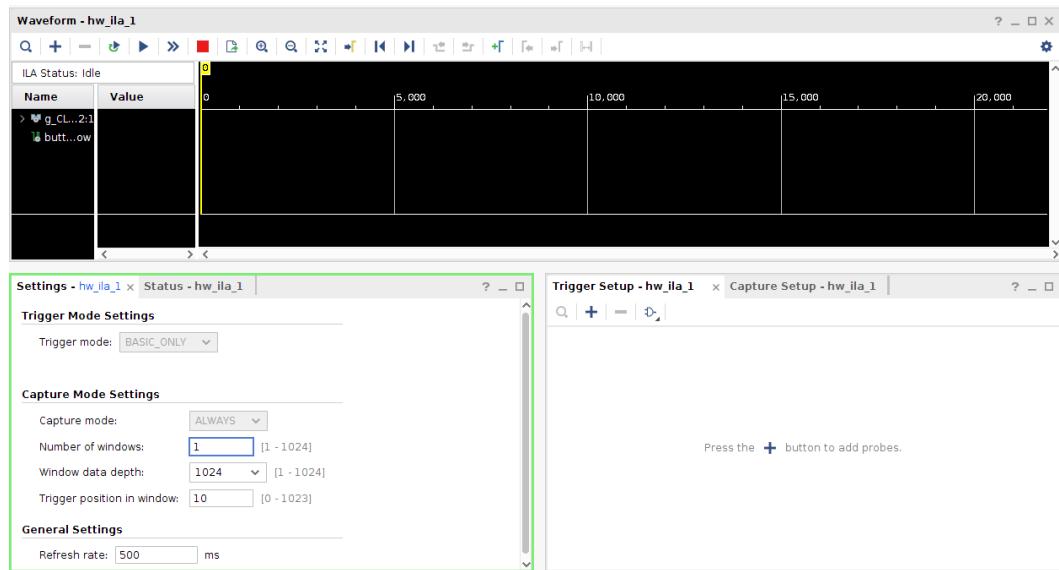


Figure 3.26 – ILA view

The **Settings** pane allows you to limit the data depth. Sometimes you may select a large storage space in a build but want to limit what you capture in a specific run and you can do that here. The trigger position sets how many samples are captured prior to the trigger. In our case, we are looking at 10 samples prior to our trigger, and the rest are after.

In the trigger setup, we'll add something to trigger on:



Figure 3.27 – Trigger Setup

I've selected to trigger whenever bit 1 of button_sync goes high:



Figure 3.28 – Run a trigger

We have a few choices for capturing signals in the ILA. **Immediate Trigger** (>>) will capture the signals in their current state. **Auto Trigger** (the right arrow with the green circle) will rearm the trigger automatically after firing. **Single Trigger** sets us up for capturing a single event. Finally, if you are unable to fire your trigger, you can stop it and set up another with the **Stop Trigger** button.

Try **Single Trigger**, then push the center button and you should see what I have in *Figure 3.28*. I was unable to capture an anomaly, but this at least shows you how to use an ILA for debugging problems on board.

What about simulation?

In this chapter, we took advantage of the capabilities of lab debug rather than running simulations. This is not something I would normally do. However, it would take too much time and effort to write a model to test the seven-segment display and also model a bouncing switch. It's an FPGA after all, and we can program and test as many times as we want. It's not a route I usually go down, but for this project it worked well.

Deep dive on synchronization

In project 2, we dipped our toes into synchronizing a signal from an external source. In later designs, we'll be interfacing between multiple clock domains. For instance, in *Chapter 5, FPGA Resources and How to Use Them*, we'll be interfacing between our main logic and a DDR controller running on a different clock domain.

Why use multiple clocks?

There are several clocking considerations when architecting your FPGA design. Sometimes you are forced to use a given clock for an interface. For example, if you are designing something that interfaces to 10G Ethernet, somewhere in your design will be a multiple of 156.25 MHz or 322.27 MHz depending on if you are interfacing with the PCS or PMA layer. This is because data must be driven out at this frequency and arrives at this frequency.

Other times, you may be looking for high performance or lower power. Increasing your clock speed can increase your throughput or calculations per second if you need data fast or perform lots of operations. Running a faster clock costs more in terms of power. If you are designing something that needs to operate in a lower power environment, you still may need to receive data at a faster rate, but save power by performing operations slower.

Two-stage synchronizer

The heart of synchronization is a two-stage synchronizer. When we define the two stages on the destination clock domain, we apply the ASYNC_REG attribute to the FF. This attribute tells Vivado that these registers are used for synchronizing and should be placed as close together as possible. A simple two-stage synchronizer looks like this:

```
(* ASYNC_REG = "TRUE" *) logic [1:0] sync;
always @(posedge dst_clk) sync <= sync << 1 | async;
```

This creates two flops on the `dst_clk` domain and tells Vivado to handle them as such.

Synchronizing control signals

Often, you may have slower speed status signals that are coming from one clock domain to another. These signals will toggle infrequently and either you handshake between both sides of the interface or the design is guaranteed to change infrequently enough that you don't need to worry about handshaking.

For this type of interface, we can use a toggle synchronizer. A toggle synchronizer toggles the signal crossing clock domains and then generates a pulse when an edge is detected on the synchronized domain. You can see why we need to make sure that we do not send toggles across faster than we can generate pulses on the receiving side. One way to prevent this is to send a similar toggle signal back as an acknowledgement:

```
logic async_toggle;
(* ASYNC_REG = "TRUE" *) logic [2:0] sync;
logic sync_pulse;
always @ (posedge src_clk)
  if (ctrl_in) async_toggle <= ~async_toggle;
always @ (posedge dst_clk) sync <= sync << 1 | async_toggle;
assign sync_pulse = ^sync[2:1];
```

In the following figure, you can see an illustration of a toggle synchronizer waveform:

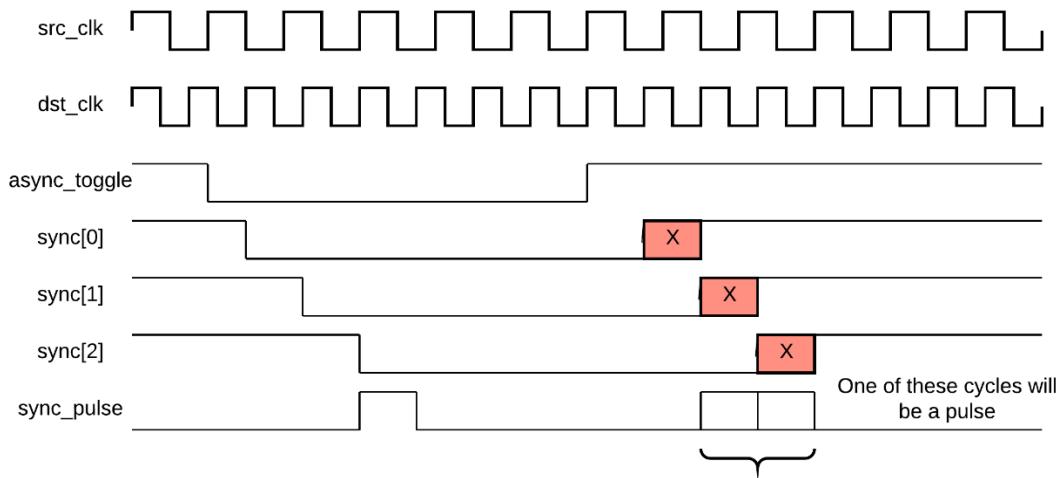


Figure 3.29 – Toggle synchronizer waveform

You'll note in the waveform that the rising edge of `async_toggle` may violate timing on `sync[0]`. This is represented by an X. Although `sync[0]` may go metastable, `sync[1]` will statistically register a 0 or 1 and that will safely pass to `sync[2]`. However, since it is statistical, we don't know which of the two cycles highlighted will generate the pulse on `dst_clk`.

Passing data

If we have data that meets the criteria of the preceding synchronizer (that is, the data will be stable for the amount of time to be captured on the `dst_clk`), we can pass it along with the control signal. The source data is captured and held on the `src_clk` for the time necessary for synchronizing. If the clock relationships are known, this can be done by waiting for a long enough period of time, or we can pass an acknowledge signal back from `dst_clk` when the sync pulse is received:

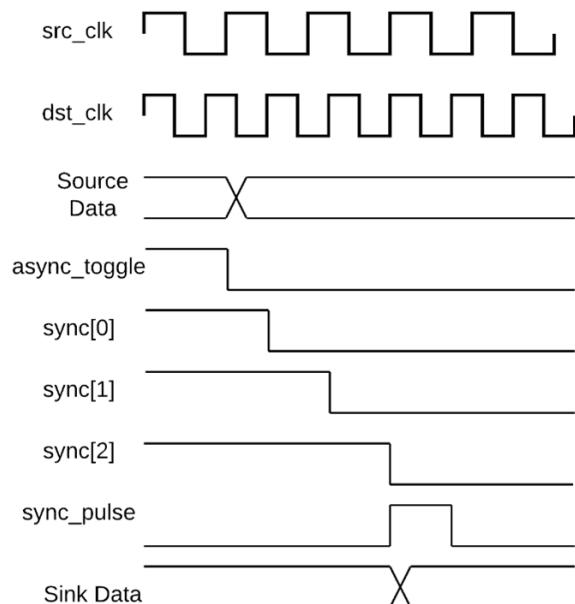


Figure 3.30 – Passing data across clock domains

We do this by registering it on the same cycle we toggle the signal crossing to the synchronizer:

```

logic async_toggle;
logic [31:0] async_data;
(* ASYNC_REG = "TRUE" *) logic [2:0] sync;
logic sync_pulse;
logic [31:0] sync_data;
always @(posedge src_clk)
  if (ctrl_in) begin
    async_toggle <= ~async_toggle;
  end
end

```

```
    async_data <= ...;
end
always @(posedge dst_clk) begin
    sync <= sync << 1 | async_toggle;
    if (sync_pulse) sync_data <= async_data;
end
assign sync_pulse = ^sync[2:1];
```

This code will safely transfer the data; however, we need to provide one constraint to Vivado:

```
set_max_delay -datapath_only \
[2*[get_property PERIOD [get_clocks dst_clk]]] \
-from [get_cells async_data*] -to [get_cells sync_data*]
```

This constrains the path from async data to sync data to be two destination clock cycles or less. The reason is that we know the two-stage synchronizer will take two or three clocks for the pulse to appear from when the toggle signal switches. We apply this constraint to all bits on the bus.

We can add an acknowledge signal if we need handshaking to let the sender know data has reached the destination. It would be synchronized the same way.

FIFOs are a third common way of synchronizing that we'll discuss in *Chapter 5, FPGA Resources and How to Use Them*.

Summary

In this chapter, we introduced sequential elements, how to store data, and how to write constraints for these elements for Vivado. Along with learning how to write combinational logic, we now have the fundamentals to create just about any design. We've developed a better way of displaying information and even made a more human-readable version of it. It's important to look at what you have accomplished thus far. You've seen how to handle external inputs operating asynchronously to the system clock. You've interfaced to a more sophisticated output display. We've also debugged on the board using an ILA. This should give you the confidence to experiment a bit and the challenge question will allow you to do just that.

In the next chapter, we'll build on the lessons and skills we learned in this chapter by building something more substantial: a calculator.

Questions

1. It's best to use blocking assignments in sequential blocks and non-blocking in combinational blocks.
 - a) True
 - b) False
2. It is best to reset all sequential elements in a design.
 - a) True
 - b) False
3. What are the most common ways of synchronizing?
 - a) `always @(posedge signal)`
 - b) `always @(negedge signal)`
 - c) FIFO or a two-stage synchronizer with or without data.
 - d) Synchronizers... who needs synchronizers?
4. When would we use `always @ (posedge clk)` rather than `always_ff @ (posedge clk)`?
 - a) When we get tired of typing.
 - b) When we need to use an initial statement to preload the register.
 - c) When we need to reset the register either synchronously or asynchronously.
5. When do we need to add debouncing logic?
 - a) When we cross clock domains
 - b) Whenever we send data from one FF to another
 - c) When we are dealing with electromechanical buttons or switches

Challenge

In *Chapter 2, with Combinational Logic*, we created a design that could perform some simple operations and display data in binary on the LEDs:

1. Modify the code to use the seven-segment display module to output the data rather than the LEDs, or in addition to the LEDs. Display in either hex or decimal. If you go the decimal route, you'll either need to use the function for adding in decimal or do a conversion before displaying it.
2. Run it on the board to verify its operation.

Further reading

Please refer to the following for more information:

- To read further about the design constraints that can be applied within Vivado:
<https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0004-vivado-applying-design-constraints-hub.html>
- More information on Vivado properties: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug912-vivado-properties.pdf

4

Let's Build a Calculator

In this chapter, we are going to take our SystemVerilog knowledge of combinational logic and sequential elements to discuss state machine design. We'll look at the classic state machine designs and develop a traffic light controller, a staple of **Electrical Engineering (EE)** projects.

We've built a controller for a 7-segment display that we can use to show numerical values and we know how to handle button and switch inputs safely. Now, we'll take this knowledge and show how we can define a state machine to keep track of the calculation we want to perform and develop our first truly useful design, a simple calculator capable of entering two 16-bit numbers and adding, subtracting, multiplying, and dividing them, placing the output on the 7-segment display.

Once you've completed this chapter, you should be able to construct simple state machines, use simple state machines to implement algorithms, and understand the basics of computer math.

In this chapter, we are going to cover the following main topics:

- Introducing the main types of state machines in a design: Mealy and Moore
- Implementing a traffic light controller using a state machine
- Designing a simple state machine for our calculator
- Designing and simulating an integer divider
- Running it on the board

Technical requirements

The technical requirements for this chapter are the same as those for *Chapter 1, Introduction to FPGA Architectures and Xilinx Vivado*.

To follow along with the examples and the project, you can find the code files for this chapter at the following repository on GitHub: <https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH4>.

Implementing our first state machine

In general, a **state machine** takes in a number of events and, based on the events, moves through a set of states that can produce one or more outputs. A state machine can be quite simple or extremely complex. In the previous chapter, we designed a simple circuit to control our 7-segment display. The 7-segment controller contained two counters that cycled a zero through the cathodes and presented the anode data for each digit. We could have written a state machine to handle this; however, it was easier to write it the way we did.

Before we dive into our calculator project, we need to go over the two ways of coding state machines and the two traditional state machine implementations.

Writing a purely sequential state machine

The first way of coding a state machine is to write it in a single always block driven by a clock.

This kind of state machine would look something like this:

```
enum bit {IDLE, DATA} state;
initial state = IDLE; // Define initial state
always @(posedge clk) begin
    case (state)
```

```

IDLE: begin
    dout_en <= '0;
    if (start) begin
        dout_en <= '1;
        state <= DATA;
    end
end
DATA: begin
    dout_en <= '1;
    if (done) begin
        dout_en <= '0;
        state <= IDLE;
    end
end
endcase
end

```

You can see from the preceding code that we are defining our states using an `enum` register called `state`. We then assign an `initial` value, the value that will be loaded when the FPGA starts up.

Tip

Use `enum` types for state machine definitions. Good names help convey design intent and, when simulating the waves, it will display state names rather than numbers.

The main code is in the `always` block. We use a `case` statement to define our state machine states, `IDLE` and `DATA`. There are two inputs to the state machine, `start` and `done`, and one output, `dout_en`. This type of code is concise and fast since there is a full clock cycle for all outputs from the state machine. It does have a potential disadvantage in that every output will be registered. In some cases, you may want an output to occur as soon as input changes and this is not possible when you write the state machine this way.

The previous version of the state machine simply used a single state variable. Another way of coding a state machine is by splitting the state variable into current and next states. Let's look more in-depth in the next section.

Splitting combination and sequential logic in a state machine

We can break up the state machine into a combinational portion based on the current state that generated the next state to be registered:

```
enum bit {IDLE, DATA} current_state, next_state;
initial current_state = IDLE; // Define initial state
always @(posedge clk) current_state <= next_state;
always_comb begin
    current_state = next_state; // avoid a latch
    dout_en = '0; // avoid a latch
    case (current_state)
        IDLE: begin
            if (start) begin
                dout_en = '1;
                next_state = DATA;
            end
        end
        DATA: begin
            dout_en = '1;
            if (done) begin
                dout_en = '0;
                next_state = IDLE;
            end
        end
    endcase
end
```

Looking at the preceding code, we can see there are a lot of similarities. There are some functional differences in that `dout_en` is组合逻辑地生成的。To make this equivalent, we would need to register `dout_en`.

Either way is acceptable for constructing a state machine. In fact, both have advantages and I tend to mix up their usage depending on the situation. Before we discuss the two classic state machine types, let's define what we want to accomplish in this project.

Designing a calculator interface

Let's look again at what we have available for inputs. Both *Basys 3* and *Nexys A7* have an array of 16 LEDs, 16 switches, and a 5-pushbutton array:

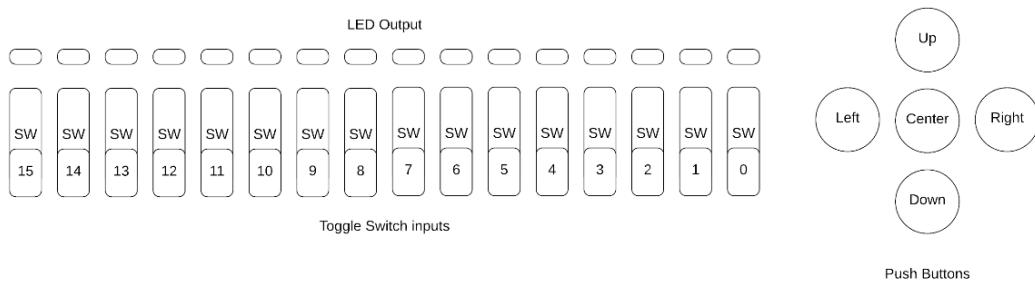


Figure 4.1 – FPGA board buttons and switches

We can use the switches to input a 16-bit value in either hex or **Binary Coded Decimal (BCD)**. We have 5 buttons available. The functions that we'll perform are as follows:

- Left: A+B
- Right: A-B
- Up: A*B
- Center: =
- Down: Clear

Using the 7-segment displays, we can handle the result output and, using an LED, we can show the sign bit.

With our input/output (I/O) designed, we can focus on a state machine:

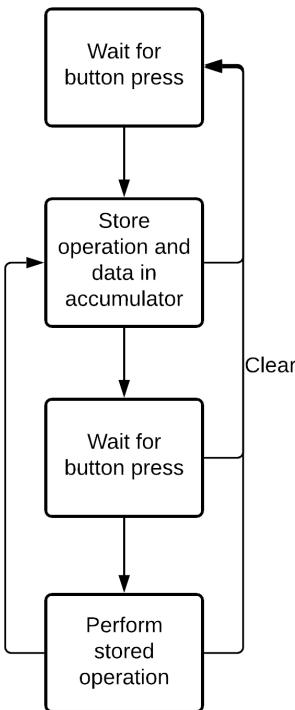


Figure 4.2 – Ideal calculator state machine

Now, let's discuss the two classic state machine types and how we could implement the calculator using either one.

Designing a Moore state machine

In 1956, Edward F. Moore presented the concept of a state machine whose outputs are governed strictly by the state of the machine in his paper, "*Gedanken-experiments on Sequential Machines*."

What this means in practice is that the current state generates the outputs. The inputs only govern the next state logic. In general, this type of state machine can reach high clock speeds since the outputs have a full clock cycle to affect the design. These state machines can be large since decision logic doesn't affect the outputs directly, but rather leads to new states, increasing the state space.

You can find the following code in the CH4/hdl/calculator_moore.sv folder:

```
typedef enum bit [2:0]
{
    IDLE,
    WAIT4BUTTON,
    ADD,
    SUB,
    MULT,
} state_t;
```

Looking at the Moore version of the calculator, we can see individual states for each operation. As we stated previously, we have a full clock cycle for each operation. This could be an advantage if we were pushing speed in our technology, but, as we saw previously, the add operation has plenty of time:

```
IDLE: begin
    accumulator <= '0;
    last_op     <= buttons; // operation to perform
    accumulator <= switch;
    if (start) state <= buttons[DOWN] ? IDLE : WAIT4BUTTON;
end
WAIT4BUTTON: begin
    op_store     <= buttons;
    if (start) begin
        case (1'b1)
            last_op[UP] :      state <= MULT;
            last_op[DOWN] :   state <= IDLE;
            last_op[LEFT] :   state <= ADD;
            last_op[RIGHT] :  state <= SUB;
            default:          state <= WAIT4BUTTON;
        endcase // case (1'b1)
    end else state <= WAIT4BUTTON;
end
MULT: begin
    last_op     <= op_store; // Store our last operation
    accumulator <= accumulator * switch;
    state       <= WAIT4BUTTON;
end
```

We won't go through the entire state machine here. Please take a few minutes and peruse the following link: https://github.com/PacktPublishing/Learn-FPGA-Programming/blob/master/CH4/hdl/calculator_moore.sv.

Once a button has been pushed, the switch values and the operation are loaded into `accumulator` and `last_op`. When the next button is pushed, we execute the previous operation on the accumulator and the second switch values. This behaves like a pocket calculator, where you would push $X + Y =$ and the result would appear on the screen. Also, you can chain operations, $X + Y - Z =$, for example.

If you look at the state machine design, you'll see that inputs only affect next state values and that operations on data take a state.

Let's examine how this would look in a Mealy state machine design.

Implementing a Mealy state machine

In 1955, George H. Mealy introduced the concept of a state machine whose outputs are determined by the current state and inputs. We can see an example of our calculator state machine implemented this way.

You can find the following code in the `CH4/hdl/calculator_mealy.sv` folder:

```
typedef enum bit
{
    IDLE,
    WAIT4BUTTON
} state_t;
```

From the state definitions, you can see that the state space is greatly reduced. We've gone from five states to two. How are we able to accomplish this?

```
case (state)
IDLE: begin
    last_op     <= buttons; // operation to perform
    accumulator <= switch;
    if (start) state <= buttons [DOWN] ? IDLE : WAIT4BUTTON;
end
WAIT4BUTTON: begin
    if (start) begin
        last_op <= buttons; // Store our last operation
```

```

    case (1'b1)
        last_op[UP] : accumulator <= accumulator * switch;
        last_op[DOWN] : state      <= IDLE;
        last_op[LEFT] : accumulator <= accumulator + switch;
        last_op[RIGHT] : accumulator <= accumulator - switch;
        default:         state      <= WAIT4BUTTON;
    endcase // case (1'b1)
    end else state <= WAIT4BUTTON;
end
endcase // case (state)

```

We no longer have the limitation of using an input to only change states. Now we are able to use the inputs to affect the operations directly. In the Moore state machine, we had states dedicated to our operations. Now we simply stay in the WAIT4BUTTON state.

Knowing about Mealy and Moore state machine design is mostly academic. If you decide to pursue a career in FPGAs, you'll probably be asked what the difference is in an interview, but mostly because, when applying for a first job in a field, there's not much beyond academics that us engineers know to ask. What you will really need going forward is some practical advice.

Practical state machine design

The reality of state machine design is that you should not worry about the formality of the code and whether it is Mealy or Moore. Use what you are comfortable with and fits the solution you are working toward. In practice, I tend to use a mix of styles based upon the task I'm working on. In some cases, it might be a Moore type, or it may be a hybrid type using a split current state/next state design.

With the basics of state machine design under our belt, let's look at a practical project whose heart is a state machine.

Project 3 – Building a simple calculator

Now that we've gone over state machine basics and showed the core of our calculator, we need to look at how we'll actually implement the calculator. The first issue that will come up is how do we store our data in the design. Previously, we used BCD when we were incrementing our values. There was a simple solution presented for the BCD incrementor.

If we wanted to keep the internal data as BCD, we would need to develop a BCD adder, subtractor, and multiplier. This is a more complicated option than a simple incrementor. Alternatively, we can explore the possibility of keeping our internal representation as binary, but convert to decimal to display. This has the added advantage that we can use the SystemVerilog add, subtract, and multiply operators as-is on binary representation and then create a conversion function.

The project files can be found in the following locations:

- Nexys A7: CH4/build/calculator/calculator.xpr
- Basys 3: CH4/build/calculator/calculator_basys3.xpr

Before we go down that route, let's consider SystemVerilog packages.

Packaging for reuse

SystemVerilog provides the capability of creating packages to encapsulate code that we want to reuse among multiple modules. It also provides a convenient way of reusing code for multiple applications.

You can find the following code in the CH4/hdl/calculator_pkg.sv folder:

```
 `ifndef NUM_SEGMENTS
`define NUM_SEGMENTS 8
`endif
`ifndef _CALCULATOR_PKG
`define _CALCULATOR_PKG
package calculator_pkg;
    localparam NUM_SEGMENTS = `NUM_SEGMENTS;
    localparam UP          = 3'd0;
    localparam DOWN        = 3'd1;
```

We create a package like we would an empty module – package <package name>;.

A **package** can contain parameters, functions, tasks, and user-defined types. In our case, we define localparams to make identifying the buttons easier and a function to convert the binary representation to BCD:

```
function bit [NUM_SEGMENTS-1:0] [3:0] bin_to_bcd;
    // we want to support either 4 or 8 segments
    input [31:0] bin_in;
    bit [NUM_SEGMENTS*4-1:0] shifted;
```

```

shifted      = {30'b0, bin_in[31:30]};
for (int i = 29; i >= 1; i--) begin
    shifted = shifted << 1 | bin_in[i];
    for (int j = 0; j < NUM_SEGMENTS; j++) begin
        if (shifted[j*4+:4] > 4) shifted[j*4+:4] += 3;
    end
end
shifted = shifted << 1 | bin_in[0];
for (int i = 0; i < NUM_SEGMENTS; i++) begin
    bin_to_bcd[i] = shifted[4*i+:4];
end
endfunction // bin_to_bcd
endpackage // calculator_pkg
`endif

```

You can see that there is some complexity to the conversion function. It's this type of thing that worries me when designing. The multiple `for` loops need to be unrolled to create the logic that will perform the actual task in the FPGA. Even at 100 MHz, this will be a challenge.

Tip

If you are using non-project flow, rather than project flow, like we are, you should add the following:

```

`ifndef _CALCULATOR_PKG
#define _CALCULATOR_PKG

```

This will prevent the package from being redefined, which can cause warnings or errors.

There is one major limitation with SystemVerilog packages. You can't pass in parameters. This is why you see I had to define `NUM_SEGMENTS` within the package itself.

We can get around this by using `define`:

```

`ifndef NUM_SEGMENTS
#define NUM_SEGMENTS 8
`endif

```

Then we can assign the definition to `localparam`:

```
localparam NUM_SEGMENTS = `NUM_SEGMENTS;
```

Within non-project mode, you can override the parameter within your TCL scripts. Within project mode, you can override the defines from within the **PROJECT MANAGER** settings. Now you'll need to do it within multiple areas as we have done before for synthesis and simulation.

Coding the top level

Take a look at the top-level calculator module.

You can find the following code in the CH4/hdl/calculator_top.sv folder:

```
`ifndef NUM_SEGMENTS
`define NUM_SEGMENTS 8
`endif

module calculator_top
#(
    parameter BITS          = 32,
    parameter NUM_SEGMENTS = `NUM_SEGMENTS,
    parameter SM_TYPE       = "MEALY" // MEALY or MOORE
) (
    input wire                  clk,
    input wire [15:0]           SW,
    input wire [4:0]            buttons,
    output logic [NUM_SEGMENTS-1:0] anode,
    output logic [7:0]          cathode
);
```

We can select the internal data storage size by setting the `BITS` parameter. We can also set the number of segments (as we discussed in the *Packages* section) and the state machine type.

We will use the `seven_segment` controller that we developed in *Chapter 3, Counting Button Presses*, as well as the button debouncing machine:

```
generate
    if (USE_PLL == "TRUE") begin : g_USE_PLL
        sys_pll u_sys_pll
        (
            .clk_in1 (clk),
```

```

.clk_out1 (clk_50)
);
end else begin : g_NO_PLL
    assign clk_50 = clk;
end
endgenerate

```

You'll notice that there is a PLL in the design and I've created an internal clock, `clk_50`. The next section will cover PLL generation. For our first run, we'll be using `USE_PLL = "FALSE"`:



Figure 4.3 – Mealy calculator design, 100 MHz

From this run, we can see that the design doesn't meet timing at 100 MHz. We can look at the failing paths in the timing analyzer and note that the binary to BCD converter doesn't meet the timing requirements:



Figure 4.4 – Mealy timing

The Moore state machine has more time to perform operations, but it likely won't do much for our violating paths as these are through the converter. Try running it again with `SM_TYPE` set to `MOORE`. You'll see that the results are very similar; a few more storage elements in the Moore design and slightly better timing.

Changing frequencies by using a PLL or MMCM

The Artix 7 has **Clock Management Tiles** (CMTs) that are the core of the clocking resources in the devices we are using. The Nexys A7 100T has 6 CMT. The Nexys A7 50T and Basys 3 both have 5 CMTs.

A CMT contains one **Phase Locked Loop (PLL)** and one **Mixed Mode Clock Manager (MMCM)**. PLLs and MMCMs can be used for frequency synthesis, that is, creating a faster or slower clock from an incoming clock:

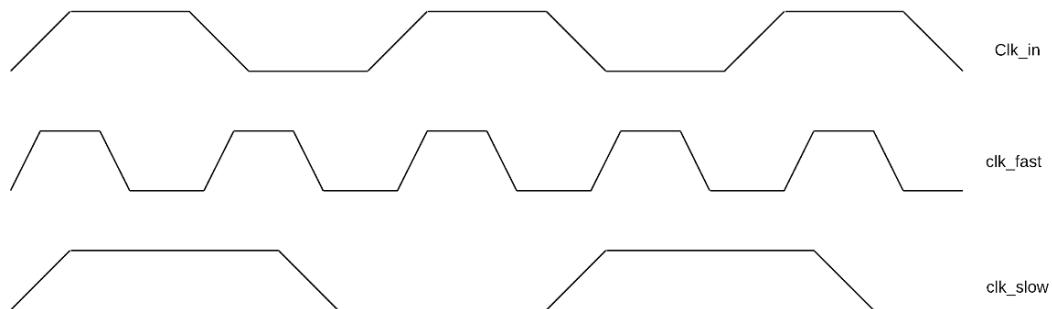


Figure 4.5 – PLL/MMCM clock synthesis

You can see from the preceding diagram that we can take a clock input. For our boards, this is a 100 MHz clock and can generate a faster or slower clock based upon our needs. There are other uses for the CMT that we won't get into in this book because of the limited interfaces our boards have.

Let's build our PLL to reduce our internal clock to 50 MHz so we can meet timing in our design. This will be our introduction to the IP catalog. Xilinx and its partners provide IP for their FPGA devices, some of which are free while others can be licensed. I'd encourage you to look over what's available, especially the free options. Let's see how to do this:

1. Select the IP catalog:

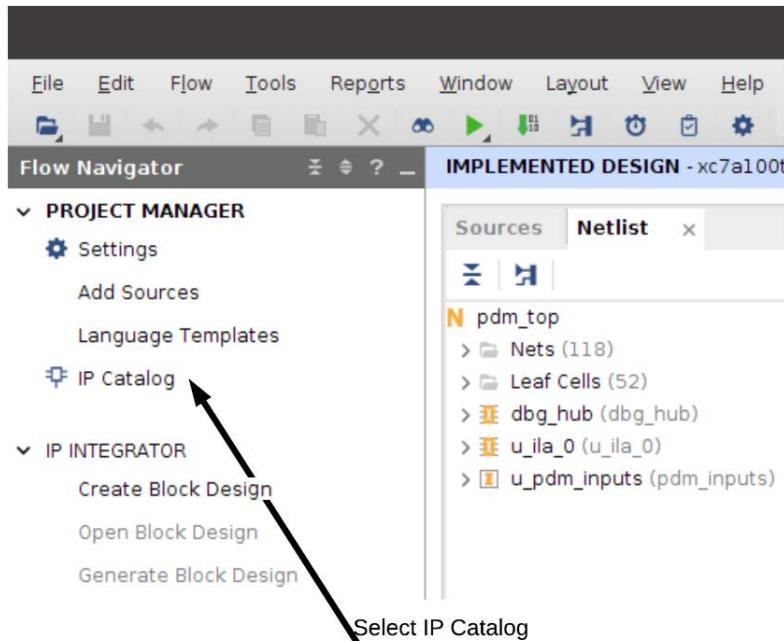


Figure 4.6 – IP Catalog

2. Next, we'll look at the clocking wizard:

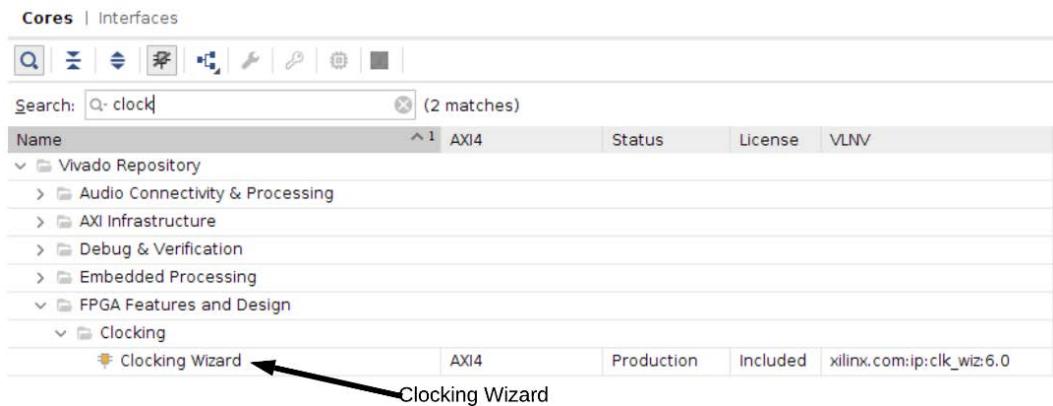


Figure 4.7 – Clocking wizard

3. The first step to creating IP from the wizard is to customize it. Select the **IP Location** button and make sure to specify CH5/build/IP. Then, change the component name to sys_pll. Make sure **CLK_IN1** is set to **sys clock**:

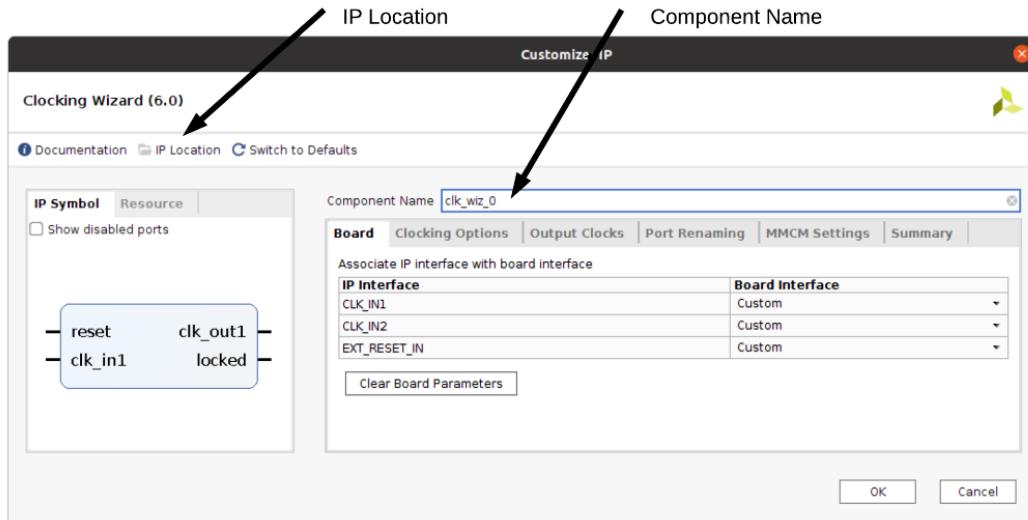


Figure 4.8 – Customize IP

4. Select output cocks and set **clk_out1** to 50 MHz. Since this is an even division of 100 MHz, it will be able to do it exactly. In some cases, it will do its best but the frequency will be off a little. Make sure to deselect **reset** and **locked** as we don't need those outputs:



Figure 4.9 – Output clocks

5. We have already built the FPGA, so you can cancel it. If you wanted to build it, then click **OK** and it will ask to add it to the project and generate the output products:
6. Now, set the **USE_PLL** parameter in **PROJECT MANAGER | Settings | General** to **TRUE**, as we have done previously.
7. Now, set **PLL** to **TRUE** and perform the following build:

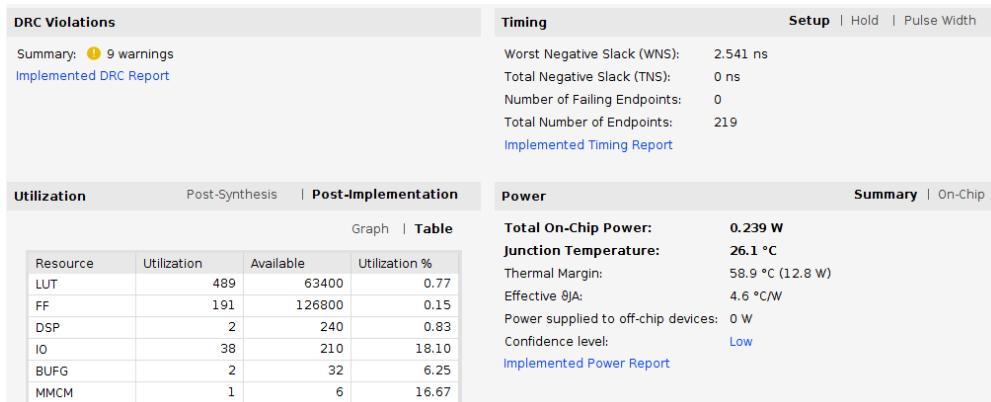


Figure 4.10 – Mealy state machine at 50 MHz

Now that we have lowered the clock speed, the tool doesn't work as hard to meet unreasonable timing. You can see the effects of this by the LUT count going down in this run. Timing is met comfortably. Try it out on the board. Try the add and multiply operations.

Now, try reloading it and subtract something from 0. You'll see the display all lit up with what looks like random values. We didn't add support for negative numbers. This is something to think about for this chapter's challenge question.

One thing to note is that our calculator is fairly simple. It supports addition, subtraction, and multiplication. Why wasn't division included? Think back to when you learned long division. It's a process of shifting and subtracting numbers when there is a large enough value to subtract from. It turns out that add/subtract and multiply operations are very simple relative to division and so are baked into the FPGA fabric. To perform division, we'll need to look at how this can be done.

Investigating the divider

There are two classic algorithms for integer division: **restoring** and **non-restoring**. The difference between the two methods are that when you perform the test subtraction at every pass, you either restore or keep the result negative based on the method used. Non-restoring division has a correction step at the end.

Let's now explore how to implement a non-restoring divider for our calculator.

Building a non-restoring divider state machine

The first step is to create a state diagram for our proposed state machine. We can find the non-restoring algorithm defined in many places on the internet. I've created a proposed state diagram here:

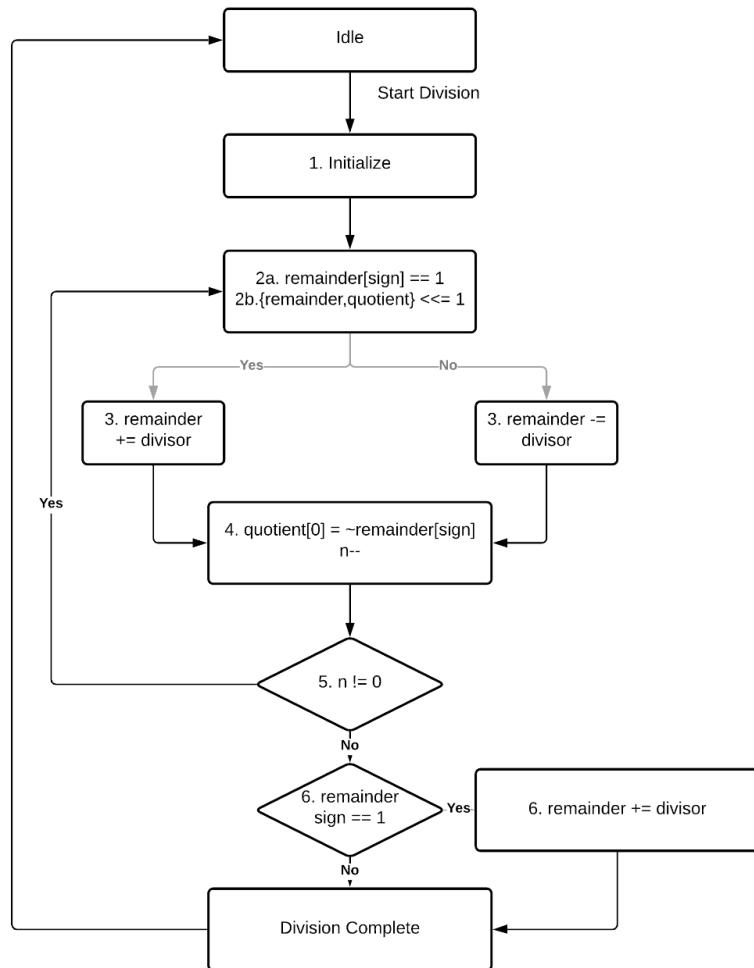


Figure 4.11 – Non-restoring division algorithm

From the preceding state machine diagram, we can see that this is one of the more involved state machine designs we have approached yet. We have multiple tests and branch conditions.

