Firmware Developer's Manual
Chapter F1
Introduction to FPGA Development

150 Lucius Gordon Drive, West Henrietta, NY 14586 - 9687
(585) 444 7074
http://www.skutek.com

Summary: This chapter of the Developer's Manual is a gentle introduction to the art of programming the field programmable gate array (FPGA) that is present on our boards. We will not cover each and every detail of the black magic. We will rather walk the reader through a set of the HDL code snippets lifted from our own work with the intent to provide useful examples.

Table of Contents

1 The objective of this chapter

This chapter of the Developer's Manual provides an introduction to the art of programming the field programmable gate array (FPGA) that is present on our boards. We will not cover each and every detail of the black art. We will rather walk the reader through a set of the HDL code snippets lifted from our own work with the intent to provide useful examples. We plan to achieve the following.

- Explain what is the FPGA and what it is not.
- Explain the difference between programming the FPGA and programming a processor.
- Advise how to start with FPGA programming and what to expect.
- Provide examples of FPGA constructs that we found useful in our own work.

The chapter was motivated by real life. One of our customers decided to develop his own FPGA code, what is not a piece of cake at all. This chapter will help jump start the effort. It will also help other board users to develop their own FPGA configurations.

Our company is helping the customers in their FPGA applications in two separate ways.

- We provide FPGA programming services to our customers.
- The FPGA firmware can be developed by the customer. The template for the FPGA firmware development is available upon request at no cost. The template consists of the User Constraint File (UCF) with all the FPGA pins prepopulated in the correct locations. The code file written in the HDL language consists of the PORT declarations corresponding to the UCF file. The body of the HDL file may be empty to allow the developers to fill it with their own code.

2 The FPGA literature

Extensive literature on FPGA programming is available. We recommend the following reading.

1. a. Pong P. Chu, FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version.
   https://academic.csuohio.edu/chu_p/rtl/fpga_vhdl.html
   b. FPGA Prototyping by VHDL Examples 2nd edition: Xilinx MicroBlaze MCS SoC
   https://academic.csuohio.edu/chu_p/rtl/fpga_mcs_vhdl.html
   c. Pong P. Chu, FPGA Prototyping by Verilog Examples:Xilinx Spartan-3 Version.
   https://academic.csuohio.edu/chu_p/rtl/fpga_vlog.html
   d. FPGA Prototyping by SystemVerilog Examples: Xilinx MicroBlaze MCS SoC.
   https://academic.csuohio.edu/chu_p/rtl/fpga_mcs_sv.html
2. H.F-W.Sadrozinski, J.Wu, *Applications of FPGA's in Scientific Research* (book).
3. V.A.Pedroni, *Circuit Design and Simulation with VHDL* (book).
4. Xilinx Reference Manuals and User Guides.
5. Xilinx Application Notes available from http://www.xilinx.com/. Xilinx keeps reshuffling their website at an amazing rate, and therefore we cannot provide web pointers that are guaranteed to work. The reader may try the following:
   https://www.xilinx.com/support.html#documentation

We recommend the book by Sadrozinski and Wu because it is the only literature position addressing applications of FPGAs in High Energy Physics. The books by V.A.Pedroni and by P.P.Chu are very

well written introductions to FPGA programming. The books by P.P.Chu describe substantial and useful projects using various Xilinx FPGA familis. Xilinx manuals cover the specifics of the Xilinx FPGA clocking, buffering, and memory structures.

3 FPGA architecture and programming

We will now describe the fundamentals of the Field Programmable Gate Array (FPGA) architecture. The name “gate array” was coined long ago when the FPGA chips indeed consisted of logic gates and little else. Nowadays the FPGAs contain many high-level blocks such as memory blocks or arithmetic-logic units. A modern FPGA consists of many silicon blocks (named cores or components) such as flip-flops, memory registers, logic gates, and arithmetic-logic units. The components are connected with data paths (“wires” or “signals”) that can be selectively enabled or disabled during the chip “configuration”. Before the configuration the chip is dormant and it is doing nothing. All the paths are closed. The configuration usually happens at power-up. It consists of loading the “configuration file” (also called the “bit file”) from the flash memory to the FPGA chip. Configuration can be also performed on demand while the system is running. After the configuration has occurred, some data paths are open, while others are still closed. Those paths, that are open, form an interconnection pattern (the “netlist”) between the components. The data words flow along the open paths, from component to component, according to a periodic “clock” which must be supplied to the FPGA chip. On each clock cycle the data item is transformed by a particular component, and then it is transferred to the next component at the end of the clock cycle. This mode of operation is commonly named “a bucket brigade”. A more formal name is a “pipeline” formed with the components strung together, one after another.

The art of FPGA programming consists of devising the network of connections between the FPGA components in order to form the pipeline, which transforms the data according to the desired algorithm. The conceptual difficulty lies in the fact that the network bears little resemblance to the algorithm, at least at the first sight. The FPGA programmer must develop the connections needed to realize the desired result. One should remember that the FPGA “program” does not describe a process unfolding in time (like a computer program would do), but rather a network of active connections between the components. Compared with the computer program, there is an additional level of indirection between the FPGA “program” and the desired result.

