FPGAs with VHDL: *first steps*

the flow

Helen DeBlumont

FPGAs with VHDL: first steps

Version: 2.2, 14 February 2021

I offer this book free-of-charge. It is my way of giving back to a community full of teachers who have given to me free-of-charge. I did my best when writing this book. However, with the world being what it is, I've been advised to say…. *I make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained in this book*.

Send comments and suggestions to *hdeblumont@gmail.com*.

Revisions:

v2.1 (15Jul19): first release

v2.2 (19Dec20):

- Improved Figure 16.5
- Removed unconventional use of sensitivity list for a VHDL process

Table of Contents

Acknowledgements

Thanks to my many teachers, most of whom participated in the Xilinx Community Forum, especially Avrum Warshawsky and Austin Lesea. Thanks also to Xilinx Inc whose cutting-edge FPGAs and awesome Vivado Design Suite help me “Too Live!” every day at work.

1. Coming and Going

A Field Programmable Gate Array (FPGA) is an expensive integrated circuit that contains reconfigurable high-speed digital hardware and is placing digital hardware engineers on the endangered list. This amazing device resulted from work done in the 1980's for the US military. The companies called Altera and Xilinx made FPGA technology available to the rest of us. That's all the background information you'll get from me because, a) we've got work to do and, b) you know how to search the internet for more information.

In short, when you work with FPGAs, you will write software that tells the FPGA how to configure its internal hardware. Once configured, the FPGA runs at the speed of hardware, which is usually very much faster than the speed of software (ie. the speed of a digital computer).

So, where are you coming from? Previous experience with writing software or with designing digital hardware/circuits is a plus for those starting FPGA work. Software people seem to startup more quickly with FPGA work than the digital-hardware people. This may seem odd to you because an FPGA is reconfigurable hardware. However, after startup, a "think hardware" approach will be needed to understand the more challenging parts of FPGA work.

So, where are we going? I wrote this book for someone starting to work with FPGAs. FPGA work is rich in variety and detail, keeping some busy and interested during their entire career. However, this richness can be overwhelming to the beginner. So, with thoughts of my beginner's experience still fresh in my head, I have tried to write something that makes a beginner's experience a little more orderly and enjoyable. This book talks about choices that you must make at the start of your FPGA adventure and about lasting concepts that do not depend much upon where you buy your FPGAs. *Warning!* I have my favorite FPGA vendor, which I'll discuss in Chapter 2.

2. Startup Choices

2.1 Choose a Language

Your first really big choice is to select a Hardware Description Language (HDL). You will use HDL to tell the FPGA how to configure its internal hardware. Henceforth in this book, we will refer to "telling the FPGA how to configure its internal hardware" as simply *FPGA configuration*. There are two popular HDLs called Verilog and VHDL. Verilog is popular in the United States and VHDL is popular elsewhere. So, select an HDL, find a good book about it, and start learning. Ok, some of you have experience in other programming languages (eg. C) and have heard about converters that translate these other languages into HDL. Don't go there - yet! These converters are useful, but only after you have mastered an HDL.

2.2 Choose a Toolset

It won't be long before you'll want to write HDL and test whether it works. For this, you need to make a second really big choice by selecting an FPGA toolset. An FPGA toolset is a software package that is also called an Integrated Development Environment (IDE). Each FPGA vendor has its own IDE, which must be used when working with their FPGAs. Most vendors offer a free version of their IDE! Often, this free version is very nearly as good as the version that they sell. Your life will be easier if you stay with one FPGA vendor and use only their IDE. However, once you learn the beginner concepts of FPGA work, you will be able to work with FPGAs and IDEs from other vendors without too much trouble. -more about this in Chapter 3.

2.3 Choose a Teacher

Hopefully this book will fill part of your need for an FPGA teacher. However, you should seriously consider taking classes offered by the vendor of the FPGAs that you plan to use. Also, consider purchasing technical support from the FPGA vendor for a year or so. You will welcome the courteous and quick response to your questions that purchased

technical support provides. Vendors often sponsor online forums where you can post questions. The answer-quality that you get from a forum is varied, but I've generally found forums to be helpful and enjoyable. Finally, for the low-budget beginner, you will find lots of free literature on the FPGA vendor's website. It's easy to become overwhelmed by this literature. If you take the "free literature" approach to learning, then start by searching the vendor website using words like "tutorial" or "getting started". These searches will bring up documents and short videos that are appropriate for the beginner.

2.4 Choose an Interface

If you're a Microsoft Windows user, then you will probably want to work with the Graphical User Interface (GUI) of the IDE that we discussed in section 2.2. After all, we're all experts at stumbling around a Windows-like GUI to get what we want (just ask my neighbor's daughter who, at 6 years of age, ordered stuff on Amazon totaling hundreds of dollars). However, if you're a Linux/Unix user, then you might prefer the command-line interface of the IDE, which uses a language called Tcl. Whether you decide to use the GUI or command-line interface for the IDE, you will need to learn the Tcl language. Later, we'll talk more about Tcl when we talk about writing FPGA constraints.

2.5 My Choices

I was forced to choose VHDL because my first FPGA job was to maintain lots of previously written VHDL. So, whenever our discussion of FPGA concepts requires an HDL example, I will use VHDL. However, there's lots to talk about that is not HDL specific. Also, I am partial to using the Xilinx IDE called Vivado. However, as explained in Chapter 3, there is much conceptual similarity between the IDEs from different FPGA vendors.

2.6 The Tally

Let's tally the count (so far) of software needed for FPGA work. There's HDL, the IDE, and Tcl for a total of three! So, you can see why I said in Chapter 1 that software people have an advantage over digital-hardware people when starting FPGA work. Also, there's lots of FPGA-specific terminology that will probably be new to you. Throughout this book, I will introduce and explain this terminology. However, as you read this book, it may be helpful to keep on-hand a copy of the glossary found at the end of this book.

3. IDEs – May the Flow Be with You

Flow is nice word, suggesting the peaceful journey made by water in a stream (see photo on the cover of this book). Flow is also a word used by FPGA vendors to mean the recommended sequence for using tools found in their IDE. Here is a typical IDE flow and a sincere wish that your journey with the flow is peaceful.

1) ***Create Project***: Launch the IDE and answer some basic questions about your FPGA project.
2) ***Add HDL***: Use the code editor tool to enter your own kick-ass HDL.
3) ***Add IP Modules***: Intellectual Property (IP) modules are HDL written by the FPGA vendor. Some are free.
4) ***Run Simulation***: Use the simulation tool to test that your HDL is logically correct.
5) ***Add Constraints***: Use Tcl language to write constraints (eg. matchup between HDL-signals and FPGA-pins).
6) ***Run Synthesis***: Create a digital circuit representation for your HDL.
7) ***Run Implementation***: Run timing analysis and place/route the digital circuit from synthesis into the FPGA.
8) ***Check, Correct, Iterate***: Check that design has passed timing analysis. Correct HDL and rerun tools as necessary.
9) ***Generate Bitstream***: Create a file the describes the results of implementation (ie. the FPGA configuration).
10) ***Program the FPGA***: Configure the FPGA hardware by downloading the bitstream into the FPGA.

The organization of this book follows this typical IDE flow.

In Fig 1 is the main window for the Vivado (v2018.3) IDE supplied by the FPGA vendor, Xilinx. This screenshot of the Vivado window was taken just after I finished ***Create Project*** in the IDE-flow list shown above. More specifically, the "Project Summary" tab on the right side of the window tells you that I have named the project, my_proj1, specified that I will use the Xilinx Kintex-7 (part# xc7k160tfbg484-3) FPGA, and that I plan to use the HDL called VHDL.

Also, in the main window for the Vivado IDE, you will see a box along the left side called the "Flow Navigator". Many IDEs will have a similar box along the left-hand side of their main window, which helps you remember the IDE flow. One typically launches tools of the IDE by double-clicking lines in "Flow Navigator", working from top-to-bottom. Of course, things are seldom as simple as writing some HDL and quickly clicking through entries found in the Flow Navigator. There will be problems along the way and hopefully this book will help you solve them.

Finally, for an alternate and expanded description of the IDE flow, go to www.xilinx.com and search for the "UltraFast Design Methodology Checklist". For the Xilinx FPGA user, mastering this extensive checklist is a great goal. Reading the rest of this book may help you reach that goal a little faster.

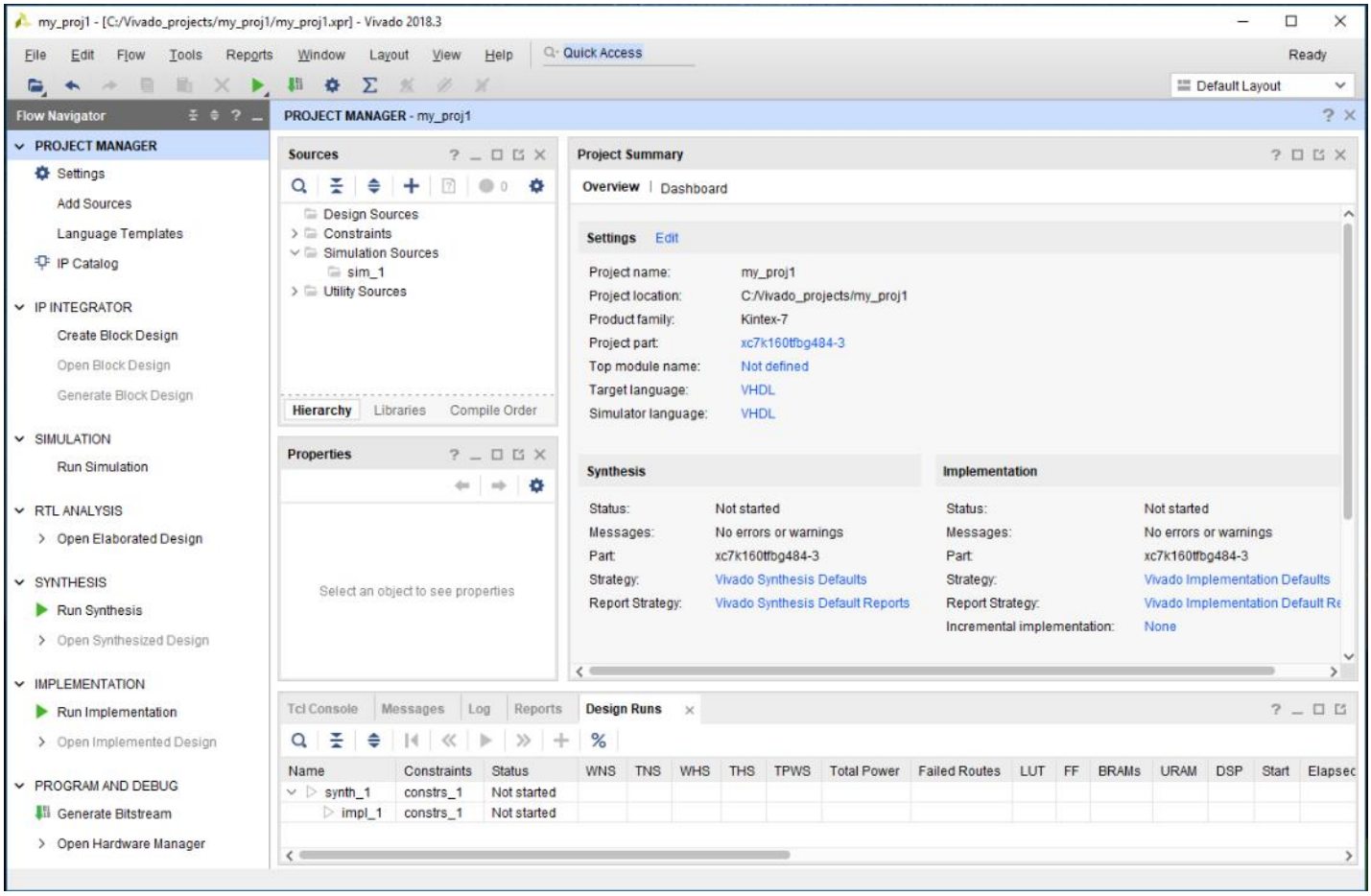

Fig 3.1 Main window of the Xilinx IDE called Vivado.

4. Add VHDL

Continuing to follow the flow described in Chapter 3, we'll now talk about entering HDL into the IDE. My apologies to the Verilog people, since I am now going to focus on VHDL. However, many of the concepts found in the following discussion apply to both VHDL and Verilog.

Warning – you cannot learn VHDL programming from reading this book. There are many other fine books about HDL programming (both VHDL and Verilog). You should grab one of them and become familiar with HDL programming before reading the rest of this book.

I remind you that the purpose of writing HDL is to define FPGA configuration (ie. to describe how the FPGA should configure its internal digital hardware). Since we are talking about high-speed digital hardware then *sequential logic* (aka *clocked logic*) will be necessary together with the more familiar *combinational logic*. Are you feeling a little rusty with digital hardware terminology? No worries, we'll talk more about it in Chapter 7. For now, all you need to know is that sequential/clocked logic requires a clock and combinational logic does not.

4.1 Component

In the old days, logic circuits were built by connecting things called discrete components. Some of you may remember fiddling with the component called a 7400-quad-NAND, which was a 14-pin IC that had four two-input NAND gates. -but I digress. The inventors of VHDL wisely chose the word, component, for the basic building-block of VHDL. You will be pleased to know that, as in the old days, we can conceptually build a logic circuit by connecting together VHDL components. When you tell the IDE that you want to add new source code to your FPGA project, it will provide a window into which you type VHDL code that describes a component.

In Fig 4.1, is a silly example of a VHDL component called TWO_BIT. The code for TWO_BIT starts with nice-to-have comments (lines 01-06). Note that VHDL comment lines begin with the symbol, -- (two consecutive dashes). Lines 07-09 indicate that standard VHDL libraries will be used. The block of code in lines 11-19 is called the *entity*. For the component, the entity names the inputs and outputs and identifies their type. The *std_logic* (1-bit), *std_logic_vector* (multibit), and *unsigned* types are commonly used in VHDL. The *std_logic* type is for, well, standard logic (ie. a signal that takes on the values of 0 or 1). The *unsigned* type is used for numerical data (ie. something used in a math operation). Inputs to TWO_BIT are a clock, CLK (used for sequential logic), and 2-bit data values called IN1 and IN2. Outputs of TWO_BIT are called CTL1 and CTL2. In general, one should use names for component inputs and outputs that are more descriptive than those I have chosen for TWO_BIT. The block of code in lines 21-39 of Fig 4.1 is called the *architecture*. It describes how the inputs to TWO_BIT are manipulated to create values for the outputs from TWO_BIT.

This ends our discussion about the composition of a VHDL component. However, in the rest of this chapter, we will continue to talk about our example component, TWO_BIT, and how it works.

```
01  ----------------------------------------------------------
02  -- Name:        TWO_BIT.vhd
03  -- Description: Two outputs that depend on the sum of two two-bit inputs
04  -- Create Date: 14Feb2021
05  -- Revisions:   none
06  ----------------------------------------------------------
07  library IEEE;
08  use IEEE.STD_LOGIC_1164.ALL;
09  use IEEE.NUMERIC_STD.ALL;
10
11  entity TWO_BIT is
12    port(
13          CLK     : in std_logic;              --logic clock
14          IN1     : in unsigned(1 downto 0);   --input #1
15          IN2     : in unsigned(1 downto 0);   --input #2
16          CTL1    : out std_logic;             --control output #1
17          CTL2    : out std_logic              --control output #2
18         );
19  end TWO_BIT;
20
21  architecture MY_CMP1 of TWO_BIT is
22      signal bCTL1 : std_logic;                --declare a signal called bCTL1
23  begin
24      CTL1 <= bCTL1;
25      CTL2 <= not(bCTL1);
26
27      MY_PRC1: process(CLK)                     --outputs: (bCTL1)
28          variable sum1 : unsigned(1 downto 0); --declare a variable called sum1
29          begin
30              if rising_edge(CLK) then
31                  sum1 := IN1 + IN2;
32                  if(sum1 > "10") then          --is sum1 greater than 2 ?
33                      bCTL1 <= '1';
34                  else
35                      bCTL1 <= '0';
36                  end if;
37              end if;
38      end process MY_PRC1;
39  end MY CMP1;
```

Fig 4.1 VHDL component called TWO_BIT.

4.2 Signal

In VHDL lingo, the word *signal* refers to a digital signal being routed through the digital hardware. That is, when the example component, TWO_BIT, is connected with other components to form a digital circuit, words declared as VHDL signals will be used to make the connections. In VHDL, you create a signal by *declaring* it. For example, a signal called bCTL1 is declared in line-22 of Fig 4.1. Signals are assigned a value using a *signal assignment statement*, that is identified by the *signal assignment operator*, <=. In line-33 and in line-35 we see assignments for the 1-bit signal called bCTL1.

4.3 Process

If you are familiar with the C-language function, then you have a pretty good idea of how a VHDL *process* works. Lines 27-38 of Fig 4.1 show a process called MY_PRC1. Roughly, inputs to the process are identified in the *sensitivity list* of the process. The sensitivity list for MY_PRC1 is the signal name, CLK, within the parenthesis on line-27. Unlike a C-language function, listing the outputs of a VHDL process is not required. Hence, adding a comment to identify the outputs is a good idea. The comment on line-27 indicates that MY_PRC1 has only one output called, bCTL1.

In the previous paragraph, I said that the *sensitivity list* is a list of all inputs to a VHDL process. More specifically, a change to any signal found in the *sensitivity list* will cause execution of the associated VHDL process. Appendix B gives a complete description/definition for the sensitivity list. However, at this point in our learning, the Appendix B description may not make much sense to you.

The second line (line-28 in Fig 4.1) of MY_PRC1 declares a VHDL *variable* called sum1. A VHDL variable has behavior and purpose similar to a variable declared within a C-language function. That is, the VHDL variable *usually* has local-scope

(ie. it has meaning only inside a process) and its purpose is to help create values for the outputs of the process. Variables are assigned a value (eg. line-31) using the *variable assignment operator*, **:=**, instead of **<=**.

Generally, and *with few exceptions (see section 4.4)*, one can interpret the lines of VHDL inside a process as instructions that are executed sequentially. This is the same way that lines of code in a C-language function are executed. Inspection of lines 30-37 shows that the processing done by MY_PRC1 is repeated at every rising-edge of CLK. Further, we see that bCTL1 is assigned the value of 1 if the sum of inputs, IN1 and IN2, is greater than "10"(binary), otherwise bCTL1 is assigned the value of 0. The value of CTL1 is assigned to the value of bCTL1 in line-24. Thus, we now understand how IN1 and IN2 (inputs to both TWO_BIT and to MY_PRC1) are manipulated to create a value for the TWO_BIT output, CTL1. Line-25 shows that the other output, CTL2, of TWO_BIT is simply the digital-NOT of CTL1.

4.4 Signal vs Variable

Here, I'll discuss why I used the words "*with few exceptions*" at the start of the previous paragraph. An important exception to the sequential execution of instructions inside a VHDL process has to do with distinction between VHDL signals and VHDL variables. This distinction is about when signals and variables take-on values assigned to them within a VHDL process.

Variable assignments done inside a process take effect immediately. That is, in line-31 of Fig 4.1, the value of variable, sum1, is set equal to the sum of IN1 and IN2. On the very next line (line-32), this new value for sum1 (ie. the quantity, IN1 + IN2) is available and is compared to "10". In sharp contrast, the signal assignments done inside a process take effect after the process ends. Hence, if we tried to inspect the value of bCTL1 after line-33 has executed, then the value of bCTL1 would come from the previous (and not the current) execution of process, MY_PRC1. That is, the assignment of value 1 to bCTL1 in line-33 takes effect only after process, MY_PRC1, ends.

Variables *usually* exist only within the process where they are declared. So, variable assignment statements cannot be located outside of a VHDL process. However, signal assignment statements can be placed outside of a VHDL process (eg. see line-24 and line-25 in Fig 4.1). Signal assignments made outside of a process occur immediately upon change of any signal found on the right-hand side of the assignment statement.

To help clarify the distinction between signals and variables used within a process, I'll now discuss other simple (and rather silly) examples of a VHDL process. In Fig 4.2, the signal assignment in line-06 is effectively ignored, always! That is, after line-06 is executed, the value of signal, bCTL1, remains at whatever value it had when execution of MY_PRC2 began. It may surprise you that the code in line-07 is also ignored! This is because the signal assignment in line-08 ignores whatever lines 06-07 were trying to do with bCTL1. Instead, line-08 uses the value that bCTL1 had when execution of MY_PRC2 began. So, the output of MY_PRC2 is determined entirely by the code in line-08. What value does input, bCTL1, have when execution of MY_PRC2 starts? Well, it has the value of bCTL1 that was calculated during the prior execution of MY_PRC2.

```
01  signal CLK, bCTL1, IN1, IN2 : std_logic;
02  ……
03      MY_PRC2: process(CLK)                    --outputs: (bCTL1)
04          begin
05              if rising_edge(CLK) then
06                  bCTL1 <= '1';
07                  bCTL1 <= bCTL1 or IN1;
08                  bCTL1 <= bCTL1 and IN2;
09              end if;
10      end process MY_PRC2;
```

Fig 4.2 VHDL process called MY_PRC2.

In Fig 4.3, the process called MY_PRC3 does what process, MY_PRC2, in Fig 4.2 tried to do and failed. That is, the output, bCTL1, of MY_PRC3 will be calculated as, `('1' or IN1) and IN2`. Finally, if lines 06-08 in Fig 4.2 were replaced by the single line of code, `bCTL1 <= ('1' or IN1) and IN2`, then both MY_PRC2 and MY_PRC3 would calculate the same value for bCTL1.

```
01  signal CLK, bCTL1, IN1, IN2 : std_logic;
02  ……
03      MY_PRC3: process(CLK)                    --outputs: (bCTL1)
04          variable my_var : std_logic;
05          begin
06              if rising_edge(CLK) then
07                  my_var := '1';
08                  my_var := my_var or IN1;
09                  bCTL1 <= my_var and IN2;
10              end if;
11      end process MY_PRC3;
```

Fig 4.3 VHDL process called MY_PRC3.

4.5 Concurrency

When you power-up digital hardware, each component comes ON and, in parallel with all other components, does its own individual thing (ie. processing inputs to produce outputs). Since we are using VHDL to describe hardware and to describe the inherent parallel operation of hardware, then we should expect that parts of our VHDL code will execute in parallel. In other words (ie. in VHDL lingo), parts of our VHDL code execute *concurrently*.

The VHDL concept of concurrency is foreign for many programmers since the code of most programming languages executes *sequentially*. Also, managing blocks of code that execute concurrently may seem daunting. Be brave! It's not very hard and we'll talk more about it in Chapter 5. After all, anyone who has built a digital circuit the old-fashioned way by wiring up discrete components has, perhaps unknowingly, managed the concept of concurrency.

We will baby-step our way towards learning about VHDL concurrency by carefully studying the code in Fig 4.1. In the lingo of VHDL, the process called MY_PRC1 is called a *clocked process* because of the way it uses the clock called CLK. Most of the VHDL processes that you write will be clocked processes. In MY_PRC1, and in most clocked process, the rising-edge of CLK launches the following sequence of events:

1) Each *signal* input to a VHDL process takes-on and remains at a fixed value while code within the process executes. The fixed value for each input is the value of the input at the time of the rising-edge for CLK.
2) Code within the process executes sequentially. The fixed values of the input signals and the values of variables are used to produce values for the outputs of the process.
3) After the last line of code in the process is executed, values for the outputs of the process are assigned.
4) Outputs of the clocked process hold their current value and the process waits for the next rising-edge of CLK.

Using this sequence of events, a clocked process produces outputs and makes them available to "other code" within the VHDL component. For component, TWO_BIT, this "other code" consists of the assignment statements shown in lines 24-25 of Fig 4.1. That is, the single output called bOUT1 of the clocked process, MY_PRC1, is used in lines 24-25 to assign values to OUT1 and OUT2 immediately after execution of MY_PRC1 ends.

Most VHDL components are more complicated that MY_PRC1 and have more than one VHDL process. When a component has more than one process, all the processes execute concurrently. Further, it is quite possible that the output of one process could be an input to another process (ie. processes can talk to each other). However, don't let this fact bring up images of endless loops and cause you to panic. Calm yourself and remember that each clocked process takes a snap-shot of its inputs on the rising-edge of CLK. Then, each process calmly does its own thing, oblivious of what the other processes are doing. That is, separate processes talk to each other (exchange signal values) only for the brief instant of time at the rising-edge of CLK.

I'll end our current discussion about VHDL concurrency by saying that separate components also execute concurrently. That is, like VHDL processes, VHDL components pass signal values (ie. data) to each other at the rising-edge of a clock. You may recall my comment (in early sections of this chapter) that FPGA work uses sequential logic (aka clocked logic). Those of you familiar with sequential logic hardware know that data is passed from one part of the circuit to the next at the rising-edge of a clock. So, now it all comes together! That is, we now understand that use of concurrency and a clock in VHDL are the necessary means by which we tell the FPGA to configure its internal hardware into a sequential logic (aka clocked logic) circuit. -more about this in Chapter 7.

Before moving on to the next section, I'll leave you with a few thoughts. It is a fact that sequential execution of the code within a clocked process takes some time. This fact is entirely equivalent to saying that propagation of a digital signal through digital hardware takes some time. So, you might question what happens when the time required to execute the code within a clocked process is longer than the time between rising-edges of the clock. Well, that is a very good question! Being able to deal with that question is what makes FPGA programmers truly special. -more about this in Chapter 7.

4.6 Top-Level Component and Constraints File

We use VHDL to describe components and to describe how these components are connected. The result of our VHDL coding is the circuit that we want the FPGA to create using its internal hardware. This resulting circuit is usually a component itself and, in VHDL lingo, is called the *top-level component*.

In our discussions so far, we have used VHDL to describe only the component called TWO_BIT. So, for now, it is the top-level component for our example project called my_proj1. However, most projects will have many components. In multi-component projects, the IDE is usually smart enough to automatically identify the top-level component. However, the IDE will want you to specify how the inputs and outputs of the top-level component connect to the physical pins of the FPGA. These connections are specified by writing *physical constraints* that are placed in a *constraints file*.

For our silly one-component example project, my_proj1, I have created a constraints file called **constraints1.xdc**. In section 2.4, I mentioned that the Tcl language comes up often when working with IDEs. For most IDE, the specific Tcl commands used to write constraints will follow what is called the Synopsys Design Constraints (SDC) format, created over 20 years ago by a company called Synopsys. Anyway, I have used Tcl/SDC to write some physical constraints for my_proj1 and have placed them in **constraints1.xdc** as shown in Fig 4.4. If you don't already know Tcl, don't get stressed about having to learn yet another programming language. The lines of Tcl code found in a constraints file are often very repetitive, allowing you to cut and paste many times while making small changes to each line. For example, lines 04-09 in Fig 4.4 all look very similar and indicate that FPGA pins, (E13, H13, F13, G13, W20, V20), are connected to the ports, (IN1[0], IN1[1], IN2[0], IN2[1], CTL1, CLT2), respectively on our top-level component, TWO_BIT.

```
01  # Xilinx Vivado constraints file for project, my_proj1
02  # Name: constraints1.xdc
03  # Date: 21Mar2019
04  set_property PACKAGE_PIN E13 [get_ports {IN1[0]}]
05  set_property PACKAGE_PIN H13 [get_ports {IN1[1]}]
06  set_property PACKAGE_PIN F13 [get_ports {IN2[0]}]
07  set_property PACKAGE_PIN G13 [get_ports {IN2[1]}]
08  set_property PACKAGE_PIN W20 [get_ports CTL1]
09  set_property PACKAGE_PIN V20 [get_ports CTL2]
10
11  set_property IOSTANDARD LVTTL [get_ports {IN1[0]}]
12  set_property IOSTANDARD LVTTL [get_ports {IN1[1]}]
13  set_property IOSTANDARD LVTTL [get_ports {IN2[0]}]
14  set_property IOSTANDARD LVTTL [get_ports {IN2[1]}]
15  set_property IOSTANDARD LVTTL [get_ports CTL1]
16  set_property IOSTANDARD LVTTL [get_ports CTL2]
```

Fig 4.4 Part of the constraints file for the Vivado example project called my_proj1.

The pins of an FPGA are often multipurpose, able (for example) to be configured as either inputs or outputs. Our VHDL description of the TWO_BIT component and the constraints file have told the FPGA to configure pins (E13, H13, F13, G13) as inputs and to configure pins (W20,V20) as outputs. The FPGA pins are multipurpose in other ways too; able to duplicate the voltage input/output requirements of different logic families. For example, lines 11-16 in Fig 4.4 indicate that all the FPGA pins associated with the TWO_BIT are to operate as low-voltage TTL (LVTTL) logic.

Throughout the rest of this book, we will be placing many Tcl/SDC commands in the constraints file. For now, just remember that some of these commands match-up input and outputs of your top-level component with physical pins on the FPGA. Finally, you may have noticed that the input, CLK, to component, TWO_BIT, is missing from Fig 4.4. We will talk more about FPGA clocks in Chapters 8 and 9.

4.7 Quirks

Occasionally in this book, you will find a section called Quirks where unusual or unexpected concepts are listed and explained. After reading this Quirks section, you'll see what I mean.

Signal vs Variable (VHDL quirk): I am calling the signal vs variable distinction quirky because: a) failing to remember this distinction is one of the most common mistakes made by VHDL beginners, b) your experience with variables in other programming languages may lead you astray, and c) I want you to reread section 4.4.

Sensitivity list is sometimes optional (VHDL quirk): Some IDE tools will not throw an error if you make a mistake in sensitivity list for a VHDL processes. The error is not thrown because some tools are smart enough to know what signal names belong in the sensitivity list. After reading a few more chapters in this book, you should read Appendix B, which contains more information about the sensitivity list.

Why bCTL1? (VHDL quirk): It is a VHDL rule that you cannot use the name of an output from your component on the right-hand side of an assignment statement. So, for example, in line-25 of Fig 4.1, we cannot write "`CTL2 <= not(CTL1)`" because CTL1 is an output from component, TWO_BIT. As shown in Fig 4.1, a buffer signal, bCTL1, can be used to satisfy this VHDL rule. That is, a value for bCTL1 is determine in process, MY_PRC1, and then bCTL1 is legally used on the right-hand side of assignment statements in line-24 and line-25 to determine outputs, CTL1 and CTL2. When naming these buffer signals, I will often use the name of the component output (that I wanted to use in the assignment statement) preceded by the letter, b.

Signal names (VHDL quirk): Signal names in VHDL are not case-sensitive and in some situations signal names can be the same as names used for the inputs and outputs of a component. However, I find these duplicate names confusing and try never to use them.

Component input/output names (VHDL quirk): Component input/output names in VHDL are not case-sensitive and cannot be duplicated within a component. However, the name of an output from one component can be the same as the name of an input to another component. For example, the output named CTL1 from component, TWO_BIT, could also be used as the name of the input to another component. This type of duplicate naming is not too confusing and is sometimes desirable.

4.8 Key Points

We covered a lot of ground in this chapter. If this chapter was only a little confusing, then perhaps things will be clearer after you read Chapter 6 about the testing of component, TWO_BIT. If this chapter is complete gobbledygook, then you should again read your VHDL programming book. Here are some key points to remember (italicized words have special meaning in VHDL).

- The purpose of writing VHDL code is to define FPGA configuration (ie. to define how the FPGA will configure its internal digital hardware.
- In VHDL, the key building block is called a *component*.

- VHDL *signals* enter a *component* via its *inputs* and are processed by a VHDL *processes* to become other *signals*. Some of these other *signals* are then routed to the *outputs* of the *component*.
- Although a *process* can directly calculate a value for a *signal*, it is sometimes convenient to use a VHDL *variable* for intermediate calculations.
- Exactly when *signals* and *variables* take-on their assigned values is an important concept (see section 4.4).
- Most *processes* are *clocked*, meaning that they execute once after each rising-edge of a digital *clock*.
- All processes operate *concurrently*, meaning that they all execute simultaneously.
- VHDL is also used to describe how *components* connect to each other and form a *top-level component*.
- Some instructions found in the *constraints file* specify which physical pins of the FPGA connect to *inputs* and *outputs* of the *top-level component*.

Finally, did you identify all the digital hardware that was described by the VHDL code for TWO_BIT? I'll bet not. In fact, you are probably thinking that the VHDL in Fig 4.1 looks like ordinary software and not hardware at all. It's OK if that's what you think. Sometimes we need to visualize the hardware that's described by our HDL, but often not. So, you can again see why I said in Chapter 1 that software people have an advantage over digital-hardware people during startup FPGA work.

5. Managing VHDL Concurrency

In section 4.5, we wondered how a programmer manages VHDL concurrency. That is, how does one write VHDL code and manage the fact that lots of separate VHDL processes and lots of separate VHDL components are executing in parallel? The rather surprising answer is that most of the time we don't manage it! The humbling truth is that our brains, and much of the things that our brains create, operate sequentially. So, despite the awesome capability of the FPGA to do parallel processing, our brains tend to break up a problem into somewhat unrelated tasks, each of which can be done sequentially. As a result, many of our VHDL processes and components will often be in a thumb-twiddling mode. That is, they will process the same inputs over and over while waiting for us to send them new inputs or while waiting for us to trigger them into doing something new. However, all this wasted processing matters little if tasks are getting done when you need them to be done.

5.1 Making Breakfast

Perhaps I can clarify things by talking about the first component of my day called MAKE_BREAKFAST. MAKE_BREAKFAST consist of the two processes shown in Fig 5.1 and it is a team effort; I make the coffee and my wife makes the toast. Stumbling down to the kitchen each morning is the MAKE_BREAKFAST trigger that causes each of us to start work on our assigned process. The seven steps needed to complete each process are done sequentially (because multitasking early in the day usually leads to problems). After receiving the beep (ie. the done-flag) from each process, my wife and I have a nice breakfast and then go to work. -and, until the next breakfast, the MAKE_COFFEE and the MAKE_TOAST processes sit idle and wait for the next MAKE_BREAKFAST trigger.

Component: MAKE_BREAKFAST	
Process: MAKE_COFFEE	**Process: MAKE_TOAST**
1. Wait for MAKE_BREAKFAST trigger **2. Load water** **3. Load filter** **4. Place ground coffee into filter** **5. Push button (on coffee machine)** **6. Wait for the beep** **7. Return to step 1**	**1. Wait for MAKE_BREAKFAST trigger** **2. Get loaf of bread from cabinet** **3. Extract one slice from loaf** **4. Place slice into toaster** **5. Push button (on toaster)** **6. Wait for the beep** **7. Return to step 1**

Fig 5.1 The first components of my day, MAKE_BREAKFAST.

Okay, that was a really nerdy way to describe making breakfast. However, making breakfast closely matches the way I do much of my FPGA programming. That is, I create processes that have very little interaction with each other. When I want a process to do something then I send a trigger to it. I then wait, watching for the done-flag from each process. After all the done-flags are asserted, I collect the results from each process and move on to doing other things.

Sure, sometimes my approach to FPGA programming and managing VHDL concurrency leads to things getting done too slowly. When things are done too slowly (rare in my experience), you'll need to get out of your comfort zone and use other methods. One popular speed-up in FPGA work is called pipelining, which we'll discuss in section 11.4. However, these speed-ups are complicated and should only be used when necessary.

5.2 State Machine

In VHDL, there is a programming structure called a *state machine*. Don't let this fancy name scare you – because a state machine mimics the way we naturally think about things. That is, a state machine is a throwback to the way that most software programmers think and write code – sequentially.

Let's learn about the VHDL state machine by looking at the example shown in Fig 5.2, which implements the MAKE_COFFEE process shown in Fig 5.1. For the record, the Fig 5.2 example uses what is called the one-process approach. There are also two-process and three-process approaches to coding state machines, but I find they are seldom used and somewhat confusing.

```
01  -- Name:        MAKE_BREAKFAST.vhd
02  -- Description: Control for making my breakfast.
03  -- Create Date: 14Feb2021
04  ----------------------------------------------------------
05  library IEEE;
06  use IEEE.STD_LOGIC_1164.ALL;
07  entity MAKE_BREAKFAST is
08    port( CLK, RST, MAKE_BREAKFAST_TRIG                              : in std_logic;
09          WATER_DONE, FILTER_DONE, GROUNDS_DONE, BREW_DONE, BEEP_DONE : in std_logic;
10          --other inputs here...
11          WATER_TRIG, FILTER_TRIG, GROUNDS_TRIG, BREW_TRIG            : out std_logic
12          --other outputs here...
13          );
14  end MAKE_BREAKFAST;
15  architecture MY_CMP2 of MAKE_BREAKFAST is
16      type state_var1 is (s1,s2,s3,s4,s5,s6);     --Define an enumerated signal type called state_var1
17      signal coffee_state : state_var1;           --Declare coffee_state as instance of state_var1
18
19  begin
20      MAKE_COFFEE: process(CLK)
21          begin
22              if rising_edge(CLK) then
23                  if (RST = '1') then             --Did a reset occur?
24                      coffee_state <= s1;
25                  else
26                      case coffee_state is
27                          when s1 =>              --s1: wait for "make breakfast" trigger
28                              WATER_TRIG <= '0';
29                              FILTER_TRIG <= '0';
30                              GROUNDS_TRIG <= '0';
31                              BREW_TRIG <= '0';
32                              if(MAKE_BREAKFAST_TRIG = '1') then
33                                  coffee_state <= s2;
34                              else
35                                  coffee_state <= s1;
36                              end if;
37                          when s2 =>              --s2: load water
38                              if(WATER_DONE = '0') then
39                                  WATER_TRIG <= '1';
40                                  coffee_state <= s2;
41                              else
42                                  WATER_TRIG <= '0';
43                                  coffee_state <= s3;
44                              end if;
45                          when s3 =>              --s3: load filter
46                              if(FILTER_DONE = '0') then
47                                  FILTER_TRIG <= '1';
48                                  coffee_state <= s3;
49                              else
50                                  FILTER_TRIG <= '0';
51                                  coffee_state <= s4;
52                              end if;
53                          when s4 =>              --s4: load coffee grounds
54                              if(GROUNDS_DONE = '0') then
55                                  GROUNDS_TRIG <= '1';
56                                  coffee_state <= s4;
57                              else
58                                  GROUNDS_TRIG <= '0';
59                                  coffee_state <= s5;
60                              end if;
61                          when s5 =>              --s5: push brew-button
62                              if(BREW_DONE = '0') then
63                                  BREW_TRIG <= '1';
64                                  coffee_state <= s5;
65                              else
66                                  BREW_TRIG <= '0';
67                                  coffee_state <= s6;
68                              end if;
69                          when s6 =>              --s6: wait for beep and release of breakfast trigger
70                              if((BEEP_DONE='1') and (MAKE_BREAKFAST_TRIG = '0')) then
71                                  coffee_state <= s1;
72                              else
73                                  coffee_state <= s6;
74                              end if;
75                          when others=>           --others: bad value for coffee_state
76                              coffee_state <= s1;
77                      end case;
78                  end if;     --END of "if (RST = '1')..."
79              end if;         --END of "if rising_edge(CLK)..."
80      end process MAKE_COFFEE;
81      --other processes here...
82  end MY_CMP2;
83
```

Fig 5.2 VHDL state machine that implements the MAKE_COFFEE process shown in Fig 5.1.

To make sense of the Fig 5.2 code, we need to imagine how things look in my kitchen. There, you will find the coffee machine, a clock, a silly box labelled "Make Coffee", and my FPGA control board that is wired to all three of the other devices. My silly "Make-Coffee" box is divided into four sections and each section has a light and a button. The four sections are labelled as Water, Filter, Grounds, and Brew. When I stumble down to the kitchen in the morning, I wait for the FPGA to turn on the light labelled "Water", which the FPGA does exactly at 6AM because it is monitoring the clock. After I put water into the coffee machine, I push the Water-button on the Make-Coffee box, which tells that FPGA that I have loaded the water. Pushing the Water-button also causes the FGPA to turn off the Water-light. I then wait for the Filter-light to come on, whereupon I load the filter into the coffee machine and then press the Filter-button. The Filter-light goes off and the Grounds-light comes on. I then load the coffee grounds into the coffee machine and press the Grounds-button on the Make-Coffee box. The Grounds-light goes off and the Brew-light comes on. I then press the brew/start button on the coffee machine and press the Brew-button on the Make-Coffee box. Finally, I find a comfortable seat and wait for the coffee machine to beep, which tells me (and the FPGA) that the coffee is done.

In the Fig 5.2 example, each task to be performed corresponds to a *state* of the state machine. A *state variable(signal)* used to control the sequential stepping from one state to the next. VHDL allows us to create a state signal as shown in lines 16-17 of Fig 5.2. For this example, the state signal is called coffee_state and I have specified that it can have only six symbolic values called (s1, s2, s3, s4, s5, s6), that each identify a state of the state machine. The VHDL "case" command in line-27 sends control to a set of actions for the state specified by the value of coffee_state. For example, when coffee_state=s2, control is sent to line-38. Included in the set of actions for each state is a command to increment the value of coffee_state to the next state after completion of the current state. For example, in lines 39-45, the value of coffee_state is changed from s2 to s3 after WATER_DONE is found to have a value of 1 (ie. after water has been loaded to the coffee machine).

Because of the code in line-23 of Fig 5.2, the state machine executes on every rising-edge of the clock called CLK. The general flow of things in the Fig 5.2 state machine is as follows:

1) *The state machine sits in the idle state, s1, waiting for MAKE_BREAKFAST_TRIG to equal 1.* That is, another VHDL process (not shown) detects that the time-of-day is 6AM and sets MAKE_BREAKFAST_TRIG high for a few cycles of the clock called CLK. After seeing MAKE_BREAKFAST_TRIG=1, the state machine indicates it is time to go to the next state, s2, by setting the state variable, coffee_state, to a value of s2.
2) *The state machine raises a trigger to indicate that a task needs to be performed.* Each trigger will illuminate a light on the Make-Coffee box. For example, the state machine sets WATER_TRIG=1 to turn on the Water-light and to tell me to put water in the coffee machine.
3) *The state machine waits for a sign that the task has been done and lowers the trigger.* For example, after setting WATER_TRIG=1, the state machine waits for the done-flag called WATER_DONE to be raised and then sets WATER_TRIG=0. The WATER_DONE flag is raised when I push the Water-button on the Make-Coffee box. After seeing the WATER_DONE=1, the state machine indicates it is time to go to the next state, s3, by setting the state variable, coffee_state, to a value of s3.
4) *Handle each needed task sequentially by raising-a-trigger, waiting-for-a-done-flag, lowering-the-trigger, and setting the state variable equal to the identifier for the next state.*
5) *When all tasks are complete, return to step-1).*

Finally, when I apply power to my FPGA board, there is another process (not shown) that causes the signal, RST, used in line-24 to be held high until the FPGA is warmed-up and ready to go. The RST signal is also called a power-on-reset for the FPGA and it is necessary for initializing coffee_state to the value of s1 in the Fig 5.2 example. That is, we want the state machine to startup in state, s1, after power is applied to the FPGA. We will talk more about the importance of initializing things in sections 6.3 and 6.4.

6. Simulation

After some lengthy discussion about VHDL in the previous two chapters, we are now back with the typical IDE flow introduced in Chapter 3. Here, we'll talk about testing HDL code. This chapter is only for those programmers (like me) that can't write error-free code on the first attempt. The rest of you can skip this chapter ;-) .

In later chapters, we will talk about downloading your HDL code to the FPGA. Some refer to this downloading as "burning" code into the FPGA. Continuing with this colorful language, some programmers test their HDL code using the burn-and-crash method. That is, after burning the code into the FPGA, laboratory test equipment is used to send digital inputs to the FPGA and to monitor the resulting outputs from the FPGA. Despite the excitement and confidence of those using the burn-and-crash method, the usual result is that the HDL fails to work properly (ie. the HDL crashes). Furthermore, this burn-and-crash method of testing makes it difficult to troubleshoot HDL problems. Mostly because you can't easily "see" inside the FPGA to find exactly where the HDL has gone wrong.

A *partial* alternative to the burn-and-crash method of HDL testing is the simulation method. I call it a *partial* alternative because simulation can test logical behavior of the HDL but sometimes not the timing aspects (more on this in Chapter 11 when we talk about timing analysis). Simulation is a software method of testing HDL that is found in the flow of many IDEs. As with the burn-and-crash test method, simulation allows you to send inputs to your HDL and to monitor the resulting outputs from your HDL. More importantly, simulation allows you to easily monitor signals that are internal to your HDL, which makes troubleshooting easier. One method of doing simulation is to write a special VHDL component called a *testbench*, as described next.

6.1 Testbenches

A VHDL component called a *testbench* can be used to send digital inputs to another VHDL component-under-test. When the testbench is executed by the simulator tool found in many IDEs, you will get time-line graphs of the digital outputs from the component-under-test. These graphs are similar to what is seen on the screen of a oscilloscope or logic analyzer.

Shown in Fig 6.1 is a testbench component called TB_TWO_BIT that was written to test the component called TWO_BIT shown in Fig 4.1. Discussing the lines of code in Fig 6.1 will help us learn more about VHDL and about how this type of testbench is written. The code for TB_TWO_BIT starts with the usual nice-to-have comments (lines 01-06). Lines 07-09 indicate that standard VHDL libraries will be used. The entity block (lines 11-13) indicates that this component has no *external* inputs or outputs, which may seem odd because we are expecting TB_TWO_BIT to provide inputs to TWO_BIT. However, as discussed below, TWO_BIT, will become an internal part of TB_TWO_BIT and signals generated inside TB_TWO_BIT will become the inputs to TWO_BIT. Specifically, the signals called sCLK, sIN1, and sIN2 that are declared in lines 16-17 will be assigned values by TB_TWO_BIT and used as inputs to TWO_BIT. The signals, sCTL1 and sCTL2, declared on line 18 will be connected to the outputs of TWO_BIT. It is a nice feature of the Xilinx Vivado IDE that all signals declared in a testbench are automatically monitored and graphically displayed by the simulator tool – more on this later.

It is a rule of VHDL that when a small component is used in the description of a larger component then the smaller component must be *declared* and *instantiated* within the larger component. This is exactly the relationship between TWO_BIT (the smaller component) and TB_TWO_BIT (the larger component) and exactly what we must do. For TWO_BIT, the VHDL declaration is shown in lines 20-29 and the VHDL instantiation is shown in lines 32-40. Note that the declaration for TWO_BIT is almost a copy of the entity part of TWO_BIT shown in Fig 4.1. In the instantiation for TWO_BIT, ports of TWO_BIT are tied to the input/output signals declared in lines 16-18. The only remaining task is to write some HDL that assigns values to the input signals, (sCLK, sIN1, and sIN2), and this is done in lines 42-62.

The process called MY_CLK in lines 42-49 of Fig 6.1 assigns values to the signal called sCLK. It does this by repeatedly executing lines 45-48 which do the following: set sCLK equal to 1, wait for 5 ns (ie. 5 x 10^{-9} seconds), set sCLK equal to 0, wait for 5 ns. Because the wait statements in MY_CLK each have the value of 5 ns, a time-series plot of sCLK will be a square wave with a frequency of 100 MHz. That is, the process called MY_CLK has created a 100 MHz clock.

The process called MY_DATA in lines 51-62 of Fig 6.1 assigns values to the signals called sIN1 and sIN2. Like process, MY_CLK, the process called MY_DATA uses signal assignment statements and wait-statements. The only conceptual difference between the two processes is the wait statement in line 61. Since this wait statement has no time parameter then it will wait forever. That is, the process called MY_DATA will execute only once and then wait on line 61 until we manually stop the simulation.

```
01  ----------------------------------------------------------------------------------
02  -- Name:        TB_TWO_BIT.vhd
03  -- Description: Test Bench for component called TWO_BIT.vhd
04  -- Create Date: 05Apr2019
05  -- Revisions:   none
06  ----------------------------------------------------------------------------------
07  library IEEE;
08  use IEEE.STD_LOGIC_1164.ALL;
09  use IEEE.NUMERIC_STD.ALL;
10
11  entity TB_TWO_BIT is
12  -- no ports for this entity
13  end TB_TWO_BIT;
14
15  architecture MY_TB1 of TB_TWO_BIT is
16      signal sCLK : std_logic;
17      signal sIN1,sIN2 : unsigned(1 downto 0);
18      signal sCTL1,sCTL2 : std_logic;
19
20      --Declaration for component-under-test
21      component TWO_BIT is
22          port(
23                  CLK     : in std_logic;
24                  IN1     : in unsigned(1 downto 0);
25                  IN2     : in unsigned(1 downto 0);
26                  CTL1    : out std_logic;
27                  CTL2    : out std_logic
28              );
29           end component;
30
31  begin
32      --Instantiation for component-under-test
33      TBT: TWO_BIT
34            port map(
35                  CLK     => sCLK,
36                  IN1     => sIN1,
37                  IN2     => sIN2,
38                  CTL1    => sCTL1,
39                  CTL2    => sCTL2
40                     );
41
42      --This process creates a 100 MHz square-wave on signal, sCLK
43      MY_CLK: process
44          begin
45              sCLK <= '1';
46              wait for 5ns;
47              sCLK <= '0';
48              wait for 5ns;
49      end process MY_CLK;
50
51      --This process puts data on sIN1 and sIN2
52      MY_DATA: process
53          begin
54              wait for 2ns;     --silly wait
55              sIN1 <= "01";
56              sIN2 <= "10";
57              wait for 12ns;
58              sIN1 <= "00";
59              wait for 4ns;
60              sIN2 <= "01";
61              wait;
62      end process MY_DATA;
63  end MY_TB1;
```

Fig 6.1 VHDL testbench component called TB_TWO_BIT.

6.2 Running Simulation

With the testbench, TB_TWO_BIT, from section 6.1 in-hand, we are ready to test (ie. run a simulation for) component, TWO_BIT, shown in Fig 4.1. In IDEs that have simulation tools, running the simulation is a simple matter of pointing the tool at TB_TWO_BIT, specifying the time-length of the simulation, and hitting the GO-button. When using the Vivado IDE, simulation for TWO_BIT produces the graphical output shown in Fig 6.2.

There are five time-series graphs in Fig 6.2 corresponding to the five signals on lines 16-18 of Fig 6.1. The time axis can be seen above the graphs and extends from 0 ns to 50 ns. It should be clear from the top graph, that sCLK is exactly what we intended to create with process, MY_CLK, in Fig 6.1. Similarly, the graphs for sIN1 and sIN2 show what we intended to create with process, MY_DATA, in Fig 6.1. Recall, that sCLK, sIN1, and sIN2 are routed to the inputs, (CLK, IN1, IN2) of TWO_BIT, our component-under-test. So, for example, when I refer to sCLK then I am also referring to CLK – because they are tied together. In response to these inputs, TWO_BIT has produced the outputs called sCTL1 and sCTL2, which are also graphed in Fig 6.2. These graphs seem to show that TWO_BIT is working properly. That is, procedure MY_PRC1 of TWO_BIT inspects the sum of sIN1 and sIN1. When this sum is greater than 2 then sCTL1 is set equal to 1, otherwise sCTL1 is set equal to 0. Further, Fig 6.2 shows that sCTL2 is simply the digital NOT of sCTL1, as it should be.

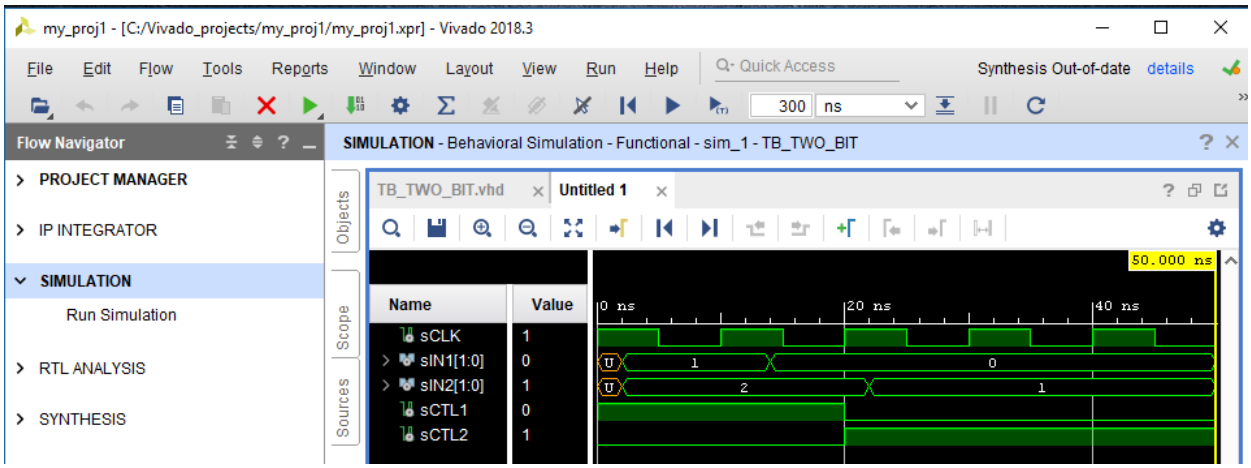

Fig 6.2 Graphical output of the Xilinx Vivado Simulation tool for the testbench component called TB_TWO_BIT.

Some of you are probably saying, “Hey, what’s going on with sIN1 and sIN2 between 0ns and 2ns?”. -more on that in section 6.3. I also suspect that some of you are saying, “Hey, why aren’t sCTL1 and sCTL2 changing at 14ns when sIN1 is set equal to 1?”. The answer comes from line 30 in Fig 4.1, which tells us that process, MY_PRC1, looks at its inputs and calculates (almost instantaneously) its outputs only on the rising-edge of CLK. Since CLK is driven by the signal called sCLK in Fig 6.1 and because sCTL1 and sCTL2 come from the outputs of MY_PRC1 (and of TWO_BIT), we should expect that sCTL1 and sCTL2 will change only on the rising-edge of sCLK. -and, of course, that is what Fig 6.2 shows.