As in many state machines, we begin idle. When we are instructed to divide, we initialize our internal variables. Then, enter the main loop at *Step 2*. We test the remainder sign and then shift our remainder and quotient. If the remainder is negative, we add the divisor, otherwise we subtract it. Then we feed the inverted remainder sign back into the quotient lowest bit and subtract our counter. If the counter is not zero, we return to *Step 2*, otherwise we test the remainder sign one last time and if it is negative, we add the divisor in again, otherwise we are done.

Let's take a look at the divider source code and hopefully it will be clear. You can find the following code in the CH4/hdl/divider_nr.sv folder:

```
module divider_nr #(parameter BITS      = 16)
(
    input wire           clk,
    input wire           start,
    input wire unsigned [BITS-1:0] dividend,
    input wire unsigned [BITS-1:0] divisor,

    output logic          done,
    output logic unsigned [BITS-1:0] quotient,
    output logic unsigned [BITS-1:0] remainder
);
```

The first thing to note is that we've added two signals to control the state machine execution, `start` and `done`. We have a couple of options when handling division. We can either always take the number of clock cycles equal to the number of bits of the dividend, or we can adjust the dividend to remove leading 0's as these will always result in those bits being 0 on the quotient. Since we already developed a leading 1's detector, I chose to use this to speed up the division operation when we can. For our current application, this is not important. However, I tend to design for performance, so I wanted to reuse the module.

When we receive a `start` signal, the divider will begin operation:

```
always @ (posedge clk) begin
    done <= '0;
    case (state)
        IDLE: begin
            if (start) state <= INIT;
        end
        INIT: begin
            state      <= LEFT_SHIFT;
            quotient   <= dividend << (BITS - num_bits_w);
            int_remainder <= '0;
            num_bits     <= num_bits_w;
        end
    end
```

Note that `done` has a default value of '`0`', so it will only go high for as long as we assert it. When we enter the `INIT` state, we'll left-shift the dividend to remove leading 0's and set `num_bits` to know how long the divider will run for. We also define an intermediate remainder result of 0:

```
leading_ones
#(
    .SELECTOR  ("DOWN_FOR"),
    .BITS      (BITS)
)
u_leading_ones
(
    .SW        (dividend),
    .LED       (num_bits_w)
);
```

Currently, `leading_ones` is built as a module for the board and we are using the port names that the boards provided. To make this truly portable, we would rename the ports to be something more in line with our leading ones detector, such as `vector_in` and `ones_position`:

```
LEFT_SHIFT: begin
    {int_remainder, quotient} <= {int_remainder, quotient} << 1;
    if (int_remainder[$left(int_remainder)])
        state   <= ADJ_REMAINDER0;
    else
        state   <= ADJ_REMAINDER1;
end
```

The number of bits is returned from the `leading_ones` function we developed. `LEFT_SHIFT` represents states 2a and 2b in our state diagram. Left-shift our intermediate results. When we do the comparison on the sign bit, `int_remainder[$left(int_remainder)]`, remember that `$left` will return the uppermost bit, or the sign bit of our internal remainder. Also recall that since we are using non-blocking, we can shift and test the previous value in the same clock cycle.

Tip

`if (A)` is a shortcut for `if (A!=0)`.

We have two states for updating the internal remainder, AJD_REMAINDER [2]. These two states represent the third steps in the state diagram:

```
UPDATE_QUOTIENT: begin
    state      <= TEST_N;
    quotient[0] <= ~int_remainder[$left(int_remainder)];
    num_bits     <= num_bits - 1'b1;
end
TEST_N: begin
    if (|num_bits)
        state <= LEFT_SHIFT;
    else
        state <= TEST_REMAINDER1;
end
```

We then update the quotient least significant bit, adjust the number of bits processed, and then test whether we have completed shifting:

```
TEST_REMAINDER1: begin
    if (int_remainder[$left(int_remainder)])
        state <= ADJ_REMAINDER2;
    else
        state <= DIV_DONE;
end
ADJ_REMAINDER2: begin
    state      <= DIV_DONE;
    int_remainder <= int_remainder + divisor;
end
DIV_DONE: begin
    done <= '1;
    state      <= IDLE;
end
```

If we have completed shifting, we then test and update the intermediate remainder if appropriate. Finally, we assert the done signal to signify that the result is ready.

With the design complete, let's try it in a simulation.

Simulating the divider

I've created a divider testbench. This will allow us to verify our division algorithm prior to implementation.

You can find the following code in the CH4/tb/tb_divider_nr.sv folder:

```
for (int i = 0; i < 100; i++) begin
    dividend <= $random;
    divisor  <= $random;
    start     <= '1;
    @(posedge clk);
    start     <= '0;
    while (!done) @(posedge clk);
    repeat (5) @(posedge clk);
end
```

The heart of the code creates a random dividend and divisor. We start and then wait until done goes high before injecting the next values:

```
always @(posedge clk) begin
    if (done &&
        (quotient != dividend/divisor) &&
        (remainder != dividend%divisor)) begin
        $display("failure!");
        $display("quotient: %d", quotient);
        $display("remainder: %d", remainder);
        $display("expected Q: %d", dividend/divisor);
        $display("expected R: %d", dividend%divisor);
        $stop;
    end
end
```

The testing logic checks the quotient and remainder against the values returned by the SystemVerilog division operator (/) and the modulo operator (%).

Important note

SystemVerilog has a division operator. However, it is only synthesizable if it is a power of 2 or returns a fixed value. There is also a modulo operator (%), which will return the remainder of a value divided by another value. This is only synthesizable if it returns a fixed value or the right-hand side is a power of 2.

Sizing the intermediate remainder

Note that I've declared the intermediate remainder result as 1 bit larger than our actual remainder:

```
logic signed [BITS:0] int_remainder;
```

I'm not ashamed to admit that I struggled verifying the design. I had initially declared the internal remainder as BITS - 1 : 0. What I had neglected to consider was that we are dealing with unsigned numbers up to 64,535. Since we are adding or subtracting the divisor from int_remainder, we have a 16-bit unsigned value +/- a 16-bit unsigned value. This means we need a 17-bit internal remainder to preserve the sign bit.

We've now finished our simple calculator. We've made it complete by adding a complex division function. Now, let's move on to another staple from the engineering curriculum – the traffic light controller.

Project 4 – Keeping cars in line

A classic design challenge for budding engineers is designing a traffic light controller. The Xilinx project files for the Nexys A7 can be found in CH4/build/traffic_light/traffic_light.xpr. Basys 3 doesn't provide the tricolor LEDs, so this project cannot be done directly using it:



Figure 4.12 – Traffic light controller intersection

The preceding diagram shows the basic scenario. We have an intersection with four traffic lights and four sensors labeled up, down, left, and right.

Some ground rules are as follows:

- When a light is green, it will stay green for a minimum of 10 seconds.
- When a car goes through a green light, it is ignored.
- When a car waits at the red light, it signals the green to switch after it has been green for 10 seconds.
- The light will stay yellow for 1 second when transitioning from green to red.

We've defined the problem. The first step, as always, is to create our state diagram.

Defining the state diagram

Oftentimes, I'll dive right in and code, and so a state diagram can be a good way of documenting intent and finding potential problems ahead of time:

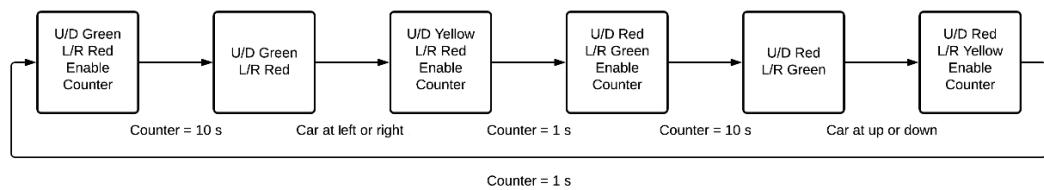


Figure 4.13 – Traffic light controller state diagram

The state machine looks relatively straightforward. It has a linear flow, with only two inputs and counters that hold state for a minimum amount of time. We know how to run our traffic lights, but how can we use our board to display the state of our traffic lights.

Displaying our traffic lights

Luckily, our boards have three colored LEDs on them. With the tricolor LEDs, we can display just about any color on the spectrum by using **Pulse Width Modulation (PWM)**. There are three outputs for each LED: red, green, and blue.

Pulse width modulation

In digital logic, we convey information at its most basic level as strings of ones and zeros. When we lit LEDs before, we applied a '1 for the LED to be lit and a '0 for it to be off. With these tricolor LEDs, we have control over three outputs, one each for red, green, and blue. If we set any of these to a '1 by itself, we will get that color very brightly lit. It is recommended that we apply a signal with a duty cycle of less than 100% to keep the LED from being blinding. In the case of the traffic light controller, I have it set at 50%:

```
always @ (posedge clk) begin
    light_count <= ~light_count;
    R          <= '0;
    G          <= '0;
    B          <= '0;
    if (light_count) begin
```

By creating a single-bit signal, called `light_count`, and using the `if` statement, I make sure the LED is only lit 50% of the time.

You might notice one potential problem with the tricolor LED. We have red and green for our traffic light, but not yellow. If you remember color theory, you know we can mix red and green to make yellow:

```
GREEN: begin
    G[0] <= '1;
end
YELLOW: begin
    R[0] <= '1;
    G[0] <= '1;
end
RED: begin
    R[0] <= '1;
end
```

It's not really practical to create a state space large enough to implement the 1 second or 10 second delay. We'll need to find a way to implement the delay circuit. One method is to create a counter and reference it within our state machine.

Implementing delays with a counter

The last element we need is a counter. If you take a look at our state diagram in *Figure 4.13*, you'll see we are only ever counting 1 second or 10 seconds. We could make two separate counters, but what I did was create one counter large enough to count to 10 seconds and reuse it for both:

```
localparam COUNT_1S = int'(100000000 / CLK_PER);
localparam COUNT_10S = 10 * int'(100000000 / CLK_PER);
bit [$clog2(COUNT_10S)-1:0] counter;
```

We've sized the counter to 10 seconds, and we have two parameters defined for terminal counts. When we want to count, we simply enable the counter via `enable_count` and test for the terminal value using the `if` statement. When the counter is not in use, we reset it to 0:

```
always @(posedge clk) begin
    lr_reg      <= lr_reg << 1 | SW[0];
    ud_reg      <= ud_reg << 1 | SW[1];
    enable_count <= '0;

    if (enable_count) begin
        counter <= counter + 1'b1;
    end else begin
        counter <= '0;
    end

    case (state)
        INIT_UD_GREEN: begin
            up_down      <= GREEN;
            left_right   <= RED;
            enable_count <= '1;
            if (counter == COUNT_10S) state <= UD_GREEN_LR_RED;
        end
    endcase
end
```

Look over the state machine and see that its flow matches the state diagram. Run it on the board.

Now you should have a functional traffic light controller. Play with the switches and verify that the lights will stay in a given state until a car is detected. Verify the lights cycle correctly. It's a lot simpler to design a traffic light controller nowadays. I had to do it on a bread board with discrete parts when I went through university.

Summary

In this chapter, we've seen how we can use our knowledge of SystemVerilog sequential and combinational elements to develop state machines. We've looked at two classical state machine designs and then developed a simple calculator using this knowledge. We also touched on some basic math as well as exploring how to develop an integer divider using SystemVerilog.

We looked at design reuse by implementing a package for our calculator and also reusing the leading ones detector we developed previously.

We briefly went over implementation of our state machine and saw at a high level how we can control our clock speed using a PLL so the design will run on the board.

With this knowledge, you can now look at expanding the calculator. We are currently only handling unsigned numbers. However, it wouldn't be that hard to make it handle signed numbers.

In the next chapter we are going to take a look at some of the board resources. We'll learn how to capture audio data and play it back. We'll learn about the temperature sensor, make a thermostat, and display the temperature on our trusty 7-segment display. We'll also learn about data processing and smooth out our sensor to make it a little less jumpy.

Questions

1. In the divider module, we perform a shift of the intermediate results. Why did we use the following:

```
{int_remainder, quotient} <= {int_remainder, quotient} <<  
1;
```

Rather than this:

```
{int_remainder, quotient} <=> 1;
```

- a) It better conveys design intent.
- b) <<= is a blocking assignment and we are using it in a clocked block, which violates the principles we laid out regarding safe design practices.
- c) When we use a concatenation function, { }, we cannot use <<=.

2. Which of the following are synthesizable SystemVerilog?

```
logic [15:0] A, B;
```

- a) A / B
- b) $A / 4$
- c) $A \% B$
- d) $5 \% 4$
3. Experiment with the colors in the traffic light controller design. Can you come up with different colors by expanding the counter size and enabling the RGB outputs at different times? The color space is practically unlimited.
4. Our calculator doesn't currently implement the divide function. Can you modify it to support division? On the Basys 3 board, you'll need to replace one of the currently mapped buttons. On the Nexys A7 board, you can use the CPU reset button or remap the buttons.

Challenge

Our simple calculator currently only handles unsigned numbers. In the case of addition, subtraction, and multiplication, the binary representation is already in two's compliment representation. Remember that to take the two's compliment of a number, you invert and add one. Can you modify the design to handle negative number representation? You might want to use one of the LEDs to represent the sign of the result.

Extra challenge

It is harder to handle negative numbers from the divider. Can you modify the non-restoring divider to make it also handle negative numbers?

Further reading

Please refer to the following links for more information regarding what was covered in this chapter:

- <https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/reference-manual>
- https://www.xilinx.com/support/documentation/user_guides/ug472_7Series_Clocking.pdf

5

FPGA Resources and How to Use Them

In this chapter, we are going to take a look at some of the underlying FPGA resources in more detail. You've been introduced to some of these in brief, such as **Random Access Memories (RAMs)** and DSP blocks, while others have been glossed over, such as PLLs, where we used one to fix a timing problem in our calculator design. We'll build upon our previous experience by incorporating these new resources.

By the completion of this chapter, you'll have a good idea of how to interface with external components. You'll be introduced to a few different data formats, pulse width modulation, and pulse data modulation. You'll see a simple serial bus in action, i2c, as well as how to implement storage in the form of a FIFO.

In this chapter, we are going to cover the following main topics:

- What is a PDM microphone?
- Simulating the microphone
- Introducing storage
- Handling i2c temperature sensor data
- Smoothing out the data
- A look at FIFOs

Technical requirements

The technical requirements for this chapter are the same as those for *Chapter 1, Introduction to FPGA Architectures and Xilinx Vivado*.

To follow along with the examples and the project, you can find the code files for this chapter at the following repository on GitHub: <https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH5>.

Project 5 – Listening and learning

This project requires the microphone on the Nexys A7 board. To run this on the Basys 3 board, an additional pmod microphone needs to be installed, interfaced, and the XDC file modified accordingly.

The Nexys A7 board has a digital microphone on board that we can use to capture the ambient noise, speech, and suchlike from the environment the board is in. We'll be utilizing this microphone to capture sound. In order to do that, we'll need to explore the format of the data and how to sample it.

It's also possible to play it back.

What is a PDM microphone?

A digital microphone needs to take analog audio data and convert it to digital data usable by electronics. A **Pulse Density Modulation (PDM)** signal is captured by a 1-bit DAC that encodes its output as a string of pulses. When the pulses are denser over a period of time, they represent larger values. In *Figure 5.1*, we see a signal from the testbench as a sine wave. The following signal shows an example of what a PDM form of that waveform might look like:

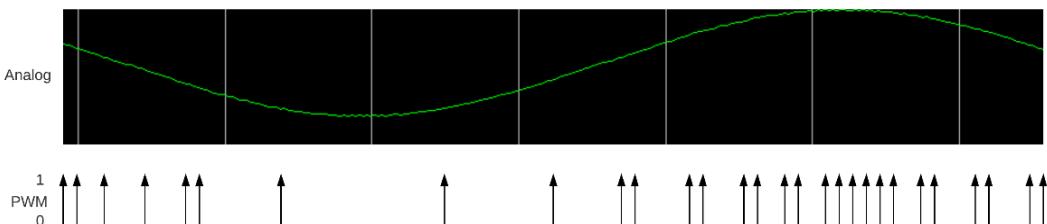


Figure 5.1 – PDM waveform example

The advantage of this type of signal is that we only need a single wire to transmit the information since typically, audio is limited to about 24 KHz and our clock rate will be orders of magnitude above this.

Let's take a look at interfacing with the microphone. The project is located at CH5/build/pdm_audio/pdm_audio.xpr.

You can find the following code in the CH5/hdl/pdm_top.sv folder:

```
module pdm_top
  #(parameter CLK_FREQ = 100)
  (
    input wire    clk, // 100Mhz clock
    // Microphone interface
    output logic m_clk,
    output logic m_lr_sel,
    input wire    m_data,
    output logic R,
    output logic G,
    output logic B,
    output logic [0:0] LED
  );

```

We can see here that we are using the 100 MHz clock as we will need a clock source to communicate with the microphone. There are two outputs to the microphone and the data back (`m_data`). The two outputs are a clock at the frequency of the data * the number of sampling cycles; in this case, 12 KHz * 128 samples = 1.536 MHz. The last signal, `m_lr_sel`, selects whether data is presented on the rising or falling edge of the clock. This is so two devices could share the same data bus to give a left and right channel, as shown in *Figure 5.2*:



Figure 5.2 – Microphone timing – left/right channel

I've also brought out one of our tricolor LEDs to display the amplitude information as feedback.

The clock to the microphone needs to be between 1 and 3.3 MHz. Following the design guidance from the Nexys A7 manual, I've constructed a circuit to generate timing for the device.

You can find the following code in the CH5/hdl/pdm_inputs.sv folder:

```
module pdm_inputs
#(
    parameter CLK_FREQ      = 100,      // Mhz
    parameter SAMPLE_RATE   = 2400000 // Hz
)
(
    input wire      clk, // 100Mhz
    // Microphone interface
    output logic     m_clk,
    output logic     m_clk_en,
    input wire      m_data,
    // Amplitude outputs
    output logic [6:0] amplitude,
    output logic     amplitude_valid
);
```

The pdm_inputs module generates the clock to the microphone and receives the data back:

```
localparam CLK_COUNT = int'((CLK_FREQ*1000000)/(SAMPLE_
RATE*2));
...
if (clk_counter == CLK_COUNT - 1) begin
    clk_counter <= '0;
    m_clk      <= ~m_clk;
    m_clk_en   <= ~m_clk;
end else
    clk_counter <= clk_counter + 1;
```

In terms of generating a new clock, we have a couple of choices. The best one is to use a PLL or MMCM to generate a precise clock that we can use. In the case of such a slow clock, this is not possible. What we can do, since we have a fairly high-speed clock running at 100 MHz, is that we can create a counter that counts to a value representing half the clock period of our generated clock and then we can create a clean clock from a flip flop. We don't want to see this clock directly in our design, so we create a `m_clk_en` pulse that lets us know when we can capture data on the rising edge of this clock.

To quantize the PDM data, we need to create a set of overlapping windows as shown in *Figure 5.3*. The overlapping windows allow us to sub-sample the results to increase the resolution of our samples, doubling our sampling frequency:



Figure 5.3 – Sampling

We have a routine that collects 12 KHz samples interleaved to create a 24 KHz output:

```

if (m_clk_en) begin
    counter[0]      <= counter[0] + 1'b1;
    counter[1]      <= counter[1] + 1'b1;
    if (counter[0] == 199) begin
        counter[0]      <= '0;
        amplitude       <= sample_counter[0];
        amplitude_valid <= '1;
        sample_counter[0] <= '0;
    end else if (counter[0] < 128) begin
        sample_counter[0] <= sample_counter[0] + m_data;
    end
end

```

```

if (counter[1] == 227) begin
    counter[1]      <= '0;
    amplitude       <= sample_counter[1] + m_data;
    amplitude_valid <= '1;
    sample_counter[1] <= '0;
end else if (counter[1] > 100) begin
    sample_counter[1] <= sample_counter[1] + m_data;
end
end

```

The sampling logic operates on the rising edge of the 2.4 MHz clock generated to the microphone. We accomplish this by using `m_clk_en` to limit when the logic operates. The timers are set up according to the Nexys A7 documentation: <https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/reference-manual>.

Now the coding is complete. In this case, it's fairly easy to simulate. We can create a testbench that feeds in a sine wave and see how our design reacts.

Simulating the microphone

I've created a testbench that we can use to verify our core. Let's verify our code and make sure we can capture the data.

You can find the following code in the CH5/tb/tb_pdm.sv folder:

```

`timescale 1ns/10ps
module tb_pdm;

```

The testbench is made up of a sine wave generator and a PDM encoder. If you run the testbench, you can see by the waves that the `pdm_inputs` module tracks the data from the sine wave generator:



Figure 5.4 – Sine wave data versus amplitude data

I've also added the amplitude signals into chipscope so we can see the results on the board. Build the design and run it. We can use an online tone generator, such as the one at <https://www.szynalski.com/tone-generator/>, as a sine wave to generate a test tone using a computer:

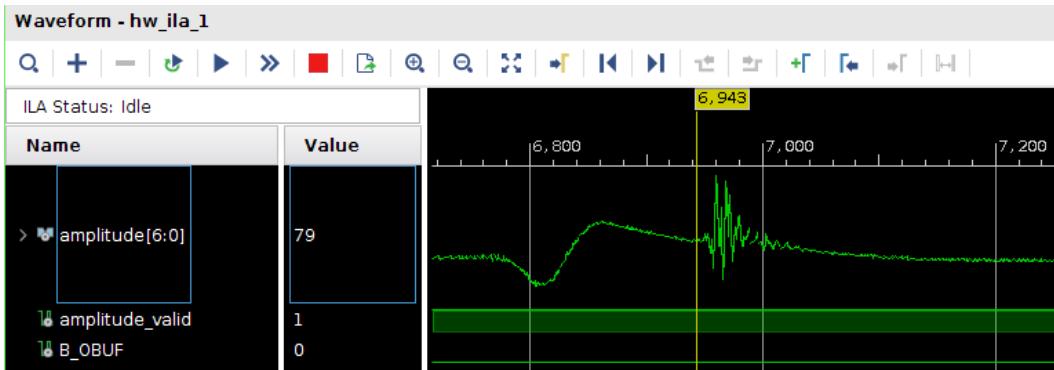


Figure 5.5 – Chipscope wave capture

In *Figure 5.5*, we can see a capture of my voice in the microphone. I found that you need to be very close to capture anything of use.

One setting in chipscope that is very helpful for a very slow clock speed is to enable capture control:

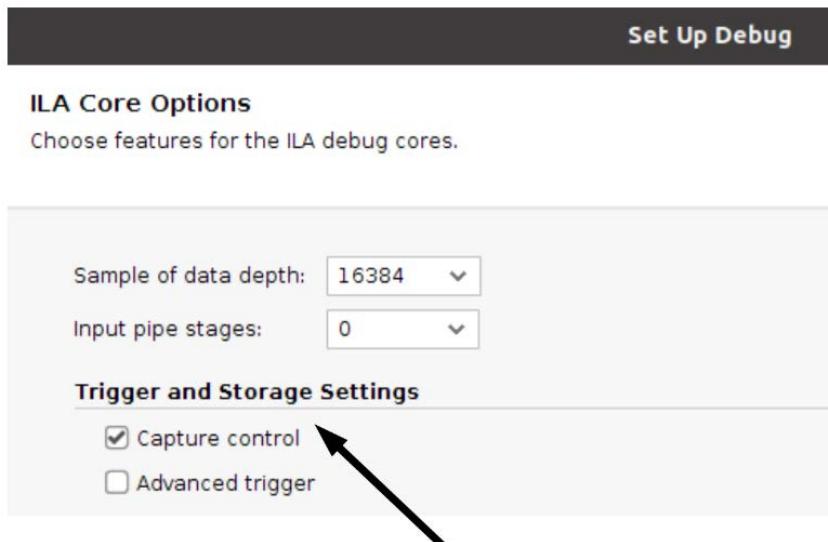


Figure 5.6 – Capture control

When capture control is enabled, you can access another pane in the ILA, which allows you to limit when data is captured to your buffer:

Name	Operator	Radix	Value	Port
amplitude_valid	==	[B]	1	prob1[0]

Figure 5.7 – Capture settings

`amplitude_valid` is set at a rate of 24 KHz. Capture setup allows us to capture only what we really want to capture to extract the valid amplitude information. Looking at the preceding waveform in *Figure 5.5*, the center point is at $\sim 7 \text{ 'd}64$, which corresponds to 0.

We've shown that we can receive the PDM data and generate amplitude values at 24 KHz, but we aren't doing anything with it yet. Let's add some storage.

Introducing storage

Now that we have managed to capture some audio data, we need to do something with it. Currently, the data register is constantly overwritten every 24 KHz. There's a lot we can do with audio data, but initially we need to store it away. What we can do is create a RAM and when we push a button, start capturing the data, lighting an LED when complete.

RAM – Inferring versus instantiating versus the IP catalog

We have a number of choices in creating a RAM. The most flexible method of creating a RAM is to infer it. This has the advantage of having cross-platform support by most FPGA vendors as well as devices. Xilinx has provided a number of templates you can use that will aid you in getting started:

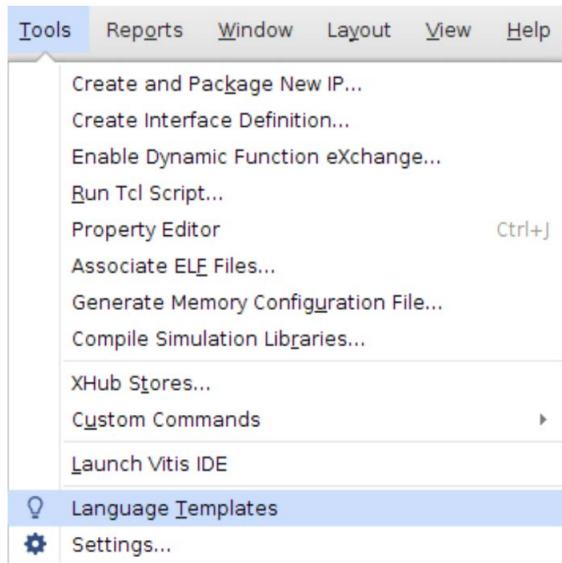


Figure 5.8 – Language templates

If you select **Tools | Language Templates**, you can access a variety of code samples that can help jump start your design. Before we get into coding our RAM, let's look at the basic types.

Basic RAM types

There are three basic RAM types:

- **Single Port (SP)**
- **Simple Dual Port (SDP)**
- **True Dual Port (TDP)**

Often, we'll look at schematics to see what the ports of a core look like. In this case, we can show how the RAMs would connect to a design, as shown in *Figure 5.9*:



Figure 5.9 – RAM types

Internally, the RAM blocks in the Artix 7 are fairly complex. They support configurable data width, byte enables, and error correction codes. I would recommend reading through the 7-series memory resource guide (reference in the *Further reading* section). I've found ways around seemingly insurmountable engineering problems, such as how to effectively implement a 64 KB deep RAM, by realizing that different configurations of the RAM can solve the problem inherently.

Single port RAMs

A **single port RAM** is the simplest type. This RAM has one address port, meaning that it can read from and write to memory at a single location on a clock cycle. This type of RAM is simple to infer using SystemVerilog:

```

localparam MEM_DEPTH = 256;
localparam MEM_WIDTH = 8;
logic [MEM_WIDTH-1:0] memory [MEM_DEPTH];
logic [$clog2(MEM_DEPTH)-1:0] address;
logic [MEM_WIDTH-1:0] write_data, read_data;
logic wren;
initial memory = '{default: '0};
always @ (posedge clk) begin
    if (wren) memory[address] <= write_data;
    read_data <= memory[address];
end

```

The preceding code shows how to infer a single port RAM. We define a `WIDTH` parameter of 8 bits and a `DEPTH` parameter of 256 x 8-bit bytes. We can see the memory array is written when `wren` is high. The RAM is read every cycle based on the address. The initial statement is optional. Block RAMs and SLICEM memory can have an initial value, whereas UltraRAM cannot.

A few important points to consider are the following:

- The storage must be declared as unpacked:

```
logic [MEM_WIDTH-1:0] memory [MEM_DEPTH];
```

- You can initialize the memory types with the exception of UltraRAM, which the Artix 7 doesn't have. To keep things portable, you shouldn't initialize very large RAM blocks.
- Block RAMs must have synchronous read data. Distributed RAMs (remember those SLICEMs?) can have asynchronous read ports. Having an asynchronous read port cuts into your timing budget significantly.

Tip

Using the '`{ }`' representation makes it easier to initialize sparsely populated unpacked arrays. You can initialize individual locations by using their address and the defaults for all others. For example, `^{0: 8'hFF, 7'h40, default: '0};` would initialize location 0 to 0xFF, 1 to 0x40, and the rest to 0.

Single port memories can only read from, or write to, a single location at a time. Simple dual port memories are a little more flexible in that you have independent control of the read and write addresses, so it's possible to read from a different location than where you are writing from. This is particularly useful when building FIFOs.

Simple dual port RAM

A **simple dual port RAM** consists of two clocks, a read and write clock. It is a subset of the true dual port RAM where you have one write port and one read port both utilizing the same or different clocks. These types of RAMs are often used for elasticity, in the case of a FIFO, or when data needs to be written from different addresses from where writes are currently occurring. These types of RAMs are also fairly easy to code up:

```
localparam MEM_DEPTH = 256;
localparam MEM_WIDTH = 8;
logic [MEM_WIDTH-1:0] memory [MEM_DEPTH];
```

```
logic [$clog2(MEM_DEPTH)-1:0] wr_address, rd_address;
logic [MEM_WIDTH-1:0]           write_data, read_data;
logic                           wren;
initial memory = '{default: '0};
always @(posedge wr_clk)
  if (wren) memory[wr_address] <= write_data;
always @(posedge rd_clk) read_data <= memory[rd_address];
```

The SDP RAM looks very much like a single port RAM, the main difference being the separation of the write and read address. Note that `wr_clk` and `rd_clk` can be the same clock or different. The underlying resource is the same.

The first two memory types are the ones you'll use the most. However, there is one other type that is less frequently used.

True dual port RAM

A **true dual port RAM** allows full read/write access from both ports. There are some restrictions in terms of what happens if there are address collisions on a read or write. I usually will use the option of `xilinx_xpm_memory` if I am using a true dual port RAM; however, they too can be inferred:

```
localparam MEM_DEPTH = 256;
localparam MEM_WIDTH = 8;
logic [MEM_WIDTH-1:0]           memory [MEM_DEPTH];
logic [$clog2(MEM_DEPTH)-1:0]   address_a, address_b;
logic [MEM_WIDTH-1:0]           write_data_a, read_data_a;
logic [MEM_WIDTH-1:0]           write_data_b, read_data_b;
logic                           wren_a, wren_b;
initial memory = '{default: '0};
always @(posedge clk_a) begin
  if (wren_a) memory[address_a] <= write_data_a;
  read_data_a <= memory[address_a];
end
always @(posedge clk_b) begin
  if (wren_b) memory[address_b] <= write_data_b;
  read_data_b <= memory[address_b];
end
```

From the preceding code, you'll notice it looks very much like two single port memories. The difference is that both always blocks reference the same storage, providing dual ports in the memory.

We've seen how to infer all three memory types. Xilinx provides macros for common functions such as **Clock Domain Crossing (CDC)**, FIFOs, and RAMs as part of their **Xilinx Parameterized Macro (XPM)** functions.

Important note

Vivado will map your inferred memory based on size. You can force a particular implementation by using the `ram_style` attribute, for example:

```
(* ram_style = "block" *) logic [7:0] memory[256];  
// Block RAM  
  
(* ram_style = "distributed" *) logic [7:0]  
memory[256]; // SLICEM based memory  
  
(* ram_style = "registers" *) logic [7:0]  
memory[256]; // Use FFs
```

Now, let's look at the Xilinx macros provided.

Instantiating memories using `xpm_memory`

The XPM functions are all located in `< vivado install >/data/ip/xpm`. We want to look at `xpm_memory`. There are six variants within the file:

- `xpm_memory_dpdistram`: Dual port distributed RAM
- `xpm_memory_dprom`: Dual port ROM
- `xpm_memory_sdpram`: Simple dual port RAM
- `xpm_memory_spram`: Single port RAM
- `xpm_memory_sprom`: Single port ROM
- `xpm_memory_tdpram`: True dual port RAM

If you need a true dual port RAM, I would explore this file and use `xpm_memory_tdpram` to instantiate it.

Using the IP catalog to create memory

The last option is to use the IP catalog to create a specific memory function. I would not recommend this option as it limits your ability to target newer FPGA families or other vendor's FPGAs without regenerating the components.

In the previous section, we've seen how to infer or instantiate memory. We have choices to make in the next section, where we'll capture the audio data.

Capturing audio data

Now that we know how to build a RAM, we can infer one to capture some audio data:

```
// Capture RAM
logic [6:0] amplitude_store [RAM_SIZE];
logic [$clog2(RAM_SIZE)-1:0] ram_wraddr;
logic [$clog2(RAM_SIZE)-1:0] ram_rdaddr;
logic ram_we;
logic [6:0] ram_dout;
always @(posedge clk) begin
    if (ram_we) amplitude_store[ram_wraddr] <= amplitude;
    ram_dout <= amplitude_store[ram_rdaddr];
end
```

To activate the capture, we can utilize one of the push buttons to initiate the capture:

```
// Capture the Audio data
always @(posedge clk) begin
    button_csync <= button_csync << 1 | BTNC;
    ram_we      <= '0;
    for (int i = 0; i < 16; i++)
        if (clr_led[i]) LED[i] <= '0;
    if (button_csync[2:1] == 2'b01) begin
        start_capture <= '1;
        LED          <= '0;
    end else if (start_capture && amplitude_valid) begin
        LED[ram_wraddr[$clog2(RAM_SIZE)-1:$clog2(RAM_SIZE)-4]] <=
        '1;
        ram_we           <= '1;
        ram_wraddr       <= ram_wraddr + 1'b1;
```

```

if (&ram_wraddr) begin
    start_capture <= '0;
    LED[15]       <= '1;
end
end // always @ (posedge clk)

```

The preceding code synchronizes the center button press and then starts a counter to capture RAM_SIZE samples. It's good to give feedback to the user, so we'll use the upper bits of the address to light the LEDs one at a time.

Once we capture the audio data, we need to do something with it to show our mastery of memory creation. Luckily, the Nexys A7 board does offer audio out:

```

// Playback the audio
always @(posedge clk) begin
    button_usync <= button_usync << 1 | BTNU;
    m_clk_en_del <= m_clk_en;
    clr_led      <= '0;
    if (button_usync[2:1] == 2'b01) begin
        start_playback <= '1;
        ram_rdaddr     <= '0;
    end else if (start_playback && m_clk_en_del) begin
        clr_led[clr_addr] <= '1;
        AUD_PWM_en <= '1;
        if (amplitude_valid) begin
            ram_rdaddr <= ram_rdaddr + 1'b1;
            amp_counter <= 7'd1;
            amp_capture <= ram_dout;
            if (ram_dout != 0) AUD_PWM_en <= '0; // Activate pull up
        end else begin
            amp_counter <= amp_counter + 1'b1;
            if (amp_capture < amp_counter) AUD_PWM_en <= '0; // Activate pull up
        end
        if (&ram_rdaddr) start_playback <= '0;
    end

```

```
end  
assign AUD_PWM = AUD_PWM_en ? '0 : 'z;
```

Again, we'll need to capture the button press and look for an edge. We have similar code to walk through the memory to output the data. We also turn off the LEDs one by one to show our activity.

There's one thing to note about driving the speaker. The output is an open drain output. This means that in order to drive a signal to the circuit driving, we will drive the signal low for a 0, but when we want the output to be a '1', we'll tristate it and a pull-up resistor on the board will take it to the correct level. The way we build a tristate signal is as follows:

1. Define the output as a wire:

```
output wire AUD_PWM
```

2. Define the internal control signal:

```
logic AUD_PWM_en;
```

3. Infer the tristate output:

```
assign AUD_PWM = AUD_PWM_en ? '0 : 'z;
```

When AUD_PWM_en is driven high, the output will be 0. When it's low, the output will be tristated.

Now that we can drive our headphone jack, we need to look at the output format. Much like our input uses PDM, the output uses **Pulse Width Modulation (PWM)**. In PDM, we received a string of ones and zeros that we could count over a period of time to determine the amplitude of a signal.

Now we've got the amplitude and we need to turn it into a PWM signal. Luckily, that is a fairly easy process. We can accomplish this by creating a 7-bit counter and comparing the count value to the amplitude over that period and sending a one out as long as it's less:

```
if (button_usync[2:1] == 2'b01) begin  
    start_playback <= '1;  
    ram_rdaddr     <= '0;  
end else if (start_playback && m_clk_en_del) begin  
    clr_led[clr_addr] <= '1;  
    AUD_PWM_en <= '1;  
    if (amplitude_valid) begin
```

```
ram_rdaddr <= ram_rdaddr + 1'b1;
amp_counter <= 7'd1;
amp_capture <= ram_dout;
if (ram_dout != 0) AUD_PWM_en <= '0; // Activate pull up
end else begin
    amp_counter <= amp_counter + 1'b1;
    if (amp_capture < amp_counter) AUD_PWM_en <= '0; // Activate pull up
end
if (&ram_rdaddr) start_playback <= '0;
end
```

Now it's time to build and try it on the board. Press the *center* button. The lights should turn on one by one. When complete, press the *up* button and the lights will go off one by one. If you plug in headphones, you should hear some noise. I tapped the microphone and heard the taps when played back.

In Project 5, we learned how to capture PDM data and store it in a RAM. We also learned how to play the data back using PWM through the audio port. It's just an introduction and gives a lot of opportunity for you to improve upon it. With an FPGA with hundreds of DSP blocks and RAM, you could add audio effects, play sounds backward, amplify, filter, and so on.

Project 6 – Using the temperature sensor

The Nexys A7 board has an Analog Device ADT7420 temperature sensor. This chip uses an industry-standard I₂C interface to communicate with. This two-wire interface is used primarily for slower speed devices. It has the advantage of allowing multiple chips to be connected through the same interface and be addressed individually. In our case, we will be using it to simply read the current temperature from the device and display the value on the 7-segment display.

Our first step will be to design an I2C interface. In *Chapter 7, Introduction to AXI*, we'll be looking at designing a general-purpose I2C interface, but for now, we'll use the fact that the ADT7420 comes up in a mode where we can get temperature data by reading two locations. First, let's take a look at the timing diagram for the I2C bus and the read cycle we'll be using:



Figure 5.10 – I2C timing

We can see from the timing diagram that we have setup and hold times, which we have seen before relative to our own designs. We also have minimum clock widths we need to maintain. We can define parameters to handle these.

You can find the following code in the CH5/hd1/i2c_temp.sv folder:

```

localparam TIME_1SEC    = int'(INTERVAL/CLK_PER); // Clock
ticks in 1 sec
localparam TIME_THDSTA = int'(600/CLK_PER);
localparam TIME_TSUSTA = int'(600/CLK_PER);
localparam TIME_THIGH  = int'(600/CLK_PER);
localparam TIME_TLOW   = int'(1300/CLK_PER);
localparam TIME_TSUDAT = int'(20/CLK_PER);
localparam TIME_TSUSTO = int'(600/CLK_PER);
localparam TIME_THDDAT = int'(30/CLK_PER);

```

I would encourage you to look at the state machine in the i2c_temp module. The state machine controls the access to the temperature sensor on the Nexys A7 board. It's fairly straightforward:

1. Wait for 1 second.
2. Send the start pattern on the SDA/SCL wires. This sends out the read command to the temperature sensor and then reads back the two 8-bit registers that contain the current temperature in Celsius.

3. Iterate until we have transmitted and received the data back.
4. Stop the transfer and go back to *Step 1*.

To access the temperature sensor, we've defined three buses:

- Send the predefined start, address, read, and stop signals.
- Force the bus to tristate during the ACK cycles and data cycles.
- Capture the data from the SDA bus.

The state machine provides us with access to the temperature sensor itself and returns the data. Now we'll need to find a way to display the data so a human can understand it.

Handling the data

We need to determine how best to display the temperature. The ADT7420 returns the data as a 16-bit value:

```
[15:7] Integer
[6:3] fraction * 0.0625
[2:0] Don't Care
```

We can use our `bin_to_bcd` function on the integer portion to generate our 7-segment display data, but what can we do to calculate the fractional portion? We only have 16 values, so we could create a lookup table and simply look up the lower 4 digits. This is effectively creating a ROM that can be indexed into. A ROM is created much like a RAM:

```
logic [15:0] fraction_table[16];
initial begin
    for (int i = 0; i < 16; i++) fraction_table[i] = i*625;
end
```

We can then convert the temperature based on the output of the temperature sensor chip:

```
// convert temperature from
always @(posedge clk) begin
    convert_frac <= convert;
    if (convert) begin
        encoded_int  <= bin_to_bcd(temp_data[15:7]); // Decimal
        portion
        fraction     <= bin_to_bcd(fraction_table[temp_data[6:3]]);
```

```
    decimal      <= 8'b00010000;  
  end  
end // always @ (posedge clk)  
assign encoded = {encoded_int[3:0], encoded_frac[3:0]};
```

One disadvantage of converting the temperature every second and having a fractional precision is that the display will change quite a bit depending on your environment. We can apply what is essentially a filter to the data so that we take the average temperature over a period of time.

Now that we have learned how to handle the data, let's learn more about how we can filter and improve the quality of that data.

Smoothing out the data

In base 10, it's very inexpensive to divide by a multiple of 10. Every multiple of ten is simply a shift to the right of 1 digit:

```
12345/10 = 1234.5 Truncated = 1234, Rounded = 1235  
12345/100 = 123.45 Truncated = 123, Rounded = 123
```

Similarly, in binary, every shift to the right is a division by 2:

```
10110>>1 = 1011.0 Truncated = 1011, Rounded = 1011  
10110>>2 = 101.10 Truncated = 101, Rounded = 110
```

How does this help us with filtering? If we want to filter over a period of 2, 4, 8, 16, 32... 2^n samples, the division operation is virtually free since it is simply a shift and possible rounding.