Fortunately, understanding of the FPGA operation does not need to be detailed to the last bit. The task of FPGA programming is greatly simplified with a high-level Hardware Description Language (HDL) such as VHDL or Verilog. The connections are specified in a synthetic form in HDL, and then automatically translated into the actual network for the particular FPGA. Nevertheless, the FPGA programming is pretty demanding even when using the high-level HDL. Details like numerical error propagation or integer fixed-point arithmetic need to be understood in order to transform the abstract algorithm into the FPGA implementation.

The DDC digitizer is composed of two independent processing units, the FPGA and the ARM Sitara processor (DSP). The FPGA is connected to the General Purpose Memory Controller (GPMC) bus of the CPU. The FPGA is a *memory-mapped peripheral* with several *registers* assigned to hardcoded addresses in the CPU memory space. The FPGA addresses are collected in a dedicated *header file* referenced by the software programs. The definitions contained in the header file must agree with the addresses that are hardcoded in the FPGA firmware.

4 Structure of the FPGA project

We will use VHDL for the most of this chapter. A Verilog code is very similar, but it uses different keywords. Both languages are known as Hardware Description Languages (HDL). We will use the word “program”, even though a more appropriate term would be “netlist configuration”. The HDL project under the ISE development environment is composed of the following.

- The User Constraint File (UCF) collects the physical information such as pin locations, I/O voltages, I/O slew rates, clock frequency, and timing requirements. The format of the file is documented in the Xilinx Constraints Guide available from ISE Help menu. Our free template includes the UCF file with all the information already prepopulated.
- The top level HDL file establishes the communication between the netlist that is internal to the FPGA and the FPGA pins. The correspondence is defined in the PORT declaration of the top level file. The signal names in the PORT and in the UCF file should be identical.
- In addition to the PORT, the top level file also contains the ARCHITECTURE of the design written in HDL. It is OK to put all the code in this place in small projects. Larger projects should be divided into several files to improve maintainability.
- The lower level files are similar to the main file. They also have the port and the architecture sections. The names of their port “pins” do not refer to the physical pins, but rather to the internal “pins” of the main file. The lower-level files “plug into” the top level file as “components” defined in the lower level files.
- The lower level files can themselves refer to the next lower level. The hierarchy continues, in principle indefinitely.
- In addition to components defined in the HDL files by the developer, there are two types of components provided by Xilinx. The *library components* are described in *Libraries Guides* available from ISE help. These components are typically small. The Core Generator components are large components that can be extensively customized with Core Generator interactive menu system. Core Generator can be started from the Windows Start menu.
- The detailed information is available in the *XST User Guide* and other guides that are available from ISE Help menu.

ISE development system also supports the mixed-language projects where components are developed in VHDL, Verilog, ABEL, or schematic diagrams. We will not cover these details in the present tutorial. Extensive documentation on this and other advanced topics is available from the ISE Help menu.

The new development environment is named Vivado. It is similar to ISE, though of course different. It is also better, at least according to Xilinx. Practically speaking, Spartan-6 development is free under ISE 14.7 (VirtualBox version), while Artix development is free under Vivado.

5 Two kinds of logic circuits, combinatorial and clocked

Logic circuits can be broadly divided into two kinds. A *combinatorial* circuit is composed of logic gates, multiplexers, and other standard logic components connected together. Such a circuit is realizing a *logic function*, also known as a *boolean function*. An example of a logic function might be

```
Y <= A and B or C;            – Example in VHDL
assign Y = A & B | C;         // Example in Verilog
```

where A, B, C represent *signals* (named *wires* in Verilog) whose values can be 0 or 1, represented by LOW or HIGH voltage levels. The logic functions are constructed according to the boolean algebra of logic values 0 and 1. The boolean circuits are also known under the name *asynchronous logic*. The boolean operators *and* and *or* (& and | in Verilog) are implemented as logic gates in the actual circuit.

The boolean circuits can be of any complexity. Complex such circuits will have many *levels*. A level means “output of this gate is connected to the input of the next gate”. Every gate contributes certain *propagation delay* to the overall response time of the circuit. The propagation time across the gate is specified in the FPGA data sheet. It depends on the FPGA implementation technology, which is fixed for a given FPGA family. In addition to the propagation across the gates, there is also a propagation between the gates, which depends on the inter-gate separation. This part of the delay varies from project to project. It can change between implementations of the same project, because component *place and route* is performed stochastically by the software. There is no guarantee that the component gates are placed at fixed locations without resorting to manual *floor planning*, which is an advanced technique better to be avoided. Propagation between the gates also depends on temperature.

The response time of a combinatorial circuit can vary from single nanoseconds for simple circuits to tens of nanoseconds in case of many levels. Multilevel asynchronous circuits can produce *glitches*, also known as *hazards*. A glitch is a momentary pulse at the output that is due to unequal propagation delays through parallel branches of the multilevel circuit. Since the propagation delays are never exactly equal, there is always a risk of a glitch in every asynchronous circuit of appreciable complexity. The glitches can appear or disappear when the temperature is changing because of the temperature dependence of the propagation delays.