Since managing concurrency is just a chapter-old for us (see Chapter 5), we should mention it again to help with our understanding. As with all VHDL processes, the processes called MY_CLK and MY_DATA in Fig 6.1 run concurrently (ie. in parallel). Managing this concurrency was not difficult for us because MY_CLK and MY_DATA do not interact with each other. Instead, MY_CLK and MY_DATA each do their own thing, toggling sCLK or (sIN1, sIN2) respectively. Component, TWO_BIT, also runs concurrently and interacts with processes MY_CLK and MY_DATA. Generally, interaction makes it more difficult to manage concurrency. However, in this case, it helps to imagine that TWO_BIT simply samples sIN1 and sIN2 at every rising-edge of sCLK and almost immediately calculates values for sCTL1 and sCTL2 based on these samples. That’s mostly all that needs to be said about managing concurrency (or perhaps I should say “understanding the concurrency”) in Fig 6.1. -not too difficult, right?

So, all went well with testing of the component called TWO_BIT using the method of simulation. Of course, more extensive testing of components is often needed to ensure that they are truly working. More extensive testing requires that you write testbenches that are more complex than our example called TB_TWO_BIT. -but you get the idea, and it’s time for us to move on.

6.3 Initialization for Simulation

Now, back to your question from section 6.2 about what's going on in Fig 6.2 with sIN1 and sIN2 between 0ns and 2ns. The answer is that the Vivado simulation tool is telling you that sIN1 and sIN2 are undefined (hence the big "U"). They are undefined because of the silly wait statement that I wrote on line 54 of Fig 6.1. The only purpose of line 54 is to create an example of *undefined* so we could talk about it. If we remove line 54 then sIN1 and sIN2 will have defined values immediately at the start of the simulation, which is what the simulation wants.

Simulation is usually quite picky about things be initialized (ie. having a defined value at the start of the simulation). The initialization feature of VHDL is a convenient way to keep simulation happy. Fig 6.3 shows how VHDL can be used to initialize sIN1 and sIN2 to the values of "01" and "10" respectively. Thus, the undefined problem shown in Fig 6.2, can be solved either by removing the silly wait (Line 54) or by replacing Line 17 in Fig 6.1 with the VHDL initializations shown in Fig 6.3.

```
17a     signal sIN1 : unsigned(1 downto 0) :="01";
17b     signal sIN2 : unsigned(1 downto 0) :="10";
```

Fig 6.3 VHDL initialization for signals, sIN1 and sIN2.

Sometimes, simulation complains about undefined (ie. uninitialized) signals found in the component-under-test. Again, VHDL initialization can be used to correct the problem. Also, uninitialized variables can cause signals to be uninitialized. Again, the solution is to use VHDL initialization. Note that VHDL initialization for signals and variables looks the same (ie. just add the ":=" clause at the end of the declaration for the signal or the variable). Finally, be sure to read section 6.4, which contains further discussion of VHDL initialization.

6.4 Only for Simulation

VHDL is a rich language, but some of its features can only be used when writing simulation testbench components. -and not when defining FPGA configuration. Now, don't get too worried about this. If you try to use a VHDL feature in the wrong place, then the IDE will flag it as an error. The wait statement found in Fig 6.1 is an example of a VHDL feature that should only be used in a testbench component.

In section 4.3, I described the sensitivity list for a VHDL process. However, it is a VHDL rule that a process using wait statements cannot have a sensitivity list. That is why each of the VHDL processes found in Fig 6.1 do not have a sensitivity list. So, these testbench processes with wait statements are an exception to the section 4.3 description of the sensitivity list. After reading to the end of chapter 10, you may want to read Appendix B where you will find many details about the sensitivity list for a VHDL process.

In section 6.3, we saw how VHDL initialization is useful for simulation. When you are done with simulation, you can leave the VHDL initializations in your HDL and the IDE will not complain. However, some argue that VHDL initialization is only useful for simulation and is not reliable for FPGA configuration. Another (arguably better) way to initialize the FPGA configuration is called the power-ON reset that is discussed in Appendix A.

7. High Speed Digital

You can go rather far in your FPGA work by just thinking about HDL coding and not thinking about digital hardware at all. However, skilled FPGA programmers understand the intimate connection between HDL code and hardware. So, let's talk a little about hardware so that we can begin to make this important HDL-to-hardware connection. Also, we need the concepts discussed in this chapter before moving on to the next part of the IDE flow called synthesis, in chapter 10.

7.1 Clocked vs Combinational

In brief, the high-speed digital hardware found inside an FPGA consists of both *sequential-logic* (aka *clocked-logic*) and *combinational-logic*. You are probably familiar with combinational logic, which includes digital "AND" and digital "OR" gates. Perhaps less familiar is sequential logic, which I prefer to call clocked-logic because it requires a clock signal. The *flip-flop* is an example of clocked-logic hardware that is used *very often* inside the FPGA. In fact, without flip-flops and clocks, much of what we do with FPGAs would not be possible!

7.2 Digital Register

Flip-flop is a fun word to say, but it's a long word. So, to simplify my typing, I will usually call a flip-flop by its alternate name, which is register. In Fig 7.1 are three examples of a simple register (also called a D-flip-flop), which I have identified as, S1_reg, S2_reg, and S3_reg. This simple register has inputs, C and D, output, Q, and operation that is described as follows. When the clock signal entering the C-input transitions from low-to-high (ie. on the rising-edge of the clock), then the value of the signal at the D-input is captured and transferred to the Q-output.

In this book, you will sometimes see digital registers that have the other pins listed below.

- CE: clock enable (when CE=1, the C-input is enabled/active)
- CLR: asynchronous clear/reset (when CLR goes high, Q goes low immediately)
- R: synchronous clear/reset (when R goes high, Q goes low on next rising-edge at C)
- PRE: asynchronous preset/set (when PRE goes high, Q goes high immediately)
- S: synchronous preset/set (when S goes high, Q goes high on next rising-edge at C)

7.3 Important Example

Careful study of Fig 7.1 will introduce some important concepts for FPGA work. So, please find a quiet place and carefully read the rest of this section. First, some terminology. Instead of writing "the D-input to register, S1_reg", I will simply write S1_reg/D. Also, instead of writing "the a-input to AND-gate, A2", I will write A2/a. Finally, instead of writing "the Q-output of register, S1_reg, is in the high logic state", I will write S1_reg/Q=1 (or S1_reg/Q=0 for the low logic state). Finally, a reminder that the two-input AND-gate is a device whose output is high when both of its inputs are high, otherwise its output is low.

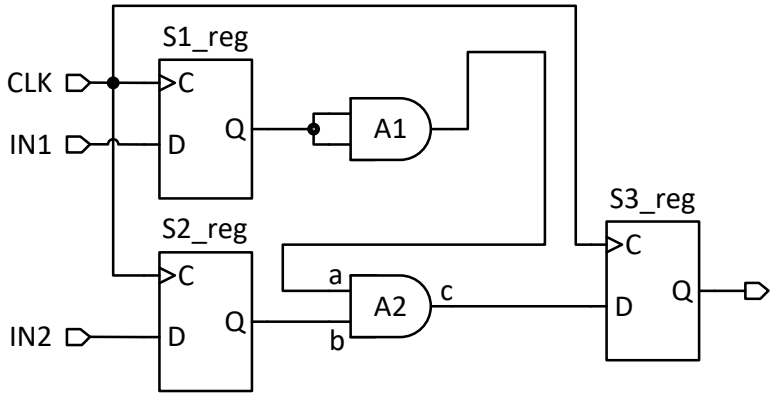

Fig 7.1 Example digital circuit containing clocked-logic and combinational-logic.

There are three input signals, (CLK, IN1, IN2), to the Fig 7.1 circuit that enter the circuit from the left side. A single output signal, S3_reg/Q, exits the circuit on the right side. The timing diagram in Fig 7.2 shows how example-inputs travel through the circuit to become the output. At time-t0, we see from Fig 7.2 that S1_reg/Q=1 and S2_reg/Q=0, from which we can infer that on the rising-edge (not shown) of CLK that precedes time-t0, IN1=1 and IN2=0. At time-t1, CLK has a rising-edge and shortly thereafter we see that S1_reg/Q and S2_reg/Q transition to the new states of 0 and 1 respectively. From this, we can infer that IN1=0 and IN2=1 at time-t1.

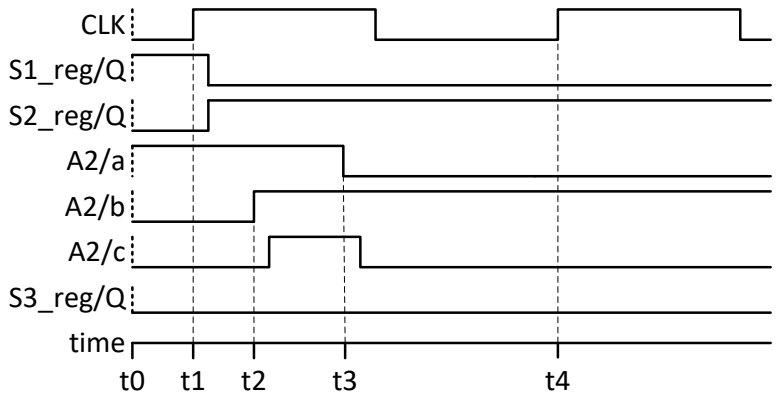

Fig 7.2 Timing diagram for digital signals found in the circuit of Fig 7.1.

The fact that Fig 7.2 shows S1_reg/Q and S2_reg/Q transitioning a short time after the rising-edge of CLK demonstrates an important concept. The concept being that things happen fast in a digital circuit but not instantaneously. That is, upon seeing a rising-edge at input-C, a register needs a little time to sample input-D and pass that sampled value to output-Q. Also, digital signals need a little time to travel down wires. We can see this in Fig 7.2 by noting that the signal coming out of S1_reg/Q travels through AND-gate, A1, and down a long wire to finally reach A2/a at time-t3. The signal coming out of S2_reg/Q travels down a short wire to reach A2/b at time-t2. Shortly after time-t2, the AND-gate, A2, responds to the fact that both of its inputs are high and sets A2/c=1. Shortly after time-t3, A2 responds to the fact that one of its inputs has gone low and sets A2/c=0.

Fig 7.2 demonstrates the important concept that *different arrival times* for inputs to combinational-logic (ie. to A2) can produce a potential problem called a *glitch* on the output of the combinational-logic. In this example, the glitch is the fact that A2/c has gone high for a short period of time during the time interval from t1 to t4. Preventing glitches from becoming a problem is the primary reason that we use clocked-logic (ie. registers). In the Fig 7.2 example circuit, it is important to note how use of S3_reg has prevented the glitch on A2/c from reaching S3_reg/Q. That is, at time-t4, the rising-edge of CLK causes S3_reg to sample a glitch-free signal coming from A2/c and to pass the sample to S3_reg/Q.

It's important for you to recognize the glitch as something that is out-of-place. That is, it exists only because of the different arrival times for inputs to combinational-logic. Said another way, one expects the output, S3_reg/Q, of the Fig 7.1 circuit to be the logical-AND of the two inputs, (IN1 and IN2). Fig 7.2 shows that the sampled values of our example inputs are (S1_reg/Q=1, S2_reg/Q=0) immediately before time-t0 and are (S1_reg/Q=0, S2_reg/Q=1) immediately after time-t0. Hence, we expect the logical-AND for both sets of inputs to be 0. Fortunately, the overall circuit of Fig 7.1 behaves as we expect. That is, Fig 7.2 shows that S3_reg/Q=0 for both sets of inputs. However, without the use of clocked-logic (in particular, S3_reg), the glitch would become an output of the circuit and could cause problems downstream (ie. in circuits connected to S3_reg/Q).

In summary, glitches are BAD and we use clocked-logic to prevent glitches from becoming a problem.

7.4 RTL Paradigm

Paradigm roughly means "a way of looking at things or doing things". Paradigm is not a fun word to say and you should not use it among friends (sorry). However, in the title for this section I have purposely used the abbreviation, RTL, and it stands for Register-Transfer-Level. In short, RTL is a VHDL programming style that causes clocked-logic to be used for FPGA configuration. Clocked-logic is GOOD, right? If you are not so sure, then you need to again read sections 7.1 thru 7.3. Generally, one writes RTL-style code by using VHDL *clocked-processes* that look like the one shown in Fig 7.3. The important thing to see in Fig 7.3 is that operations in the process are placed between the line of code that reads "`if rising_edge(CLK) then`" and the line of code that reads "`end if;`". There will be exceptions to this simplified

description of RTL-style coding, but not very many. This simplified description of RTL-style coding is something you should commit to memory. With so many things to learn about FGPAs, we need to start with simplified descriptions that we can remember and later build upon.

```
01      MY_RTL: process(CLK)                        --outputs: (OUT1)
02          begin
03              if rising_edge(CLK) then
04                  --
05                  -- operations that create a value for OUT1
06                  --
07              end if;
08      end process MY_RTL;
```

Fig 7.3 A VHDL process written in the RTL style.

Writing testbenches (see section 6.1) is one big exception where I (and many others) do not write RTL style code. The reason being that I like to use VHDL wait-statements when writing testbenches and VHDL does not allow the use of wait-statements inside a clocked process.

7.5 HDL-to-Hardware Connection

In this short section, we take our first steps towards making the important HDL-to-hardware connections. That is, the concepts listed below indicate how things in our VHDL code become things inside the FPGA.

1. VHDL *signals* that are assigned a value inside a clocked-process become digital registers.
2. VHDL *signals* that are *not* assigned a value inside a clocked-process represent a connection (ie. a wire).
3. *Operations* (eg. boolean algebra, arithmetic, if-then-else) performed on signals become combinational-logic.
4. Usually, a VHDL *variable* is used inside a clocked-process to help describe an *operation*. Generally, the VHDL variable becomes part of the combinational-logic for the operation. However, the value of a VHDL variable must sometimes be remembered from execution-to-execution of a clocked-process. In this case, the variable becomes a register.

For example, the schematic in Fig 7.4 shows how TWO_BIT.vhd will look inside the FPGA after applying these concepts to the VHDL in Fig 4.1. Inside TWO_BIT.vhd, note that the inputs, (CLK, INP1, INP2), and the outputs, (CTRL1, CTRL2), are not assigned values inside a clocked-process. Hence, by concept-2, these signals become connections/wires inside TWO_BIT.vhd. Next, note that only the signal called bCTL1 is assigned a value inside a clocked process. Hence, by concept-1, it becomes a register. All the operations, (IN1+IN2, if-then-else, boolean NOT), become combinational logic according to concept-3. Finally, the variable called sum1 is calculated anew during every execution of the clocked-process, MY_PRC (ie. the value of sum1 need not be remembered from execution-to-execution this process). Hence, by concept-4, sum1 becomes part of the combinational logic.

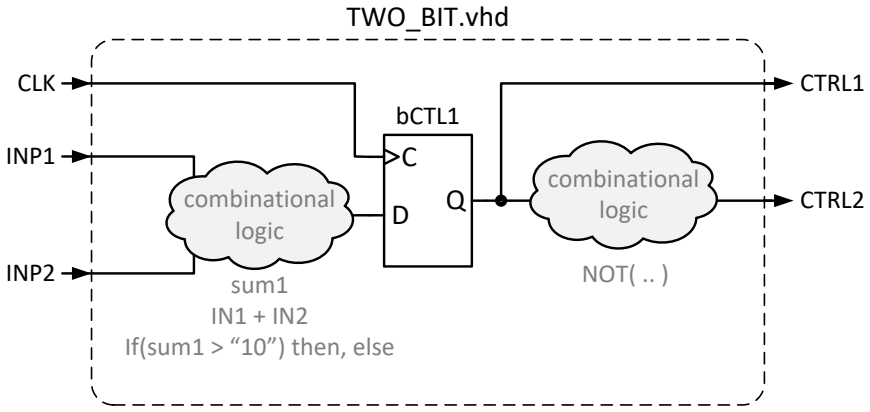

Fig 7.4 How component, TWO_BIT.vhd, would look inside the FPGA.

7.6 What Ifs

Some of you probably have "what-if" questions about the circuit shown in Fig 7.1. For example, what if the time (t4 minus t1) between the rising-edges of CLK where shorter? Specifically, what if the time between rising-edges of CLK were shortened so that time, t4, aligned with time, t3? Well, then we've got a problem because the A2/c glitch discussed in section 7.3 would be captured by S3_reg and appear at S3_reg/Q. In FPGA lingo, a clock having shorter time between rising-edges is called a *faster* clock. So, our "what-if" question has helped us discover the important fact that *clocked-logic might not function properly (ie. remove glitches) if we use a clock that is too fast*.

A subtler "what-if" question is, what if the D-input to a register is changing at the exact same time as the rising-edge of CLK? You get bonus points if you thought of that question! Anyway, if the D-input to a register changes too close in time to the rising-edge of CLK then a bad thing called *metastability* happens. Metastability is a fancy word indicating that the Q-output of the register becomes unstable until at least the next rising-edge of CLK. Here, the word unstable means that the Q-output can oscillate or wander in an unpredictable fashion between the usual logic levels of high and low. How does one prevent metastability from occurring? Again, as with the previous what-if question, the answer is to use a clock that is not too fast. That is, in the Fig 7.1 example, we must choose a clock speed so that the CLK rising-edge time, t4, occurs when A2/c is stable at its post-glitch value.

So, we have learned that slow clocks simplify FPGA work because they prevent bad things from happening. However, you are probably still full of questions? Perhaps you are wondering how to determine slow-enough? Perhaps you note that your design has lots of code (ie. lots of clocked-logic circuits) and are wondering how to ensure that all these circuits have a slow-enough clock? The good news is that the IDE has a tool called *timing analysis* that will check all of this (and more) for you. Knowing what do when timing analysis identifies a problem is what makes FPGA programmers truly special in the programming world. We will talk more about FPGA timing analysis in Chapters 11 and 14.

7.7 Key Points

- HDL code tells the FPGA how to configure its internal digital hardware (ie. HDL describes the FPGA configuration). Skilled FPGA programmers understand the connection between code and hardware.
- The high-speed digital hardware found inside an FPGA consists of both *sequential-logic* (aka *clocked-logic*) and *combinational-logic*.
- The clocked *flip-flop* (aka *register*) is an example of clocked-logic hardware that is used very often inside the FPGA. Without registers and clocks, much of what we do with FPGAs would be impossible!
- Register-Transfer-Level (RTL), is an HDL programming style that causes clocked-logic to be used for FPGA configuration. Generally, the VHDL signals in RTL-style code correspond to digital registers and the operations performed on the signals correspond to combinational-logic.
- *Different arrival times* for inputs to combinational-logic can produce a potential-problem called a *glitch* on the output of the combinational-logic.
- If the D-input to a register changes too close in time to the rising-edge of CLK then a bad thing called *metastability* happens, whereupon the Q-output of the register can become unstable.
- Preventing metastability and glitches from becoming a problem can sometimes be done by using a slower clock for the clocked-logic.
- The IDE has a tool called *timing analysis* that is used to determine if metastability, glitches, and other bad things could result from your HDL.

8. FPGA Clocks

As we discussed in Chapter 7, without registers and clocks, much of what we do with FPGAs would not be possible. In Chapter 7, we also found that a clocked-logic circuit in the FPGA can develop problems (metastability and glitches) when the clock used by the circuit is too fast. In this chapter, we will discuss how to create clocks for your project and how

clocks get special treatment inside the FPGA. Again, concepts discussed here are necessary before we move on to the next part of the IDE flow called synthesis.

8.1 Jitter and Skew

In our chapter 7 discussion about the circuit shown in Fig 7.1, we assumed that the CLK signal was steady and reached all registers at the same time. In FPGA lingo, a clock that is not steady is said to have *jitter*. Also, in FPGA lingo, when the clock does not reach all registers in the circuit at the same time then the clock is said to be *skewed*. Both jitter and skew can cause an otherwise perfectly good clock to be too fast (ie. to cause metastability and glitch problems).

With these cautions in mind, let's talk about how clocks are created and then distributed in our FPGA configuration. In the previous sentence, I have used the plural word, clocks, on purpose because our HDL (and the corresponding FPGA configuration) will often use more than one clock. In FPGA lingo, all the clocked-logic circuits that operate from a specific clock are called the *clock-domain* for that clock. Creating multiple clocks and using them during HDL coding is not much more difficult than creating/using one clock. However, passing a digital signal from one clock-domain to another requires the use of a *clock-crosser* circuit and complicates the timing analysis that we introduced in Chapter 7. We'll talk more about multiple clocks and clock-crosser circuits in Chapter 12.

8.2 Source and Buffers

The source of clock signals used inside the FPGA is usually a very stable oscillator that is located outside the FPGA. Typically, one buys a crystal-based oscillator because of its low jitter (typically near 1 picosecond RMS). Output from the oscillator is sent through a buffer circuit to a special clock-capable pin (or pin-pair) on the FPGA. The clock-buffer circuit can be one of the many commercially-available integrated circuits (ICs) and it is usually placed on the circuit board that contains the FPGA. The purpose of the buffer circuit is to both protect the expensive FPGA from an accidental connection (don't tell me you've never made an accidental connection) and to convert the oscillator output into the type of digital signal required at the FPGA clock-capable pin(s). The buffer also protects the FPGA from possible damage that can occur when the external oscillator is powered ON before the FPGA.

8.3 Clock-Module and Tree

Regardless of whether your FPGA project uses one or many clocks, it is highly recommended that you use only one source clock. In FPGA lingo, this one source clock is called the *base-clock*. Inside the FPGA and near the clock-capable pin(s) that accept the base-clock is a special circuit called a *clock-module*. The clock-module is used as a buffer (in one-clock projects) and as a creator of other clock signals (in multi-clock projects). Older clock-modules could only create clocks that were equal to or slower than the base-clock. However, newer clock-modules use something called a Phase Locked Loop (PLL) to create clocks of almost any frequency from the base-clock.

There are many advantages to using one base-clock and to using the FPGA clock-module:

1) The clock-module can "clean up" the base-clock by reducing jitter.
2) Each time the clock-module is powered ON, all the generated clocks have a consistent relationship with each other. This is often called clock phase alignment and typically means that the power-ON rising edge for all generated clocks occurs at the same time. Without phase alignment for the clocks used in the FPGA, the timing analysis introduced in Chapter 7 becomes impossible.
3) The clock-module automatically routes each of its clock outputs to special circuits in the FPGA called a *clock tree*. A clock tree is designed to distribute clock signals throughout the FPGA with low skew.
4) Most IDE have an easy-to-use *wizard* that helps setup the clock-module. Often, this setup is little more than simply specifying the frequency and jitter of the base-clock and the frequency of each desired output clock. All the other nasty details of clock-modules and clock trees are automatically taken care of for you. We'll talk more about IDE *wizards* in Chapter 9.
5) Some of the newer setup wizards for the clock-module will also automatically write clock constraints needed for timing analysis. We'll talk more about constraints when we discuss timing analysis in chapter 11.

While learning VHDL, one usually sees an example process that "divides down" a fast clock to create a slow clock. In FPGA lingo, this is called creating a clock in the FPGA *fabric* (more about FPGA fabric in section 10.3). It is strongly recommended that you create all your clocks by using the FPGA clock-module and that you never create clocks in the FPGA fabric. Why? Well, in simple terms, you lose all the advantages (listed above an more) that come with using a clock-module.

8.4 Jitter (again)

A common measurement of clock jitter is called *period-jitter*. In short, period-jitter is a measure of random variations in the clock period. Typically, period-jitter is specified by its Root-Mean-Squared (RMS) value in picoseconds (ps). However, as the acronym implies, the RMS value for period-jitter is a kind of average value. Thus, it is possible for period-jitter to occasionally rise above the RMS value and cause problems in the otherwise orderly transfer of data between registers in the FPGA. Unfortunately, we live in an imperfect world and must tolerate some error. So, if you are transferring a million bits of data between to registers, do you care that one bit is wrong? Often, the answer to this question is yes – we do care – and we want a much lower Bit-Error-Rate (BER) than one in one-million (ie. 1 in 10^6). It many applications, it common to accept a BER of 1 in 10^{12}.

We can combine the *measured* RMS value for period-jitter with our *desired* BER to arrive at a new specification called Peak-to-Peak (Pk-Pk) jitter. For a BER of 1 in 10^{12}, the formula is simply: Pk-Pk jitter = 14 times RMS period-jitter. When the Vivado IDE asks you to specify clock jitter (eg. during setup of a clock-module), it is asking for a value of Pk-Pk jitter. Note that Pk-Pk jitter is hybrid quantity that combines a measured characteristic (period-jitter) of the clock and a desired specification (BER). However, here and elsewhere, you will see Pk-Pk jitter used as if it were simply a measured characteristic of a clock.

Finally, you will find that the FPGA clock module creates clocks having quite a lot of jitter. For example, a good crystal oscillator will produce a clock having jitter of about 14 ps Pk-Pk. However, clock outputs of an FPGA clock module have typical jitter of 100 ps Pk-Pk. Further, you will find that jitter on the output clocks of the clock module depend little upon jitter found on the input clock. For example, when input clock jitter ranges from 14-40 ps Pk-Pk, the Vivado IDE shows that jitter on the output clock(s) changes by less than 1%, for the clock module found in Xilinx 7-Series FPGAs.

9. IP and Wizards

Xilinx provides a version of the Vivado IDE that is free of charge! This so-called WebPACK version of Vivado lacks only a few features and some device support found in the not-for-free Vivado. Also, inside Vivado are lots of free VHDL components that Xilinx calls Intellectual Property (IP) code/cores. Don't worry, despite all this free stuff, Xilinx makes plenty of money selling FPGAs.

You can view available IP by clicking on "IP Catalog" found in the Flow Manager window of Vivado (see top-left corner of Fig 9.1). In the catalog that appears (see Fig 9.1), you can simply find and click on the IP that you need. Double-clicking on an IP found in the catalog will bring up a multi-tab window (aka a *wizard*) that allows you to configure the IP to your special needs. Fig 9.2 and Fig 9.3 show the first two tabs of the wizard for the IP called "Clocking Wizard" that I have highlighted (in blue) in Fig 9.1. The Clocking Wizard IP will be discussed often in this book and helps us configure the important clock-module that we discussed in section 8.3.

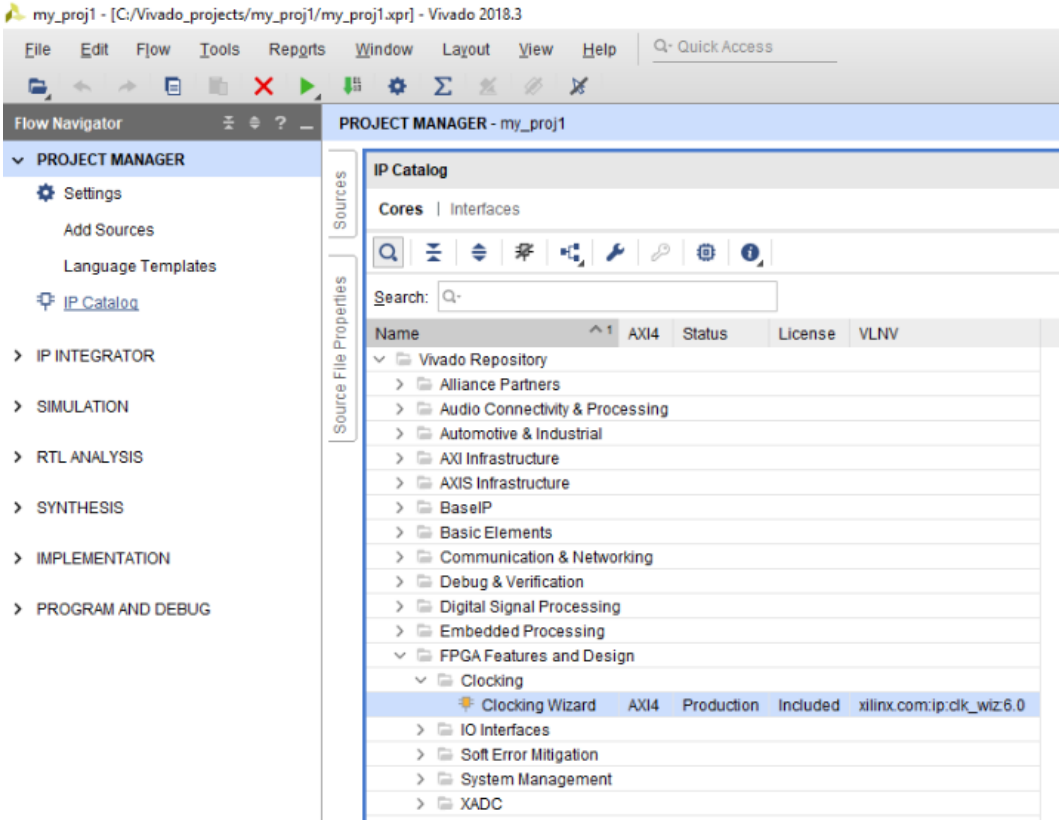

Fig 9.1 The Xilinx Vivado IP Catalog.

Let's look a little closer at the Clocking Wizard screen-shots shown in Fig 9.2 and Fig 9.3. First, as shown in Fig 9.2, I have given this IP component the name, CLK_GEN (short for clock generator), since I want it to generate the digital clocks for my FPGA project. Also in Fig 9.2, I have specified that I want to use a specific clock-module device found inside the FPGA called a *Mixed Mode Clock Manager (MMCM).* In the bottom of Fig 9.2, I have specified "Input Frequency" to the MMCM is a 100MHz base-clock that will be coming from "Differential clock capable pins" of the FPGA that received the base-clock from an external source (eg. crystal oscillator). That is, the 100MHz clock input to the FPGA will be a two-wire differential-digital signal sometimes called LVDS (see section 15.3.8 for more about LVDS). Finally, as discussed in section 8.4, I have told the Clocking Wizard that "Input Jitter" is 14ps Pk-Pk for the base-clock. All other setup options for the MMCM shown in Fig 9.2 have been left at their default values.

In Fig 9.3, I have specified the frequency in MHz, (166.666, 250.000, 100.000), for each of the three clocks needed by my FPGA project. I have also specified (using a checkbox) that the MMCM will have an output called "locked". This locked-output is low for a short time after power-up of the MMCM and then goes high when all the clock outputs of the MMCM are stable. This locked-output is often used by HDL programmers as part of a Power-ON Reset (POR) for the FPGA (see Appendix A). All the other setup options shown in Fig 9.3 and in all the other tabs of the Clocking Wizard have been left at their default values.

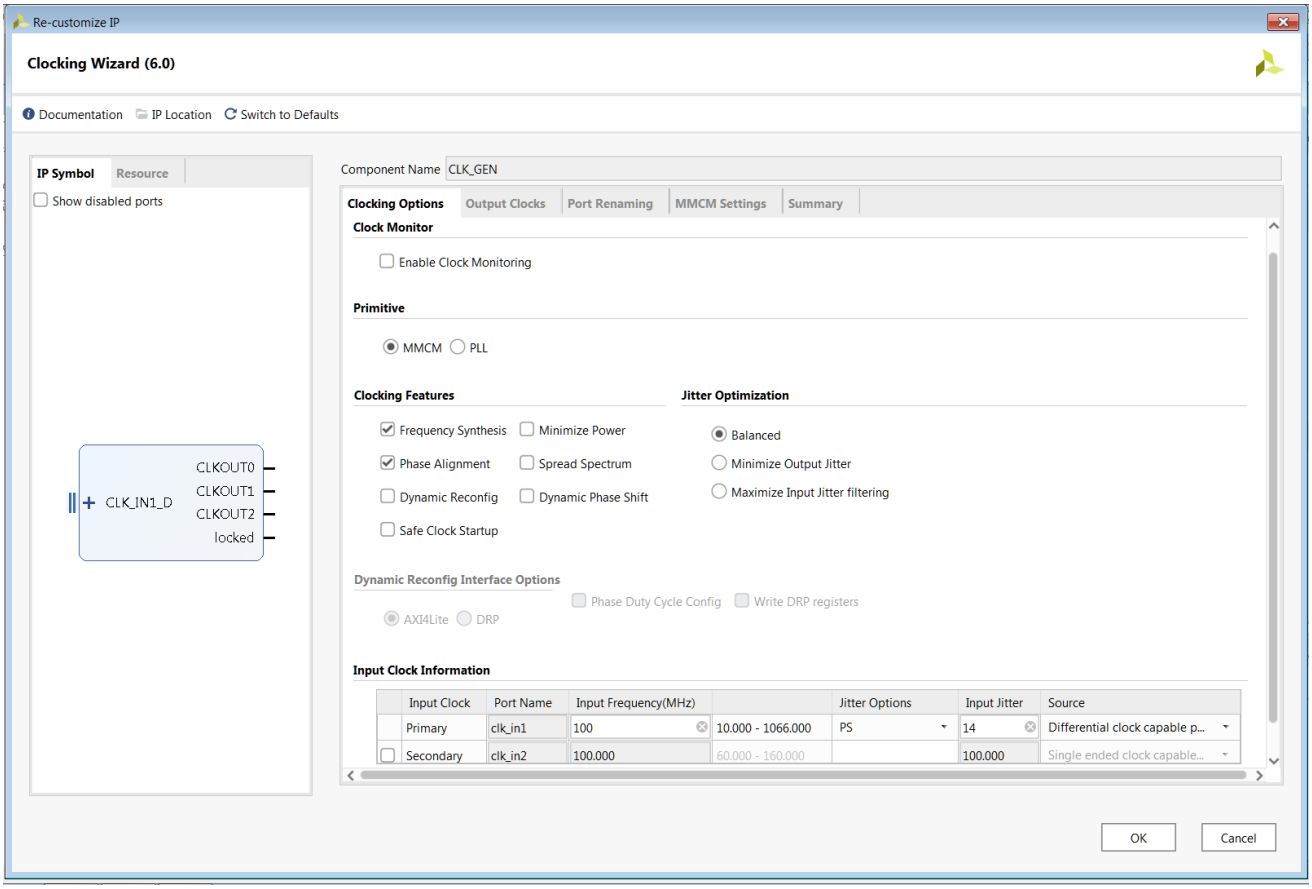

Fig 9.2 Window for first tab of the Xilinx IP called the Clocking Wizard.

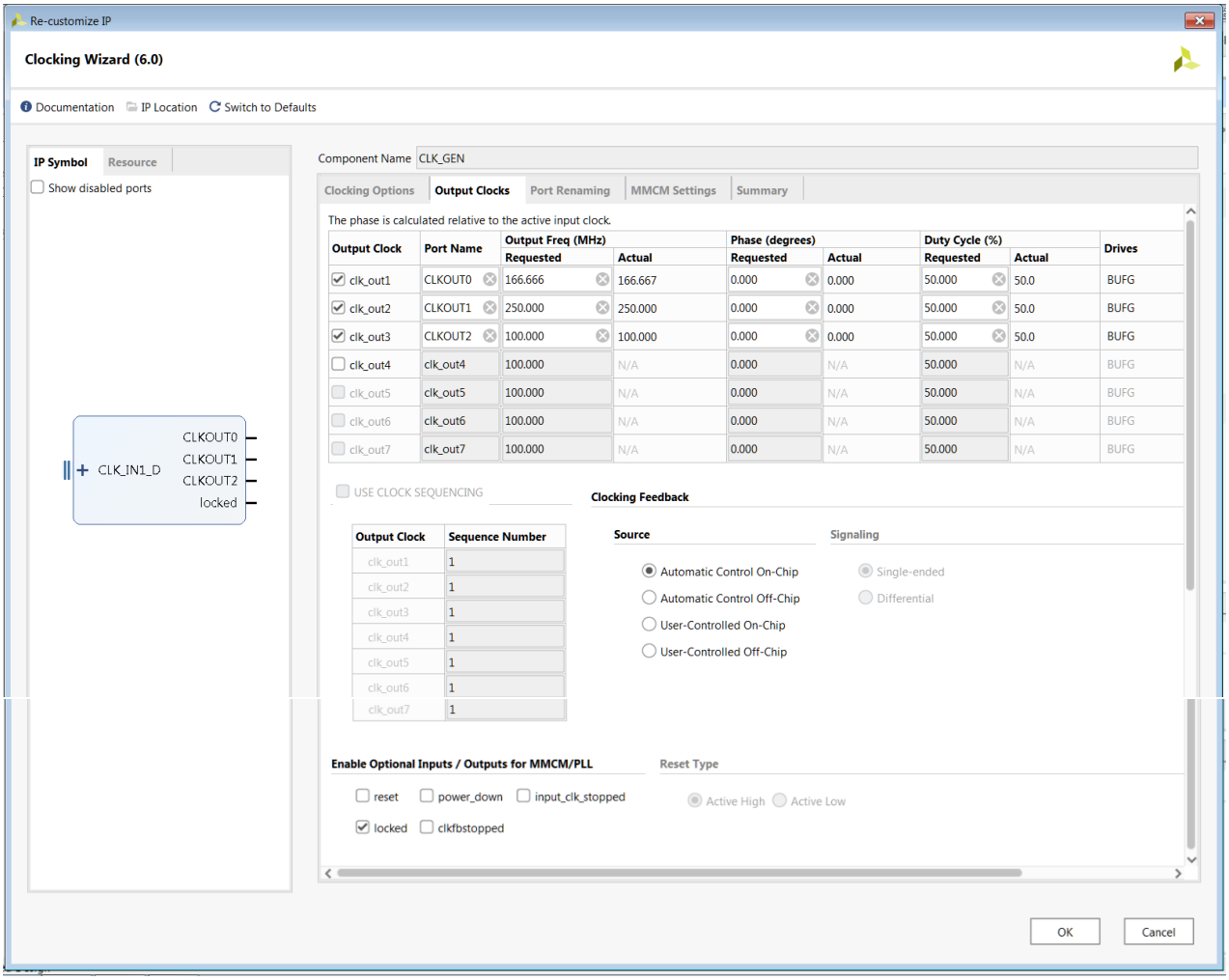

Fig 9.3 Window for second tab of Xilinx IP called the Clocking Wizard.

The Clocking Wizard IP will automatically generate and store several files in your Vivado project. However, a dialog box will popup at the end of IP configuration and will give you the option of generating some of these files as “Global” or “Out of context per IP”. Selecting “Out of context per IP” is generally preferred since it will save time later when you do Synthesis for your entire project. Of all the files generated after IP configuration, perhaps the most important file for the VHDL programmer is called the *instantiation template*. For the MMCM called CLK_GEN that we configured above, the instantiation template file is called CLK_GEN.vho and is found by clicking on the “IP Sources” tab of the “Sources” window in Vivado (see Fig 9.4).

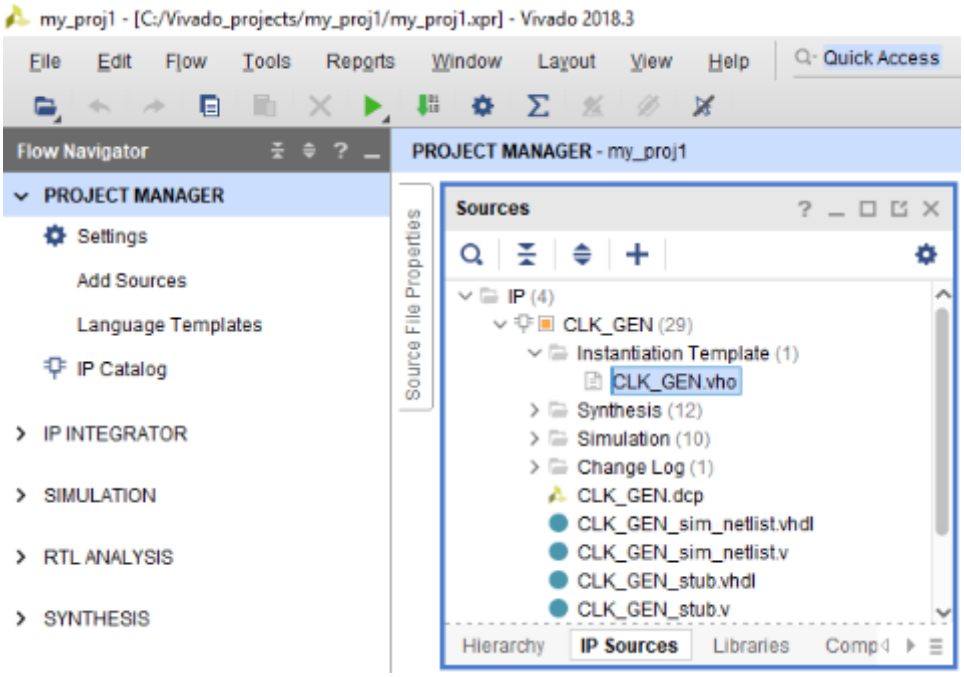

Fig 9.4 Locating the instantiation template, CLK_GEN.vho, for the Xilinx Clocking Wizard component called CLK_GEN.

Essential parts of the VHDL instantiation template for the IP component called CLK_GEN are shown in Fig 9.5. The contents of this template indicate that the CLK_GEN component is used like any other VHDL component. That is, the CLK_GEN component can be used inside another VHDL component (eg. our FPGA project) by simply *declaring* and *instantiating* CLK_GEN using the Fig 9.5 template. If you have forgotten about declaring and instantiating VHDL components, then no worries. In section 10.1, I will show you how to do this for the CLK_GEN component.

In conclusion, this chapter has shown how to configure Xilinx IP using the example of an MMCM clock-module. Configuring all Xilinx IP is made simple by windows-of-options called wizards. After working through the wizard, the Vivado IDE generates files (including an instantiation template) and stores them in the Vivado project folder. Thereafter, the IP looks like any other VHDL component and is used by simply declaring and instantiating the IP into your project, using the instantiation template as a guide.

```
-- The following code must appear in the VHDL architecture header:
------------ Begin Cut here for COMPONENT Declaration ------ COMP_TAG
component CLK_GEN
port
 (-- Clock in ports
  -- Clock out ports
  CLKOUT0           : out    std_logic;
  CLKOUT1           : out    std_logic;
  CLKOUT2           : out    std_logic;
  -- Status and control signals
  locked            : out    std_logic;
  clk_in1_p         : in     std_logic;
  clk_in1_n         : in     std_logic
 );
end component;

-- COMP_TAG_END ------ End COMPONENT Declaration ------------
-- The following code must appear in the VHDL architecture
-- body. Substitute your own instance name and net names.
------------ Begin Cut here for INSTANTIATION Template ----- INST_TAG
your_instance_name : CLK_GEN
   port map (
  -- Clock out ports
   CLKOUT0 => CLKOUT0,
   CLKOUT1 => CLKOUT1,
   CLKOUT2 => CLKOUT2,
  -- Status and control signals
   locked => locked,
   -- Clock in ports
   clk_in1_p => clk_in1_p,
   clk_in1_n => clk_in1_n
 );
-- INST_TAG_END ------ End INSTANTIATION Template ------------
```

Fig 9.5 Essential parts of the VHDL instantiation template for the Xilinx Clocking Wizard component called CLK_GEN.

10. Synthesis

We departed from the IDE flow in Chapter 6 after talking about simulation. Finally, after some preparatory work in Chapters 7-thru-9, we are ready to talk about the synthesis, which is the next step in the IDE flow after simulation. In short, synthesis (aka logic synthesis) is a tool that converts your HDL into a *netlist*. Here, the meaning of netlist is a circuit schematic for the FPGA configuration. This circuit schematic, like all good schematics, shows parts (ie. digital components), identifies each part with a unique name, and shows how the parts are connected. The HDL-to-hardware discussion in section 7.5 is a nice introduction to the circuit creation rules followed by synthesis.

Few of us can appreciate the synthesis tool because we've never had to work without it. Here's what one old-timer, Ed Klingman, said about synthesis in a 2004 Embedded Systems Programming article, "*That is, your program logic gets synthesized, or mapped into, logical gates, not into processor instructions that control multigate structures. This is absolutely amazing, and good FPGA programmers give thanks every day for living in the rare time in history (post 1990+) when you can design architectures with words and then synthesize your logic into (mostly silicon) gates that execute your logic. Not to get carried away, but it's absolutely wonderful.*"

Before we launch the synthesis tool and talk about what happens, there is still a little preparation to be done. First, we need to finish the discussion started in Section 4.6 about the top-level component for our VHDL project.

10.1 Top-Level Component (again)

In chapter 4 we talked about using VHDL to both describe components and to describe how these components are connected. The result of our VHDL coding is the circuit that we want the FPGA to create using its internal hardware. This resulting circuit is usually a component itself and is called the *top-level component*. That is, the top-level component is what ties together all the other VHDL components in our project. This "tying together" of other

components often means that we simply declare and instantiate the other VHDL components inside the top-level component. So, let's create a top-level component for our VHDL example project, my_proj1, and talk about it.

As discussed in chapter 4, we use the Vivado IDE to add a file of source code to the project. For this new source, we will use the name, TOP.vhd, which seems like a good name for our VHDL top-level component. The VHDL that I typed into TOP.vhd is shown in Fig 10.1. Lines 10-20 of Fig 10.1 are called the entity part of component, TOP.vhd. Since, this is now the top-level component for our project, the inputs and outputs shown in lines 12-17 represent pins on the FPGA. Pins of the FPGA that are associated with these inputs and outputs are specified by typing entries (as shown in Fig 10.2) into the project constraints file, which we have been calling **constraints1.xdc**. Note that the Fig 10.2 version of **constraints1.xdc** replaces the version shown in Fig 4.4 since TOP.vhd (and not TWO_BIT.vhd) is now the top-level component for our project.

```
01  ----------------------------------------------------------------------------------
02  -- Module Name: TOP.vhd
03  -- Description: Top-level VHDL component of the Xilinx Vivado project called my_proj1
04  -- Date: 15Jun2019
05  ----------------------------------------------------------------------------------
06  library IEEE;
07  use IEEE.STD_LOGIC_1164.ALL;
08  use IEEE.NUMERIC_STD.ALL;
09
10  entity TOP is
11      port(
12           CLK_IN1_P  : in std_logic;
13           CLK_IN1_N  : in std_logic;
14           INP1       : in unsigned(1 downto 0);
15           INP2       : in unsigned(1 downto 0);
16           CTRL1      : out std_logic;
17           CTRL2      : out std_logic
18             -- other inputs and outputs can be placed here
19           );
20  end TOP;
21
22  architecture MY_TOP of TOP is
23      signal CLK166, CLK250, CLK100, clks_locked : std_logic;
24
25      --Declaration of IP component copied from CLK_GEN.vho (main clocks for this project)
26      component CLK_GEN
27          port(
28               clk_in1_p   : in     std_logic;
29               clk_in1_n   : in     std_logic;
30               CLKOUT0     : out    std_logic;
31               CLKOUT1     : out    std_logic;
32               CLKOUT2     : out    std_logic;
33               locked      : out    std_logic
34           );
35      end component;
36
37      --Declaration of custom component, TWO_BIT.vhd
38      component TWO_BIT is
39          port(
40               CLK     : in std_logic;
41               IN1     : in unsigned(1 downto 0);
42               IN2     : in unsigned(1 downto 0);
43               CTL1    : out std_logic;
44               CTL2    : out std_logic
45           );
46       end component;
47
48  begin
49
50      --Instantiation of IP component copied from CLK_GEN.vho
51      MMCM1: CLK_GEN
52      port map(
53              clk_in1_p   => CLK_IN1_P,      --input
54              clk_in1_n   => CLK_IN1_N,      --input
55              CLKOUT0     => CLK166,         --output
56              CLKOUT1     => CLK250,         --output
57              CLKOUT2     => CLK100,         --output
58              locked      => clks_locked     --output
59              );
60
61      --Instantiation of custom component, TWO_BIT.vhd
62      TBT: TWO_BIT
63      port map(
64              CLK     => CLK250,             --input
65              IN1     => INP1,               --input
66              IN2     => INP2,               --input
67              CTL1    => CTRL1,              --output
68              CTL2    => CTRL2               --output
69              );
70  end MY_TOP;
```

Fig 10.1 Initial version of the top-level VHDL component called TOP.vhd for the Vivado example project called my_proj1.

```
01  # Xilinx Vivado constraints file for project, my_proj1
02  # Name: constraints1.xdc
03  # Date: 11May2019
04
05  #Pins W9 and Y9 are a clock-capable LVDS pin-pair
06  set_property IOSTANDARD LVDS [get_ports CLK_IN1_P]
07  set_property PACKAGE_PIN W9 [get_ports CLK_IN1_P]
08  set_property IOSTANDARD LVDS [get_ports CLK_IN1_N]
09  set_property PACKAGE_PIN Y9 [get_ports CLK_IN1_N]
10
11  set_property IOSTANDARD LVTTL [get_ports {INP1[0]}]
12  set_property PACKAGE_PIN E13 [get_ports {INP1[0]}]
13  set_property IOSTANDARD LVTTL [get_ports {INP1[1]}]
14  set_property PACKAGE_PIN H13 [get_ports {INP1[1]}]
15
16  set_property IOSTANDARD LVTTL [get_ports {INP2[0]}]
17  set_property PACKAGE_PIN F13 [get_ports {INP2[0]}]
18  set_property IOSTANDARD LVTTL [get_ports {INP2[1]}]
19  set_property PACKAGE_PIN G13 [get_ports {INP2[1]}]
20
21  set_property IOSTANDARD LVTTL [get_ports CTRL1]
22  set_property PACKAGE_PIN W20 [get_ports CTRL1]
23  set_property IOSTANDARD LVTTL [get_ports CTRL2]
24  set_property PACKAGE_PIN V20 [get_ports CTRL2]
```

Fig 10.2 Part of the constraints file for Vivado example project called my_proj1.

Continuing our discussion of Fig 10.1, we see that the VHDL component called TWO_BIT.vhd from chapter 4 is declared in lines 37-46 and instantiated in lines 61-69. Note how the instantiation of TWO_BIT.vhd ties most of its inputs and outputs directly to the inputs and outputs of TOP.vhd. In this way, these inputs and outputs of TWO_BIT.vhd remain tied directly to the FPGA pins, as they were in chapter 4. In the same way that we declare and instantiate a VHDL module, the IP clock-module called CLK_GEN (discussed in chapter 9) is declared in lines 25-35 and instantiated in lines 50-59. Outputs of CLK_GEN are the digital clock signals called (CLK166, CLK250, CLK100) and a status signal called clks_locked. Note from line 64 that the CLK250 output of CLK_GEN is also the CLK input of TWO_BIT.vhd. Other outputs of CLK_GEN are not yet used in TOP.vhd.

So, all the components found in our example project, my_proj1, are now tied together using the top-level component called TOP.vhd. Also, we've updated the project constraints file, **constraints1.xdc**, with entries that specify which FPGA pins connect to the inputs and outputs of TOP.vhd. Let's run synthesis and see what happens!

10.2 Running Synthesis

Before running synthesis, you should be aware that synthesis settings exist and that these settings may need tweaking for larger projects where synthesis is having trouble. These settings are accessed by clicking on the line that says "Settings" in the Vivado Flow Navigator (see Fig 3.1). For now, we will use the default settings for synthesis. Vivado synthesis is run by simply clicking on the line in the Vivado Flow Navigator that says "Run Synthesis".

While synthesis is running, it will produce messages and write them into the Vivado Log window (see "Log" tab at the bottom of the Vivado screen). These messages can be classified into five groups: 1) Errors, 2) Critical Warnings, 3) Warnings, 4) Information (Info), and 5) Status. Generally, you must investigate and resolve messages that are errors or critical warnings. Often, these message types refer to syntax errors (eg. you forget a comma) in your HDL. Warning messages can generally be ignored, although I once read through them faithfully. In recent years, Xilinx has adopted a philosophy that more warnings are better. For example, one of my projects had 20 synthesis warnings (all ignorable) in Vivado v2014.2. The same project had over 500 synthesis warnings in Vivado v2018.3! I found that these over-500 warnings are ignorable, but sadly I don't read through them very often. Finally, and unless you have a lot of free time, you can ignore the information and status messages produced by synthesis. A good way to sort through all the messages in the Log window is to use the Messages window (see "Messages" tab at the bottom of the Vivado screen). There you will find check boxes that allow you to select messages of one or more types. In Fig 10.3 is an example of the Vivado Messages window showing some errors that I intentionally created in TOP.vhd (by erasing clks_locked in line 23

and adding a semicolon at the end of line 33). In Fig 10.3, clicking on the part of a message that is blue-colored and underlined takes me directly to the line of HDL referenced by the message. However, you'll notice that my intentional errors had far-reaching effects (ie. causing error messages for lines 51 and 58, even though these lines are error-free).

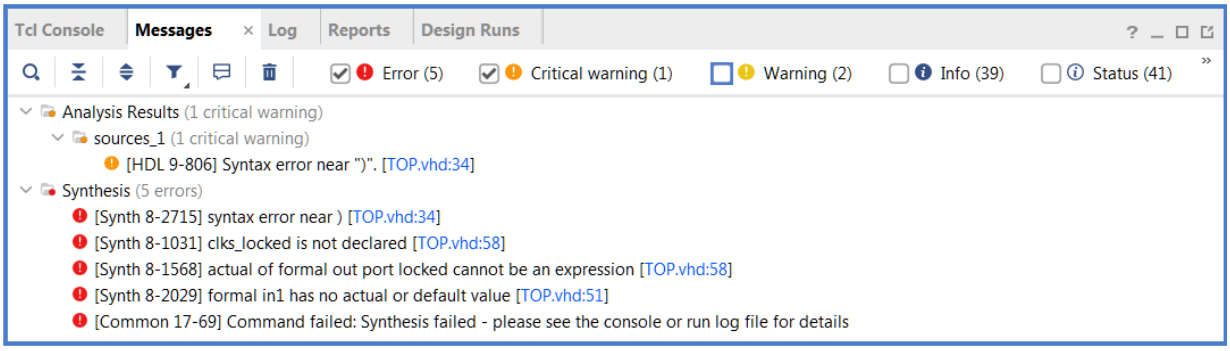

Fig 10.3 Vivado Messages window showing synthesis errors and critical warnings for the Vivado example project called my_proj1.

After getting a synthesis run that is free of errors and critical warnings, we can again go to the Flow Navigator and click on the line that says, "Open Synthesized Design". Then, as shown in Fig 10.4, we go to the Netlist tab, right-click on our component called TWO_BIT and select "Schematic". Almost magically, the digital circuit schematic shown in Fig 10.5 for TWO_BIT.vhd will be displayed!