We can create a simple filter by summing our temperature data over a period of time:

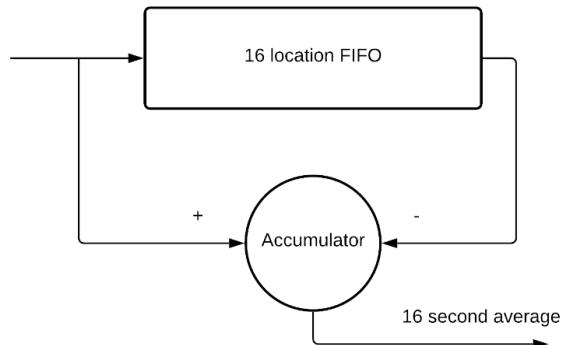


Figure 5.11 – Simple moving average filter

The way we create a moving average filter is to keep a running average over a period of time. In the preceding case, I picked 16 cycles, although any power of two is a good choice. A non-power of two is an option, but until we discuss how to improve this filter in *Chapter 6, Math, Parallelism, and Pipelined Design*, I wouldn't consider it.

The way the filter works is that we add incoming data to an accumulator and subtract the output from 16 clock cycles previously. The accumulator then contains the sum of the incoming data over the last 16 cycles. If we divide this by 16, we have an average over the last 16 cycles.

Deeper dive into FIFOs

The heart of a FIFO is a RAM, a read and a write pointer, and flag generation logic. In a synchronous FIFO, it is very easy to implement:

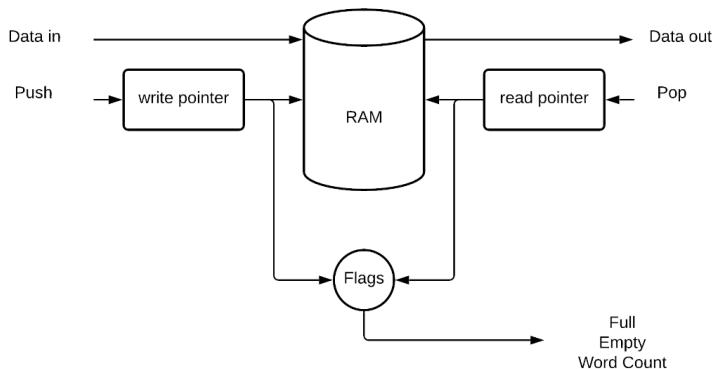


Figure 5.12 – Synchronous FIFO

The write pointer increments on every push and the read pointer increments on every pop.

Generating the flags boils down to comparing the read and write pointers against one another. When we are dealing with a synchronous FIFO, these comparisons are easy since everything is generated by the same clock and is properly timed. What about an asynchronous FIFO, that is, a FIFO with separate read and write clocks?

Remember our discussions on synchronization and multibit buses. What happens if we try to compare a read and write address on different clocks?



Figure 5.13 – Asynchronous FIFO (non-functional)

The difference between a synchronous FIFO and an asynchronous FIFO is shown by the dotted line down the middle of *Figure 5.13*. Each half is an independent clock domain. In *Figure 5.12*, everything is on a single clock domain.

Let's consider a read and write pointer on different clock domains. Assume the clocks have no relationship and that the depth of the FIFO is 16 with 4-bit addresses:



Figure 5.14 – FIFO addressing

Looking at the preceding diagram, you can see that when the counter has multiple bits changing at the same time and the clocks are asynchronous, we really can't determine what the captured address will be.

We can fix this issue by gray coding the address pointers.

Gray coding

Binary counts increment by adding a 1 to the lowest bit. This results in a count such as the following:

00 – 01 – 10 – 11.

Gray coding only allows one bit to change at a time, such as the following sequence:

00 – 01 – 11 – 10

Important note

Gray coded counters have a limited range as they must always only have one-bit change at a time. A power of 2 is always safe to implement, but other combinations, such as $2^n + 2^m$, also work.

Let's take a look at a FIFO using a gray code:



Figure 5.15 – Asynchronous FIFO with gray coding

By adding a gray code module and synchronizers across each clock domain, we can compare the gray coded values against each other or convert them back to binary on the destination clock.

Since gray codes only allow one bit to change at a time, we are guaranteed to either capture the old value or the new one and not capture a transitional value that would cause a FIFO empty, full, or word count error.

Constraints

This is a case where we will want to utilize the `set_max_delay` constraint between the write pointer and the first register in the synchronizer:

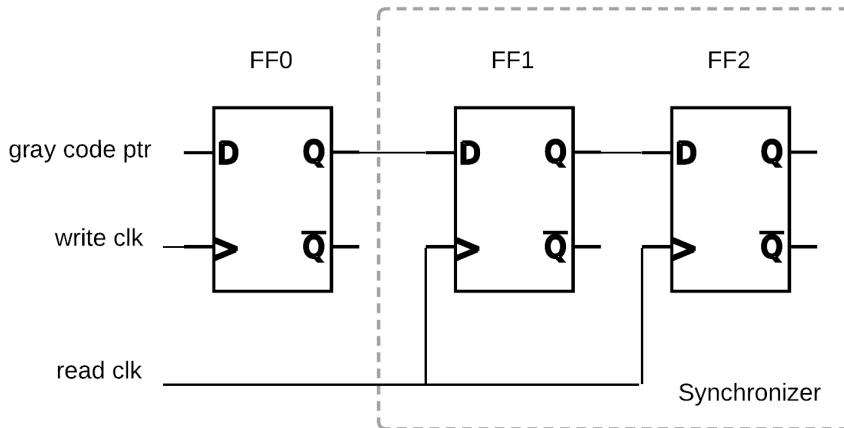


Figure 5.16 – Using `set_max_delay`

Looking at one side of the gray code and synchronizer logic (there is a similar circuit on the read side), we can see where we need to apply the `set_max_delay` timing constraint:

```
Set_max_delay -datapath_only <delay> -from [get_pins FF0/Q] -to [get_pins FF1/D] -datapath_only
```

To be absolutely safe, the delay should be set to the destination clock period or less. It can be up to 2 destination clocks, but to be safe, use between 1 and 1.5* the destination clock period.

Generating our FIFO

We will be using a synchronous FIFO, but understanding how an asynchronous FIFO works is of critical importance if you decide to pursue a career utilizing FPGAs. You will almost certainly be asked an interview question regarding this.

We have the advantage the Xilinx has created a macro for us to use, `xpm_fifo_(sync | async)`. You can view the instantiation in `i2c_temp.sv`. You'll see that there are a lot of ports that we are not using. We'll simply be pushing data into the FIFO on every convert signal and we have a small state machine that generates our data to be converted to send to our 7-segment display interface:

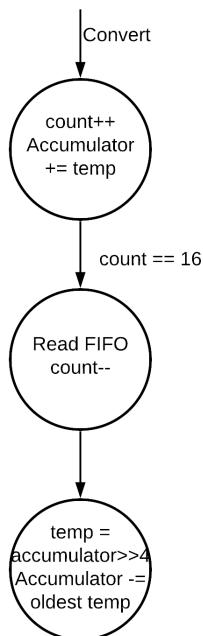


Figure 5.17 – Simple temperature filter state machine

We can look at the state machine and it's very straightforward. We create an elastic buffer using the FIFO to hold 16 samples. As each new sample comes in, we build up our accumulator value. When we hit 16 samples, we divide the accumulator by 16 and that gives us the average temperature over the previous 16 seconds:

```
always @(posedge clk) begin
    rden        <= '0;
    rden_del    <= rden;
    smooth_convert <= '0;
    if (convert) begin
        smooth_count          <= smooth_count + 1'b1;
        accumulator           <= accumulator + temp_data;
    end else if (smooth_count == 16) begin
```

```

rden           <= '1;
smooth_count   <= smooth_count - 1'b1;
end else if (rden) begin
    accumulator      <= accumulator - dout;
end else if (rden_del) begin
    smooth_convert    <= '1;
    smooth_data       <= accumulator >> 4;
end
end

```

Take a look at the preceding code and see whether you can pick out the state progression. It's not written as we have before, but you should be able to make out the flow. Once you've had a chance to look over the code, build it and try it out on the board.

One thing that may be a surprise is that the output stays 0 for a long period of time, 16 seconds to be precise. This is because we are waiting to fill the FIFO. What would happen if we output the accumulator/16 every cycle? Think about it for a minute or change the code and see.

If we always output accumulator/16, we would end up with a temperature creeping up from 1/16 the current temperature to the average current temperature of the room. For something non-critical such as this, either method would be acceptable, but what if we didn't want to do this and we always wanted the current value. For that, you'll need to wait until *Chapter 6, Math, Parallelism, and Pipelined Design*, when we discuss fixed point representation.

Summary

In this chapter, we've explored how to do some simple communication with the outside world. We've gathered microphone data, stored it, and played it back. We've also explored the I2C bus, a common way of communicating with slower speed devices. We captured temperature data and showed how we could display a fixed-point number on the 7-segment display. We introduced FIFOs and discussed how we can filter the data to remove the noisiness of the temperature data varying.

I2C interfaces are used to communicate with many low speed devices such as A/Ds and D/As and are very important for a lot of FPGA designs. You should feel comfortable that you can do it at this point, and we will explore making a more generic version of the interface in a later chapter. If you are interested in audio data, you should have some confidence in capturing, manipulating, and generating audio.

In the next chapter, we are going to look at some mathematical operations. We'll explore how we can smoothly bring up our temperature sensor with fixed point math. We'll take a look at floating point numbers and operations we can perform on the audio data.

Questions

1. What are the advantages of an I2C bus?
 - a) We can move large amounts of data quickly.
 - b) We only need two wires to communicate.
 - c) Multiple devices can be connected using only two wires.
 - d) All of the above.
 - e) Only (b) and (c).
2. What would be the preferred order of preference when you require a memory?
 - a) Use the IP catalog, infer, use `xpm_memory`
 - b) Use `xpm_memory`, use the IP catalog, infer
 - c) Infer, use `xpm_memory`, use the IP catalog
 - d) Use the IP catalog, use `xpm_memory`, infer
3. `assign data = (data_en) ? 'z : '0;`
 - a) Infers a multiplier
 - b) Infers a register
 - c) Infers a tristate IO
4. Gray coding is used in FIFOs.
 - a) Always
 - b) To pass counter information across clock domains in an asynchronous FIFO
 - c) Only in synchronous FIFOs

The following code creates what kind of memory?

```
always @(posedge clk) begin
    if (wren) store[addr] <= din;
    dout <= store[addr];
end
```

- a) Simple dual port
- b) True dual port
- c) Single port
- d) ROM

Further reading

Please refer to the following links for more information regarding what was covered in this chapter:

- <https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/reference-manual>
- **Temperature sensor specification:** <https://www.analog.com/media/en/technical-documentation/data-sheets/adt7420.pdf>
- **Deeper dive into FIFOs:** http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO2.pdf
- https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf

6

Math, Parallelism, and Pipelined Design

Microprocessors are custom designed ASICs that can have very high performance when running at very high frequencies – up to 5 Ghz as of writing this book. These processors are general-purpose, meaning they need to balance their operations for a wide variety of tasks. In contrast, the Artix 7 we are targeting can hit speeds of up to 300-400 Mhz. Higher-end FPGAs can hit speeds of up to 800 Mhz. Unlike microprocessors, FPGAs can be targeted for a specific application. Because of this, we can utilize design techniques such as parallelism; that is, replicating logic in order to perform more tasks for a given clock cycle than a microprocessor can. We can also use pipelining to achieve a high throughput.

In this chapter, we will look deeper at fixed-point numbers with regards to our temperature sensor. We'll also look at floating-point numbers and see why we might want to use one over the other. Then, we'll look at the limitations of using floating-point numbers in an FPGA (or even in general). We'll then explore parallelism and pipelined designs by looking at the Xilinx FFT IP and how we can apply them to our audio file.

By the end of this chapter, you should have a good handle on fixed- and floating-point math. You'll have an understanding of AXI streaming and how you can connect multiple components together using it. This will demonstrate pipelining, one of the two ways of getting performance from an FPGA. We'll briefly discuss how FPGAs are used in parallel systems.

In this chapter, we are going to cover the following main topics:

- Fixed-point arithmetic
- Single precision and double precision floating-point numbers
- Floating-point arithmetic
- Pipelined designs
- Parallel designs

Let's get started!

Technical requirements

The technical requirements for this chapter are the same as those for *Chapter 1, Introduction to FPGA Architectures and Xilinx Vivado*.

To follow along with the examples and projects in this chapter, take a look at the code files for this chapter by going to this book's GitHub repository: <https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH6>.

Introduction to fixed-point numbers

We've worked extensively with binary and BCD numbers throughout this book. Binary is great for math because addition, subtraction, and multiplication are cheap and easy. Division isn't too bad, but more time-consuming. We have only really used BCD numbers for displaying output.

In the previous chapter, we needed to introduce fixed-point numbers. Recall the temperature sensor format:

```
[15:7] Integer  
[6:3] fraction * 0.0625  
[2:0] Don't Care
```

If we look at mathematical operations, we know that adding two numbers increases the result size by 1 bit and that to multiply two numbers, we need to add the sizes together. The one question is where the fixed point goes in both cases:



Figure 6.1 – Addition/subtraction and multiplication of fixed-point numbers

The important thing to remember is that when you're adding two fixed-point numbers, the digit point will remain at the same location. When multiplying, you add both the integer bit positions to get the resultant integer bits and the number of both fractional bits to get the resultant fractional bits. This can be seen in the preceding diagram. Here, we are multiplying two numbers of 9.4, which results in 18.8.

One of the important things to remember is that when you add, you must make sure the decimal points of both numbers are aligned. When you multiply, there is no such requirement.

The advantage of dealing with fixed-point numbers is that the same logic is used for math operations in binary as fixed point. The only the difference is maintaining where the decimal point is in your logic.

Now that we have learned how to utilize fixed-point arithmetic, we'll put it to work in our temperature sensor.

Project 7 – Using fixed-point arithmetic in our temperature sensor

Let's take a look at how we can optimize our temperature averaging to handle the 16 seconds where the temperature is incorrectly calculated. This happens because we are dividing an invalid temperature over the first 15 clock cycles.

There are cases where either a delay or inaccurate results can't occur. I was actually asked a job interview question regarding how to make sure that the output from this type of filter was valid during the bring-up time, so this really is a practical question that you may need to address someday.

Using fixed-point arithmetic to clean up the bring-up time

First, let's take a look at what a fixed-point scaling factor looks like. In the end, we want to scale to that of a single value from the sensor. To do this, we want to scale by a fraction. The following table shows the first 15 cycles, plus the steady state of the accumulator. I've populated the following table to show the fractions we need to calculate:

Cycle	Accumulator value	Multiply
1	T0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0	1
2	T0 + T1 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0	$\frac{1}{2}$
3	T0 + T1 + T2 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0	$\frac{1}{3}$
4	T0 + T1 + T2 + T3 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0	$\frac{1}{4}$
5	T0 + T1 + T2 + T3 + T4 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0	$\frac{1}{5}$
6	T0 + T1 + T2 + T3 + T4 + T5 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0	$\frac{1}{6}$
7	T0 + T1 + T2 + T3 + T4 + T5 + T6 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0	$\frac{1}{7}$
8	T0 + T1 + T2 + T3 + T4 + T5 + T6 + T7 + 0 + 0 + 0 + 0 + 0 + 0 + 0	$\frac{1}{8}$
9	T0 + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + 0 + 0 + 0 + 0 + 0 + 0	$\frac{1}{9}$
10	T0 + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + T9 + 0 + 0 + 0 + 0 + 0	$\frac{1}{10}$
11	T0 + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + T9 + T10 + 0 + 0 + 0 + 0	$\frac{1}{11}$
12	T0 + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + T9 + T10 + T11 + 0 + 0 + 0 + 0	$\frac{1}{12}$
13	T0 + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + T9 + T10 + T11 + T12 + 0 + 0 + 0	$\frac{1}{13}$
14	T0 + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + T9 + T10 + T11 + T12 + T13 + 0 + 0	$\frac{1}{14}$
15	T0 + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + T9 + T10 + T11 + T12 + T13 + T14 + 0	$\frac{1}{15}$
16+	T0 + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + T9 + T10 + T11 + T12 + T13 + T14 + T15	$\frac{1}{16}$

We'll need to multiply the accumulator by the fractional value and shift it to get the scaled result. If we do this, rather than see the temperature ramp up, we should see it stay fairly consistent.

I recommend the following site for converting fractional values into a binary representation: <https://www.exploringbinary.com/binary-converter/>. By converting the fractions into a decimal fraction and using this website, I came up with the following conversion table:

divide[0]	= 17'b1_0000000_0000000; // 1
divide[1]	= 17'b0_1000000_0000000; // 1/2
divide[2]	= 17'b0_01010101_01010101; // 1/3
divide[3]	= 17'b0_0100000_0000000; // 1/4
divide[4]	= 17'b0_00110011_00110011; // 1/5
divide[5]	= 17'b0_00101010_10101010; // 1/6
divide[6]	= 17'b0_00100100_10010010; // 1/7
divide[7]	= 17'b0_00100000_00000000; // 1/8
divide[8]	= 17'b0_00011100_01110001; // 1/9
divide[9]	= 17'b0_00011001_10011001; // 1/10
divide[10]	= 17'b0_00010111_01000101; // 1/11
divide[11]	= 17'b0_00010101_01010101; // 1/12
divide[12]	= 17'b0_00010011_10110001; // 1/13
divide[13]	= 17'b0_00010010_01001001; // 1/14
divide[14]	= 17'b0_00010001_00010001; // 1/15
divide[15]	= 17'b0_00010000_00000000; // 1/16
divide[16]	= 17'b0_00010000_00000000; // 1/16

Remember from the previous chapter that we can create a ROM using an initial statement. The first thing we need to do is decide how many bits of precision we need. A **DSP 48** can handle an 18x25 two's compliment multiplication. Because of this, I chose a 17-bit unsigned scaling factor. Making the factor larger could impact the number of multipliers used or the speed of operation; making it smaller won't make a difference in terms of resources needed, but could reduce the accuracy.

The preceding table is of the format 1.16 and is truncated and not rounded. You can consider rounding the values. Rounding a binary number is as simple as adding the uppermost bit you are going to truncate to the bits you are going to keep:



Figure 6.2 – Rounding

0.00000 rounding to 4 bits would be 0.0000.

0.00001 rounding to 4 bits would be 0.0001.

0.11111 rounding to 4 bits would be 1.0000.

So, we've got our scaling factors, but how can we utilize them? I've modified the pipeline we created so that the accumulator now calculates on every input, the scaling factor is applied, and the data is displayed. Previously, we only outputted the data once 16 cycles of data had been accumulated:

```
always @ (posedge clk) begin
    rden      <= '0;
    smooth_convert <= '0;
    convert_pipe   <= convert_pipe << 1;
    if (convert) begin
        convert_pipe[0]          <= '1;
        smooth_count             <= smooth_count + 1'b1;
        accumulator              <= accumulator +
                                    temp_data[15:3];
    end else if (smooth_count == 16) begin
        rden                  <= '1;
```

```

smooth_count           <= smooth_count - 1'b1;
end else if (rden) begin
    accumulator          <= accumulator - dout;
end else if (convert_pipe[2]) begin
    if (~sample_count[4]) sample_count <= sample_count + 1'b1;
    smooth_data          <= accumulator *
                           divide[sample_count];
end else if (convert_pipe[3]) begin
    smooth_convert        <= '1;
    smooth_data          <= smooth_data >> 16;
end
end

```

The pipeline is now free running since it shifted `convert_pipe` freely. What we've changed is that we scale the accumulator value by the scaling factor we defined. Remember our discussion regarding multiplication. Here, we've added 16 bits of the fraction to our scaled value, so we need to remove that in the end. We can accomplish this by adding a stage, `convert_pipe[3]`.

We can run the simulation and see that the data is fairly constant. I've modified the testbench to input a constant 25 degrees Celsius, `0x19` in hex. Now, build the bitstream and try it on the board. Rather than 16 seconds of 0, you should see 1 second and then data being displayed. It should be fairly constant for the time the board is up.

From this project, you should notice that fixed-point arithmetic isn't costly and is very easy to implement. The pipeline only has four clock cycles to calculate the smoothed-out temperature over the last 16 cycles.

Temperature conversion using fixed-point arithmetic

Our temperature sensor project should be good enough for everyone. We can display the device temperature to 1/16 of a degree Celsius precision and we've added averaging, which means we can now bring it up cleanly. However, it is missing one thing. If you are outside the US, I'm sure you could care less what the temperature is in anything except Celsius, but in the US, we stubbornly hold onto imperial measurements. Bear with me as I add Fahrenheit conversion so that I can tell what the temperature actually is.

First, let's take a look at the formula that's used to convert Celsius into Fahrenheit:

$$T_{Fahrenheit} = \left(T_{Celsius} \times \frac{9}{5} \right) + 32$$

Figure 6.3 – Formula for converting Celsius into Fahrenheit

As we can see, the formula is straightforward. We implement a divider and a multiplier, but since $9/5$ is a constant, we can create a fixed-point representation of it and then multiply it by the constant. Recall how long it took for our divider and that we can perform multiplication in a single cycle? This highlights something important to bear in mind. Often, there are multiple ways of tackling a problem. The first or most obvious solution isn't always the best, so it's a good idea to keep an open mind about other ways to implement a solution.

As is usually the case, we have a choice regarding where we can perform this operation. To keep things small and simple, I propose we do this once we scale down our intermediate result. This reduces the multiplier's size.

We'll also need a way to select Celsius or Fahrenheit, so we'll add `SW[0]` to control Celsius/Fahrenheit and `LED[0]` to indicate if we are displaying Fahrenheit.

I've modified our pipeline slightly so that we can apply the conversion:

```
end else if (convert_pipe[3]) begin
    smooth_data      <= smooth_data >> 16;
    smooth_convert <= ~SW;
end else if (convert_pipe[4]) begin
    smooth_convert      <= SW;
    smooth_data          <= ((smooth_data * NINE_FIFTHS) >>
    16) + (32 << 4);
end
```

The main change is that, in `convert_pipe[3]`, we selectively send the Celsius data to the BCD conversion by using `smooth_convert <= ~SW`. `convert_pipe[4]` handles the heavy lifting of the Fahrenheit conversion. Note that in this case, we are taking advantage of more of the DSP 48 than we have in the past since we are performing a multiply and an add in a single clock cycle.

Build the design and verify the Fahrenheit display:

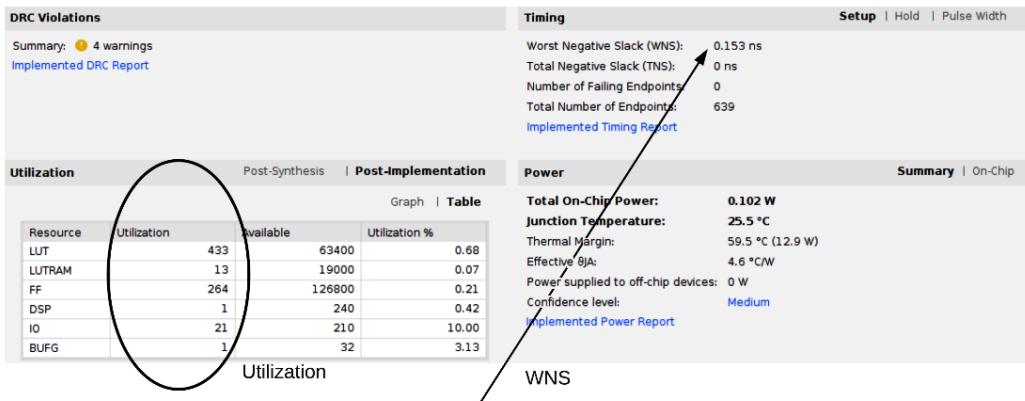


Figure 6.4 – i2c_temp, fixed point with Fahrenheit conversion

With a positive WNS, we are making timing comfortably and are using very few device resources. Let's take a look at the conversion pipeline in the simulation:



Temperature captured

Fahrenheit conversion ends

Figure 6.5 – Fixed-point I2C simulation with Fahrenheit conversion

Here, you can see that the pipeline is short and performs the conversion with a small number of device resources when we use a fixed-point implementation.

Looking at our design, we didn't have to modify the I2C interface. In fact, we can make it a more general-purpose core that we could use to connect to other I2C devices. We'll address this in *Chapter 7, Introduction to AXI*.

What about floating-point numbers?

You've probably heard of floating-point numbers. Where fixed-point numbers can only represent a very limited, defined range of fractional values, floating-point numbers can represent numbers from the very small to the very large, although their precision is limited based on the standard that's used. The **Institute of Electrical and Electronics Engineers (IEEE)** has defined a number of floating-point formats. Graphics card companies such as Nvidia have also contributed to the standard over the years.

Floating-point arithmetic operations are expensive compared to fixed-point ones. To give you an idea of this, it wasn't until the Pentium processor that Intel standardized on integrating its floating-point coprocessor into the main microprocessor. Prior to the Pentium processor, every x86, from 8086 to the 80486 had a corresponding x87 processor (8087, 80287, and so on) that provided floating-point operations.

Some of the reasons for Xilinx's choices in designing the DSP48 blocks in their designs was to better support floating point in FPGAs. Floating point is no longer as prohibitive to include as it was when I started FPGA design, though it is still generally slower and more complex than fixed point.

Let's take a look at the IEEE single and double precision floating-point representations of numbers:



Figure 6.6 – IEEE single precision floating-point representation

The **sign** bit is the same as a two's compliment number in that a '1' indicates a negative value and a '0' indicates a positive value. The fraction is an unsigned 24-bit number. You'll notice that only 23 bits have been defined. The 24th bit is an implied 1, where the number represented is $1.\text{fraction}$. This is made possible since the value of 0 is represented by the 32-bit field being set to all 0s.

The exponent is biased from 127, so the actual exponent is -127 to give a value from -126 to +127. -127 and + 128 are reserved.

What we can infer from this is that floating point is an excellent choice when we have numbers relatively close together but have a large potential range of values. Floating point covers these cases, but at the cost of resources and processing time. Math is more costly and slower than fixed point, but if in one set of calculations you are operating at a microscopic scale and in another set, you are at a galactic scale, you can use a single format.

Double precision extends the exponent to 11 bits and the fraction to 52 bits.

In the past, if you wanted to design something using floating point, you'd need to design your own floating-point operators. An example of a floating-point operator can be found in my GPLGPU project at <https://github.com/asicguy/gplgpu/tree/master/hdl/math>.

I'll discuss the main components here. For this analysis, I'll be focusing on the floating-point designs for the GPLGPU that were implemented in ASICs in 1998 and re-implemented in FPGAs in the early 2000s. Depending on the speed you are targeting, the pipelining may be more or less the same, but this is a good baseline and starting point for discussion.

Floating-point addition and subtraction

Where multiplication is typically a slower operator for binary or fixed-point numbers, addition/subtraction can actually have more latency in terms of floating point. The reason for this is that we need to align our decimal point as if we were doing the calculations by hand. Remember that the fraction has an implied 1. This means that once the addition or subtraction is complete, we need to adjust the exponent so that the final fraction is of the form $1.x$.

Floating-point multiplication

Floating-point multiplication is not as complex. We simply add the exponents and multiply the mantissa.

Floating-point reciprocal

Here is where things get interesting. **Integer division** is a series of subtractions that are performed by restoring and non-restoring division. This gives us a precise answer, though for large integer values, it can take hundreds of clock cycles to complete.

Like our integer division algorithms, we'll need a similar algorithm for floating-point multiplication. What I've used in the past is **Newton-Raphson**. It consists of an initial guess that's provided via a lookup table. This is precalculated. Then, successive approximations converge on a solution. You may remember (or have heard about) the Pentium division bug. This bug occurred due to bad table values in their division algorithm.

A more practical floating-point operation library

You are welcome to explore or use the functions in the GPLGPU. They are licensed under GPL v3. However, let's explore what Xilinx has to offer for floating-point libraries:

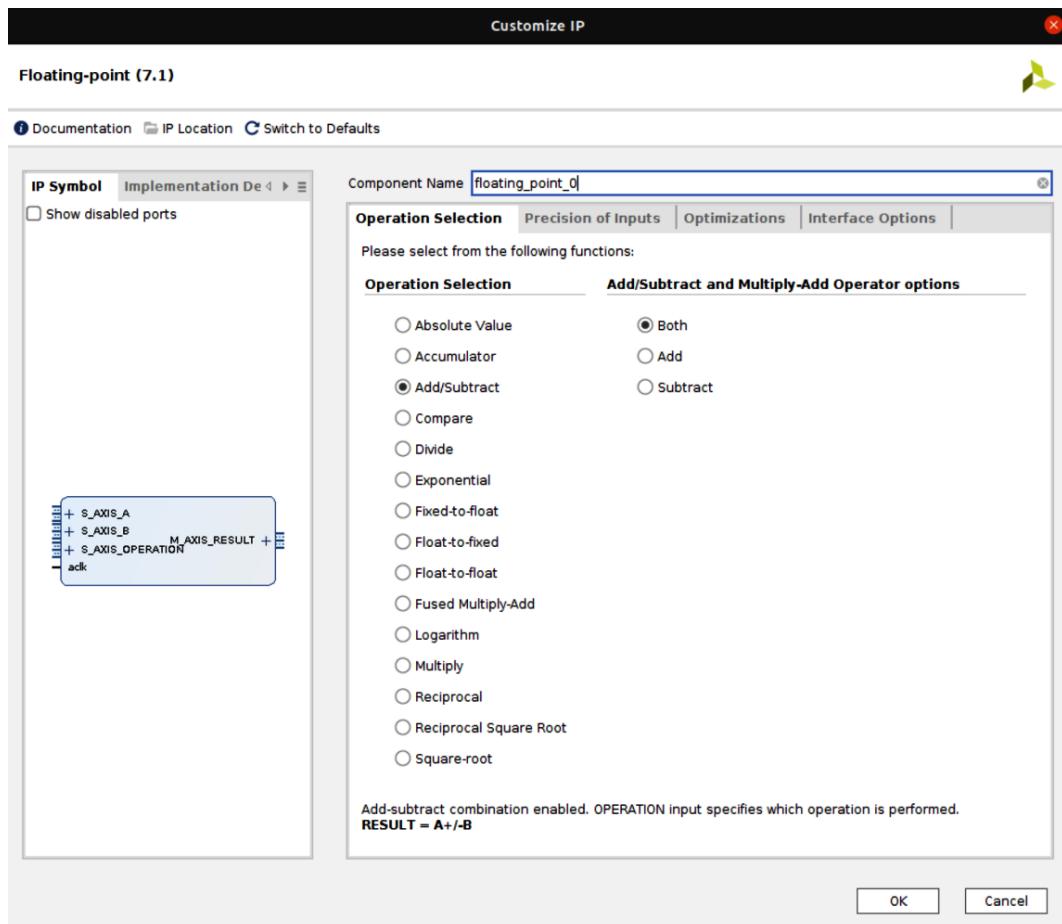


Figure 6.7 – Xilinx floating point IP

We talked about the Xilinx IP catalog previously. Search for `floating` and you'll find the floating-point IP wizard. You will see that Xilinx provides a complete set of operators with Vivado.

Like much of Xilinx IP, it utilizes an AXI interface – specifically the AXI streaming interface. Let's explore it in more detail.

A quick look at the AXI streaming interface

Xilinx IP has almost universally migrated to using the AXI bus, which is a set of protocols defined by ARM. There are three AXI variants; we'll look at them in more detail in *Chapter 7, Introduction to AXI*. However, we will need to take a quick look at the AXI streaming interface in order to use the Xilinx floating-point IP.

When Xilinx started building SOC FPGAs with integrated processors, they needed a bus standard for their IP, as well as user-produced IP. Since ARM already had IP interfaces defined, Xilinx took its IP with native interfaces or older style interfaces and ported them to AXI for compatibility with the ARM processors.

AXI streaming is point-to-point communication that's been optimized for data movement. It is the simplest of the AXI protocols in that it doesn't require address decoding, so it's used for a lot of IP work. First, let's take a look at how an AXI streaming interface works:



Figure 6.8 – AXI stream example

The interface itself is straightforward. The data source drives a valid signal, along with data. A transfer is terminated when **tlast** is asserted. The slave can throttle the data using **tready**. You can see from the waveform that data is only transferred when **tread** and **tvalid** are asserted. If **tready** goes low, the source must hold **tlast**, **tdata**, and **tvalid** until **tready** goes high.

Now, let's examine what we need to add or change in the design to convert it into floating point.

Project 8 – Updating the temperature sensor project to a pipelined floating-point implementation

First, let's put our proposed design into a diagram to determine what we need:



Figure 6.9 – Floating-point conversion pipeline

The pipeline looks very similar to our previous temperature pipeline. The main differences are that we are now converting to/from floating point on the input and output. Internally, the old 4-5 stage pipeline is handled similarly. However, each stage is no longer a single clock cycle since floating-point operations take longer to process.

To convert our temperature sensor and Fahrenheit conversion, we will need the following floating-point operations, all of which we can generate from the Vivado IP catalog as we'll see in the next section.

Fix to floating point conversion

We'll need to make a couple of modifications to customize the fix_to_float operator for our particular use case:



Figure 6.10 – fix_to_float format configuration

Recall that the format of the temperature sensor is 4 bits of a fraction and 9 bits of an integer. Xilinx likes to make the streaming interface a multiple of 8, so I set the conversion to 12.4:



Figure 6.11 – Changing the interface

We'll want to modify the flow control so that it's non-blocking. It's not strictly necessary for this design, but it will maximize our resource usage when pipelining it to give us an idea of the worst-case floating-point resources. This pane also gives you the option of adding some of the optional components of the AXI bust: the **tlast** and **tuser** signals. **Tlast** is useful if you'll be passing large amounts of data and you need to determine when a grouping of data finishes. On the other hand, **tuser** allows you to pass information, along with data, so that you can use it in your design.

Floating-point math operations

If we construct our pipeline with a little extra control logic, we can share the addition and subtraction, as we did in our fixed-point case:

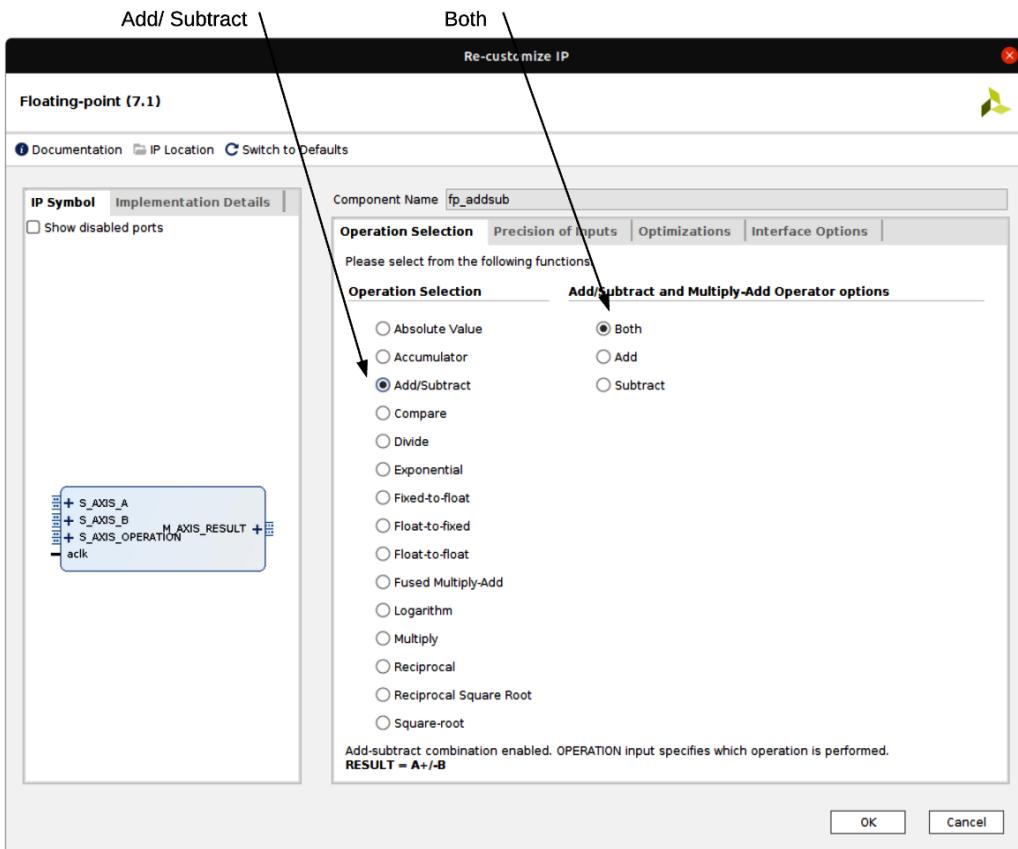


Figure 6.12 – Add/Subtract

Make sure that you change the interface, as shown in *Figure 6.11*.

We'll need two additional components: a multiplier and a fused multiply add. Make sure that you follow *Figure 6.11* when modifying the interfaces for both. When you generate the fused multiply add, make sure you select only **Add**:

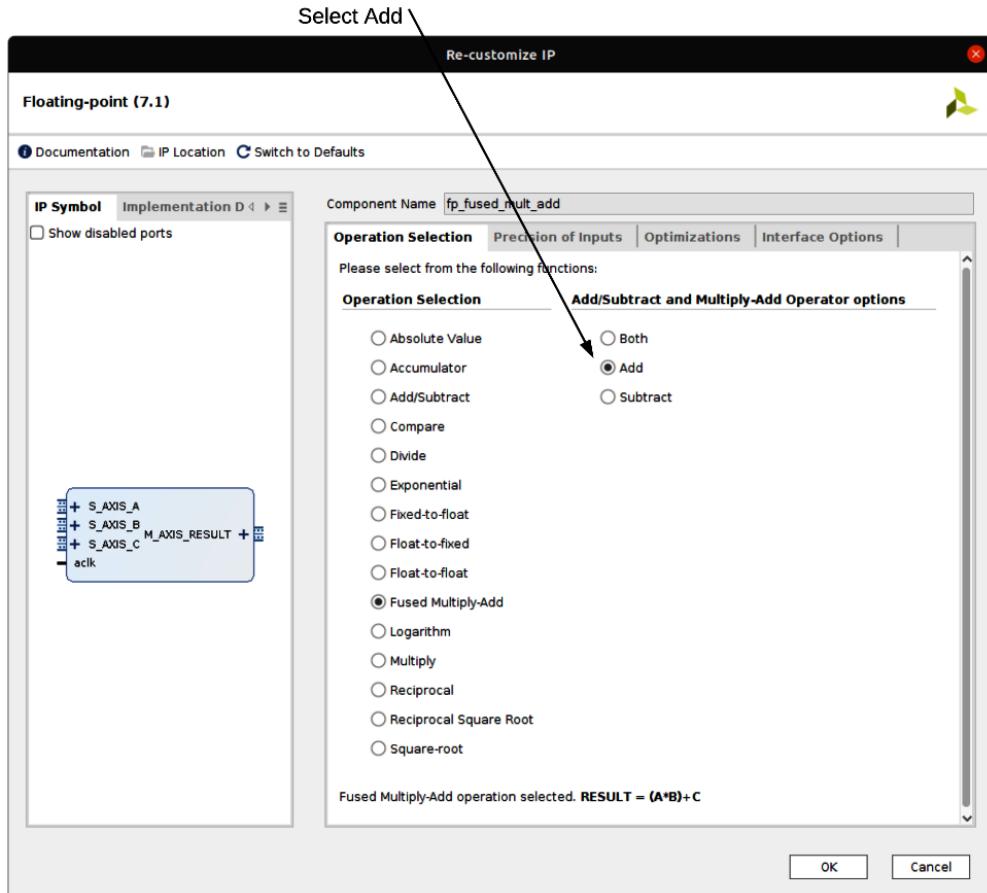


Figure 6.13 – Fused multiply add

Finally, we'll need to convert back to fixed point for our seven-segment display.

Float to fixed point conversion

Let's look at what we need to do to generate our fixed-point output:



Figure 6.14 – Customizing float to fixed output

With that, we've got all the components of our pipeline so that we can take fixed points in, operate entirely on the data as a floating point, and then write the data out as a fixed point.

Let's take a look at our simulation to see what our latency looks like.

Simulation

If you take a look at the latency in the components, you'll see that each floating-point operation is adding quite a bit of latency over our fixed-point attempt. We can take a look at our simulation to see what the actual latency looks like:



Figure 6.15 – Floating-point temperature simulation

Here, you can see how much latency has been added. We've gone from 5 clocks to about 50. We have plenty of time, so this isn't really a problem. Let's take a look at our resource usage:

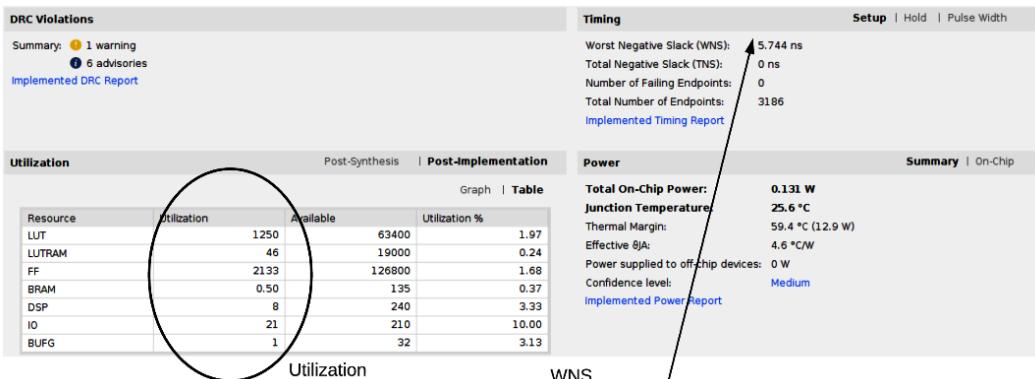


Figure 6.16 – Floating-point temperature utilization

Let's compare the utilization of our fixed-point implementation to our floating-point implementation:

Resource	Fixed	Floating	change
LUT	433	1250	2.82x
LUTRAM	13	46	3.53x
FF	264	2133	8.07x
BRAM	0	0.5	+0.5
DSP	1	8	8x

Even though the design is quite small, the difference is pretty remarkable in terms of latency and area.