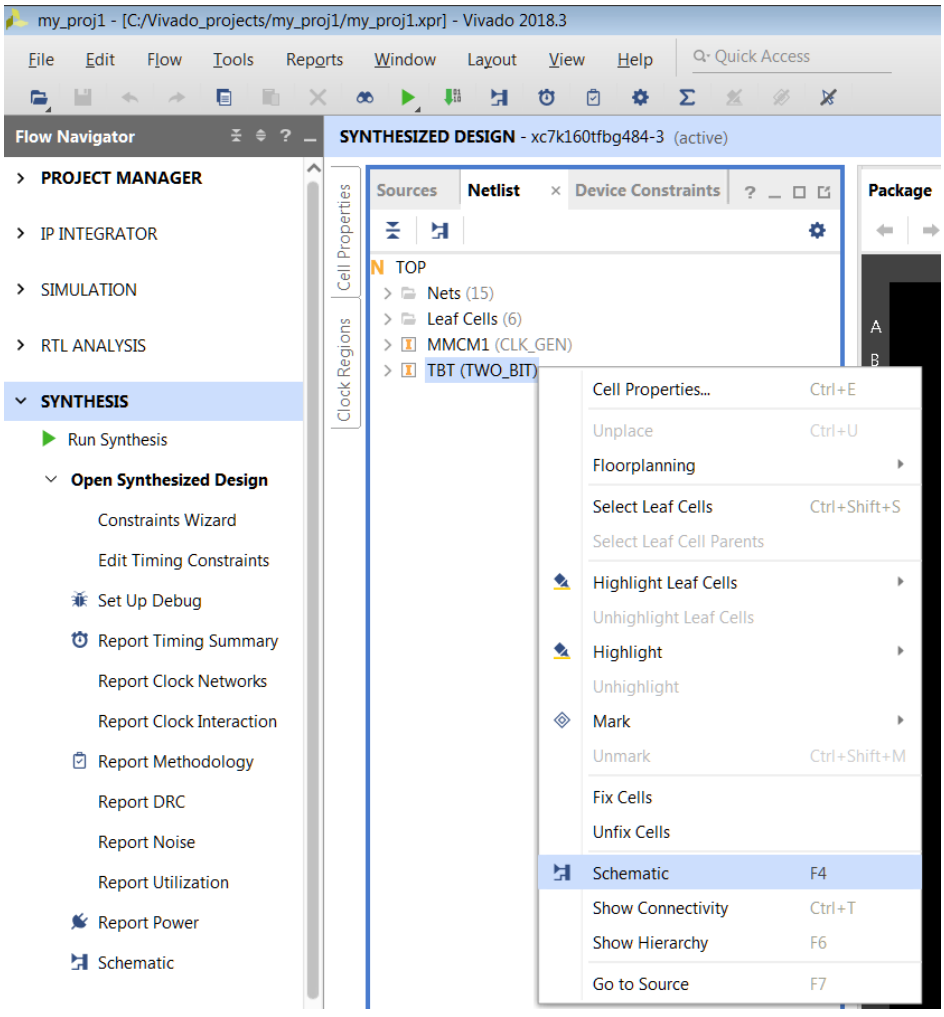

Fig 10.4 Obtaining a schematic for component, TWO_BIT.vhd, in the Vivado example project called my_proj1.

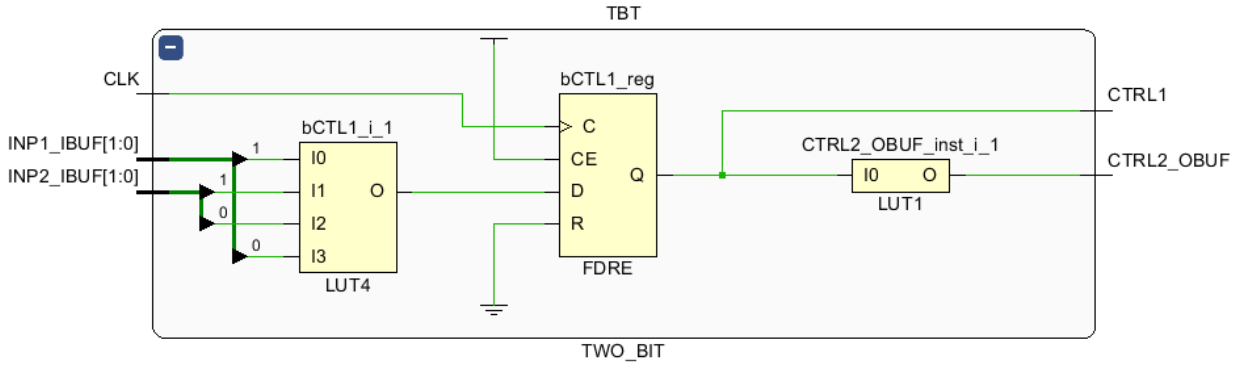

Fig 10.5 Schematic for component, TWO_BIT.vhd, in the Vivado example project called my_proj1.

It is OK for you to say WOW (I'm sure Mr. Klingman would). Well, maybe you are looking at Fig 10.5 and saying wow (not WOW). Maybe you are using a lower-case wow because the Fig 10.5 schematic is pretty simple and maybe because you're saying, "What's a LUT?". I'll answer that question in section 10.3 and hopefully get us back to upper-case WOW.

10.3 FPGA Anatomy

Inside the FPGA, you will find lots of digital components. These components get selected and connected according the instructions found in your HDL. This would be a good time for you to review section 7.5 where we discussed this HDL-to-Hardware connection. After reading this section, you will understand why Fig 10.5 is conceptually identical to Fig 7.4. Here is a list of the more common digital components that you will see in schematics produced by the synthesis tool found in the Vivado IDE.

- **Register** (aka flip-flop): Many of the signals that you declared in your VHDL will be synthesized into registers. For example, the signal called bCTRL1 from line-22 of Fig 4.1 (the VHDL for TWO_BIT.vhd) was synthesized into the register called bCTRL1_reg that is found in Fig 10.5. More precisely, the full netlist name assigned by Vivado is TBT/ bCTRL1_reg, because this register is contained in the instantiation called TBT of TWO_BIT.vhd (see line-62 of Fig 10.1). Throughout this book, I will follow what is done by the Vivado IDE and append "_reg" to a VHDL signal name when I want to talk about the digital register associated with the VHDL signal.
- **LUT** (**L**ook **U**p **T**able): Logic gates (eg. NAND, NOR, inverter) are perhaps the most familiar components in digital electronics. However, you will rarely find logic gates in a schematic produced by synthesis! Why? Because, Vivado synthesis and Xilinx FPGAs use a LUT to implement almost every type of logic gate. *In fact, synthesis uses LUTs for many things, including shift-registers, digital multiplexers, memory, and arithmetic!* An example is the four-input LUT (LUT4) called bCTRL1_i_1 found in Fig 10.5. This LUT4 performs the arithmetic and logic operations found in lines 31-36 of Fig 4.1. We can use Vivado to view the look-up table associated with this LUT4. As shown in Fig 10.6, I have simply clicked on the LUT4 called bCTRL1_i_1 (highlighted in blue) and then clicked on the "Truth Table" tab in the "Cell Properties" window. Of course, we almost never need to view or modify the look-up table associated with a LUT, but it is cool to know that Vivado lets us do this.
- **RAM and ROM** (distributed): The distributed Random Access Memory (RAM) and Read-Only Memory (ROM) found in Xilinx FPGAs is constructed from LUTs and is sometimes called LUTRAM and LUTROM.

For the Xilinx Kintex-7 FPGA that I am using, the components listed above are packed into a building-block called a *slice*, and this slice is repeated many thousands of times throughout the FPGA. These slices and the digital components they contain are sometimes called the *fabric* of the FPGA. When two components are found in the same slice then we know that the two components are located physically near each other. When we talk about synchronizers in chapter 12, we will need to know if components are physically located near each other.

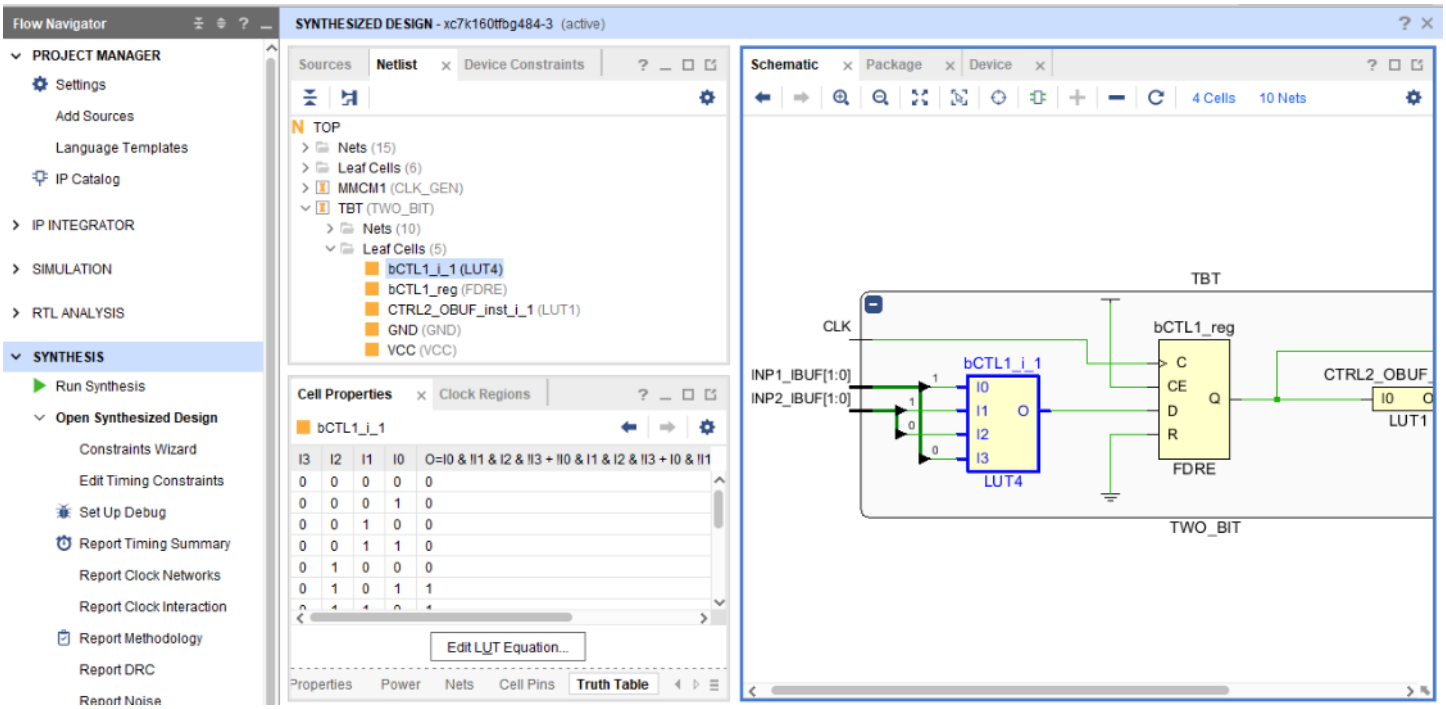

Fig 10.6 Viewing the look-up-table for the LUT4 called bCTRL1_i_1.

Found in lesser number than the slice components are other components distributed throughout the FPGA. Here is a list of the more common non-slice components.

- **DSP** (Digital Signal Processor): These blocks are used for arithmetic that is beyond the capability of LUTs.
- **Clock-module** (MMCM): As discussed in chapter 8, digital clocks are important for FPGA work. In the Xilinx Kintex-7 FPGA, I will usually manage and create clocks using a Mixed Mode Clock Manager (MMCM), as we discussed in chapter 9.
- **Clock Tree:** The clock tree is not actually a component, nor will you see it in a schematic from synthesis. As discussed in chapter 8, the clock tree is a set of high-speed paths throughout the FPGA that carry clock signals from the clock-module (MMCM) to digital registers and other components that need a clock input.
- **Clock Buffer:** These components are used to route digital clocks into and through the clock tree.
- **RAM** (block): Large amounts of Random Access Memory (RAM).
- **Input/Output Buffers and Devices:** There are blocks of digital components grouped near each of the FPGA pins. Some of these components are buffers designed to protect the delicate circuits inside the FPGA from the (sometimes crazy) voltages that we apply to the FPGA pins. Also, FPGA pins are designed to have many functions (eg. they can be either an input or an output) that we control with our HDL. So, some of the components grouped near the FPGA pins are needed to implement this multifunctionality. We'll talk more about the components located near each FPGA pin in chapter 15.

10.4 Optimization

Someday, you'll be looking at schematics produced by synthesis (as we did in section 10.2) and you will feel certain that synthesis has made the error of omitting your HDL signals from the design. You may spend lots of time trying to figure out what happened while cursing and doing other things that are out of character – unless you've already read this section.

In short, synthesis can (and does by default) perform several types of *optimization*. The types of optimization most likely to confuse you are called *LUT-combination* and *register-merging*. I will discuss these types of optimization using Fig 10.7

and Fig 10.8. First, some background information. Let's suppose that your absent-minded boss assigns you and your co-worker, Joe, the exact same task. That is, your boss wants a simple VHDL component that accepts two one-bit inputs (called SIG4 and SIG5), does some processing on them, and produces a one-bit output. *By themselves*, your component (called MY_COMP) and Joe's component (called JOE_COMP) will synthesis to become a LUT and a register as shown in Fig 10.7. Fig 10.7 also shows how the inputs, (SIG4, SIG5), to MY_COMP and JOE_COMP come from other LUTs and other signals, (SIG1, SIG2, SIG3).

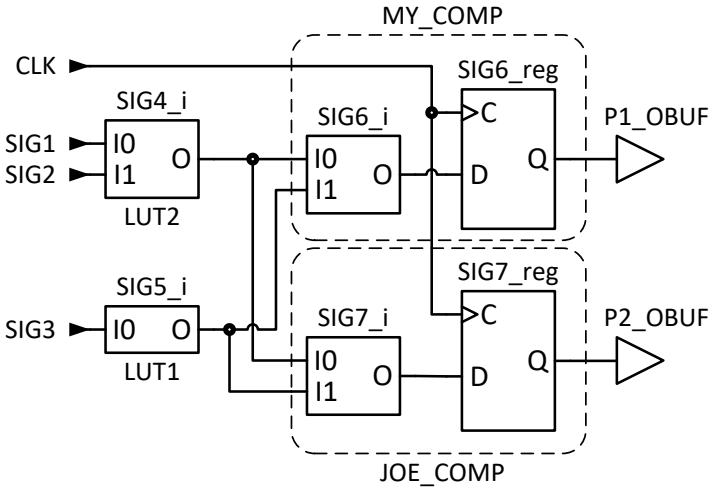

Fig 10.7 Example circuit produced by synthesis before optimization.

Next, somebody declares and instantiates MY_COMP and JOE_COMP into your workgroup's VHDL project. Synthesis is run on the entire project and produces the circuit shown in Fig 10.8. Everyone is looking at the circuit. Accusations and questions abound! Joe's work, JOE_COM, is completely gone! Also, signals called SIG4 and SIG5 are gone! You and Joe were the only one's using these signals so you must be at fault! Chaos reigns!!

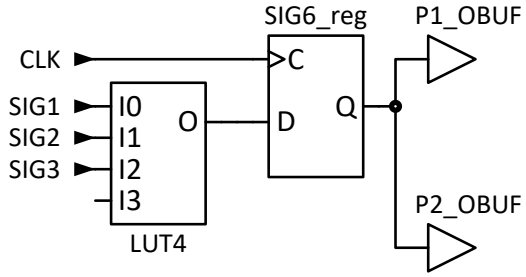

Fig 10.8 The example circuit from Fig 10.7 after synthesis optimization.

The answer that should calm everyone is that synthesis optimization was doing its job and has made things better for you. First, unlike your absent-minded boss, synthesis optimization recognized that the VHDL components written by you and Joe were functionally identical. So, synthesis got rid of Joe's component (sorry Joe). In the process of getting rid of Joe's work, synthesis used *register-merging* to combine the parallel registers called SIG6_reg and SIG7_reg, splitting the output of SIG6_reg so that both P1_OBUF and P2_OBUF are still driven by the correct signal. Next, synthesis optimization realized it could reduce circuit size and perhaps reduce power consumption if it used one big LUT instead of four small LUTs to do the necessary combinational logic. So, *LUT-combination* was used, resulting in both the LUT4 shown in Fig 10.8 and in the elimination of the signals called SIG4 and SIG5.

In the above example, it is important to note that LUT-combination will combine LUTs that are either in parallel or in series. However, register-merging will only merge registers that are in parallel. Also, note that register-merging increases the output *fanout* of registers that remain after the merge.

Once your workgroup gets past the confusion caused by not understanding synthesis optimization, there may be some real issues to address. These real issues will be discussed when we return to the topic of synthesis optimization in section 14.6.

10.5 Perspective

After running synthesis, do we really need to open the synthesized design and look at the digital circuits that were produced? Usually the answer is NO, especially if your HDL passed simulation testing (see chapter 6) and synthesis gave no critical warnings or errors. However, if your background/experience is more in hardware than software, then looking at the circuits can be comforting and fun. Regardless of your background, when errors and critical warnings arise then we must often "think hardware" and look at the circuits produced by synthesis. We'll talk a lot more about "think hardware" in the upcoming chapters.

11. Implementation and Level-1 Timing Analysis

We are now ready to talk about the next step in the IDE flow called implementation. In short, all the digital circuits that synthesis generated from your HDL are mapped into the FPGA hardware during implementation. Sometimes implementation is called "placement and routing" because it spends a lot of time selecting component positions inside the FPGA and routing signals and clocks to these components (ie. creating complete digital circuits). Most importantly, implementation does all this while ensuring that everything passes *timing analysis*. As briefly discussed in section 7.6, timing analysis refers to checks done by the IDE that determine whether some of the FPGA digital circuits could become metastable or suffer from glitches.

When there are problems during implementation, they are usually problems identified by timing analysis. In the rest of this book, we will talk much about timing analysis and about solving problems identified by timing analysis.

11.1 You Are Special

When implementation reports that our HDL has resulted in a FPGA configuration that passes timing analysis then we've reached a very happy place called *timing closure*. However, we sometimes achieve timing closure for the wrong reasons (a sad place). That is, timing analysis sometimes needs information from us, which we provide by writing *timing constraints*. If we omit or use the wrong timing constraints, then we might achieve timing closure for the wrong reasons. Also, some of the digital circuits that result from our HDL need not undergo timing analysis. If these circuits are failing timing analysis, then we can often get back to the happy place (timing closure) by writing *timing exceptions.* Timing exceptions tell timing analysis not to analyze certain circuits or to analyze them in a special way. Finally, your HDL can be logically perfect (ie., it can pass the simulation discussed in Chapter 6) but it can fail timing analysis.

If you are frightened by the above paragraph, then I am sorry to say that you should be (a little). Perhaps it is better to say that you feel challenged rather than frightened? Anyway, being able to understand and deal with problems reported by FPGA timing analysis is what makes HDL programmers *truly special* in the programming world. You will get no help from C-programmers and Java-programmers on solving timing analysis problems. They will have no idea what you are talking about. However, you will get help from your fellow HDL programmers and hopefully from this book. Remember, we're all in this together.

11.2 Levels of Understanding

We often understand complex subjects on two levels. The lower level (let's call it Level-1) consists of a few important concepts that we carry around and use in our daily work. The higher level, Level-2, understanding has all the gritty details. We usually can't remember the Level-2 stuff, but we know where to look for it and when we see it again, we can usually remember what it means. In the rest of this chapter, we will discuss a Level-1 understanding of FPGA timing analysis. In chapter 14, we'll discuss Level-2 timing analysis.

11.3 Level-1 Timing Analysis

11.3.1 Inside and Outside: For Level-1 understanding, we will restrict our discussion to timing analysis for circuits that lie entirely inside the FPGA. It is possible to run timing analysis on circuits that extend from the FPGA to electrical components located outside the FPGA. However, you must specifically tell timing analysis to analyze these outside circuits using special Input/Output (I/O) timing constraints. Outside circuit timing analysis is Level-2 stuff that we will discuss in chapter 15.

11.3.2 Setup-WNS and Hold-WNS: After you use the IDE to run implementation for your project, you will get a report containing two numbers that summarize the results of timing analysis. These two numbers are called *setup worse negative slack* (or *setup-WNS*) and *hold worse negative slack* (or *hold-WNS*). When both setup-WNS and hold-WNS are positive then you have achieved timing closure (the happy place). I know, I know "worst *negative* slack being *positive*" is confusing, but this wording has become a standard in the FPGA world. However, from now on, I will (mostly) stop saying WNS and instead use the word, *slack*.

A small bit of good news is that we usually need not worry about hold-WNS. That is, under the restrictions of section 11.3.1, if we can somehow make setup-WNS positive then hold-WNS will usually be positive too. The bad news is that a negative value for setup-WNS is usually just the beginning of more bad news. That is, since setup-WNS is only the "worse" negative slack for setup, then other negative slack usually exists elsewhere in your design.

11.3.3 Slack Meaning: For Level-1 understanding, we can consider slack to be a grade for how well our *HDL handles the processing of signals*. A positive value for slack means that our HDL is awesomely good! A negative value for slack means that our HDL failed miserably. Here, "failed miserably" does not mean that the HDL is logically bad. In fact, HDL that passes simulation (see Chapter 6) can sometimes fail timing analysis! Instead, "failed miserably" means that when our HDL is synthesized (see Chapter 10) into digital circuits, then these circuits will not work properly. The good news is that small tweaks to your HDL can often turn negative slack into positive slack.

11.3.4 Be Positive!: For Level-1 understanding, we don't need to know how the slack numbers are calculated. Instead, we will focus on the meaning of slack from the HDL viewpoint (instead of from the digital hardware viewpoint). I defined slack in the preceding section as a grade for how well our HDL handles the processing of signals. So, when timing analysis reports negative slack for a signal, you need to know 1) which signal, and 2) how did the HDL mishandle processing of the signal?

The answer to "Which signal?" is easy. The Vivado IDE will provides a way (see "Open Implemented Design" on the left of Fig 3.1) for you to open the implementation of your HDL. There you will find a list of circuit snippets (formally called *timing paths*) that have failed timing analysis. Timing paths typically extend from one digital register to another digital register – similar to the circuit from S2_reg through A2 to S3_reg shown in Fig 7.1. The name for one of the registers on the failed path will usually contain the name of HDL signal that you have mishandled. For example, if you mishandled processing of your HDL signal called S3, then one of the registers in the failed path will contain the word, S3_reg.

The question of how our HDL mishandled processing of a signal is usually answered by one or more of the following statements:

1) ***Too Much Processing***: We tried to do too much processing of the signal inside an HDL clocked process.
2) ***Clock Crossings***: We are calculating a signal in a clocked process that uses clock, CLK1, and then using the signal in another clocked process that uses clock, CLK2 – and CLK1 is not equal to CLK2. This is also called crossing a signal from the CLK1 *clock-domain* into the CLK2 clock-domain.
3) ***FPGA Input/Output (I/O) Problems***: Data coming into or going out of the FPGA has improper skew with respect to the FPGA I/O clock.

Corrections for "Too Much Processing" are discussed in the section 11.4. Corrections for "Clock Crossings" are discussed in chapter 12. Corrections for "FPGA I/O Problems" are discussed in Chapters 15 and 16.

11.4 Too Much Processing

When the IDE reports that a path has failed timing analysis and this path involves a signal on which your HDL does lots of processing then perhaps you are faced with the "too much processing" problem. Specifically, "too much processing" means that your HDL is trying to do too much processing in one clock cycle. The solution is to spread the processing out over multiple clock cycles. All of this will become clear as we discuss the HDL shown in Fig 11.1.

The VHDL component in Fig 11.1 is called PIPE_TEST.vhd. This name is derived from the word, *pipelining*, which is FPGA lingo for "spreading the processing out over multiple clock cycles". Using input, IN1, the PIPE_TEST component performs the simple calculation, [5 * (IN1 + 2)]. This calculation is done in four different ways and the four results are placed in outputs, OUT1, OUT2, OUT3, and OUT4 of PIPE_TEST.vhd. In Fig 11.1, the clocked process called MY_PRC1 is the straightforward approach to doing this simple calculation. The output from MY_PRC1 is placed in OUT1. For this example, we will assume that the path involving OUT1 has failed timing analysis because of the "too much processing" problem.

For our simple calculation, [5 * (IN1 + 2)], the processes called MY_PRC2A and MY_PRC2B in Fig 11.1 show a way to "spread the processing out over multiple clock cycles". On one cycle (let's call it CYCLE1) of the clock called CLK1, the process called MY_PRC2A calculates (IN1 + 2) and stores the result in signal, TMP2. On the next cycle, CYCLE2, of CLK1, the process called MY_PRC2B calculates (5 * TMP2) and places the result in OUT2. Note that the value of OUT2 is identical to the value that MY_PRC1 calculated and placed in OUT1. However, the processing needed to calculate OUT2 is now spread out over two cycles of CLK1. Note that MY_PRC2B calculates a value for OUT2 during CYCLE1 of CLK1. This value of OUT2 is undetermined unless we know the value of TMP2 that was calculated by MY_PRC2A during the cycle of CLK1 prior to CYCLE1 – which brings up an interesting question! What happens if we feed a new value to IN1 on every cycle of CLK1? Well, on every cycle of CLK1, we get a new value for OUT2 that is calculated using the value of IN1 from two cycles of CLK1 prior to the current cycle. So, the throughput for OUT1 and OUT2 is the same (ie. a new value is calculated during every cycle of CLK1). The only difference being that after a value is assigned to IN1, the associated value of OUT1 is ready after 1-cycle of CLK1 whereas the associated value of OUT2 is ready after 2-cycles of CLK1. In FPGA lingo, we say that calculation for OUT1 has *latency* of 1-cycle of CLK1 and calculation for OUT2 has *latency* of 2-cycles of CLK1.

In Fig 11.1, it may surprise you that the process called MY_PRC3 pipelines our simple calculation, [5 * (IN1 + 2)], in exactly the same way as the combined efforts of MY_PRC2A and MY_PRC2B. If this equivalency is not apparent to you, then please reread section 4.4. Then, note that the value of TMP3 used in line-51 does not come from the value that was just calculated in line-50. Rather, the value of TMP3 used in line-51 come from that value that was calculated in line-50 during the execution of MY_PRC3 that occurred on the previous cycle of CLK1.

Finally, just to make sure you're paying attention, look at the process called MY_PRC4. After studying it (and perhaps rereading section 4.4), you should conclude that MY_PRC4 is equivalent to MY_PRC1 *(Hint: because TMP4 is a variable and not a signal)*. That is, both processes attempt to perform the simple calculation, [5 * (IN1 + 2)], in one cycle of CLK1.

The previous two paragraph are important! Make sure you understand them and that you understand the differences between VHDL signals and variables (ie. that you understand section 4.4).

In conclusion, if the calculation of [5 * (IN1 + 2)] done in MY_PRC1 is failing timing analysis because this process was trying to do too much processing in one cycle of CLK1, then pipelining the calculation may solve the problem. Examples of pipelining the calculation are given by process, MY_PRC3, and by the combined efforts of processes, MY_PRC2A and MY_PRC2B. Like MY_PRC1, the process called MY_PRC4 attempts to perform the calculation in one cycle of CLK1. Thus, if calculation of OUT1 is failing timing analysis then calculation of OUT4 will also fail timing analysis.

```
01  ----------------------------------------------------------
02  -- Name:        PIPE_TEST.vhd
03  -- Description: pipeline testing
04  -- Date: 14Feb2021
05  -- Revisions:   none
06  ----------------------------------------------------------
07  library IEEE;
08  use IEEE.STD_LOGIC_1164.ALL;
09  use IEEE.NUMERIC_STD.ALL;
10
11  entity PIPE_TEST is
12    port(
13          CLK1    : in std_logic;              --logic clock
14          IN1     : in unsigned(7 downto 0);   --input #1
15          OUT1    : out unsigned(11 downto 0); --output #1
16          OUT2    : out unsigned(11 downto 0); --output #2
17          OUT3    : out unsigned(11 downto 0); --output #3
18          OUT4    : out unsigned(11 downto 0)  --output #4
19          );
20  end PIPE_TEST;
21
22  architecture MY_CMP1 of PIPE_TEST is
23      signal TMP2, TMP3 : unsigned(7 downto 0);
24
25  begin
26      MY_PRC1: process(CLK1)              --outputs: (OUT1)
27          begin
28              if rising_edge(CLK1) then
29                  OUT1 <= x"5" * (IN1 + x"02");
30              end if;
31      end process MY_PRC1;
32
33      MY_PRC2A: process(CLK1)             --outputs: (TMP2)
34          begin
35              if rising_edge(CLK1) then
36                  TMP2 <= IN1 + x"02";
37              end if;
38      end process MY_PRC2A;
39
40      MY_PRC2B: process(CLK1)             --outputs: (OUT2)
41          begin
42              if rising_edge(CLK1) then
43                  OUT2 <= x"5" * TMP2;
44              end if;
45      end process MY_PRC2B;
46
47      MY_PRC3: process(CLK1)              --outputs: (TMP3,OUT3)
48          begin
49              if rising_edge(CLK1) then
50                  TMP3 <= IN1 + x"02";
51                  OUT3 <= x"5" * TMP3;
52              end if;
53      end process MY_PRC3;
54
55      MY_PRC4: process(CLK1)              --outputs: (OUT4)
56          variable TMP4 : unsigned(7 downto 0);
57          begin
58              if rising_edge(CLK1) then
59                  TMP4 := IN1 + x"02";
60                  OUT4 <= x"5" * TMP4;
61              end if;
62      end process MY_PRC4;
63  end MY_CMP1;
```

Fig 11.1 VHDL component called PIPE_TEST.

11.5 Timing Constraints

In section 4.6 we discussed how the Tcl language is used to write *physical constraints*, which are placed in the *constraints file* that we are calling **constraints1.xdc**. As you may recall, one type of physical constraint tells the IDE how the inputs and outputs of the top-level HDL component connect to the physical pins of the FPGA. This mapping between

words in our HDL code and pins on the FPGA is information that the IDE needs for implementation (ie. for planning the layout of our HDL-described digital circuit inside the FPGA).

Similarly, before the IDE can perform timing analysis, it needs certain timing information from us. Again, we provide this information using Tcl to write *timing constraints* into the file called **constraints1.xdc**. For example, the IDE always needs a timing constraint that specifies the period of each logic clock (eg. a 100MHz clock has a period of 10ns). Below, I'll talk more about this clock constraint and other timing constraints.

In the old days, there was only one constraints file for the IDE project and constraints were manually typed into this master constraints file. Newer IDEs still want you to create and maintain a master constraints file. However, the newer IDEs will sometimes automatically create other constraints files and write constraints into them. For example, when we used the Vivado IDE in Chapter 9 to setup the clock-module for our project, the IDE automatically created the small constraints file called **CLK_GEN.xdc** that contains the timing constraints shown in Fig 11.2. Without getting into too much detail, the line-06 constraint tells the IDE that a clock signal with a period of 10.0ns comes into the FPGA via port, clk_in1_p, of the CLK_GEN component (see lines 28 and 53 of Fig 10.1). The line-07 constraint tells the IDE that this clock signal has jitter of 0.014ns Pk-Pk (= 14ps Pk-Pk). See section 8.4 for more about clock jitter. When the IDE automatically creates a separate constraints file and places constraints into this file then we should not duplicate these constraints in our master constraints file.

We'll postpone further discussion of timing constraints until chapters 15 and 16, since writing them usually requires a Level-2 understanding of timing analysis – and we're now working on Level-1 understanding.

```
01  # file: CLK_GEN.xdc
02  #
03  # (c) Copyright 2008 - 2013 Xilinx, Inc. All rights reserved.
04  #------------------------------------------------------------------
05  #
06  create_clock -period 10.000 [get_ports clk_in1_p]
07  set_input_jitter [get_clocks -of_objects [get_ports clk_in1_p]] 0.014
```

Fig 11.2 Timing constraints found in the automatically-generated constraints file called CLK_GEN.xdc.

11.6 Timing Exceptions

Closely related to timing constraints are *timing exceptions*. Again, we use the Tcl language to write timing exceptions and we place them in the master constraints file that we are calling **constraints1.xdc**. We use timing constraints to describe facts (eg. clock period) to the IDE. We use timing exceptions to guide timing analysis and to help it achieve timing closure. Timing constraints are often easy to write whereas timing exceptions are more difficult and can get us into trouble. At the beginning of this chapter, I mentioned that we sometimes achieve timing closure for the wrong reasons. Well, those wrong reasons can be the result of writing incorrect timing exceptions (and incorrect timing constraints). In later chapters, I will often mention timing exceptions/constraints and point out where they are needed. However, I haven't yet given you enough information to write timing exceptions/constraints. So, please stay-tuned until chapters 14-17, where we'll write lots of them.

11.7 Not Yet Done

In this chapter, we talked in general terms about implementation. Implementation is launched by a click-of-the-mouse on one of the items found in the IDE flow and often runs without problems. However, when problems occur with implementation, they usually involve the timing analysis portion of implementation. Specifically, timing analysis will report a negative value of slack for a path associated with a problematic signal in your HDL. Timing analysis problems often fall into one or more of the three categories listed in section 11.3.4. As explained in section 11.4, the method of pipelining helps solve timing analysis problems that fall into the "Too Much Processing" category. Solutions for problems that fall into the other two categories will be given in other chapters of this book.

I suspect that some readers will criticize this chapter for not talking more about the calculation of slack and about FPGA hardware aspects of timing analysis. However, as I explained in section 11.2, this detailed understanding is not something that we must "carry around" to do our daily FPGA work. Rest assured, we are not yet done with timing analysis. After reading the rest of this book, I suspect you'll have seen quite enough of timing analysis.

12. Multiple Clock Problems

In this chapter, we'll talk about solving timing analysis problems that fall into the "Clock Crossings" category mentioned in section 11.3.4. From my experience, most timing analysis problems fall into this category.

You'll recall from chapter 7 that clocks are needed for the clock-logic hardware used throughout the FPGA. If all FPGA projects used only one clock (instead of multiple clocks) then our life would be so much easier. So, why use multiple clocks and what problems do they cause? -please read on.

12.1 Why Multiple Clocks?

Despite our awesome intelligence and worth, the result of our FPGA is usually part of a large system. So, we are often told by our boss (the system engineer) what clocks to use. Sometimes our FPGA must talk with an external device that requires a specific clock speed. In which case, that stupid external device becomes our clock boss. Finally, we sometimes inherit a project that already has lots of clocks and no one remembers why they're all needed. My point being that factors beyond our control often cause our FPGA projects to use multiple clocks. However, when you are allowed to make decisions about clocks, then here are some rules to follow:

1) Fewer clocks are better.
2) Slower clocks (ie. clocks with a longer period) are better. Slower clocks make HDL coding easier, mostly because it will be easier for your HDL to pass timing analysis.
3) Synchronous clocks are better (than asynchronous clocks). Generally, we get synchronous clocks by bringing only one base clock into the FPGA and then using an FPGA clock-module (see chapter 8 and 9) to derive all needed clocks from the base clock.

I suspect some of you will disagree with me on 3), but we'll go with it for now. In the next section, we'll talk about problems resulting from the use of multiple clocks inside the FPGA.

12.2 Metastability and Clock-Crossings

Problems caused by using multiple clocks generally result from operational limitations of the digital register that we discussed in section 7.2. Specifically, a register has time-requirements for setup, T_{su}, and hold, T_h. That is, the signal coming into the D-input of the register must be stable for T_{su} seconds before and for T_h seconds after the clock input (ie. the input to pin-C) transitions from low-to-high. If these register-requirements for setup and hold are not met, then the Q-output of the register can enter a *metastable* state. As mentioned in section 7.6, metastable means that the Q-output of the register can become unstable until the next rising-edge of the clock reaches the register's C-input. Of course, register metastability can ruin everything the FPGA circuit is trying to do (and everything that your HDL code associated with the register is trying to do).

So, how is all this setup and hold stuff related to multiple clock problems? I'll answer this question using the example VHDL shown in Fig 12.1. There, the process called PRC1, which is clocked by CLK1, calculates a value for signal, SIG1, from the signal, SIG0. Also, the process called PRC2, which is clocked by CLK2, calculates a value for signal, SIG2, from the signal, SIG1. According to our "really important concepts" from section 7.5, the implemented design for our example will contain digital registers corresponding to signals, (SIG1, SIG2), and will contain combinational logic corresponding to processing done on SIG1 as shown in Fig 12.2. In FPGA lingo, the register-to-register transfer of data shown in Fig 12.2 where each register uses a different clock is called a *clock-crossing*. Also, in FPGA lingo, we sometimes

say that SIG1 is being crossed from the CLK1 *clock-domain* into the CLK2 *clock-domain*. Finally, in answer to the question at the start of this paragraph, setup and hold violations leading to metastability can occur at the clock-crossings that results from use of multiple clocks.

```
01   signal SIG0, SIG1, SIG2 : std_logic;
02  ...
03   PRC1: process(CLK1)            --outputs: (SIG1)
04       begin
05           if rising_edge(CLK1) then   --CLK1-domain
06               SIG1 <= NOT(SIG0);
07           end if;
08   end process PRC1;
09
10
11   PRC2: process(CLK2)            --outputs: (SIG2)
12       begin
13           if rising_edge(CLK2) then   --CLK2-domain
14               SIG2 <= NOT(SIG1);
15           end if;
16   end process PRC2;
```

Fig 12.1 VHDL for a crossing of SIG1 from the CLK1 domain to the CLK2 domain.

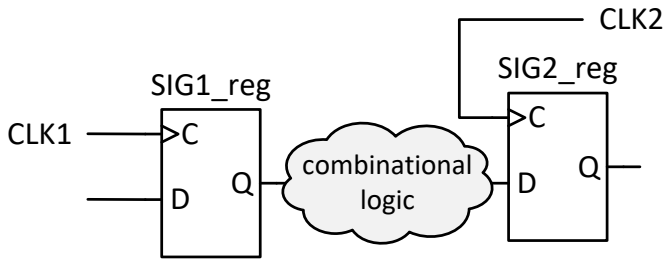

Fig 12.2 Implementation of VHDL from Fig 12.1 showing a clock-crossing.

How does a clock-crossing cause a metastability problem? We can answer this question by looking at the Fig 12.3 timing diagram of what happens in the example VHDL of Fig 12.1. At time, 0ns, process PRC1 sees a rising-edge for CLK1 and starts calculating a value for SIG1. PRC1 has the SIG1-result ready at time, 1.5ns, as indicated by the SIG1 transition at 1.5ns in Fig 12.3. Hence, the SIG1-result is stable for 2.5ns before the rising-edge of CLK2 at 4ns, which is when process PRC2 needs the SIG1-result. Next, consider what happens when PRC1 sees the rising-edge of CLK1 at 6ns and starts calculating a value for SIG1. PRC1 has the SIG1-result ready 1.5ns later (ie. at time, 7.5ns) which is only 0.5ns before PRC2 needs the result at time, 8ns (ie. on the next rising-edge of CLK2). So, in this example, the SIG1-result is sometimes ready 2.5ns before it is needed and is sometimes ready only 0.5ns before it is needed. -and sometimes, 0.5ns is small enough to cause a setup, T_{su}, violation at the SIG2 register. Since this setup violation can cause metastability, we have (by example) answered the question at the start of this paragraph.

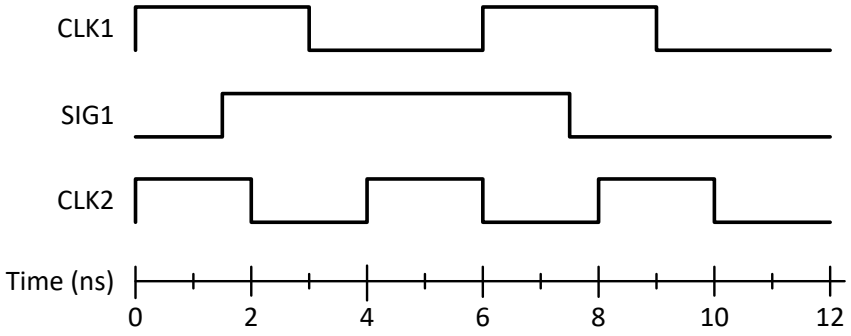

Fig 12.3 Timing diagram for a signal, SIG1, being crossed from the CLK1-domain to the CLK2-domain.

Rest assured that timing analysis is on the lookout for circuits/paths that have metastability problems and will report them to have negative slack (see section 11.3 for discussion of negative slack). Further, some IDE have special reports that identify all clock-crossings and then highlight those crossings that have negative slack. So, we now know that multiple clocks can lead to clock-crossings and some of these crossings can have metastability problems. In the next section, we'll discuss the proper way to handle a clock-crossing that has the metastability problem.

12.3 One-Bit Synchronizers

As discussed in the previous section, timing analysis reports negative slack for any clock-crossing where metastability might occur. Fortunately, these problem crossings can often be repaired using a standard digital circuit called a *synchronizer* [Cummings, 2008], which is easy to create using HDL. The strategy of the synchronizer is to contain rather than prevent metastability. That is, once a synchronizer component is inserted at the clock-crossing, metastability will occur only inside the synchronizer where it will seldom do harm.

We'll explain the synchronizer concept by using a *2-flip-flop* (2FF) synchronizer to repair the 1-bit clock-crossing found in the example VHDL of Fig 12.1. The name of this synchronizer comes from that fact that its construction uses two flip-flops (ie. two digital registers) as shown by the VHDL in Fig 12.4 and by the circuit in Fig 12.5. In Fig 12.5, both registers of the 2FF-synchronizer use a clock called CLK. However, the input, IN_2FF, to the 2FF-synchronizer comes from a clock-domain that does not use CLK. Thus, in Fig 12.5, setup and hold violations can occur at the input to meta_2ff and the output of meta_2ff_reg can become metastable. However, the theory of the synchronizer circuit states that this metastability will very likely settle to a valid logic state (1 or 0) before the next rising-edge of CLK. Hence, on the next rising-edge of CLK, the second register of the synchronizer called OUT_2FF_reg will "see" a valid logic state at its D-input and pass this valid input to its Q-output. The result being that the problematic metastability is contained between the two registers of the synchronizer and the output of the synchronizer is *very likely* to be a valid logic state.

```
01  ------------------------------------------------------------
02  -- Name:        SYNC1_2FF.vhd
03  -- Description: 1-bit two-flip-flop (2FF) synchronizer
04  -- Date:        14Feb2021
05  ------------------------------------------------------------
06  library IEEE;
07  use IEEE.STD_LOGIC_1164.ALL;
08
09  entity SYNC1_2FF is
10      port(
11          CLK     : in std_logic;     --clock in the receiver domain
12          IN_2FF  : in std_logic;     --signal coming from sender domain
13          OUT_2FF : out std_logic     --stable copy of IN_2FF for receiver domain
14          );
15  end SYNC1_2FF;
16
17  architecture MY_CMP1 of SYNC1_2FF is
18      signal meta_2ff : std_logic;
19
20      attribute ASYNC_REG : string;
21      attribute ASYNC_REG of meta_2ff,OUT_2FF: signal is "TRUE";
22
23  begin
24
25      PRC_2FF: process(CLK)          --outputs: (OUT_2FF)
26      begin
27          if rising_edge(CLK) then
28              meta_2ff <= IN_2FF;     --meta_2ff may be metastable,
29              OUT_2FF <= meta_2ff;    --but OUT_2FF will be stable
30          end if;
31      end process PRC_2FF;
32
33  end MY_CMP1;
```

Fig 12.4 VHDL component called SYNC1_2FF that is a 2-flip-flop synchronizer.

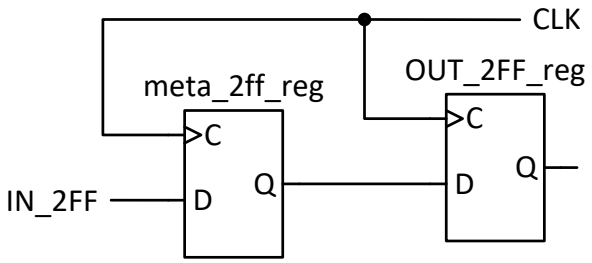

Fig 12.5 FPGA circuit corresponding to the VHDL in Fig 12.4 for a 2-flip-flop synchronizer.

Using the SYNC1_2FF.vhd component described by Fig 12.4, we can now repair the 1-bit clock-crossing shown in Fig 12.1. This repair is described by the VHDL in Fig 12.6, which differs from the VHDL in Fig 12.1 by the addition of signal, s_SIG1, and by the addition of both declaration and instantiation for SYNC1_2FF.vhd.

```
01  signal SIG0, SIG1, SIG2, s_SIG1 : std_logic;
02 ...
03    -- >> declare SYNC1_2FF at appropriate place in your VHDL <<
04 ...
05  PRC1: process(CLK1)          --outputs: (SIG1)
06      begin
07          if rising_edge(CLK1) then   --CLK1-domain
08              SIG1 <= NOT(SIG0);
09          end if;
10  end process PRC1;
11
12  S2FF: SYNC1_2FF                     --instantiation of SYNC1_2FF
13     port map(
14              CLK     => CLK2,
15              IN_2FF  => SIG1,
16              OUT_2FF => s_SIG1
17              );
18
19  PRC2: process(CLK2)          --outputs: (SIG2)
20      begin
21          if rising_edge(CLK2) then   --CLK2-domain
22              SIG2 <= NOT(s_SIG1);
23          end if;
24  end process PRC2;
```

Fig 12.6 VHDL for a crossing of SIG1 from the CLK1-domain to the CLK2-domain using synchronizer, SYNC1_2FF, from Fig 12.4.

Although the 2FF-synchronizer can contain the metastability problem associated with a clock-crossing, an uncertainty problem remains. This uncertainty problem results because we cannot be sure which logic-state the metastability will settle into. Continuing with our Fig 12.1 example from above will further explain this uncertainty problem. Suppose that on EDGE1 of CLK2 that SIG1 transitions from 0-to-1, causing a setup violation for register, meta_2ff_reg, and metastability at the output of this register. Between the times for EDGE1 and EDGE2 of CLK2, the metastable output of meta_2ff_reg will settle to either 1 or 0 (we cannot be certain which state). So, on EDGE2 of CLK2, OUT_2FF_reg will capture the uncertain output of meta_2ff_reg and pass it to the output of the synchronizer (ie. to the Q-pin of OUT_2FF_reg). At this point, you may be thinking that the 2FF-synchronizer is worthless since its output is uncertain! However, this is not true.

The key to making the 2FF-synchronizer work is to *hold SIG1 in each new state for at least two cycles of CLK2*. Holding SIG1 constant for two cycles of CLK2 means that if the input to meta_2ff_reg sees a changing value for SIG1 at EDGE1 (of CLK2), then it will see an unchanging value of SIG1 at EDGE2. Hence, at the EDGE2 time, the output of meta_2ff_reg will (very likely) be stable and be equal to the SIG1 logic state that we are trying to pass through the synchronizer. Thus, the SIG1 state that we are trying to pass through the synchronizer can reach the output of the synchronizer at either the EDGE2 time (if metastability just happens to settle in the correct state) or at the EDGE3 time. Again, the key to making

the synchronizer work properly is to ensure that SIG1 remains in each new state for at least two cycles of CLK2 (ie. two cycles of the clock used in the clock-domain that is receiving SIG1).

From the discussion above, you may have noticed that *the synchronizer will both slow-down and delay data transfer through the clock-crossing*. The slow-down results from the requirement that we change SIG1 no more often than once every other cycle of CLK2. The delay results from the fact that it takes an extra two cycles of CLK2 to get SIG1 through the two registers used to build the synchronizer. When using the 1-bit synchronizer, the associated slow-down and delay are sometimes not important. That is, the 1-bit synchronizer is often used as follows:

- A pulse having width of at least two-cycles of CLK2 is sent through a 2FF-synchronizer from the CLK1 domain to the CLK2 domain to tell the CLK2 domain to do something.
- When the CLK2 domain is done doing something, then the CLK2 domain sends a pulse having width of at least two-cycles of CLK1 through a separate 2FF-synchronizer to the CLK1 domain, indicating that it is done doing something.

However, the slow-down and delay associated with a *multibit* synchronizer is important and can cause problems as we'll discuss in section 12.5.

12.4 Synchronizer Failure

In the previous section, you will note that I use ambiguous words such as "very likely" and "uncertain". This wording is necessary because metastability settling-time has statistical properties. I also stated that metastability is expected to settle to a valid logic state in the 2FF-synchronizer during one cycle of CLK2. Usually, this settling occurs with very high probability (ie. the synchronizer has a very low probability of failure). All this probability stuff about the 2FF-synchronizer can be summarized using the following equation, which is explained in detail by *Ginosar[2011]*.

$$MTBF = t_C \cdot e^{\left(\frac{t_C - t_P}{\tau}\right)} / (f_D \cdot t_W) \qquad \textbf{(12.1)}$$

In equation 12.1,

- $MTBF$ = Mean Time(sec) Between Failure
- t_C = period (sec) of the clock used by the synchronizer,
- f_D = frequency (Hz) at which the data coming into the synchronizer is changing,
- τ, t_W = parameters of the flip-flops used to build the synchronizer,
    - τ = time (sec) constant of settling (out of metastability),
    - t_W = time (sec) width of window where incoming data to the flip-flop can cause metastability,
- t_P = time (sec) that it takes data to traverse the data path between flip-flops of the synchronizer.
- It is assumed that the time of each data-change is uniformly distributed over the interval, t_C. Hence, the probability that a particular data-change will cause metastability is equal to (t_W/t_C).

Note that a small value for MTBF is a bad thing because it means that failures will occur often. So, let's look at equation (12.1) and see what conditions cause the calculated value of MTBF to be small. Most noticeably, a fast clock (ie. small values of t_C) will cause MTBF to be small. Also, large values for τ, t_W, f_D, and t_P cause MTBF to be small.

The effect of parameter, t_P, on MTBF requires a little explanation. If the flip-flops of the synchronizer are located far apart then the travel time, t_P, of data between the flops will be large. So, if the data output from the first flop of the synchronizer goes metastable, then this metastable data will reach the second flop of the synchronizer in t_P seconds. Then, to ensure that metastability is contained within the synchronizer, the second flop must wait additional time for the metastability to settle before it samples and passes-on the data. So, as far as the second flop is concerned, the value of t_P adds to the time it must wait for metastability to settle. Hence, any value for t_P that is greater than zero makes it harder for the synchronizer to work properly. In the Xilinx Vivado IDE, we can ensure that implementation places the

synchronizer flip-flops close together (ie. makes t_P small) by setting the ASYNC_REG property for each register to TRUE. This is done by the VHDL attribute specifications shown in lines 20-21 of Fig 12.4.

Let's do some calculations using equation (12.1). We'll assume that f_C=100MHz (ie. t_C = 10ns), f_D=50MHz, τ= 50ps, t_W=50ps, and t_P=2ns. For reasons not entirely clear to me, it is difficult to get values for τ and t_W from FPGA vendors. However, the values I have provided are close enough for our rough calculations, which result in a value for MTBF of 3.9×10^{56} years. Well, that's a long time considering that the universe is estimated to be 13.8×10^{9} years old. So, we can expect the 2FF-synchronizer to be very reliable under these operating conditions. However, if we instead use f_C=200MHz (ie. t_C = 5ns) then MTBF drops to 7.3×10^{12} years. Finally, if we f_C=200MHz and t_P=4ns then MTBF drops to only 970 seconds!

Next, we need to talk a little about the assumption shown below equation (12.1) that "data-change is uniformly distributed over the interval, t_C". This is a reasonable assumption if the clock used to produce the data is truly asynchronous to the clock used by the flip-flops of the synchronizer. However, let's look at a worse case condition. Suppose these two clocks are actually synchronous *and* one is an integer multiple of the other. Then, for certain clock phasing, it is possible that *every* data-transition entering the synchronizer could occur on the rising-edge of the synchronizer clock. Hence, *every* data-transition entering the synchronizer would cause the first flip-flop of the synchronizer to enter a metastable condition. When this occurs, the probability that a particular data-change will cause metastability is equal to 1 and not to the assumed value of (t_W/t_C). Using numbers from our example calculations in the previous paragraph, the value of (t_W/t_C) is equal to $\left(\frac{50ps}{10ns}\right) = \frac{1}{200}$. Thus, when *every* data-transition entering the synchronizer causes metastability then MTBF is reduced (in this example) by the significant factor of 1/200. My point being that equation (12.1) is perhaps too optimistic and should be multiplied by (t_W/t_C) to account for the worse-case condition that I have just described.

In closing, we find that the 2FF-synchronizer usually gives very acceptable performance in terms of MTBF. In the rare situations when you need better MTBF, you can use synchronizers having more than two flip-flops. However, I will not discuss these other synchronizers in this book.

12.5 Multi-Bit Synchronizers

In section 12.3, we talked about using a 1-bit synchronizer called the 2FF-synchronizer to safely pass 1-bit of data through a clock-crossing. Simultaneously passing multi-bit data through a clock-crossing is more complicated than simply using multiple 2FF-synchronizers. In this section, we will discuss complications of the multi-bit clock-crossing and then describe operation of an easily-understood and effective multi-bit synchronizer called a handshake synchronizer.

In section 12.3, we noted that operation of the 1-bit synchronizer requires us to hold the synchronizer input, SIG1, constant for at least two cycles of the synchronizer clock, CLK2. This requirement is necessary because it is uncertain whether a change of SIG1 will pass through the synchronizer on the first edge of CLK2 or on the second edge of CLK2 after SIG1 changes. This uncertainty makes multi-bit synchronizer operation more complicated. For example, suppose we need to pass a 4-bit signal, SIG4, through a clock-crossing from the CLK1 domain into the CLK2 domain. We could simply try using four 2FF-synchronizers in parallel. However, consider how the outputs of these four 2FF-synchronizers might look during a typical transfer of the 4-bit data. On EDGE1 of CLK2, suppose that all 4 bits of SIG4 transition from 0-to-1, causing a setup violation at all four of the meta_2ff_reg registers and metastability at the output of these registers. Further suppose that after EDGE1 of CLK2, SIG1 remains unchanged for many cycles of CLK2. At the next edge, EDGE2, of CLK2, all four metastable outputs will likely have settled to a valid logic state – *but probably not all to the same logic state*. So, at the time of EDGE2, the output of our attempted 4-bit synchronizer will be an uncertain mix of 1's and 0's (instead of the correct value of all 1's). On EDGE3 of CLK2, the output of our attempted 4-bit synchronizer will, with high probability, arrive at the all-1's data that was input to the synchronizer at the time of EDGE1. This all seems clear enough for you and me. However, things would certainly be confusing to someone viewing this from the CLK2 domain without any knowledge of what's happening in the CLK1 domain.