This project should give you some insight into some of the advanced math options of using floating point, as well as how the AXI streaming interface can be used to connect multiple pieces of the IP together. You may have noticed that we didn't have to keep track of pipelining as we simply used the valid signals in and out of each core as our control signals.

With floating point out of the way, let's touch briefly on parallelism.

Parallel designs

FPGAs, being a blank slate, provide the fabric we can use to construct various applications. People use FPGAs for signal processing applications such as **software-defined radio (SDR)**, high performance computing applications, and, more recently, **artificial intelligence (AI)** and **machine learning (ML)**.

ML and AI and massive parallelism

In recent years, ML and AI have boomed. Self-driving cars, deep fake generation and analysis, and market predictions are but a few of the topics that these applications have been applied to.

It's easy to see why. The Artix part we are targeting has up to 240 DSP blocks. The largest Virtex Ultrascale+ that Xilinx makes has almost 4,000 DSP blocks and 9,000 Logic cells. Xilinx advertises up to 38.3 TOP/s for INT8 operations in the VU13P.

It's beyond the scope of this book to provide an overall introduction, but I would certainly encourage investigating the resources available for parallel designs.

Parallel design – a quick example

Let's take a look at a quick example that shows a massively parallel implementation. In this case, we want to create an adder tree that will output the sum of 256 inputs in 8 clock cycles.

Here, we need to discuss latency and throughput. **Latency** is the number of clock cycles (or time) it takes to produce a result. In the parallel example presented here, we have a latency of 8 clock cycles. Because the design is pipelined, we can produce a new result every clock cycle after the initial data, as long as new data is being fed in:

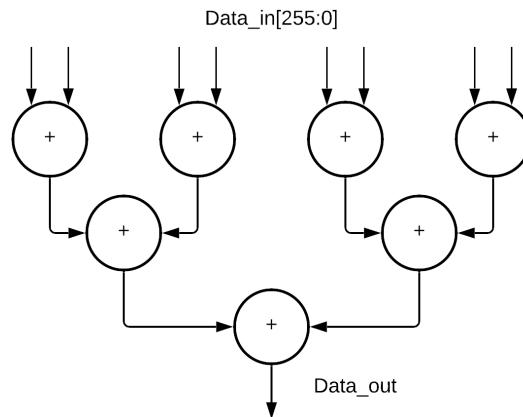


Figure 6.17 – Parallel design example

Let's see how this can be implemented in SystemVerilog:

```

always @(posedge clk) begin
    for (int i = 0; i < 128; i++)
        int_data0[i] <= in_data[i*2+0] + in_data[i*2+1];
    for (int i = 0; i < 64; i++)
        int_data1[i] <= int_data0[i*2+0] + int_data0[i*2+1];
    for (int i = 0; i < 32; i++)
        int_data2[i] <= int_data1[i*2+0] + int_data1[i*2+1];
    for (int i = 0; i < 16; i++)
        int_data3[i] <= int_data2[i*2+0] + int_data2[i*2+1];
    for (int i = 0; i < 8; i++)
        int_data4[i] <= int_data3[i*2+0] + int_data3[i*2+1];
    for (int i = 0; i < 4; i++)
        int_data5[i] <= int_data4[i*2+0] + int_data4[i*2+1];
  
```

```
for (int i = 0; i < 2; i++)
    int_data6[i] <= int_data5[i*2+0] + int_data5[i*2+1];
out_data <= int_data6[0] + int_data6[1];
int_valid <= int_valid << 1 | in_valid;
out_valid <= int_valid[6];
end // always @ (posedge clk)
```

This will create a tree of 255 adders. The operation is both parallel, in the sense that we are going to handle all the inputs simultaneously, and pipelined in that new data can be fed into every clock cycle. As we mentioned previously, after 8 clock cycles, we'll have the first sum available for use. Every cycle after that, a new sum will be available.

Summary

In this chapter, we took our temperature sensor project and improved upon it using fixed-point math. We removed our startup condition so that the temperature is output almost immediately and constantly filtered through the life of the design. We then looked at floating-point operations and converted the design into a floating-point pipeline. This led us to introducing AXI streaming, which will only become more important as we proceed throughout this book.

In the next chapter, we are going to delve further into AXI interfaces, package up some of our IP into AXI format so that we can reuse it, and introduce the IP integrator and block design tool.

Questions

1. If we have a large dynamic range in our numbers, what are we better off using?
 - a. Integers
 - b. Fixed point
 - c. Floating point
 - d. Imaginary

2. Which order represents the number complexity from least complex to most complex?
 - a. Fixed point, integers, floating point
 - b. Integer, fixed point, floating point
 - c. Floating point, fixed point, integer
 - d. Integer, floating point, fixed point
3. The following code is an example of what kind of design?

```
always @(posedge clk) begin
    if (stage[0]) out[0] <= fp_out[0];
    if (stage[1]) out[1] <= out[0] + fp_out[1];
    if (stage[2]) out[2] <= out[1] + out[0] + fp_out[2];
end
```

- a. Pipelined
- b. Parallel
- c. State machine
4. The following code is an example of what kind of design?

```
always @(posedge clk) begin
    for (int i = 0; i < 128; i++) dout[i] <= din[i*2] +
    din[i*2+1];
end
```

- a. Pipelined
- b. Parallel
- c. State machine
5. Which of the following signals makes up an AXI streaming interface?
 - a. tdata
 - b. tvalid
 - c. tready
 - d. tlast
 - e. tuser
 - f. taddr

6. A 16.16 * 8.16 fixed-point multiplier would result in an output of what?
 - a. 16.16
 - b. 17.16
 - c. 32.32
 - d. 24.32

Challenge

We are not using all of the seven-segment display. Earlier, we used an LED to indicate degrees Celsius or Fahrenheit. Can you modify the code so that it uses one (or two) of the seven segments to display C/F or °C/°F?

Further reading

Please refer to the following links for more information regarding what was covered in this chapter:

- The Nexys A7 reference manual: <https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/reference-manual>
- Temperature sensor specification: <https://www.analog.com/media/en/technical-documentation/data-sheets/adt7420.pdf>
- Xilinx DSP 48 users guide for 7-series parts (Artix-7): https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

Section 3: Interfacing with External Components

Up until now, we've limited our connections to the outside world to buttons, switches, LEDs, and 7-segment displays. We are going to update them to make them more reusable and add some more interesting components to the mix. By the end of this section, we will be able to replace the buttons and switches with a PS/2 keyboard and replace the LEDs and 7-segment display with a **Video Graphics Array (VGA)** controller.

This part of the book comprises the following chapters:

Chapter 7, Introduction to AXI

Chapter 8, Lots of Data? MIG and DDR2

Chapter 9, A Better Way to Display – VGA

Chapter 10, Bringing It All Together

Chapter 11, Advanced Topics

7

Introduction to AXI

As **Field-Programmable Gate Arrays (FPGAs)** became larger and more complex, vendors such as Xilinx began offering **Intellectual Property (IP)**, designed and tested to accelerate design implementation. These first IPs often had simple interfaces, sometimes referred to as native interfaces. Xilinx offered early high-end parts with PowerPC cores and their own MicroBlaze cores, each of which had differing interfaces. When Xilinx adopted ARM processors as part of their Zynq family, they standardized the ARM processor interfaces, using the **Advanced eXtensible Interface (AXI)**. In order to best use Xilinx IPs, we have already looked at the streaming interface. There are two other interfaces that are commonly used: AXI-Lite and AXI full.

By the end of this chapter, you'll have a good handle on the flavors of AXI and when to use them. You'll know how to create your own IPs using AXI to make integration with other IPs easier. Finally, you'll have developed a temperature sensor using AXI and the IP integrator.

In this chapter, we are going to cover the following main topics:

- AXI streaming
- Project 9 – creating IPs for Vivado using AXI streaming interfaces
- Introduction to the IP integrator
- AXI4 interfaces (full and AXI-Lite)
- Developing IPs – AXI-Lite, full, and streaming

Technical requirements

The technical requirements for this chapter are the same as those for *Chapter 1, Introduction to FPGA Architectures and Xilinx Vivado*.

To follow along with the examples and the project, you can find the code files for this chapter at the following repository on GitHub: <https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH7>.

AXI streaming

We took a brief dip into AXI and the streaming interface in *Chapter 6, Math, Parallelism, and Pipelined Design*. AXI streaming is used primarily as a lightweight conduit to move data between two points, as shown in *Figure 7.1*:

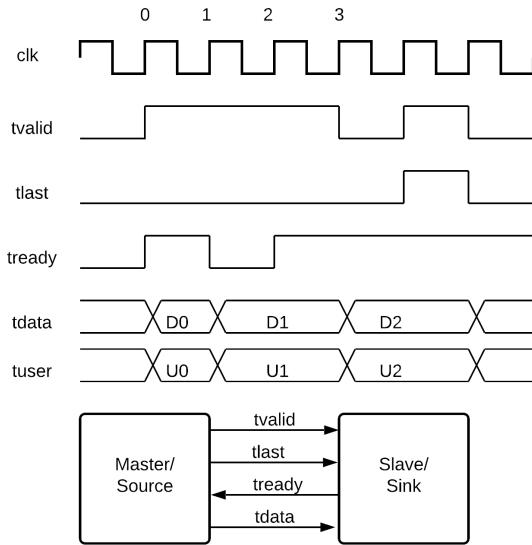


Figure 7.1 – AXI streaming with an optional tuser signal

There is an optional sideband signal included for completeness, **tuser**. This signal can be passed along with the stream, but it's up to the source and sink to understand how this signal is used.

Before we dive into the other AXI types, let's break up our I2C temperature sensor into AXI streaming-based IPs.

Project 9 – creating IPs for Vivado using AXI streaming interfaces

In this project, we are going to take our I2C temperature sensor and split it into IPs that we can use in the IP integrator to reconstruct our project.

Our initial design looked like this:

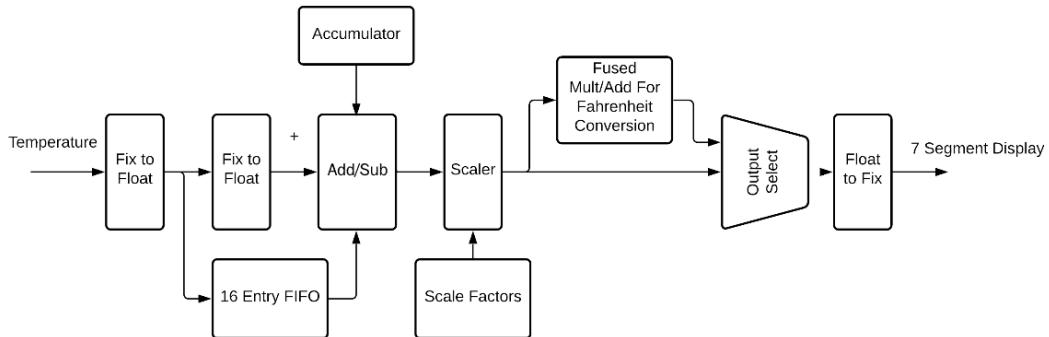


Figure 7.2 – Original temperature sensor pipeline

Looking at the Xilinx floating-point IP, fix to float, float to fix, add/sub, scaler, and fused multiply/add are all IP blocks with streaming interfaces. What we need to address is the I2C interface that reads the temperature from the ADT7420, the temperature pipeline itself, and the seven-segment display interface. Let's tackle the seven-segment display first.

Seven-segment display streaming interface

The first thing we need to do is create a directory to house our IP sources. This will make packaging easier. We'll do this by creating a directory under CH7/build/IP/seven_segment. Inside this directory, we have an hd1 directory that contains the stripped-down seven-segment portion of our temperature sensor.

If we abide by a few rules, it will be easier to create the IP. The complete manual can be found at https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug1118-vivado-creating-packaging-custom-ip.pdf. We have already named our clock signal clk, which is one of the ways the tool can automatically identify clock signals. To create an AXI streaming bus that can be extracted automatically, we need to name our interface signals in a consistent way:

- <interface name>_tdata (required)
- <interface name>_tvalid (required)

- <interface name>_tready
- <interface name>_tstrb
- <interface name>_tkeep
- <interface name>_tlast
- <interface name>_tid
- <interface name>_tdest
- <interface name>_tuser

Our port list for the IP now looks like this:

```
module seven_segment
#(
    parameter NUM_SEGMENTS = 8,
    parameter CLK_PER      = 10, // Clock period in ns
    parameter REFR_RATE    = 1000 // Refresh rate in Hz
) (
    input wire                  clk,
    input wire                  seven_segment_tvalid,
    input wire [NUM_SEGMENTS*4-1:0] seven_segment_tdata,
    input wire [NUM_SEGMENTS-1:0] seven_segment_tuser,
    output logic [NUM_SEGMENTS-1:0] anode,
    output logic [7:0]           cathode
);
```

anode and cathode will become external interfaces on the board. We've named our AXI streaming interface `seven_segment` and `clk` should be recognized without any special handling.

Let's look at packaging our IP:

1. The first step in this process would be to create an IP:

tools->Create and package IP



Next

Figure 7.3 – Creating and packaging the IP

The first dialog box simply summarizes what we are going to do. We've modified the code to encapsulate the IP.



Figure 7.4 – Packaging the directory

2. Make sure to select **Package a Specified Directory** and click **Next**.



Figure 7.5 – Specifying the source directory

3. Make sure that **Package as a library core** is unchecked. Select the **seven_segment** directory under **Create and Package New IP**.



Figure 7.6 – Destination directory

4. We need to specify the project name and the destination location. In this case, we'll create an `ip_repo` directory within the build directory. Then, select **Next**.



Figure 7.7 – Basic settings

5. You can define the basic IP settings on the page shown in the preceding screenshot. Let's verify the ports and interfaces, as shown in the following figure:

Ports and Interfaces										
Name	Interface Mode	Enablement Dependency	Direction	Driver Value	Size Left	Size Right	Size Left Dependency	Size Right Dependency	Type	Name
seven_segment	slave		in	0	31	0	((NUM_SEGMENTS * 4))		wire	
seven_segment_tdata			in	0	7	0	(NUM_SEGMENTS - 1)		wire	
seven_segment_tuser			in						wire	
seven_segment_tvalid			in						wire	
Clock and Reset Signals										
clk	slave									
anode			out		7	0	(NUM_SEGMENTS - 1)		logic	
cathode			out		7	0			logic	

Figure 7.8 – Verifying ports

- We can see that the IP packager was able to find the clock port and the seven-segment AXI slave port automatically. The other two ports are simply brought out as is.



Figure 7.9 – Review the IP

- Finally, review the IP and then press **Package IP**.

At this point, the IP is ready to use. You will notice that I've added the optional `tuser` interface in this version of the streaming interface. I did this because we need a way of passing along the digit point for the display. Since there is no standard for using `tuser`, it's important that the master interface knows how to drive it and the slave knows how to interpret it.

I've packaged up two more IPs for us to use: the `adt7420_i2c` interface and the floating-point temperature sensor core. The reason I created the temperature sensor core itself is so that we can use the **Block Design (BD)** tool as an alternative to developing this version of the design. We could have kept our old top level and simply instantiated our two new cores. Typically, I prefer to work in SystemVerilog directly, but there are times, such as when developing an **FPGA System on Chip (SOC)**, when you will need to use the **BD** tool for at least some of the design.

Let's take a quick look at the ADT7420 code that makes up our new IP.

Developing the ADT7420 IP

First, let's take a look at the new top level for our IP:

```
module adt7420_i2c
#(
    parameter INTERVAL      = 1000000000,
    parameter CLK_PER       = 10
) (
    input wire              clk, // 100Mhz clock
    // Temperature Sensor Interface
    inout wire              TMP_SCL,
    inout wire              TMP_SDA,
    inout wire              TMP_INT,
    inout wire              TMP_CT,
    output logic             fix_temp_tvalid,
    output logic [15:0]      fix_temp_tdata
);
```

We've got our clock, our I2C bus, and connections to the temperature-sensing chip.

This file contains the I2C state machine. Feel free to take a look. The difference is that the output of the temperature is now AXI streaming. The IP is already built, but if you want, you can build it again as an exercise.

Finally, we can take a quick look at the core of the design where we perform the floating-point temperature smoothing, Fahrenheit conversion, and output to our seven-segment display IP.

Understanding the `flt_temp` core

The floating-point temperature sensor core is the most complex block in our design. It connects to our other two IP cores as well as the floating-point logic we previously generated. The core of the `flt_temp` module is unchanged from our previous design; however, we now have a new interface definition for the AXI streaming interfaces.

I encourage you to take a look at the **Hardware Description Language (HDL)**. We'll see in the next section as we build the BD what the cores look like.

IP integrator

The IP integrator provides a graphical user interface for hooking up IP cores using schematic capture with BD. We've gone through the preceding steps for our three IP cores. This procedure has added the cores to our project for use:

1. Our first step is to create our BD and add the cores:



Figure 7.10 – Create Block Design

Selecting **Create Block Design** will bring up the tool we'll be using in this section.



Figure 7.11 – Adding an IP

- With the main BD window up, the first thing to do is add the IP. We'll start with our `adt7420_i2c` core IP and take a look at it.

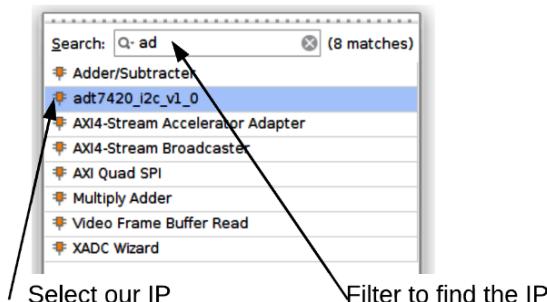


Figure 7.12 – Selecting our IP

3. Clicking + will bring up a popup to search for our IP. We can type in adt and find the interface we are looking for. Double-click to add to the design. Now we can take a look at what the core looks like when placed and the options we can set:



Figure 7.13 – IP core configuration

If you recall from our original design, we specified two parameters for this core. Unless you modify them, they will appear as is within the core configuration when you double-click on the core instance in the block diagram:

```
module adt7420_i2c
#(
    parameter INTERVAL      = 1000000000,
    parameter CLK_PER       = 10)
```

Since the parameters in our cores are already set to what we need for the Nexys A7 board, we don't need to change anything. Take a look at the design instance and you'll see our port list:

```
input wire          clk, // 100Mhz clock
// Temperature Sensor Interface
inout wire         TMP_SCL,
inout wire         TMP_SDA,
inout wire         TMP_INT,
inout wire         TMP_CT,
```

```

output logic      fix_temp_tvalid,
output logic [15:0] fix_temp_tdata

```

Because the streaming interface was identified when packaging the IP, it's collapsed in the diagram and marked as `fix_temp`. The clock port will be connected to our clock source and we'll hook up the `TMP_*` ports to external interfaces:

1. Let's continue and add all of the IPs we'll need for the design:

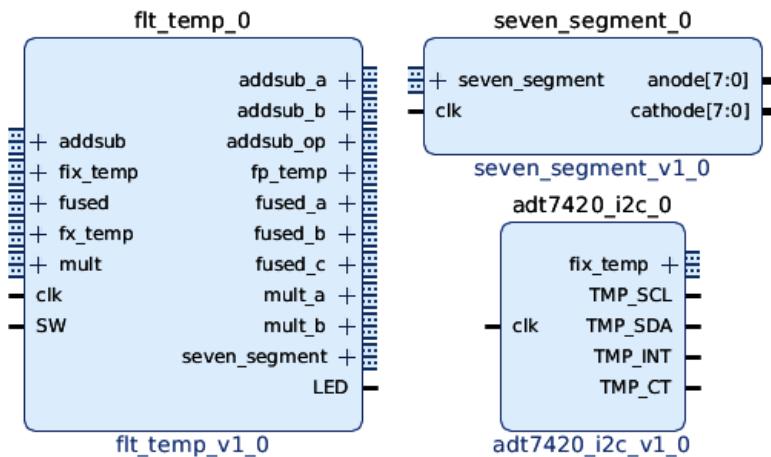


Figure 7.14 – User IPs added to the design

We still have quite a few IPs to add to our design; we need all our floating-point cores. Take a look back at *Figure 7.2* if you need to, but our names should also give us a clue as to what's needed. All the floating-point cores are simply configurations of the floating-point IP, so search in the add IP popup for `floating point`. We'll need to do this step five times.



Figure 7.15 – Block properties

The default name for the IP is simply the IP name with an incrementing number appended to it. We can rename our instances so they are more user-friendly. Select the IP after you configure it and then on the left side of the BD window, you'll see the **Block Properties** pane. This gives easy access to the configuration as well as the ability to change the instance name.

- Let's continue with the rest of the floating-point IPs:

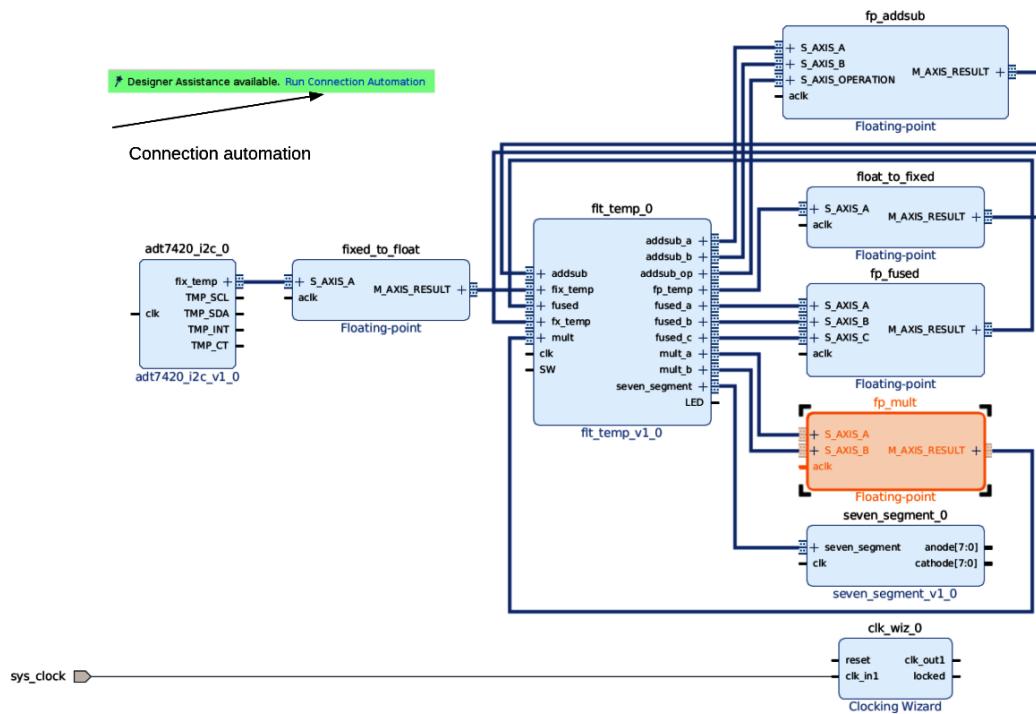


Figure 7.16 – Full design

We can now look at the design. We'll need to hook up the streaming interfaces manually as I did in *Figure 7.16*. You'll also have noticed that a banner appeared at some point when adding the IP telling you that the BD tool recognizes that it can automate some of the connections for you:



Figure 7.17 – Connection automation

3. In this case, the tool will connect up our clocks and system reset for us. Once that's complete, we can connect up our external ports. In this case, we'll do it manually.



Figure 7.18 – Validate Design

When you have completed your schematic, click the validate design button. This will ensure that there are no errors with your design.

Important note

If you were building something with AXI-Lite or full interfaces, you would need to make sure the addressing was set up prior to building. We'll look at that in the next chapter.

4. The design has no errors, so we need to build the output products and then generate the bitstream:

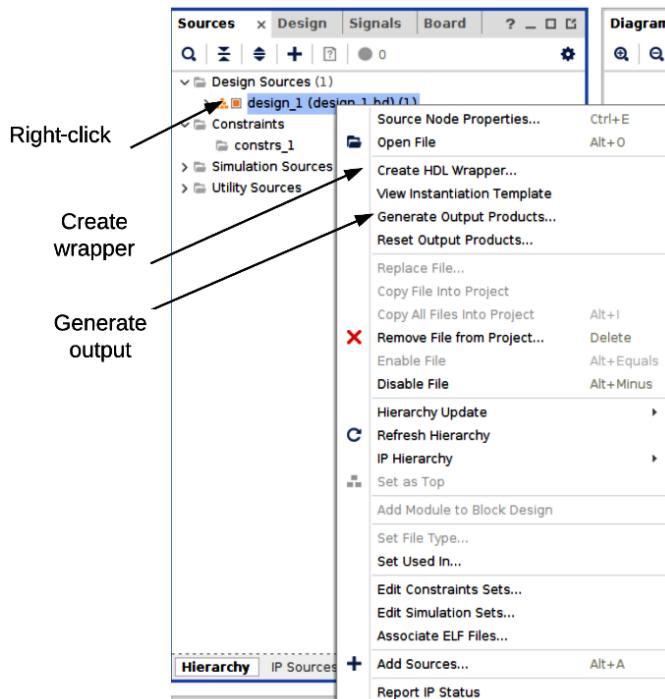


Figure 7.19 – Generating output products

5. One important thing is that our design has tri-state buffers internally. In order to have the design function properly, we need to select **Global** for the synthesis options. This appears to be a limitation that doesn't allow you to use the **Out Of Context (OOC)** synthesis if tri-state ports are embedded in the design:



Figure 7.20 – Global synthesis

6. Finally, we need to create the HDL wrapper, which will become the top level of our design (refer to *Figure 7.18*):



Figure 7.21 – Create HDL Wrapper

Let's take a look at the final design. Here the schematic is complete:

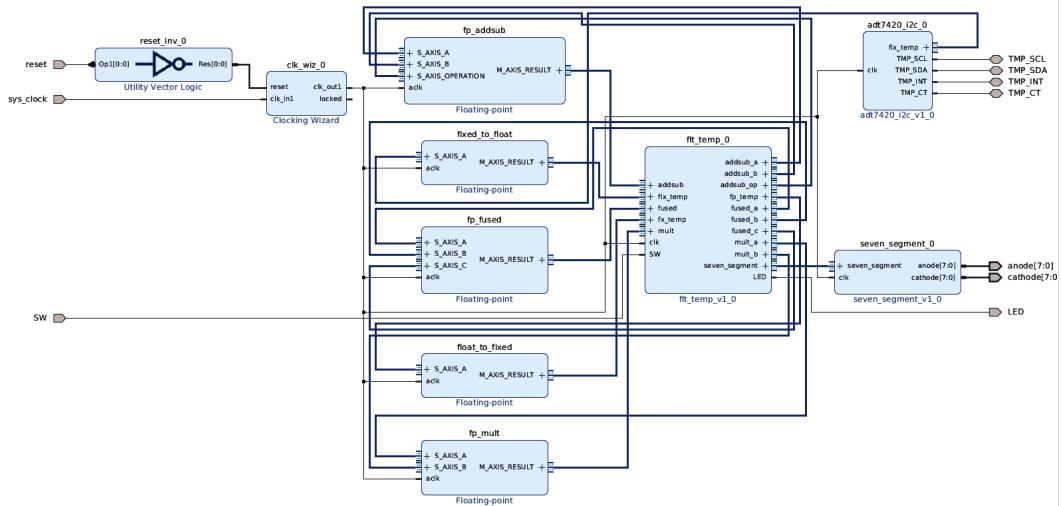


Figure 7.22 – Completed BD

Functionally, this design will be equivalent to what we developed in *Chapter 6, Math, Parallelism, and Pipelined Design*. This is simply another way of jumpstarting your designs if you are using a lot of IPs. It is most useful when using processors and is practically required for **SOC** designs. In the end, you can embed your own IPs as we have done or actually package up your BD and include it in an HDL design.

Let's look at debugging our design.

IP integrator debugging

The IP integrator makes it easy to debug your design. Simply right-click on any net or bus and select **Debug**. This is especially useful when used on AXI buses as the **Integrated Logic Analyzer (ILA)** understands the bus structure and the transaction protocol and will display information in a meaningful way:

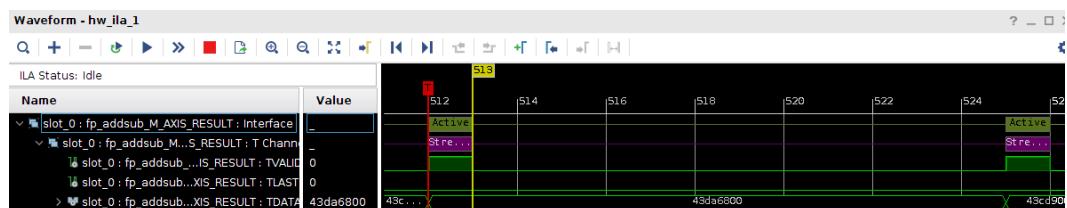


Figure 7.23 – AXI Streaming ILA

Add a debug core by right-clicking on the `fp_addsub` master output. Generate the bitstream and set a trigger on the rising edge (R) of `tvalid`. You should trigger on a pair of transactions. Try adding some other ILAs and watch the transactions. As the temperature sensor is running, you can trigger on the `tvalid` signals and watch as data propagates through the floating-point pipeline.

Now that we have examined using the IP integrator to build and debug a design, let's take a look at the other two flavors of AXI interfaces we will be using throughout the rest of the book.

AXI4 interfaces (full and AXI-Lite)

The AXI4 interface is a full-featured processor interface used by ARM to allow the easy connection of peripherals to their processors. Xilinx has adopted this interface to connect its hard and soft processors to other cores, whether AXI-Lite, full, or streaming. Because it is full-featured, it can be costly to implement and should really only be considered when you need an addressable interface with high-performance bursting capability. There are five components to an AXI full or AXI-Lite interface. Reads consist of an address component and data component:

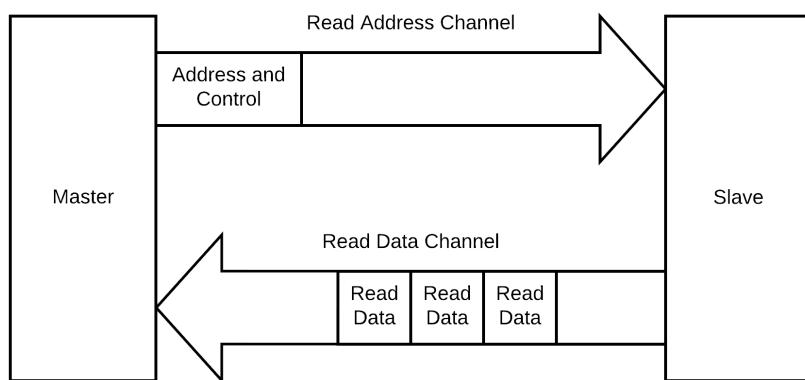


Figure 7.24 – AXI read channel

The preceding figure conceptually shows how a read operation in AXI occurs. An address and a control bus signal the slave to perform a read. In an AXI-Lite interface, this is a single location; in a full interface, it can be for a burst of data. These types of reads are posted reads, meaning that if the interface supports it, multiple read requests can be made before data starts coming back.

When the slave interface is ready, it can start sending data back. If the master and slave support it, the slave can perform out-of-order reads, using the ID channel to signal which group the data is associated with. This can allow increased performance in not having to reorder data.

Writes have three components: the write address, write data, and write response channels:

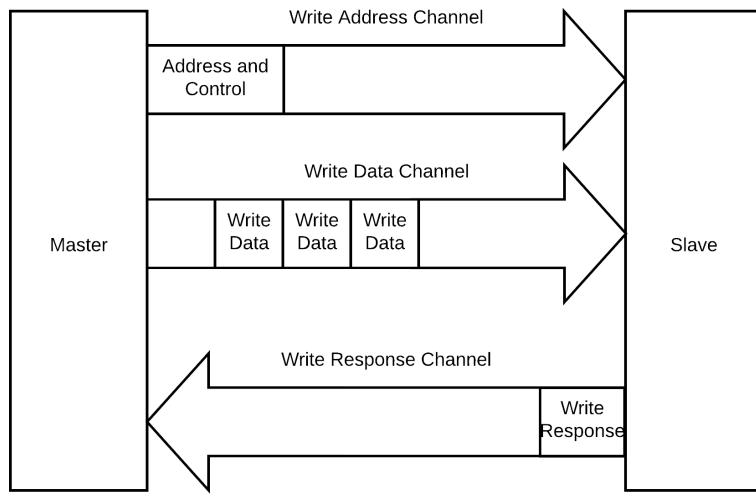


Figure 7.25 – AXI write channel

The master will issue write commands via the write address channel, which is either preceded by data or followed by write data. The slave will respond via the write response channel, which will signal whether the write completed successfully or with an error.

The main differentiators between full and AXI-Lite interfaces are bursting capabilities for full interfaces and single beat transfers for AXI-Lite interfaces. AXI-Lite interfaces have a few more restrictions, such as smaller bus width support and non-cacheable and normal non-locked accesses.

There are a variety of cores available for instantiating in BD or instantiating in your design:

Search: <input type="text" value="axi"/> (68 matches)
AXI AHBLite Bridge
AXI APB Bridge
AXI BRAM Controller
AXI CAN
AXI Central Direct Memory Access
AXI Chip2Chip Bridge
AXI Clock Converter
AXI Crossbar
AXI Data FIFO
AXI DataMover
AXI Data Width Converter
AXI Direct Memory Access
AXI EMC
AXI EPC
AXI EthernetLite
AXI GPIO
AXI HB ICAP
AXI HWICAP
AXI IIC
AXI Interconnect
AXI Interrupt Controller
AXI Memory Init
AXI Memory Mapped to Stream Mapper
AXI MMU
AXI Multi Channel Direct Memory Access
AXI Performance Monitor
AXI Protocol Checker
AXI Protocol Converter
AXI Protocol Firewall
AXI Quad SPI
AXI Register Slice
AXI Sideband Utility
AXI SmartConnect
AXI TFT Controller
AXI Timebase Watchdog Timer
AXI Timer
AXI Traffic Generator
AXI UART16550
AXI Uartlite
AXI USB2 Device
AXI Verification IP
AXI Video Direct Memory Access
AXI Virtual FIFO Controller
DFX AXI Shutdown Manager
JTAG to AXI Master
PR AXI Shutdown Manager
Video AXI4S Remapper
Video In to AXI4-Stream

Figure 7.26 – Available AXI IPs

A typical design would consist of a microcontroller, such as the ARM Cortex or Xilinx MicroBlaze, or your core logic and connected peripherals. The peripherals typically connect to an AXI interconnect block, which can become costly to implement depending on the number of active master interfaces, whether buffering is enabled, and whether it's a full crossbar or not.

We'll take a quick look at an alternative way to package a core from scratch.

Developing IPs – AXI-Lite, full, and streaming

We'll take a look at how we can develop an IP through packaging it by defining the interfaces first:



Figure 7.27 – Creating a new AXI4 peripheral

This is a way of creating an IP by creating a wrapper first and then inserting your IP:



Figure 7.28 – Defining the IP

We'll create a pdm_capture module that will have a register to trigger a read. We can then read back the same register to determine whether the read is completed. Data can then be read from a second register.



Figure 7.29 – Default interface definition

The default interface definition is perfect for what we need. You can investigate the options and see that it is very easy to add any of the AXI interfaces we've discussed. If you explore the IP directory, you'll see the following files created under the HDL directory:

- `pdm_capture_v1_0.v`: The top level of our IP. We'll add our interface to the microphone here.
- `pdm_capture_v1_0_S00_AXI.v`: The AXI portion of the design with registers.

You'll see in both generated modules places to put logic and ports:

```
module pdm_capture_v1_0_S00_AXI #
(
    // Users to add parameters here
    Place USER parameters here
    // User parameters ends
    // Do not modify the parameters beyond this line
```

Looking at the port list, you can see where to place the ports needed for the top level:

```
(

    // Users to add ports here
    Place USER ports here
    // User ports ends
    // Do not modify the ports beyond this line
```

Finally, within the design itself, there are bracketing comments on where to add your code:

```
// Add user logic here
Place USER logic here
// User logic ends
```

There is an additional way to add an IP in the IP integrator that doesn't involve explicitly creating an IP block; however, currently, it does require that the top level of the IP be a Verilog file and not SystemVerilog.

Adding an unpackaged IP to the IP integrator

At https://github.com/PacktPublishing/Learn-FPGA-Programming/blob/master/CH7/build/i2c_temp_flt_bd/i2c_temp_flt_bd.xpr, there are two files, adt7420_i2c_mod.sv and adt7420_i2c_bd.v. Now, adt7420_i2c_mod.sv is a copy of the adt7420_i2c.sv IP file that is renamed so it doesn't cause problems with a naming conflict. adt7420_i2c_bd.v provides the Verilog wrapper.

Add an IP by doing the following:

1. Open https://github.com/PacktPublishing/Learn-FPGA-Programming/CH7/build/i2c_temp_dlt_bd/i2c_temp_dlt_bd.xpr.
2. Right-click on **Design Sources** and add the files as shown in *Figure 7.30*:

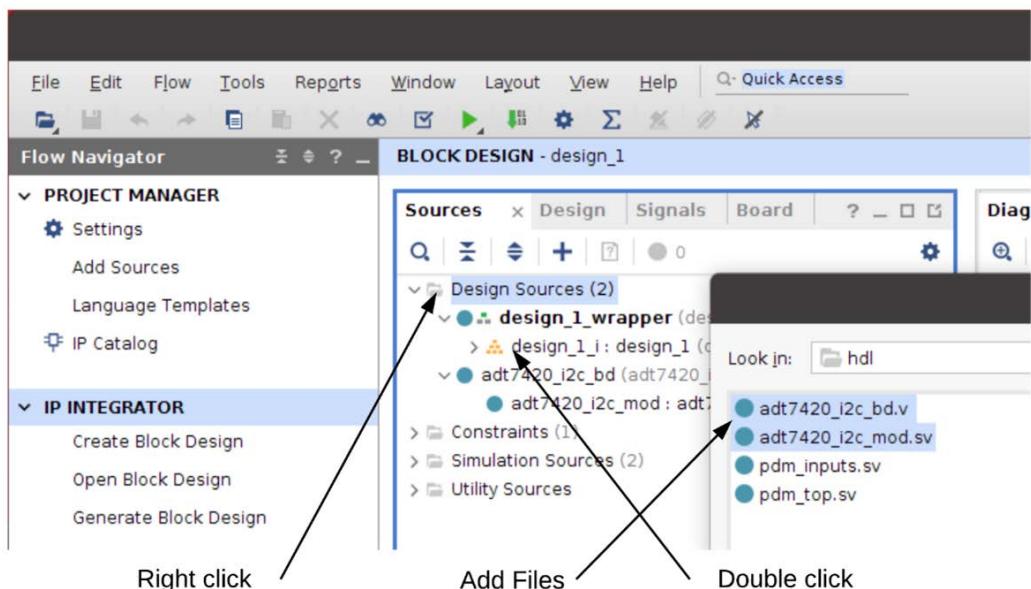


Figure 7.30 – Adding files to the design

3. Open the IP integrator.
4. In the BD, right-click and select **Add Module**:

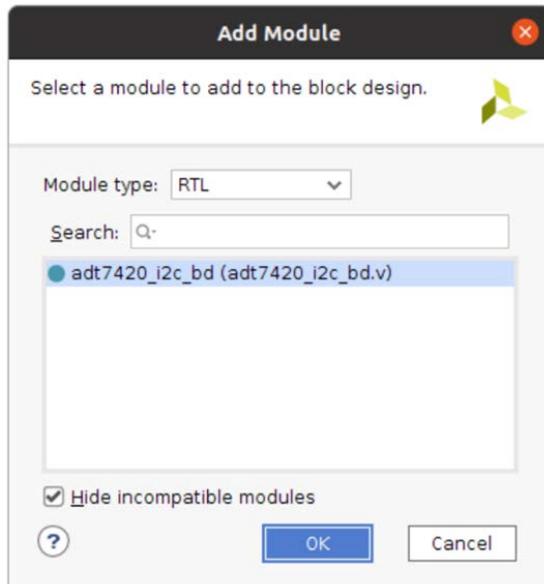


Figure 7.31 – Adding a module to the IP integrator

5. Select **OK**.

Now we can compare the HDL module to the IP we created earlier:

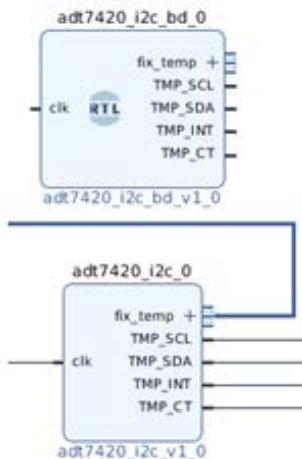


Figure 7.32 – HDL module versus created IP

The RTL module acts just like an IP block created earlier in the chapter. You can modify the parameters and the interfaces are detected if named properly.

We'll be leaving the IP integrator and will explore other FPGA features in the next few chapters.

Summary

We've seen how to generate IPs from an existing SystemVerilog file and used this to recreate our temperature sensor project using the IP integrator. We looked at how we can easily debug using the IP integrator and how the ILA is AXI-aware. We've also looked at how we can package IPs by using the IP packager to generate a wrapper with AXI interfaces that we can use to create our core designs.

We've gone from flashing LEDs in *Chapter 1, Introduction to FPGA Architectures and Xilinx Vivado*, to using a seven-segment display to display information in *Chapter 3, Counting Button Presses*. In *Chapter 8, Lots of Data? MIG and DDR2*, we are going to look at developing a display controller using the **Video Graphics Array (VGA)** interface, which will give us much more capability in displaying the outputs from our temperature sensor, microphone, and calculator.

Questions

1. What are AXI streaming interfaces best for?
 - a) Burst transactions to multiple memory addresses
 - b) Point-to-point connections
 - c) High-performance connections
 - d) B&C
2. What is the IP integrator?
 - a) An easy way to create block-based designs using Xilinx or user-defined IP
 - b) A context-sensitive editor for HDL designs
 - c) Not very good at aiding design debug
3. If you want to create an IP from an existing design, you would use **Create and package new IP**. True or false?
4. You cannot use **Create and package new IP** to generate a design wrapper with AXI interfaces to create your own designs. True or false?

5. When should full AXI interfaces be used?
 - a) When you need a high-performance interface that can burst data to multiple memory addresses.
 - b) When you only write a single register at a time infrequently.
 - c) When you have lots of data to move between two cores where the destination is a FIFO-like interface.
 - d) All the time. They are cheap to implement and can do everything.

Further reading

For more information about what was covered in the chapter, please refer to the following:

- https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug1119-vivado-creating-packaging-ip-tutorial.pdf
- <https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification>

8

Lots of Data? MIG and DDR2

We've been working our way up toward a more functional design that can gather information, do some useful work, and present it in a meaningful manner. In previous chapters, we captured audio data and temperature data. We also looked at wrapping some of the interfaces so that we could use the IP integrator . The IP integrator also allowed for easily instancing floating-point operations. This has given us some functional designs, but we've been limited to using LEDs and then seven-segment displays, making it difficult to visualize information such as the PDM waveform data or even the temperature.

We have another option when it comes to displaying using our boards: the VGA connector. To effectively use it, we will need access to quite a bit of memory. To display 640x480 8-bit color, we would need 300 kilobytes, almost 1 megabyte for true color. We can certainly play some games to stretch out our memory, but alternatively, we can use the onboard **Double Data Rate, 2nd generation (DDR2)** as a frame buffer and draw what we want to be displayed into it.

By the end of this chapter, you'll have been introduced to external memory, generated a DDR2 controller, and tested it both in simulation and on the board. You'll be comfortable with how to use external memory in your own designs.

In this chapter, we are going to cover the following main topics:

- DDR memory basics
- Using the Xilinx **Memory Interface Generator (MIG)**
- A brief look at other memory types

Technical requirements

The technical requirements for this chapter are the same as those for *Chapter 1, Introduction to FPGA Architectures and Xilinx Vivado*.

To follow along with the examples and the project, you can find the code files for this chapter at the following repository on GitHub: <https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH8>.

Project 10 – introducing external memory

Up until now, we've been using internal **Block RAM (BRAM)** or distributed RAM. These types of memory are very fast. BRAMs can be accessed in a single clock cycle up to the **maximum frequency (fmax)** of the device given certain constraints. **Look up table memories (LUTRAMs)** are a little more flexible in that they can be used asynchronously. Both types of memory are very convenient for small storage, lookup tables, fast memory for things such as cache, and if you have enough for a design, keeping costs and complexity down.

There are many external memory types available for use in designs. Looking just at synchronous **Dynamic RAMs (DRAMs)** that are still available, we can see how the performance has changed with each generation:

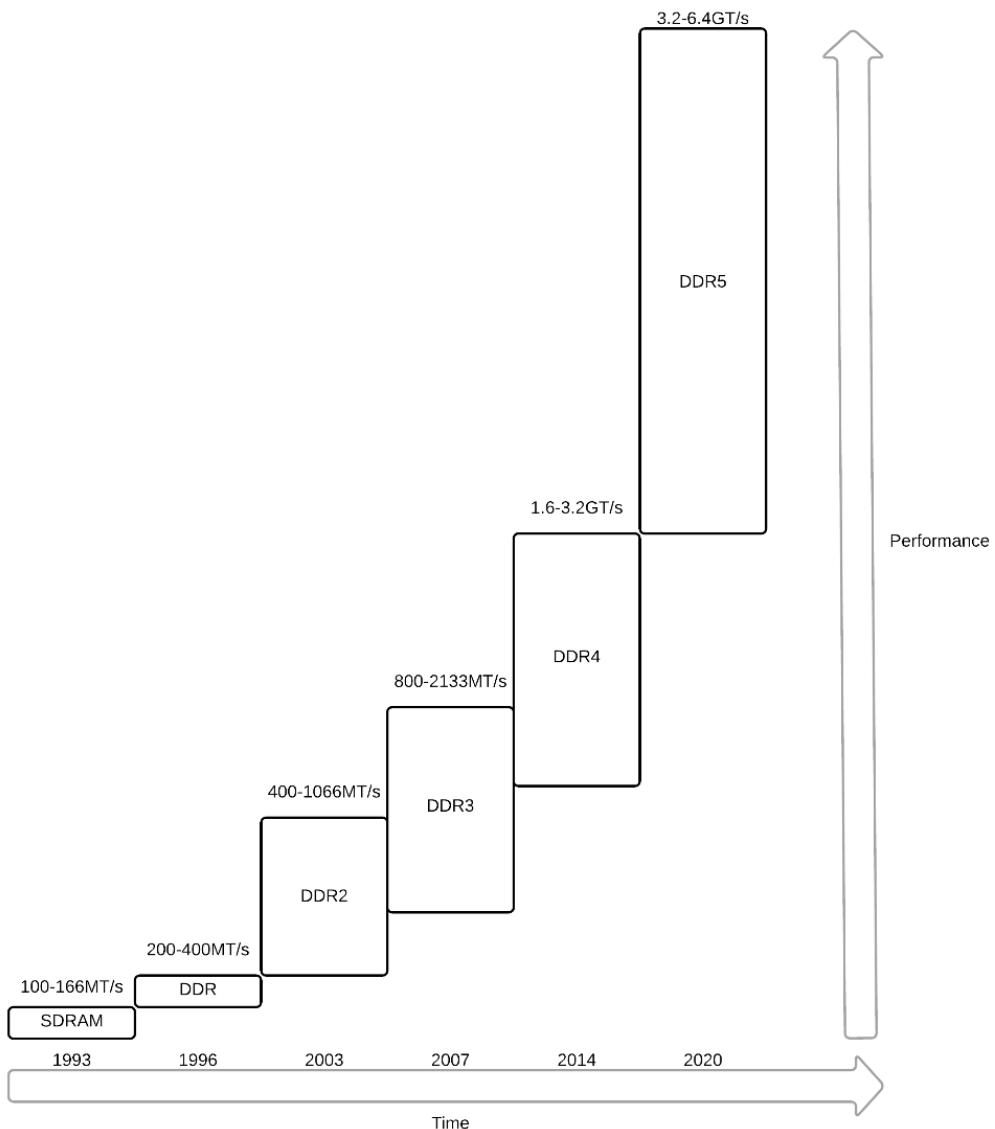


Figure 8.1 – External DRAM performance

Looking at the preceding chart, the first question would be what is the performance of the internal BRAM versus the external memory? What are the trade-offs? A single FPGA BRAM would be in the range of 300 MT/s. As we discussed previously, we can take advantage of parallelism and access many BRAMs in parallel, although effectively using all that memory would require a highly parallel structure.

The disadvantage of external memory and each passing generation is that latency, the time it takes from requesting data to getting it back, goes up. SDRAM can take five or so clocks at 100 MHz. DDR4 can take 80 clocks at 300 MHz. The logic to interface to the memories also gets very complex. You can write a DDR2 controller yourself; I have done it before. DDR4 controllers from Xilinx have embedded processors to handle initialization and periodic operations. When we generate our memory controller, we can look at the timing in a simulation.

We won't be tackling memory controller design here. If you are interested, I would advise looking for SDRAM or DDR memory controllers online. There are many freely available.

Finally, as we mentioned previously, DRAM brings with it capacity. Newer FPGAs have optional **High-Bandwidth Memory (HBM)** that provides internal DRAM with large capacities, but for most FPGAs, the cost-effective solution is to use external memory when capacity is important.

With our introduction out of the way, let's look at DDR2 specifically.

Introduction to DDR2

Behind the scenes, there is a lot that the Xilinx memory controller handles for us. BRAM/LUTRAM is **Static RAM (SRAM)**. Static memory is very large compared to dynamic memory. Static memory can take four transistors per bit (4T) cell. Dynamic memory, in contrast, is primarily a capacitor used to hold a charge with a transistor attached to it. It is also built on an optimized ASIC process to minimize the area of the chip. This allows for much higher densities, as we see in *Figure 8.1*.

Dynamic memory is partitioned into **rows**, **columns**, and **banks**. A single row containing multiple columns within a bank can be active at any one time. This is accomplished by sending an activate command to the memory. Once a row is opened in a particular bank, the columns can be accessed very rapidly.

Within a DRAM, you can have multiple banks, each with a different row open at the same time. This allows quicker access to larger blocks of data. When you want to switch rows within a bank, the open row must be precharged to close it before the new bank/row can be activated.

Finally, a refresh command must be issued periodically to the DRAM, which requires all open banks to be precharged prior to the refresh. The refresh reads out a row and writes it back again to refresh the charges in the capacitors holding the data.

With this background behind us, let's look at the steps to generate memory.

Generating a DDR2 controller using the Xilinx MIG

The Nexys A7 shares the same pinout as the Nexys DDR and you can find a premade project on the website at https://digilentinc.com/reference/programmable-logic/nexys-4-ddr/start?_ga=2.168036321.1345263114.1604794648-84804473.1599434198#additional_resources. We will, however, go through generating a component here so that you can see the options and how they relate to the underlying DDR2 architecture:

1. First, we need to use the IP catalog to generate the MIG controller:

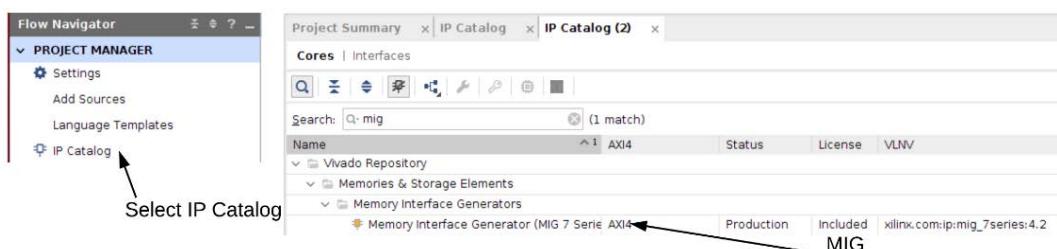


Figure 8.2 – Start MIG

2. Select IP Catalog and search for MIG. Then, double-click to start the MIG wizard.

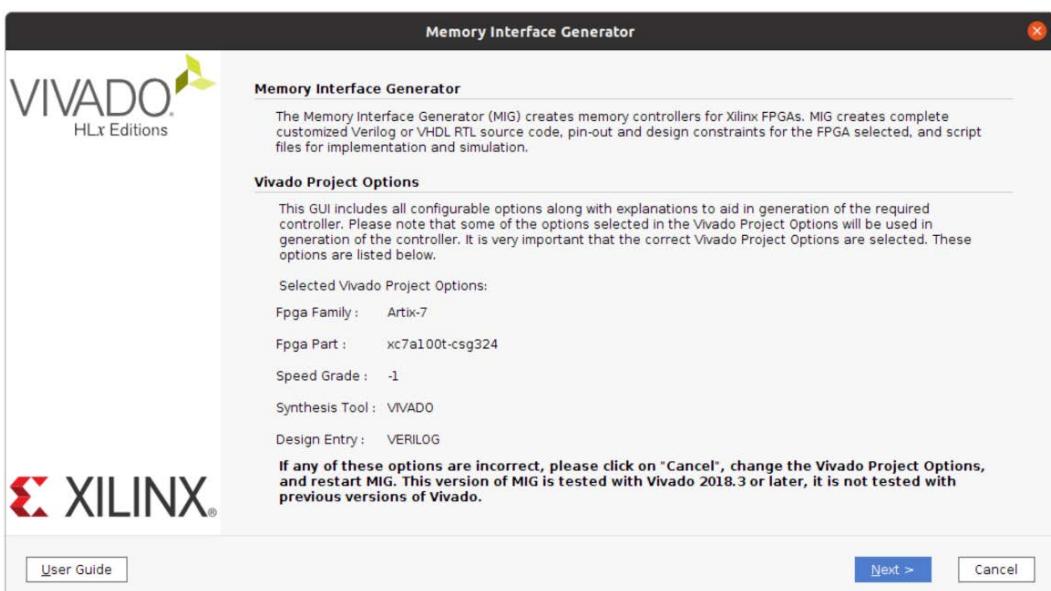


Figure 8.3 – MIG start screen

3. Your project should already be set up for our board. Confirm the FPGA part for the board, then select **Next**:

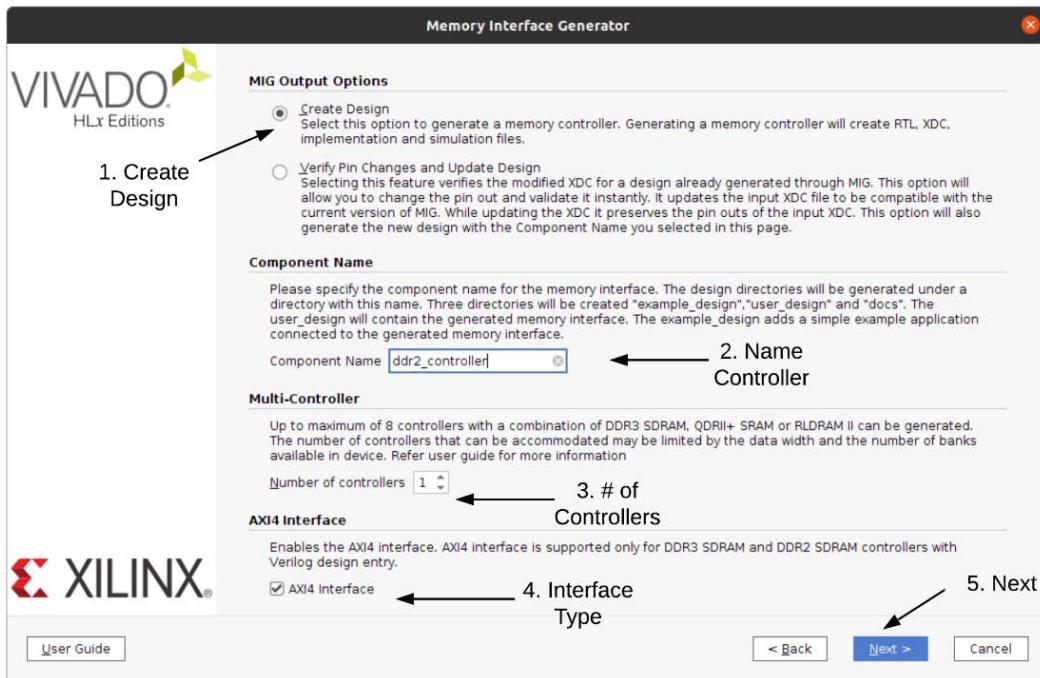


Figure 8.4 – MIG options

4. Now, we need to select **Create Design** to generate our new MIG design. I've renamed the controller to something more meaningful. On our board, we only need one controller and I've chosen an AXI4 interface as opposed to a native interface. Then, select **Next** so that we can look at pin compatibility:



Figure 8.5 – Pin compatibility

5. If you were generating a MIG controller for your own board, you may want to have the option of using a larger or smaller part. In this case, you could select multiple pin-compatible FPGAs to make sure your MIG controller would work in any part you eventually put on your board. In this case, we can select the 50 and 100 ones, but Digilent has already picked the correct spot for both boards. Select **Next** to choose the memory type:



Figure 8.6 – Memory selection

6. The Artix 7 supports DDR2 and 3 using the MIG. Our board has DDR2, so make sure to pick that one. Although all DDRs have similar names, they are not compatible, so make sure you pick the correct one for your board. Select **Next** so that we can select the controller options:



Figure 8.7 – Controller options

Looking at *Section 3.1* of the Nexys A7 documentation (link in *Further reading*), it recommends the settings we will use in this section. Most importantly, we need to select the correct part. Digilent also recommends a slightly slower clock speed for easier implementation. Select a data width of 16. There are two more options that are more or less optional.

The first is the number of bank machines. Remember that the device has eight banks. To improve performance, we can implement multiple bank machines at the expense of area in our design.

Finally, we can choose to maintain the strict ordering of requests or allow reordering. If we allow reordering, we can improve performance.

Next, we'll look at the AXI parameters.

Setting AXI parameters

There are a few parameter options that we have control over:

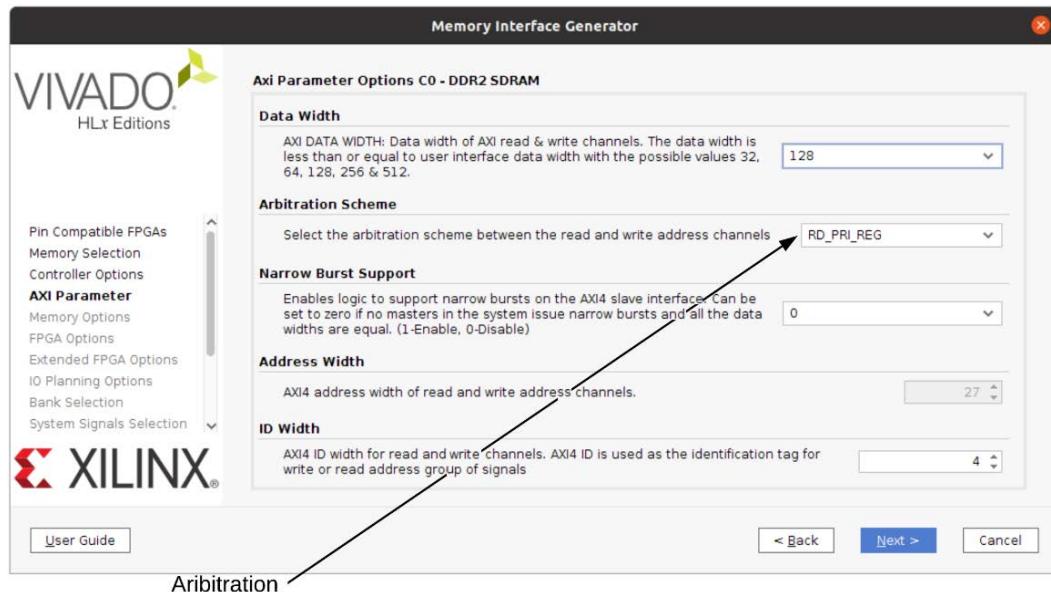


Figure 8.8 – AXI parameters

The main thing we need to consider for AXI parameters is the arbitration scheme. Because a major aspect of the controller is to handle our display, we need to make sure we never starve our display controller. To do this, we specify a read priority for arbitration. In the next section, we'll set the memory options.

Setting memory options

There are two options we need to set according to the Digilent documentation. These are shown in *Figure 8.9*. The first is the burst type, which can be sequential or interleaved. Over the course of my time designing with external memory, I've never used interleaved; however, the option is there if you have a use for it:

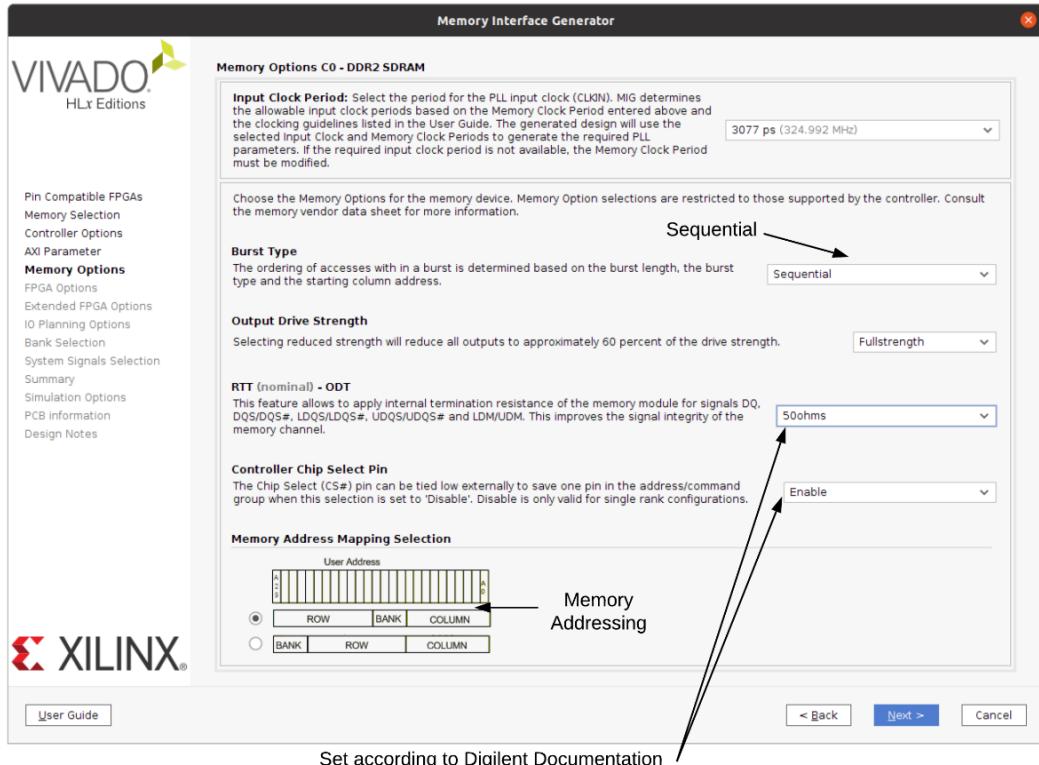


Figure 8.9 – Memory options

We also have a choice of how we address the memory. There are three components to any address: the bank, the row, and the column. Remember that the memory can only open up one row per bank, but can have one active row for each of the eight banks. Ideally, we would analyze our usage pattern to maximize our performance. Often, this is done by analyzing a C model of your system or through simulation. For now, I'm going to choose [ROW, BANK, COLUMN] so that we access multiple banks when we implement our VGA controller in a later chapter.

Next, we'll address the FPGA options for the controller.

Defining the FPGA options

There are a number of options available for the FPGA, some of which we need to change for the board we are using:

1. We'll be generating the clock internally, so we will specify both the system clock and the reference clock as **No Buffer**. I've also enabled the debug interface so that we can take a closer look inside the FPGA while it's running.

We can leave the rest of the settings as default:



Figure 8.10 – MIG FPGA options

2. On the next screen, we can leave everything as default:



Figure 8.11 – Extended FPGA options

3. On the next screen, we'll define the pins. Digilent has provided a .ucf file for the DDR constraints. It's included in CH8/build/xdc/mig.ucf:

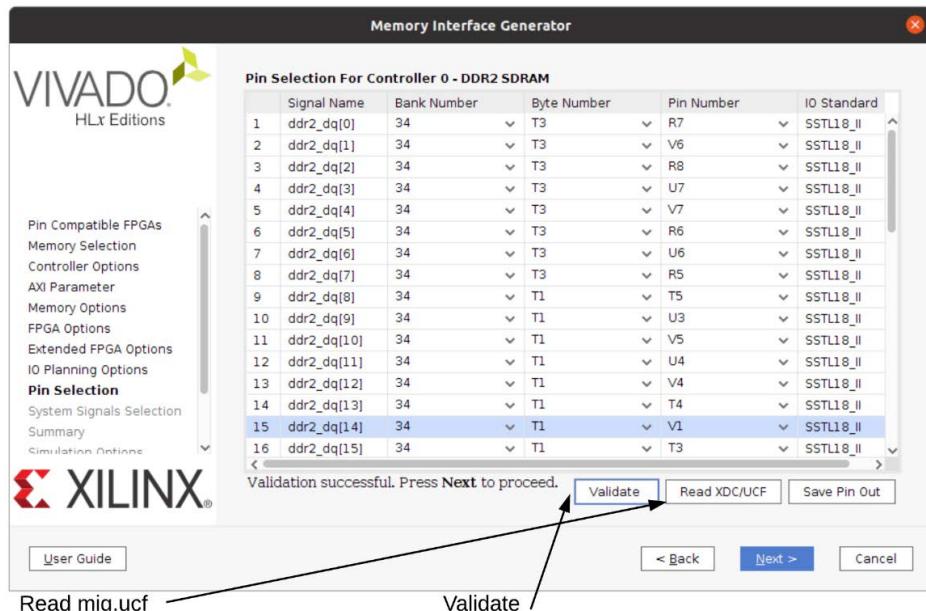


Figure 8.12 – Pin selection

- First, select **Read XDC/UCF** and read the `mig.ucf` file. Then, you must select **Validate** to verify the pinout and unlock the **Next** button. Now, we can move on to the system signals selection:



Figure 8.13 – System signals selection

- We have already selected internal signals for some signals. The others we will also use internally since they are not brought out on the board. Don't change anything on this screen and we'll look at the summary screen after hitting **Next**:



Figure 8.14 – MIG summary

6. On this screen, you can review the summary of the DDR2 controller that we are going to generate. Verify that it matches the parameters we've selected and then hit **Next:**



Accept for memory model

Figure 8.15 – Simulation options

- Xilinx provides a micron model for use with simulations of the DDR2 core. In order to generate it, you must accept the license. This is advisable since it will allow you to simulate the core against a real DDR2 model. Select **Accept** (recommended) or **Decline**, then select **Next**:

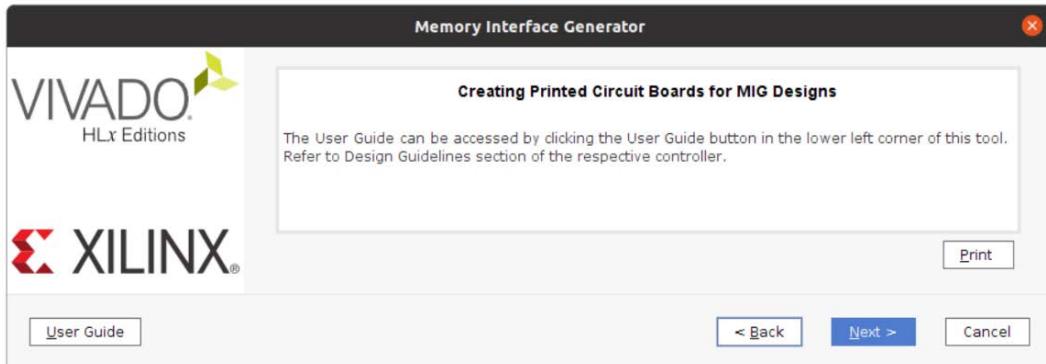


Figure 8.16 – Printed circuit board information

- The MIG reminds you about the user guide in the event that you are designing your own **Printed Circuit Board (PCB)**. Since we can assume that Digilent has already handled this, it can be safely ignored. If you decide to design your own board in the future, Xilinx provides many resources and checklists that you should follow. Feel free to inspect the user guide or just select **Next**:

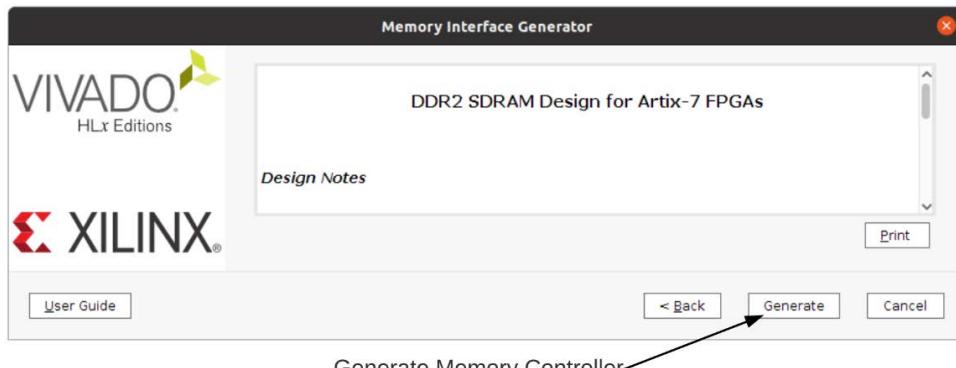


Figure 8.17 – Generating the core

We've reached the end of the options and you can simply select **Generate**. One more window will pop up to generate OOC modules. Select **Generate** again.

- Now, the core generation is complete. You'll see the core inside your design sources:



Figure 8.18 – Creating the example design

One very nice thing about a lot of the Xilinx cores – the MIG cores in particular – is that you can generate an example design for simulation and sometimes implementation. As previously mentioned, when optimizing addressing, you will likely want to simulate, and the example design can give you a jump start on this.

- To generate the example design, right-click on the .xci file in the design sources and select **Open IP Example Design**:

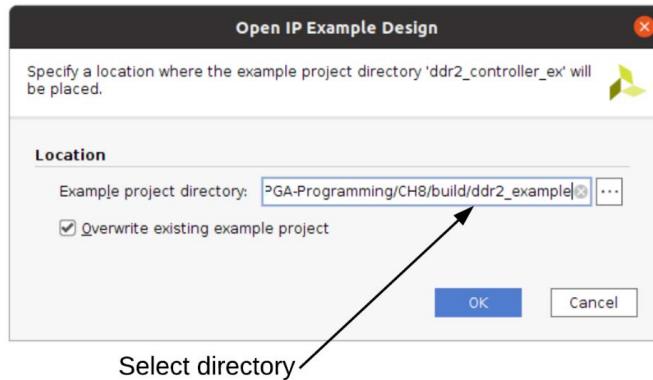


Figure 8.19 – Open IP Example Design

11. Make sure you've selected a good subdirectory for your example design and select OK:



Figure 8.20 – Viewing the example design

You can now see that we have the example design loaded, along with a simple testbench. We'll want to modify the top level so that we can run it on the board and take a look at it using the **Integrated Logic Analyzer (ILA)**. Before we change anything, let's verify that the generated testbench runs without problems. This is a good engineering practice. There is nothing worse than modifying and running, then realizing that the original generated design didn't work properly in the first place.

To do this, select **Run Simulation** as we have done previously.

Notice that this simulation takes a lot longer than anything we've done so far. You can see from the timescale the testbench is operating at that we need very precise timing for the DDR simulation:

```
`timescale 1ps/100fs
```

We can also take a look at the simulation output to see just how long a simulation is. The initialization and calibration phase takes 100 microseconds. The actual test takes about half that time:



Figure 8.21 – MIG simulation

If your simulation completed successfully, you should see `finish` highlighted in your simulation window:

```

548 //*****
549 // Reporting the test case status
550 // Status reporting logic exists both in simulation test bench (sim_tb_top)
551 // and sim.do file for ModelSim. Any update in simulation run time or time out
552 // in this file need to be updated in sim.do file as well.
553 ****
554 initial
555 begin : Logging
556   fork
557     begin : calibration_done
558       wait (init_calib_complete);
559       $display("Calibration Done");
560       #5000000.0;
561       if (!tg_compare_error) begin
562         $display("TEST PASSED");
563       end
564       else begin
565         $display("TEST FAILED: DATA ERROR");
566       end
567       disable calib_not_done;
568       $finish;
569   end

```

Figure 8.22 – Simulation completed successfully

Now, we have proven out the Xilinx testbench using the Xilinx traffic generator module. Let's go ahead and modify our design so that we can run it on the board and look at it with the ILA.

Modifying the design for use on the board

I've copied the example design over so that we can implement the changes without affecting the original design. Looking at the Xilinx implemented design, `sys_clk_i` is being generated at 325 MHz. The first thing we'll need to do is modify our top level for the board, which means we'll need to add a **Mixed Mode Clock Manager (MMCM)** to take our 100 MHz system clock and generate the 325 MHz clock. We'll also need to generate `clk_ref_i`. If we look at the testbench, we'll see that this clock should be 200 MHz:



Figure 8.23 – sys_pll settings

Note that in the `sys_pll` clock output clock definitions, one interesting thing about the output clocks is that order does matter. If you try to swap the clocks, that is, instead of `clk_out1` being 200 MHz and `clk_out2` being 325 MHz they are 325/200 MHz, you'll find that the clocks are no longer precise. Sometimes, you need to try different combinations of output clocks to get close to what you are targeting.

Next, we'll address our top level. I've modified it to contain the ports we'll need and remove the ones we don't:

```
inout [15:0]      ddr2_dq,
inout [1:0]       ddr2_dqs_n,
inout [1:0]       ddr2_dqs_p,
```

```

    output [12:0]      ddr2_addr,
    output [2:0]      ddr2_ba,
    output           ddr2_ras_n,
    output           ddr2_cas_n,
    output           ddr2_we_n,
    output [0:0]      ddr2_ck_p,
    output [0:0]      ddr2_ck_n,
    output           ddr2_cke,
    output [0:0]      ddr2_cs_n,
    output [1:0]      ddr2_dm,
    input            ext_clk,
    output          tg_compare_error,
    output          init_calib_complete,
    input            sys_rst,
    output          LED
);
wire clk_ref_i;
wire sys_clk_i;
assign LED = sys_rst;
sys_pll u_sys_pll
(.clk_out1      (clk_ref_i),
 .clk_out2      (sys_clk_i),
 
.clk_in1       (ext_clk),
.resetn        (sys_rst));

```

I've removed the two clocks, `clk_ref_i` and `sys_clk_i`, and made them internal. I've added an external clock, `ext_clk`, so that we can bring in 100 MHz from the crystal oscillator and use the PLL to generate 200 and 325 MHz clocks that we need for the DDR2. I've also instantiated the PLL in the top-level design.

We'll need to also modify the testbench for the 100 MHz clock:

```

parameter CLKIN_PERIOD      = 10000; //3077;
// Input Clock Period

```

I've commented out the old clock period for 325 MHz and replaced it with a 100 MHz clock. Now, our PLL will generate the correct clocks and we can verify this in simulation.

Finally, we can build our design for the Nexys A7. Unfortunately, we won't be able to run this on the Basys 3 board.

Once you've downloaded the image to the board, you can bring up the ILA. The pattern generator and checker are running constantly, so you can trigger immediately to see activity:



Figure 8.24 – DDR2 ILA

You can trigger immediately and see activity on the DDR2 internal interfaces. Looking at the LED on the board should show no errors detected and the ILA will show the activity generated.

The DDR2 core as we generated it has one other trick up its sleeve. It also has the Vivado debug logic core **Virtual I/O (VIO)** interface. This interface can be used in your designs to provide input and output functionality from within Vivado to aid on-chip debugging. In the case of the DDR2 interface, it provides insight into what is going on in the core and also allows changing configuration on the fly.

You can bring up the VIO by selecting the **hw_vios** tab:



Figure 8.25 – Adding Hardware (HW) VIOs

You can press the + button to add probes. For now, simply add them all:

Name	Value	Activity	Direction	VIO
vio_modify_enable	[B] 0		Output	hw_vio_1
vio_tg_rst	[B] 0		Output	hw_vio_1
> vio_tg_simple_data_sel[1:0]	[H] 0		Output	hw_vio_1
vio_wdt_en_w	[B] 0		Output	hw_vio_1
vio_win_active	[B] 0		Input	hw_vio_1
> vio_win_byte_select[6:0]	[H] 00		Input	hw_vio_1
> vio_win_current_bit[6:0]	[H] 00		Input	hw_vio_1
> vio_win_current_byte[3:0]	[H] 0		Input	hw_vio_1
vio_win_start_1	[B] 0		Input	hw_vio_1
> vio_addr_mode_value[2:0]	[H] 2		Output	hw_vio_1
vio_dbg_pi_f_inc	[B] 0		Output	hw_vio_1
vio_dbg_po_f_dec	[B] 0		Output	hw_vio_1
vio_dbg_po_f_stg23_sel	[B] 0		Output	hw_vio_1
> vio_fixed_instr_value[2:0]	[H] 0		Output	hw_vio_1
> vio_po_win_right_ram_out[8:0]	[H] 000		Input	hw_vio_1
vio_mem_pattern_init_done	[B] 0		Input	hw_vio_1
> vio_dbg_pi_counter_read_val[5:0]	[H] 24		Input	hw_vio_1
> vio_dbg_po_counter_read_val[8:0]	[H] 097		Input	hw_vio_1
vio_dbg_tg_compare_error	[B] 0		Input	hw_vio_1
> vio_dbg_tg_wr_data_counts[47:0]	[H] 0000_0000_00	*	Input	hw_vio_1
> vio_dbg_wi_chk[164:0]	[H] 00_0000_0000	*	Input	hw_vio_1
vio_pause_traffic	[B] 0		Output	hw_vio_1
> vio_sel_mux_rdd[3:0]	[H] 0		Output	hw_vio_1
vio_win_byte_select_dec	[B] 0		Output	hw_vio_1
vio_win_byte_select_inc	[B] 0		Output	hw_vio_1
vio_sel_pi_pon	[B] 0		Output	hw_vio_1
vio_start	[B] 0		Output	hw_vio_1
> vio_win_left_ram_out[5:0]	[H] 00		Input	hw_vio_1
vio_clear_error	[B] 0		Output	hw_vio_1
> vio_dbg_bit[8:0]	[H] 000		Output	hw_vio_1
> vio_dbg_dqs[4:0]	[H] 00		Output	hw_vio_1
vio_dbg_po_f_inc	[B] 0		Output	hw_vio_1
> vio_fixed_bt_value[9:0]	[H] 000		Output	hw_vio_1
> vio_vio_bt_mode_value[1:0]	[H] 2		Output	hw_vio_1
vio_data_mask_gen	[B] 0		Output	hw_vio_1
vio_dbg_sel_po_indec	[B] 0		Output	hw_vio_1
> vio_instr_mode_value[3:0]	[H] 0		Output	hw_vio_1
> vio_data_mode_value[3:0]	[H] 7		Output	hw_vio_1
vio_dbg_sel_pi_indec	[B] 0		Output	hw_vio_1
vio_dbg_plf_dec	[B] 0		Output	hw_vio_1
> vio_win_right_ram_out[5:0]	[H] 00		Input	hw_vio_1
> vio_po_win_left_ram_out[8:0]	[H] 000		Input	hw_vio_1
> vio_dbg_tg_rd_data_counts[47:0]	[H] 0000_0000_00	*	Input	hw_vio_1

Figure 8.26 – VIO signals

From *Figure 8.26*, you can see signals labeled as **Output** and **Input**. The direction is relevant to the VIO core. You can monitor input signals and make changes to output signals that will be reflected in the pattern generator and checker. In the preceding figure, I have marked the order to apply a change to the pattern generator. To change the configuration, you would perform the following operations:

1. Set `vio_modify_enable` to 1.
2. Set `vio_addr_mode_value` to 1 = fixed address, 2 = **Pseudo Random Binary Sequence (PRBS)** address or sequential address.
3. Set `vio_bl_mode_value` to 1 = Fixed bl or 2 = PRBS bl.
4. Set `vio_data_mode_value` to 1 = fixed, 2 = DGEN_ADDR, 3 = DGEN_HAMMER, 4 = DGEN_NEIGHBOR, 5 = DGEN_WALKING1, and 6 = DGEN_WALKING0, DGEN_PRBS.

Now, we've looked at DDR2 and in the next chapter, we'll be using it for our display controller. Briefly, we'll look at other external memory types that are used with FPGAs.

Other external memory types

There are a variety of memory types that have been introduced over the years that are or have become more common with FPGAs. I want to briefly touch on them as you might be interested in them for your own projects in the future.

Quad Data Rate (QDR) SRAM

Quad Data Rate (QDR) SRAM is commonly used in networking applications. Like DDR memory, data is transferred on both edges of the clock for performance. Unlike DDR, QDR has both read and write channels, so you can issue read and write commands simultaneously. Also, unlike DDR DRAM, this is an SRAM, so there are no refresh cycles and the latency for a read or write can be as low as about 13 clock cycles at 300 MHz.

QDR has a much larger capability than FPGA internal memory, but much less than DDR. It's also relatively expensive, which is why it's mostly used in networking applications.

HyperRAM

HyperRAM is a type of self-refreshing DRAM designed for **Low Pin Count (LPC)** applications. It has performance and sizes similar to DDR (not 2+) memories making it ideal for some applications. There are PMOD boards available for HyperRAM.

SPI RAM

There is a very LPC RAM-utilizing **Serial Peripheral Interface (SPI)**. These RAMs have similar capabilities to DDR (not 2+) and fairly good performance using as few as eight pins. PMOD boards are available with these memories also.

Summary

In this chapter, we've looked at external memory, in particular, DDR2, as that is what we have readily available on the Nexys A7. We've looked at generating a core using the Xilinx MIG controller and how to generate the example design. We've then run the example design on the board and, using the ILA, seen it in operation. We've also taken a quick look at other external memory types.