The attempted 4-bit synchronizer described in the previous paragraph needs only slight modification to turn it into a reliable multi-bit synchronizer called the handshake synchronizer drawn in Fig 12.7. Specifically, an instance of the 2FF-synchronizer is still used for each data bit. Two more instances of the 2FF-synchronizer are used to form the handshake signals called "ready(R)" and "busy(B)". Handshake synchronizer operation is described below by the separate explanations for what occurs in the CLK1-domain and in the CLK2-domain.

1) CLK1 domain (sends the data):
    a. **Rest State:** Set R=0 and wait until new data needs to be sent to the CLK2 domain.
    b. Place new multi-bit data on inputs to the 2FF-synchronizers used for data.
    c. Set R=1.
    d. Wait for B=1.
    e. Wait for B=0.
    f. Return to "**Rest State**".
2) CLK2 domain (receives the data):
    a. **Rest State:** Set B=0 and wait for R=1.
    b. After seeing R=1, wait one additional cycle of CLK2. This wait ensures that outputs of the 2FF-synchronizers used for data have settled.
    c. Read the multi-bit data from outputs of the 2FF-synchronizers used for data.
    d. Set B=0.
    e. Wait for R=0.
    f. Return to "**Rest State**".

The above description for a handshake synchronizer is not optimal (ie. it can be made to work faster) and it is not unique (ie. there are other types of handshake synchronizers). However, I think most readers will find the description to be straightforward and will easily see how it translates into HDL state machines - see also Ginosar[2003]. Also, you'll find that many FPGA vendors offer free VHDL components (aka IP – see chapter 9) that implement the handshake (and other types) of multi-bit synchronizers.

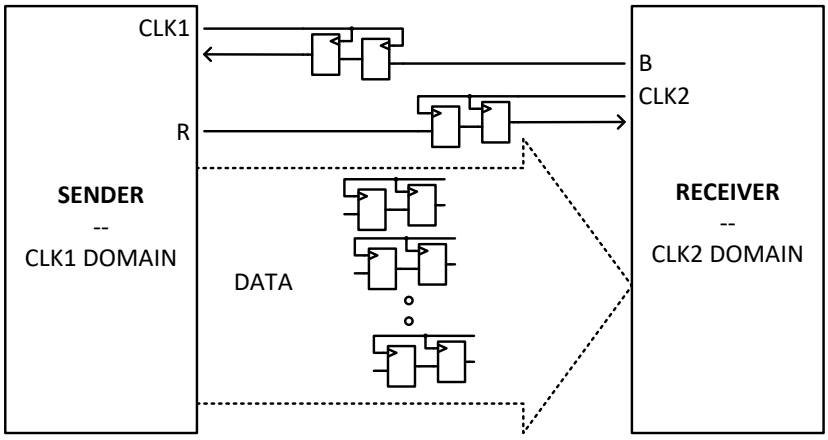

Fig 12.7 Block diagram for a multi-bit handshake synchronizer.

12.6 Other Considerations

As explained in section 12.2, not all clock-crossings cause metastability. Hence, not all clock-crossings need a synchronizer. At a clock-crossing where the two clocks are synchronous (eg. generated by the same FPGA clock-module), metastability may occur. We can rely on timing analysis to tell us (by reporting negative slack) about possible metastability at a synchronous clock-crossing. At a clock-crossing where the two clocks are asynchronous, metastability will almost surely occur and you must use a synchronizer.

However, metastability is not the only thing we need to worry about at clock-crossings. The simple fact that things happen at different rates in each clock domain means that we must give careful thought to every clock-crossing. For example, suppose that one domain uses a 100MHz clock, CLK100, and the other domain uses a 50MHz clock, CLK50. It is simply not possible to generate data on every tic of CLK100 and then transfer this data in real time to the CLK50-domain. However, you could generate and store bursts of data in the CLK100-domain, and then slowly transfer the data to the CLK50-domain on every tic of CLK50. Also, if CLK100 and CLK50 in this example are synchronous then timing analysis will probably not report negative slack for this crossing. So, although we can rely on timing analysis to flag a crossing with metastability problems, we cannot rely on timing analysis to flag a crossing with engineering problems. Hence, we must develop the habit of identifying clock-crossing as we write our HDL and giving each of them careful thought. Examples of good engineering solutions to clock-crossing problems are given by the handshake synchronizer in section 12.5 and the oversampled interface in chapter 13.

Here is another example of a clock-crossing where careful thought can save us some work. At power-up, suppose your FPGA receives setup data from a slow asynchronous serial interface for which your HDL uses a 10MHz clock, CLK10. Suppose the setup data is stored in the FPGA as a simple VHDL signal, SETUP1. Sometime later, SETUP1 is read by VHDL code that uses a 100MHz clock, CLK100. Thus, the signal called SETUP1 is crossed from the CLK10-domain to the CLK100-domain. What should we do about this crossing? Well, it may surprise you to learn that "doing (almost) nothing" can be the answer. That is, suppose your VHDL assigns a value to SETUP1 that simply sits there unchanged for at least a few cycles of CLK10 before it is read. Then, the CLK100-domain will see a value for SETUP1 coming through the clock-crossing that is metastable-free (ie. enough time has passed for any metastability on SETUP1 to settle out). So, for this example, just ensure that your VHDL in the CLK100-domain waits a few cycles of CLK10 after SETUP1 is received from the serial interface and before reading SETUP1.

There is a hidden clock-crossing in the example of the previous paragraph. Did you see it? Well, the hidden clock-crossing is the asynchronous serial interface itself, which is usually one wire of digital data entering the FPGA. That is, changes on the incoming serial data line are usually asynchronous with all clocks used by the FPGA. Hence, the serial data line is a clock-crossing between two clock domains that are asynchronous. The way to handle this clock-crossing is to simply place a 1-bit synchronizer (see section 12.3) on the serial data line. Further, a serial interface is usually handled by oversampling (see chapter 13) the serial data line and by knowing the bits-per-second (bps) data rate. In appendix C, you'll find details of handling a serial communications interface called RS232.

After placing a synchronizer at a clock-crossing, the IDE may still warn us that the clock-crossing failed timing analysis. These warnings can be ignored if the synchronizer has been properly constructed. However, if you tire of seeing these warnings, then you can write something called a *timing exception* command for each synchronizer. The timing exception will prevent the IDE from running timing analysis on the clock-crossing and thus suppress the timing analysis warning. Applicable timing exceptions are *"set_max_delay -datapath_only"* and *"set_clock_groups".* However, we are not yet ready to discuss timing exception commands. So, we'll return to this topic in chapter 17.

13. Level-1 FPGA I/O

The exchange of data between an FPGA and a device that is external to the FPGA will here be called FPGA input/output or simply FPGA I/O. As we'll discover in chapter 14, high-speed FPGA I/O can be complicated. However, as discussed in this chapter, low-to-medium speed (<40 MHz clocks) FPGA I/O is both intuitive and usually easy to implement. So, consider yourself forewarned – don't use the FPGA I/O methods described in chapter 14 unless necessary!

There are many data-producing devices that output a clock along with the data. For example, many digitizers output a so-called data-clock. For this type of digitizer, the datasheet will have a statement like, "the data outputs from the digitizer change on the rising-edge of the data-clock". From this statement, we know *not* to read the digitizer data on the rising-edge of the data-clock (ie. when the data are changing). Instead, our intuition tells us to read the digitizer data when it is stable and unchanging (ie. on the falling-edge of the data-clock). Hence, we simply tell our FPGA to "watch"

the data-clock and when a falling-edge appears then read the data outputs of the digitizer. Easy, right!? Well, it is easy but let's look at the details by discussing an example.

13.1 Oversampled Interface

The example FPGA I/O that we'll discuss is described by Fig 13.1. The external device (which we can think of as a 1-bit digitizer) sends 1-bit data called DAT to the FPGA on every rising-edge of the data-clock called CLKd. Our FPGA has a clock called CLKs that runs approximately eight times faster than CLKd (ie. there are 8 cycles of CLKs for every cycle of CLKd). Further, CLKs and CLKd are unrelated (ie. they are asynchronous).

First, note that Fig 13.1 has clock-crossings, which we discussed in section 12.2. Specifically, the signals called DAT and CLKd both come into the FPGA from the CLKd-domain and are captured by registers called Ds_reg and Cs_reg that operate in the CLKs-domain. Hence, as discussed in section 12.3, a 2-flip-flop synchronizer has been placed on each of these signals to prevent problems resulting from metastability.

Next, note that DAT and CLKd change at a much slower rate than CLKs. Because Ds_reg and Cs_reg are clocked by CLKs, their outputs will be samples of DAT and CLKd. Hence, the circuit in Fig 13.1 is generally called an *oversampled interface*.

So, our task is now clear. We instruct the FPGA to monitor the output of Cs_reg. When this output changes from high-to-low (ie. when a falling-edge for CLKd is detected), then the output of Ds_reg is captured and saved as valid data from the external device. See, I told you it was easy!

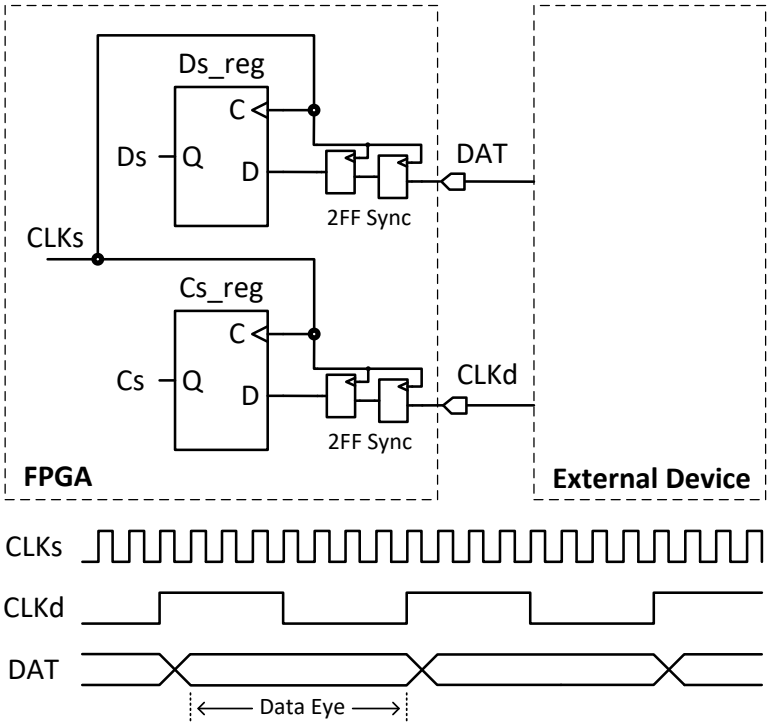

Fig 13.1 Oversampled interface between FPGA and an external device.

Finally, in Fig 13.1, you'll see that I have identified a segment of the DAT plot as the "data-eye", which is the time interval when the DAT line is stable (ie. not changing). In digital lingo, it is said that data lines should be read in the middle of the data-eye, which is just what we have accomplished by reading DAT on the falling-edge of CLKd. *Capturing/reading the data in the middle of the data-eye is perhaps the most fundamental and the most important concept of doing FPGA I/O*.

13.2 Simple Output

The previous section discussed simple *input* to the FPGA using the oversampled interface approach. You'll be pleased to know that FPGA output is often easier to do than FPGA input. In this section's discussion about simple output, we'll assume that the FPGA is sending a data-clock, CLKd, along with the data, DAT.

For FPGA output, you may be tempted to just push all the hard-to-do stuff onto the device that is receiving the data. That is, we could output the data in any way that is convenient and let the other device worry about receiving the data properly. Unfortunately, most of the "other devices" you'll encounter are neither smart nor flexible. Instead, these devices will often receive data in only "one way" and we must design our FPGA output around this fact. For example, specifications for an external device might say that data is read/received from the data lines immediately after the device sees a falling-edge of the data-clock. From this statement, we infer that the receiving device interprets the center of the data-eye to be at the *falling-edge* of the data-clock. Hence, the FPGA must make this interpretation true by transmitting data (ie. placing a new value on the data line) coincident with the *rising-edge* of the data-clock. Similarly, if the external device receives data on the rising-edge of the data-clock, then the FPGA should be programmed to transmit data coincident with the falling-edge of the data-clock.

So, that's about all we need to say about simple output from the FPGA. Unlike simple input to the FPGA discussed in section 13.1, synchronizers are not needed and there is usually no need to have the fast clock, CLKs. Instead, we only need the data-clock, CLKd, and we use CLKd to clock the data out of the FPGA. See, I told you it was easy!

13.3 Bit Banging I2C

Using the oversampled-interface concepts of using HDL (instead of using dedicated hardware) to handle FPGA I/O is sometimes called bit-banging. Simple and slow serial interfaces such as the Serial Peripheral Interface (SPI) or Inter-Integrated Circuit (I2C) are easily implemented using bit-banging. When using bit-banging, it is sometimes said that we are satisfying timing requirements of the I/O "by design". The following example of bit-banging for I2C will help clarify "by design". Our example I2C interface consists of three wires called SDA (the data line), SCL (the clock line), and a ground reference. Typical waveforms for SDA and SCL are shown in Fig 13.2 along with symbols, (tHDS, tSUD, tHDD), that represent timing requirements. For this I2C example, we'll assume that SCL has a speed of 100kHz and CLKs has a speed of 1MHz.

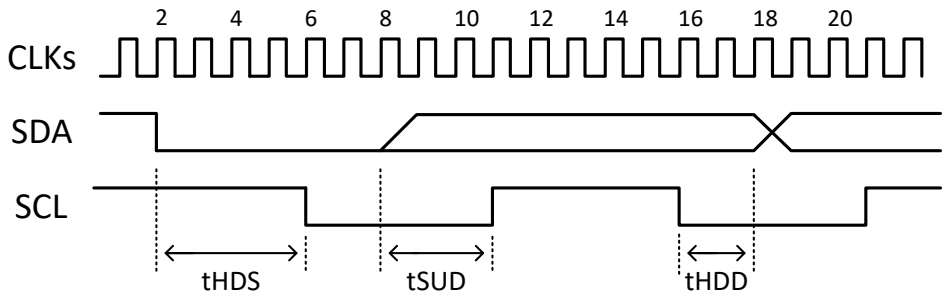

Fig 13.2 Example of startup waveforms for I2C serial I/O.

During the idle state for I2C, both SDA and SCL are held high. Communications over I2C begins with a start condition whereby SDA is pulled low and then SCL is pulled low no sooner that tHDS=4.0us later. Thus, tHDS=4.0us, is a timing requirement for I2C. Communications over I2C proceeds as follows, where you'll see that we count cycles of CLKs to ensure that the tHDS timing requirement and other timing requirements are met.

1. Ensure that the I2C interface in the idle state (ie. SCL=1, SDA=1)
2. Begin communications by setting SDA=0
3. Count 4 cycles of CLKs and set SCL=0. Note that 4 cycles of CLKs is 4us which satisfies the I2C timing requirement of tHDS=4.0us.

4. Count 2 cycles of CLKs and place the 1st bit of data to be transmitted on the SDA line.
5. Count 3 cycles of CLKs and set SCL=1. Note that 3 cycles of CLKs is 3us, which satisfies the I2C timing requirement of tSUD=0.25us.
6. Count 5 cycles of CLKs and set SCL=0. Note that 5 cycles of CLKs is 5us and is half the period of the 100kHz SCL.
7. Count 2 cycles of CLKs and place 2nd bit of data to be transmitted on the SDA line. Note that 2 cycles of CLKs is 2us, which satisfies the I2C timing requirement of tHDD=0.30us.
8. Count 3 cycles of CLKs and set SCL=1.
9. …and so on….

The previous I2C example describes what the I2C-master does to start communication with other devices called I2C-slaves. Note that the I2C-master transmits data bits shortly after a falling-edge for SCL. Consequently, I2C-slaves know to receive data bits shortly after a rising-edge for SCL. This I2C method of sending and receiving data bits will roughly place the data-sampling time in the middle of the data-eye (compare Figs 13.1 and 13.2), which is often our goal for FPGA I/O.

In appendix-C, you'll find that the RS232 asynchronous communications interface can also be handle using the oversampled-interface methods described in this chapter.

13.4 Wrap-Up

In Fig 13.1, the sampling clock, CLKs, just happened to have eight times the speed of the data-clock, CLKd. A faster CLKs would have worked too. However, the oversampled interface will have problems when the speed of CLKs is slowed to near the speed of CLKd. That is, a slow-speed CLKs makes it difficult for the FPGA to find the exact time of the CLKd falling-edge. Thus, with a slow-speed CLKs, sampling of DAT might occur at a time that is far from the middle of the data-eye. Also, we must consider the rule from section 12.3 that a 2FF-synchronizer operates properly when the data input to the synchronizer is held constant for at least two cycles of the clock used in the receiving clock-domain. For the oversampled interface, this rule means that the period of CLKd must be at least twice the period of CLKs. Otherwise, some changes in CLKd and in DAT might not be detected.

Adding to the uncertainty of finding the middle of the data-eye is that uncertainty discussed in section 12.3 about when data inputs to the 2FF-synchronizer reach the output of the synchronizer. That is, when our FPGA detects a falling-edge of CLKd by looking at the output of the Cs_reg, then this falling edge might have just occurred, or it might have occurred one cycle of CLKs in the past. Again, using a fast CLKs (ie. a CLKs with short period) will help minimize this one-cycle-of-CLKs uncertainty.

For the Fig 13.1 example, it is desirable to have equal-length paths for CLKd-to-Cs_reg and for DAT-to-Ds_reg. When these two paths are not of equal length then, as viewed by the FPGA, the CLKd and DAT plots in Fig 13.1 become shifted/skewed with respect to each other. Hence, when the FPGA saves the value of Ds_reg after detecting a high-to-low transition on Cs_reg, then it may not be saving a value for DAT from the middle of the data-eye. Obtaining equal-length for the portion of these two paths that lie outside the FPGA is the circuit board designers job. However, obtaining equal-length for the portion of these two paths that lie inside the FPGA is your job, which you accomplish by using appropriate timing constraints. For example, the Xilinx Vivado IDE has a timing exception called "set_max_delay -datapath_only" that can be used to keep implementation from laying out the Fig 13.1 circuit inside the FPGA with a path-length for CLKd-to-Cs_reg that differs greatly from the path-length for DAT-to-Ds_reg. The "set_max_delay -datapath_only" timing exception is further described in section 17.2.

The unequal path-lengths discussed above are unlikely to be a problem for a slow-speed CLKd but must be considered as the speed of CLKd increases. Also, other problems with the oversampled interface method will develop as the speed of CLKd increases. Eventually, for high speed CLKd, we will need the help of timing analysis (as described in chapters 14

and 15) to design and *proof* the FPGA I/O. Here, proof means determining whether the interface will work (from a timing analysis point of view). What is considered "high speed CLKd"? Well, FPGAs have an upper limit for clock frequency of about 400MHz. -and, for the oversampling method, we want CLKs to have a frequency that is about 10-times the frequency of CLKd. So, the oversampling method can be used for CLKd frequencies up to about 40MHz.

14. Level-2 Timing Analysis

This chapter is about the gritty details of timing analysis that I glossed over in section 11.3. I stand by my comments of section 11.2 that most of us don't need the gritty details of timing analysis to do our daily FPGA work. However, in this book, we are building towards high-speed FPGA I/O (see chapter 15) – and for that we will need the gritty details.

Personally, I find timing analysis and high-speed FPGA I/O to be the most challenging parts of my FPGA work. So, whenever possible, I try to operate with Level-1 concepts (ie. follow section 11.3 and chapter 13). However, sometimes life is complicated. Hopefully, this chapter and chapter 15 will help get you through some of those complications.

14.1 Overview

In short, the level-2 understanding of timing analysis that we need for solving FPGA I/O problems is a combination of level-1 understanding discussed in section 11.3 and the HDL-to-Hardware connection discussed in section 7.5.

I'll start our discussion with the following one-sentence recap of section 7.5. *Generally, the signals in our HDL correspond to digital registers and the operations/processing (eg. logical-AND, arithmetic, if-then-else) that we perform on the signals correspond to combinational-logic*. For example, suppose that our HDL processes the 1-bit signals, S1 and S2, and stores the result in the 1-bit signal, S3. Implementation of this HDL might look something like the digital circuit shown previously in Fig 7.1. In that figure, we see registers called S1_reg, S2_reg, and S3_reg that correspond to each of the three HDL signals. Further, the figure shows that after the signals called S1 and S2 are clocked-out of their registers, they are "processed" by digital AND-gates (ie. combinational logic) on the way to becoming the signal called S3 that is stored in S3_reg. This brief background should allow you to partially understand the following key statement, which is an important concept for understanding level-2 timing analysis. *Timing analysis checks that signals travel between registers (and possibly through combinational logic) in a timely manner*.

You may be unsatisfied with the key statement at the end the previous paragraph – especially the rather ambiguous part about "timely manner". However, from previous discussions in this book, you know that I won't leave you hanging. In the rest of this chapter, we'll discuss "timely manner" in detail. These discussions will focus on digital hardware found inside the FPGA and will give you a hardware-view of timing analysis. Hopefully in the end, you will see how this chapter's hardware-view of timing analysis relates to the software-view of timing analysis presented in chapter 11.

14.2 Setup Timing Analysis

In section 11.3, I mentioned that timing analysis reports values for setup-slack and hold-slack. In this section, we will discuss exactly how timing analysis calculates setup-slack and what it means from a hardware viewpoint. For this discussion, we will use Fig 7.1 (repeated below as Fig 14.1). First, Fig 14.1 shows that the clock, CLK, used by all three digital registers comes from an FPGA clock-module called MMCM1. As discussed in chapter 8, it is common and highly recommended that all clocks used in your design be generated using an FPGA clock-module.

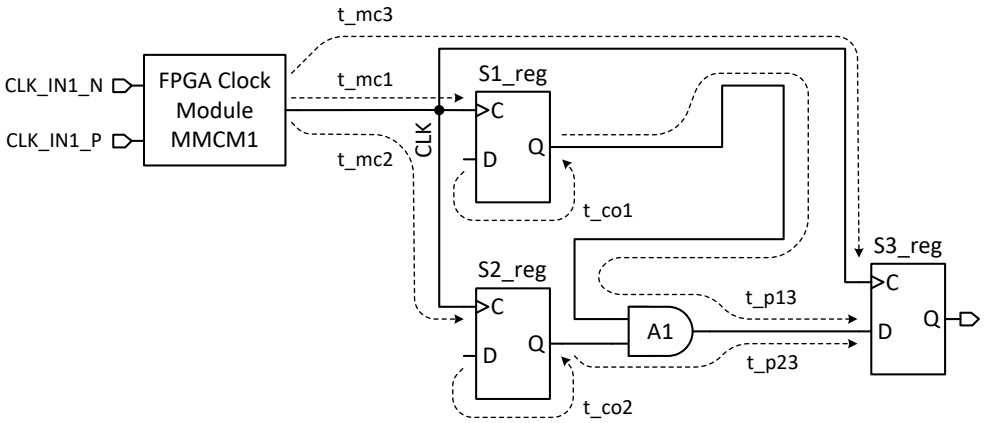

Fig 14.1 Typical FPGA circuit with path propagation times(arcs) that are important for timing analysis.

From the hardware viewpoint, timing analysis focuses on the time it takes for clocks and signals to propagate through traces and digital components. In FPGA lingo, these propagation times are called *timing-arcs*. In Fig 14.1, timing-arcs needed by timing analysis are identified by dashed lines and by labels that begin with "t_". For example, the arc called t_mc1 is the time it takes the clock signal, CLK, to travel from MMCM1 through the FPGA clock tree to the clock pin, S1_reg/C. Similarly, t_p13 is the time it takes the signal, S1, to propagate from the output pin, S1_reg/Q, and through component, A1, and to the input pin, S3_reg/D. Finally, t_co1 is the time it takes S1_reg to transfer data from its input, S1_reg/D, to its output, S1_reg/Q, after receiving the rising-edge of CLK on S1_reg/C. In the lingo of FPGAs and digital hardware, t_co1, is often called the *clock-to-output time* of register, S1_reg.

From our key statement in section 14.1, we know that timing analysis checks whether signals travel between registers in a timely manner. In Fig 14.1 for example, timing analysis will check that signal, S1, travels from S1_reg to S3_reg in a timely manner. As a first step towards quantifying what we mean by "timely manner", we will discuss how timing analysis calculates a quantity called *data-arrival-time* for S1 at S3_reg. For this calculation, we consider the following sequence of events:

1. EDGE1 of CLK is launched from MMCM1 at time, t_mle. (EDGE1 is called the *launch-edge* of CLK)
2. EDGE1 of CLK travels through the FPGA clock tree, taking t_mc1 seconds to reach S1_reg/C
3. S1_reg sees EDGE1 of CLK and takes t_co1 seconds to transfer data from S1_reg/D to S1_reg/Q
4. The signal, S1, travels from S1_reg/Q and through A1, taking t_p13 seconds to reach S3_reg/D

Adding up these times, timing analysis calculates the data-arrival-time for S1 at S3_reg to be (t_mle + t_mc1 + t_co1 + t_p13) seconds.

After calculating data-arrival-time for S1 at S3_reg, timing analysis will then calculate the *data-required-time* for S1 at S3_reg. In our example, data-required-time is the latest time that a signal can arrive at S3_reg and be properly received by S3_reg. Timing analysis starts the calculation for data-required-time at the time, t_mce, for the *capture-edge*, EDGE2, of CLK. That is, EDGE2 is the CLK edge that will eventually cause capture (by S3_reg) of the data that was launched (from S1_reg) by EDGE1. In this example, EDGE2 will be the rising-edge of CLK that comes one-period-of-CLK after EDGE1. For calculation of data-required-time, we consider the following sequence of events:

1. EDGE2 of CLK is launched from MMCM1 at time, t_mce. (EDGE2 is called the *capture-edge* of CLK)
2. EDGE2 of CLK travels through the FPGA clock tree, taking t_mc3 seconds to reach S3_reg/C
3. Hence, S3_reg will capture the S1 signal appearing at S3_reg/D at time = (t_mce + t_mc3) seconds.
4. However, to properly capture the S1 signal, the setup time, t_su3, and hold time, t_hd3, of S3_reg must be satisfied.

5. Satisfying the setup requirement for S3_reg means that the data-arrival time of (t_mle + t_mc1 + t_co1 + t_p13) must come t_su3 seconds before the capture time of (t_mce + t_mc3). That is, (t_mle + t_mc1 + t_co1 + t_p13) must be less than (t_mce + t_mc3 - **t_su3**).
6. Thus, for satisfying the setup requirement of S3_reg, the data-required-time is (t_mce + t_mc3 - t_su3).
7. Satisfying the hold requirement for S3_reg will be discussed in section 14.3.

Now that we know how timing analysis calculates data-arrival-time and data-required-time(setup), we are finally ready to quantify "timely manner", which we will do by calculating *setup-slack*. In short, setup-slack is equal to *data-required-time minus data-arrival-time*. For our example where data/signal, S1, is transferred from S1_reg to S3_reg, the calculation for *setup-slack* is [(t_mce + t_mc3 – t_su3) - (t_mle + t_mc1 + t_co1 + t_p13)]. When the setup-slack calculation results in a positive number then the register-to-register transfer of data is said to have happened in a "timely manner" (and is said to have passed setup timing analysis).

Timing analysis will repeat the setup-slack calculation for every register-to-register transfer of data that it finds in our FPGA project. For example, in Fig 14.1, setup-slack will also be calculated for the transfer of signal, S2, from S2_reg to S3_reg. Using the labeled timing-arcs found in Fig 14.1, this setup-slack calculation becomes [(t_mce + t_mc3 – t_su3) - (t_mle + t_mc2 + t_co2 + t_p23)].

Finally, using what was discussed in this section, we can understand the concepts of setup-slack and setup-WNS that were introduced in section 11.3.2. Specifically, *setup-WNS is the most-negative (ie. worst) value of setup-slack found during timing analysis of our entire FPGA project*.

14.3 Hold Timing Analysis

The concept of hold-slack was very casually introduced in section 11.3.2. Specifically, I told you not to worry about it. Well, for things happening inside the FPGA, my "don't worry about it" advice still stands. However, we are now preparing to talk about FPGA input/output (ie. things that, in part, happen outside the FPGA). So, now I'm telling you that for things that happen outside the FPGA (eg. FPGA input/output) hold-slack is important – and we must understand how it is calculated by timing analysis.

Like the setup-slack calculation, the hold-slack calculation is used by timing analysis to ensure the timely transfer of data between digital registers. We will again use the Fig 14.1 example circuit and the transfer of data from S1_reg to S3_reg to help describe hold-slack. Recall from section 14.2 that the setup-slack calculation focuses on what happens shortly before S3_reg is clocked. That is, does data coming from S1_reg arrive at S3_reg too late and cause a setup violation at S3_reg? In contrast, the hold-slack calculation focuses on what happens shortly after S3_reg is clocked. That is, does the next data coming from S1_reg arrive at S3_reg too early and cause a hold violation at S3_reg?

The calculation for setup-slack and hold-slack both use the same expression for data-arrival time, which for our example is (t_mle + t_mc1 + t_co1 + t_p13). However, the data-required-time is calculated differently for hold-slack and setup-slack. When we calculated data-required-time for setup-slack, the calculation followed the capture-edge, EDGE2, which was defined as the edge of CLK that results in the capture of data that was launched by EDGE1 of clock. In contrast, the calculation of data-required-time for hold-slack follows the capture-edge of CLK that precedes EDGE2, which (in this example) is simply EDGE1. The calculation of data-required-time(hold) is described by the following sequence of events.

1. EDGE1 of CLK is launched from MMCM1 at time, t_mle.
2. EDGE1 of CLK travels through the FPGA clock tree, taking t_mc3 seconds to reach S3_reg/C
3. After EDGE1 of CLK reaches S3_reg/C, the input to S3_reg/D must remain stable for t_hd3 seconds, which is the hold time requirement for S3_reg.
4. Satisfying the hold requirement for S3_reg means that the data-arrival time of (t_mle + t_mc1 + t_co1 + t_p13) must come t_hd3 seconds after the capture time of (t_mle + t_mc3). That is, (t_mle + t_mc1 + t_co1 + t_p13) must be greater than (t_mle + t_mc3 + **t_hd3**).

5. Thus, for satisfying the hold requirement of S3_reg, the data-required-time is (t_mle + t_mc3 + t_hd3).

Putting all this together, we find that *hold-slack is calculated as data-arrival-time minus data-required-time*. Note that this is different from setup-slack, which is calculated as data-required-time minus data-arrival-time. For our Fig 14.1 example, the hold-slack calculation is [(t_mle + t_mc1 + t_co1 + t_p13) – (t_mle + t_mc3 + t_hd3)]. When the calculation for hold-slack results in a positive number then the register-to-register transfer of data to which this calculation applies is said to have happened in a "timely manner" (and is said to have passed hold timing analysis).

Finally, using what was discussed in this section, we can understand the concepts of hold-slack and hold-WNS that were introduced in section 11.3.2. Specifically, *hold-WNS is the most-negative (ie. worst) value of hold-slack found during timing analysis of our entire FPGA project*.

14.4 Slack Uncertainty

From sections 14.2 and 14.3, we found that calculations for setup-slack and for hold-slack are done during timing analysis and they depend on the time it takes for data and clocks to propagate through wires and digital devices. Further, we learned that setup-slack and hold-slack are calculated for every register-to-register transfer of data that timing analysis finds in our FPGA project. When circuits resulting from implementation of our HDL lie entirely within the FPGA, then the IDE and its timing analysis tool will know everything about these circuits, including values for all the propagation times (aka timing-arcs) needed to calculate setup and hold-slack. So, for these circuits, timing analysis will automatically calculate setup and hold-slack without any help from us. However, when circuits resulting from implementation of our HDL lie partially outside the FPGA, then timing analysis needs our help to calculate setup and hold-slack – more on this in chapter 15. For now, we'll focus on circuits resulting from implementation of our HDL that lie entirely within the FPGA.

Now, we'll discuss other things that timing analysis does during calculation of setup-slack and hold-slack. Specifically, timing analysis factors into these calculations the natural variations found in electrical devices and the fact that clocks are imperfect. For example, manufacturing variations can cause setup and hold requirements to vary a little from register to register. Likewise, the response time of combinational logic can vary from gate to gate and cause variations in the propagation time for a signal that passes through different gates of the same type. Finally, clocks are never perfect. So, for example, when we use an FPGA clock-module (see chapter 9) to create a digital clock for our project then this clock will sometime run a little too fast and sometimes run a little too slow. The point being that the IDE usually knows these hardware variations/imperfections and takes them into account during timing analysis.

Occasionally, you'll hear about the *four corners of timing analysis*, which refers to the uncertain/imperfect part of timing analysis discussed in the previous paragraph. More specifically, the four corners are all four combinations of the terms called fast-clock and slow-clock with the terms called max-delay and min-delay. As you can probably guess, fast/slow clock refers to the clock imperfections and max/min delay refers to the variations in hardware propagation delay caused by variations in process(manufacturing), voltage, and temperature (aka PVT).

The details of how timing analysis handles uncertainty is usually not important to you and me. However, you should understand that it is done using statistics and is done in a very conservative or pessimistic manner. So, if your design passes timing analysis, then it is highly probable that your design will operate properly (from a timing analysis point of view) on any FPGA that you buy from the vendor whose timing analysis tool (ie. IDE) you are using.

14.5 Timing Path Report

As we'll discover in chapter 15, we sometimes need to see the details of timing analysis. Most IDEs are able to give you these details using what is generally called a *path report*. In this section, we will talk about how to obtain and to read the path report.

In Fig 14.2 you will see a circuit schematic drawn by the Vivado IDE that corresponds to an implementation of some HDL code that I wrote. I got the Fig 14.2 schematic from the Vivado IDE using "Open Implemented Design", similar to the way I got schematics from "Open Synthesized Design" in section 10.2. The Fig 14.2 circuit does exactly the same thing as the Fig 14.1 circuit. Fig 14.2 differs from Fig 14.1 by showing details of what's inside the MMCM1 block. Note that the combinational logic (AND-gate, A1) in Fig 14.1 has been implemented in Fig 14.2 using the LUT that we discussed in section 10.3. Also, inside the MMCM1 block of Fig 14.2, you will see buffer components called IBUFDS and BUFG which represent circuitry found inside the FPGA.

In Fig 14.2, let's consider the path from S1_reg to S3_reg. For this path, we can obtain the path report for *setup* timing by typing the following Tcl command into the Tcl Console of the Vivado IDE.

```
report_timing -from [get_pins S1_reg/C] -to [get_pins S3_reg/D] -setup
```
(14.1)

We can obtain the path report for *hold* timing using the following Tcl command.

```
report_timing -from [get_pins S1_reg/C] -to [get_pins S3_reg/D] -hold
```
(14.2)

The setup path report generated by the **`report_timing`** command is shown in Fig 14.3. At first glance, I'm sure this report looks scary. That's because it contains lots of stuff needed for the *four corners of timing analysis* that I mentioned in section 14.4. However, the path report also contains the data needed for the timing analysis simple calculations that we did in section 14.2. Let's slowly go through Fig 14.3 to discover what it's telling us.

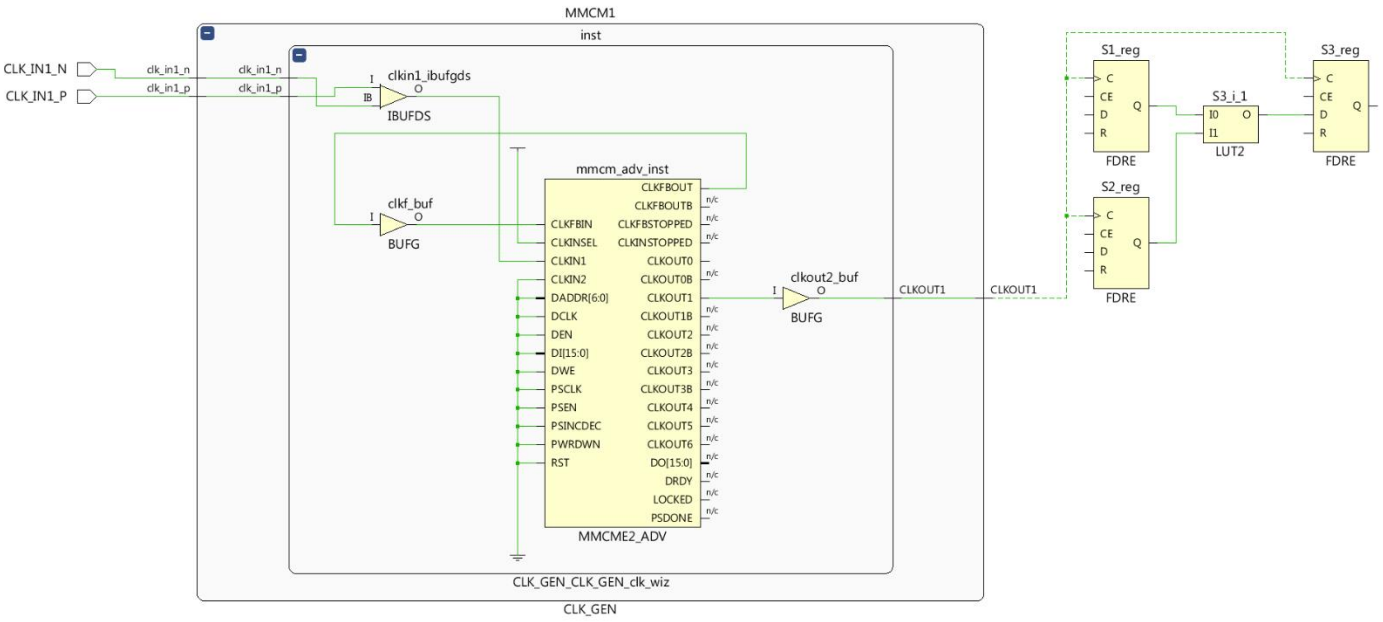

Fig 14.2 Xilinx Vivado implemented design schematic for some simple register-to-register paths.

The path report in Fig 14.3 is divided into the following sections:

- *Header* (lines 01-08): Miscellaneous stuff. Line-05 shows the Tcl command used to generate the report
- *Summary* (lines 09-32): A collection of the important results, with line-12 showing the key result, slack.
- *Source Clock Path* (lines 34-46): Path traveled by the clock to reach S1_reg.
- *Data Path* (lines 47-52): Path traveled by signal, S1, as it moves from S1_reg to S3_reg.
- *Destination Clock Path* (lines 53-68): Path traveled by the clock to reach S3_reg.
- *Slack Calculation* (lines 69-73): Calculation of setup-slack = data-required-time minus data-arrival-time

In line 33, you'll see the column labels called "Incr(ns)" and "Path(ns)". A line of the report that has values in these columns represents a component or a wire (aka net). Further, the number found in the "Incr(ns)" column is the propagation time (aka delay) for a signal that travels through this component or wire. Values in the "Path(ns)" column represent a running total of values found in the "Incr(ns)" column.

Now, let's get our hands dirty by playing with some numbers. In Fig 14.3, you'll see several references to the clock output, CLKOUT1, of MMCM1 in Fig 14.2. You'll note from the Fig 9.3 setup of MMCM1 that I specified the frequency of CLKOUT1 to be 250MHz (ie. CLKOUT1 has a period of 4ns). You'll note from Fig 10.1 that when I instantiated MMCM1 into TOP.vhd that I associated CLKOUT1 with the VHDL signal name CLK250. So, in Fig 14.3 you'll see reference to CLKOUT1 and to CLK250, but they are both the same clock. Arguably, the most important part of Fig 14.3 is line-12 where the slack value is reported as +3.513ns. This positive value of slack tells us that the path from S1_reg to S3_reg has passed setup timing analysis. Let's manually calculate setup-slack using the formulas discussed in section 14.2 and see if we get a number that is near the reported value of +3.513ns. So, using the notation from section 14.2, we find:

- data arrival time(setup) = (t_mle + t_mc1 + t_co1 + t_p13) = (0.00 + 1.371 + 0.198 + 0.139 + 0.119) = 1.827
    - t_mle = 0.000 = time that launch-edge of CLKOUT1 leaves MMCM1
    - t_mc1 = 1.371(line-45) = time for propagation of CLKOUT1 from MMCM1 to S1_reg/C
    - t_co1 = 0.198(line-48) = clock-to-output(Q) time for register, S1_reg.
    - t_p13 = (0.139 + 0.119)(lines 49-51) = time for propagation of S1 from S1_reg/Q thru LUT2 to S3_reg/D
- data-required-time(setup) = (t_mce + t_mc3 – t_su3) = (4.000 + 1.178 – 0.033?) = 5.145?
    - t_mce = 4.000 = time that capture-edge of CLKOUT1 leaves MMCM1
    - t_mc3 = 1.178(line-64) = time for propagation of CLKOUT1 from MMCM1 to S3_reg/C
    - t_su3 = 0.033?(line-68) = setup time for S3_reg

Using these data, our simple calculation of setup-slack (ie. data-required-time minus data-arrival-time) equals (5.145 – 1.827) or 3.318ns. Next, let's explore why I have placed a question mark by the value for t_su3 above and why our manually calculated value for slack does not exactly match the value of 3.513ns in line-12 of Fig 14.3.

With regards to the value for t_su3, things become clearer if line-68 of Fig 14.3 is rewritten as shown in Fig 14.4. That is, setup time for the receiving register (S3_reg in this example) is always subtracted during the calculation of data-required-time. However, in this example, the S3_reg has setup time value that is negative! This seemingly odd property of S3_reg is normal for some Xilinx FPGA registers. Presumably, this is because the S3_reg block shown in Fig 14.2 is not a pure register and is instead a register with other stuff (probably Xilinx proprietary). Anyway, now that we understand line-68, we need to adjust the above calculation for data-required-time(setup) to be (t_mce + t_mc3 – t_su3) = 4.000 + 1.178 – (– 0.033) = 5.211. Our simple calculation for setup-slack is now (5.211 – 1.827) or 3.384ns, bringing us closer to the value of 3.513ns from line-12 of Fig 14.3. The remaining difference between these two slack values is due to the *four corners of timing analysis*, which we will briefly discuss next.

Bonus Question: Why does it appear that the numbers from line-70 and line-71 of Fig 14.3 have been added to give the slack time in line-73?

Hint: Look at the sign of the "Path(ns)" number in line-51.

```
------------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.3 (win64) Build 2405991 Thu Dec  6 23:38:27 MST 2018
| Date         : Sat Jun 15 10:48:02 2019
| Host         : Rosetta running 64-bit Service Pack 1  (build 7601)
| Command      : report_timing -from [get_pins S1_reg/C] -to [get_pins S3_reg/D] -setup
| Design       : TOP
| Device       : 7k160t-fbg484
| Speed File   : -3  PRODUCTION 1.12 2017-02-17
------------------------------------------------------------------------------------
Timing Report

Slack (MET) :             3.513ns  (required time - arrival time)
  Source:                 S1_reg/C
                            (rising edge-triggered cell FDRE clocked by CLKOUT1_CLK_GEN {rise@0.000ns
                                  fall@2.000ns period=4.000ns})
  Destination:            S3_reg/D
                            (rising edge-triggered cell FDRE clocked by CLKOUT1_CLK_GEN {rise@0.000ns
                                  fall@2.000ns period=4.000ns})
  Path Group:             CLKOUT1_CLK_GEN
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            4.000ns  (CLKOUT1_CLK_GEN rise@4.000ns - CLKOUT1_CLK_GEN rise@0.000ns)
  Data Path Delay:        0.456ns  (logic 0.317ns (69.562%)  route 0.139ns (30.438%))
  Logic Levels:           1  (LUT2=1)
  Clock Path Skew:        0.000ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    -1.194ns = ( 2.806 - 4.000 )
    Source Clock Delay      (SCD):    -1.664ns
    Clock Pessimism Removal (CPR):    -0.470ns
  Clock Uncertainty:      0.065ns  ((TSJ^2 + DJ^2)^1/2) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Discrete Jitter          (DJ):    0.108ns
    Phase Error              (PE):    0.000ns

    Location             Delay type                Incr(ns)  Path(ns)    Netlist Resource(s)
  -------------------------------------------------------------------    -------------------
                         (clock CLKOUT1_CLK_GEN rise edge)
                                                    0.000     0.000 r
    W9                                              0.000     0.000 r  CLK_IN1_P (IN)
                         net (fo=0)                 0.000     0.000    MMCM1/inst/clk_in1_p
    W9                   IBUFDS (Prop_ibufds_I_O)   0.843     0.843 r  MMCM1/inst/clkin1_ibufgds/O
                         net (fo=1, routed)         0.962     1.805    MMCM1/inst/clk_in1_CLK_GEN
    MMCME2_ADV_X1Y1      MMCME2_ADV (Prop_mmcme2_adv_CLKIN1_CLKOUT1)
                                                   -6.572    -4.767 r  MMCM1/inst/mmcm_adv_inst/CLKOUT1
                         net (fo=1, routed)         1.652    -3.115    MMCM1/inst/CLKOUT1_CLK_GEN
    BUFGCTRL_X0Y0        BUFG (Prop_bufg_I_O)       0.080    -3.035 r  MMCM1/inst/clkout2_buf/O
                         net (fo=8, routed)         1.371    -1.664    CLK250
    SLICE_X1Y230         FDRE                                     r  S1_reg/C
  -------------------------------------------------------------------    -------------------
    SLICE_X1Y230         FDRE (Prop_fdre_C_Q)       0.198    -1.466 r  S1_reg/Q
                         net (fo=1, routed)         0.139    -1.327    S1
    SLICE_X1Y230         LUT2 (Prop_lut2_I0_O)      0.119    -1.208 r  S3_i_1/O
                         net (fo=1, routed)         0.000    -1.208    S3_i_1_n_0
    SLICE_X1Y230         FDRE                                     r  S3_reg/D
  -------------------------------------------------------------------    -------------------
                         (clock CLKOUT1_CLK_GEN rise edge)
                                                    4.000     4.000 r
    W9                                              0.000     4.000 r  CLK_IN1_P (IN)
                         net (fo=0)                 0.000     4.000    MMCM1/inst/clk_in1_p
    W9                   IBUFDS (Prop_ibufds_I_O)   0.734     4.734 r  MMCM1/inst/clkin1_ibufgds/O
                         net (fo=1, routed)         0.895     5.629    MMCM1/inst/clk_in1_CLK_GEN
    MMCME2_ADV_X1Y1      MMCME2_ADV (Prop_mmcme2_adv_CLKIN1_CLKOUT1)
                                                   -5.645    -0.016 r  MMCM1/inst/mmcm_adv_inst/CLKOUT1
                         net (fo=1, routed)         1.572     1.556    MMCM1/inst/CLKOUT1_CLK_GEN
    BUFGCTRL_X0Y0        BUFG (Prop_bufg_I_O)       0.072     1.628 r  MMCM1/inst/clkout2_buf/O
                         net (fo=8, routed)         1.178     2.806    CLK250
    SLICE_X1Y230         FDRE                                     r  S3_reg/C
                         clock pessimism           -0.470     2.336
                         clock uncertainty         -0.065     2.271
    SLICE_X1Y230         FDRE (Setup_fdre_C_D)      0.033     2.304    S3_reg
  -------------------------------------------------------------------
                         required time                        2.304
                         arrival time                         1.208
  -------------------------------------------------------------------
                         slack                                3.513
```

Fig 14.3 The Xilinx Vivado setup timing report for the path between S1_reg and S3_reg in Fig 14.2.

```
65      SLICE_X1Y230         FDRE                                          r  S3_reg/C
66                           clock pessimism          +(-0.470)     2.336
67                           clock uncertainty        -(+0.065)     2.271
68      SLICE_X1Y230         FDRE (Setup_fdre_C_D)    -(-0.033)     2.304    S3_reg
69    -------------------------------------------------------------------
70                           required time                          2.304
71                           arrival time                           1.208
72    -------------------------------------------------------------------
73                           slack                                  3.513
```

Fig 14.4 Alternate way to write last few lines of the timing report shown in Fig 14.3.

As I mentioned in section 14.4, the *four corners of timing analysis* deals with clock imperfections and variations in hardware propagation delay over PVT. When "four corners" analysis is added to the simple timing analysis of sections 14.2 and 14.3, we will call it *full* timing analysis. The details of how the two are combined are usually not important to you and me. However, I'd like you to remember that full timing analysis produces a pessimistic value of slack. Here, pessimistic means a smaller value of slack. More importantly, it means that when a path passes timing analysis (ie. has a positive value of slack) then you can be confident that the path will pass timing analysis over a very wide range of PVT.

An example of full timing analysis pessimism can be seen in the source-clock-path calculations found in (lines 34-46) of Fig 14.3. Whereas our simple calculation for source-clock-path, t_mc1, started at the output of MMCM1, the full calculation starts way back at the clock inputs, CLK_IN1_P/N, to the FPGA. Without getting into too much detail, the concept is that delay variations thru components (eg. IBUFDS and MMCME2_ADV in Fig 14.2) can cause CLKOUT1 edges to come out of MMCM1 either early or late. Since full timing analysis is pessimistic, it selects a source-clock edge that comes out late and a destination-clock edge that comes out early. This pushes the two clock edges closer together in time and results in a smaller (more pessimistic) value for slack.

Sometimes, initial calculations of full timing analysis are too pessimistic. So, a normally-positive number called clock pessimism (aka clock pessimism removal (CPR)) is added to the data-required-time (see line-66 in Fig 14.3). However, in this weird example that I chose to discuss with you, initial calculations of full timing analysis were too optimistic (ie. not pessimistic enough). So, full timing analysis added an unusual negative value for clock pessimism. Finally, clock jitter effects are expressed in an always-positive term called "clock uncertainty" that is subtracted from data-required-time (see line-67 in Fig 14.3) to make slack more pessimistic.

That's enough about timing analysis, path reports, and pessimism to get us through most of our FPGA struggles. Let's be optimistic and move on!

14.6 DONT_TOUCH

With all our talk about registers, clocks, and timing paths, we are really starting to "think hardware". Now is good time to revisit the topic of synthesis, which is where software becomes hardware. I hope you'll take a few minutes to reread section 10.4 and remind yourself that synthesis optimization can make signals from your HDL seem to disappear.

Specifically, recall that synthesis optimization transformed the unoptimized circuit shown in Fig 10.7 into the circuit shown in Fig 10.8 using *LUT-combination* and *register-merging*. As are result, the HDL signals called SIG4, SIG5, and SIG7 no longer have a direct representation in hardware. As discussed in section 10.4, these missing signals can cause confusion and can occasionally cause real (but usually minor) problems that need to be addressed.

For example, if (in the Fig 10.7 example) you had written a constraint using pins of SIG7_reg then this constraint will not be valid because SIG7_reg has been removed by synthesis optimization. One solution is to rewrite this constraint so that it uses the pins of SIG6_reg in Fig 10.8.

An example of another problem caused by synthesis optimization is explained by Fig 14.5, where merging of the parallel registers called SIG2_reg and SIG3_reg has created a path that is too long to pass timing analysis. This problem is also called the *fanout* problem. In this example, the result of merging SIG2_reg and SIG3_reg into SIG2_reg has increased the output fanout of SIG2_reg from 1 to 2.

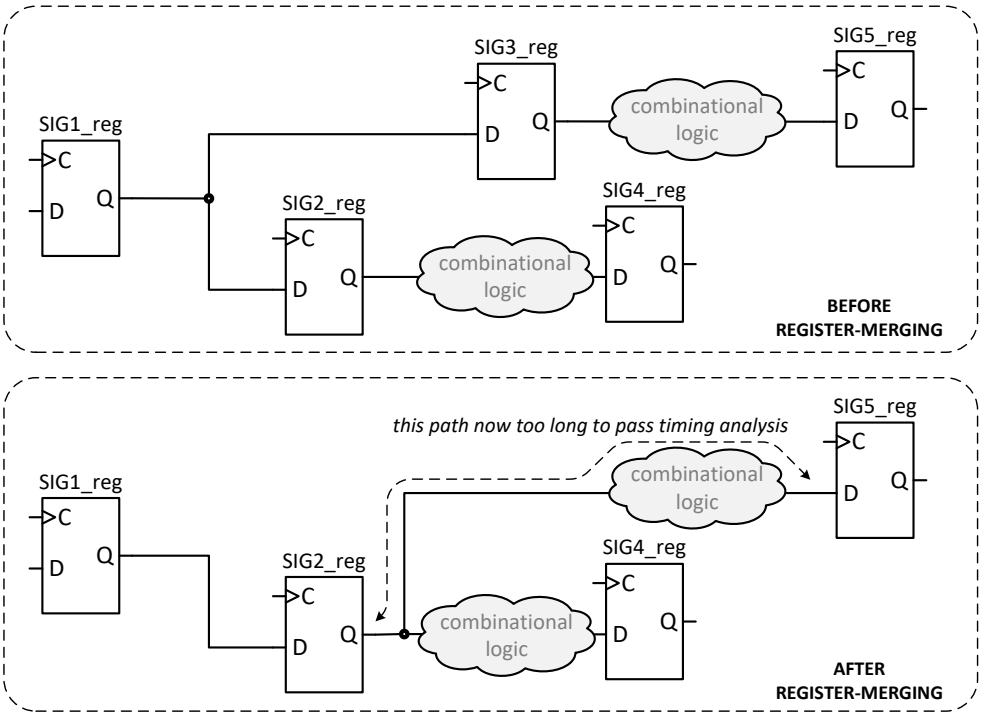

Fig 14.5 Example of timing-analysis failure caused by register-merging (aka the fanout problem).

Most of the time, synthesis optimization does good things for us. However, the fanout problem described above is an example of something done by synthesis optimization that we would sometimes like to prevent. One method is to use a synthesis setting that (available in Vivado) that prevents all merging of registers. This settings approach is generally not recommended because it is rather extreme and far-reaching. A more precise approach is to place an attribute called DONT_TOUCH=TRUE on registers that you don't want merged with other registers. Setting the DONT_TOUCH attribute is best done using VHDL code. An example of this code is shown in Fig 14.6, which can be used to prevent merging of the registers called SIG2_reg and SIG3_reg in Fig 14.5 with each other or with other registers in the design.

```
01  attribute DONT_TOUCH : string;
02  attribute DONT_TOUCH of SIG2, SIG3 : signal is "TRUE";
```

Fig 14.6 Using the DONT_TOUCH attribute in VHDL to prevent merging of SIG2_reg and SIG3_reg in Fig 14.5.

In addition to LUT-combination and register-merging, there are other types of synthesis optimization. You can read about them in the synthesis documents for your IDE. Usually, synthesis will tell us about the optimization it is doing by issuing information and warning messages. Before I used the DONT_TOUCH attribute in Fig 14.6, synthesis wrote the messages shown in Fig 14.7 to the "Message" window of the Vivado IDE. These messages explain that the registers called SIG2_reg and SIG3_reg underwent register-merging and that SIG3_reg has been removed from the design.

Finally, it was obvious from our discussions here and in section 10.4 that some HDL/hardware was redundant and could be removed (ie. optimized out). However, in big projects, redundancies happen a lot and are often not obvious. Further, synthesis usually issues lots of messages and many are not as self-evident as those shown in Fig 14.7. That is, with big projects, we must usually "have faith" that synthesis optimization is doing the right things.

```
01  [Synth 8-4471] merging register 'SIG3_reg' into 'SIG2_reg' [TOP.vhd":115]
02  [Synth 8-6014] Unused sequential element SIG3_reg was removed [TOP.vhd":115]
```

Fig 14.7 Examples of messages that describe synthesis optimization actions.

15. Level-2 FPGA I/O

As described in chapter 13, FPGA I/O is the exchange of data between an FPGA and a device that is external to the FPGA. As also described in chapter 13, when the clock associated with the I/O is slow (<40MHz) then the oversampling method is often used – mostly because oversampling is easy to understand and easy to describe in HDL. When using oversampling, it is sometimes said that timing analysis of the I/O is being satisfied "by design". However, for I/O with a fast clock, the methods described in this chapter are needed to get confirmation/proofing from timing analysis that the I/O will work properly.

Also, in section 11.3.4, I listed the three types of timing analysis problems that are typically encountered. Solutions to the first two problems on that list were addressed in chapters 11 and 12. In this chapter, we address the third problem which involves FPGA I/O. Our discussion of the FPGA I/O problem will be complex since it pulls together many concepts. So, find a nice quiet place and read on – carefully – please!

15.1 Simple at First Glance

First, let's be clear about our definition for FPGA I/O. *Fundamentally, we are talking about the transfer of data between a digital register located inside the FPGA and a digital register found in an external device*. As with all register-to-register data-transfers (see chapter 7), a clock is used. In this chapter we will assume that both registers receive the same clock. That is, we will not discuss the uncommon FPGA I/O that have a clock-crossing (see chapter 12). That completes our definition of FPGA I/O. At this point, you may be wondering what could possibly go wrong with register-to-register transfer of data. After all, this successfully happens all the time when both registers are located inside the FPGA and we hardly give it a thought. Well, I'm sorry to say that FPGA I/O is deceivingly simple at first glance.

Beginners often find that timing analysis says nothing about their FPGA I/O and wrongly conclude that the I/O will work properly. Sadly, and often much later, hardware tests can show that the I/O does not work. The simple explanation for this sad story is usually that the I/O circuit has not been properly described. As we'll learn in this chapter, *timing constraints* are used to describe the I/O circuit. Writing these constraints requires a good understanding of the timing analysis concepts in chapter 14. So, please ensure that you understand chapter 14 before moving on from here.

15.2 Check Direction

I hope that you just reviewed chapter 14 before reading this sentence. For complicated topics (like FPGA I/O), I find it helpful to frequently stop, take a breath, and review where I'm going. So, at the risk of being overly redundant, I will remind you of why I have written chapters 14 and 15. Because, for high speed FPGA I/O:

- we need timing analysis results to help us design the FPGA I/O interface and verify that it works, and
- timing analysis will analyze the FPGA I/O only if we describe it by writing proper timing constraints.

Also, as I said in section 13.1, *"Capturing/reading the data in the middle of the data-eye is perhaps the most fundamental and the most important concept of doing FPGA I/O"*. As the I/O speed increases, the size of the data-eye can get quite small and the data-eye center can be located at an unexpected position (in time). This is especially true when PVT variations are considered. However, timing analysis takes all this into account. Hence, the results of timing analysis will help us to correctly position the I/O clock edges and achieve "*capturing/reading the data in the middle of the data-eye*". Positioning the I/O clock edges in the middle of the data-eye is usually equivalent to making the I/O setup-slack and hold-slack both positive and equal.

Finally, here is my last (redundant) explanation of our direction forward. Specifically, we must learn how to:

1. **Design the Interface** (sometimes using special FPGA components)
2. **Shift the I/O Clock** (so that clock capture-edges lie approximately in the middle of the data-eye)
3. **Write Timing Constraints** (that describe the I/O circuit)
4. **Run Timing Analysis** (on the I/O interface)
5. **Check Results** (of timing analysis):
    a. If the I/O interface *passes* timing analysis, then you are done.
    b. If the I/O interface *fails* timing analysis, then:
        i. Use results of timing analysis to make setup-slack and hold-slack both positive and equal.
        ii. Return to step-4 and iterate until the interface passes timing analysis.

15.3 Some New Things

Before moving forward, as described in section 15.2, we need to learn some new terms. First, I remind you that FPGA I/O is defined here as the simple transfer of data from one digital register to another, with both registers using the same clock. So, in simplest terms (ie. 1-bit data), FPGA I/O uses only two wires; one for the data and one for the clock. For most of the discussions that follow, we will continue to talk about this 1-bit FPGA I/O example since it demonstrates all the things we need to learn.

15.3.1 Source-synchronous

There are two general categories for FPGA I/O called *source-synchronous* and *system-synchronous*. For system-synchronous I/O, the clock source is located external to both the FPGA and to the device containing the external register. For source-synchronous I/O, the clock is generated by the *source-device*. That is, the source-device (either FPGA or external device) is responsible for sending both the clock and the data to the receiving device. Often, the clock associated with source-synchronous I/O is called the *data-clock*. I have found source-synchronous I/O to be both the most common and the easiest to understand/use of the two I/O types. All the discussions that follow will focus on source-synchronous FPGA I/O. We will not discuss system-synchronous FPGA I/O in this book.

15.3.2 Center-Aligned and Edge-Aligned

In section 15.2, I emphasized the importance of placing the I/O clock edges so that data is read in the middle of the data-eye. When this relationship between clock and data is achieved, we say (in FPGA lingo) that the clock and data are *center-aligned*. So, our goal is to have things center-aligned at the receiving(capture) register in an FPGA I/O interface.

Let's assume that the receiving register is like most registers and captures data on the rising-edge of the clock. If data coming into the receiving register is changing when the clock rising-edge enters the register, then we say that the clock and data are *edge-aligned* at the register. In some respects, edge-aligned is the opposite of center-aligned. At the input to a register, the center-aligned condition usually means that the path coming into the register will *pass* timing analysis. However, the edge-aligned condition usually means that the path coming into the register will *fail* timing analysis and that the register will become metastable (see section 12.2).

In the rest of this book, I will use the words "center-aligned" to mean that the "I/O clock rising-edge is positioned in the middle of the data-eye". So, please remember that at the receiving register found in our FPGA I/O interface, we want clock and data to be center-aligned and not edge-aligned (see Fig 15.1 for details).

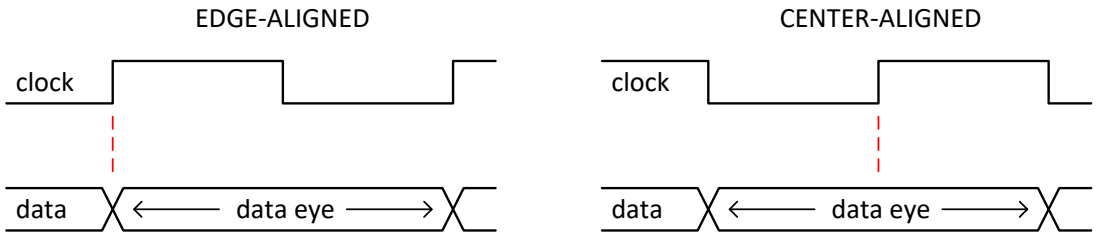

Fig 15.1 Examples of clock and data waveforms that are edge-aligned and center-aligned.

15.3.4 The I/O Block

In most FPGAs, there is a block of components beside every FPGA pin, which we will call the I/O Block (IOB). For our discussions, we'll focus on the IOB found in Xilinx FPGAs. Each IOB contains several components that are often used for FPGA I/O. One of these often-used components is called the IOB-register. When our HDL specifically requests use of IOB-register for FPGA I/O, then after implementation we can be certain of the physical location for the register (ie. it will always be in the IOB next to the FPGA pin we are using for I/O). When our HDL does not specifically request the IOB-register, then implementation is free to choose from the many registers found throughout the FPGA fabric (see section 10.3). Further, separate implementations may select registers at separate locations. So, why should we care about the physical location of an FPGA register used for FPGA I/O? Because, changes in physical location of the FPGA register can change the length/delay for the data-path and the clock-path associated with the FPGA I/O. These changes in length/delay can upset our attempt to consistently place the edges of the data-clock in the middle of data-eye. So, *for FPGA I/O, we should try to use components found in the IOB*. In the rest of this section, 15.3, some of the commonly used IOB components are described.

15.3.5 IOB-Register

Recall that FPGA I/O is the transfer of data between two registers; one found inside the FPGA and one found outside the FPGA. After our discussion in section 15.3.4, it should be clear that a register found in the IOB is the perfect choice for the I/O register found inside the FPGA. VHDL attribute statements can be used to specify use of the IOB-register. For example, the following attribute statements specify that the IOB register is to be used for the register (probably called DAT_reg by synthesis) associated with a VHDL signal called DAT.

```
attribute IOB : string;
attribute IOB of DAT: signal is "TRUE";
```

Of course, there are many IOB-registers in the FPGA. However, if your HDL connects the DAT signal to a FPGA pin/port, then implementation is smart enough to select the correct IOB-register for DAT (ie. the IOB register located closest to the pin). Later, I'll give more examples of the VHDL attribute statement and of using the IOB-register.

15.3.6 ODDR

In the IOB, we also find a block of components called the ODDR (Output Dual Data Rate). We can think of the ODDR as a special kind of digital register with two data inputs, D1 and D2, as shown in Fig 15.2.

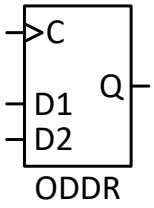

Fig 15.2 Simplified symbol for the ODDR.

Typical operation of the ODDR is easily explained as follows. When a *rising*-edge appears at clock input, C, then the data appearing at input, D1, is transferred to output, Q. When a *falling*-edge appears at clock input, C, then the data appearing at input, D2, is transferred to output, Q. When you connect output, Q, of the ODDR to an FPGA pin/port then implementation is smart enough to select the correct ODDR (ie. the ODDR located closest to the pin).

Although the ODDR was specifically designed for a special kind of FPGA I/O called Double Data Rate (DDR), it is also convenient to use the ODDR for Single Data Rate (SDR) I/O. Specifically, for source-synchronous output from the FPGA, the ODDR is frequently used to send the clock signal to the external device. In FPGA I/O lingo, the clock signal sent to the external device is called the *forwarded-clock*. For example, if we fix D1=1 and D2=0 on the ODDR, then the forwarded-clock (ie. the ODDR output, Q) will have 50% duty-cycle and will have the same period and same phase as the

clock presented to the ODDR input, C. Also useful is the situation where we fix D1=0 and D2=1. Then, the forwarded-clock will be an inverted (ie. phase-shifted by 180 deg) version of the clock presented to the ODDR input, C.

Perhaps you can see how the clock-inverting capability of the ODDR will be useful for FPGA I/O? No worries if you can't since I'll explain later. Also, you may be wondering why the clock-copy capability of the ODDR is useful. That is, why not directly forward the clock itself instead of using the ODDR to forward a copy of a clock? Well, there are two reasons why forwarding the clock directly is a bad idea. First, some FPGA I/O (eg. DDR) require the forwarded-clock to be 50% duty-cycle. So, the ODDR is a convenient way to ensure 50% duty-cycle. Second, we learned in section 8.3 that clocks are routed through the FPGA using special pathways called the clock tree. In some FPGAs, it is impossible to route a clock from the tree to an FPGA pin or to anything other than the clock-input of a component (eg. the input, C, of a register). Hence, using the ODDR to "*get a clock out of the clock tree*" is considered good practice and is sometimes necessary.

Xilinx has made it easy to use their ODDR by placing a VHDL component called ODDR in their UNISIM library. So, using the ODDR is done by simply instantiating it into your VHDL. Later (see section 15.4.1), I'll give examples of this.

15.3.7 IDDR

In what we are calling the IOB, we also find a block of components called the IDDR (Input Dual Data Rate). In fact, the IOB-register discussed in section 15.3.5 is one of the components found in the IDDR. We can think of the IDDR as a special kind of digital register with two data outputs, Q1 and Q2, as shown in Fig 15.3.

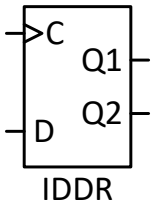

Fig 15.3 Simplified symbol for the IDDR.

Typical operation of the IDDR is easily explained as follows. When a *rising*-edge appears at clock input, C, then the data appearing at input, D, is transferred to output, Q1. When a *falling*-edge appears at clock input, C, then the data appearing at input, D, is transferred to output, Q2. As with the IOB-register, when you connect the input, D, of the IDDR to an FPGA pin then implementation is smart enough to select the correct IDDR (ie. the IDDR located closest to the pin).

The IDDR was specifically designed for a special kind of FPGA I/O called Double Data Rate(DDR), which we'll talk more about in section 16.2.

15.3.8 Differential Digital (LVDS)

Inside the FPGA, a 1-bit digital signal is almost always sent over just one wire (referenced to ground). However, for a 1-bit signal sent between the FPGA and an external device, it is common to use two wires (each referenced to ground). This two-wire digital connection is often called Low-Voltage Differential Signaling (LVDS). For a high-speed digital interface, LVDS is preferred since it both reduces electromagnetic emission (ie. noise) produced by interface wires and makes the interface less susceptible to interference caused by noise from other devices.

So, for FPGA I/O, we sometimes need to convert between 1-wire and 2-wire digital. Conveniently, Xilinx (and other FPGA vendors) have placed 1-to-2 wire converters/buffers in the IOB. Xilinx calls them IBUFDS and OBUFDS and their symbols are shown in Fig 15.4.

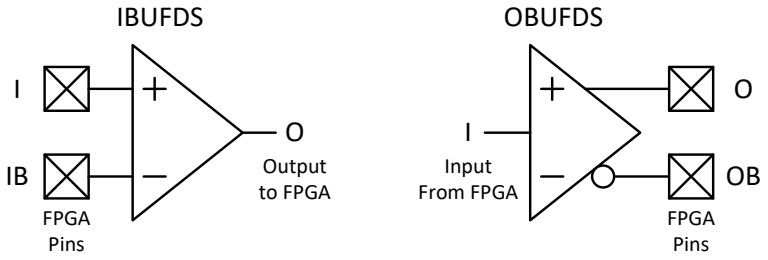

Fig 15.4 Symbols for the 1-to-2 wire converters called IBUFDS and OBUFDS found in Xilinx FPGAs.

Xilinx has made it easy to use their IBUFDS and OBUFDS by placing VHDL components called IBUFDS and OBUFDS in their UNISIM library. So, use of either component is done by simply instantiating it into your VHDL. Later (see section 16.2), I'll give examples of this.

You must follow certain rules when routing the two wires of LVDS across the circuit board between the external device and the FPGA. In short, you should run the two wires/traces side-by-side and always the same distance apart. Also, you should ensure that the two wires/traces are the same length, which can be a little tricky at places where the wires attach to pins on the FPGA or to the external device. There are other considerations for creating the LVDS traces on the board if you need ultimate performance – but you can Google LVDS for more information.

15.4 SDR Output

In this section, we'll finally get down to the nitty-gritty. That is, here we will learn to describe an FPGA I/O circuit using timing constraints. We will learn-by-example and focus on the simple (1-bit data) source-synchronous FPGA output shown in Fig 15.5. As mentioned earlier, the constraints are needed before timing analysis will analyze the I/O circuit and tell us if it works (from a timing analysis point-of-view). Some FPGA vendors provide *templates* that help with writing the needed constraints. In the rest of this section, we will learn-by-example to use the template shown in Fig 15.6, which is from the Xilinx Vivado IDE. Xilinx calls this template, "Output Delay Constraints > Source Synchronous > Setup/Hold Based > Single Data Rate(SDR), Rising-Edge".

In Fig 15.5, you'll see that the ODDR block (see section 15.3.6) is used to forward the clock to the external device. You'll also note that an IOB-register (see section 15.3.5) forwards the data (1-bit) to the external device. Finally, we'll assume that the external register is similar the register found in the familiar 74LVC374 integrated circuit, which has setup time of t_sux=2.0ns and hold time of t_hdx=1.5ns.

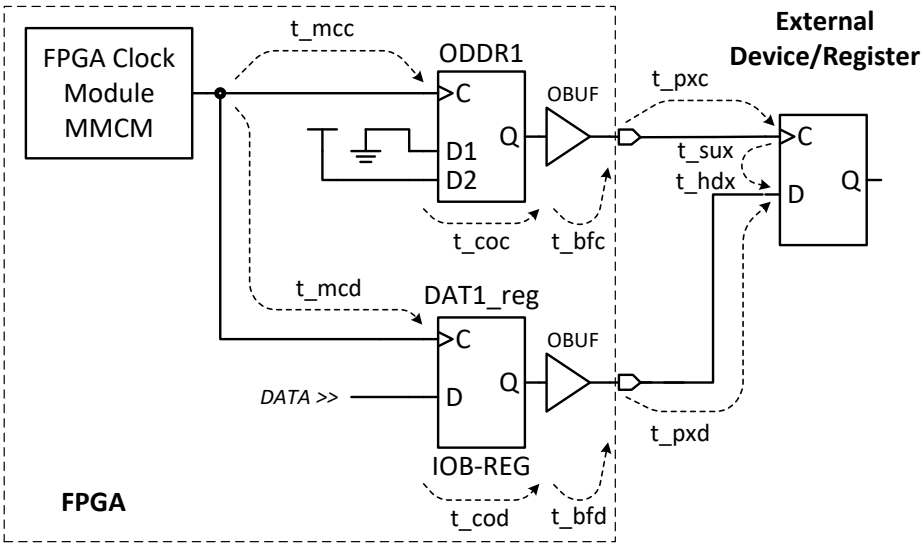

Fig 15.5 Example circuit for 1-bit source-synchronous FPGA output.

```
01  #  Rising Edge Source Synchronous Outputs
02  #
03  #  Source synchronous output interfaces can be constrained either by the max data skew
04  #  relative to the generated clock or by the destination device setup/hold requirements.
05  #
06  #  Setup/Hold Case:
07  #  Setup and hold requirements for the destination device and board trace delays are known.
08  #
09  # forwarded          ____                        ____________________
10  # clock                  |_____________________|                    |____________
11  #                                              |
12  #                                      tsu     |    thd
13  #                                  <-----------><---------->
14  #                                  _____________|____________
15  # data @ destination    XXXXXXXXX_____________________________XXXXX
16  #
17  # Example of creating generated clock at clock output port
18  # create_generated_clock -name <gen_clock_name> -divide_by 1 \
19  #       -source [get_pins <source_pin>] [get_ports <output_clock_port>]
20  # gen_clock_name is the name of forwarded clock here. It should be used below for
21  #       defining "fwclk".
22
23  set fwclk        <clock-name>;     # forwarded clock name (generated using
24                                     #    create_generated_clock at output clock port)
25  set tsu          0.000;            # destination device setup time requirement
26  set thd          0.000;            # destination device hold time requirement
27  set trce_dly_max 0.000;            # maximum board trace delay
28  set trce_dly_min 0.000;            # minimum board trace delay
29  set output_ports <output_ports>;   # list of output ports
30
31  # Output Delay Constraints
32  set_output_delay -clock $fwclk -max [expr $trce_dly_max + $tsu] [get_ports $output_ports]
33  set_output_delay -clock $fwclk -min [expr $trce_dly_min - $thd] [get_ports $output_ports]
```

Fig 15.6 Timing constraints template (uncompleted) from Xilinx Vivado IDE for FPGA source-synchronous output.

As mentioned in section 4.6, the Tcl language is used to write constraints. These constraints are placed into the Vivado project constraints file that we've been calling **constraints1.xdc**. In short, only three fundamental constraints are needed for our example FPGA output interface. The first, **`create_generated_clock`** , is shown in lines 18-19 of Fig 15.6 and is used to describe the forwarded-clock. The second and third are called **`set_output_delay`** constraints (see lines 32-33 in Fig 15.6) and are used to describe signal paths (aka connections) between the FPGA and the external device.

Also, in Fig 15.6, you will see Tcl commands starting with the word, **`set`**. They define Tcl variables (eg. fwclk) used in the **`create_generated_clock`** and **`set_output_delay`** constraints. Tcl variables and Tcl comments are a good way to document the constraints file. The symbol, #, identifies a Tcl comment. A Tcl comment can occupy an entire line as shown by line 01 in Fig 15.6. A Tcl comment can also be placed in-line (i.e. it can occupy part of a line) as shown by line 23 in Fig 15.6. The Tcl comment is a Tcl command. When you put more than one Tcl command on a line, you must end the first command on the line with a semicolon. Hence, the Tcl **`set`** constraints in lines 25-29 end with a semicolon since each has an in-line comment. Fig 15.6 also has commented-out Tcl commands, which we'll talk about below.

15.4.1 HDL Snippets

For this example of source-synchronous output from the FPGA, I have listed snippets from my VHDL in Fig 15.7. To keep netlist names simple, I have placed this VHDL in the top-level component (see section 4.6) of my VHDL project. Hence, the ports for this component (lines 10-11 in Fig 15.7) are associated with FPGA pins. Lines 04-05 in Fig 15.7 allow use of the Xilinx UNISIM library where the ODDR component is found. The ODDR component used in this interface is instantiated in lines 34-48 and is called ODDR1. Near line 29, you would insert VHDL that generates values for the data output, DAT1, of this interface. Finally, lines 20-21 ensure that the IOB-register is used to hold the DAT1 data.

```
01  library IEEE;
02  use IEEE.STD_LOGIC_1164.ALL;
03  use IEEE.NUMERIC_STD.ALL;
04  library UNISIM;              --allows use of Xilinx library where ODDR component is found
05  use UNISIM.VComponents.all;  --allows use of Xilinx library where ODDR component is found
06
07  entity TOP is
08      port(
09            --Other inputs and outputs are listed here
10             SSO1_CLK  : out std_logic;   --outgoing clock for FPGA output
11             SSO1_DAT  : out std_logic    --outgoing data for FPGA output
12           );
13  end TOP;
14
15  architecture MY_TOP of TOP is
16
17    signal CLK100 : std_logic;  --CLK100 is 100MHz clock output of the FPGA clock module, MMCM1
18    signal DAT1 : std_logic;    --DAT1 holds 1-bit data for output to external device
19
20    attribute IOB : string;
21    attribute IOB of DAT1: signal is "TRUE";  --ensures that IOB-register is used for DAT1
22
23    --Declare the MMCM clock module here
24
25  begin
26
27    --Instantiate the MMCM clock module here and call it MMCM1
28
29    --Insert HDL here that generates values for DAT1 on rising-edge of CLK100
30
31    SSO1_DAT <= DAT1;           --associate DAT1 with the FPGA port, SSO1_DAT
32
33    --The following 15 lines show how to instantiate ODDR component from the UNISIM library
34    ODDR1: ODDR
35    generic map(
36        DDR_CLK_EDGE => "OPPOSITE_EDGE", --"SAME_EDGE" or "OPPOSITE_EDGE"
37        INIT =>'0',             --initial value for Q port ('1' or '0')
38        SRTYPE => "SYNC"        --reset Type ("ASYNC" or "SYNC")
39    )
40    port map(
41        Q => SSO1_CLK,          -- 1-bit data output
42        C => CLK100,            -- 1-bit clock input
43        CE => '1',              -- 1-bit clock enable input
44        D1 => '0',              -- 1-bit data input (out-of-Q on rising-edge of C)
45        D2 => '1',              -- 1-bit data input (out-of-Q on falling-edge of C)
46        R => '0',               -- 1-bit reset input
47        S => '0'                -- 1-bit set input
48    );
49
50   end MY_TOP;
```

Fig 15.7 Snippets of VHDL for FPGA source-synchronous output (1-bit).

15.4.2 create_generated_clock

Let's discuss the **`create_generated_clock`** constraint using the Vivado-generated schematic shown in Fig 15.8 for the forwarded-clock used in our example FPGA output interface. First, note that ODDR1 is clocked by CLKOUT2 (aka CLK100), which is a 100MHz clock output from the FPGA clock-module, MMCM1 (see Fig 9.3). Next, note that I have fixed (D1=0, D2=1) on ODDR1, which means that the output of ODDR1 will be an *inverted* version of CLK100 (see ODDR discussion in section 15.3.6). Finally, this inverted version of CLK100 is sent through a digital buffer, SSO1_CLK_OBUF_inst, to the FPGA port/pin called SSO1_CLK, and out to the external device.

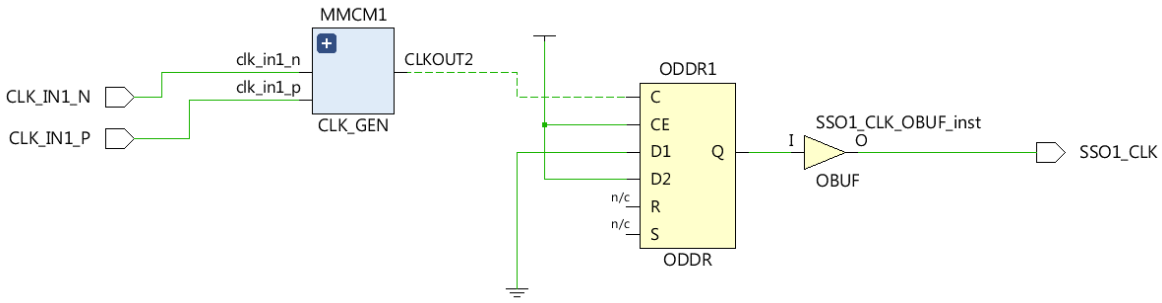

Fig 15.8 Schematic from Vivado IDE for the forwarded-clock of a FPGA source-synchronous output interface.

The basic **create_generated_clock** constraint is written as follows,

```
create_generated_clock [get_ports SSO1_CLK]
```
(15.1)

where (in this example) the object, **[get_ports SSO1_CLK]**, of the constraint identifies the FPGA port/pin used for the forwarded-clock. As shown in (15.2), we then add some easily-understood attributes that identify the name (I chose FCLK1) and the source of the forwarded-clock.

```
create_generated_clock -name FCLK1 -source [get_pins ODDR1/C] [get_ports SSO1_CLK]
```
(15.2)

Note that the source, CLKOUT2(CLK100), is indirectly identified using a component pin, ODDR1/C, to which CLKOUT2 is connected. Oddly, this constraint does not allow direct identification of the source using something like **[get_clocks CLK100..]**. Next, we'll add some not-so-easily-understood attributes to the **create_generated_clock** constraint.

One finds that **create_generated_clock** is smart is some-ways and not-so-smart in others. It is smart because after we specify the source clock (here CLKOUT2) and the output-port (here SSO1_CLK), the constraint can figure out the entire path of the forwarded-clock. However, **create_generated_clock** is not-so-smart because (together with timing analysis) it can only determine the delay associated with wires and components found along this path and it cannot determine how components along the path will modify the source clock. In this example, **create_generated_clock** will not recognize that ODDR1 is inverting the source clock. So, we must manually specify that inversion of the clock is occurring. Fortunately, we can easily do this using the "**-invert**" attribute. Similarly, **create_generated_clock** will not recognize components we *might* use to divide the source clock frequency by 2. Fortunately, this is also easily specified using the attribute, "**-divide_by**". Some FPGA tools (eg. Vivado) require that **create_generated_clock** have either the "**-divide_by**" or the "**-multiply_by**" attribute to create a forwarded-clock. For this example, I will satisfy this requirement using the do-nothing attribute, "**-divide_by 1**". Putting all this together results in the following one-line command, which we must type into the constraints file for our HDL project.

```
create_generated_clock -name FCLK1 -source [get_pins ODDR1/C] -invert -divide_by 1 [get_ports SSO1_CLK]
```
(15.3)

15.4.3 set_output_delay

Now, we'll discuss the two **set_output_delay** constraints using the Vivado-generated schematic shown in Fig 15.9 for the forwarded data used in our example FPGA output interface. First, note that the 1-bit data for this interface is called DAT1 and is forwarded from the register called DAT1_reg, through a digital buffer, SSO1_DAT_OBUF_inst, to the FPGA pin/port called SSO1_DAT, and out to the external device. As noted in section 15.4.1 and in line-21 of Fig 15.7, my HDL places the IOB=TRUE attribute on DAT1, which ensures that an IOB-register is used for DAT1_reg. It is the job of the **set_output_delay** constraints to finish describing the clock and data paths associated with the interface. Specifically, the **set_output_delay** constraints describe portions of these paths that lie outside the FPGA (ie. to the right of ports SSO1_CLK and SSO1_DAT in Fig 15.8 *and* Fig 15.9, respectively).

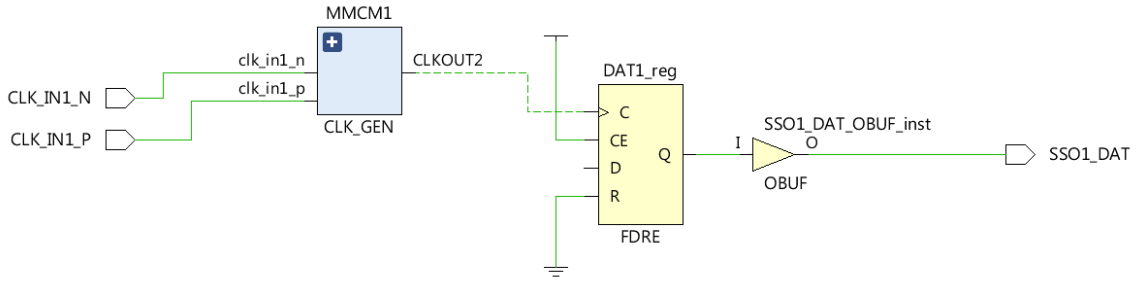

Fig 15.9 Schematic from Vivado IDE for the forwarded-data of a FPGA source-synchronous output interface.

Please again look at our timing analysis discussion in section 14.2 about register-to-register transfer of data – and keep in mind that FPGA I/O is register-to-register transfer of data. In section 14.2, we calculated data-arrival-time at the receiving register by adding up delays along a path that ended at the data pin, D, of the receiving register. So, comparing Fig 15.5 and Fig 15.9, we see that timing analysis knows about all the delays needed to calculate data-arrival-time except the delay, t_pxd, shown in Fig 15.5. Similarly, in section 14.2, we calculated data-required-time(setup) by adding up delays along a path that ended at the clock pin, C, of the receiving register. Timing analysis knows about all these delays except the delay, t_pxc, shown in Fig 15.5. Finally, in section 14.2, we calculated setup-slack as data-required-time minus data-arrival-time. Using information that is known by timing analysis about our output interface, the calculation of setup-slack would be in error by the *delay-error*, (t_pxc – t_pxd – t_sux), where t_sux is the setup time of the external register shown in Fig 15.5. The *negative* of this delay-error term is used to write one of the needed **`set_output_delay`** constraints. For example, if (t_pxd=0.3ns, t_pxc=0.2ns, t_sux=2.0ns) then delay-error(setup) is calculated as (t_pxc – t_pxd – t_sux = –2.1ns) and is used to write the following **`set_output_delay`** constraint.

```
set_output_delay -clock FCLK1 -max +2.1 [get_ports SSO1_DAT]
```
(15.4)

In (15.4), the basic constraint is identified as "**`set_output_delay [get_ports SSO1_DAT]`**", where the object, **`[get_ports SSO1_DAT]`**, identifies the FPGA port/pin to which delay-error will be assigned. The attribute, "**`-max +2.1`**", specifies that the delay-error is -2.1 (in nanoseconds). The attribute, "**`-clock FCLK1`**", specifies that timing analysis for paths connected to port, **`SSO1_DAT`**, should use the forwarded-clock, FCLK1, that was identified by the **`create_generated_clock`** constraint that we wrote in section 15.4.2.

Analysis, similar to that in the previous paragraph, shows that timing analysis will calculate hold-slack with error if it does not know the values for (t_pxd, t_pxc, t_hdx), where t_hdx is the hold time of the external register shown in Fig 15.5. Specifically, the hold-slack calculation will be in error by the delay-error, (t_pxd – t_pxc – t_hdx), which is *used directly* (rather than negating it) in another **`set_output_delay`** constraint. For example, if (t_pxd=0.3ns, t_pxc=0.2ns, t_hdx=1.5ns) then delay-error(hold) is calculated as (t_pxd – t_pxc – t_hdx = –1.4ns) and is used to write the following **`set_output_delay`** constraint.

```
set_output_delay -clock FCLK1 -min -1.4 [get_ports SSO1_DAT]
```
(15.5)

You can usually get values for t_sux and t_hdx from the data sheet for the external device/register. For the example calculations here, we've been using values from the 74LVC374 data sheet. However, you may be wondering how to get values for the delays t_pxd and t_pxc. Well, these are usually signal propagation delays through traces on the circuit board. The rule-of-thumb for trace delay is about 2ns per foot-length of trace. However, the person who designed your FPGA board may have a more accurate estimate of these trace delays for you. It may interest you to know that in old days, shifting the data-clock edges into the middle of the data-eye (ie. making things center-aligned) was done by making the board trace for the forwarded-clock longer than the board trace for the forwarded-data.

For both **`set_output_delay`** constraints (ie. (15.4) and (15.5)), we find that only the *delay-difference*, (t_pxd - t_pxc) (ie. *data-trace-delay minus clock-trace-delay),* is important rather than the individual values of t_pxd and t_pxc. Often,

some “four corners pessimism” (see section 14.4) is done when specifying this delay-difference. That is, we use a slightly-high value of (t_pxd - t_pxc) for the setup **set_output_delay** constraint and a slightly-low value for the hold **set_output_delay** constraint. For example, using (t_pxd=0.3ns, t_pxc=0.2ns), we get a nominal value of (t_pxd - t_pxc = 0.1). So, to make our setup-slack calculation more pessimistic, this value of (t_pxd - t_pxc) could be increased to 0.2, giving “**-max +2.2**” in (15.4). Also, to make our hold-slack calculation more pessimistic, this value of (t_pxd - t_pxc) could be decreased to 0.0, giving “**-min -1.5**” in (15.5).

15.4.4 Constraints Template

In sections 15.4.2 and 15.4.3, we spent lots of time discussing the **create_generated_clock** constraint and the **set_output_delay** constraints needed for the source-synchronous output interface described by Fig 15.5. The goal of those sections was to give you in-depth understanding of these constraints. However, we could have obtained the constraints using the simpler approach of filling out the template shown in Fig 15.6. That is, if we copy this template into our constraints file and complete it properly, then it will automatically become the needed constraints. The properly completed template is shown in Fig 15.10. In Fig 15.10, I have changed much of what appears in Fig 15.6 to help us focus on the important stuff and to clarify the description of some parameters.

```
01  # Source Synchronous Output, Single Data Rate (SDR), times used below are nanoseconds(ns)
02  # -------
03  set tsu       1.500;          #destination device setup time
04  set thd       1.000;          #destination device hold time
05  set tdif_max  0.100;          #max trace delay diff (data-delay minus clock-delay)
06  set tdif_min -0.100;          #min trace delay diff (data-delay minus clock-delay)
07  set dat_port  SSO1_DAT;       #name of FPGA port(s) used to forward the data
08  set clk_port  SSO1_CLK;       #name of FPGA port used to forward the clock
09  set fclk_nam  FCLK1;          #name of forwarded-clock assigned by create_generated_clock
10  set fclk_src [get_pins ODDR1/C]; #source of the forwarded-clock
11
12  # Describe the forwarded clock
13  create_generated_clock -name $fclk_nam -source $fclk_src -divide_by 1 \
14                         -invert [get_ports $clk_port ]
15
16  # Output delay constraints
17  set_output_delay -clock $fclk_nam -max [expr $tdif_max + $tsu ] [get_ports $dat_port ]
18  set_output_delay -clock $fclk_nam -min [expr $tdif_min - $thd ] [get_ports $dat_port ]
19
20  # Sometimes the following constraint is needed by MMCM to correctly interpret phase shifts
21  set_property PHASESHIFT_MODE LATENCY [get_cells MMCM1/inst/mmcm_adv_inst]
```

Fig 15.10 Timing constraints template (completed) for FPGA source-synchronous SDR output shown in Fig 15.8 and Fig 15.9.

Of course, the fill-in-the-template approach will only work if you start with the correct template. I recommend using templates as a rough guide, keeping in mind what we learned from sections 15.4.2 and 15.4.3 to ensure that the template is not leading you astray.

15.4.5 Run Timing Analysis

We are now on step-4 (Run Timing Analysis) of the work-direction outlined in section 15.2. Timing analysis is now able analyze our example FPGA output interface because we have properly described the circuits of the interface using the constraints called **create_generated_clock** and **set_output_delay**. Since our example interface is simply the register-to-register transfer of data, timing analysis will produce a timing report like the one discussed in section 14.5.

Shown in Fig 15.11 is the setup-timing report for the path between register, DAT1_reg, and the external-device-register shown in Fig 15.5. I generated the report using the following Tcl command.

```
report_timing -from [get_pins DAT1_reg/C] -to [get_ports SSO1_DAT] -setup
```
(15.6)

Unlike the Tcl command (14.1) that we used in section 14.5, (15.6) does not specify the data pin of the external register – because the pin and register have no official name. Instead, the Vivado IDE allows us to indirectly do this by specifying the FPGA port (here SSO1_DAT) that directly connects to the data pin of the external register.

Line-12 in Fig 15.11 says that the calculated setup-slack is a positive +2.379ns, indicating that the interface has passed setup timing analysis. As discussed in section 14.5, the timing path report contains numbers from the four-corners-analysis that causes the calculation of slack to be pessimistic. Let's ignore the four-corners-analysis stuff and do the simple calculation for setup-slack that we learned in section 14.2. For this *simple calculation*, we'll use the time-arc symbols (eg. t_mcd) from Fig 15.5 and we'll extract numbers we need from lines of the path report shown in Fig 15.11.

- data-arrival-time = (t_mle + t_mcd + t_cod + t_bfd + t_pxd) = (0.000 + 3.027 + 0.347 + 3.171 + 0.300) = 6.845ns.
    - t_mle = 0.000 = creation time for clock launch-edge
    - t_mcd = 3.027 (lines 43-45) = delay from MMCM1 to DAT1_reg/C
    - t_cod = 0.347 (line 48) = clock-to-output(Q) time for DAT1_reg
    - t_bfd = 3.171 (lines 49-51) = delay from DAT1_reg/Q thru OBUF to port, SSO1_DAT
    - t_pxd = 0.300 = delay of external trace that carries forwarded data to data-pin of external register
- data-required-time = (t_mce + t_mcc + t_coc + t_bfc + t_pxc) = (5.000 + 2.815 + 0.302 + 2.893 + 0.200) = 11.21ns
    - t_mce = 5.000 (line 54) = creation time for clock capture-edge that occurs *after* the clock launch-edge
    - t_mcc = 2.815 (lines 61-63) = delay from MMCM1 to ODDR1/C
    - t_coc = 0.302 (line 64) = clock-to-output(Q) time for ODDR1
    - t_bfc = 2.893 (lines 65-67) = delay from ODDR1/Q thru OBUF to port, SSO1_CLK
    - t_pxc = 0.200 = delay of external trace that carries the forwarded-clock to clock-pin of external register
- setup-slack:
    - = (data-required-time) minus (data-arrival-time) minus (external register setup time, t_sux)
    - = 11.21 – 6.845 – 2.000 = +2.365ns

Thus, our simple calculation for setup-slack of +2.365 agrees nicely with the value of +2.379 shown in line-12 of Fig 15.11. If you understood our timing analysis discussion from chapter 14 (especially section 14.5), then almost everything in the above simple calculation for setup-slack will look familiar.

One thing needing a little explanation is the value of 5.000 that I used above for t_mce. From section 14.2, we defined t_mce to be the time when the clock launch-edge was generated by MMCM1. Further, we said that the clock capture-edge will eventually cause capture of the data that was launched from DAT1_reg towards the external register by the launch-edge. So, in this example, the launch-edge is a *rising-edge* of CLK100 that is generated by MMCM1 at time, t_mle=0.000. Keeping in mind that the 100 MHz clock, CLK100, has a period of 10ns, we note that a *falling-edge* of CLK100 is generated by MMCM1 at 5.000ns. This falling-edge is inverted by ODDR1 to become an FCLK1 rising-edge, which eventually arrives at the external register and causes capture of the data that was launched from DAT1_reg by the clock launch-edge. Hence, in this example, the capture-edge is the *falling-edge* of CLK100 that is generated by MMCM1 at 5.000ns (ie. t_mce = 5.000).

```
 ------------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.3 (win64) Build 2405991 Thu Dec  6 23:38:27 MST 2018
| Date         : Thu Jun 20 10:29:42 2019
| Host         : Rosetta running 64-bit Service Pack 1  (build 7601)
| Command      : report_timing -from [get_pins DAT1_reg/C] -to [get_ports SSO1_DAT] -setup
| Design       : TOP
| Device       : 7k160t-fbg484
| Speed File   : -3  PRODUCTION 1.12 2017-02-17
------------------------------------------------------------------------------------
Timing Report

Slack (MET) :             2.379ns  (required time - arrival time)
  Source:                 DAT1_reg/C
                            (rising edge-triggered cell FDRE clocked by CLKOUT2_CLK_GEN {rise@0.000ns
                                  fall@5.000ns period=10.000ns})
  Destination:            SSO1_DAT
                            (output port clocked by FCLK1  {rise@5.000ns fall@10.000ns period=10.000ns})
  Path Group:             FCLK1
  Path Type:              Max at Slow Process Corner
  Requirement:            5.000ns  (FCLK1 rise@5.000ns - CLKOUT2_CLK_GEN rise@0.000ns)
  Data Path Delay:        3.518ns  (logic 3.518ns (100.000%)  route 0.000ns (0.000%))
  Logic Levels:           1  (OBUF=1)
  Output Delay:           2.200ns
  Clock Path Skew:        3.171ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    1.994ns = ( 6.994 - 5.000 )
    Source Clock Delay      (SCD):    -1.740ns
    Clock Pessimism Removal (CPR):    -0.563ns
  Clock Uncertainty:      0.074ns  ((TSJ^2 + DJ^2)^1/2) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Discrete Jitter          (DJ):    0.129ns
    Phase Error              (PE):    0.000ns

    Location             Delay type                Incr(ns)  Path(ns)    Netlist Resource(s)
  -------------------------------------------------------------------    -------------------
                         (clock CLKOUT2_CLK_GEN rise edge)
                                                      0.000     0.000 r
    W9                                                0.000     0.000 r  CLK_IN1_P (IN)
                         net (fo=0)                   0.000     0.000    MMCM1/inst/clk_in1_p
    W9                   IBUFDS (Prop_ibufds_I_O)     0.843     0.843 r  MMCM1/inst/clkin1_ibufgds/O
                         net (fo=1, routed)           0.962     1.805    MMCM1/inst/clk_in1_CLK_GEN
    MMCME2_ADV_X1Y1      MMCME2_ADV (Prop_mmcme2_adv_CLKIN1_CLKOUT2)
                                                     -6.572    -4.767 r  MMCM1/inst/mmcm_adv_inst/CLKOUT2
                         net (fo=1, routed)           1.652    -3.115    MMCM1/inst/CLKOUT2_CLK_GEN
    BUFGCTRL_X0Y0        BUFG (Prop_bufg_I_O)         0.080    -3.035 r  MMCM1/inst/clkout3_buf/O
                         net (fo=2, routed)           1.295    -1.740    CLK100
    OLOGIC_X0Y157        FDRE                                        r  DAT1_reg/C
  -------------------------------------------------------------------    -------------------
    OLOGIC_X0Y157        FDRE (Prop_fdre_C_Q)         0.347    -1.393 r  DAT1_reg/Q
                         net (fo=1, routed)           0.000    -1.393    SSO1_DAT_OBUF
    B22                  OBUF (Prop_obuf_I_O)         3.171     1.778 r  SSO1_DAT_OBUF_inst/O
                         net (fo=0)                   0.000     1.778    SSO1_DAT
    B22                                                         r  SSO1_DAT (OUT)
  -------------------------------------------------------------------    -------------------
                         (clock FCLK1 rise edge)      5.000     5.000 f
    W9                                                0.000     5.000 f  CLK_IN1_P (IN)
                         net (fo=0)                   0.000     5.000    MMCM1/inst/clk_in1_p
    W9                   IBUFDS (Prop_ibufds_I_O)     0.734     5.734 f  MMCM1/inst/clkin1_ibufgds/O
                         net (fo=1, routed)           0.895     6.629    MMCM1/inst/clk_in1_CLK_GEN
    MMCME2_ADV_X1Y1      MMCME2_ADV (Prop_mmcme2_adv_CLKIN1_CLKOUT2)
                                                     -5.645     0.984 f  MMCM1/inst/mmcm_adv_inst/CLKOUT2
                         net (fo=1, routed)           1.572     2.556    MMCM1/inst/CLKOUT2_CLK_GEN
    BUFGCTRL_X0Y0        BUFG (Prop_bufg_I_O)         0.072     2.628 f  MMCM1/inst/clkout3_buf/O
                         net (fo=2, routed)           1.171     3.799    CLK100
    OLOGIC_X0Y158        ODDR (Prop_oddr_C_Q)         0.302     4.101 r  ODDR1/Q
                         net (fo=1, routed)           0.000     4.101    SSO1_CLK_OBUF
    C22                  OBUF (Prop_obuf_I_O)         2.893     6.994 r  SSO1_CLK_OBUF_inst/O
                         net (fo=0)                   0.000     6.994    SSO1_CLK
    C22                                                         r  SSO1_CLK (OUT)
                         clock pessimism             -0.563     6.431
                         clock uncertainty           -0.074     6.357
                         output delay                -2.200     4.157
  -------------------------------------------------------------------
                         required time                          4.157
                         arrival time                          -1.778
  -------------------------------------------------------------------
                         slack                                  2.379
```

Fig 15.11 The Vivado *setup* timing report for the path between DAT1_reg and the external register shown in Fig 15.5.

Let's also do the simple calculation for hold-slack that we learned in section 14.2. For this *simple calculation*, we will again use the time-arc symbols (eg. t_mcd) from Fig 15.5 and we'll extract numbers we need from lines of the path report shown in Fig 15.12 that I generated using the following Tcl command.

```
report_timing -from [get_pins DAT1_reg/C] -to [get_ports SSO1_DAT] -hold
```
(15.7)

- data-arrival-time = (t_mle + t_mcd + t_cod + t_bfd + t_pxd) = (0.000 + 1.389 + 0.192 + 1.412 + 0.300) = 3.293ns.
    - t_mle = 0.000 = creation time for clock launch-edge
    - t_mcd = 1.389 (lines 43-45) = delay from MMCM1 to DAT1_reg/C
    - t_cod = 0.192 (line 48) = clock-to-output(Q) time for DAT1_reg
    - t_bfd = 1.412 (lines 49-51) = delay from DAT1_reg/Q thru OBUF to port, SSO1_DAT
    - t_pxd = 0.300 = delay of external trace that carries forwarded-data to data-pin of external register
- data-required-time = (t_mce + t_mcc + t_coc + t_bfc + t_pxc)= (-5.00 + 1.669 + 0.221 + 1.704 + 0.200) = -1.206ns
    - t_mce = -5.000 = creation time for clock capture-edge that occurs *before* the clock launch-edge
    - t_mcc = 1.669 (lines 61-63) = delay from MMCM1 to ODDR1/C
    - t_coc = 0.221 (line 64) = clock-to-output(Q) time for ODDR1
    - t_bfc = 1.704 (lines 65-67) = delay from ODDR1/Q to thru OBUF port, SSO1_CLK
    - t_pxc = 0.200 = delay of external trace that carries forwarded-clock to clock-pin of external register
- hold-slack:
    - = (data-arrival-time) minus (data-required-time) minus (external register hold time, t_hdx)
    - = 3.293 – (-1.206) – 1.500 = +2.999ns

Thus, our simple calculation for setup-slack of +2.999 agrees nicely with the value of +3.089 shown in line-12 of Fig 15.12. If you understood our timing analysis discussion from chapter 14 (especially section 14.5), then almost everything in the above simple calculation for hold-slack will look familiar.

Again, the value of -5.000 that I have used for t_mce may look odd. However, after the setup-slack calculation above, we learned that capture-edge, EDGE2, leaves the MMCM at +5.000ns. As explained in section 14.3, the start time needed for the calculation of data-required-time(hold) is the time of the capture-edge that precedes EDGE2. Since EDGE2 left the MMCM at +5.000ns and the CLK100 period is 10ns then the capture-edge that precedes EDGE2 left the MMCM at (5.000 – 10.000)= -5.000ns. Hence, I used the value of t_mce= -5.000 for the hold-slack calculations above.

```
------------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.3 (win64) Build 2405991 Thu Dec  6 23:38:27 MST 2018
| Date         : Thu Jun 20 10:31:38 2019
| Host         : Rosetta running 64-bit Service Pack 1  (build 7601)
| Command      : report_timing -from [get_pins DAT1_reg/C] -to [get_ports SSO1_DAT] -hold
| Design       : TOP
| Device       : 7k160t-fbg484
| Speed File   : -3  PRODUCTION 1.12 2017-02-17
------------------------------------------------------------------------------------
Timing Report

Slack (MET) :             3.089ns  (arrival time - required time)
  Source:                 DAT1_reg/C
                            (rising edge-triggered cell FDRE clocked by CLKOUT2_CLK_GEN {rise@0.000ns
                                  fall@5.000ns period=10.000ns})
  Destination:            SSO1_DAT
                            (output port clocked by FCLK1  {rise@5.000ns fall@10.000ns period=10.000ns})
  Path Group:             FCLK1
  Path Type:              Min at Fast Process Corner
  Requirement:            -5.000ns  (FCLK1 rise@5.000ns - CLKOUT2_CLK_GEN rise@10.000ns)
  Data Path Delay:        1.604ns  (logic 1.604ns (100.000%)  route 0.000ns (0.000%))
  Logic Levels:           1  (OBUF=1)
  Output Delay:           -1.500ns
  Clock Path Skew:        1.941ns (DCD - SCD - CPR)
    Destination Clock Delay (DCD):    1.387ns = ( 6.387 - 5.000 )
    Source Clock Delay      (SCD):    -0.520ns = ( 9.480 - 10.000 )
    Clock Pessimism Removal (CPR):    -0.034ns
  Clock Uncertainty:      0.074ns  ((TSJ^2 + DJ^2)^1/2) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Discrete Jitter          (DJ):    0.129ns
    Phase Error              (PE):    0.000ns

    Location             Delay type                Incr(ns)  Path(ns)    Netlist Resource(s)
  -------------------------------------------------------------------    -------------------
                         (clock CLKOUT2_CLK_GEN rise edge)
                                                    10.000    10.000 r
    W9                                               0.000    10.000 r  CLK_IN1_P (IN)
                         net (fo=0)                  0.000    10.000    MMCM1/inst/clk_in1_p
    W9                   IBUFDS (Prop_ibufds_I_O)    0.368    10.368 r  MMCM1/inst/clkin1_ibufgds/O
                         net (fo=1, routed)          0.503    10.871    MMCM1/inst/clk_in1_CLK_GEN
    MMCME2_ADV_X1Y1      MMCME2_ADV (Prop_mmcme2_adv_CLKIN1_CLKOUT2)
                                                    -2.780     8.091 r  MMCM1/inst/mmcm_adv_inst/CLKOUT2
                         net (fo=1, routed)          0.744     8.835    MMCM1/inst/CLKOUT2_CLK_GEN
    BUFGCTRL_X0Y0        BUFG (Prop_bufg_I_O)        0.026     8.861 r  MMCM1/inst/clkout3_buf/O
                         net (fo=2, routed)          0.619     9.480    CLK100
    OLOGIC_X0Y157        FDRE                                      r  DAT1_reg/C
  -------------------------------------------------------------------    -------------------
    OLOGIC_X0Y157        FDRE (Prop_fdre_C_Q)        0.192     9.672 r  DAT1_reg/Q
                         net (fo=1, routed)          0.000     9.672    SSO1_DAT_OBUF
    B22                  OBUF (Prop_obuf_I_O)        1.412    11.084 r  SSO1_DAT_OBUF_inst/O
                         net (fo=0)                  0.000    11.084    SSO1_DAT
    B22                                                       r  SSO1_DAT (OUT)
  -------------------------------------------------------------------    -------------------
                         (clock FCLK1 rise edge)     5.000     5.000 f
    W9                                               0.000     5.000 f  CLK_IN1_P (IN)
                         net (fo=0)                  0.000     5.000    MMCM1/inst/clk_in1_p
    W9                   IBUFDS (Prop_ibufds_I_O)    0.449     5.449 f  MMCM1/inst/clkin1_ibufgds/O
                         net (fo=1, routed)          0.553     6.002    MMCM1/inst/clk_in1_CLK_GEN
    MMCME2_ADV_X1Y1      MMCME2_ADV (Prop_mmcme2_adv_CLKIN1_CLKOUT2)
                                                    -3.209     2.793 f  MMCM1/inst/mmcm_adv_inst/CLKOUT2
                         net (fo=1, routed)          0.807     3.600    MMCM1/inst/CLKOUT2_CLK_GEN
    BUFGCTRL_X0Y0        BUFG (Prop_bufg_I_O)        0.030     3.630 f  MMCM1/inst/clkout3_buf/O
                         net (fo=2, routed)          0.832     4.462    CLK100
    OLOGIC_X0Y158        ODDR (Prop_oddr_C_Q)        0.221     4.683 r  ODDR1/Q
                         net (fo=1, routed)          0.000     4.683    SSO1_CLK_OBUF
    C22                  OBUF (Prop_obuf_I_O)        1.704     6.387 r  SSO1_CLK_OBUF_inst/O
                         net (fo=0)                  0.000     6.387    SSO1_CLK
    C22                                                       r  SSO1_CLK (OUT)
                         clock pessimism             0.034     6.421
                         clock uncertainty           0.074     6.494
                         output delay                1.500     7.994
  -------------------------------------------------------------------
                         required time                        -7.994
                         arrival time                         11.083
  -------------------------------------------------------------------
                         slack                                 3.089
```

Fig 15.12 The Vivado *hold* timing report for the path between DAT1_reg and the external register shown in Fig 15.5.