Up until now, we've limited ourselves to LEDs and seven-segment displays for our output. In the next chapter, we are going to take the DDR controller and create a VGA controller. Dust off your CRT or LCD with a VGA connector and we'll work on displaying our temperature sensor data, our audio data, and calculator data using a real display.

Questions

1. Which of the following are true about internal versus external memory?
 - a) DDR memory storage capacity is much smaller than BRAM.
 - b) For the same memory data width, DDR has much higher performance than BRAM.
 - c) The latency to access data from BRAM and DDR is identical.
 - d) You should always use LUTRAM first before using any other memory type.
2. To generate DDR2 memory for our project, we used the Xilinx:
 - a) **Massive IP Goliath (MIG)**
 - b) **Minimally Informative Google (MIG)**
 - c) **Memory Interface Generator (MIG)**
3. We can use ILAs to examine data in the FPGA and VIOs to read and write data.
 - a) True
 - b) False
4. Artix 7 FPGAs can use which of the following memories from the MIG?
 - a) DDR2
 - b) DDR3
 - c) DDR4
 - d) LPDDR2
 - e) QDR

5. It is possible to use HyperRAM, SPI RAM, and SDRAM if you are willing to write your own controllers.
 - a) True
 - b) False

Challenge

We've created the DDR2 using the MIG and we have the example design. We don't have a way of inserting errors. Can you utilize a button or switch on our board and use it to inject an error into the data either to or from the memory.

Hint: You can use an XOR gate to inject the error. When the bit coming from the pushbutton or switch is 0, then the output of the XOR will be unchanged. If you set the bit, it will invert the data passing through.

Further reading

For more information about what was covered in this chapter, please refer to the following:

- <https://www.micron.com/products/dram/ddr2-sdram/part-catalog/mt47h64m16hr-25>
- <https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/reference-manual>

9

A Better Way to Display - VGA

Up until now, we've been limited to displaying information using LEDs, single color or RGB, as well as the 7-segment display. We are quite capable of performing operations and displaying limited information, as we have demonstrated with our temperature sensor and calculator. The Nexys A7 offers an additional output that can provide us with an almost unlimited method of displaying information, the **Video Graphics Array (VGA)** connector. The VGA connector on the Nexys A7 can display resolutions of up to 1600x1200, with up to 2^{12} or 4,096 colors. What allows us to unlock this capability is that we now know how to use our external memory, which will provide our framebuffer.

By the end of this chapter, we'll have created a method of displaying data on a **Cathode Ray Tube (CRT)** or LCD monitor via the VGA connector. In *Chapter 10, Bringing It All Together*, we'll use this methodology to upgrade some of our projects to utilize the new display.

In this chapter, we are going to cover the following main topics using a project, **Introducing the VGA**:

- Defining registers
- Generating timing for the VGA
- Displaying text

Technical requirements

The technical requirements for this chapter are the same as those for *Chapter 1, Introduction to FPGA Architectures and Xilinx Vivado*.

To follow along with the examples and the project, you can find the code files for this chapter at the following repository on GitHub: <https://github.com/PacktPublishing/Learn-FPGA-Programming/CH8>.

A VGA-capable monitor and cable are also required if you want to implement the project on the board.

Project 11 – Introducing the VGA

The earliest professional computer displays were simple monochrome text displays. The earliest personal computers, such as the Apple 2, could display 280x192 pixels with a small number of colors. The Commodore 64 and IBM/PC could display 320x200, again with limited color palettes. The original IBM VGA was introduced in 1987 and it allowed for higher resolutions and standardized the connector going forward until digital displays such as LCDs became the norm.

The first thing we'll need to look at is how the screen is drawn. Whether you are using a CRT display or a modern LCD, the timing is still supported to provide backward compatibility. Originally, the VGA output was designed to drive an electron gun to light up phosphors on a CRT. This meant timing spanned the entire display, plus time for the gun to shift from one side of the screen to the other, or from the bottom back to the top. *Figure 9.1* shows the various timing parameters and their relationship to what is displayed on the screen:

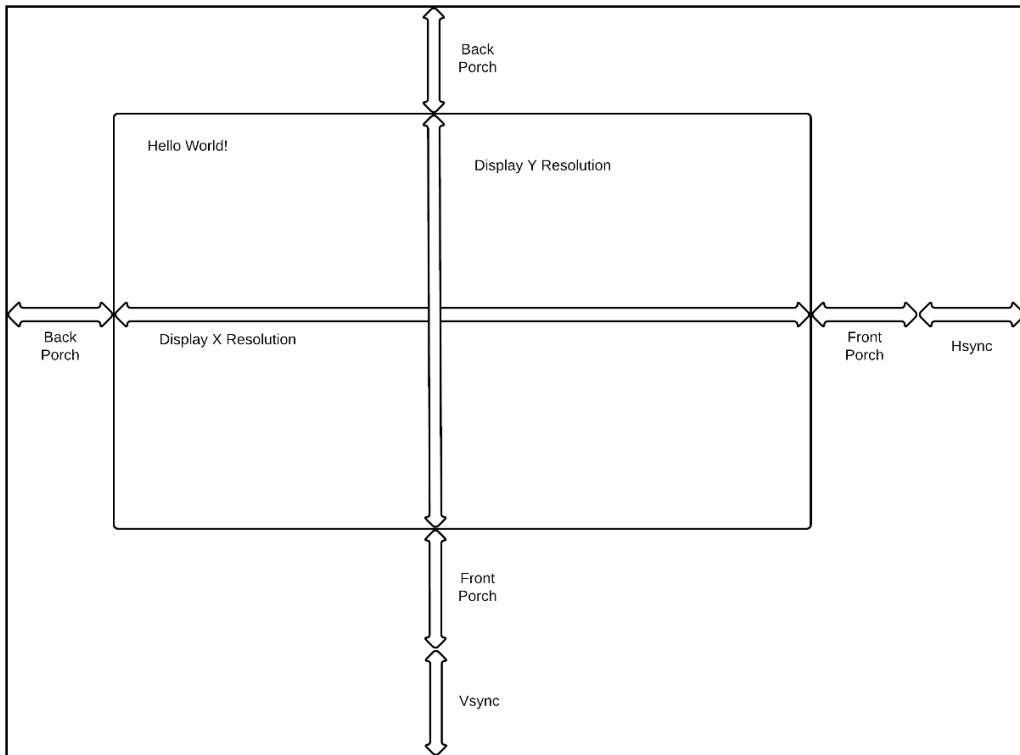


Figure 9.1 – VGA screen timing information as displayed

The main components of the output that we need to generate are as follows:

- **Hsync** – Horizontal synchronization signal
- **Vsync** – Vertical synchronization signal
- Red[3:0], Green[3:0], Blue[3:0] – RGB color values. In a real display these would be 8 bits per color (**24 bits per pixel (bpp)**). However, Digilent opted to create a resistor array implementation of a **Digital to Analog Converter (DAC)** rather than a true DAC, thereby limiting the bpp to 444 or 12 bpps.

We're going to want to make the VGA controller be as generic as possible. To do this, we'll create a register-based interface, so we'll need to figure out what the values we'll need for a given resolution are.

Important note

The following timing list is fairly comprehensive for a 4:3 aspect ratio display (older CRT/ TV). Depending on the display you use, some of these will not work. We'll default to 640x480 @ 60Hz since this is the base VGA display and should be supported by everything.

We can look at a list of **Video Electronics Standards Association (VESA)** standards to get an idea of what we want to display:

Resolution	Refr rate	Horiz total	Vert total	Clock	FP	tHS	Pol	BP	FP	tVS	Pol	BP
640x480	60	800	525	25.18	16	96	-	48	10	2	-	33
640x480	72	832	520	31.5	16	40	-	128	9	3	-	28
640x480	75	840	500	31.5	16	64	-	120	1	3	-	16
640x480	85	832	509	36	56	56	-	80	1	3	-	25
800x600	60	1056	628	40	40	128	+	88	1	4	+	23
800x600	72	1040	666	50	56	120	+	64	37	6	+	23
800x600	75	1056	625	49.5	16	80	+	160	1	3	+	21
800x600	85	1048	631	56.25	32	64	+	152	1	3	+	27
1024x768	60	1344	806	65	24	136	-	160	3	6	-	29
1024x768	70	1328	806	75	24	136	-	144	3	6	-	29
1024x768	75	1312	800	78.75	16	96	+	176	1	3	+	28
1024x768	85	1376	808	94.5	48	96	+	208	1	3	+	36
1280x1024	60	1688	1066	108	48	112	+	248	1	3	+	38
1280x1024	75	1688	1066	135	16	144	+	248	1	3	+	38
1280x1024	85	1728	1072	157.5	64	160	+	224	1	3	+	44
1600x1200	60	2160	1250	162	64	192	+	304	1	3	+	46
1920x1200	60	2616	1242	195	96	200	+	400	3	3	+	36

The preceding table contains the timing for the possible modes our display will support. The first thing to note is that the clock frequency that we'll need varies quite a bit from 25.18 MHz through 195 MHz. We'll address this by introducing clock reconfiguration, which is available in the clocking wizard. We can also make registers for storing the various parameters we'll need, so we'll use an AXI Lite interface for our registers.

Now let's take the relevant numbers from the preceding table and put them into a timing diagram so that we can visualize the actual signals going to the display:



Figure 9.2 – Video timing diagram

In *Figure 9.2*, we can see how the timing works. The timing is broken up into two sections. The first is the frame time, which can be looked at as the Vsync timing, which is based on the number of scanlines. Each scanline is similarly composed of the Hsync plus data.

To keep things simple, we'll assume that data is stored as 1-bit values. Typically, VGA and VESA modes would have 8-, 16-, or 32-bit colors. 8-bit values would be used as an index into a palette of 256 colors out of 16 million colors. 16-bit colors would typically be 565 or 555 (RGB) values, and 32-bit color was actually 888 RGB values capable of displaying 16 million colors. For our purposes, and since we are tackling quite a bit, I'll keep to storing 1-bit color. A pixel will be on or off.

Important note

When dealing with colors, we'll reference them by the number of bits used to represent them (8, 16, 24, or 32) and the number of bits per color channel (565, 555, or 888), where each digit represents the number of bits used for each color: red, green, and blue.

Defining registers

The first step we'll need for our VGA controller is to define a set of registers we can use to solve our problem. We know the timing parameters from *Figure 9.1* and the associated table. From this we can derive some parameters. For our VGA, I would propose the following set:

- Horizontal display start – The number of horizontal pixels before the display starts, equivalent to the horizontal back porch minus one.
- Horizontal display width – The width of the display.
- Horizontal sync width – Hsync width.
- Horizontal display total width – The total width of display and non-display portions.
- Vertical display start – The number of display lines before the display starts, equivalent to the vertical back porch minus one.
- Vertical display height – The height of the visible display.
- Vertical sync width – Vsync height in scan lines.
- Vertical display total height – The total height of the display and non-display portions.
- VGA format – The pixel depth for a given screen.
- Display address – The display address to read from. This can be written at any time, but will not take effect until the beginning of the next frame to prevent tearing.
- Horizontal and vertical polarity selections – Because different modes have active high or low polarities for the sync pulses, we need to provide a way of selecting between them.

- Display pitch – We need to know how many display pages to read for a given scanline as well as how many to count by for each subsequent scanline.
- Load mode – Typically, in complex designs, we may have multiple registers that make up a complete set of values necessary for a given function. We must provide a way to update them all simultaneously once updating is complete.

Our registers will be accessible via an AXI Lite interface.

Coding a simple AXI Lite interface

The write side of the AXI interface involves three components: the address bus, the data bus, and the response bus. We can see the address interface in the core interface definition:

```
input wire      reg_awvalid,  
output logic    reg_awready,  
input wire [11:0] reg_awaddr,  
  
input wire      reg_wvalid,  
output logic    reg_wready,  
input wire [31:0] reg_wdata,  
input wire [3:0]  reg_wstrb,  
  
input wire      reg_bready,  
output logic    reg_bvalid,  
output logic [1:0] reg_bresp,
```

The slave device must be able to handle the address and data buses independently. In our design, we'll write both at the same time, but it's possible that a master device may provide either an address or data before the other. We'll address this in our register state machine:

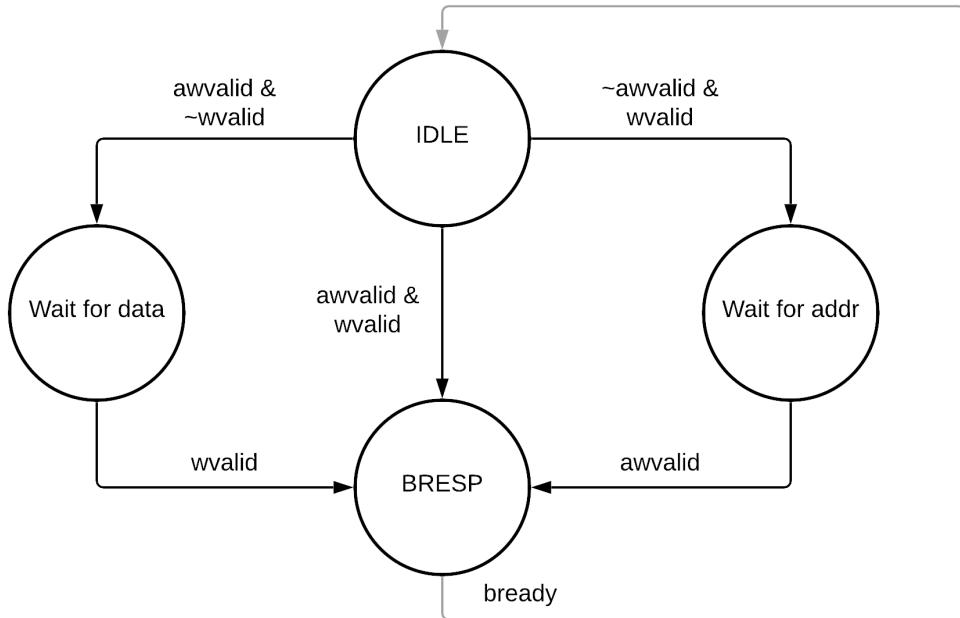


Figure 9.3 – AXI Lite state machine

Figure 9.3 shows the basic state machine. If **awvalid** and **wvalid** are both high, then we can generate the response. If we are missing one of the components to a full transfer, either **awvalid** or **wvalid**, we proceed to a wait state and then to the **BRESP** state.

Finally, in the **BRESP** state, as soon as we see **bready**, we generate a success response and transition back to idle.

Now let's examine the actual timing generation.

Generating timing for the VGA

We'll need two **Phase Locked Loops (PLLs)** or **Mixed Mode Clock Managers (MMCMs)** for our design. The first PLL will be a duplicate of the one we created in *Chapter 8, Lots of Data? MIG and DDR2*, to generate clocks for the DDR2 memory controller and also our internal clocks. We will generate the second one so that we can change the timing parameters. By default when the design powers up it will display a VGA resolution of 640x480 @ 60 Hz. The main difference in our configuration is to select **Dynamic Reconfig**:

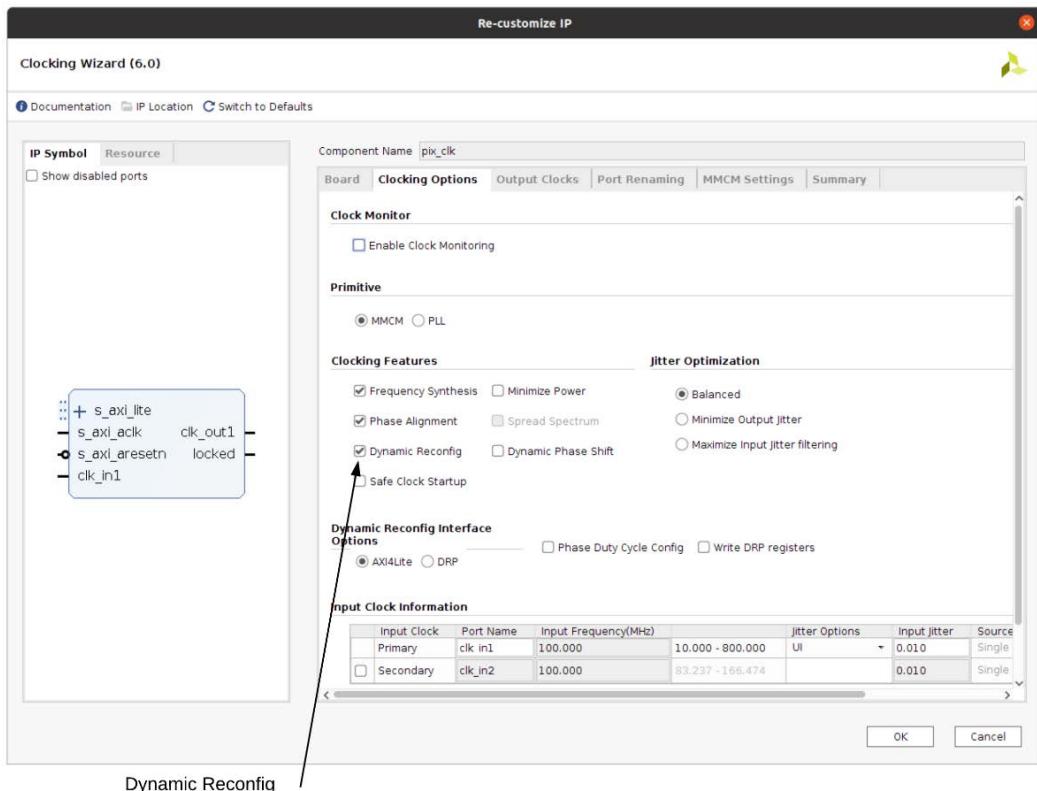


Figure 9.4 – Dynamic reconfiguration

Adding dynamic reconfiguration exposes an AXI Lite interface in the clocking wizard that we can use to reconfigure the PLL on the fly. The registers we need to focus on can be found in the clocking wizard drive 6.0 at https://www.xilinx.com/support/documentation/ip_documentation/clk_wiz/v6_0/pg065-clk-wiz.pdf. We'll only be reconfiguring `clk0`. In the following screenshot, I've extracted the information we need from the clocking wizard. In *Figure 9.5*, you can see how to extract these numbers yourself:



Figure 9.5 – Extracting reconfiguration parameters

The parameters we need (based on a 200 MHz input clock) are as follows:

- **Divide Counter:** In *Figure 9.5*, for 25.18 MHz, this is 9.
- Clock feedback multiplier (**Mult Counter**) integer: For 25.18 MHz, this is 50.
- Clock feedback multiplier (**Mult Counter**) fraction: For 25.18 MHz, this is 000.
- Clock feedback divider value integer: For 25.18 MHz, this is 44.
- Clock feedback divider value fraction: For 25.18 MHz, this is 125.

The values can be calculated, but you need to be careful to make sure you don't exceed the maximum PLL frequency. The following table shows the values we need for all our VGA frequencies:

Resolution	VGA clk	0x200			0x208	
		[7:0]	[15:8]	[25:16]	[7:0]	[17:8]
640x480 60 Hz	25.18	3	21	625	28	625
640x480 72/75 Hz	31.5	4	39	375	31	250
640x480 85 Hz	36	5	49	500	27	500
800x600 60 Hz	40	1	10	000	25	000
800x600 75 Hz	49.5	5	49	500	20	000
800x600 72 Hz	50	1	10	000	20	000
800x600 85 Hz	56.25	1	10	125	18	000
1024x768 60 Hz	65	5	50	375	15	500
1024x768 70 Hz	75	4	40	125	13	375
1024x768 75 Hz	78.75	4	39	375	12	500
1024x768 85 Hz	94.5	5	47	500	10	000
1280x1024 60 Hz	108	1	10	125	9	375
1280x1024 75 Hz	135	1	10	125	7	500
1280x1024 85 Hz	157.5	4	39	375	6	250
1600x1200 60 Hz	162	1	10	125	6	250
1920x1200 60 Hz	195	1	9	750	5	000

We can use the preceding table to create code to load our pixel PLL. We also need to load the register values for the resolution we need. Let's set that table up first.

First, we'll create a structure to hold the table we'll use to set up the PLL and VGA controller for each of the 17 resolutions that we'll support. We'll create a simple AXI Lite state machine that can configure the desired resolution, but in the future, we could use a microcontroller in the system:

```
typedef struct packed {
    logic [7:0]    divide_count;
    logic [15:8]   mult_integer;
    logic [25:16]  mult_fraction;
    logic [7:0]    divide_integer;
    logic [17:0]   divide_fraction;
```

```
    logic [11:0] horiz_display_start;
    logic [11:0] horiz_display_width;
    logic [11:0] horiz_sync_width;
    logic [11:0] horiz_total_width;
    logic [11:0] vert_display_start;
    logic [11:0] vert_display_width;
    logic [11:0] vert_sync_width;
    logic [11:0] vert_total_width;
    logic          hpol;
    logic          vpol;
    logic [12:0]   pitch;
} resolution_t;
```

The structure encapsulates all the necessary parameters. We can define a variable and initialize it in an initial block to use as constants:

```
resolution_t resolution[17];
initial begin
    // 640x480 @ 60Hz
    resolution[0].divide_count      = 8'd3;
    resolution[0].mult_integer      = 8'd21;
    resolution[0].mult_fraction     = 10'd625;
    resolution[0].divide_integer    = 8'd28;
    resolution[0].divide_fraction   = 10'd625;
    resolution[0].horiz_display_start = 12'd15;
    resolution[0].horiz_display_width = 12'd640;
    resolution[0].horiz_sync_width   = 12'd96;
    resolution[0].horiz_total_width  = 12'd799;
    resolution[0].vert_display_start = 12'd9;
    resolution[0].vert_display_width = 12'd480;
    resolution[0].vert_sync_width    = 12'd2;
    resolution[0].vert_total_width   = 12'd524;
    resolution[0].hpol              = '0;
    resolution[0].vpol              = '0;
    resolution[0].pitch             = 13'd5;
```

We will define all 17 modes in our code. Only the first/default mode is shown here. With this we can now create our state machine to load the VGA and PLL.

The state machine is divided into two sections: `CFG_WR0` - 2 loads the **MMCM** with our clock configuration settings, while `CFG_WR3` - 5 loads the resolution for the VGA controller. The state machine operates as follows:

1. It detects a button press and starts loading MMCM parameters.
2. `CFG_WR0` checks which valids are active. In the event that only `wvalid` or `awvalid` is active, we have two substates, `CFG_WR1` and `CFG_WR2`, to await the missing valid.
3. Once both valids are valid, we advance the register write. We must write all 24 registers in the MMCM before we move on to the VGA.
4. The VGA portion of the state machine operates similarly to the MMCM portion and we advance through the VGA parameters. Once complete, we go back to idle.

The VGA core handles the monitor timing and the display output.

Important note

Depending on your monitor type, you may not be able to display all resolutions. Some monitors are also more forgiving than others of timing problems. My particular monitor could go to 1280x1024 @ 85 Hz but no higher. Due to timing constraints, I would recommend not going higher than 1280x1024 @ 75 Hz.

Let's now take a deeper dive into the timing generator.

Monitor timing generator

To handle sync generation, we'll need two counters. The first counter, `horiz_count`, will generate the timing and pixel output for each scanline. The second counter, `vert_count`, counts the number of scanlines to determine when to start displaying pixels and generate the Vsync:

```
if (horiz_count >= horiz_total_width) begin
    horiz_count <= '0;
    if (vert_count >= vert_total_width) vert_count <= '0;
    else vert_count <= vert_count + 1'b1;
    scanline <= vert_count - vert_display_start + 2;
    mc_addr  <= scanline * pitch;
    mc_words <= pitch;
```

```
end else  
    horiz_count <= horiz_count + 1'b1;
```

The preceding code zeroes out the `horiz_count` signal when we reach the end of a scanline. You'll notice that the comparison is a greater than or equal to `horiz_total_width` signal. The way we update counters doesn't stop or restart the timing generation. This will ensure that if we were to accidentally put something out of range, the counts will recover. Similarly, we do the same with the vertical count.

This block also generates a few other parameters we need for displaying pixels. The first is the scanline information. This calculates the scanline currently being operated on. Scanline zero would be the first displayable scanline.

We also register the address for the current scanline and the pitch, which is also the number of 16-byte words to be read for each scanline. Note that this number can be greater than or equal to the number of bytes we need.

It helps when you are using slower parts or trying to achieve a higher clock speed to look for opportunities to precalculate mathematical operations when you can. In the preceding code, I'm calculating the address we need:

```
mc_addr <= scanline * pitch;
```

This is because you'll see in the code where we read from memory that we need to make sure we don't violate AXI rules:

```
vga_hblank <= ~((horiz_count > horiz_display_start) &  
                    (horiz_count <=  
                     (horiz_display_start + horiz_display_width))) ;  
vga_hsync      <= polarity[1] ^  
    ~(horiz_count > (horiz_total_width - horiz_sync_width)) ;  
vga_vblank    <= ~((vert_count > vert_display_start) &  
                    (vert_count <=  
                     (vert_display_start + vert_display_width))) ;  
vga_vsync      <= polarity[0] ^  
    ~(vert_count > (vert_total_width - vert_sync_width)) ;
```

You'll see that we are generating the Hsync and Vsync as shown in *Figure 9.1* at the end of the scanline and the display window. We calculate the time to generate this by creating the sync from the horizontal or vertical total minus the sync width. We also need to use our polarity registers to generate the correct sync polarity. Exclusive-OR gates can be used as programmable inverters.

We also generate the blank signals. These aren't technically necessary unless you are using a real DAC, when those signals are used to zero out the pixel output, although you could use them similarly. I've included them since, in simulation, it can assist in locating when data is expected to be output.

In this section of code, we also generate a toggle mc_req signal for requesting data to be displayed:

```
if (vga_hblank && ~last_hblank && ~vga_vblank)
    mc_req <= ~mc_req;
last_hblank     <= vga_hblank;
```

We are taking advantage of the *dead time* of the display to prefetch the next scanline of data. When hblank is going away, in other words, the rising edge of hblank, we'll generate a request as long as we are not in the vertical blanking period.

Now that we have an operational display, we need something interesting to display on it.

Displaying text

A text character in its oldest and simplest form is a bitmap. Modern operating systems may use things such as TrueType, which can scale cleanly and easily at different resolutions. However, the oldest form of displaying text was to store a pattern in memory and then copy it to the screen.

I've included a file called `text_rom.sv`. It is essentially a lookup table:

```
module text_rom
    (input          clock, // Clock
     input [7:0]    index, // Character Index
     input [2:0]    sub_index, // Y position in character
     output logic [7:0] bitmap_out);
```

Functionally, we can view the text ROM in the following diagram:

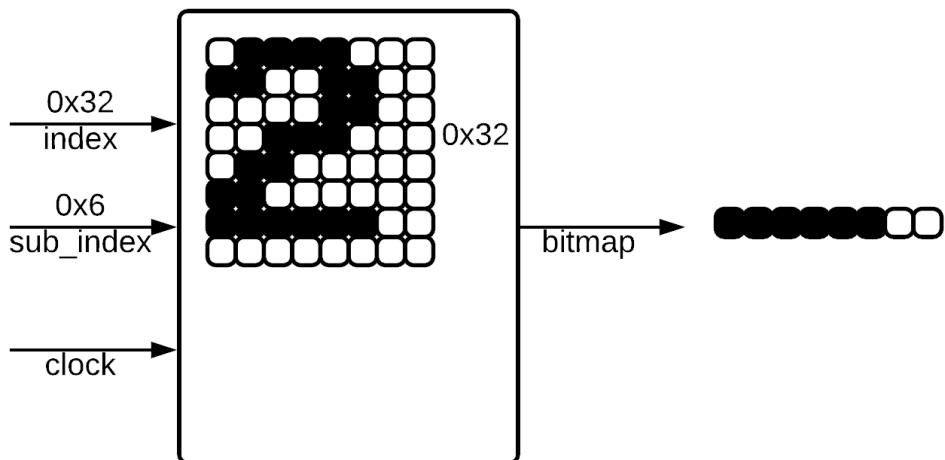


Figure 9.6 – text_rom

Every clock cycle, a character is looked up using the index and the subindex references the scanline of the character. In *Figure 9.6*, you can see an example where we are requesting character 0x32, which is **American Standard Code for Information Interchange (ASCII)** for the number 2. We are asking for the sixth scanline of the character. This returns the value 0xFC on the next cycle, which represents the pixels from the sixth scanline of the number 2.

Important note

ASCII code is one of the major standards for encoding text. One nice thing about ASCII is that the numbers 0-9 are encoded from 0x30-0x39.

`text_rom.sv` contains all the ASCII uppercase and lowercase characters and numbers as well as a few fill characters. ASCII is normally represented by an 8-bit value, so there is plenty of room to add new characters to display:

```
always @(posedge clock)
  case ({index, sub_index})
    ...
    // 2
    {8'h32, 3'h0}: bitmap <= 8'h78;
    {8'h32, 3'h1}: bitmap <= 8'hCC;
    {8'h32, 3'h2}: bitmap <= 8'h0C;
    {8'h32, 3'h3}: bitmap <= 8'h38;
```

```

{8'h32, 3'h4}: bitmap <= 8'h60;
{8'h32, 3'h5}: bitmap <= 8'hC0;
{8'h32, 3'h6}: bitmap <= 8'hFC;
{8'h32, 3'h7}: bitmap <= 8'h00;

```

Here we can see what the lookup for the number 2, 0x32, looks like. This is the same as what is represented in *Figure 9.6*.

One thing about the way data is stored is that in the application I developed, we'll need to flip the data coming out. I've added the following code:

```

always @* begin
    for (int i = 0; i < 8; i++) begin
        bitmap_out[i] = bitmap[7-i];
    end
end

```

This code flips the bits. Without it the text will appear reversed. You may or may not need this for your own applications, so it's good to know it exists in the event you need it or need to remove it.

Back to our top-level VGA. We'll add a string of text with each resolution setting so that when the display is set, we can print out what we have set it to:

```
res_text[0] = " zH06 @ 084x046";
```

Notice that the text is written backward as a string. This is because we are starting from bit 0 of character 0 and building it up to character 15, bit 7.

Requesting memory

For our display, we need to take our memory request signal and synchronize it to the memory controller clock. We'll also use this opportunity to reset the pixel FIFO. Note that we are toggling the request signal at the end of the line, so this provides a couple of key features for our design:

- We won't be displaying anything, so we can simply reset the FIFO.
- We should have quite a bit of time to reset the FIFO and start getting data back for displaying.

We can construct a state machine to handle our memory accesses as shown in the following code block:

```
case (mem_cs)
    MEM_IDLE: begin
        mem_arvalid <= '0;
        if (^mc_req_sync[2:1]) begin
            fifo_rst <= '1;
            mem_cs     <= MEM_W4RSTH;
        end
    end
    MEM_W4RSTH: begin
        next_addr <= mc_addr + mc_words;
        len_diff  <= 2047 - mc_addr[10:0];
        if (wr_rst_busy) begin
            fifo_rst <= '0;
            mem_cs     <= MEM_W4RSTL;
        end
    end
end
```

When we synchronize and detect an edge on the request, we reset the FIFO. The FIFOs provide an output to indicate when they are busy during a reset, so in the second state, we wait for the reset to go high, then release the reset, and enter the state to wait for it to go low again. We can take advantage of our wait time to calculate the next address and see how many scanlines there are before we reach the 2K (2,048) byte boundary.

Important note

When making a burst request over AXI, you cannot cross the 2,048-byte boundary. We must take this into account and break up bursts that might possibly violate this rule.

We'll use these parameters to test for a boundary crossing in the following code block:

```
MEM_W4RSTL: begin
    if (~wr_rst_busy) begin
        // Make a request from the current address
        mem_araddr  <= mc_addr;
        if (next_addr[31:11] != mc_addr[31:11]) begin
```

```

// look if we are going to cross 2K boundary
mem_arlen <= len_diff;
if (mem_arready) mem_cs <= MEM_REQ;
else mem_cs <= MEM_W4RDY1;
end else begin
    // Make a single request
    mem_arlen <= mc_words - 1;
    if (mem_arready) mem_cs <= MEM_IDLE;
    else mem_cs <= MEM_W4RDY0;
end // else: !if(next_addr[12])
// Calculate the parameters for second request
next_addr <= mc_addr + len_diff + 1'b1;
len_diff <= mc_words - len_diff;
end
end // case: MEM_W4RSTH

```

When the reset goes away, we can make a request to the memory controller. We have already calculated the next address, so we can test the upper bits to see whether the next address falls into the next 2,048-byte page. Based on the test, we'll either make a single request or a request for the last part of the current 2,048-byte page. In either case, we can move directly to the second request or back to IDLE if the awready signal is high, otherwise we need to move to a state to wait for awready.

We'll also pre-calculate the address and length of the second request in the event we need it:

```

MEM_REQ: begin
    if (~wr_rst_busy) begin
        mem_araddr <= next_addr;
        mem_arlen <= len_diff;
        if (mem_arready) mem_cs <= MEM_IDLE;
        else mem_cs <= MEM_W4RDY0;
    end
end // case: MEM_W4RSTH

```

The final state handles the remainder of the scanline if it crossed the 2,048-byte boundary.

To handle the data coming back, we'll use a Xilinx async, `xpm_fifo`, as shown in the following code block:

```
// Pixel FIFO
// large enough for one scanline at 1920x32bpp (480 bytes)
xpm_fifo_async
#(.FIFO_WRITE_DEPTH      (512),
 .WRITE_DATA_WIDTH      (128),
 .READ_MODE             ("fwft"))
u_xpm_fifo_async
(.rst                  (fifo_rst),
 .wr_clk               (mem_clk),
 .wr_en                (mem_rvalid),
 .din                  (mem_rdata),
 .wr_RST_BUSY          (wr_RST_BUSY),
 .rd_clk               (vga_clk),
 .rd_en                (vga_pop),
 .dout                 (vga_data),
 .empty                (vga_empty),
 .rd_RST_BUSY          (rd_RST_BUSY));
```

The main things to observe regarding the FIFO is that we are writing on the memory clock and reading on the VGA pixel clock. In this design, I haven't taken any precautions to make sure the data is loaded for a scanline or to handle exceptions. This results in the memory reads being *fire and forget*. We have a state machine that makes the request and the data is pushed back into a FIFO to be read out.

The FIFO is configured as **first-word fall-through (fwft)**, which means the data is ready on the output for immediate use.

Finally, we need to read from the FIFO and display on the screen:

```
initial scan_cs = SCAN_IDLE;
always @(posedge vga_clk) begin
    vga_pop <= '0;
    case (scan_cs)
        SCAN_IDLE: begin
            if (horiz_count == horiz_display_start) begin
                if (vga_data[0]) vga_rgb <= ~vga_empty;
```

```
    else vga_rgb <= '0;
    scan_cs    <= SCAN_OUT;
    pix_count <= '0;
end
SCAN_OUT: begin
    pix_count <= pix_count + 1'b1;
    // Right now just do single bit per pixel
    if (pix_count == 126) begin
        vga_pop <= ~vga_empty;
    end
    if (vga_data[pix_count]) vga_rgb <= '1;
    else vga_rgb <= '0;
    if (rd_rst_busy) scan_cs <= SCAN_IDLE;
end
endcase // case (scan_cs)
end
```

The display state machine is pretty simple. We wait until we reach the first scanline and then, based on the pixel format, we can display on the screen. This version of the code only supports 1 bpp.

At this point, we can run on the board and we should see VGA output. We've initialized the core to run at 640x480 @ 60 Hz:



Figure 9.7 – First VGA screen attempt

I've fixed this problem in the code you are running. Without a startup clearing of the memory, we are at the mercy of old data or data from the memory controller initialization being displayed:

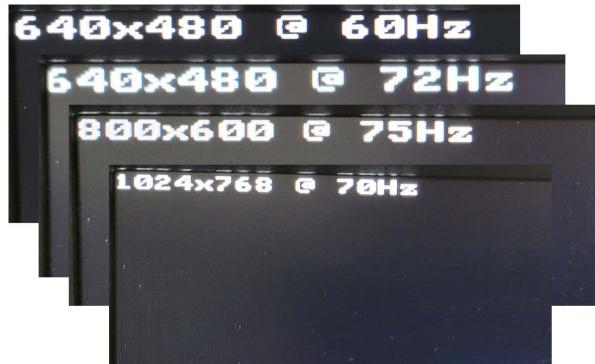


Figure 9.8 – Sample resolutions

When you first bring up the project, it will clear the screen and display 640x480 @ 60 Hz. This is accomplished by one state that is only executed when powering on:

```
CFG_IDLE0: begin
    update_text    <= ~update_text;
    cfg_state     <= CFG_IDLE1;
end
```

We've now completed a simple, yet useful, VGA controller. I hope when looking at this that you can see that writing hardware that is useful isn't out of reach. Certainly, there is a lot of work that goes on behind the scenes in order to make sure it works for what you want.

Testing the VGA controller

Most of the testing was done on the board. The turnaround time for a compile is short. The simulation time for a full frame is very long using the Vivado Simulator due to the PLLs and the memory controller. It is, however, a good way to check the first few scanlines of a display to make sure they look okay and that the timing works alright.

The two main pieces that we need for simulating are the clock generator and the register load:

```
initial clk = '0;
always begin
```

```

clk = #5 ~clk;
end
...
initial begin
    SW      <= 8;
    button_c <= '1;
    repeat (1000) @(posedge clk);
    while (~u_vga.init_calib_complete) @(posedge clk);
    $display("DDR calibration complete");
    while (~u_vga.locked) @(posedge clk);
    button_c <= '1;
    repeat (100) @(posedge clk);
    button_c <= '0;
    repeat (10000) @(posedge clk);
end

```

A more complete testbench could contain tasks for saving video frames, but given the speed, you are better off running on the board. To run on the board, we need to examine the constraints necessary.

Examining the constraints

In the VGA we have quite a bit of clock domain crossing to handle. The FIFO handles the data, but we have data going from our AXI interface to our memory controller clock and then to the VGA display clock. On top of this, we have a variable frequency VGA clock from the programmable MMCM.

When you implement an MMCM or PLL, Vivado will automatically create a generated clock on the output. Since we will reprogram the PLL during operation, we'll need to override this with the maximum clock we expect to see during operation:

```

create_clock -period 7.41 -name vga_clk -add [get_pins u_clk/
clk_out1]

```

We'll also need the clock periods for setting up the following constraints. We can get the `period` parameter from a clock by using `get_property`. `get_clocks` can be used to access the clock information:

```

set vga_clk_period [get_property PERIOD [get_clocks vga_clk]]
set clk200_period  [get_property PERIOD [get_clocks clk_out1_]

```

```
sys_clk]
set clkui_period [get_property PERIOD [get_clocks clk_pll_i]]
```

Experimenting a bit, I was able to discover that we could reliably run up to about 135 MHz, so I provided this as a clock on the PLL output.

Now we need to add constraints for our synchronizer inputs. Since these are single signal toggle synchronizers, we'll false-path the input to the first stage of the synchronizer flip flops:

```
set_false_path -from u_vga_core/load_mode_reg*/C -to */load_
mode_sync_reg[0]/D
set_false_path -from u_vga_core/mc_req_reg*/C -to */mc_req_
sync_reg[0]/D
set_false_path -from update_text_reg/C -to update_text_sync_
reg[0]/D
```

We'll also add in `max_delay` constraints to make sure to properly constrain the registers between clock domains and to not push the tool to meet unreasonable timing requirements. We do this as follows:

```
set_max_delay -datapath_only -from */horiz_display_start_reg*
[expr 1.5 * $vga_clk_period]
set_max_delay -datapath_only -from */horiz_display_width_reg*
[expr 1.5 * $vga_clk_period]
...
set_max_delay -datapath_only -from *sw_capt_reg*/C [expr 1.5 *
$clkui_period]
```

`set_max_delay` allows us to set the amount of time from any point to any other point. `-datapath_only` tells the timing engine to not consider clock delays in computing the delays.

With this we have implemented our design on the board and met timing. In the next chapter, we'll add in a keyboard and use the VGA as a capstone project where we can use it to display data from our previous projects.

Summary

In this chapter, we've introduced a better way of displaying data. Previously, we were limited to the physical outputs: a row of 16 LEDs, two tricolor LEDs, and the 7-segment display. We made good use of them for the simple testing of logic functions, our traffic light controller, and our simple calculator. We've used a ROM to display text. We've introduced a programmable PLL and used our DDR2 controller. We're now ready to tackle our capstone project.

In the next chapter, we'll wrap up the book by putting everything together. We can use our VGA to display the output from our temperature sensor, calculator, and microphone. We'll also introduce the PS/2 keyboard interface to provide an easier way to control the system.

Questions

1. You can use an XOR gate as:
 - a) A way to add two bits
 - b) A way to multiply two bits
 - c) A programmable inverter
2. We are limited to what resolution when generating a VGA controller?
 - a) 640x480 @ 60 Hz
 - b) 1280x1024 @ 85 Hz
 - c) 1920x1200 @ 60 Hz
 - d) A resolution our monitor can handle and a pixel clock that we can reliably meet timing for in our design
3. Building a VGA controller in an FPGA is impractical.
 - a) True
 - b) False
4. How many colors can we represent with 888 or 24 bpp?
 - a) 2 colors
 - b) 16 colors
 - c) 64K colors
 - d) True color, or 16 million colors

5. An AXI Lite write interface consists of which of the following?
 - a) A write address
 - b) Write data
 - c) A write response
 - d) All of the above
6. An AXI Lite read interface consists of which of the following?
 - a) A read address
 - b) Read data
 - c) A read response
 - d) All of the above
 - e) (a) and (b)

Challenge

The current VGA design only displays black and white. Can you change the design to display two different colors? Can you modify it to use some switches on the board to select these colors?