In chapter 14, my simple calculations for setup-slack and hold-slack used the same value for data-arrival-time. After all, there should be only one value for the time it takes data (in our example) that is launched from DAT1_reg to arrive at

the external register. So, some of you may be wondering why I have calculated data-arrival-time twice in this section. Further, you may be wondering why the two values, (6.845, 3.293), that I calculated for data-arrival-time are not the same!? Well, the difference between the two calculated values of data-arrival-time results from the four-corners stuff that I mentioned in section 14.4 which causes timing analysis to produce pessimistic values for slack. Because of this four-corners stuff, the timing reports often show the length of a specific path to be different for setup and hold analysis.

15.4.6 Check Results

We are now on step-5 (Check Results) of the direction outlined in section 15.2. The timing path reports shown in Fig 15.11 and Fig 15.12 indicate that our example source-synchronous output interface has positive and roughly equal slack values, (2.379, 3.089), for setup and hold. Hence, the interface has passed timing analysis and we can expect the interface to perform properly. We are done!

Let's wrap-up this chapter by discussing Fig 15.13 since it nicely summarizes our discussion about the example source-synchronous output interface described in Fig 15.5. In Fig 15.13, we see that a rising-edge of CLK100 leaves MMCM1 at time, t0. This rising-edge of CLK100 is inverted by ODDR1 to become a falling-edge of FCLK1, which reaches the register external to the FPGA at time, t2. The rising-edge of CLK100 also clocks DAT1_reg, causing it to send data that reaches the external register at time, t1. Note that time, t1, also marks the data-arrival-time for the setup-slack calculation. At time, t4, the external register is clocked by the rising-edge of FCLK1 and receives the data that arrived at time, t1. Note that this data has arrived well *before* time, t3, which is the data-required-time for the setup-slack calculation. Roughly, the difference, (t4 – t3), is the setup time for the external register. More data reaches the external register at time, t6, which also marks the data-arrival-time for the hold-slack calculation. Note that this data has arrived well *after* time, t5, which is the data-required-time for the hold-slack calculation. Roughly, the difference, (t5 – t4), is the hold time for the external register.

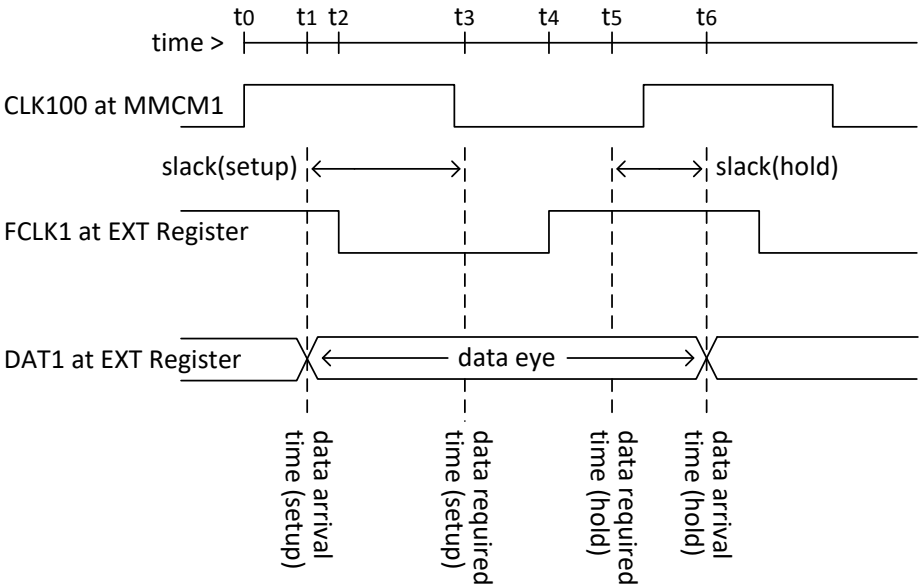

Fig 15.13 Time-plots of clock and data waveforms associated with the FPGA output interface shown in Fig 15.5.

I'm sure that understanding this chapter and chapter 14 have been a challenge for you. I remember from my early FPGA days that this understanding came to me after much reading and thought. Anyway, I hope my explanations have helped. In the next chapter, we'll discuss other types of FPGA I/O interfaces.

16. Source-Synchronous I/O

In section 15.4, we discussed our first example of FPGA I/O by looking at source-synchronous output from the FPGA. In this chapter, we'll discuss other examples of FPGA I/O.

16.1 SDR Input

Here we'll discuss the 1-bit, source-synchronous, single-data-rate (SDR), FGPA input interface shown in Fig 16.1 that uses a 100MHz clock. In Fig 16.1, you'll note use of the IOB-register (see section 15.3.5), DAT2_reg, to capture the forwarded-data from the external device. We'll assume that the external register is similar to the register found in the familiar 74LVC374 integrated circuit, which has a clock-to-output time, t_cox, of about 4.0ns. For this FPGA input, we'll probably need to shift the forwarded-clock so that rising-edges lie *approximately* in the middle of the data-eye (ie. making things center-aligned). For the FPGA-output interface discussed in section 15.4, this clock-shifting was done using the clock-inverting capability of the ODDR. For this FPGA-input interface, the clock-shifting will be done using an MMCM clock-module.

It is best (and sometimes necessary) to route a forwarded-clock from an external device into what are called a *clock-capable* pins on the FPGA. The reason being that it is easy to pass signals/clocks coming into these special pins directly to an MMCM. The data sheet for your FPGA will tell you which pins on your FPGA are clock-capable. Usually, these pins come in pairs that are labelled P and N. When your forwarded-clock is differential-digital (ie. two-wire as described in section 15.3.8), then each wire is routed to one pin of a clock-capable pin-pair. If, as in this example, your forwarded-clock is single-end (ie. one-wire) then route it to the P-pin of a clock-capable pin-pair.

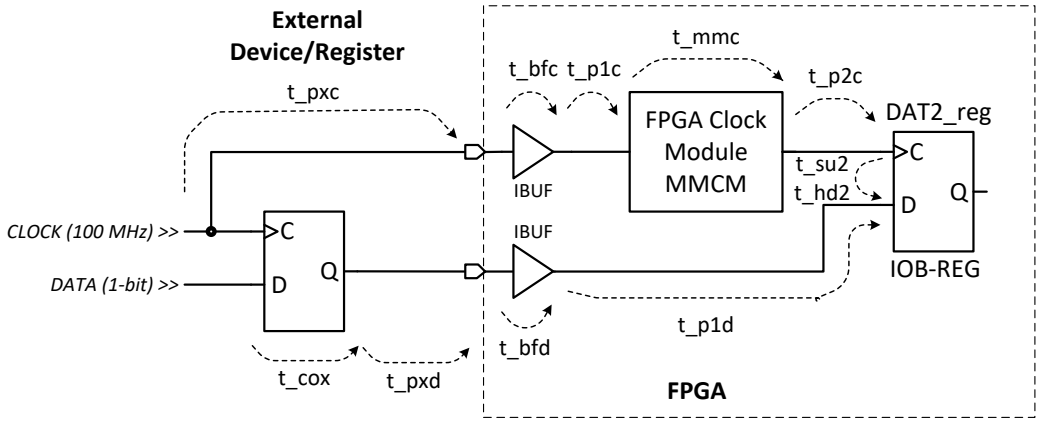

Fig 16.1 Example circuit for 1-bit source-synchronous FPGA input.

16.1.1 Configure the MMCM

The MMCM clock-module needed for this interface is the same kind of MMCM that we used in chapter 9 to create the main clocks for our FPGA project. As before, we simply go to the IP catalog in the Vivado IDE, click on the Clocking Wizard and answer all the questions that it asks. As shown in Fig 16.2, this MMCM is setup to have a single input called clk_in1 and a single output called CLK_OUT0, both having frequency, 100MHz, which is the same frequency as the forwarded-clock shown in Fig 16.1. The forwarded-clock jitter is specified to be 14 ps Pk-Pk. Usually a digital register will output data that is edge-aligned with its clock. Knowing this about our external register and knowing that data and clock need to be center-aligned for proper data-capture at the FPGA, I have initially phase-shifted CLK_OUT0 of the MMCM by 180degs with respect clk_in1.

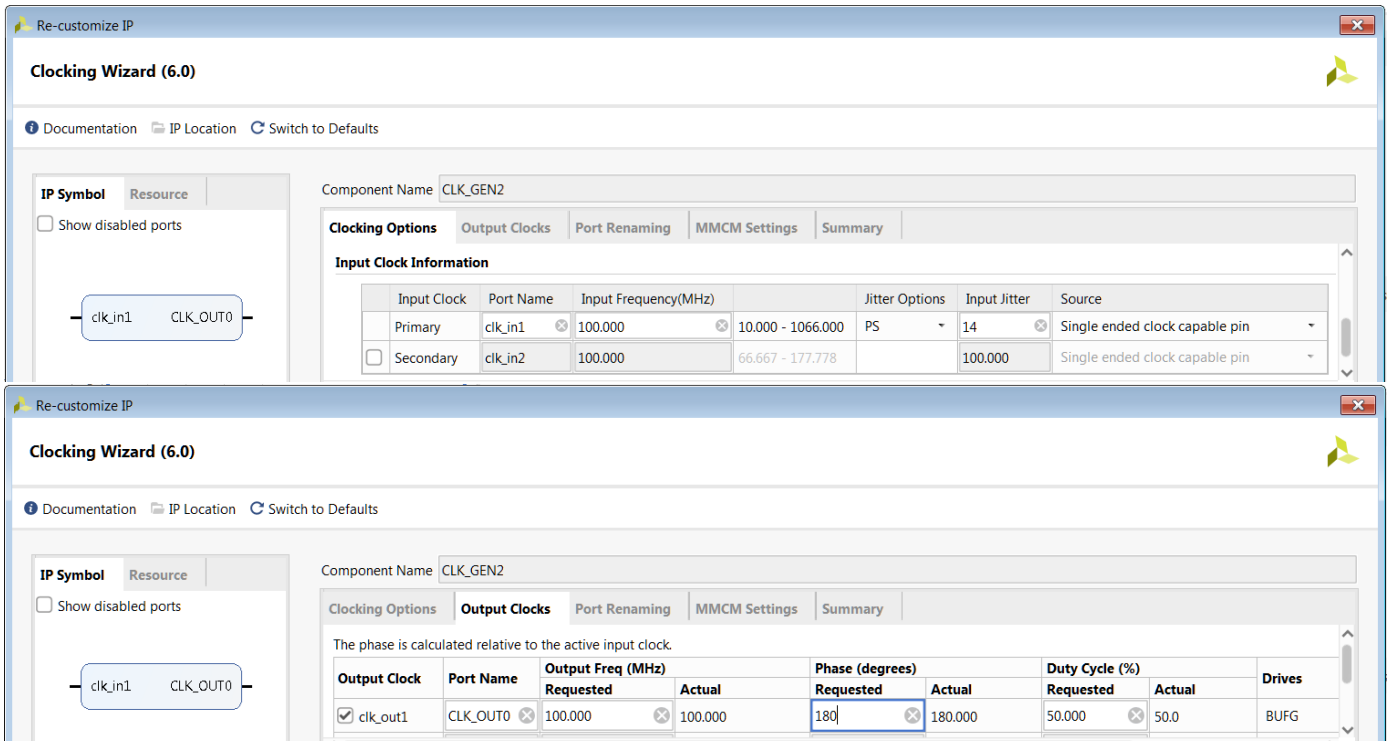

Fig 16.2 Configuring the MMCM used for the example FPGA input interface shown in Fig 16.1.

16.1.2 HDL Snippets

For this example of a source-synchronous FPGA input, I have listed snippets from my VHDL in Fig 16.3. To keep netlist names simple, I placed this VHDL in the top-level component (see section 4.6) for my VHDL project. Hence, the ports for this component (lines 08-09 in Fig 16.3) are associated with FPGA pins. The MMCM module that we configured in section 16.1.1 is declared in lines 22-27 and instantiated in lines 32-36. As indicated by comments in line 21 and line 31, the VHDL needed to declare and instantiate the MMCM can be found in a file called CLK_GEN2.vho, which was automatically generated when we configured and created the MMCM. The VHDL signal, SSI2_CLKB, is declared in line-15 and associated with the output of the MMCM in line 35. The VHDL signal, DAT2, is declared in line 16 and is used in line 42 to capture the data coming from the external device on the rising-edge of SSI2_CLKB. Near line-46, you would insert VHDL that does something with the data found in DAT2. Finally, lines 18-19 ensure that the digital register, DAT2_reg, associated with DAT2 is placed in the IOB of the FPGA.

```
01  library IEEE;
02  use IEEE.STD_LOGIC_1164.ALL;
03  use IEEE.NUMERIC_STD.ALL;
04
05  entity TOP is
06      port(
07          --Other inputs and outputs are listed here
08          SSI2_CLK : in std_logic;   --incoming clock for FPGA input
09          SSI2_DAT : in std_logic    --incoming data for FPGA input
10          );
11  end TOP;
12
13  architecture MY_TOP of TOP is
14
15    signal SSI2_CLKB : std_logic; --output of clock module, MMCM2 (shifted version of SSI2_CLK)
16    signal DAT2 : std_logic;      --DAT2 holds 1-bit data coming from external device
17
18    attribute IOB : string;
19    attribute IOB of DAT2: signal is "TRUE";  --ensures that IOB-register is used for DAT2
20
21    --Declaration of IP clock module component copied from CLK_GEN2.vho
22    component CLK_GEN2
23       port(
24            clk_in1  : in  std_logic;
25            CLK_OUT0 : out std_logic
26           );
27    end component;
28
29  begin
30
31    --Instantiation of IP clock module component copied from CLK_GEN2.vho
32    MMCM2: CLK_GEN2
33       port map(
34                clk_in1  => SSI2_CLK,
35                CLK_OUT0 => SSI2_CLKB
36                );
37
38    --Latch data coming from port, SSI2_DAT, into register, DAT2, on rising edge of SSI2_CLKB
39    SSI2P: process(SSI2_CLKB)           --outputs: (DAT2)
40      begin
41          if rising_edge(SSI2_CLKB) then
42              DAT2 <= SSI2_DAT;
43          end if;
44    end process SSI2P;
45
46    --Insert code here that does something with data found in DAT2
47
48   end MY_TOP;
```

Fig 16.3 Snippets of VHDL for FPGA source-synchronous input (1-bit).

The result of running Vivado Synthesis on the HDL shown in Fig 16.3 is the circuit schematic shown in Fig 16.4. You'll find that Fig 16.4 is conceptually identical to the FPGA portion of Fig 16.1.

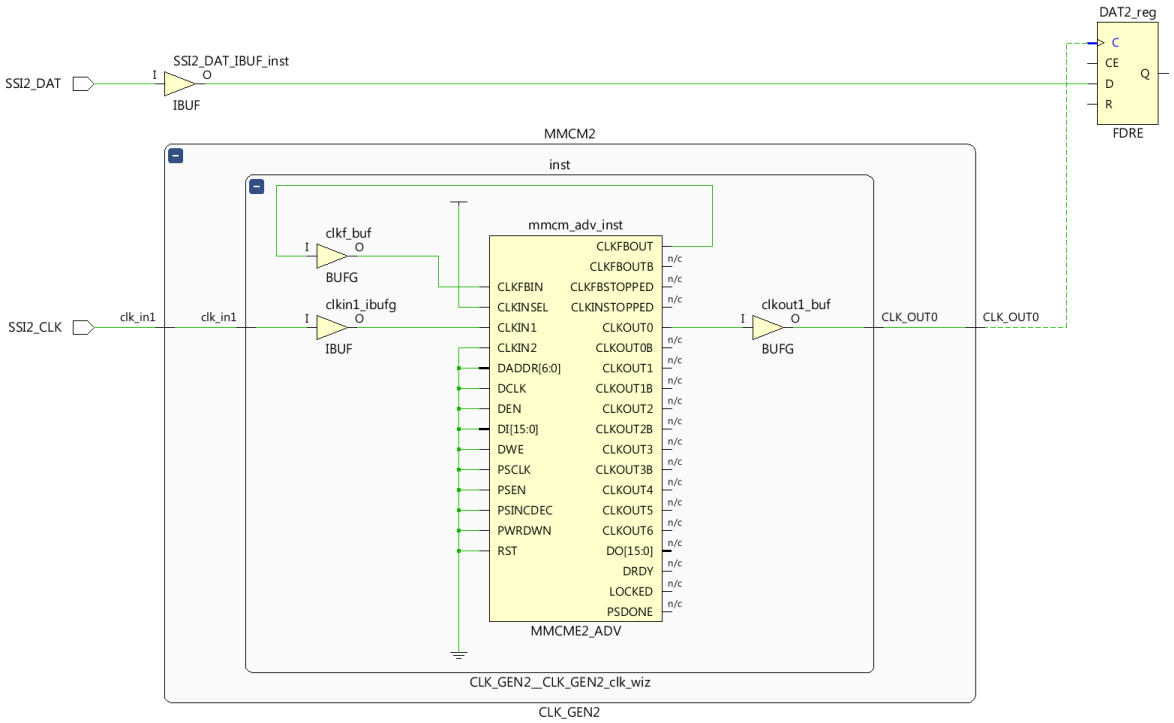

Fig 16.4 Schematic from Vivado IDE for the source-synchronous SDR input interface.

16.1.3 Timing Analysis

Rather than grabbing and using a constraints template for our FPGA input interface, we are going to create our own template since this will reinforce concepts that we learned in section 15.4. There, we learned that three basic constraints are needed for a source-synchronous *output* interface. One constraint, **`create_generated_clock`**, identified the clock used by the interface. The other two constraints, both **`set_output_delay`**, provided delay-times that timing analysis needs to correct its calculation for setup-slack and hold-slack of the interface. For our example source-synchronous *input* interface, three similar constraints are needed.

We'll start by returning to basic concepts. That is, we will do the simple calculations for slack that we learned in chapter 14 using the timing-arc designations shown in Fig 16.1. As done previously for this simple calculation, we'll ignore the four-corners-analysis stuff. So, the setup-slack simple calculations are as follows.

- data-arrival-time at DAT2_reg = (t_cle + t_cox + t_pxd + t_bfd + t_p1d)
  - t_cle = creation time for clock launch-edge
  - t_cox = clock-to-output time of external register
  - t_pxd = delay of path from output (Q-pin) of the external register to the data-input pin on the FPGA
  - t_bfd = delay of IBUF
  - t_p1d = delay of path from output of IBUF to DAT2_reg/D
- data-required-time(setup) at DAT2_reg = (t_cce + t_pxc + t_bfc + t_p1c + t_mmc + t_p2c)
  - t_cce = creation time for clock capture-edge that occurs *after* the clock launch-edge
  - t_pxc = delay from external-device clock-generator to the clock-input pin on the FPGA
  - t_bfc = delay of IBUF
  - t_p1c = delay of path from output of IBUF to input of MMCM
  - t_mmc = delay thru the MMCM
  - t_p2c = delay of path from output of MMCM to DAT2_reg/C
- setup-slack:

- = (data-required-time) minus (data-arrival-time) minus (DAT2_reg setup time, t_su2)
- = **[**t_pxc – t_pxd – t_cox**]** + **[**(t_cce + t_bfc + t_p1c + t_mmc + t_p2c) – (t_cle + t_bfd + t_p1d)**]** – t_su2

The last bulleted item above is the formula used to calculate setup-slack for our example interface. In this formula, you'll see two terms enclosed in square brackets, [..]. The first contains timing-arcs located outside the FPGA. The second contains timing-arcs located inside the FPGA. Timing analysis automatically knows about the timing-arcs found in the second square-brackets-term and about t_su2. However, timing analysis does not know about the timing-arcs found in the first square-brackets-term. So, unless we give this information to timing analysis, the setup-slack calculation for the interface will be in error by [t_pxc – t_pxd – t_cox]. In section 15.4.3 for the source-synchronous *output* interface, we used a "**`set_output_delay -max`**" constraint and the negative of a similar term to give timing analysis the information it needs. Here, for the source-synchronous *input* interface, the "**`set_input_delay -max`**" constraint and the negative of [t_pxc – t_pxd – t_cox] is used to give timing analysis the information it needs.

For the simple calculation of hold-slack, we have the following.

- data-arrival-time at DAT2_reg = (t_cle + t_cox + t_pxd + t_bfd + t_p1d) = same as for setup-slack
- data-required-time(setup) at DAT2_reg = (t_cce + t_pxc + t_bfc + t_p1c + t_mmc + t_p2c)
  - t_cce = creation time for clock capture-edge that occurs *before* the clock launch-edge
  - t_pxc = delay from external-device clock-generator to the clock-input pin on the FPGA
  - t_bfc = delay of IBUF
  - t_p1c = delay of path from output of IBUF to input of MMCM
  - t_mmc = delay thru the MMCM
  - t_p2c = delay of path from output of MMCM to DAT2_reg/C
- hold-slack:
  - = (data-arrival-time) minus (data-required-time) minus (DAT2_reg hold time, t_hd2)
  - = **[**t_cox + t_pxd – t_pxc**]** – **[**(t_cce + t_bfc + t_p1c + t_mmc + t_p2c) – (t_cle + t_bfd + t_p1d)**]** – t_hd2

The last bulleted item above is the formula used to calculate hold-slack for our example interface. In this formula, you'll see two terms enclosed in square brackets, [..]. The first contains timing-arcs located outside the FPGA. The second contains timing-arcs located inside the FPGA. Timing analysis automatically knows about the timing-arcs found in the second square-brackets-term and about t_hd2. However, timing analysis does not know about the timing-arcs found in the first square-brackets-term. So, unless we give this information to timing analysis, the hold-slack calculation for the interface will be in error by [t_cox + t_pxd – t_pxc]. In section 15.4.3 for the source-synchronous *output* interface, we used a "**`set_output_delay -min`**" constraint and a similar term to give timing analysis the information it needs. Here, for the source-synchronous *input* interface, the "**`set_input_delay -min`**" constraint and [t_cox + t_pxd – t_pxc] is used to give timing analysis the information it needs.

Thus, for the source-synchronous *input* interface, we find that both the "**`set_input_delay -max`**" constraint and the "**`set_input_delay -min`**" constraint use a value that is roughly computed as [t_cox + t_pxd – t_pxc]. From Fig 16.1, we see that [t_cox + t_pxd – t_pxc] equates to data-arrival-time minus clock-arrival-time *at the FPGA pins*. In FPGA lingo, this quantity is also called the clock-to-data *skew*. Finally, as explained in section 15.4.3, "four corners pessimism" should be used when creating the actually numbers for the **`set_input_delay`** constraints. That is, we should use a value that is a *little-larger* than [t_cox + t_pxd – t_pxc] for the "**`set_input_delay -max`**" constraint and a value that is a *little-smaller* than [t_cox + t_pxd – t_pxc] for the "**`set_input_delay -min`**" constraint. A good way to define a *little-larger* and a *little-smaller* is to use maximum and minimum values of t_cox that are specified in the datasheet for the external device.

Finally, did you notice that both **`set_output_delay`** constraints depend on the *delay-difference*, (t_pxd – t_pxc), rather than the individual values of t_pxd and t_pxc. We also found this to be true for the source-synchronous *output* interface described in section 15.4.

16.1.4 No Register?

In section 16.1.3, we found that the quantity, [t_cox + t_pxd – t_pxc], and the max/min values of t_cox are most of what's needed to write both **`set_input_delay`** constraints for our source-synchronous input interface. Nice!

However, manufacturers for external devices tend not give us the t_cox specifications! That is, although external devices like Analog-to-Digital-Converters (ADCs) use a digital register to send data to the FPGA, the ADC manufacturer describes alignment of the clock and data outputs from an ADC using specifications other than t_cox. So, our job is to convert these alternate specifications to values of t_cox that can be used in our **`set_output_delay`** constraints. Manufacturers often draw the digital output waveforms from an ADC as shown in Fig 16.5 and then specify values for setup, t_sux, and hold, t_hdx. Normally, we associate setup and hold specifications with the *input* to a digital device. So, it is odd to see the same terms associated with the *output* of a digital device. Nonetheless, the conversions we need are generally that the maximum value t_cox equals (t_clk – t_sux) and the minimum value of t_cox equals t_hdx, where t_clk is the period of the clock output from the ADC.

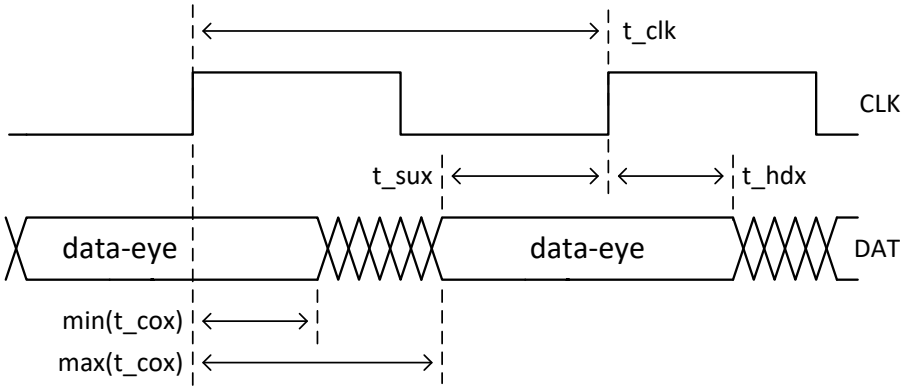

Fig 16.5 Clock and data waveforms at the output of a typical ADC.

So, in the **`set_input_delay`** constraints for our source-synchronous input interface, we find that the maximum and minimum values of [t_cox + t_pxd – t_pxc] can be computed as shown in (16.1) and (16.2).

$$\textbf{max}[t\_cox + t\_pxd - t\_pxc] = t\_clk - t\_sux + t\_pxd - t\_pxc \quad \textbf{(16.1)}$$

$$\textbf{min}[t\_cox + t\_pxd - t\_pxc] = t\_hdx + t\_pxd - t\_pxc \quad \textbf{(16.2)}$$

Sometimes, we are told only that the external device delivers its clock and data *edge-aligned*, which implies that t_cox=0. Does this mean that the external device uses an amazing digital register with 0ns clock-to-output time? Well, no. What this probably means is that the manufacturer put extra delay on the clock output line so that clock and data will leave the device at about the same time (ie. are edge-aligned). However, when we are told only that outputs are edge-aligned then the bigger question is what do we use for the maximum and minimum values of t_cox? It would be wrong to simply use the same value (ie. t_cox=0) for both the maximum and minimum values, since this would under-constrain the interface (which is another way of saying – not enough four-corners stuff). So, what are we to do? Well, it is best to call the manufacturer for more information.

Similarly, we are sometimes told only that the external device delivers its clock and data *center-aligned*, which implies that t_cox equal one-half the period of the forwarded-clock. Again, you should call the manufacturer of the external device for more information.

All our talk about **`set_input_delay`** constraints in this and the previous section will be pulled together in section 16.1.6 when we make our own constraints template for a source-synchronous input interface.

16.1.5 create_generated_clock

In section 15.4.2 we learned that a **create_generated_clock** constraint is needed for the source-synchronous *output* interface. This constraint is also needed for a source-synchronous *input* interface. However, for the particular interface described by Fig 16.1, we don't have to write the constraint ourselves. Why? Because, when we created and configured the MMCM in section 16.1.1, a **create_generated_clock** constraint was automatically generated for us. Where is it? Well, the Vivado IDE that I'm using hides the constraint from our view. Like the **create_generated_clock** constraint shown in section 15.4.2, this hidden constraint informs timing analysis that the output of MMCM2 is an inverted (for now) version of the interface clock, SSI2_CLK. If we'd needed to write this constraint ourselves, then (with the help of Fig 16.4) it might look something like the following.

```
create_generated_clock -name FCLK2 -source [get_pins MMCM2/inst/mmcm_adv_inst/CLKIN1] \
                       -invert -divide_by 1 [get_pins MMCM2/inst/mmcm_adv_inst/CLKOUT0]
```
(16.3)

However, since the auto-generated **create_generated_clock** constraint is hidden, we cannot see the "**-name**" that the constraint has assigned to forwarded clock. Use of the following Tcl command will get the name of the forwarded clock that is coming into the FPGA port called SSI2_CLK.

```
[get_clocks -of_objects [get_ports SSI2_CLK]]
```
(16.4)

Also, when we created the MMCM in section 16.1.1, a **create_clock** constraint was automatically written for us, which looked something like the following.

```
create_clock -period 10.000 [get_ports clk_in1]
```
(16.5)

If MMCM2 in Fig 16.4 had not been used, then we'd need to type the **create_clock** constraint ourselves into the constraints file, **constraints1.xdc**, for our Vivado project, my_proj1.

All our talk about the **create_generated_clock** constraint in this section will be pulled together in section 16.1.6 when we make our own constraints template for a source-synchronous input interface.

16.1.6 Constraints Template

For the source-synchronous input interface shown in Fig 16.4, constraints can be written using the template shown in Fig 16.6. If you've understood our discussions in sections 16.1.3-thru-16.1.5 then the template should make sense to you.

The template shows the two needed **set_input_delay** constraints in lines 14-15. As discussed in section 16.1.5, the **create_generated_clock** constraint should not be placed in the template because this constraint was automatically written (and hidden) when we created the MMCM in section 16.1.1.

Lines 04-12 of the template define Tcl variables that identify terms needed in the **set_input_delay** constraints. The maximum and minimum values for t_cox are shown in lines 04-05. They are from the datasheet for the digital register found in the familiar 74LVC374 integrated circuit. If your source-synchronous input interface has an external device without t_cox specifications then the discussion of t_sux and t_hdx in section 16.1.4 may help you get the needed t_cox specifications.

```
01  # Source Synchronous Input, Single Data Rate (SDR), times used below are nanoseconds(ns)
02  # ( an MMCM is used to shift the forwarded-clock with respect to the forwarded data )
03  # -------
04  set tcox_max   7.000;        #max clock-to-output time of external device
05  set tcox_min   1.500;        #min clock-to-output time of external device
06  set tdif_max   0.200;        #max trace delay diff (data-delay minus clock-delay)
07  set tdif_min   0.000;        #min trace delay diff (data-delay minus clock-delay)
08  set dly_max   [expr $tcox_max + $tdif_max ]
09  set dly_min   [expr $tcox_min + $tdif_min ]
10  set dat_port  {SSI2_DAT};    #name of FPGA port(s) where forwarded-data enters FPGA
11  set clk_port   SSI2_CLK;     #name of FPGA port where forwarded-clock enters FPGA
12  set fclk_nam [get_clocks -of_objects [get_ports $clk_port ]]; #net-name of forwarded-clock
13  # --
14  set_input_delay -clock $fclk_nam -max $dly_max [get_ports $dat_port ]
15  set_input_delay -clock $fclk_nam -min $dly_min [get_ports $dat_port ]
16  # -------
17  # Sometimes the following constraint is needed by MMCM to correctly interpret phase shifts
18  set_property PHASESHIFT_MODE LATENCY [get_cells MMCM2/inst/mmcm_adv_inst]
```

Fig 16.6 Timing constraints template (completed) for FPGA source-synchronous SDR input shown in Fig 16.4.

16.1.7 Timing Path Report

Timing analysis will now analyze our source-synchronous interface because we have properly described the interface using the constraints shown in Fig 16.6. Also, recall from section 16.1.1 that we configured the MMCM to shift the forwarded-clock by 180 degrees (ie. inverting the clock). This 180-degree clock-shift was our first attempt to ensure that *clock and data are center-aligned* at the capture register, DAT2_reg, shown in Fig 16.1. If you are uncertain why the clock and data need to be center-aligned then please reread sections 15.1, 15.2, and 15.3.2.

So, with these constraints and the 180-degree clock shift, my first run of timing analysis reports a negative value for hold-slack. Of course, this means that our example interface fails timing analysis. Why? Well, mostly because of the large values for t_cox. I am calling the values, (1.5ns and 7.0ns), for t_cox large because they comparable in size to the period, 10.0ns, of the 100 MHz forwarded-clock, SSI2_CLK.

The good news is that we can make this interface pass timing analysis by fiddling with the clock shift done by the MMCM. You'll note that the current shift of 180 degrees equates to a 5.0ns delay for our 100 MHz clock. Since t_cox of the external register is delaying our data by about [(7.0+1.5)/2=4.2ns], then clock and data are almost edge-aligned at DAT2_reg. We know from section 15.3.2 that this is the opposite of what we need (ie. we need center-aligned clock and data at DAT2_reg). After some trial-and-error testing, I found that telling the MMCM to shift the forwarded-clock by 25 degrees instead of 180 degrees gave center-aligned clock and data at DAT2_reg. That is, with a forwarded-clock shift of 25 degrees, timing analysis reported nearly equal values of +1.3ns for setup-slack and hold-slack. This interface now passes timing analysis!

Someday, you may find an interface that passes timing analysis when the MMCM is told to shift the forwarded-clock by 0 degrees. So, can we get rid of the MMCM since it no longer shifts the forwarded-clock (ie. it is shifting the forwarded-clock by 0 degrees)? Well, no. Here are reasons why we need to keep the MMCM.

1. All clocks should be routed into the FPGA clock tree (see section 8.3). The MMCM does this automatically for us.
2. If we remove the MMCM, chances are that the clock shift will not be 0 degrees. This is because removal of the MMCM will reroute and reconfigure things inside the FPGA and have an unpredictable effect on the clock phase.
3. Any clock shift (including 0 degrees) needed for the interface must remain stable over PVT for the interface to work properly. The MMCM provides a stable clock shift by *automatically compensating for changes in PVT*.

Finally, Fig 16.7 shows the setup-timing report for the path between the external register and DAT2_reg in Fig 16.1, which I generated using the following Tcl command.

```
report_timing -from [get_ports SSI2_DAT] -to [get_pins DAT2_reg/D] -setup
```
(16.6)

Again, let's do the simple calculation for setup-slack that we learned in section 14.2. For this calculation, we'll use the time-arc symbols from Fig 16.1 and we'll extract values for the time-arcs from lines of the path report shown in Fig 16.7.

- data-arrival-time at DAT2_reg = (t_cle + t_cox + t_pxd + t_bfd + t_p1d) = 8.720 ns
  - t_cle = 0.000 = creation time for clock launch-edge
  - t_cox = 7.000 = clock-to-output time of external register
  - t_pxd = 0.300 = delay of path from output (Q) of external register to data-input pin on FPGA
  - t_bfd = 1.420 (line 43) = delay of IBUF
  - t_p1d = 0.000 (line 44) = delay of path from output of IBUF to DAT2_reg/D
- data-required-time(setup) at DAT2_reg = (t_cce + t_pxc + t_bfc + t_p1c + t_mmc + t_p2c) = 10.272 ns
  - t_cce = 10.000 = creation time for clock capture-edge that occurs *after* the clock launch-edge
  - t_pxc = 0.200 = delay from external-device clock-generator to the clock-input pin on the FPGA
  - t_bfc = 1.284 (line 51) = delay of IBUF
  - t_p1c = 0.895 (line 52) = delay of path from output of IBUF to input of MMCM2
  - t_mmc = -4.625 (line 54) = delay thru the MMCM2
  - t_p2c = 2.518 (lines 55-57) = delay of path from output of MMCM2 to DAT2_reg/C
- setup-slack:
  - = (data-required-time) minus (data-arrival-time) minus (DAT2_reg setup time from line 61)
  - = 10.272 – 8.720 – (–0.010) = 1.562 ns

Thus, our simple calculation for setup-slack of +1.562 agrees nicely with the value of +1.260 shown in line-12 of Fig 16.7.

```
------------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.3 (win64) Build 2405991 Thu Dec  6 23:38:27 MST 2018
| Date         : Sat Jun 22 16:36:01 2019
| Host         : Rosetta running 64-bit Service Pack 1  (build 7601)
| Command      : report_timing -from [get_ports SSI2_DAT] -to [get_pins DAT2_reg/D] -setup
| Design       : TOP
| Device       : 7k160t-fbg484
| Speed File   : -3  PRODUCTION 1.12 2017-02-17
------------------------------------------------------------------------------------
Timing Report

Slack (MET) :             1.260ns  (required time - arrival time)
  Source:                 SSI2_DAT
                            (input port clocked by SSI2_CLK  {rise@0.000ns fall@5.000ns period=10.000ns})
  Destination:            DAT2_reg/D
                            (rising edge-triggered cell FDRE clocked by CLK_OUT0_CLK_GEN2 {rise@0.000ns
                                  fall@5.000ns period=10.000ns})
  Path Group:             CLK_OUT0_CLK_GEN2
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            10.000ns  (CLK_OUT0_CLK_GEN2 rise@10.000ns - SSI2_CLK rise@0.000ns)
  Data Path Delay:        1.420ns  (logic 1.420ns (100.000%)  route 0.000ns (0.000%))
  Logic Levels:           1  (IBUF=1)
  Input Delay:            7.200ns
  Phase Shift in Clock Latency:
    Destination Clock:    0.694ns
  Clock Path Skew:        0.073ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    0.073ns = ( 10.073 - 10.000 )
    Source Clock Delay      (SCD):    0.000ns
    Clock Pessimism Removal (CPR):    0.000ns
  Clock Uncertainty:      0.182ns  ((TSJ^2 + TIJ^2 + DJ^2)^1/2) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.014ns
    Discrete Jitter          (DJ):    0.136ns
    Phase Error              (PE):    0.105ns

    Location             Delay type                Incr(ns)  Path(ns)    Netlist Resource(s)
  -------------------------------------------------------------------    -------------------
                         (clock SSI2_CLK rise edge)
                                                      0.000     0.000 r
                         input delay                  7.200     7.200
    B18                                               0.000     7.200 r  SSI2_DAT (IN)
                         net (fo=0)                   0.000     7.200    SSI2_DAT
    B18                  IBUF (Prop_ibuf_I_O)         1.420     8.620 r  SSI2_DAT_IBUF_inst/O
                         net (fo=1, routed)           0.000     8.620    SSI2_DAT_IBUF
    ILOGIC_X0Y160        FDRE                                       r  DAT2_reg/D
  -------------------------------------------------------------------    -------------------
                         (clock CLK_OUT0_CLK_GEN2 rise edge)
                                                     10.000    10.000 r
    C17                                              0.000    10.000 r  SSI2_CLK (IN)
                         net (fo=0)                  0.000    10.000    MMCM2/inst/clk_in1
    C17                  IBUF (Prop_ibuf_I_O)        1.284    11.284 r  MMCM2/inst/clkin1_ibufg/O
                         net (fo=1, routed)          0.895    12.179    MMCM2/inst/clk_in1_CLK_GEN2
    MMCME2_ADV_X0Y3      MMCME2_ADV (Prop_mmcme2_adv_CLKIN1_CLKOUT0)
                                                    -4.625     7.555 r  MMCM2/inst/mmcm_adv_inst/CLKOUT0
                         net (fo=1, routed)          1.278     8.833    MMCM2/inst/CLK_OUT0_CLK_GEN2
    BUFGCTRL_X0Y16       BUFG (Prop_bufg_I_O)        0.072     8.905 r  MMCM2/inst/clkout1_buf/O
                         net (fo=2, routed)          1.168    10.073    SSI2_CLKB
    ILOGIC_X0Y160        FDRE                                       r  DAT2_reg/C
                         clock pessimism             0.000    10.073
                         clock uncertainty          -0.182     9.891
    ILOGIC_X0Y160        FDRE (Setup_fdre_C_D)      -0.010     9.881    DAT2_reg
  -------------------------------------------------------------------
                         required time                         9.881
                         arrival time                         -8.620
  -------------------------------------------------------------------
                         slack                                 1.260
```

Fig 16.7 The Vivado *setup* timing report for the path between external register and DAT2_reg shown in Fig 16.1.

16.2 DDR Output

The Double Data Rate (DDR) source-synchronous interface is used often, especially with computer memory. This interface has evolved much over the years and has several features that make it faster than the Single Data Rate (SDR) source-synchronous interface. One feature is that DDR sends data on both edges (rising and falling) of the forwarded-clock, whereas SDR sends data only on one edge (usually the rising edge) of the clock. Thus, for a specific clock speed, the DDR interface can potentially transfer data twice as fast as the SDR interface.

We'll learn more about DDR by discussing the example 1-bit source-synchronous output interface shown in Fig 16.8 that uses a 100MHz clock called CLK100. As always, I've drawn lots of timing-arcs (dashed lines with arrows) in Fig 16.8, which we'll use later for simple timing analysis calculations. You'll notice this interface uses the ODDR (see section 15.3.6). Understanding the DDR interface is easier if we know something about the inner-workings of the ODDR. So, in the next section, we'll talk more about the ODDR. Finally, you'll notice that the external device shown in Fig 16.8 has two registers connected to the interface, instead of the usual one register. I'll talk more about this in section 16.2.1.

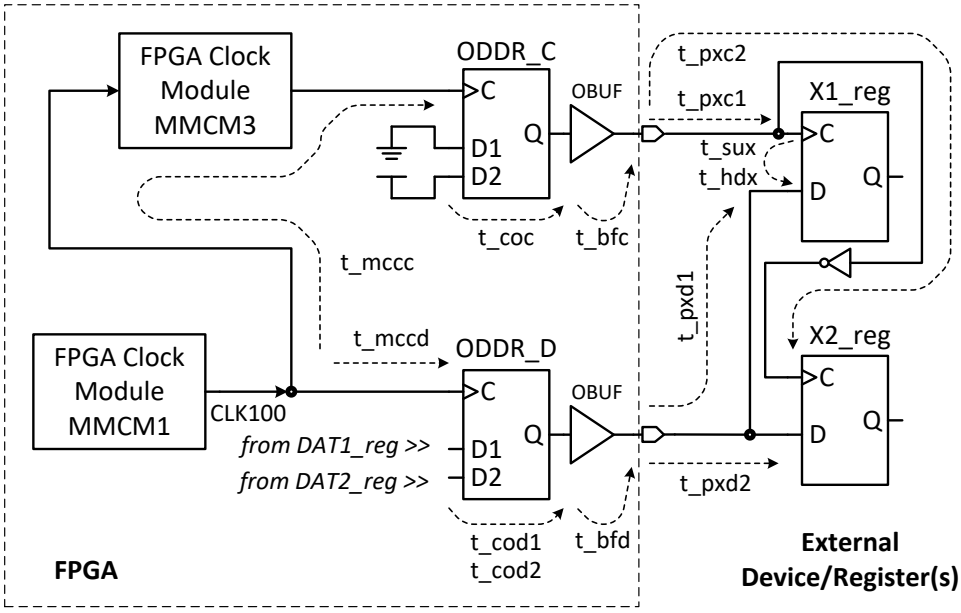

Fig 16.8 FPGA source-synchronous 1-bit output interface with DDR.

16.2.1 Inside ODDR and IDDR

Actual construction of the ODDR is proprietary information. However, Fig 16.9 shows a circuit that conceptually describes ODDR traditional operation. The circuit contains two registers that I've labelled R1 and R2. The register called R1 receives the clock from pin-C of the ODDR and latches data from pin-D1 of the ODDR on the rising-edge of the clock. The register called R2 receives an inverted version of the clock and thus latches data from pin-D2 on falling-edge of the clock seen at pin-C. The outputs of registers R1 and R2 feed a digital multiplexer that is controlled by the clock. After a clock rising-edge, the multiplexer passes the output of R1 to pin-Q of the ODDR. After a clock falling-edge, the multiplexer passes the output of R2 to pin-Q of the ODDR.

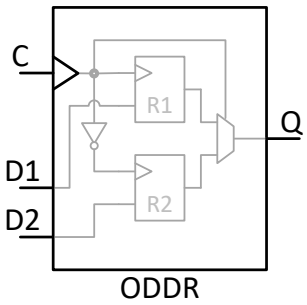

Fig 16.9 Conceptual circuit diagram for the ODDR component.

Construction of the IDDR is also proprietary information. However, Fig 16.10 shows a circuit that conceptually describes IDDR traditional operation. The circuit contains two registers that I've labelled R1 and R2. The register called R1 receives the clock from pin-C of the IDDR. Thus, R1 latches data from pin-D of the IDDR on the rising-edge of the clock and sends this data to output-Q1. The register called R2 receives an inverted version of the clock. Thus, R2 latches data from pin-D on falling-edge of the clock and sends this data to output-Q2. You'll note that the 2-register conceptual circuit for the IDDR corresponds exactly to the 2-register (X1_reg and X2_reg) circuit of the external device shown in Fig 16.8. So, Fig 16.8 indicates that the external device is effectively using an IDDR component to receive data from the DDR interface with the FPGA.

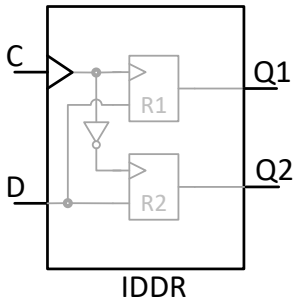

Fig 16.10 Conceptual circuit diagram for the IDDR component.

Since section 15.1, I've been reminding you that FPGA I/O is simply the transfer of data between a digital register located inside the FPGA and a digital register found in an external device. This statement is still true but perhaps not apparent at first glance of Fig 16.8. If we replace the ODDR symbols in Fig 16.8 with the ODDR conceptual circuit from Fig 16.9 then things become more apparent. That is, after a clock rising-edge, there will be register-to-register transfer of data between R1 of the ODDR and X1_reg of the external device. Also, after a clock falling-edge, there will be register-to-register transfer of data between R2 of the ODDR and X2_reg of the external device. As we'll discuss in section 16.2.4, this "*two data paths*" property of the DDR interface will affect how we write constraints for the interface.

16.2.2 Configure the MMCM

The MMCM3 in the Fig 16.8 interface is configured similarly to what we did in section 16.1.1 for the Fig 16.1 interface. One difference is that under the "Clock Options" tab (see Fig 16.11) the input to the MMCM3 will be specified as "Global buffer" instead of "Single ended clock capable pin". Here, "Global buffer" means that the input, CLK100, to MMCM3 is coming from the FPGA global clock tree (see chapter 8), which is where MMCM1 places CLK100.

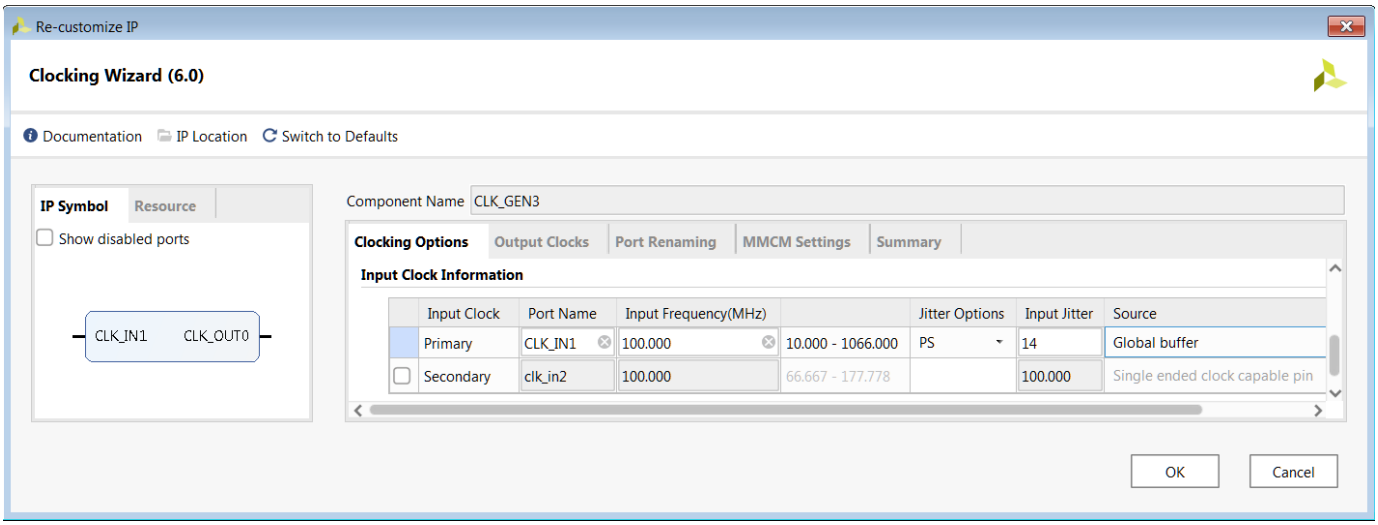

Fig 16.11 Configuring MMCM3 used for the source-synchronous 1-bit output interface with DDR.

Also, as done in section 16.1.1, we'll use MMCM3 to phase-shift the forwarded-clock so that edges lie approximately in the middle of the data-eye (ie. making things center-aligned). However, where is the "middle of the data-eye" for DDR? Well, it is not half way between rising-edges of the forwarded-clock as it was for an SDR interface. Why not? Because, the DDR interface sends and receives data on both the rising-edge and the falling-edge of the clock whereas the SDR interface sends and receives data only on the rising-edge of the clock. In short, for DDR we need a forwarded-clock that is shifted by approximately 90 degrees (instead of the 180 degrees used for SDR) to make things center-aligned. So, under the "Output Clocks" tab in Fig 16.11, we will initially specify that MMCM3 is to shift the clock by 90 degrees.

16.2.3 HDL snippets

For this example of source-synchronous FPGA DDR output, I have listed snippets from my VHDL in Fig 16.12. To keep netlist names simple, I placed this VHDL in the top-level component (see section 4.6) for my VHDL project. Hence, the ports for this component (lines 08-09 in Fig 16.12) are associated with FPGA pins. The MMCM3 module that we configured in section 16.2.2 is declared in lines 20-25 and instantiated in lines 30-34. As indicated by comments in line 19 and line 29, the VHDL needed to declare and instantiate MMCM3 can be found in a file called CLK_GEN3.vho, which was automatically generated when we configured and created MMCM3. The VHDL signal, SSI3_CLKB, is declared in line 16 and associated with the output of the MMCM3 in line 33. The VHDL signals, SSO3_DAT1 and SSO3_DAT2, are declared in line 17 and used to hold data that will be sent to the external device. As indicated by line 36, SSO3_DAT1 and SSO3_DAT2 should be filled on the rising-edge of CLK100. Also on the rising-edge of CLK100, data found in SSO3_DAT1 and SSO3_DAT2 are captured by ODDR_D in lines 50-60 and eventually sent to FPGA pin, SSO3_DAT. In lines 38-48, ODDR_C forwards SSI3_CLKB without inversion (since D1=1, D2=0) to FPGA pin, SSO3_CLK. You will recall from section 15.3.6 that the ODDR is already (and only) found in the FPGA IOB. So, unlike the HDL snippets in Fig 15.7 for the FPGA SDR output, VHDL attribute statements are not needed here to place the ODDR in the IOB.

```
01  library IEEE;
02  use IEEE.STD_LOGIC_1164.ALL;
03  use IEEE.NUMERIC_STD.ALL;
04
05  entity TOP is
06    port(
07             --Other inputs and outputs are listed here
08             SSO3_CLK : out std_logic;  --outgoing clock for FPGA output (DDR)
09             SSO3_DAT : out std_logic   --outgoing data for FPGA output (DDR)
10          );
11  end TOP;
12
13  architecture MY_TOP of TOP is
14
15    signal CLK100  : std_logic;         --output (100MHz) from clock module, MMCM1
16    signal SSI3_CLKB : std_logic;       --output (100MHz) from clock module, MMCM3
17    signal SSO3_DAT1, SSO3_DAT2 : std_logic; --holds data to be sent to port, SSO3_DAT
18
19    --Declaration of IP clock module component copied from CLK_GEN3.vho
20    component CLK_GEN3
21      port(
22           clk_in1  : in  std_logic;
23           CLK_OUT0 : out std_logic
24           );
25      end component;
26
27  begin
28
29    --Instantiation of IP clock module component copied from CLK_GEN3.vho
30    MMCM3: CLK_GEN3
31      port map(
32                clk_in1  => CLK100,
33                CLK_OUT0 => SSI3_CLKB
34               );
35
36    --Insert code here that places values in SSO3_DAT1 and SSO3_DAT2 on rising-edge of CLK100
37
38    ODDR_C: ODDR  --forwarded clock
39    generic map( DDR_CLK_EDGE => "OPPOSITE_EDGE", INIT =>'0', SRTYPE => "SYNC" )
40    port map(
41      Q => SSO3_CLK,       -- 1-bit DDR output
42      C => SSI3_CLKB,      -- 1-bit clock input
43      CE => '1',           -- 1-bit clock enable input
44      D1 => '1',           -- 1-bit data input (out-of-Q on rising-edge of C)
45      D2 => '0',           -- 1-bit data input (out-of-Q on falling-edge of C)
46      R => '0',            -- 1-bit reset input
47      S => '0'             -- 1-bit set input
48    );
49
50    ODDR_D: ODDR  --forwarded data
51    generic map( DDR_CLK_EDGE => "OPPOSITE_EDGE", INIT =>'0', SRTYPE => "SYNC" )
52    port map(
53      Q => SSO3_DAT,       -- 1-bit DDR output
54      C => CLK100,         -- 1-bit clock input
55      CE => '1',           -- 1-bit clock enable input
56      D1 => SSO3_DAT1,     -- 1-bit data input (out-of-Q on rising-edge of C)
57      D2 => SSO3_DAT2,     -- 1-bit data input (out-of-Q on falling-edge of C)
58      R => '0',            -- 1-bit reset input
59      S => '0'             -- 1-bit set input
60    );
61
62   end MY TOP;
```

Fig 16.12 Snippets of VHDL for the FPGA source-synchronous output (1-bit) DDR interface shown in Fig 16.8.