Further reading

For more information about what was covered in this chapter, please refer to the following links:

- https://www.xilinx.com/support/documentation/ip_documentation/clk_wiz/v6_0/pg065-clk-wiz.pdf
- <https://glenwing.github.io/docs/VESA-DMT-1.13.pdf>

10

Bringing It All Together

Take a deep breath and reflect on what you've accomplished in getting to this point in the book. You started the journey with little or no SystemVerilog knowledge and were unaware of how to build hardware in an FPGA. Over the course of this book, you've gone from simple logic functions utilizing switches to light LEDs to as far as writing text out on a VGA screen.

In this chapter, we'll investigate the PS/2 interface, which is a way of communicating with a keyboard or mouse that Digilent has chosen to use. We'll then be taking our VGA from *Chapter 9, A Better Way to Display – VGA*, and adapting it to display more data than the resolution we currently have selected. We'll use it to output scan codes from the keyboard so you can see how it operates. We'll also adapt our temperature sensor to display on the VGA. Finally, we'll take the audio captured by the PDM microphone and display it as a waveform on the screen.

By the end of this chapter, you'll have an interactive piece of hardware that displays keyboard scancodes, the temperature in Fahrenheit or Celsius (selectable via the keyboard), and the audio data as a waveform.

In this chapter, we are going to cover the following main topics:

- Using a keyboard – introduction to the USB to PS/2 interface
- Displaying data from the keyboard on the screen
- Converting the temperature sensor to use the VGA display
- Converting the PDM microphone to use the VGA display
- Bringing everything together in a final project

Technical requirements

The technical requirements for this chapter are the same as those for *Chapter 1, Introduction to FPGA Architectures and Xilinx Vivado*.

To follow along with the examples and the project, you can find the code files for this chapter at the following repository on GitHub: <https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH10>.

If you want to implement the project on the board, you'll require a VGA-capable monitor, cable, and USB keyboard.

Important note

The Nexys A7 supports a USB keyboard capable of supporting PS/2 BIOS modes. While writing this chapter, I was only able to find one older keyboard that worked 100%. This is a limitation of the Digilent board as the PIC source code for interfacing the USB to PS/2 is closed source. If you can't find a compatible keyboard or don't want to buy one, you can still view the PS/2 output in the **Integrate Logic Analyzer (ILA)**. Here is one keyboard that is known to work: https://www.amazon.ca/gp/product/B07THJFXJN/ref=ppx_yo_dt_b_search_asin_title?ie=UTF8&psc=1&fkw=alm. Modern gaming-type keyboards do not appear to work reliably.

Now let's look at the keyboard interface on the Digilent boards.

Investigating the keyboard interface

I'm sure you are familiar with computer keyboards as a user, but perhaps not how keyboards are physically implemented.

Keyboards consist of a matrix of switches. When you depress a key, you close a circuit. A keyboard controller activates one line at a time and checks to see which lines are connected, which will identify a unique key (assuming only one key is pressed). It will also detect when a key is released:



Figure 10.1 – Keyboard matrix

The keyboard controller will apply a voltage across each input one at a time. With the voltage applied, it will look at the outputs to identify whether any key is pressed. In *Figure 10.1*, when the controller scans input 2, and key K is depressed, output 2 will be active high.

When IBM introduced the PS/2 computer, they introduced a new keyboard and mouse standard. The keyboard pulled the matrix decoder into the keyboard and simplified the interface to two wires. The protocol consists of 11-bit transfers that consist of a start bit, data byte, odd parity, and stop bit. The data is transmitted from **least significant bit (LSB)** to **most significant bit (MSB)**. On Digilent boards, the keyboard is connected to a USB interface, a PIC microcontroller acts as the PS/2 device, and the FPGA acts as the host:



Figure 10.2 – PS/2 device to host timing

Figure 10.2 shows how a device communicates with the host. The device is always responsible for generating the clock to the host. We can look at how this protocol works by examining the PS/2 state machine:

```
IDLE: begin
    if (counter_100us != COUNT_100us) begin
        counter_100us <= counter_100us + 1'b1;
        xmit_ready      <= '0;
    end else begin
        xmit_ready      <= '1;
    end
    data_counter     <= '0;
    if (~ps2_clk_clean && ps2_clk_clean_last) begin
        counter_100us <= '0;
        state         <= CLK_FALL0;
    end else if (~tx_ready && xmit_ready) begin
        counter_100us <= '0;
        tx_data_out    <= {1'b1, ~^tx_data, tx_data, 1'b0};
        state          <= XMIT0;
    end else if (send_set && xmit_ready) begin
        clr_set         <= '1;
        counter_100us <= '0;
        tx_data_out    <= {1'b1, ~^send_data, send_data, 1'b0};
        state          <= XMIT0;
    end
end
```

In the idle state, we watch for a falling edge of `ps2_clock`: `~ps2_clk_clean && ps2_clk_clean_last`, we receive an external request to send data, or we send the initialization data:

```
CLK_FALL0: begin
    // capture data
    data_capture <= {ps2_data_clean, data_capture[10:1]};
    data_counter <= data_counter + 1'b1;
    state         <= CLK_FALL1;
end
```

```

CLK_FALL1: begin
    // Clock has gone low, wait for it to go high
    if (ps2_clk_clean) state <= CLK_HIGH;
end
CLK_HIGH: begin
    if (data_counter == 11) begin
        counter_100us <= '0;
        done          <= '1;
        err           <= ~^data_capture[9:1];
        state         <= IDLE;
    end else if (~ps2_clk_clean) state <= CLK_FALL0;
end

```

The next three states handle capturing the data from the device:

1. Capture the data when the clock goes low in CLK_FALL0.
2. Wait for the clock to go high in CLK_FALL1.
3. In CLK_HIGH, if we receive 11 data bits, go back to idle, or wait for clock to fall and return to CLK_FALL0.

You can see from the FPGA perspective that the receive protocol is very straightforward. We package up the data for use by the instantiating design using the following code:

```

initial begin
    out_state = OUT_IDLE;
    rx_data   = '0; rx_user   = '0; rx_valid  = '0;
end
always @(posedge clk) begin
    rx_valid <= '0;
    case (out_state)
        OUT_IDLE: begin
            if (done && rx_ready) begin
                rx_data           <= data_capture[8:1];
                rx_user           <= err; // Error indicator
                rx_valid          <= '1;
                if (~rx_ready) out_state <= OUT_WAIT;
            end
        end
    end

```

```

OUT_WAIT: if (rx_ready) out_state <= OUT_IDLE;
endcase
if (reset) out_state <= OUT_IDLE;
end

```

This creates an AXI streaming interface out of the ps2_host module.

The PIC microcontroller that acts as the USB to PS/2 interface is essentially a black box into which we have no visibility. This causes a problem if the keyboards are not behaving as expected. As I debugged the problem of finding a keyboard that worked with the Nexys A7, I developed a complete host interface and generated a startup sequence similar to the ones I found online captured by people during startup:

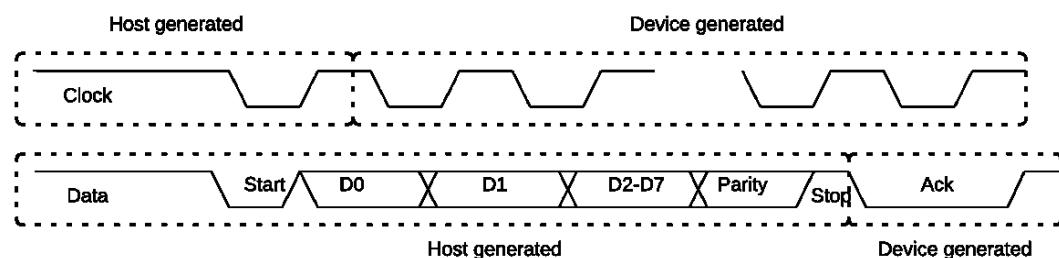


Figure 10.3 – PS/2 host to device timing

When using a PS/2 mouse, host to device communication is required. For keyboards, it's not strictly necessary, although to set the keyboard lights, such as caps lock or num lock, the host controls this by issuing commands. *Figure 10.3* shows how this communication occurs. The protocol operates as per the write portion of the following state machine:

```

XMIT0: begin
    // Drop the clock to signal to device and hold low 100us
end
XMIT1: begin
    // put out the data and release the clock to device
    // Wait 20us
end
XMIT2: begin
    // Every clock negedge advance the data
end
XMIT3: begin
    // Wait for clock to drop
end

```

```

XMIT4: begin
    // Wait for ACK
end
XMIT5: begin
    // Wait for data to go high
end
XMIT6: begin
    // Wait for clock to rise then go back to idle
end

```

A mouse or keyboard can be connected at any one time, but not both. The project only supports a keyboard. As part of my debug efforts, I generated the entire initialization represented in the preceding `start_state` state machine:

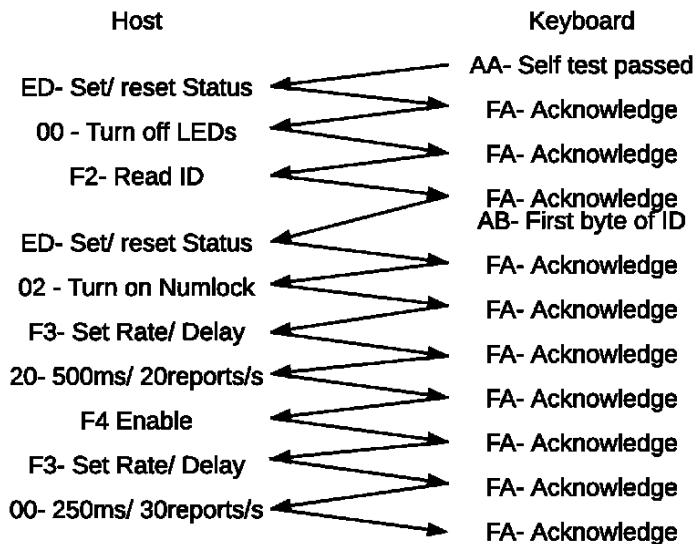


Figure 10.4 – PS/2 initialization

The state machine sends and receives the sequence before bringing up the device. If you experience problems with the keyboard you are trying to use, you can use the **ILA** that we introduced in *Chapter 3, Counting Button Presses*, to determine whether the sequence has been followed properly.

During normal operation, once the keyboard is initialized, scancodes are generated for every keypress:

ESC 76	F1 05	F2 06	F3 04	F4 0C	F5 03	F6 0B	F7 83	F8 0A	F9 01	F10 09	F11 78	F12 07
~ OE	! 16	@ 1E	# 26	\$ 25	% 2E	^ 36	& 3D	* 3E	(46) 45	- 4E	= + 55
TAB 0D	Q 15	W 1D	E 24	R 2D	T 2C	Y 35	U 3C	I 43	O 44	P 4D	{ 54	} 5B
Caps Lock 58	A 1C	S 1B	D 23	F 2B	G 34	H 33	J 3B	K 42	L 4B	:: 4C	;" 52	Enter 5A
Shift 12	Z 1A	X 22	C 21	V 2A	B 32	N 31	M 3A	,< 41	.> 49	/? 4A	Shift 59	
Ctrl 14	Alt 11	Space 29					Alt E0 11	Ctrl E0 14				

Figure 10.5 – PS/2 keyboard scan codes

On the Digilent board, when a key is pressed on the keyboard, the PIC will convert the USB protocol to PS/2 and generate a scancode representing the keypress to the FPGA. *Figure 10.5* shows the scancodes for most keys. If a key is shifted, a shift modifier code is sent prior to the key's scancode. If a key is held down, the key will be repeatedly sent every 100 ms. When a key is released, an F0 scancode is sent along with the keycode. Finally, there are some extended keys that will have an E0 code sent prior to the scancode. When this type of key is released, an 0xE0 and 0xF0 value will be sent to represent a key up event. The FPGA can also communicate with the keyboard for setting the caps lock or num lock LEDs.

Now, let's look at developing a project where we can test the capabilities of the keyboard.

Project 12 – keyboard handling

We've looked at what the PS/2 interface looks like. Let's now put together a simple interface so that we can test our knowledge before we move on to our design integration. The first step is that we need to debounce our PS/2 signals. I've put together a debounce circuit and test bench so we can verify it. This cannot be built as is, but let's look at it. Open up <https://github.com/PacktPublishing/Learn-FPGA-Programming/blob/master/CH10/build/debounce/debounce.xpr>. This version of the code will act as a reusable core. We want to make sure that we only change state after we've seen the CYCLES number of the same value. This will act as our debouncing circuit.

The interface is straightforward, as we can see in the following code:

```
module debounce
  #(parameter CYCLES = 16)
  (input wire clk,
   input wire sig_in,
   output logic sig_out);
```

The actual debouncing is handled by checking to see whether we have maintained the same state for the CYCLES period. Notice that we double clock the `sig_in` signal to make sure we don't have metastability problems:

```
always @(posedge clk) begin
  sig_in_sync <= sig_in_sync << 1 | sig_in;
  if (sig_in_sync[1] != current_state) begin
    current_state       <= sig_in_sync;
    cycle_count        <= '0;
  end else if (cycle_count == CYCLES) begin
    cycle_count        <= '0;
    sig_out            <= current_state;
  end else begin
    cycle_count        <= cycle_count + 1'b1;
  end
end
```

The nice thing about a small design like this is that it's easy to test exhaustively, as can be seen in the following test bench:

```
initial begin
  sig_in      = '0;
  // Test that we don't switch states too soon
  for (int i = 0; i < CYCLES; i++) begin
    sig_in      = '1;
    repeat (i) @(posedge clk);
    sig_in      = '0;
    repeat (CYCLES-i) @(posedge clk);
  end
```

```

    sig_in      = '1;
repeat (100) @(posedge clk);
for (int i = 0; i < CYCLES; i++) begin
    sig_in      = '0;
repeat (i) @(posedge clk);
    sig_in      = '1;
repeat (CYCLES-i) @(posedge clk);
end
sig_in      = '0;
repeat (100) @(posedge clk);
$display("Test Finished!");
$finish;
end // initial begin

```

The first loop incrementally changes the number of cycles, the signal is high until we reach the CYCLES threshold and the debounced output switches. Similarly, the second loop does the opposite to change from high to low. *Figure 10.6* shows the simulation output:



Figure 10.6 – Debounce simulation

We have a good debounce circuit, so we can move on to our PS/2 code. We'll make our keyboard handler use AXI streaming to more easily integrate into other designs. The interface to our core will be designed as follows:

```

module ps2_host
#(parameter          CLK_PER = 10,
parameter          CYCLES  = 16)
(input wire          clk,
inout              ps2_clk,
inout              ps2_data,
// Transmit data to the keyboard from the FPGA
input wire          tx_valid,
input wire [7:0]     tx_data,

```

```

    output logic      tx_ready,
    // Data from the device to the FPGA
    output logic [7:0] rx_data,
    output logic      rx_user, // Error indicator
    output logic      rx_valid,
    input wire        rx_ready
);

```

We have our two tristate signals, ps2_clk and ps2_data. We have a transmit interface that is, as yet, undeveloped. This interface could be used to set caps lock, repeat rate, or other parameters the keyboard can receive. There is a second bus that reports data received from the keyboard. We do have a user signal, which we'll use to report a parity error if detected:

```

// Clean up the signals coming in
debounce
#(.CYCLES  (CYCLES))
u_debounce[2]
(.clk      (clk),
.sig_in   ({ps2_clk,          ps2_data}),
.sig_out  ({ps2_clk_clean, ps2_data_clean})
);

```

The first step will be to add an array of instances to instantiate two debouncing circuits on the ps2 data lines:

```

// Enable drives a 0 out on the clock or data lines
assign ps2_clk = ps2_clk_en ? '0 : 'z;
assign ps2_data = ps2_data_en ? '0 : 'z;

```

We'll need the tristate on data regardless. The tristate on clock is needed for the master implementation. Note that when enable is asserted, we drive a low signal. We tristate the output when driving a one on the enable, relying on the pullup to raise the logic level high.

With the keyboard interface designed, let's look at how we might test this.

Testing the PS/2

Now we've got our PS/2 keyboard state machine and we can write a quick test bench. Open <https://github.com/PacktPublishing/Learn-FPGA-Programming/blob/master/CH10/build/ps2/ps2.xpr>. We have a test that can be used to verify that scancodes from the keyboard can be received properly. The main component of the state machine is the `send_key` task. This task takes in a scancode and converts it to a PS/2 interface:

```
task send_key;
    input [7:0] keycode;
    input      error;
begin
    // Generate the PS/2 timing to send the keycode and use
    // error to generate good/ bad parity
end
endtask // send_key
```

A SystemVerilog task is used to encapsulate a series of events that can have timing. In this particular case, we are generating the PS/2 data stream for the host.

We'll also add another task to handle the keyboard receive:

```
task rx_key;
    input [7:0] exp_data;
begin
    // Wait for ED
    edge_count = '0;
    // Wait for first falling edge, then rising edge
    @(negedge ps2_clk);
    @(posedge ps2_clk);
    while (edge_count < 10) begin
        repeat (100) @(posedge clk);
        ps2_clk0 = '1;
        repeat (100) @(posedge clk);
        if (edge_count == 10) ps2_data0 = '1;
        data_capt[edge_count++] <= ps2_data;
```

```
ps2_clk0 = '0;
end
repeat (100) @(posedge clk);
ps2_data0 = '1;
repeat (100) @(posedge clk);
ps2_clk0 = '1;
repeat (100) @(posedge clk);
ps2_data0 = '0;
ps2_clk0 = '0;
repeat (100) @(posedge clk);
$display("Captured data: %h", data_capt[8:1]);
if (data_capt[8:1] != exp_data) begin
    $error("Data miscompared! Expected %h != Received %h",
          exp_data, data_capt[8:1]);
end
end
endtask // rx_key
```

This task, or a similar block in the test bench, is required since there is handshaking between the test bench (device) and the host, and the device is responsible for generating the clock. One thing to note about the `rx_key` task is that it doesn't maintain proper timing. I chose to go down this route as the host is only detecting edges and this will speed up the simulation. In general, however, it's good practice to match your simulations to what the actual signals look like as this can uncover obscure problems you may otherwise miss.

We'll also want to implement a self-checking function. In order to do this, we'll also introduce a construct that allows parallel operation, the `fork...join` function. *Figure 10.7* shows conceptually what we are trying to accomplish by running the stimulus in one process and the checking logic in another:

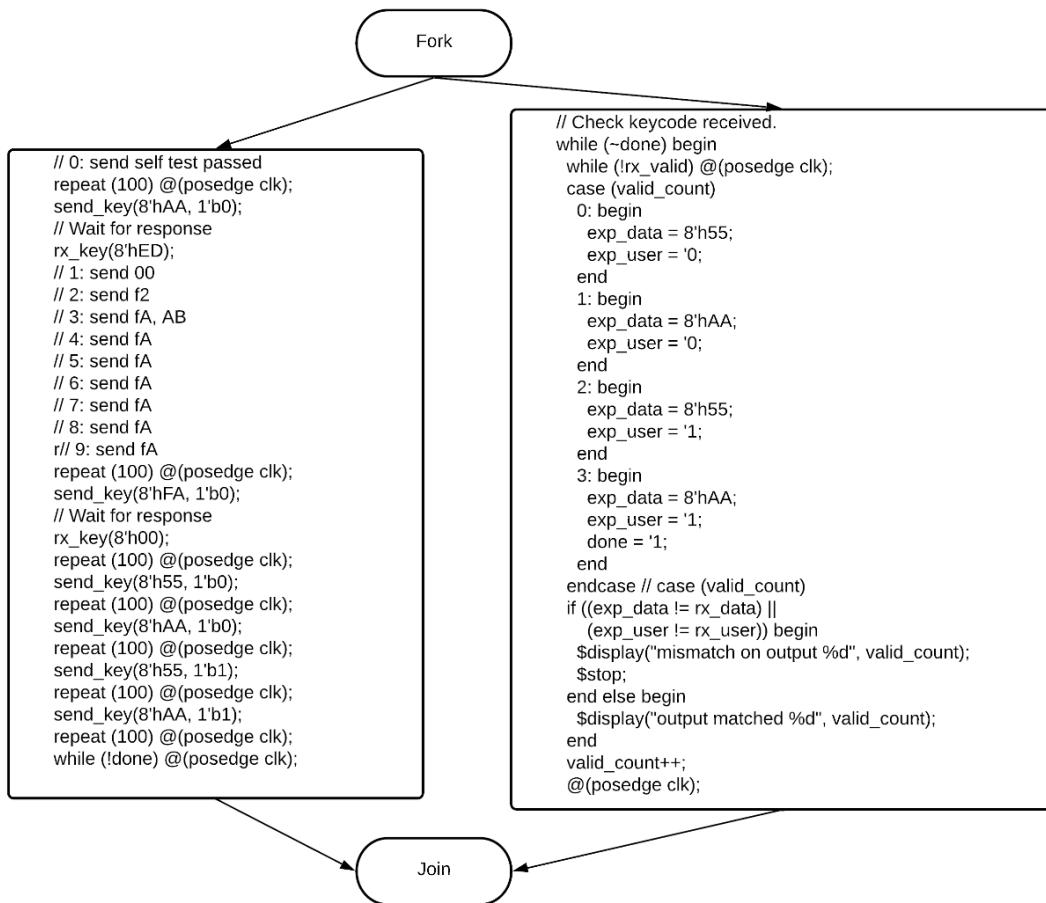


Figure 10.7 – Fork join conceptually used in the test bench

The left and right blocks after the `fork` represent `begin...end` blocks. Since they are within the `fork...join` keywords, the two `begin...end` blocks will run in parallel to one another. The left side generates the stimulus and responses, while the right side checks the last four send key combinations. We can run this in the simulator:

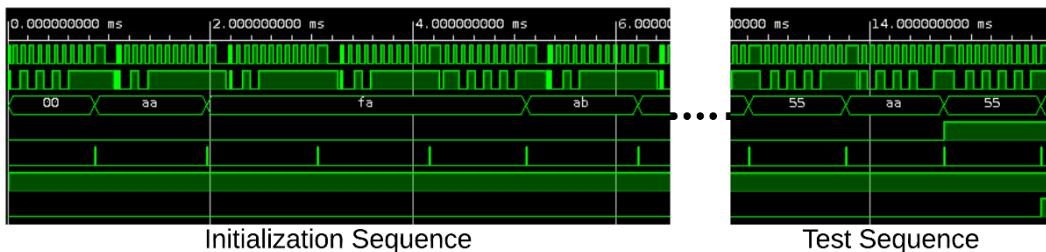


Figure 10.8 – PS/2 test sequence

Figure 10.8 shows the waves from simulating the PS/2 test bench. The initialization sequence represents the keyboard self test passed through the boot sequence of a computer. The test sequence contains two sets of good data and two sets of errored data.

Now we've got a better way of getting data into the FPGA via the keyboard and we'll use it in our final project.

Project 13 – bringing it all together

You should take a moment to consider the path you've taken over the course of the book. In the beginning, you toggled some switches and lit some lights. You've built some simple designs, such as a calculator and traffic light controller. You've captured and converted temperature sensor information, captured audio data, and displayed data on a VGA monitor.

Now we'll look back on these projects to gather a few of them and combine them into a final design. The base will be the VGA we created in *Chapter 9, A Better Way to Display – VGA*. This will allow us to easily display text or graphics. In the previous section, we simulated the PS/2. However, we haven't seen it in operation. Luckily, every keypress generates at least 3 bytes, 1 byte for keydown and 2 bytes for keyup for most keys. We can come up with a clever way of displaying this to the screen. Finally, we can look at the audio data. We can see the data in the ILA, but what if we could view the waveform on the screen?

This project is contained within https://github.com/PacktPublishing/Learn-FPGA-Programming/blob/master/CH10/build/final_project/final_project.xpr.

Displaying PS/2 keycodes on the VGA screen

Our PS/2 host interface block provides a convenient way of capturing information via the streaming interface. This interface provides one byte at a time, which we receive from the keyboard. Let's take a look at how we might capture and display the information on the screen.

To display the VGA mode information, we created a 16-byte (128-bit) array to store the information. This fits nicely into our DDR2 interface implementation, so we'll maintain similar arrays for the PS/2 characters. Since every byte from the PS/2 takes two bytes in our character array, we can define our storage as follows:

```
typedef struct packed
{
    logic [7:0] data;
    logic       error;
} ps2_t;

localparam PS2_DEPTH = 8;

ps2_t          ps2_data_capt;
logic [PS2_DEPTH*2-1:0] [7:0] ps2_data_store;
logic           ps2_toggle;
(* async_reg = "TRUE" *) logic [2:0] ps2_sync;
logic           update_ps2;
logic           clear_ps2;
```

We'll run the PS/2 interface on the 200 MHz clock. Luckily, we made our design clock frequency independent and we can let it know how fast we'll run it by specifying the clock period. The interface from the PS/2 to the VGA will be asynchronous, so we do have to consider clock domain crossing.

Since we'll also be using the keyboard to select between Fahrenheit and Celsius, we'll need to detect when the C or F key is pressed and keep track of the state:

```
// toggle sync and capture the data
always @(posedge clk200) begin
    if (ps2_rx_valid) begin
        ps2_toggle     <= ~ps2_toggle;
        ps2_data_capt <= '{data: ps2_rx_data, error: ps2_rx_err};
        case (ps2_rx_data)
            8'h2B: ftemp <= '1; // F = fahrenheit
            8'h21: ftemp <= '0; // C = celsius
```

```

endcase
end
end

```

If you recall the AXI streaming interface, we'll get a valid signal along with the data. When `ps2_rx_valid` goes high, we'll toggle a signal that we can capture on the `ui_clk` signal. We'll store the data in the structure alongside the signal. Finally, we'll look for the scancode for *F* and *C*, `0x2B` and `0x21`, respectively. We'll keep track of whether we want Fahrenheit or Celsius by using an `ftemp` signal.

On the `ui_clk` domain, we'll look for an edge on the synchronized toggle signal. We'll create a shift register, as shown in *Figure 10.9*. PS/2 scancodes will get the hexadecimal number converted to ASCII characters, they will be pushed into location 0, and each character position will be pushed along the pipeline:



Figure 10.9 – Shift register

Shift registers are a common design component, so much so that the slices in the Xilinx FPGAs have special modes to more optimally use them as shift registers.

We can see how this is coded in the final project:

```

if (^ps2_sync[2:1]) begin
    update_ps2 <= '1;
    for (int i = PS2_DEPTH-1; i >= 0; i--) begin
        if (i == 0) begin
            for (int j = 1; j >= 0; j--) begin
                // Convert nibble into a character
            end
        end else begin
            ps2_data_store[i*2+2] <= ps2_data_store[(i-1)*2+2];
        end
    end // for (int i = 0; i < PS2_DEPTH; i++)
end

```

The outer loop works from the uppermost character, copying the next lower character into it until it reaches character 0, when we convert each nibble into an ASCII character.

Once the register is loaded, we signal the text state machine to update the screen:

```
end else if (update_ps2) begin // if (^update_text_sync[2:1])
    // We'll start the PS2 output on line 8
    y_offset      <= 8 * real_pitch;
    clear_ps2     <= '1;
    char_index    <= ps2_data_store[0];
    capt_text     <= ps2_data_store;
    s_ddr_awvalid <= '0;
    s_ddr_wvalid  <= '0;
    text_sm       <= TEXT_WRITE0;
```

We are reusing the same interface we had with the VGA, modified slightly so that we can pass in the `char_index` signal, but also the `capt_text` signal that we'll use. We'll expand upon this for the temperature sensor and audio data. When we run this in the final project, you'll see the scan codes' output on the display, as shown in *Figure 10.10*:



Figure 10.10 – Displaying scancodes as keys are pressed

Each keystroke pushes one or more bytes into the shift register from the left to the right. This is why the third byte, B2, which is actually 0x2B in hex, is the make code, f0 is the break code, and B2 is the F key. These three bytes represent pressing and releasing the F key. You'll also notice a string of AF values, again 0xFA, since it is nibble swapped, which represents the keyboard acknowledge if you examine the initialization routine.

With this code, we now can display 8 bytes of PS/2 data on the screen. Let's now look at adding the temperature sensor.

Displaying the temperature sensor data

Previously in *Chapter 6, Math, Parallelism, and Pipelined Design*, we developed the floating-point temperature sensor module. We now need to take the data that would be displayed to the 7-segment display and convert it to ASCII and display it on the VGA screen. To accomplish this, I've created a wrapper, `i2c_wrapper`. This module encapsulates the `i2c_temp_flt` module we developed previously and creates the output string, again adhering to the 16 characters we defined previously.

Recall that `temp_valid` and `encoded` are the outputs from the temperature sensor core. The value on `encoded` is a decimal representation, with the decimal point always at position 4. We also have the ability to select Fahrenheit or Celsius, so we'll want to add an `F` or `C` to the output to differentiate the mode we are in:

Final_project.sv

```
always @(posedge clk) begin
    if (temp_valid) begin
        update_temp           <= ~update_temp;
        capt_temp             <= "      F 0000.0000";
        capt_temp[9]          <= 8'h0C; // Degree symbol
        if (ftemp) capt_temp[10] <= "F";
        else      capt_temp[10] <= "C";
        for (int i = 7; i >= 0; i--) begin
            if (i > 3) begin
                capt_temp[7-i] <= 8'h30+encoded[i];
            end else begin
                capt_temp[8-i] <= 8'h30+encoded[i];
            end
        end
    end
end // always @ (posedge clk)
```

https://github.com/PacktPublishing/Learn-FPGA-Programming/CH10/hdl/text_rom.sv

The key points are the `update_temp` toggle signal, since we are running on the 200 MHz clock domain and need to have a clean way of signaling our display function that new data is available. We define the format of `capt_temp` and override the F/C based upon the `ftemp` signal.

We can use a trick that the ASCII for 0-9 is 0x30-0x39. The loop spaces the digits around the decimal point, and we add the integer value to 0x30 to give us the ASCII character to display.

To display the string, we'll use the same function in the text state machine:

```
end else if (update_temp_capt) begin
    // We'll start the temperature output on line 16
    y_offset      <= 16 * real_pitch;
    update_temp_capt <= '0';
    char_index     <= capt_temp[0];
    capt_text      <= capt_temp;
    s_ddr_awvalid  <= '0';
    s_ddr_wvalid   <= '0';
    text_sm        <= TEXT_WRITE0;
```

When the synchronized update is captured and we are not working on another text update, we'll pass along the captured temperature string and output to the display. One additional enhancement we can make is to create a custom character to represent the degrees symbol.

Adding a custom character to the text ROM

One addition we made is to add the degree symbol to the ROM characters. You can see where we set this:

```
capt_temp[9]          <= 8'h0C; // Degree symbol
```

I've selected the first empty location, 0x0C, in the text ROM and created a representation for the degrees symbol. In *Figure 10.11*, you can see how a character is constructed:

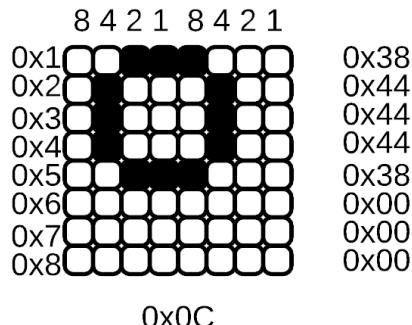


Figure 10.11 – Constructing the degree symbol

There are eight scanlines for every character. Each scanline represents the bits to display. To calculate the bytes, every lit pixel in each nibble needs to be added together. By doing this, we can construct the lookup for the symbol:

```
// Degree Symbol
{8'h0C, 3'h0}: bitmap <= 8'h38;
{8'h0C, 3'h1}: bitmap <= 8'h44;
{8'h0C, 3'h2}: bitmap <= 8'h44;
{8'h0C, 3'h3}: bitmap <= 8'h44;
{8'h0C, 3'h4}: bitmap <= 8'h38;
{8'h0C, 3'h5}: bitmap <= 8'h00;
{8'h0C, 3'h6}: bitmap <= 8'h00;
{8'h0C, 3'h7}: bitmap <= 8'h00;
```

Now that we've completed the temperature sensor data, let's look at how we can display the audio data.

Displaying audio data

The final display section will actually display the waveform as a raw graphic. There are a couple of choices in terms of how we display the information, but in the end, we'll want to show the amplitude of the wave.

Normally, these types of waveforms are displayed across a screen from left to right, as in *Figure 10.12*:

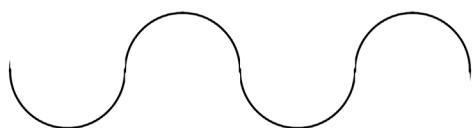


Figure 10.12 – Typical sine wave representation

This type of implementation cannot be done easily and efficiently at the hardware level. For the purposes of the final project, I propose that we display the output vertically. Recall the pdm_input module. It captures a 7-bit audio sample that can natively be represented as a dot on a 128-bit scanline segment.

Since we created a reusable design where we can specify the clock frequency, we can use the `pdm_inputs` core as we created it. We will need to add some external logic to buffer the data. Since we'll be displaying vertically, we are limited to fewer than 480 scan lines. We'll limit our display area to 256 scanlines of audio. Since we'll be plotting data and capturing samples in parallel, we'll need a simple dual port RAM, one read, and one write port:



Figure 10.13 – Buffering audio data for display

We'll create a storage buffer of 1,024 samples, although 512 would be more than enough. The reason for not using only 256 samples is that there is a possibility of overwriting data before it's read. This will ensure that the 256 samples plotted are consecutive:

```
always @(posedge clk200) begin
    if (amplitude_valid) begin
        amplitude_store[amp_wr] <= amplitude;
        amp_wr                  <= amp_wr + 1'b1;
    end
    amp_data <= amplitude_store[amp_rd];
end
```

The storage is inferred as we discussed back in *Chapter 5, FPGA Resources and How to Use Them*, the only difference being that the `amp_wr` pointer is incremented on every sample.

Finally, we create a signal that will update the display on every VSync. This required a modification to the VGA to generate a toggle signal on every VSync. I decided to put the logic in the `vga_core` module, as VGA sync polarity changes for different resolutions.

To pass the data to the text state machine, I decided to use a FIFO. This allows us to easily cross clock domains, and we can pass the vertical location as well as the scanline segment we want to display.

I create a state machine, `wave_sm`, that handles the scanline generation:

```

case (wave_sm)
    WAVE_IDLE: begin
        if (^vga_sync_toggle_sync[2:1]) begin
            // get the amplitude data from behind the write pointer
            // by 256 samples
            amp_rd    <= amp_wr - 256;
            rd_count <= '0;
            wave_sm   <= WAVE_READ0;
        end
    end
    WAVE_READ0: begin
        // address to ram valid this cycle
        amp_rd    <= amp_rd + 1'b1;
        rd_count <= rd_count + 1'b1;
        wave_sm   <= WAVE_READ1;
    end
    WAVE_READ1: begin
        // address to ram valid this cycle
        amp_rd      <= amp_rd + 1'b1;
        rd_count    <= rd_count + 1'b1;
        pdm_push    <= '1;
        pdm_din.address <= 31 + rd_count;
        pdm_din.data    <= 1'b1 << amp_data;
        if (rd_count[8]) wave_sm <= WAVE_IDLE;
    end
endcase // case (wave_sm)

```

When `vga_sync` toggles, we look back by rewinding 256 samples and then read out and push the display segments for 256 scanlines. On the text state side, we will write one scanline at a time:

```

end else if (!pdm_empty) begin
    pdm_pop          <= '1;
    char_y           <= '1; // Force only one line to be written
    update_temp_capt <= '0;
    s_ddr_awvalid   <= '1;

```

```
s_ddr_awaddr    <= pdm_dout.address * real_pitch;  
s_ddr_wvalid    <= '1;  
s_ddr_wdata     <= pdm_dout.data;  
text_sm         <= TEXT_WRITE2;
```

We handle this by setting the `char_y` count to 7, so we'll only write one scanline for every FIFO pop. We'll also trigger the write here since we don't need to loop over multiple scanlines.

At this point, you should build the project and see what it looks like on the board. Once you download the bitstream, you'll be greeted with a display as seen in *Figure 10.14*:



Figure 10.14 – Initial bringup

I would recommend playing some audio or downloading a tone generator to get a more interesting output, as shown in *Figure 10.15*:



Figure 10.15 – Final project output

This screenshot was taken of the final project capturing a tone from a cell phone tone generator application. You can see all the components represented here:

- Resolution
- Scancodes
- Temperature sensor in Fahrenheit
- The audio capture

You've now completed the capstone project of the book. You've brought together some reusable components and created a useful application from them. No longer restricted by a few LEDs, you've created a graphical display and added output text and graphics to it.

Summary

In this chapter, we've explored the PS/2 keyboard interface by creating an interface that can write to and receive data from the keyboard. With the PS/2 ready to use, we've then taken pieces from the last few chapters: the VGA to display our data, the temperature sensor to provide some numerical output, and the PDM interface so we can add something more graphics-oriented. You've now completed the journey from basic logic gates to coding something that can display text and graphics on the screen. It's possible to go much further with writing pure SystemVerilog, but looking at a soft processor is a good next step.

There is one final chapter containing some more advanced constructs that can help in your design and simulation and then you'll be ready to tackle your own design challenges.

Questions

1. PS/2 keyboards use a two-wire interface consisting of:
 - a) Keyup/keydown
 - b) Clock/data
 - c) Data in/data out
2. A scancode is generated whenever a key is:
 - a) Pressed
 - b) Released
 - c) Held down
 - d) All of the above

3. To display the scancodes on the VGA, we used:
 - a) A hex to ASCII converter
 - b) A shift register
 - c) A BCD encoder
 - d) (a) and (b)
4. To display audio data, how was the text state machine modified?
 - a) It takes in 128 bits of graphical data and writes that to the correct address for that scanline using `text_sm`.
 - b) The graphics are mapped to characters and we reuse the `text_sm` state variable.
 - c) We created a new graphics state machine.
5. To trigger an audio update, we:
 - a) Update on every sample captured
 - b) Update every second
 - c) Update on every vertical sync
 - d) Update whenever nothing else is going on

Challenge

Usually, audio data is displayed horizontally. Can you modify the code to create a horizontal display? This is a challenging problem and could take a little while to get right. Here are a couple of hints:

- You may want to clear the area and then simply plot the dots that need to be set.
- You may want to buffer up 128 bits of each scanline and use the existing FIFO interface to display the data.

There are a bunch of ways to accomplish this, but you should be able to find one that works.

Further reading

For more information about what was covered in this chapter, please refer to the following link:

- <https://www.avrfreaks.net/sites/default/files/PS2%20Keyboard.pdf>

11

Advanced Topics

Over the course of the book, you've had the opportunity to try your hand at a few different projects. To get you started quickly, we limited some of the syntax. This chapter will introduce a few new constructs you may find useful for synthesis and verification. I'll also introduce some things to watch out for.

By the end of this chapter, you'll have been exposed to almost all the useful SystemVerilog constructs for designing and testing FPGAs.

In this chapter, we are going to cover the following main topics:

- Exploring more advanced SystemVerilog constructs
- Exploring some more advanced verification constructs
- Other gotchas and how to avoid them

Technical requirements

The technical requirements for this chapter are the same as those for *Chapter 1, Introduction to FPGA Architectures and Xilinx Vivado*.

To follow along with the examples and the project, you can find the code files for this chapter at the following repository on GitHub: <https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH11>.

Exploring more advanced SystemVerilog constructs

We've used many basic constructs in our designs. The syntax we've used is enough to construct anything you would like to design. There are some other design constructs that can be useful, so I'd like to at least introduce them with an example of how to use them. The most useful construct is the interface.

Interfacing components using the interface construct

SystemVerilog interfaces can be thought of as modules that straddle other modules. An interface in its simplest form is a bundle of wires, very much like a structure. However, unlike a structure, the direction of each individual signal is independent, meaning that you can have both inputs and outputs defined within the interface.

I've created a project to show ps2_host implemented using an interface: https://github.com/PacktPublishing/Learn-FPGA-Programming/blob/master/CH11/build/ps2_host/ps2_host.xpr

Interfaces also have the added advantage that you can encapsulate functions, tasks, and assertions that are related to the given interface signals. This improves design reusability and improves design development. Most of the designs we have worked with so far have only had a few levels of nested modules. Large designs can have signals that can delve many times deeper. By encapsulating an **interface**, adding, removing, or resizing a signal becomes as easy as modifying the interface definition.

We can look at modifying the ps2_host module to use an interface. Remember that our original version back in *Chapter 10, Bringing It All Together*, had an interface that looked as shown in the following code:

```
// Transmit data to the keyboard from the FPGA
input wire          tx_valid,
input wire [7:0]    tx_data,
output logic        tx_ready,
// Data from the device to the FPGA
output logic [7:0]  rx_data,
output logic        rx_user, // Error indicator
output logic        rx_valid,
input wire          rx_ready
```

This type of design is a good example to implement an interface for. It's a good idea to maintain a constant naming convention. What I typically do is name an interface <interface_name>.intf and save each interface in its own .sv file.