16.2.4 Constraints Template

Shown in Fig 16.13 is a template that will help us write constraints for the example source synchronous DDR output interface shown in Fig 16.8. In this template, I've entered values that apply to our example DDR interface and added my own comments. There are plenty of things to discuss. So, let's carefully go through them one by one.

As describe in section 16.2.1, DDR output has two pairs of registers that each exchange data in SDR fashion. That is, we can consider the DDR output interface to be a pair of SDR output interfaces. Thus, we should expect the Fig 16.13 template to be similar but with "twice the stuff" found in the Fig 15.10 template for source synchronous SDR output. The so-called *rising-edge path* from ODDR-R1 to X1_reg is used when data is transferred on the rising-edge of CLK100. The *falling-edge path* from ODDR-R2 to X2_reg is used when data is transferred on the falling-edge of CLK100. So, for each of these paths, we need to write a pair of **`set_output_delay`** constraints as shown in lines 26-36 of Fig 16.13.

Continuing with our "pair of SDR outputs" viewpoint, we find that the setup and hold parameters, (tsur, thdr), in lines 04-05 are for the register called X1_reg because it is the register that participates in data transfer over the rising(r)-edge path. Similarly, (tsuf, thdf), in lines 10-11 are for the register called X2_reg because it is the register that participates in data transfer over the falling(f)-edge path. I have assumed that X1_reg and X2_reg have the same requirements for setup and hold and have used 2.0ns and 1.0ns, respectively. These times are typical for the digital register found in the familiar 74LVC374 integrated circuit.

```
01  # Source Synchronous Output, Double Data Rate (DDR), times used below are nanoseconds(ns)
02  # -------
03  # For register-to-register path clocked on the rising-edge of the forwarded-clock:
04  set tsur      2.000;         #destination device setup time
05  set thdr      1.000;         #destination device hold time
06  set tdfr_max  0.000;         #max trace delay diff (data-delay minus clock-delay)
07  set tdfr_min -0.100;         #min trace delay diff (data-delay minus clock-delay)
08  # --
09  # For register-to-register path clocked on the falling-edge of the forwarded-clock:
10  set tsuf      2.000;         #destination device setup time
11  set thdf      1.000;         #destination device hold time
12  set tdff_max -0.100;         #max trace delay diff (data-delay minus clock-delay)
13  set tdff_min -0.200;         #min trace delay diff (data-delay minus clock-delay)
14  # --
15  # Specify ports and names
16  set dat_port {SSO3_DAT};     #name of FPGA port(s) used to forward the data
17  set clk_port  SSO3_CLK;      #name of FPGA port used to forward the clock
18  set fclk_nam   FCLK3;        #name of forwarded-clock assigned by create_generated_clock
19  set fclk_src [get_pins ODDR_C/C]; #source of the forwarded-clock
20  # --
21  # Describe the forwarded clock
22  create_generated_clock -name $fclk_nam  -source $fclk_src  -divide_by 1 \
23                         [get_ports $clk_port ];     #NOTE: "-invert" is not used
24  # --
25  # Output delay constraints
26  set_output_delay -clock $fclk_nam  -max [expr $tdfr_max + $tsur ] \
27                    [get_ports $dat_port ]
28  set_output_delay -clock $fclk_nam  -min [expr $tdfr_min - $thdr ] \
29                    [get_ports $dat_port ]
30  set_output_delay -clock $fclk_nam  -max [expr $tdff_max + $tsuf ] \
31                   -clock_fall -add_delay [get_ports $dat_port ]
32  set_output_delay -clock $fclk_nam  -min [expr $tdff_min - $thdf ] \
33                   -clock_fall -add_delay [get_ports $dat_port ]
34  # --
35  # Sometimes the following constraint is needed by MMCM to correctly interpret phase shifts
36  set_property PHASESHIFT_MODE LATENCY [get_cells MMCM3/inst/mmcm_adv_inst]
```

Fig 16.13 Timing constraints template (completed) for FPGA source-synchronous DDR input shown in Fig 16.14 and Fig 16.15.

Our "pair of SDR outputs" viewpoint continues as we talk about parameters called (tdfr_max, tdfr_min, tdff_max, tdff_min) in Fig 16.13. As in the Fig 15.10 template, these parameters represent delay differences, (data-delay minus clock-delay), caused by things outside the FPGA. In Fig 15.10, we attributed these delay-differences to length-differences for the board traces used to carry the forwarded-data and forwarded-clock signals. For the DDR interface, these delay differences are also caused by other things. For example, Fig 16.8 shows the delay difference is computed as (t_pxd1 – t_pxc1) for data transfer over the rising-edge path. Further, this delay difference is caused only by board trace-length differences. However, for data transfer over the falling-edge path, the delay difference is computed as (t_pxd2 – t_pxc2). Note from Fig 16.8 that t_pxc2 includes delay caused by a digital inverter found in the external device

that is used to invert the forwarded-clock. For example, suppose that the t_pxc2 is composed of a trace-delay, 0.1ns, and a digital inverter delay, 0.3ns. If t_pxd2 is composed only of trace-delay, 0.2ns, then (t_pxd2 – t_pxc2) is computed to be (0.2 – (0.1 + 0.3)) = -0.2ns.

To complete our "pair of SDR outputs" viewpoint for DDR, I need to comment on the **`set_output_delay`** arguments called **`-clock_fall`** and **`-add_delay`** found in line 31 and line 33 of Fig 16.13. The argument called **`-clock_fall`** simply means that this constraint should be used for timing analysis of the falling-edge path. The **`set_output_delay`** constraints that do not have the **`-clock_fall`** argument will be used for timing analysis of the rising-edge path. The argument called **`-add_delay`** is a bit confusing since there is no "adding a delay" as implied by the wording. The **`-add_delay`** arguments tells timing analysis that there is no interaction between the **`set_output_delay`** used for the rising-edge path and the **`set_output_delay`** used for the falling-edge path. If you were to accidentally omit the **`-add_delay`** arguments in lines 30-33 then the calculated values for **`-max`** and **`-min`** from these lines would overwrite the calculated values for **`-max`** and **`-min`** from lines 26-29.

Finally, as with the SDR interface, only one **`create_generated_clock`** constraint is needed for the DDR interface and it is shown in lines 22-23 of Fig 16.13. This constraint is almost identical to the **`create_generated_clock`** constraint found in lines 13-14 of Fig 15.10. For the SDR output interface, we used the ODDR to invert (ie. phase shift the forwarded clock by 180 degrees). Because of this, the **`-invert`** argument was used in line 14 of the Fig 15.10 template. However, for our DDR output interface, we are using MMCM3 to phase shift the forwarded clock. Thus, the **`-invert`** argument is not needed for the DDR **`create_generated_clock`** constraint.

16.2.5 Timing Path Report

The Vivado IDE created the schematics shown in Fig 16.14 and Fig 16.15 for our DDR interface circuit after implementation of the VHDL shown in Fig 16.12. You'll find the combination of these schematics to be conceptually identical to Fig 16.8.

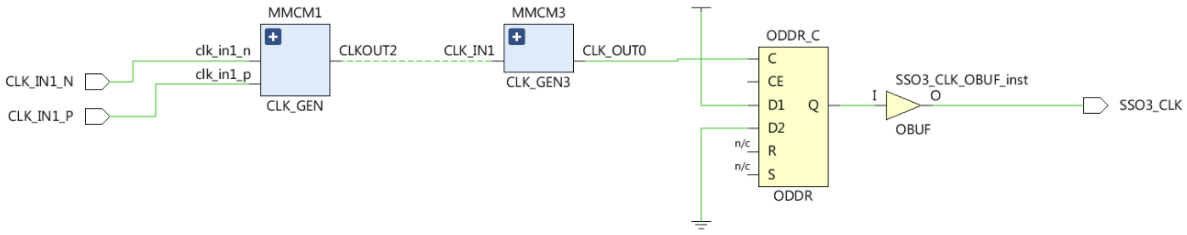

Fig 16.14 Schematic from Vivado IDE for forwarded-clock of the source-synchronous DDR output (1-bit) interface.

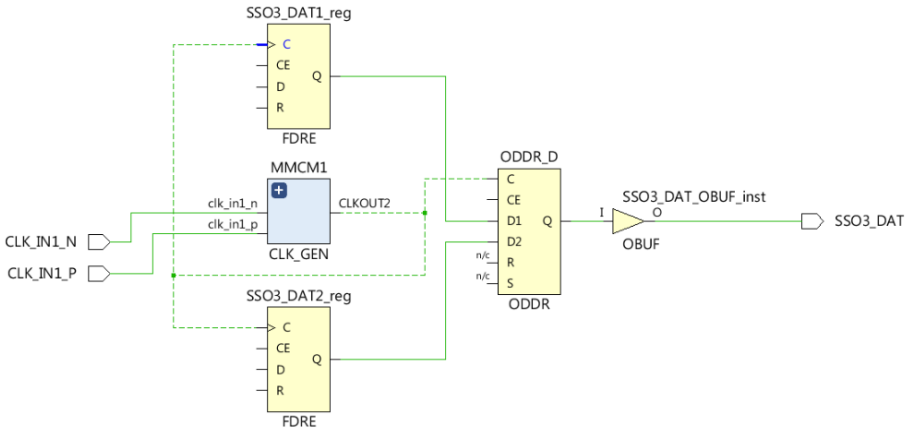

Fig 16.15 Schematic from Vivado IDE for forwarded-data of the source-synchronous DDR output (1-bit) interface.

Our example DDR interface did not initially pass timing analysis. However, when I again configured MMCM3 as described in section 16.2.2 but used a clock shift of -74 degrees instead of +90 degrees, then the interface passed timing analysis with setup-slack of +0.175 and hold-slack of +0.137. These small positive values of slack mean that the interface is just barely passing timing analysis and that the data-eye is small. Does this make sense? Well, the maximum width of the data-eye for a DDR interface is one-half the interface clock period. Since our example DDR interface uses a 100 MHz clock, then the maximum width of the data-eye is 5.0 ns. From this maximum width we must subtract the setup time, 2.0 ns, and hold time, 1.0 ns, requirements for the external device. This leaves us with a maximum width for the data-eye of about 2.0 ns. The four-corners stuff that we talked about in section 14.4 and the fact that the two SDR interfaces associated with our DDR interface have slightly different characteristics has further narrowed the data-eye to a width of only (0.175 + 0.137 = 0.312 ns).

Also, does the -74 degrees phase shift for the interface clock make sense? Well, the clock-to-output time for the ODDR is only about 0.2 ns. So, the FPGA is sending data over the interface that is almost edge-aligned with CLK100 (identified by CLKOUT2 in Fig 16.5). So, if the FPGA shifts CLK100 backwards in time by half the data-eye width, 2.5ns, and sends it out as the interface clock, SSO3_CLK, then the FPGA would be sending center-aligned data over the interface. This is exactly what has been done, because MMCM3 is shifting CLK100 by -74 degrees (= 74degs * 10ns/360degs = 2.06 ns).

Finally, Fig 16.16 shows the setup-timing report for the *rising-edge* path in our example DDR interface which I generated using the following Tcl command.

```
report_timing -from [get_pins ODDR_D/C] -rise_to [get_ports SSO3_DAT] -setup          (16.7)
```

Again, let's do the simple calculation for setup-slack that we learned in section 14.2. For this calculation, we'll use the time-arc symbols from Fig 16.8 and we'll extract numbers we need from the *rising-edge* path report shown in Fig 16.16.

- data-arrival-time = (t_mle + t_mccd + t_cod1 + t_bfd + t_pxd1) = 3.664 ns
  - t_mle = 0.000 = creation time for clock launch-edge
  - t_mcdd = 1.677 (lines 43-45) = delay from MMCM1 to ODDR_D/C
  - t_cod1 = 0.221 (line 48) = clock-to-output(Q) time for register, R1, of ODDR_D (see also Fig 16.9)
  - t_bfd = 1.666 (lines 49-51) = delay from ODDR_D/Q thru OBUF to port, SSO3_DAT
  - t_pxd1 = 0.100 = delay of external trace that carries the forwarded data to X1_reg/D
- data-required-time = (t_mce + t_mccc + t_coc + t_bfc + t_pxc1) = 5.985 ns
  - t_mce = 5.000 = creation time for clock capture-edge that occurs *after* the clock launch-edge
  - t_mccc = -0.661 (lines 61-68) = delay from MMCM1 thru MMCM3 to ODDR_C/C
  - t_coc = 0.192 (line 69) = clock-to-output(Q) time for ODDR_C
  - t_bfc = 1.354 (lines 70-72) = delay from ODDR_C/Q thru OBUF to port, SSO3_CLK
  - t_pxc1 = 0.100 = delay of external trace that carries the forwarded-clock to X1_reg/C
- setup-slack:
  - = (data-required-time) minus (data-arrival-time) minus (external register setup time)
  - = 5.985 – 3.664 – 2.000 = +0.321 ns

Our simple calculation for setup-slack of +0.321 agrees nicely with the value of +0.175 shown in line-12 of Fig 16.16.

For our example DDR interface, there are two setup timing reports; one for rising-edge path and one for the falling-edge path through the interface. We could generate the setup timing report for the falling-edge path using the following Tcl command.

```
report_timing -from [get_pins ODDR_D/C] -fall_to [get_ports SSO3_DAT] -setup          (16.8)
```

Finally, for our example DDR interface, we could also generate the two hold timing reports using the (16.7) and (16.8) Tcl commands with **`-setup`** replaced by **`-hold`**.

```
------------------------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.3 (win64) Build 2405991 Thu Dec  6 23:38:27 MST 2018
| Date         : Sun Jun 23 15:19:56 2019
| Host         : Rosetta running 64-bit Service Pack 1  (build 7601)
| Command      : report_timing -from [get_pins ODDR_D/C] -rise_to [get_ports SSO3_DAT] -setup
| Design       : TOP
| Device       : 7k160t-fbg484
| Speed File   : -3  PRODUCTION 1.12 2017-02-17
------------------------------------------------------------------------------------------------
Timing Report

Slack (MET) :             0.175ns  (required time - arrival time)
  Source:                 ODDR_D/C
                            (falling edge-triggered cell ODDR clocked by CLKOUT2_CLK_GEN {rise@0.000ns
                                        fall@5.000ns period=10.000ns})
  Destination:            SSO3_DAT
                            (output port clocked by FCLK3  {rise@0.000ns fall@5.000ns period=10.000ns})
  Path Group:             FCLK3
  Path Type:              Max at Fast Process Corner
  Requirement:            5.000ns  (FCLK3 rise@10.000ns - CLKOUT2_CLK_GEN fall@5.000ns)
  Data Path Delay:        1.887ns  (logic 1.887ns (100.000%)  route 0.000ns (0.000%))
  Logic Levels:           1  (OBUF=1)
  Output Delay:           2.000ns
  Clock Path Skew:        -0.717ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    -1.024ns = ( 8.976 - 10.000 )
    Source Clock Delay      (SCD):    -0.530ns = ( 4.470 - 5.000 )
    Clock Pessimism Removal (CPR):    -0.223ns
  Clock Uncertainty:      0.221ns  ((TSJ^2 + DJ^2)^1/2) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Discrete Jitter          (DJ):    0.138ns
    Phase Error              (PE):    0.144ns

    Location             Delay type                Incr(ns)  Path(ns)    Netlist Resource(s)
  -------------------------------------------------------------------    -------------------
                         (clock CLKOUT2_CLK_GEN fall edge)
                                                     5.000     5.000 f
    W9                                               0.000     5.000 f  CLK_IN1_P (IN)
                         net (fo=0)                  0.000     5.000    MMCM1/inst/clk_in1_p
    W9                   IBUFDS (Prop_ibufds_I_O)    0.449     5.449 f  MMCM1/inst/clkin1_ibufgds/O
                         net (fo=1, routed)          0.553     6.002    MMCM1/inst/clk_in1_CLK_GEN
    MMCME2_ADV_X1Y1      MMCME2_ADV (Prop_mmcme2_adv_CLKIN1_CLKOUT2)
                                                    -3.209     2.793 f  MMCM1/inst/mmcm_adv_inst/CLKOUT2
                         net (fo=1, routed)          0.807     3.600    MMCM1/inst/CLKOUT2_CLK_GEN
    BUFGCTRL_X0Y0        BUFG (Prop_bufg_I_O)        0.030     3.630 f  MMCM1/inst/clkout3_buf/O
                         net (fo=7, routed)          0.840     4.470    CLK100
    OLOGIC_X0Y110        ODDR                                      f  ODDR_D/C
  -------------------------------------------------------------------    -------------------
    OLOGIC_X0Y110        ODDR (Prop_oddr_C_Q)        0.221     4.691 r  ODDR_D/Q
                         net (fo=1, routed)          0.000     4.691    SSO3_DAT_OBUF
    P19                  OBUF (Prop_obuf_I_O)        1.666     6.357 r  SSO3_DAT_OBUF_inst/O
                         net (fo=0)                  0.000     6.357    SSO3_DAT
    P19                                                        r  SSO3_DAT (OUT)
  -------------------------------------------------------------------    -------------------
                         (clock FCLK3 rise edge)    10.000    10.000 r
    W9                                               0.000    10.000 r  CLK_IN1_P (IN)
                         net (fo=0)                  0.000    10.000    MMCM1/inst/clk_in1_p
    W9                   IBUFDS (Prop_ibufds_I_O)    0.368    10.368 r  MMCM1/inst/clkin1_ibufgds/O
                         net (fo=1, routed)          0.503    10.871    MMCM1/inst/clk_in1_CLK_GEN
    MMCME2_ADV_X1Y1      MMCME2_ADV (Prop_mmcme2_adv_CLKIN1_CLKOUT2)
                                                    -2.780     8.091 r  MMCM1/inst/mmcm_adv_inst/CLKOUT2
                         net (fo=1, routed)          0.744     8.835    MMCM1/inst/CLKOUT2_CLK_GEN
    BUFGCTRL_X0Y0        BUFG (Prop_bufg_I_O)        0.026     8.861 r  MMCM1/inst/clkout3_buf/O
                         net (fo=7, routed)          0.697     9.558    MMCM3/inst/CLK_IN1
    MMCME2_ADV_X1Y0      MMCME2_ADV (Prop_mmcme2_adv_CLKIN1_CLKOUT0)
                                                    -3.709     5.849 r  MMCM3/inst/mmcm_adv_inst/CLKOUT0
                         net (fo=1, routed)          0.931     6.780    MMCM3/inst/CLK_OUT0_CLK_GEN3
    BUFGCTRL_X0Y1        BUFG (Prop_bufg_I_O)        0.026     6.806 r  MMCM3/inst/clkout1_buf/O
                         net (fo=1, routed)          0.624     7.430    SSI3_CLKB
    OLOGIC_X0Y124        ODDR (Prop_oddr_C_Q)        0.192     7.622 r  ODDR_C/Q
                         net (fo=1, routed)          0.000     7.622    SSO3_CLK_OBUF
    N18                  OBUF (Prop_obuf_I_O)        1.354     8.976 r  SSO3_CLK_OBUF_inst/O
                         net (fo=0)                  0.000     8.976    SSO3_CLK
    N18                                                        r  SSO3_CLK (OUT)
                         clock pessimism            -0.223     8.753
                         clock uncertainty          -0.221     8.532
                         output delay               -2.000     6.532
  -------------------------------------------------------------------
                         required time                         6.532
                         arrival time                         -6.357
  -------------------------------------------------------------------
                         slack                                 0.175
```

Fig 16.16 The Vivado *setup* timing report for the *rising-edge* path between ODDR_D and the external device shown in Fig 16.8.

16.3 Multi-Bit Parallel

In section 15.4 and throughout this chapter we've focused on source-synchronous 1-bit FPGA I/O. However, source-synchronous I/O can sometimes be multi-bit parallel. As demonstrated by the example in this section, it is usually straightforward to extend our 1-bit discussions and results to multi-bit parallel.

A source-synchronous multi-bit parallel interface consists of a single clock line and multiple data lines in parallel. For example, a 2-bit parallel SDR output interface is shown in Fig 16.17, which should be compared with the 1-bit SDR output interface shown in Fig 15.5. This comparison shows that FPGA hardware used for the single data line of 1-bit I/O is simply copied to implement the additional data lines of multi-bit parallel I/O.

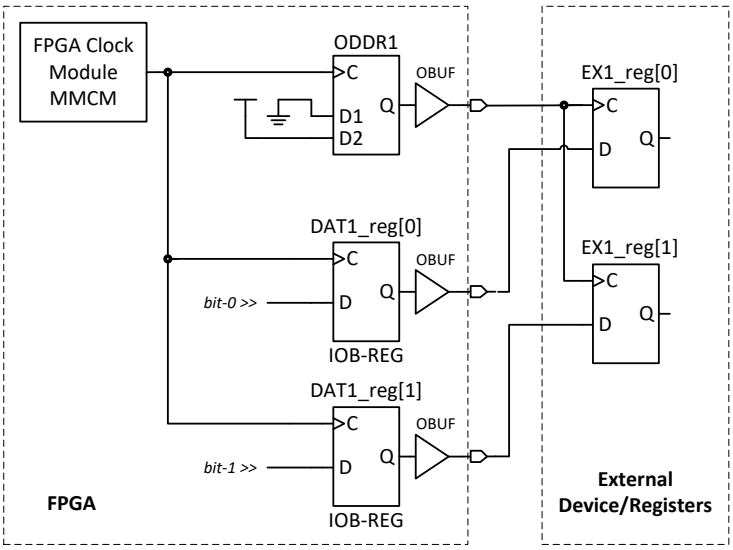

Fig 16.17 Example circuit for 2-bit parallel source-synchronous FPGA output.

VHDL code snippets for implementing the 1-bit SDR output interface are shown in Fig 15.7. As shown in Fig 16.18, changes to only two lines of code in Fig 15.7 are needed to implement the 2-bit parallel SDR output interface.

```
11          SSO1_DAT  : out std_logic_vector(1 downto 0)  --outgoing data for FPGA output
18  signal DAT1 : std_logic_vector(1 downto 0);   --DAT1 holds 2-bit data for output
```

Fig 16.18 Snippets of VHDL from Fig 15.7 that need to be changed for 2-bit FPGA source-synchronous parallel output.

Timing constraints needed for the 1-bit SDR output interface are shown in Fig 15.10. As shown in Fig 16.19, a change to line 07 in Fig 15.7 is all that's needed to implement the 2-bit parallel SDR output interface. In Tcl, the asterisk is understood to be a wildcard. That is, use of the asterisk in line 07 of Fig 16.19 causes this line to refer to both ports, (SSO1_DAT[0] and SSO1_DAT[1]), of the 2-bit interface. Finally, the Fig 16.19 changes to Fig 15.10 assume that the board trace delay is the same for each of the data lines in the 2-bit interface.

```
07  set out_ports1 {SSO1_DAT[*]};    #name of FPGA port(s) used to forward the data
```

Fig 16.19 Timing constraints template lines from Fig 15.10 that need to be changed for 2-bit FPGA source-synchronous parallel output.

16.4 SERDES

Found in most FPGAs is special hardware used for SERDES (SERialization and DESerialization) of digital data. When this special hardware is used for serialization (ie. as a parallel-to-serial converter) it is called the Output-SERDES (OSERDES). When this special hardware is used for deserialization (ie. as a serial-to-parallel converter) it is called the Input-SERDES (ISERDES). Block diagrams for typical ISERDES and OSERDES are shown in Fig 16.20, where we see that both components have data-inputs, D, data outputs, Q, clock inputs labelled CLK and CLKDIV, and an internal shift-register with storage locations labelled S1 through S8.

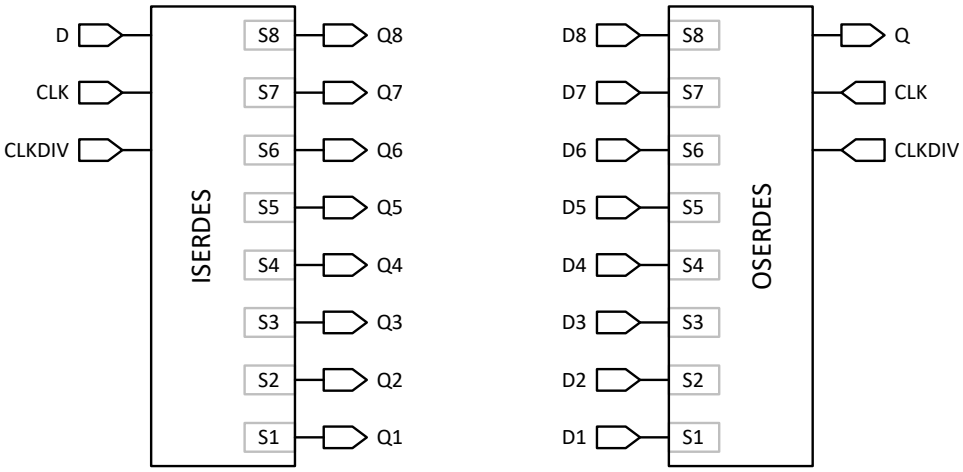

Fig 16.20 Simplified block diagrams for ISERDES and OSERDES components.

For ISERDES, data coming into pin-D is captured on the rising-edge of CLK and transferred to shift-register storage-S1. On the next rising-edge of CLK, data coming into pin-D is again captured and transferred to storage-S1 - and, data that was previously in storage-S1 is shifted to become the data in storage-S2. That is, on each rising-edge of CLK, data is transferred from pin-D to storage-S1 and data in all eight storage locations is shifted upwards by one position (data that was in storage-S8 is no longer available). For a Single-Data-Rate (SDR) 1:8 ISERDES, the input clock, CLKDIV, is required to have a frequency that is $1/8^{th}$ the frequency of CLK. On every rising-edge of CLKDIV, data from all eight storage locations of the shift-register are sent to the ISERDES outputs, Q1-Q8. So, on every rising-edge of CLKDIV, a new set of data is seen on Q1-Q8.

In section 16.1, our example of a source-synchronous input interface used an IOB-register to capture data coming from the interface. Both the IOB-register and registers that received data from the IOB-register were clocked by the forwarded clock, SSI2_CLK, of the interface. For that interface, the ISERDES could have been used instead of the IOB-register to capture the data. When using ISERDES for the section 16.1 interface, SSI2_CLK connects to pin-CLK of the ISERDES, the forwarded data, SSI2_DAT, connects to pin-D of ISERDES, and the FPGA divides SSI2_CLK by eight and send the result to pin-CLKDIV of ISERDES. The advantage of using ISERDES (instead of the IOB-register) is that FPGA registers receiving data from ISERDES can be clocked by the slow clock, CLKDIV, instead of by the faster clock, CLK = SSI2_CLK. As explained in section 12.1, using slower clocks instead of faster clocks inside the FPGA can help a design pass timing analysis. That is, the FPGA can usually handle faster interfaces when using ISERDES than when using the IOB-register.

The OSERDES component works in reverse of ISERDES. That is, OSERDES is used for FPGA *output* whereas ISERDES is used for FPGA *input*. For the synchronous *output* interface described in section 15.4, the OSERDES could replace the IOB-register, DAT1_reg. The advantage of using OSERDES is that the slow clock, CLKDIV, (instead of the faster clock, CLK = SSO1_CLK) can be used inside the FPGA to prepare the data being sent to the interface.

Finally, a nice feature of ISERDES and OSERDES is that the clock-crossing (see section 12.2) between the CLK-domain and the CLKDIV-domain is usually handled automatically by circuits found inside the ISERDES and OSERDES.

17. Timing Exceptions

As discussed at the beginning of chapter 11, implementation and timing analysis work together to achieve timing closure. That is, implementation attempts to layout circuits inside the FPGA in a way that makes all the circuits pass timing analysis. However, implementation is sometimes unable to achieve timing closure without a little help from us. We can sometimes provide this help by writing *timing exceptions*.

We first discussed timing exceptions in section 11.1 and since then I have mentioned them often. However, not until our chapter 14 discussion of timing analysis and timing paths did will have the tools needed to understand and write timing exceptions. As with timing constraints, we use the Tcl language to write timing exceptions and we place them in our project's constraint file that we've been calling **constraints1.xdc**. You'll recall from section 4.6 and section 11.5 that timing constraints are used to describe facts (eg. clock period, matchup between HDL names and FPGA pins) to the IDE. However, timing exceptions tell timing analysis not to analyze certain circuits or to analyze them in a special way.

Said another way, timing exceptions help ensure that the combined efforts of implementation and timing analysis are focused on circuits that need full timing analysis - and are not focused on circuits that need simplified timing analysis or no timing analysis at all.

17.1 set_false_path

Let's start with the easy stuff and discuss circuits that need no timing analysis. For this discussion, we will consider the following common sequence of events for an FPGA.

1. Power is applied to the FPGA.
2. An external computer sends setup-parameters to the FPGA via a serial port and puts the FPGA in IDLE-mode.
3. The FPGA stores the setup-parameters as signals.
4. Much time (ie. many cycles of the FPGA clocks) passes as the system warms up.
5. An external computer tells the FPGA to exit IDLE-mode and enter RUN-mode.
6. The FPGA uses the setup-parameters to accomplish tasks that it needs to do while in RUN-mode.

In step-3 above, let's consider a specific setup-parameter that is stored in the following VHDL signal.

```
signal SPAR1 : std_logic_vector(7 downto 0);
```
(17.1)

As stated in step-4 above, SPAR1 is assigned a value that remains unchanged for many clock cycles before it is used. Hence, as seen by HDL that is active during the FPGA RUN-mode, SPAR1 is effectively a constant. Since timing analysis is needed only for signals that change with time, there is no need to run timing analysis on any paths associated with reading the value of SPAR1.

Here's another way to look at things. Recall from section 7.5 that signals in our RTL-style VHDL code correspond to digital registers. So, implementation will represent the 8-bits of SPAR1 using eight digital registers called (SPAR1_reg[0], SPAR1_reg[1]….SPAR1_reg[7]). After these registers are loaded with effectively constant values, then any timing path having SPAR1_reg[*] as the source-register need not undergo timing analysis. The way we tell timing analysis not to analyze these specific paths is to place the following **set_false_path** timing exception in the constraints file.

```
set_false_path -from [get_cells {SPAR1_reg[*]}]
```
(17.2)

In the above constraint, Tcl considers the asterisk to be a wild-card character. Use of the asterisk allows us to write one constraint that applies to all eight bits of SPAR1. Also in the above constraint, the curly-braces, {.}, are necessary escape characters. That is, square-braces, [.], have special meaning in Tcl but square-braces are also part of the VHDL-name for

the std_logic_vector signal called SPAR1. So, we must surround the text, SPAR1_reg[*], with curly-braces to prevent the square-braces in this text from being misinterpreted by Tcl.

17.2 set_max_delay -datapath_only

In chapter 12 we learned that exchanging signals/data between clock domains (aka clock-crossings) can produce paths that fail timing analysis and cause digital-register metastability. Before continuing our discussion of timing exceptions, let's review these chapter 12 concepts by considering the example clock-crossing shown in Fig 17.1. In this clock-crossing, 1-bit of data is being transferred from SIG1_reg in the CLK1-domain to SIG2_reg in the CLK2-domain. Fig 17.1 also shows that a 2FF-synchronizer has been placed between SIG1_reg and SIG2_reg to help remove metastability.

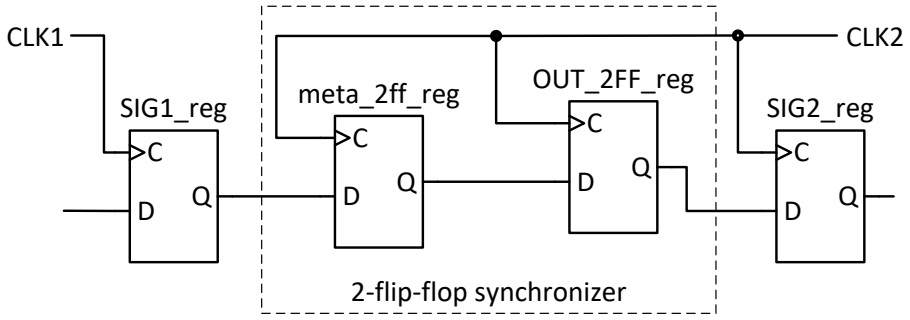

Fig 17.1 Use of 2-flip-flop (2FF) synchronizer at a 1-bit clock-crossing.

Despite use of a synchronizer in Fig 17.1, timing analysis will sometimes report that the timing-path from SIG1_reg\C to meta_2ff_reg\D fails timing analysis. Since we have properly designed and placed our synchronizer, this failed-path warning can be ignored. However, it's better to write a timing exception that prevents timing analysis from doing unnecessary analysis of the path. Thus, we can both eliminate the warning and free-up timing analysis to do more important things.

So, what timing exception should we write for the timing-path from SIG1_reg\C to meta_2ff_reg\D in Fig 17.1? Using **`set_false_path`** tells timing analysis to completely ignore this *timing-path* and allows implementation to choose any length for the *data-path* from SIG1_reg\Q to meta_2ff_reg\D. Here, the word, length, means the time-length that it takes for a signal to propagate through the path. Having no length-restriction for this *data-path* is usually not a problem for a 1-bit clock-crossing but is almost always a problem for a multi-bit clock-crossing. That is, suppose SIG1 were 4-bit instead of 1-bit. Then, we'd use a handshake synchronizer (see section 12.5) containing four circuits like the one shown in Fig 17.1 to cross each bit of SIG1 from the CLK1-domain into the CLK2-domain. Each of these four circuits would have a data-path like the one from SIG1_reg\Q to meta_2ff_reg\D. The question is, "What if implementation used wildly different lengths for each of these four data-paths?". Well, things would be a mess unless we could design our handshake synchronizer to somehow compensate for these path-length differences. Making matters worse is the fact that without restrictions for the data-paths, each run of implementation could result in a different length for the data-paths! So, since **`set_false_path`** removes all restrictions on length of the data-paths, it is usually the wrong timing exception to use for multi-bit clock-crossings.

Placing no timing exception on the timing-path from SIG1_reg\C to meta_2ff_reg\D is usually acceptable, because timing analysis and implementation will continue to work together, keeping the data-path length short in an attempt to pass timing analysis. However, as mentioned above, these almost-pointless attempts to make the path pass full timing analysis are sometimes done at the expense of other paths that really do need to pass full timing analysis. Almost always, it is best to write a timing exception called "**`set_max_delay -datapath_only`**" on the timing-path from SIG1_reg\C to meta_2ff_reg\D . For the Fig 17.1 example, this timing exception is written as shown below and is placed in the constraints file.

```
set_max_delay -datapath_only -from SIG1_reg\C -to meta_2ff_reg\D 10.0
```
(17.3)

In (17.3), the number, 10.0, restricts the data-path delay time to 10ns for the timing-path from SIG1_reg\C to meta_2ff_reg\D. Instead of 10.0, you should use a value equal to smaller period of the two clocks, CLK1 and CLK2. Finally, when SIG1 is multibit, you can use the following one-line exception to restrict delay time for all the bits.

```
set_max_delay -datapath_only -from [get_cells {SIG1_reg[*]}] -to [get_cells {meta_2ff_reg[*]}] 10.0
```
(17.4)

After writing the example timing exception (17.3), the timing-path from SIG1_reg\C to meta_2ff_reg\D will be analyzed by timing analysis in a special way. First, timing analysis calculates delay of the data-path. For our example, this data-path delay is the sum of the following timing-arcs:

- SIG1_reg\C to SIG1_reg\Q (clock-to-output time for SIG1_reg)
- SIG1_reg\D to meta_2ff_reg\D (propagation time through interconnecting wire)
- Setup time for meta_2ff_reg

Next, timing analysis subtracts this data-path-only delay from the max-delay (10.0ns in our example) specified in the timing exception. The result of this calculation is reported as the setup-slack for the timing-path. Since this special data-path-only calculation for setup-slack does not consider clock-path delays, we sometimes say that it ignores clock-skew. Timing analysis will neither calculate nor report a value of hold-slack for a timing-path having the "**set_max_delay -datapath_only**" timing exception.

In conclusion, we can use the "**set_max_delay -datapath_only**" timing exception on a timing-path to: a)prevent full timing analysis of the path, and to b)restrict length of the data-path associated with the timing-path. If, after writing this timing exception, timing analysis issues no warnings for the timing-path (ie. the timing-path has positive slack) then we know that implementation has successfully restricted the data-path length to the value specified in the timing exception. Finally, the **set_max_delay** timing exception is recognized by the IDE of other FPGA vendors. However, the **datapath_only** argument seems to be unique to Xilinx.

17.3 set_clock_groups

In short, the **set_clock_groups** timing exception is a quick way to effectively put a **set_false_path** exception on *every* path between two clock domains. My opinion is that **set_clock_groups** should almost never be used. I justify this opinion as follows. First, using **set_clock_groups** is too much of a blanket statement for my liking. Not only does this exception say that we can effectively use **set_false_path** for all existing clock-crossings, it also says this is true for all clock-crossings that we add in the future. That is, after writing a **set_clock_groups** exception between two clock domains, if you accidentally (or purposely) create a clock-crossing in your HDL then timing analysis will simply ignore it and issue no warning that may prompt you to look more carefully at the crossing. I firmly believe that you should be watchful for clock-crossings as you write HDL and address each crossing with both proper HDL and with a timing exception that is specific to the crossing. My second reason for almost never using **set_clock_groups** comes from section 17.2. That is, if you have clock-crossings then we almost never use **set_false_path** (which is effectively the same as **set_clock_groups**) and almost always use "**set_max_delay -datapath_only**". Finally, as will be explained in section 17.5, the **set_clock_groups** exception has the highest priority of all timing exceptions. Hence, for two clock domains, it is pointless to use **set_clock_groups** along with other timing exceptions because the other exceptions will be ignored.

In conclusion, I recommend never using the **set_clock_groups** timing exception.

17.4 set_multicycle_path

For completeness, I will only mention **set_multicycle_path** as one of the lesser-used timing exceptions. In short, this exception relaxes timing analysis for register-to-register data transfers. In particular, it is used when data is latched at the receiving-register after multiple cycles of the clock have passed since the data was launched from the sending-register. If you think you may need this exception, then search for more information on the Xilinx website.

17.5 Timing Exception Priority

Timing exceptions have something called priority. Priority means that if you write two timing exceptions for the same path then only the exception with the highest priority will be used (and the other one will be ignored). The timing exceptions discussed in this chapter are listed in priority order (from highest to lowest) below.

- **`set_clock_groups`**
- **`set_false_path`**
- **`set_max_delay -datapath_only`**
- **`set_multicycle_path`**

Please read cautions about using **`set_clock_groups`** that are found in section 17.3.

17.6 Constraints File Organization

Throughout this book, we've been placing things into the IDE project's master constraint file that we've been calling **constraints1.xdc**. Because things in the constraints file are read and applied sequentially by the IDE, the order of these things matters. Xilinx recommends the follow order for things in the constraints file.

1. Timing constraints (see chapters 15 and 16)
2. Timing exceptions (see chapter 17)
3. Physical constraints (see section 4.6)

As explained in section 11.5, when we setup a clock-module (and other IP) then separate small constraints files are automatically created. Usually, we need not worry about the contents of these automatically generated constraints files nor about how they are merged with the master constraints file.

Don't get too stressed about the proper order of things in the master constraints file. The IDE will usually warn you about ordering mistakes. For example, in the Fig 16.13 constraints template, if I move the **`create_generated_clock`** constraint to a position after line 33 then both synthesis and implementation will issue a warning for the **`set_output_delay`** constraints saying that "clock FCLK3 not found". This warning tells us that the **`create_generated_clock`** constraint must create/define the clock called FCLK3 before FCLK3 can be referenced in **`set_output_delay`**. Hence, in the constraints file, the **`create_generated_clock`** constraint must appear on a line that is located before the **`set_output_delay`** constraint.

18. Generate Bitstream

Since chapter 11, we've been on the implementation step of the IDE flow (see chapter 3) whereby digital circuits generated from our HDL are mapped into the FPGA hardware. As explained in previous chapters, implementation does this mapping while trying to ensure that every part of the digital circuit passes timing analysis. Now, we are ready to end our journey through the IDE flow with a last step called "Generate Bitstream". In this last step, we create a file that is used to communicate the implementation results to the FPGA. This special instruction file goes by several names including bitstream, bit-file, and FPGA-configuration-file. I will call it a bit-file.

After successfully running implementation, we can usually generate the bit-file with a simple mouse-click. In the Vivado IDE (see Fig 3.1), this mouse-click is done on the words "Generate Bitstream" found in the lower-left corner of the main screen. The resulting file will, by default, have the extension, .bit, and a name that is the same as the name for the top-level component (see section 4.6) in your Vivado-VHDL project. So, if your top-level component is called TOP, then the default name for the bit-file will be "TOP.bit". This bit-file will be saved into the disk directory for your Vivado project.

18.1 Programmer Hardware and JTAG

The Vivado IDE can be used to send the bit-file to the FPGA. However, you must first make a physical connection between your computer and the FPGA. This physical connection uses a device that is generally called a programmer module and specifically called a "Platform Cable USB" by Xilinx. The Xilinx programmer module has two connectors. One connector attaches to the USB port on the computer that is running Vivado and the other attaches to the JTAG pins on the FPGA. The acronym, JTAG, stands for Joint Test Action Group, which is an organization that created standards for communicating with integrated circuits such as an FPGA. Hopefully, the person who designed your FPGA board put a JTAG connector on the board and connected it properly to the JTAG pins on the FPGA.

So, the steps needed to send the bit-file to the FPGA are:

1) Find your programmer module. Connect one of its cables to the USB port on your computer. Connect the programmer module's other cable to the JTAG port on your FPGA board.
2) Apply the clock input(s) to the FPGA board.
3) Apply electrical power to the FPGA board.
4) Use the Vivado Hardware Manager to send the bit-file from your computer to the FPGA via the programmer module. You can read Xilinx's documentation about using the Vivado Hardware Manager.

18.2 RAM vs PROM

The procedure described in section 18.1 for sending the bit-file to the FPGA is typically fast and easy. That procedure loads the bit file to volatile memory in the FPGA, which I will call RAM (Random Access Memory) for lack of a better word. Hence, when power is cycled to the FPGA, the bit-file stored in RAM will be erased. So, after power-up, you must again send the bit-file to the FPGA to get things going again.

Using the programmer module to manually resend the bit-file every time power is applied to the FPGA is fine for initial testing. However, when the FPGA board is placed into operation, one typically uses the FPGA's ability to automatically load the bit-file at power-up from a Programmable Read-Only Memory (PROM) device that is attached to the FPGA. So, a second way to send the bit-file to the FPGA is to use Vivado and the programmer module to first send the bit-file to PROM. For this method to work properly, the person who designed your FPGA board must have properly connected the PROM to the FPGA and to the board's JTAG connector. Also, the original bit-file must be reformatted before it can be sent to PROM. The Xilinx Vivado IDE helps you write a Tcl command called WRITE_CFGMEM that converts the bit-file into a file with the needed format. The PROM file created in this way will have extension, .mcs, and it therefore sometimes called an mcs-file. Finally, the Vivado IDE can be used with the Xilinx programmer module to send the mcs-file to a PROM device.

THANK YOU !!!

References:

C. E. Cummings, *Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog*, http://www.sunburst-design.com/papers/CummingsSNUG2008Boston_CDC.pdf, SNUG-2008, Boston MA, 2008.

C. E. Cummings and D. Mills, *Synchronous Resets? Asynchronous Resets? I am so confused! How will I ever know which to use?*, http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_Resets.pdf, SNUG-2002, San Jose CA, 2002.

R. Ginosar, *Metastability and Synchronizers: A Tutorial*, IEEE Design & Test of Computers, http://webee.technion.ac.il/~ran/publications.html , pp. 23-35, 2011.

R. Ginosar, *Fourteen Ways to Fool Your Synchronizer*, Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC'03), http://webee.technion.ac.il/~ran/publications.html, pp. 1-8, 2003.

B. Mealy and F. Tappero, *Free Range VHDL*, www.freerangefactory.org , 2013.

R. Scoville, *TimeQuest User Guide*, http://www.alterawiki.com/wiki/TimeQuest_User_Guide , 2010.

R. Scoville, *Source-Synchronous Timing with TimeQuest*, http://www.alterawiki.com/wiki/Source_Synchronous_Analysis_with_TimeQuest, 2011.

Appendix A. Power-ON Reset

Back in chapter 6, we talked about the importance of initializing VHDL signals for simulation. In section 6.3, we learned that VHDL allows us to initialize signals when they are declared (eg. see Fig 6.3). However, in section 6.4, I cautioned you about relying on VHDL initialization during FPGA configuration. Here, we'll talk more about initializing signals during FPGA configuration, which is sometimes called the power-ON reset for the FPGA (or simply the FPGA reset).

From the software/HDL viewpoint, the FPGA reset can be described as a response to a global-reset signal that we'll call RSTG. That is, when RSTG=1 (ie. when the global-reset is ON) then key signals are initialized and the VHDL processes that modify these key signals are halted. For example, Fig A.1 shows a slightly modified version of the VHDL component called TWO_BIT.vhd that we talked about in section 4.1 (and in Fig 4.1). As indicated by line-05 in Fig A.1, I have revised TWO_BIT to have the reset input called RST shown in line-14. Further, as shown in lines 32-33, when RST=1 then process, MY_PRC1, does nothing except initialize/set bCTRL1 to the value of 1. So, if other parts of our VHDL project set RST=1, then the outputs of TWO_BIT (all derived from the value of bCTRL1) will be initialized to known values. In the paragraphs that follow, we will first talk about where RSTG comes from and then how RSTG is used to produce local resets like RST in Fig. A.1.

In Xilinx FPGAs, there are signals called Global Set/Reset (GSR) and Global Write-Enable (GWE) that are sometimes used to control RSTG during power-ON of the FPGA. However, GSR and GWE are usually unsuitable for this task, mostly because the clocks produced by the FPGA clock-modules (eg. MMCM – see chapter 8) are not guaranteed to be stable when the GSR and GWE signals toggle. However, the MMCM itself outputs a signal called "locked" (see Figs 9.2 and 9.5 in chapter 9) that is asserted (ie. set equal to 1) when the MMCM clock outputs are stable (aka locked). Further, the locked-output of the MMCM is usually asserted well after completion of all the other events associated with FPGA power-ON. For these reasons, the MMCM locked-output is often used to set the value of RSTG.

There are a few easily-solved problems with using the MMCM "locked-output" to set the value of RSTG. First, recall from the HDL-to-Hardware Connection discussion in section 7.5 that signals in our VHDL code correspond to digital registers (flip-flops). So, initializing a VHDL signal is equivalent to initializing a flip-flop, which is often done by asserting the reset (R) input to a flip-flop (see section 7.2). However, the locked-output of the MMCM is de-asserted (ie. set to 0) while waiting for clocks to stabilize, which is just the opposite of what we need for resetting flip-flops. The simple solution is to set RSTG equal to the digital-NOT of the locked-output from the MMCM.

```
01  ----------------------------------------------------------
02  -- Name:        TWO_BIT.vhd
03  -- Description: Two outputs that depend on the sum of two two-bit inputs
04  -- Create Date: 21Mar2019
05  -- Revisions:   14Feb2021 - added reset input called RST, fixed sensitivity list
06  ----------------------------------------------------------
07  library IEEE;
08  use IEEE.STD_LOGIC_1164.ALL;
09  use IEEE.NUMERIC_STD.ALL;
10
11  entity TWO_BIT is
12    port(
13          CLK     : in std_logic;              --logic clock
14          RST     : in std_logic;              --reset
15          IN1     : in unsigned(1 downto 0);   --input #1
16          IN2     : in unsigned(1 downto 0);   --input #2
17          CTL1    : out std_logic;             --control output #1
18          CTL2    : out std_logic              --control output #2
19         );
20  end TWO_BIT;
21
22  architecture MY_CMP1 of TWO_BIT is
23      signal bCTL1 : std_logic;                --declare a signal called bCTL1
24  begin
25      CTL1 <= bCTL1;
26      CTL2 <= not(bCTL1);
27
28      MY_PRC1: process(CLK)                     --outputs: (bCTL1)
29          variable sum1 : unsigned(1 downto 0); --declare a variable called sum1
30          begin
31              if rising_edge(CLK) then
32                  if(RST = '1') then
33                      bCTL1 <= '1';
34                  else
35                      sum1 := IN1 + IN2;
36                      if(sum1 > "10") then      --is sum1 greater than 2 ?
37                          bCTL1 <= '1';
38                      else
39                          bCTL1 <= '0';
40                      end if;
41                  end if;
42              end if;
43      end process MY_PRC1;
44  end MY_CMP1;
```

Fig A.1 VHDL component called TWO_BIT with RST added.

A second problem with using the MMCM locked-output to set the value of RSTG is also easily-solved. Specifically, the locked-output is asynchronous with each of the clocks generated by the MMCM. In chapter 12, we learned that passing around asynchronous signals can lead to metastability. Also in chapter 12, we learned that the metastability problem can (with high probability) be solved using a 2-flip-flop (2FF) synchronizer. Hence, a modified 2FF-synchronizer called the *reset-bridge* (aka *reset-synchronizer*) is often used to bring RSTG into each clock-domain. The reason for modifying the 2FF-synchronizer is that the FPGA power-ON reset needs to be "asynchronous-ON and synchronous-OFF". That is, RSTG and the local resets (ie. the version of RSTG in each clock-domain), need to come ON as soon as power is applied to the FPGA. However, to prevent metastability, the reset in each clock-domain needs to come OFF after RSTG=0 and simultaneously with the rising-edge of the clock for the clock-domain. Shown in Fig A.2 is a VHDL component called RST_BRIDGE.vhd that is a reset-bridge. As I noted in the previous paragraph, there is much similarity between the VHDL in Fig A.2 for the reset-bridge and the VHDL in Fig 12.4 for the 2FF-synchronizer.

```
01  ------------------------------------------------------------
02  -- Name:        RST_BRIDGE.vhd
03  -- Description: reset-bridge brings global resets into a clock domain
04  -- Date: 14Feb2021
05  -- Revisions:   none
06  ------------------------------------------------------------
07  library IEEE;
08  use IEEE.STD_LOGIC_1164.ALL;
09
10  entity RST_BRIDGE is
11      port(
12          CLK      : in std_logic;    --clock in the "receiving" clock-domain
13          RIN_ON   : in std_logic;    --asynchronous reset input #1
14          RIN_OFF  : in std_logic;    --asynchronous reset input #2 (typically set to 0)
15          OUT_RST  : out std_logic    --reset produced for the "receiving" clock-domain
16          );
17  end RST_BRIDGE;
18
19  architecture BEHAVIOR of RST_BRIDGE is
20      signal meta_rst : std_logic;
21
22      attribute ASYNC_REG : string;
23      attribute ASYNC_REG of meta_rst,_RST: signal is "TRUE";
24
25  begin
26
27      PRC_RSB: process(CLK, RIN_ON)   --outputs: (OUT_RST)
28      begin
29          if(RIN_ON = '1') then
30              meta_rst <= '1';
31              OUT_RST <= '1';         --OUT_RST comes ON asynchronously with IN_RST
32          elsif rising_edge(CLK) then
33              meta_rst <= RIN_OFF;    --meta_rst may be metastable, but
34              OUT_RST <= meta_rst;    --OUT_RST is stable and comes OFF
35          end if;                     --synchronously with CLK
36      end process PRC_RSB;
37
38  end BEHAVIOR;
```

Fig A.2 VHDL component called RST_BRIDGE that is a reset-bridge.

The RST_BRIDGE component is used by simply declaring it along with the MMCM component as shown in Fig A.3. Note in line 43 how the global reset, RSTG, for this project is derived from the MMCM locked-output. In the Fig A.3 example, there are domains for each of the three clocks, (CLK166, CLK250, CLK100), and the reset for each of these domains is called RST166, RST250, and RST100. Further, as shown in Fig A.3, each of these separate resets is created from RSTG using a separate instantiation of RST_BRIDGE. So, for example, if we were to pass CLK100 to the input, CLK, of component, TWO_BIT.vhd (see Fig A.1), then we should also pass RST100 to input, RST, of this component.