Interfaces can contain parameters and a port list like a module. In ps2_intf, we won't need these; however, we will take advantage of the function encapsulation:

```
interface ps2_intf;
    // Interfaces can contain parameter lists like a module
    // Interfaces can contain IO like a module
    logic          tx_valid;
    logic [7:0]     tx_data;
    logic          tx_ready;

    logic [7:0]     rx_data;
    logic          rx_user;
    logic          rx_valid;
    logic          rx_ready;
```

The first part of the interface defines the signals within the interface itself. The second section contains **modports**. Modports allow us to define the direction of signals. If a modport is not used, the signal is considered bidirectional:

```
modport master
    (output tx_valid,
     output tx_data,
     input  tx_ready,
     input  rx_data,
     input  rx_user,
     input  rx_valid,
     output rx_ready);
modport slave
    (input  tx_valid,
     input  tx_data,
     output tx_ready,
     output rx_data,
     output rx_user,
     output rx_valid,
```

```

    input rx_ready,
import parity_gen,
import parity_check);

```

The things to notice here are the `import` keywords on the slave modport. These allow the functions to be used within the slave interface. This allows us to limit the visibility of functions or internal signals to only authorized modports. This allows related functions such as the parity functions in the following code to be used by the instantiating module:

```

function parity_gen(input [7:0] din);
begin
    return ~^din;
end
endfunction // parity_gen
function parity_check(input [8:0] din);
begin
    return ~^din;
end
endfunction // parity_check

```

By encapsulating the functions within the interface, we keep everything needed to generate a PS/2 command or check the incoming scancode. This will help reusability in the future:



Figure 11.1 – Interface signals in the Vivado simulator

You can see from *Figure 11.1* that the interface shows up separately in the signal list almost like a module. You can also see the modports show up when expanded. If you add the signals to the wave viewer, they look the same as any other signal:

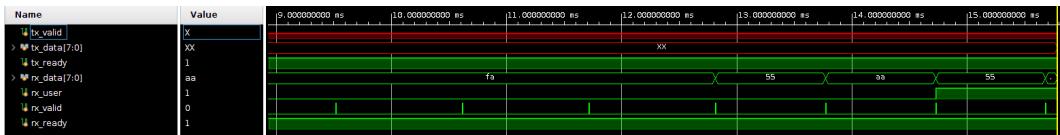


Figure 11.2 – Simulation waveform for interfaces

The test included in this project will pass when run. We'll look a little closer at the testbench when we discuss queues. Interfaces are completely optional to use. Some people find them very useful, others don't like to use them. If they work for you, they can really come in handy.

There are some Vivado limitations currently, however. The top-level module cannot use interfaces as ports. **Block Design (BD)** subdesigns cannot use interfaces either. If you limit the connections within a design using SystemVerilog, you will be fine.

Now let's look at structures in a little more detail.

Using structures

Throughout the book, we've used structures. They provide a convenient way to package data and make your design easier to follow. Recall in *Chapter 9, A Better Way to Display - VGA*, we used a structure to hold our resolution information for the VGA:

```
typedef struct packed {
    logic [7:0] divide_count;
    logic [15:8] mult_integer;
    logic [25:16] mult_fraction;
    ...
    logic hpol;
    logic vpol;
    logic [12:0] pitch;
} resolution_t;
```

Creating a `typedef` of a structure as in the preceding code effectively creates a new type. You can create packed and unpacked arrays of structures or use them as you would any other type. An example of this is creating our table of resolutions:

```
resolution_t resolution[18];
```

We have a couple of ways of assigning to. The first is assigning by component, which we used in *Chapter 9, A Better Way to Display – VGA*:

```
// 25.18 Mhz 640x480 @ 60Hz
resolution[0].divide_count      = 8'd9;
resolution[0].mult_integer     = 8'd50;
resolution[0].mult_fraction    = 10'd000;
...
resolution[0].hpol             = '0;
resolution[0].vpol             = '0;
resolution[0].pitch            = 13'd5*16; // 5 rows at
1bpp
```

Another way is to assign by name:

```
// 25.18 Mhz 640x480 @ 60Hz
resolution[0] = '{default:          '0,
                  divide_count:   8'd9,
                  mult_integer:   8'd50,
                  mult_fraction: 10'd000,
...
                  Pitch:           13'd5*16}; // 5 rows at 1bpp
```

When assigning in this way, you can use the `default` keyword to assign values to anything not specified. The preceding two snippets of code are equivalent.

Block labels

Any `begin...end` block can be labeled. I've created an example project to show this: <https://github.com/PacktPublishing/Learn-FPGA-Programming/blob/master/CH11/build/labels/labels.xpr>.

This is highly recommended for generate statements as this will be a requirement in future versions of SystemVerilog. This can help with readability. Block labels can also be useful if using `disable`, as we'll see. In the following code, we'll look at an example of how block labels can help catch coding errors:

```
always_ff @(posedge clk) begin
// mismatched block label
if (subtraction) begin : l_addition_op
dout <= in0 - in1;
```

```

end : l_subtraction_op
// reusing a label
if (addition) begin : l_addition_op
    dout <= in0 + in1;
end : l_addition_op
end

```

Labeling can also help to keep track of which block a section of code is in. Labels cannot be reused. Also, errors will result if the start and end label are not the same. Running the preceding code through the Vivado simulator yields the following:

```

ERROR: [VRFC 10-3516] mismatch in closing label 'l_subtraction_
op'; expected 'l_addition_op' [/home/fbruno/git/books/Learn-
FPGA-Programming/CH11/hdl/labels.sv:12]
ERROR: [VRFC 10-2934] 'l_addition_op' is already declared [/
home/fbruno/git/books/Learn-FPGA-Programming/CH11/hdl/labels.
sv:16]
ERROR: [VRFC 10-3516] mismatch in closing label 'l_addition_
op'; expected '<unnamed>' [/home/fbruno/git/books/Learn-FPGA-
Programming/CH11/hdl/labels.sv:16]
ERROR: [VRFC 10-2865] module 'labels' ignored due to previous
errors [/home/fbruno/git/books/Learn-FPGA-Programming/CH11/hdl/
labels.sv:1]

```

Labeling is optional, but can be worthwhile if you practice it. Now let's look at looping constructs.

Looping using for loops

We have been using `for` loops throughout the book. In every case, we have defined the loop variables within the `for` loop and this is highly desirable. `for` loops also allow for multiple loop variables, although only one test is allowed for ending the loop. Here's an example:

```
for (int i = 0, j = 0; i *j < 256; i++, j+=8) begin
```

The preceding example is perfectly fine for synthesis and implementation.

Looping using do...while

We have seen while loops, particularly in our PS/2 testing in *Chapter 10, Bringing It All Together*. do...while and while loops are synthesizable.

We can see two implementations of a `last_ones` function using both types of loops:

```
always_comb begin
    done = '0;
    i = 0;
    while (!done) begin
        if (vector[i] || (i==15)) done = '1;
        else i += 1;
    end
    last_ones = i;
end
```

Another way of coding this would be to use a do...while loop as shown in the following code. do...while loops are particularly useful when you want the loop to execute at least once:

```
always_comb begin
    done = '0;
    i = 0;
    do
        if (vector[i] || (i==15)) done = '1;
        else i += 1;
    while (!done);
    last_ones = i;
end
```

These could both be used to detect the smallest bit position set. As mentioned with `for` loops, as long as you can unroll the loop it's synthesizable.

We have looked at simple loops, but what if you have a more complex operation involving nested loops?

Exiting a loop using disable

We have used `break` in a `for` loop to exit when finding a bit set. What can we do if we have nested loops? The `disable` statement allows you disable a named block. This is synthesizable, much like the `break` statement is synthesizable:

```
always_comb begin
    first_ones = '0;
    for (int i = 0; i < 4; i++) begin : outer_loop
        for (int j = 0; j < 8; j++) begin : inner_loop
            if (din[i][j]) begin
                first_ones = 6'(i*8 + j);
                disable outer_loop;
            end
        end : inner_loop
    end : outer_loop
end // always_comb begin
```

In the preceding code, we are searching a two-dimensional array, `din`, for the first one detected. Once found, we want to capture its position in the array, `6' (i*8 + j)`, and stop the search.

In addition to `break` and `disable`, SystemVerilog also supports `continue`.

Skipping code using continue

Back in *Chapter 4, Let's Build a Calculator*, we developed a leading 1s indicator. We could recode it to use `continue`:

```
always_comb begin
    LED = '0;
    for (int i = $low(SW); i <= $high(SW); i++) begin
        if (~SW[i]) continue;
        LED = i + 1;
    end
end
```

As an alternative to breaking a loop, we might want to skip over something in the loop if we should encounter it. The preceding loop shows how we can skip all 0s in a vector, only capturing the position of the final 1 detected.

Using constants

Constants provide a way of letting the tool know something cannot be changed during execution:

```
// do not allow changing this during execution
const int bus_width = 8;
```

If bus_width is used on the left-hand side of an equation in the design, the tool will produce an error.

In this section, we had a look at a few constructs for design. You may find some of them worthy of further investigation and use, but they are certainly optional, so use or ignore them as you see fit.

Now let's look at some constructs for improving the simulation of your designs.

Exploring some more advanced verification constructs

The testing we have done thus far has been pretty simple, even when we used self-checking. There is one construct that I have found very useful over the years. The queue is easy to use and understand.

Introducing SystemVerilog queues

Often, you need to generate an input in a design that will produce an expected output some time later. Examples of this are parsing engines, data processing engines, and, as we saw in *Chapter 9, A Better Way to Display – VGA, the PS/2 interface*.

When I modified the ps2_host module, I decided to upgrade the testbench for it using queues. I had to create a structure to define what I wanted to store in the queue:

```
typedef struct packed
{
    logic [7:0] data;
    logic       parity;
} ps2_rx_data_t;
```

This structure will store our expected data as we generate data in the ps2_host for testing.

A queue is defined as follows:

```
ps2_rx_data_t ps2_rx_data[$];
```

It looks much like an unpacked array, except the size is defined as `[$]`, which defines it as a queue that can be manipulated in tests. We can access the queue by pushing to the front or back and popping from the back or front. These functions, along with the `size()` operator, are the most useful for simulating. There are other functions for inserting or deleting, and I recommend further research if you think it might be useful for your application.

The following diagram shows the queue conceptually. Typically, you will push into one side and pop from the other. The choice is arbitrary. You will note that it's possible to push or pop from both sides, which can come in handy if you need to test the value on one side and possibly write it back to the same location:



Figure 11.3 – SystemVerilog queue structure

This becomes useful in a testbench by storing what you expect to see. To do this for the `ps2` testbench, I added the following to the `send_key` task. Since we know what we are sending into the PS/2 interface, we can store the expected output:

```
task send_key;
    input [7:0] keycode;
    input      error;
    ps2_rx_data_t local_data;
begin
    local_data.data = keycode;
    local_data.parity = error;
    ps2_rx_data.push_front(local_data);
end
```

When the `send_key` task is called, we build the structure that represents the expected data and push it into the queue. I've replaced the checking function with the queue:

```
while (~done) begin
    while (!rx_valid) @(posedge clk);
    popped_data = ps2_rx_data.pop_back();
    exp_data    = popped_data.data;
```

```
exp_user      = popped_data.parity;
if ((exp_data != rx_data) ||
    (exp_user != rx_user)) begin
    $display("mismatch on output %d", valid_count);
    $stop;
end else begin
    $display("output matched %d", valid_count);
end
valid_count++;
@(posedge clk);
if (valid_count == 16) done = '1;
end
```

Whenever there is an rx_valid signal, the queue is popped and the data from the queue is compared to the data from the design. This is a good way of identifying errors in a design. In this testbench, we are checking a certain number of expected outputs. In other operations you may use the size function to determine if there is data available:

```
if (popped_data.size() != 0)
```

Next, let's take a look at some improvements to the display system function.

Display enhancements

We've used \$display in our simulations. This system task is originally from SystemVerilog and it supported a way of displaying the basic types:

- %h, %H – Hexadecimal value
- %d, %D – Decimal value
- %b, %B – Binary value
- %m, %M – Hierarchical name
- %s, %S – String
- %t, %T – Time
- %f, %F – Real number in decimal format
- %e, %E – Real number in exponential format

If you use these as is, the display will pad data to fit the output. Here's an example:

```
int a, b;
$display("a=%h b=%h", a, b);
// example output
A=00000001 b=0000FFFF
```

SystemVerilog offers a few enhancements. You can use %0h to completely remove leading 0s or % (number) h to limit the output to a certain number of digits. Also, %x can be used in the place of %h:

```
int a, b;
$display("a=%0x b=%4x", a, b);
// example output
A=1 b=FFFF
```

%p allows you to print structures in a formatted fashion. I've modified `tb_ps2.sv` to use %p to print passing values:

```
$display("output matched %d: %p", valid_count, popped_data);
output matched      12: '{data:85,parity:1'b0}
output matched      13: '{data:170,parity:1'b0}
output matched      14: '{data:85,parity:1'b1}
output matched      15: '{data:170,parity:1'b1}
```

You can also display the name of an enumerated type by using .name:

```
enum bit {TRUE = 1'b1, FALSE = 1'b0} my_bool;
$display("The state of my_bool is %s", mybool.name);
```

SystemVerilog also adds \$sformats, which is like \$display, but it returns a string that you can pass into \$display or a log.

I want to at least introduce you to assertions in SystemVerilog at a very high level. Assertions could take up a book all to themselves as verification is a topic unto itself.

A quick introduction to assertions

Assertions are a way of adding self-checking into your code. Assertions are generally ignored by synthesis, and they can be stored in separate files and bound to design modules. I won't go into assertions in depth here, but I encourage you to look into them via the link in the *Further reading* section. I would, however, like to introduce a few other additions for displaying information. They behave like `$display`, but they have severity levels attached to them. These are as follows:

- `$info`
- `$warning`
- `$error`
- `$fatal`

These severity levels allow messages in a simulation to be filtered more easily. For example, you may want to mask `$info` messages or even `$warning` messages during long runs when the design is being regressed. There is another interesting use for `$error` or `$fatal`.

Using `$error` or `$fatal` in synthesis

Often, you may have a reusable module that will only work with certain combinations of parameters. You can use `$error` or `$fatal` to test for these conditions and cause synthesis to abort if they occur. When using these tasks in this way, the evaluation needs to be static, that is, not dynamically changing, but something like testing a parameter setting. For example, if we look back at *Chapter 3, Counting Button Presses*, and our seven segment encoder, we might want to limit it to four or eight segments:

```
module seven_segment #(parameter NUM_SEGMENTS = 8, ...
initial begin
    if (NUM_SEGMENTS != 4 || NUM_SEGMENTS != 8)
        $fatal("Number of segments must be set to 4 or 8");
end
```

In the preceding code snippet, if the number of segments is not 4 or 8, Vivado will fail the synthesis.

Finally, let's take a look at some gotchas and things to watch out for.

Other gotchas and how to avoid them

As we near the end of our journey, there are a few more things that we should look at, along with how we can detect them or avoid them all together.

Inferring single bit wires

From the advent of Verilog, it has always been legal to use a wire without defining it. This can happen if it is a port on an instantiate module. There is an example project: https://github.com/PacktPublishing/Learn-FPGA-Programming/blob/master/CH11/build/inferred_wire/inferred_wire.xpr.

You can see that I've created a variable-width adder module and connected three of them up:

```
adder #(4) u_add0 (.in0(SW[3:0]), .in1(SW[7:4]),
                    .out(add0_out));
adder #(4) u_add1 (.in0(SW[11:8]), .in1(SW[15:12]),
                    .out(add1_out));
adder #(5) u_add2 (.in0(add0_out), .in1(add1_out),
                    .out(LED[5:0]));
```

There is no testbench, but if you try to simulate you will get the following warnings:

```
WARNING: [VRFC 10-3091] actual bit length 1 differs from formal
bit length 5 for port 'out' [/home/fbruno/git/books/Learn-FPGA-
Programming/CH11/hdl/inferred_wire.sv:9]
WARNING: [VRFC 10-3091] actual bit length 1 differs from formal
bit length 5 for port 'out' [/home/fbruno/git/books/Learn-FPGA-
Programming/CH11/hdl/inferred_wire.sv:10]
WARNING: [VRFC 10-3091] actual bit length 1 differs from formal
bit length 5 for port 'in0' [/home/fbruno/git/books/Learn-FPGA-
Programming/CH11/hdl/inferred_wire.sv:12]
```

You can see that single bit wires were inferred. To avoid these problems, uncomment the first and last lines of the file. `default_netttype allows us to define what happens with inferred wires. By specifying none, we tell the synthesis and simulation it's an error if we don't define the netttype of a signal. Now if we run it, you should see this:

```
ERROR: [VRFC 10-2989] 'add0_out' is not declared [/home/fbruno/
git/books/Learn-FPGA-Programming/CH11/hdl/inferred_wire.sv:10]
ERROR: [VRFC 10-2989] 'add1_out' is not declared [/home/fbruno/
git/books/Learn-FPGA-Programming/CH11/hdl/inferred_wire.sv:11]
```

```
ERROR: [VRFC 10-2989] 'add0_out' is not declared [/home/fbruno/git/books/Learn-FPGA-Programming/CH11/hdl/inferred_wire.sv:13]
```

It is best practice to always use `default_nettype` set to `none` at the top of a module and `default_nettype` set to `wire` at the end. The latter is useful in the case of legacy IP that may have inferred wires that you don't have the ability to change. Bit width problems should not be overlooked.

Bit width mismatches

Since it's only a warning, they can easily be overlooked; however, as the design is being developed you should take care to watch for these issues.

We've talked about latches previously in *Chapter 2, Combinational Logic*, but how can we make sure to avoid them?

Upgrading or downgrading Vivado messages

Vivado will display many messages during the design flow. As we discussed previously, latches should be considered an error if they are inferred.

I've created a project to illustrate this: https://github.com/PacktPublishing/Learn-FPGA-Programming/blob/master/CH11/build/latch_error/latch_error.xpr.

Without `tcl.pre` in the project that I set up as specified in *Figure 11.4*, we would encounter the following warning. However, the design would generate a bitstream. If this is missed on something critical such as a state machine, the design is destined to fail at some point:

```
WARNING: [Synth 8-327] inferring latch for variable 'LED_reg'  
[/home/fbruno/git/books/Learn-FPGA-Programming/CH11/hdl/latch_error.sv:9]
```

We can change the severity of any message in the flow by creating a `tcl` file, which is read in prior to synthesis:

```
set_msg_config -id {[Synth 8-327]} -new_severity ERROR
```

We will then specify to use this file prior to synthesis by changing an option in the synthesis settings:



Figure 11.4 – Setting up tcl.pre

Now, if you try to generate a bitstream, you will encounter the following:

```
ERROR: [Synth 8-327] inferring latch for variable 'LED_reg' [/home/fbruno/git/books/Learn-FPGA-Programming/CH11/hdl/latch_error.sv:9]
```

This same methodology can be used to promote or demote any message.

Finally, let's look at timing closure.

Handling timing closure

One of the biggest problems you will run into as a new design engineer is meeting the timing requirements in your designs. There are multiple sets of problems that you will encounter, as we have seen throughout the book. The first type of problem is missing a clock domain crossing. I've removed one of the constraints from the final project to demonstrate this failure:



Figure 11.5 – Clock domain crossing problem

When you encounter a timing problem with inter-clock path violations, address them first. The tools will often not continue to optimize paths once timing cannot be met, so the intra-clock paths may be a false alarm at this point. Let's investigate the paths:



Figure 11.6 – Timing violation report

Looking at the report we notice a couple of things. The first is the clock domains. We know we are crossing between two domains. The second is the requirement. When you see a requirement of 0, or sometimes a very small fraction of the clock period, you should realize that it's a clock domain problem. In this case, we already properly synchronize; I just removed the constraint.

The second type of failure is simply not having enough time in a clock period to do the operation as designed. This can be caused by any of the following:

- **Placement:** You can use pblock constraints to try to guide placement. This is hit or miss and is beyond the scope of this book.

The core of the design is the multiplier. I've defined a PIPELINE parameter to enable pipelining of the multiplier in the event we cannot make timing. The way PIPELINE works is to add register stages after the multiplier, as we can see in *Figure 11.7*:



Figure 11.7 – Pipeline before retiming

We are trying to do this in one clock cycle. I've set the clock frequency to 200 MHz to challenge the tool. This results in a timing failure:

Timing		Setup Hold Pulse Width
Worst Negative Slack (WNS):	-2.454 ns	
Total Negative Slack (TNS):	-257.8 ns	
Number of Failing Endpoints:	125	
Total Number of Endpoints:	535	
Implemented Timing Report		

Figure 11.8 – 32x64-bit multiplier timing failure

As a first step, we can try adjusting the synthesis and implementation settings:



Figure 11.9 – Synthesis options

We also want to enable opt_design and phys_opt_design, as well as play with some of the options, in this case enabling **Explore**, as seen in *Figure 11.10*:

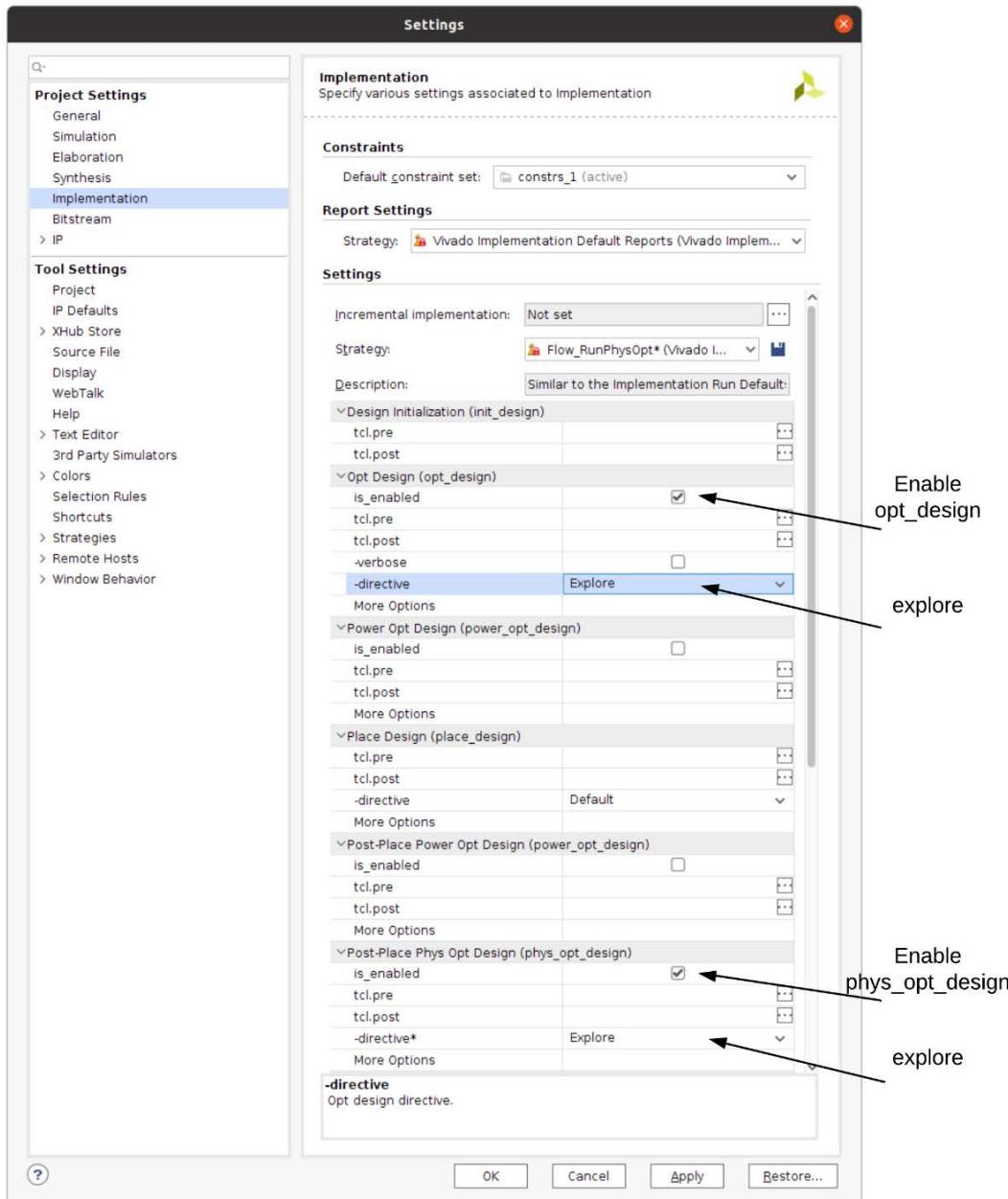


Figure 11.10 – Implementation options

With these implementation options set, we can try another attempt at meeting timing. We are now met with a slightly smaller timing violation:

Timing		Setup Hold Pulse Width
Worst Negative Slack (WNS):	-2.376 ns	
Total Negative Slack (TNS):	-250.84 ns	
Number of Failing Endpoints:	125	
Total Number of Endpoints:	535	
Implemented Timing Report		

Figure 11.11 – Advanced settings timing report

This is obviously not enough to significantly change the results, but it did buy us a little time. Now we can try to add some pipelining. I've made the design such that we can insert pipelining after the math operation. Retiming can move these registers into the design to break up long timing paths. This can be done on logic, DSP elements, and BRAMs. Let's initially try setting **PIPELINE = 1** in the **General** settings tab:

Timing		Setup Hold Pulse Width
Worst Negative Slack (WNS):	-1.139 ns	
Total Negative Slack (TNS):	-101.029 ns	
Number of Failing Endpoints:	97	
Total Number of Endpoints:	631	
Implemented Timing Report		

Figure 11.12 – PIPELINE=1 timing report

This setting actually uses the inserted registers to break up some of the external timing paths, leaving the DSP block intact. You can see this by looking at the schematic in Vivado and searching for the result registers.

Finally, if we set **PIPELINE = 2** in the general settings and rerun, we'll see that retiming will push our design over the edge into positive slack:

Timing		Setup Hold Pulse Width
Worst Negative Slack (WNS):	0.444 ns	
Total Negative Slack (TNS):	0 ns	
Number of Failing Endpoints:	0	
Total Number of Endpoints:	617	
Implemented Timing Report		

Figure 11.13 – PIPELINE=2 timing report

We gain this positive slack by the retiming engine pushing one set of registers into the multiplier. Conceptually, this would look like *Figure 11.14*:



Figure 11.14 – PIPELINE=2 Conceptual

This section should give you some ideas about addressing timing problems. Sometimes, you do need to go a little deeper and rewrite the code. For example, the 32×64 multiplier could be broken into two 32×32 multipliers plus an adder, which I'll leave as an exercise for you.

Hopefully, this chapter has provided some additional resources and things to watch out for as you design your own projects.

Summary

In this chapter, we've looked at some more advanced and lesser used SystemVerilog constructs. The main one is interfaces, which allow better design reuse and encapsulation. We've investigated some more advanced looping, structures, and labels.

We've also looked at some more advanced verification constructs. These will help you as your designs grow and get more complex.

Finally, we looked at some gotchas, how to avoid them, and some basics of timing closure.

You've now completed the book and should be able to tackle some tasks on your own. As I mentioned at the beginning, there are many community efforts, such as the Mister Project, that could use some people with FPGA knowledge. There are also projects you can try to tackle on your own to land a job. Whatever you choose, I hope that you find it as fun and rewarding as I do.

Questions

1. Interfaces are useful for:
 - a) Encapsulating signals belonging together
 - b) Encapsulating functions, tasks, and assertions associated with the interface
 - c) Changing a design deeply embedded within other designs
 - d) *All of the above*
2. Structures can be assigned by:
 - a) Component
 - b) Name
 - c) Interface
 - d) (a) and (b)
3. Block labels allow easier matching of begin...end blocks.
 - a) True
 - b) False
4. If we want to exit a loop, we can use:
 - a) break on any loop
 - b) disable on any loop label
 - c) break on an outer loop or disable on any loop label
5. Continue can be used to skip the rest of a loop.
 - a) True
 - b) False
6. Queues are useful for:
 - a) Creating a flexible FIFO for use in verification
 - b) Creating a flexible FIFO for use in design and verification
 - c) Nothing

7. The following code snippet does what?

```
initial begin
    if (NUM_SEGMENTS != 4 || NUM_SEGMENTS != 8)
        $fatal("This design only supports 4 or 8 segments");
    end
end
```

- a) Causes a fatal error in simulation that the design can only support 4 or 8 segments
b) Causes a fatal error in synthesis that the design can only support 4 or 8 segments
c) All of the above
8. Things to watch out for in designs are:
- a) Accidentally inferring single bit wires
b) Mismatched bit widths
c) Latch inference
d) Clock domain crossing issues
e) All of the above
9. We have seen that we can use retiming to implement a 32x64 multiplier. In that section, I mentioned that you can implement this as two 32x32 multipliers and an adder. Write the SystemVerilog to implement this. Bonus: Does your implementation need to change if you implement signed multiplication?

Further reading

For more information about what was covered in the chapter, please refer to the following:

- <http://staging.doulos.com/knowhow/systemverilog/systemverilog-tutorials/systemverilog-assertions-tutorial/>



Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

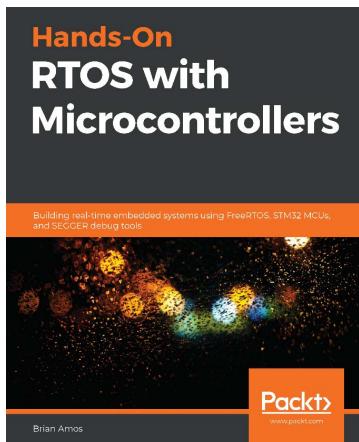


Practical Python Programming for IoT

Gary Smart

ISBN: 978-1-83898-246-1

- Understand electronic interfacing with Raspberry Pi from scratch
- Gain knowledge of building sensor and actuator electronic circuits
- Structure your code in Python using Async IO, pub/sub models, and more
- Automate real-world IoT projects using sensor and actuator integration
- Integrate electronics with ThingSpeak and IFTTT to enable automation
- Build and use RESTful APIs, WebSockets, and MQTT with sensors and actuators
- Set up a Raspberry Pi and Python development environment for IoT projects



Hands-On RTOS with Microcontrollers

Brian Amos

ISBN: 978-1-83882-673-4

- Understand when to use an RTOS for a project
- Explore RTOS concepts such as tasks, mutexes, semaphores, and queues
- Discover different microcontroller units (MCUs) and choose the best one for your project
- Evaluate and select the best IDE and middleware stack for your project
- Use professional-grade tools for analyzing and debugging your application
- Get FreeRTOS-based applications up and running on an STM32 board

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

Symbols

\$error
 using, in synthesis 326
\$fatal
 using, in synthesis 326

A

adder/subtractor
 implementing 74
add module 75
ADT7420 temperature sensor
 data, handling 163, 164
 data, smoothing 164, 165
 using 161-163
Advanced Silicon Modular Block
 (ASMBL) architecture 18
advanced verification constructs 322
arrays
 creating 51, 52
 querying 52, 53
 values, assigning to 53
artificial intelligence (AI)
 massive parallelism 195

Artix 7
 registers 90
ASCII codes 276
ASIC
 about 5
 need for 5-7
ASIC process
 used, for creating programmable
 device 8
assertions 326
assignment operators
 handling 59
assign statement 8
audio data
 capturing 158-161
 displaying 307-311
 storage 152
AXI4 interfaces 222-224
AXI Lite state machine 268
AXI streaming
 with optional tuser signal 204
AXI streaming interface 187

AXI streaming interfaces, used for creating IPs for Vivado
about 205
ADT7420 IP, developing 212
flt_temp core 212
IP integrator 213-221
seven-segment display streaming interface 205-211

B

basic register 85
Basys 3
about 22, 23
features 22
Basys 3 Artix-7 FPGA trainer board
reference link 4
Binary Coded Decimal (BCD) 94, 103
bits
adding, to signal 55
bits per pixel (bpp) 263
bit width mismatches 328
bitwise AND (&) 10
bitwise OR (||) 10
Block Design (BD) 211, 317
blocking assignments
using 59
block labels 318, 319
Block RAM (BRAM) 16, 234
built-in data types, SystemVerilog
bit 51
logic 50

C

calculator
building 123
divider, investigating 132

packaging, for reuse 124, 125
top-level calculator module,
coding 126-128
case statement
about 62, 63
used, for implementing leading ones detector 69
clock
FF, resetting 91, 92
Clock Domain Crossing (CDC) 157
clock domains
data, passing across 111, 112
clock enabled FF 91
clocking 17
Clock Management Tiles (CMTs) 128
code
packaging up, with functions 57, 58
combinational logic
adder/subtractor, implementing 74
assignment operators, handling 59
case statement 62, 63
case statement, used for implementing leading ones detector 69
conditional operator, used
for selecting data 64
creating 58, 59, 65, 66
decisions, making 60, 61
for loop, used for designing
leading ones detector 73
implementing 78, 79
number of ones, counting 74
testbench 66
Combinational Logic Block (CLB)
(CLB) registers 90
Combination Logic Block (CLB) 14, 15
comments 8
complex operations 12

components
interfacing, with interface
construct 314-316
concatenation function 55
conditional operator
used, for selecting data 64
constants
using 322
constrained randomization 69
constraint 169
continue statement
used, for skipping code 321
control signals
synchronizing 109, 110
count button presses project
about 93
asynchronous issues, viewing 97
asynchronous signal, using 98, 99
decimal representation 103
push buttons, issues 100, 101
safe implementation, designing 101, 102
timing, analyzing 96, 97
custom character
adding, to text ROM 306, 307
custom data types
using 64

D

data
selecting, with conditional operator 64
data types 50
DDR2 236
decisions
making 60, 61
design
clocking 84, 85
pipelining 331-336

D Flip Flops (DFFs) 85
Digital Signal Processing (DSP) 17
digit point (DP) 94
disable statement
used, for exiting loop 321
display system function
improvements 324, 325
divider
intermediate remainder, sizing 138
investigating 132
simulating 137
do...while
using, for looping 320
dynamic memory
banks 236
columns 236
rows 236
Dynamic RAMs (DRAMs) 234

E

enum types 117
Error Correction Code (ECC) 16
evaluation boards 19
example design
modifying, for use on board 252-256
example, Vivado
broad, programming 41
design, loading 30-37
implementation 38-40
running 29
external memory
about 234
disadvantage 236

F

FF generation

 always @(), using 87, 88

Field Programmable Gate Array (FPGA)

 about 13, 91

 need for 5-7

First in First Out (FIFO)

 about 9, 165

 addressing 167

 asynchronous FIFO 166

 generating 169-171

 synchronous FIFO 166

 with gray coding 167, 168

first word fall through (fwft) 280

fixed-point arithmetic

 used, for cleaning up bring-up
 time 178-181

 using, for temperature
 conversion 181-183

 using, in temperature sensor project 178

fixed-point numbers

 about 176, 177

 advantages 177

flip flop

 creating 86, 87

floating-point numbers

 about 184

 addition and subtraction 185

 multiplication 185

 operation library 186

 reciprocal 185

floating-point operator, GPLGPU project

 reference link 185

for loop

 used, for designing reusable

 leading ones detector 73

 using, for looping 319

functions

 used, for packaging up code 57, 58

G

gray coding

 using 167, 168

H

hardware 4

Hardware Design Language (HDL) 59, 88

High-Bandwidth Memory (HBM) 236

horizontal synchronization signal 263

HyperRAM 257

I

if statements

 about 9

 qualifying, with unique or priority 62

if-then-else 60, 61

import keywords 316

Input/Output (I/O) buffers 54

Institute of Electrical and Electronics

 Engineers (IEEE) 184

integer division 185

Integrated Logic Analyzer (ILA)

 about 104, 221, 250

 device, programming 106-108

 signals, marking for debugging 104-106

interface 314

interface construct

 used, for interfacing

 components 314-316

I/Os 17

IP catalog
using, to create memory 158

IP integrator
about 213
debugging 221
unpacked IP, adding to 228-230

IPs
developing 225-227

K

keyboard handling 294-297
keyboard interface
investigating 288-294

L

latch
about 59
adding 79

latency 196

leading ones detector
implementation, controlling
with generate 70-72
implementing, with case statement 69

least significant bit (LSB) 289

logical AND (&&) 10

logical NOT (!) 9

logical OR (||) 10

logic gates 8

look up table memories (LUTRAMs) 234

Look Up Tables (LUTs) 90

loop
disabling, with disable 321

Low Pin Count (LPC) 257

M

machine learning (ML)
massive parallelism 195

maximum frequency (fmax) 234

Mealy state machine
implementing 122, 123

memories
instantiating, with xpm_memory 157

memory
creating, with IP catalog 158
requesting 277-282

memory types
about 257
HyperRAM 257
Quad Data Rate (QDR) SRAM 257
SPI RAM 257

microphone
simulating 150-152
using, in Nexys A7 board 146

microprocessors 175

Mixed Mode Clock Manager
(MMCM) 84, 128, 252, 269, 273

modports 315

Moore state machine
designing 120-122

most significant bit (MSB) 289

multiple assignments
creating, with non-blocking
assignments 59

multiple clocks
need for 109

multiple-driven nets
handling 54

multiplier module 76, 77

multiply accumulate (MAC) 17

N

Newton-Raphson 185
Nexys A7
 about 4
 reference link 4
Nexys A7 100T (or 50T)
 about 20
 features 20
Nexys A7 board
 microphone, using 146
non-blocking assignments
 used, for creating multiple
 assignments 59
 using 88, 89
Non-Recurring Engineering (NRE) 5
non-restoring divider state machine
 building 132-136
number of ones
 counting 74

O

online tone generator
 reference link 151
Out Of Context (OOC) synthesis 220

P

package 124
parallel design implementation
 example 196, 197
parameters 49
PDM microphone
 defining 146
 interfacing with 147-150
PDM waveform
 example 147

Phase Locked Loop (PLL) 84, 128, 269
pipelined floating-point implementation
 about 188
 fix to floating point conversion 189, 190
 floating-point math operations 191, 192
 float to fixed point conversion 193
 simulation 194, 195
Printed Circuit Board (PCB) 248
priority 63
programmable device
 creating, with ASIC process 8
Programmable ROMs (PROMs) 13
PS/2 keyboard state machine
 testing 298-301
PS/2 keycodes
 displaying, on VGA screen 302-304
Psuedo Random Binary
 Sequence (PRBS) 257
Pulse Density Modulation (PDM) 146
Pulse Width Modulation (PWM) 140, 160

Q

Quad Data Rate (QDR) SRAM 257

R

RAM
 creating 152
RAM types
 about 154
 simple dual port RAM 155, 156
 single port RAM 154, 155
 true dual port RAM 156, 157
randomized testing 69
Read Only Memory (ROM) 13
registers, Artix 7 90

reusable code
 creating 49
reusable leading ones detector
 designing, with for loop 73

S

SELECTOR = DOWN_FOR
 setting 73, 74
SELECTOR = UP_FOR
 setting 74
sequential element 84
Serial-Deserial (SERDES) 14
Serial Peripheral Interface (SPI) 257
seven-segment display 93-96
signals
 accessing, values with enumerated
 types used 57
 bits, adding to 55
 incrementing 60
signed numbers
 casting 56
 handling 55
simple dual port RAM 155, 156
simulation 108
single bit wires
 inferring 327, 328
single port RAM 154, 155
software 4
software-defined radio (SDR) 195
SPI RAM 257
state machine
 calculator interface, designing 119, 120
 code, writing 116, 117
 combination and sequential
 logic, splitting 118, 119
 implementing 116
state machine design 123

Static RAM (SRAM) 236
storage 16
structures
 creating 64
 using 317, 318
subtracter module 76
Super Logic Region (SLR) 97
synchronization 109
System on Chip (SOC) 221
SystemVerilog
 about 4
 built-in data types 50, 51
SystemVerilog constructs
 about 314
 block labels 318, 319
 code, skipping with continue
 statement 321
 components, interfacing with
 interface construct 314-317
 constants, using 322
 loop, exiting with disable statement 321
 looping, with do...while loops 320
 looping, with for loops 319
 structures, using 317, 318
SystemVerilog modules
 creating 48
SystemVerilog queues 322-324

T

targeted testing 69
temperature conversion
 with fixed-point arithmetic 181-183
temperature sensor data
 displaying 305, 306

temperature sensor project
fixed-point arithmetic, using 178
updating, to pipelined floating-point implementation 188

testbench
parameters 66-68

text ROM
custom character, adding to 306, 307

timing
generating, for VGA 269-273

timing closure
handling 330

timing generator
monitoring 273, 274

toggle Flip Flops (FF) 85

top-level calculator module
coding 126-128
frequencies, modifying with PLL/MMCM 128-132

Total Hold Slack (THS) 102

Total Negative Slack (TNS) 102

traffic light controller
ground rules 139
intersection 138
state diagram, defining 139

traffic light controller, LEDs
delays, implementing with counter 141
displaying 139
displaying, with PWM 140

true dual port RAM 156, 157

tuser 204

two-stage synchronizer 109

U

UG479 7 Series DSP48E1 user guide
reference link 18

unions
creating 65

unique 63

unpackaged IP
adding, to IP integrator 228-230

unsigned numbers
casting 56
handling 55

user-defined types
creating 56

V

values
comparing 61

values, with enumerated types
used, for accessing signals 57

vertical synchronization signal 263

VGA
about 262
AXI lite interface, coding 267, 268
constraints, examining 283, 284
registers, defining 266, 267
text, displaying 275-277
timing, generating for 269-273

VGA controller
testing 282

VGA screen
PS/2 keycodes, displaying on 302-304

Video Electronics Standards Association (VESA) standards 264, 265

video timing 265

Virtual I/O (VIO) interface 255

Vivado

about 24

directory structure 25-29

installing 24

Vivado messages

downgrading 328, 329

upgrading 328, 329

Vivado toolset 19

W

wildcard equality operators

comparing 61

Worst Hold Slack (WHS) 102

Worst Negative Slack (WNS) 102

X

Xilinx Artix-7 series devices 14

Xilinx IP 187

Xilinx MIG, used for generating

DDR2 controller

about 237-241

AXI parameters, setting 242

FPGA options, defining 244-251

memory options, setting 243

Xilinx Parameterized Macro (XPM) 157

XOR (^) 11

xpm_memory

used, for instantiating memories 157