Occasionally, we want some clock-domains to come out of reset before others. For example, suppose that the CLK100 domain contains HDL that is the master-control for our project. That is, the CLK166 and CLK250 domains simply responds to requests that are sent from the CLK100 domain. For this situation, it is desirable that the CLK166 and CLK250 domains come out of reset before the CLK100 domain, ensuring that the CLK166 and CLK250 domains are ready when the CLK100 domain starts “talking” with them. Ensuring that the CLK100 domain comes out of reset last is easily done using the input called RIN_OUT of RST_BRIDGE.vhd. If we wanted resets in all three clock domains to come OFF at nearly the same time, then each instance of RST_BRIDGE in Fig A.3 should have RIN_OUT set equal to 0. However, you’ll note in line 77 that I have set RIN_OFF equal to RST166 instead of 0. The result is that RST100 is forced to come OFF after RST166 comes OFF. In other words, I have cascaded the reset-bridges for the CLK166-domain and the CLK100-domain. Thus, by cascading reset-bridges, we can make resets for each clock domain come OFF sequentially, rather than all-at-once. You can learn more about cascading reset-bridges from [Cummings, 2004].

```
01  ----------------------------------------------------------------------------------
02  -- Module Name: TOP.vhd
03  -- Description: Top-level VHDL component of the Xilinx Vivado project called my_proj1
04  -- Date: 15Jun2019
05  ----------------------------------------------------------------------------------
06  library IEEE;
07  use IEEE.STD_LOGIC_1164.ALL;
08
09  entity TOP is
10      port(
11           CLK_IN1_P  : in std_logic;
12           CLK_IN1_N  : in std_logic;
13           -- other inputs and outputs for TOP.vhd
14           );
15  end TOP;
16
17  architecture MY_TOP of TOP is
18      signal CLK166, CLK250, CLK100, clks_locked, RST166, RST250, RST100, RSTG : std_logic;
19
20      --Declaration of IP component copied from CLK_GEN.vho (main clocks for this project)
21      component CLK_GEN
22          port(
23               clk_in1_p   : in     std_logic;
24               clk_in1_n   : in     std_logic;
25               CLKOUT0     : out    std_logic;
26               CLKOUT1     : out    std_logic;
27               CLKOUT2     : out    std_logic;
28               locked      : out    std_logic
29               );
30      end component;
31
32      --Declaration of custom component, RST_BRIDGE.vhd
33      component RST_BRIDGE
34          port(
35               CLK         : in    std_logic;
36               RIN_ON      : in    std_logic;
37               RIN_OFF     : in    std_logic;
38               OUT_RST     : out   std_logic
39               );
40      end component;
41
42  begin
43      RSTG <= NOT(clks_locked);    --global reset, RSTG, formed from clks_locked output of CLK_GEN
44
45      --Instantiation of IP component copied from CLK_GEN.vho
46      MMCM1: CLK_GEN
47      port map(
48               clk_in1_p   => CLK_IN1_P,
49               clk_in1_n   => CLK_IN1_N,
50               CLKOUT0     => CLK166,
51               CLKOUT1     => CLK250,
52               CLKOUT2     => CLK100,
53               locked      => clks_locked
54               );
55
56      --Instantiation of custom component, RST_BRIDGE.vhd
57      RS250: RST_BRIDGE
58      port map(
59               CLK         => CLK250,
60               RIN_ON      => RSTG,
61               RIN_OFF     => '0',
62               OUT_RST     => RST250
63               );
64      --Instantiation of custom component, RST_BRIDGE.vhd
65      RS166: RST_BRIDGE
66      port map(
67               CLK         => CLK166,
68               RIN_ON      => RSTG,
69               RIN_OFF     => '0',
70               OUT_RST     => RST166
71               );
72      --Instantiation of custom component, RST_BRIDGE.vhd
73      RS100: RST_BRIDGE
74      port map(
75               CLK         => CLK100,
76               RIN_ON      => RSTG,
77               RIN_OFF     => RST166,
78               OUT_RST     => RST100
79               );
80  end MY_TOP;
```

Fig A.3 Code snippets from top-level VHDL component called TOP.vhd that are associated with the reset-bridge, RST_BRIDGE.vhd.

I have emphasized the importance of initializing signals in your HDL and shown how the reset clause (eg. lines 32-33 of Fig A.1) in a VHDL process can be used to do this initialization. However, the reset clause in Fig A.1 is one of the few shown in this book. My point is that signal initialization (and associated reset clause) are probably not needed for many of the VHDL signals in your project. For example, the component, TWO_BIT.vhd, in Fig A.1 does not need the reset clause if this component is used properly. That is, if your HDL uses the outputs of TWO_BIT.vhd only after valid inputs were provided to TWO_BIT.vhd and only after TWO_BIT.vhd is given time to process the valid inputs, then no reset clause is needed. Further, reset signals undergo timing analysis. Hence, overuse of reset clauses can lead to large fanout of the reset signal and make it difficult for the reset signal to pass timing analysis. The VHDL state machine (see example in chapter 5) is one example were a reset clause is needed to initialize the state of the state machine. Otherwise, the rule is to sparingly use reset clauses in your VHDL processes.

Appendix B. VHDL Sensitivity List

In section 4.3, I said that the sensitivity list for a VHDL process is a list of input-signals to the process. I like this working definition for the sensitivity list because it's easy to remember, it works (except when a process has wait statements – see section 6.4).

You're probably reading this appendix because you're asking for more information about the sensitivity list. However, you may regret (a little) asking because this is one of those subjects that does not have a tidy explanation. In fact, the VHDL creators seemed to realize that many of us were struggling with the sensitivity list concept. So, with the 2008 release of VHDL (aka VHDL-2008), they added a new feature that makes it unnecessary to create or understand the sensitivity list. That is, starting with VHDL-2008, we can simply use, "process(all)", to indicate that we want the IDE tools to decide which signal names should appear in the sensitivity list. For example, in Fig A.1, we could write line 28 as "MY_PRC1: process(all)" instead of "MY_PRC1: process(CLK)". Finally, when using "process(all)", the Vivado IDE requires that you specify VHDL files to have type, "VHDL-2008", rather than simply "VHDL".

So, you now have two easy solutions for creating sensitivity lists; either my working definition or the "process(all)" feature of VHDL-2008. However, if you still asking for more information then read-on.

First, the sensitivity list for a VHDL process is equivalent to an appropriately-placed VHDL wait-statement. For example, in Fig A.1, the sensitivity list, (CLK), found in line 28 of the process called MY_PRC1 is equivalent to the statement, "wait on CLK;", placed between line 42 and line 43. So, from the VHDL viewpoint, the process called MY_PRC1 will execute once at startup. The process will then wait until CLK changes before executing again. These facts do not explain the VHDL rule that you cannot have both a sensitivity list and wait statements in the same process. Perhaps this rule was created to prevent us from effectively creating conflicting wait statements.

Based on comments in the previous paragraph, we find that the sensitivity list is more accurately defined as a list of signals whose changes cause the process to execute (rather than being a list of inputs to the process). So, again consider the sensitivity list, (CLK), for the process called MY_PRC1 in Fig A.1. That is, only changes in the signal called CLK (and only the rising-edge changes in CLK) cause the process to actually do something. Modifying the current sensitivity list to contain more signals, (eg. (CLK,RST,IN1,IN2)), causes no problem because changes in (RST,IN1,IN2) cannot override the fact that changes in CLK must occur first for changes in (RST,IN1,IN2) to have an effect. Also, the Vivado IDE tools will not issue a warning if you put extra and unnecessary input-signals in the sensitivity list. However, the Vivado IDE tools will issue a warning if you forgot to include CLK in the sensitivity list (ie. if you put too few signals in the sensitivity list).

Next, the sensitivity list is used only by the Vivado IDE tools called synthesis and simulation. In the previous paragraph I said that leaving CLK out of the sensitivity list causes a warning. This warning is issued by the synthesis tool and not by the simulation tool. However, after issuing the warning, *the synthesis tool simply ignores the sensitivity list* and does what it needs to do (ie. creates a hardware equivalent to the VHDL process) by looking only at the body of the process.

Although simulation does not issue a warning about CLK missing from the sensitivity list, simulation will probably not execute the process properly. That is, simulation faithfully uses the definition that the sensitivity list contains signals whose changes cause the process to execute. Without CLK in the sensitivity list, simulation may execute the process at unexpected times or not at all (depending on what other signals are in the sensitivity list). In summary, without CLK in the sensitivity list, synthesis issues a warning but does its job correctly, whereas simulation does not issue a warning and does its job incorrectly (or perhaps I should say "in an unexpected way"). Weird huh! A partial explanation for this weirdness is that the synthesis warning (about the missing CLK signal) tells us that simulation results may be unexpected and may not agree with synthesis results.

Finally, the process type (clocked and unclocked) is relevant to this discussion. The only time I have used unclocked processes in this book was for simulation (see Fig 6.1) – and those unclocked processes have wait statements (and hence no sensitivity list). However, for proper simulation, an unclocked process without wait statements must have a sensitivity list that contains all inputs to the process, since changes in each of these inputs can cause the process to do something.

Appendix C. RS232

In section 13.1 we learned that the oversampled-interface method can be used to receive slow-speed data coming into the FPGA. One of my favorite uses of this method is to receive data from an RS232 serial communications interface.

The RS232 interface was once used by personal computers to connect with lots of devices (eg. printer, mouse, games). RS232 certainly qualifies as a slow interface with speeds ranging from 1200 bits-per-second (bps) to about 115200 bps. RS232 is now mostly replaced by the faster USB interface. However, you'll be surprised where and how often RS232 is still used.

In its simplest form, RS232 has only 3 lines/wires called: 1) data-transmit, TXD, 2) data-receive, RXD, and 3) electrical ground, GND. That is, from a personal computer's point of point, data is sent out the TXD line and data is received on the RXD line. Note that RS232 has no clock line! That is, when receiving data on RXD, you won't have a clock signal to use for capturing the data. Thus, RS232 is called an asynchronous communications interface as opposed to the synchronous interfaces like SDR and DDR (see chapters 15 and 16) that have a clock line.

Although RS232 has no clock line, both the sender and receiver use a clock called the *baud-clock* and both must agree on the frequency of this clock. For example, if the RS232 speed is 1200 bps then both sender and receiver agree to use a baud-clock of 1200 Hz. The period of the baud-clock is also the time between digital signal transitions on the RS232 lines, with each transition representing a new bit of data. Further, both the sender and receiver of RS232 understand that data will be sent in *packets* that have the following:

- 1 start-bit
- 8 data-bits
- either 1 or 0 parity-bits, with parity being either even or odd
- 1, 2, or 3 stop-bits.

Thus, before communications can start, both sender and receiver must agree on the packet format (ie. the number of parity-bits, whether parity is even or odd, and the number of stop-bits). Finally, it is understood that the RS232 lines rest/idle in the high (1) state. So, a time-series of the digital transitions on an RS232 line (ie. a data packet) might look like the digital waveform shown in Fig C.1. As noted above, RS232 waveform transitions in Fig C.1 occur at time intervals that are evenly spaced and equal to the period of the baud-clock.

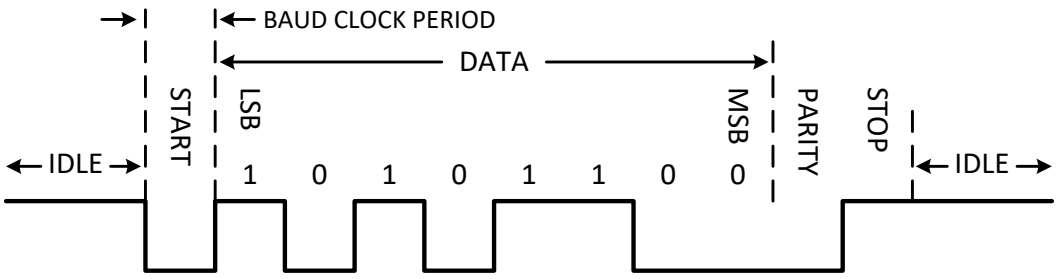

Fig C.1 Example of an RS232 data packet with 11-bits.

So, how are we supposed to receive the RS232 waveform without having a capture-clock edge to locate the center of each data-eye? Well, consider this. Your simple visual inspection of Fig C.1 allows you to easily see the value for each of the 11 bits in this example data packet. You can easily see this because you know beforehand that the packet has 11 bits equally spaced by the baud-clock period and that the packet starts as soon as the data-line transitions from the idle-state(1) to the start-bit(0). In short, the FPGA can use the oversampled-interface method (see section 13.1) to do the same "visual inspection" of the RS232 line that you did to capture/read bits of data from the packet. Specifically, steps used by the FPGA to read a RS232 data packet are:

1. Watch the line until a transition from the idle-state(1) to a start-bit(0) is seen and then go to step-2.
2. Wait **1/2** baud-clock-period and sample the line to read the start-bit (which should be 0). Go to step-3.
3. Wait **1** baud-clock-period and sample the line to get the least-significant-bit (LSB) of the 8-bit data. Go to step-4.
4. Wait **1** baud-clock-period and sample the line to get other bits. If all 11 bits of the data packet have been read and saved then go to step-1, otherwise go to step-3.

The VHDL component called SERIAL_RX shown in Fig C.2 and Fig C.3 uses the oversampled-interface method to implement the above-described steps for receiving a RS232 data packet. The VHDL *signal*, TOG8X, shown in line 14 of Fig C-2 is sometimes called a *toggle* (see appendix D) and is key to the operation SERIAL_RX. Specifically, TOG8X looks like a clock-waveform with a frequency equal to eight times the baud-clock frequency. So, in the 4-steps shown above for reading a RS232 data packet, "1 baud-clock-period" equals 8-cycles of TOG8X. The clock, CLKF, shown in line 13 of Fig C2 is used to oversample the RS232 data line, SDI, shown in line 19. As the comment in line 13 says, CLKF should have a frequency that is much faster (>10x) than the frequency of TOG8X.

SERIAL_RX is used as follows:

- Create CLKF and TOG8X and provide them as input to SERIAL_RX
- Connect the RS232 incoming data line to input, SDI
- Specify the number of parity bits using input, PARITY_EN
- Specify the type (even or odd) of parity checking using input, PARITY_TYP
- Specify the number of stop bits using input, STOP_BITS
- Toggle input, RSTF, high for at least one cycle of CLKF to get SERIAL_RX ready to receive RS232 data coming into input, SDI
- Wait for DOUT_RDY to go high, indicating that a RS232 packet has been received and that data from this packet been placed in output, DOUT. Read the data contained in DOUT.
- Read value of output, ERRB, to see if errors occurred while receiving and reading the data packet.
- Clear the value of ERRB by pulsing input, CLK_ERRB, high for at least one cycle of CLKF.

Also important for use of SERIAL_RX is that all inputs must be in the CLKF clock-domain. As you'll recall from chapter 12, having all signals in the same clock-domain prevents metastability problems that can occur when passing signals from

one clock-domain to another. The 2FF synchronizer discussed in section 12.3 and shown in Fig 12.4 should be used to bring input, SDI, into the CLKF clock-domain.

Note that the oversampling method of receiving RS232 data will (for lack of better term) auto-align an edge of TOG8X with the start-bit of every data packet. This auto-alignment is not perfect because it can be off by ½-cyle of TOG8X, but it is close enough. So, as described in the 4-steps shown above for reading a RS232 data packet, SERIAL_RX simply starts counting cycles of TOG8X to determine the times when it should read a bit of data from the RS232 line.

However, what if the period of TOG8X is off a little? That is, how much error can we tolerate in the period of TOG8X before SERIAL_RX fails to work properly? To answer this question, we'll use a data packet with the maximum length of 13 bits and consider what happens if the TOG8X period is too long by 4%. With 4% error in TOG8X, the cycle-counting to find the time position of the start-bit will end up being (1.04 * 4 = 4.16) cycles of the true TOG8X instead of the necessary 4-cycles. Since 4.16-cycles of the true TOG8X still puts us very close to the center of the start-bit, then the start-bit will be received correctly. However, consider what happens when we try to receive the 13^{th} bit (a stop-bit). When receiving the 13^{th} bit, we will have counted a total of 100-cycles of TOG8X from the start of the data packet. However, because of the 4% error, this will be (1.04 * 100 = 104) cycles of the true TOG8X. That is, when we read the 13^{th} bit, our calculated time location for this bit will be off by 4-cycles of the true TOG8X. However, 4-cycles of the true TOG8X is ½ of the baud-clock period. So, instead of being in the center of the 13^{th} bit we will be very near the edge of the 13^{th} bit and may therefore not read this bit correctly. Thus, from this example calculation, we find that the error in period for TOG8X must be less than 4% if we want to correctly receive all bits from a maximum size (13) RS232 data packet.

The companion component to SERIAL_RX is called SERIAL_TX, which can be used to send data over RS232. A VHDL implementation of SERIAL_TX is shown in Fig C.4 and Fig C.5. Basic usage of SERIAL_TX is to place the byte to be transmitted in input, DIN, and then pulse-high the input, DIN_RDY. If you plan to use and test SERIAL_RX and SERIAL_TX, then consider writing a testbench (see section 6.1) where you do the following:

- connect the output, SDO, of SERIAL_TX to the input, SDI, of SERIAL_RX
- send a byte of data to input, DIN, of SERIAL_TX
- receive a byte of data from output, DOUT, of SERIAL_RX
- compare DIN to DOUT

```
01  ----------------------------------------------------------
02  -- Name: SERIAL_RX.vhd
03  -- Description: Receives a data packet from an RS232 serial communications port
04  -- Create Date: 14Feb2021
05  ----------------------------------------------------------
06  library IEEE;
07  use IEEE.STD_LOGIC_1164.ALL;
08  use IEEE.NUMERIC_STD.ALL;
09
10
11  entity SERIAL_RX is
12      port(
13          CLKF : in std_logic;                    --fast clock (must be much faster than TOG8X)
14          TOG8X : in std_logic;                   --speed(bps) of serial port set by this toggle-signal
15          RSTF : in std_logic;                    --reset
16          PARITY_EN : in unsigned(0 downto 0);    --0=no parity bit, 1=one parity bit
17          PARITY_TYP : in unsigned(0 downto 0);   --0=even-parity, 1=odd-parity
18          STOP_BITS : in unsigned(1 downto 0);    --number of stop bits (*must be 1, 2, or 3*)
19          SDI : in std_logic;                     --serial data incoming line (**in CLKF domain**)
20          DOUT : out std_logic_vector(7 downto 0);  --data byte read from serial data line
21          DOUT_RDY : out std_logic;               --goes high when data is ready on DOUT
22          ERRB : out std_logic_vector(1 downto 0);  --error bits (nonzero values indicate errors)
23          CLR_ERRB : in std_logic                 --CLR_ERRB=1 => clear ERRB
24      );
25  end SERIAL_RX;
26
27  architecture BEHAVIOR of SERIAL_RX is
28      type state_type1 is (idle,s1,s2,s3,s4);
29      signal comm_state : state_type1;
30      signal tog8xp : std_logic;
31
32  begin
33
34      P1: process(CLKF)                           --outputs: (DOUT, ERRB)
35          variable bit_cnt : unsigned(3 downto 0);
36          variable edge_cnt : unsigned(2 downto 0);
37          variable rx_bits : std_logic_vector(12 downto 0);
38      begin
39          if rising_edge(CLKF) then
40              if(RSTF = '1') then
41                 ERRB <= "00";
42                 comm_state <= idle;
43              else
44                  case comm_state is
45                      when idle =>        --idle: wait for start-bit (SDI=0)
46                          DOUT_RDY <= '0';
47                          bit_cnt := "1001" + PARITY_EN + STOP_BITS; --packet bit count
48                          if(SDI='0') then                    --start-bit received?
49                              edge_cnt := "011";              --prepare to count 4 rising edges of TOG8X
50                              comm_state <= s1;
51                          else
52                              comm_state <= idle;
53                          end if;
54
55                      when s1 =>          --s1: read one bit of data
56                          if((tog8xp='0') and (TOG8X='1')) then --rising-edge on TOG8X received?
57                              if(edge_cnt = "000") then       --countdown of TOG8X rising-edges finished?
58                                  rx_bits(11 downto 0) := rx_bits(12 downto 1);
59                                  rx_bits(12) := SDI;         --read one bit from the serial data line
60                                  bit_cnt := bit_cnt - "0001";  --adjust bit countdown for newly received bit
61                                  comm_state <= s2;
62                              else
63                                  edge_cnt := edge_cnt - "001";
64                                  comm_state <= s1;
65                              end if;
66                          else
67                              comm_state <= s1;
68                          end if;
69
70                      when s2 =>          --s2: continue reading bits until entire data packet received
71                          if(bit_cnt = "0000") then           --were all bits for the data packet read?
72                              comm_state <= s3;
73                          else
74                              edge_cnt := "111";              --prepare to count 8 rising-edges of TOG8X
75                              comm_state <= s1;
76                          end if;
```

Fig C.2 Part-1 of a VHDL component, SERIAL_RX, used to read data packets from an RS232 data line.

```
                    when s3 =>          --s3: check for frame error and remove stop bits
                        if(rx_bits(12) = '0') then          --check for frame error (last stop-bit = 1?)
                            ERRB(1) <= '1';
                        end if;
                        if(STOP_BITS = "01") then           --shift out the stop bits
                            rx_bits(12 downto 1) := rx_bits(11 downto 0);
                        elsif(STOP_BITS = "10") then
                            rx_bits(12 downto 2) := rx_bits(10 downto 0);
                        else
                            rx_bits(12 downto 3) := rx_bits(9 downto 0);
                        end if;
                        comm_state <= s4;

                    when s4 =>          --s4: check for parity error and put data on DOUT
                        if(PARITY_EN = "1") then            --is there a parity bit?
                            for i in 4 to 11 loop           --do the parity check
                                rx_bits(12) := rx_bits(12) xor rx_bits(i); --XOR data bits with parity bit
                            end loop;
                            rx_bits(12) := rx_bits(12) xor std_logic(PARITY_TYP(0));
                            if(rx_bits(12) = '1') then      --do results indicate a parity error?
                                ERRB(0) <= '1';             --report a parity error
                            end if;
                            DOUT <= rx_bits(11 downto 4);   --put received-data on DOUT
                        else
                            DOUT <= rx_bits(12 downto 5);   --put received-data on DOUT
                        end if;
                        DOUT_RDY <= '1';                    --indicate that data is ready on DOUT
                        comm_state <= idle;

                    when others =>
                        comm_state <= idle;
                end case;
            end if;    --END of "if (RSTF = '1')..."
            tog8xp <= TOG8X;            --tog8xp used to detect a rising-edge on signal, TOG8X
            if(CLR_ERRB = '1') then     --handle user-input to clear the error bits
                ERRB <= "00";
            end if;
        end if;        --END of "if rising_edge(CLKF)..."
    end process P1;
end BEHAVIOR;
```

Fig C.3 Part-2 of a VHDL component, SERIAL_RX, used to read data packets from an RS232 data line.

```
01  ----------------------------------------------------------
02  -- Name: SERIAL_TX.vhd
03  -- Description: Sends a data packet to an RS232 serial communications port
04  -- Create Date: 14Feb2021
05  ----------------------------------------------------------
06  library IEEE;
07  use IEEE.STD_LOGIC_1164.ALL;
08  use IEEE.NUMERIC_STD.ALL;
09
10  entity SERIAL_TX is
11      port(
12          CLKF : in std_logic;                    --fast clock (must be much faster than TOG8X)
13          TOG8X : in std_logic;                   --speed(bps) of serial port set by this toggle-signal
14          RSTF : in std_logic;                    --reset
15          PARITY_EN : in unsigned(0 downto 0);    --0=no parity bit, 1=one parity bit
16          PARITY_TYP : in unsigned(0 downto 0);   --0=even-parity, 1=odd-parity
17          STOP_BITS : in unsigned(1 downto 0);    --number of stop bits (*must be 1, 2, or 3*)
18          SDO : out std_logic;                    --serial data outgoing line
19          DIN : in std_logic_vector(7 downto 0);  --data byte to be transmitted
20          DIN_RDY : in std_logic;                 --1 => data is ready on DIN
21          TX_DONE : out std_logic                 --goes high when DIN has been transmitted
22      );
23  end SERIAL_TX;
24
25  architecture BEHAVIOR of SERIAL_TX is
26      type state_type1 is (idle,s1,s2,s3,s4);
27      signal comm_state : state_type1;
28      signal tog8xp : std_logic;
29
30  begin
31
32      P1: process (CLKF)                          --outputs:(TX_DONE,ERRB)
33          variable bit_cnt : unsigned(3 downto 0);
34          variable edge_cnt : unsigned(2 downto 0);
35          variable tx_bits : std_logic_vector(12 downto 0);
36      begin
```

Fig C.4 Part-1 of a VHDL component, SERIAL_TX, used to send data packets to an RS232 data line.

```
        if rising_edge(CLKF) then
            if(RSTF = '1') then
               comm_state <= idle;
            else
                case comm_state is
                    when idle =>        --idle: wait for DIN_RDY=1 (ie. is data ready to be transmitted?)
                        SDO <= '1';
                        bit_cnt := "1001" + PARITY_EN + STOP_BITS;  --packet bit count
                        if(DIN_RDY = '1') then              --is data ready to be transmitted?
                            tx_bits(0) := '0';              --start bit
                            tx_bits(8 downto 1) := DIN(7 downto 0); --data bits
                            tx_bits(12 downto 9) := "1111";   --stop bits and parity bit (all 1 for now)
                            if(PARITY_EN = "1") then          --is parity enabled?
                                tx_bits(9) := std_logic(PARITY_TYP(0));
                                for i in 1 to 8 loop          --calculate parity bit
                                    tx_bits(9) :=  tx_bits(9) xor tx_bits(i);
                                end loop;
                            end if;
                            edge_cnt := "000";              --prepare to count 4 rising edges of TOG8X
                            TX_DONE <= '0';
                            comm_state <= s1;
                        else
                            TX_DONE <= '1';
                            comm_state <= idle;
                        end if;

                    when s1 =>          --s1: transmit one bit of data
                        if((tog8xp='0') and (TOG8X='1')) then --rising-edge on TOG8X received?
                            if(edge_cnt = "000") then         --countdown of TOG8X rising-edges finished?
                                SDO <= tx_bits(0);            --put bit on serial data line
                                tx_bits(11 downto 0) := tx_bits(12 downto 1);
                                bit_cnt := bit_cnt - "0001";  --adjust bit countdown for newly sent bit
                                comm_state <= s2;
                            else
                                edge_cnt := edge_cnt - "001";
                                comm_state <= s1;
                            end if;
                        else
                            comm_state <= s1;
                        end if;

                    when s2 =>          --s2: continue transmitting bits until entire data packet is sent
                        if(bit_cnt = "0000") then             --were all bits for the data packet sent?
                            bit_cnt := "1111";                --prepare to countdown pause in state, s3
                            comm_state <= s3;
                        else
                            edge_cnt := "111";                --prepare to count 8 rising-edges of TOG8X
                            comm_state <= s1;
                        end if;

                    when s3 =>          --s3: pause to get proper width of last stop bit and for SDO idle
                        if((tog8xp='0') and (TOG8X='1')) then
                            if(bit_cnt = "0000") then
                                comm_state <= s4;
                            else
                                bit_cnt := bit_cnt - "0001";
                                comm_state <= s3;
                            end if;
                        else
                            comm_state <= s3;
                        end if;

                    when s4 =>          --s4: wait for DIN_RDY=0, then indicate that transmit is done
                        if(DIN_RDY='0') then                --did caller lower DIN_RDY?
                            TX_DONE <= '1';                 --tell caller that transmit of DIN is done
                            comm_state <= idle;
                        else
                            comm_state <= s4;
                        end if;

                    when others =>              --others: bad state
                        comm_state <= idle;
                end case;
            end if;    --END of "if (RSTF = '1')..."

            tog8xp <= TOG8X;            --tog8xp used to detect a rising-edge on signal, TOG8X
        end if;        --END of "rising_edge(CLKF)..."
    end process P1;
end BEHAVIOR;
```

Fig C.5 Part-2 of a VHDL component, SERIAL_TX, used to send data packets to an RS232 data line.

Appendix D. Toggle

Often, when someone says they want a slow clock for their design, what they really want is a *toggle*. A toggle can be described as a VHDL *signal* that looks like a *clock* but is not a clock because it is not routed in the FPGA clock tree. Using a toggle instead of a slow clock has the advantages discussed below.

For example, suppose we want an FPGA is to blink a Light Emitting Diode (LED) at a rate of 1Hz. Perhaps you are saying "Easy! -just create a 1Hz clock inside the FPGA and send it to the LED". Well, no, for several reasons. First, it is not straightforward to create a 1Hz clock because the MMCM clock module that we talked about in chapter 8 has a minimum clock-output frequency of about 4MHz. So, we would need to use special clock buffers that can divide-down (to 1Hz) the MMCM clock-output and put the resulting 1Hz clock into the FPGA clock tree. Then, as described in section 15.3.6, an ODDR component is needed to forward the 1Hz clock out of the FPGA to the LED. So, not so easy.

Another reason we don't want to use a 1Hz clock for our LED blinker is that clocks are routed through the FPGA clock tree to reduce clock skew and help our design pass timing analysis. However, there should be no concerns about timing analysis for the LED blinker nor for anything else that we do with the 1Hz clock. So, why tie-up precious clock tree resources with a 1Hz clock and create a 1Hz clock-domain that timing analysis does not need to analyze.

Using a toggle is a simpler and more reasonable way to implement the LED blinker. The VHDL in Fig D.1 shows how to create a toggle called tog1 and how tog1 is used to create the control signal, led_ctrl, for blinking the LED. Note that tog1 is declared in line 01 to be a VHDL *signal*. The clock output, CLK1, of the MMCM in line 04 is used by the VHDL process, P1, in lines 07-20 to assign a value to tog1. The resulting tog1 waveform looks like a clock with a period equal to CNTMX (see line 08) cycles of CLK1. Further, during each period of tog1, tog1 is high for only one cycle of CLK1. Finally, in lines 24-32, the VHDL process called P2 uses tog1 to assign values to led_ctrl.

Some important things to note about the VHDL in Fig D.1 are:

- Signal, tog1, is a VHDL signal in the CLK1 clock-domain (ie. tog1 is NOT a clock).
- Because tog1 is NOT a clock, you should NOT use it as a clock (see bad-example process, P3, in lines 36-41).
- Process, P2, toggles led_ctrl at a rate equal to the period of tog1 – even though this process is clocked at the rate of CLK1.
- The frequency of tog1 is very accurate since it is derived from the very accurate CLK1. So, in terms of frequency accuracy, a toggle is not inferior to a clock. That is, the LED will blink at a very accurate 1Hz rate.

Toggles are often used to implement slow speed FPGA IO such as RS232 (see appendix C), I2C (see section 13.3), and SPI (see section 13.3). You'll recall from our earlier discussions (see section 13.3) about slow IO, that the oversampled-interface method is often used and we "count cycles" to make the IO pass timing analysis "by design". Further, we "count cycles" of a toggle rather than directly counting cycles of a clock.

Another reason we use toggles for slow IO is that they make it unnecessary to use *gated clocks*, which are clocks that do not run continuously. That is, sometimes the clock line associated with slow IO is stopped. Creating a gated clock in the FPGA and writing the associated timing constraints and timing exceptions is a hassle. Handling a gated clock that is coming into the FPGA is even more of a hassle. However, gating a toggle is easy because it is simply a VHDL signal (and NOT a clock). Gating a toggle is usually done with simple VHDL conditional logic (ie. simply ignore the toggle when it is not needed). Finally, no timing constraints/exceptions are needed for the toggle because it is NOT a clock.

In conclusion, we often use toggles instead of slow speed clocks because:

- overall, they make HDL coding easier
- they have the frequency accuracy of a clock
- they avoid having to deal with the minimum output frequency limitation of the MMCM and with the difficulties of divide-down clock buffers (see discussion of BUFR in appendix E).

- they free-up space in the FPGA clock tree for use by high-speed clocks (which really need to be in the tree)
- they can be used instead of gated clocks
- overall, they make timing analysis easier (for us and for the FPGA tools)
- they do not require timing constraints nor timing exceptions
- they do not create new clock domains (ie. they can be used in any VHDL process that uses the clock, CLK1, from which the toggle was created (ie. *the toggle is a signal in the CLK1 clock-domain*), (ie. there is no clock crossing associated with using the toggle in the CLK1 clock-domain).

```
01  signal CLK1, tog1, led_ctrl : std_logic;
02  .....
03  --
04  -- Here, use an MMCM to create the clock called CLK1
05  --
06  -- Create toggle, tog1, in the CLK1 clock-domain
07  P1: process(CLK1)
08      constant CNTMX : integer := 1000;      --divider for CLK1
09      variable cnt : integer range 0 to CNTMX-1;
10  begin
11      if rising_edge(CLK1) then
12          if(cnt = 0) then
13              tog1 <= '1';
14              cnt := CNTMX-1;
15          else
16              tog1 <= '0';
17              cnt := cnt - 1;
18          end if;
19      end if;
20  end process P1;
21  --
22  --
23  -- Proper use of tog1 blink the
24  P2: process(CLK1)
25  begin
26      if rising_edge(CLK1) then
27              -- you could do something here at the rate of CLK1...
28          if(tog1 = '1') then
29             led_ctrl <= not(led_ctrl);
30          end if;
31      end if;
32  end process P2;
33  --
34  --
35  -- An example of how *NOT* to use a toggle
36  P3: process(tog1)                          -- <<< DON'T DO THIS
37  begin
38      if rising_edge(tog1) then              -- <<< DON'T DO THIS
39              -- do something here ???...
40      end if;
41  end process P3
```

Fig D.1 VHDL code snippets showing how to create and use a toggle.

Appendix E. Alignment and Balance for Clocks

In this book, we have used the MMCM and the Vivado Clocking Wizard (see chapter 9) for almost all our clocking needs. As you'll recall from chapter 9, the MMCM accepts a single clock input called the *base clock*. From the base clock, the MMCM can generate several output clocks for use in our design. As we'll learn in this section, these wonderful Xilinx tools automatically manage some very important things for us.

Perhaps the most important thing that the MMCM manages is to *align* all its output clocks. Generally, align means that when the MMCM starts outputting clocks (eg. following power-up of the FPGA), the first rising-edge of every output clock occurs at the same time. An exception to this definition of align is when the MMCM is used to phase shift one of the clocks (see section 16.1.7). However, the align feature of the MMCM ensures that this intentional phase-shifting

occurs the same way following every power-up of the FPGA/MMCM. Finally, the MMCM ensures that the initial alignment and *frequency* of it output clocks does not drift as voltages and temperature inside the MMCM change.

Alignment of the MMCM output clocks is important because it means that the clocks are *synchronous* – which means that the timing analysis done by Vivado is valid over all PVT variations in the FPGA that are within-specifications. Conversely, if separate clocks are shifting in time with respect to each other, then the clocks are no longer synchronous and much of the timing analysis done by Vivado becomes invalid – meaning that we won't know whether our design works or not! FYI, clocks that shift in time with respect to each other are called *mesochronous* clocks.

Another important thing that the MMCM does is called *balancing* its clock outputs. Balancing means that the MMCM places the clock signals into the FPGA clock tree with only small constant offsets to the phase alignment of the clocks. Balancing clocks will be further explained with the aid of Fig E.1, which shows the MMCM clock module called MMCM1 that is used throughout this book. Note that the base clock enters the MMCM at the input called CLKIN1 and produces three output clocks called CLKOUT0, CLKOUT1, and CLKOUT2. As described above, these three output clocks will be phase aligned when they leave the main block of the MMCM. After leaving the main block of the MMCM, each clock passes through a *clock-buffer* called a BUFG, which places the clock signal into the FPGA clock tree for distribution throughout the FPGA. As a clock signal passes through a BUFG, it is delayed a little. So, if we were to pass some of the MMCM output clocks through a one BUFG and other clocks through two BUFGs then some clocks will be delayed more than others. Thus, there will be a resulting offset in the phase alignment of the clocks as they enter the clock tree. Fortunately, the MMCM knows about the offset in phase alignment caused by a BUFG and actively works to keep this offset constant over PVT. So, the clocks remain synchronous. However, an offset in the phase alignment between clocks can make it more difficult for your design to pass timing analysis. *So, when possible, we route each clock output of the MMCM through the same-number and same-type of clock buffers.* You'll note in Fig E.1 that the Vivado tools have automatically routed each clock output of the MMCM through only one BUFG, resulting in balanced clocks.

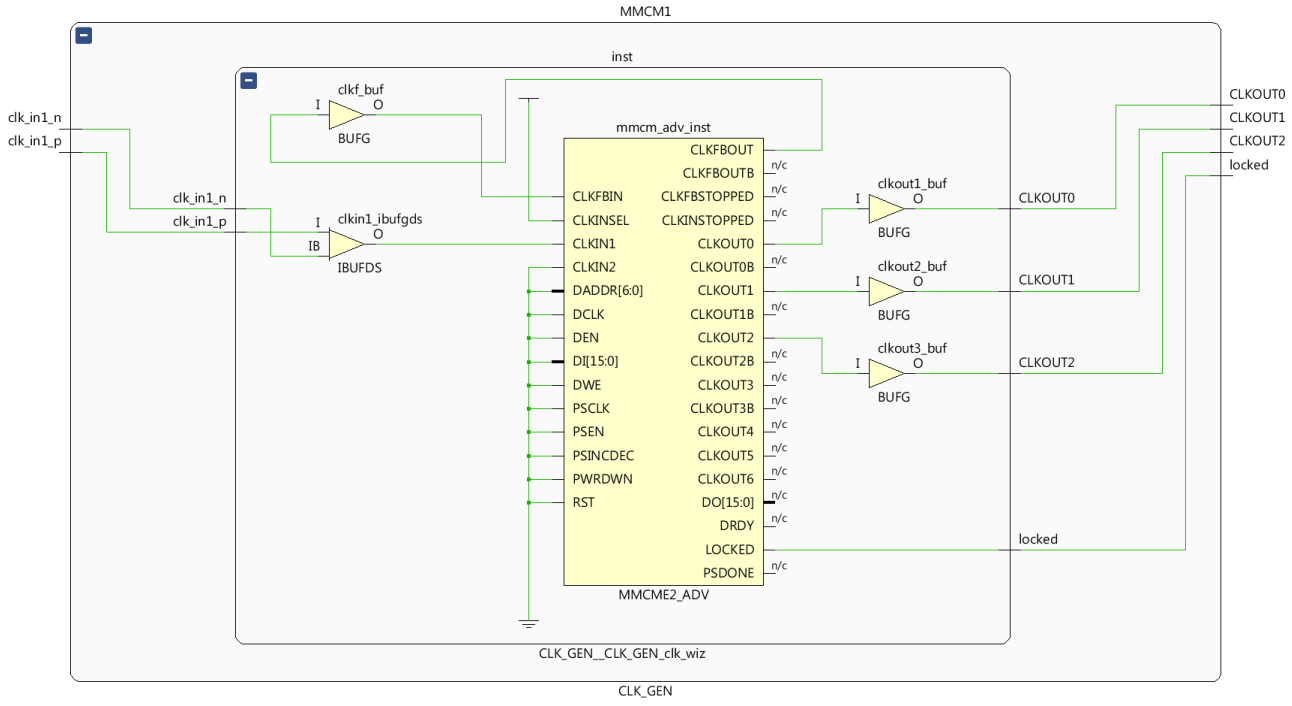

Fig E.1 The MMCM clock module called MMCM1 that is used throughout this book.

Xilinx FPGAs have other types of clock buffers, one of which is called the BUFR. The BUFR is special because you can configure it to divide-down a clock. This divide-down feature is useful when you want to create a slow clock whose frequency is below the minimum output frequency of the MMCM (see discussion in appendix D). However, the divide-down feature can affect clock alignment and prevent clocks from being synchronous if you are not careful. The Xilinx documentation describes a procedure for phase aligning the outputs of several BUFRs that have a common input clock and are used to divide-down clocks. Although the Xilinx documentation says that the outputs of several BUFRs can be phase-aligned, it is unclear whether we can make these BUFR output clocks phase aligned with clock outputs from other types of clock buffers.

In conclusion and whenever possible, use the MMCM to create clocks for your projects – because it makes life so much easier for us by automatically managing clock alignment and balance. When you cannot use the MMCM, be aware that you might need to manually manage clock alignment and balance.

GLOSSARY

- **architecture (VHDL)** – the part of a VHDL *component* that describes how the component behaves (ie. how it processes inputs to become outputs)
- **asynchronous** – a term used in sequential logic to indicate that a digital signal is NOT changing at times corresponding to digital *clock* transitions
- **bitstream** – another name for the file containing an FPGA *configuration*.
- **center aligned** – a term used in digital I/O indicating that rising-edges of the I/O clock occur halfway between the times associated with transitions for the I/O data.
- **clock** – a periodic digital signal that is typically very stable. Clocks are an important part of *clocked logic*.
- **clock crossing** – another way to say that a digital signal is being sent from one *clock domain* to another.
- **clock domain** – In a large *clocked logic* circuit, the portion of this circuit that uses only one digital clock. For example, if a large circuit has two clocks called CLK1 and CLK2, then the portion of the circuit that uses only CLK1 is called the CLK1 clock-domain and the portion that uses only CLK2 is called the CLK2 clock-domain.
- **clock tree** – special circuits and pathways in the FPGA used to buffer and distribute digital *clocks*.
- **clock-module (FPGA)** – special circuitry found in an FPGA that is used to buffer and generate digital *clocks*. Typically, a single *clock* input to the FPGA is fed to the clock-module. The clock-module is configured to produce other clocks needed for *clocked logic*.
- **clocked logic** – roughly, digital circuitry in which signals are passed between digital registers at times corresponding to digital *clock* transitions. *Clocked logic* circuits usually contain *combinational logic*.
- **combinational logic** – roughly, digital circuitry (eg. AND, OR, and NOR gates) that does not use digital registers.
- **component (VHDL)** – VHDL code can be used to describe digital circuits. This code is organized into modules called VHDL *components*, which roughly correspond to the digital components found in digital circuits.
- **concurrency (VHDL)** – a word in VHDL indicating that software modules (eg. separate VHDL components and separate VHDL processes) execute in parallel – or concurrently with each other.
- **configuration (FPGA)** – after HDL has been compiled (ie. after running synthesis and implementation) the resulting file (aka bitstream) describes *configuration* of hardware found in the FPGA
- **constraints (FPGA)** – special instructions written in the *Tcl* programming language and placed in the constraints file used by the *IDE*. There are generally two types of constraints called physical and timing. Physical constraints are used to relate VHDL signals to FPGA hardware (eg. VHDL signal, SIG1, connects to FPGA pin, E5). Timing constraints supply information needed for *timing analysis*.
- **data arrival time** – a term used to describe calculations done by *timing analysis*. Generally, this is the actual-time that data arrives at the digital *register* located at the end of a *timing path*.
- **data-clock** – another name for the clock associated with source-synchronous FPGA I/O
- **data eye** – In a *clock logic* circuit, a data signal typically changes at times corresponding to the rising-edge of a *clock* signal. The *data eye* is another name for the segment of time between consecutive rising-edges of the *clock* when the data signal is stable (ie. not changing).
- **data required time** – a term used to describe calculations done by *timing analysis*. Generally, this is the limiting-value for *data arrival time* at the digital *register* located at the end of a *timing path*. *Setup* checks done by *timing analysis* will pass if *data-arrival-time* is before *data-required-time*. *Hold* checks done by *timing analysis* will pass if *data-arrival-time* is after *data-required-time*.
- **DDR** – **D**ouble **D**ata **R**ate. A method of clocked-digital communication where data is sent on both the rising and falling-edge of a clock.

- **declaration (VHDL)** – when a VHDL *component*, C2, is used inside of another VHDL component, C1, then C2 must be *declared* inside C1. The *declaration* for C2 is a short block of code that is very like the *entity* part of the C2 *component*.
- **delay-error** – a term sometimes used to refer to a number found in a set_max_delay timing constraint
- **delay-difference** – a term sometimes used to refer to the quantity, (data-trace-delay minus clock-trace-delay), used to write a set_max_delay timing constraint
- **DSP** – **D**igital **S**ignal **P**rocessor. FPGA hardware used to do arithmetic that is beyond the capability of the *LUT*.
- **edge aligned** – a term used in digital I/O indicating that rising-edges of the I/O clock occur at the same time as transitions for the I/O data.
- **entity (VHDL)** – the part of a VHDL *component* that lists and describes inputs and outputs of the *component.*
- **exceptions (FPGA)** – special instructions written in the *Tcl* programming language and placed in the constraints file used by the IDE. Like timing *constraints*, exceptions are information used by timing analysis. Exceptions specify that timing analysis is to be done in a special way (or not at all) for certain digital circuits.
- **fabric (FPGA)** – Most of the digital circuitry found in an FPGA lies in the FPGA *fabric*. A block of digital hardware called a *slice* is repeated many times throughout the FPGA to produce the FPGA *fabric*.
- **fanout** – for a digital component, D1, fanout is a number indicating how many other digital components are connected to the output of D1
- **flip-flop** – see *register*
- **flow** – a word used by FPGA vendors that means the recommended sequence for using *tools* found in their *IDE*.
- **four corners (analysis)** – part of *timing analysis* that considers random and uncertain effects (eg. *clock* jitter, and *PVT* variations in propagation-time through hardware).
- **forwarded clock** – another name for the *clock* used in *source-synchronous* digital *I/O*.
- **FPGA** – **F**ield **P**rogrammable **G**ate **A**rray. A integrated circuit having reconfigurable digital hardware.
- **glitch** – typically, an undesirable digital output from *combinational logic* that is caused by the different arrival times of digital inputs to the combinational logic. *Clocked logic* is used to prevent glitches from becoming a problem.
- **glossary** – a fancy name for an alphabetical list of words and their definitions.
- **GUI** – **G**raphical **U**ser **I**nterface. In the old days, computers were controlled by typing text onto the computer screen. Now, computers and software are mostly controlled using a mouse to click on graphical things such as icons, menus, and buttons. This modern user interface to computers and software is called a GUI.
- **HDL** – **H**ardware **D**escriptive **L**anguage. FPGA programming is done using an HDL (eg. VHDL or Verilog).
- **hold (requirement)** – a requirement for the digital *register* that data sent to input, D, remain stable for some time (the hold-time) after a *clock* transition is sent to input, C. *Timing analysis* checks that the hold-time requirement is satisfied for every register found in the FPGA *configuration*.
- **I2C** - **I**nter-**I**ntegrated **C**ircuit communication interface. Typically used for communication between ICs on a circuit board.
- **IBUFDS** – a component found in Xilinx FPGAs used to convert a digital signal from 2-wire (LVDS) to 1-wire
- **IC** – **I**ntegrated **C**ircuit. An electrical device built on semiconductor material that often combines (ie. integrates) the functionality found in many discrete-circuit devices.
- **IDDR** – **I**nput **D**ual **D**ata **R**ate. A digital component found in Xilinx FPGAs that is used for *DDR* input.
- **IOB** – **I**nput **O**utput **B**lock. A group of digital components located next to each I/O pin of the FPGA.
- **I/O** – **I**nput/**O**utput.
- **IP** – **I**ntellectual **P**roperty. Software (HDL) modules provided by an FPGA vendor.
- **IDE** – **I**ntegrated **D**evelopment **E**nvironment. A collection of software modules provided by an FPGA vendor that is used to write HDL and to *configure* FPGAs sold by the vendor.

- **implementation (FPGA)** – the process of converting the results of *synthesis* into usage of physical devices and interconnects found inside the FPGA. The results of implementation are also called an FPGA *configuration*. Implementation is one of the *tools* found in an IDE and typically runs simultaneously with *timing analysis*.
- **instantiation (VHDL)** – when a VHDL *component*, C2, is used inside of another VHDL *component*, C1, then C2 must be instantiated inside C1. Instantiation is a short block of code that identifies the signals from C1 that connect to the inputs and outputs of C2.
- **JTAG** – **J**oint **T**est **A**ction **G**roup. An electronics industry association that develops methods of verification and test. In FPGA work, this refers to a multi-pin interface used to load a *configuration* into the FPGA.
- **jitter** – A digital clock is said to have jitter if its period is not stable.
- **latency (VHDL)** – the delay between start-time of digital signal processing and the end-time when results of the processing are available. Latency/delay time is often specified as the number-of-cycles for a digital *clock*.
- **LED** – **L**ight **E**mitting **D**iode. A semiconductor device that emits light when you pass electrical current through it.
- **LUT** – **L**ook **U**p **T**able. A digital device found in the FPGA that is often used to create *combinational* logic. A LUT can also be used to perform simple arithmetic.
- **LVDS** – **L**ow-**V**oltage **D**ifferential **S**ignaling. A two-wire method of sending a digital signal over long distance.
- **metastability** – a problem in *sequential-logic* circuits that occurs when *signal* transitions arrive near-simultaneously with *clock* transitions at a digital *register*.
- **MMCM** – **M**ixed **M**ode **C**lock **M**anager. A type of *clock-module* found in Xilinx FPGAs.
- **OBUFDS** – a component found in Xilinx FPGAs used to convert a digital signal from 1-wire to 2-wire (*LVDS*).
- **ODDR** – **O**utput **D**ual **D**ata **R**ate. A digital component found in Xilinx FPGAs that is used for *DDR* output.
- **optimization (synthesis)** – things done during synthesis that result in more efficient use of the available FPGA resources. optimization. Familiar parts of synthesis optimization are LUT-combination and register-merging.
- **pipelining** – a method of spreading digital signal processing over multiple clock cycles to help the FPGA *configuration* pass *timing analysis*.
- **PLL** – **P**hase **L**ocked **L**oop. A type of *clock-module* found in FPGAs.
- **process (VHDL)** – like subroutines in other programming languages, one or more VHDL *processes* are usually found in the *architecture* of a VHDL *component*.
- **PROM** – **P**rogrammable **R**ead-**O**nly **M**emory. In FPGA work, PROM is nonvolatile memory used to store an FPGA *configuration*. Many of the Xilinx FPGAs have no internal PROM, but they can automatically load a *configuration* from an external PROM device.
- **PVT – P**rocess(manufacturing), **V**oltage, and **T**emperature. A term that is used when describing the properties of digital devices. For example, "For this digital *register*, the maximum hold time requirement over *PVT* is 4ns.".
- **RAM – R**andom **A**ccess **M**emory. Circuits found in the FPGA used to store the value of digital *signals*. Then, while the FPGA is running, you can write values to RAM and then later read back these values.
- **register (digital)** – roughly, a device that passes the value of a digital *signal* from its input, D, to its output, Q, at times corresponding to digital *clock* transitions.
- **reset bridge** – a digital circuit used to bring a reset signal into a *clock domain* so that the reset comes ON (ie. goes high) *asynchronously* and goes OFF *synchronously* with the domain's clock.
- **ROM** - **R**ead-**O**nly **M**emory. Circuits found in the FPGA used to store constant values. That is, during FPGA configuration, constant values are placed in ROM. Then, while the FPGA is running, the constant values can be read back from ROM.
- **RS232** – an old and slow communication interface that was once popular on personal computers for talking with peripheral devices (eg. printer, mouse). However, is still found and used in some unlikely places.
- **RTL** – **R**egister **T**ransfer **L**evel. In VHDL, RTL is a programming style that generates *clocked logic*. Generally, the signals in RTL-style programming correspond to digital registers and the operations (eg. Boolean algebra, arithmetic, if-then-else) performed on the signals correspond to combinational-logic.

- **SDR** – **S**ingle **D**ata **R**ate. A method of clocked-digital communication where data is sent on only one edge (either rising or falling) of the clock.
- **sensitivity list (VHDL)** – roughly, the part of a VHDL *process* that is a list all signal-inputs to the process. The *sensitivity list* is important only for *simulation* and not for *synthesis*.
- **setup (requirement)** – a requirement for the digital *register* that data sent to input, D, remain stable for some time (the setup-time) before a *clock* transition is sent to input, C. *Timing analysis* checks that the setup-time requirement is satisfied for every register found in the FPGA *configuration*.
- **sequential logic** – see *clocked logic*
- **signal (VHDL)** – like digital signals that are passed into and out of digital components, VHDL signals are passed into and out of a VHDL *component*.
- **simulation –** a software method of testing software (see also *testbench*)
- **slack –** a number reported by timing analysis for every *timing path* found in the FPGA *configuration*. Positive slack indicates that the *timing path* passed timing analysis and by how much. Negative slack indicates that the *timing path* failed timing analysis and by how much.
- **slice (FPGA) –** Most of the digital circuitry found in an FPGA lies in the FPGA *fabric*. A block of digital hardware called a *slice* is repeated many times throughout the FPGA to produce the FPGA *fabric*.
- **source synchronous –** a type of digital I/O where the transmitting device sends both a clock and data to the receiving device.
- **state machine –** a fancy name for doing things sequentially. That is, each "thing" to be done is called a state and when a state is completed then control is passed to another state.
- **synchronous** – a term used in *clocked logic* to indicate that a digital *signal* is changing at times corresponding to digital *clock* transitions.
- **synchronizer** – a digital circuit used to pass signals from one *clock domain* to another that greatly reduces the effects of *metastability*.
- **synthesis** – roughly, the process of compiling/translating HDL code into symbols and interconnections that represent a *clocked logic* circuit. Synthesis is one of the tools found in an *IDE*.
- **system synchronous –** a type of digital I/O where a remote clock is used by both the transmitting device and the receiving device.
- **Tcl** – **T**ool **c**ontrol **l**anguage. A programming language used to write FPGA *constraints* and *exceptions*.
- **testbench –** special software written for *simulation* testing of other software. Typically, the testbench generates digital inputs and sends them to the software being tested.
- **timing analysis** – The digital circuit *configuration* inside the FPGA that implements our *HDL* can be divided into *timing paths*. Timing analysis checks every *timing path* to determine whether it could cause metastability.
- **timing arcs** – part of a timing path. Generally, diagrams associated with timing paths show curved arrows (ie. *timing-arcs*) that identify signal propagation time along each part of the *timing path*.
- **timing path** – parts of the FPGA *configuration* that are checked by *timing analysis.* Generally, a *timing path* extends from the *clock* input pin of one digital *register* through *combinational logic* to the data input pin of another digital *register*. Signal propagation time along a *timing path* is used by *timing analysis.*
- **timing closure** – another way to say that an FPGA project has passed *timing analysis.*
- **toggle** – a VHDL *signal* that often looks like a *clock* but is not a clock and is not routed in the FPGA *clock tree*
- **tools (FPGA)** – another name for the collection of software modules found in an IDE
- **variable (VHDL)** – typically, an intermediate term that is used within a VHDL *process* to help describe what the VHDL *process* is doing to a VHDL *signal*.
- **Vivado** – the name of IDE software developed and distributed by Xilinx.
- **VHDL** – a popular **H**ardware **D**escriptive **L**anguage (HDL). The "**V**" stands for VHSIC or Very High-Speed Integrated Circuit.

- **wizard (FPGA)** – a portion of the *IDE* software that helps you configure *IP code*.
- **WNS** – **W**orst **N**egative **S**lack. The most negative value of *slack* found by *timing analysis* after fully analyzing the FPGA *configuration*.
- **Xilinx** – a major manufacturer of FPGA devices