In principle, the FPGA can be used to implement a purely combinatorial circuit of tremendous complexity, because it contains many millions of logic gates. The response time of such a circuit would be long, and the potential for glitches would be huge. Therefore, the FPGA is always using a *clock* that is a periodic square wave connected to the dedicated *clocking pin*. Quite often there are more than one clock in the FPGA system. The DDC digitizers use two clocks: the *main board clock* is used for any purpose such as clocking the ADCs, while the *processor clock* is used to connect the FPGA to the processor. The clock pins are preassigned in the free template that you received from us.

The circuits that are clocked are also named *synchronous*. The FPGA works according to this principle. We call it a “bucket brigade”. A more formal name is a “pipeline” formed with the structural elements strung together, one after another. In the rest of this chapter we will be dealing with synchronous logic most of the time.

Preparing for the following discussion, we remind that the actions of a typical synchronous circuit are latched on the LOW → HIGH transition of the clock. The opposite edge is typically “wasted” for doing nothing. The “waste” is due to the fact that the D-flip flops have only one clock input. They can change their state on the particular clock edge, but not the other. The particular edge is typically chosen LOW → HIGH. In case the other edge was chosen, then the opposite one would be “wasted” anyway. There is very little benefit in using the HIGH → LOW clock edge in a regular design. The complementary edges are typically used in the Dual Data Rate (DDR) circuits, which is an advanced topic. Please read any DDR application note for more information about the DDR technology.

6 A seemingly synchronous logic, which in fact is combinatorial

The following example was discussed on the Oberon mailing list in December 2019 in the context of FPGA interfacing to an asynchronous static RAM chip (ASRAM). This chip presents a challenge because it does not have a clock pin. It rather expects a sequence of signals, such as addresses, data, and read/write strobes, whose edges follow one another with time sequence specified in nanoseconds. (I.e., the ASRAM data sheet is saying something like "*the WRn edge must follow the address transition within a time window, no sooner than X and no later than Y, specified in nanoseconds*".) The challenge of designing such an interface is discussed by P.P.Chu in references **1a** and **1c**, chapter 11. The example of the WRn signal is discussed in **1c** on page 293. This signal must transition LOW to HIGH when the address and data are stable, what happens in the 2nd half of the clock cycle, after the clock has transitioned from HIGH to LOW. This may be problematic in the system using the rising edges.

A seemingly logical way of generating the proper WRn transition LOW → HIGH during the 2nd half of the clock cycle was discussed by P.P.Chu as follows. (Professor Niklaus Wirth also used this solution in his RISC5 soft core project, which precipitated the aforementioned discussion.) The method is known as "*clock gating*". The goal is to create a WRn signal transitioning HIGH on the falling edge of the clock. This is implemented by forming a logic OR of the inverted clock signal with the WRlong signal which is LOW during the entire clock cycle, for both the HIGH and the LOW clock phases. The combination (WRlong OR ~clk) yields (0 OR 0) = 0 during the 1st half period, and (0 OR 1) = 1 during the 2nd half period. It means that (WRlong OR ~clk) is the desired logic function, which provides the transition from LOW to HIGH on the falling clock edge.

```
WRn <= WRlong OR NOT clk;          – VHDL
assign WRn = WRlong | ~ clk;       – Verilog
```

This simple solution has a problem that the instance of the transition is poorly defined because both signals contribute equally to the occurrence of the transition. Whichever signal makes the transition later than the other, will define the time instance. This creates a problem, because the clock distribution tree is rigidly defined in the FPGA, while the regular logic routing is subject to variations. The arrival time of the WRlong signal will depend on the peculiarities of the routing of a particular design. (And even for the same design it can stochastically vary with the place-and-route execution.) This creates an essentially unreproducible design, which is a bad feature indeed.

So we have to introduce the definition of *proper synchronous logic*, as opposed to *seemingly synchronous* shown above. The very fact of using the clock does *not* make the logic properly synchronous. The clock must also be used in the proper way, which means *employing the flip flops*. Combinatorial functions are not properly synchronous, even if they use the clock as one of their inputs. And indeed, techniques such as *clock gating* (shown above) are strongly discouraged by the FPGA experts because the signal timing is poorly defined with such circuits.

The properly designed clocked circuit will *use the clock signal as the input to the flip flop*. Feeding the clock to any regular gate is poor design. It can only be used if the edge timing is not at all important, which is seldom the case. The reason that the flip flops should be used for clocking is that the clock signals are distributed along their *dedicated clock distribution network tree*. All flip flops, in the entire chip, will receive their clock signals at the same time (within tight tolerances), but only in the case when the clock is distributed with the dedicated networks. The clock can never leave such a network and be routed to the regular gate, because any such paths are not well specified timing-wise.

The proper way of coding the above circuit is emphasizing the fact that the clock and the regular logic are not equal. The clock is latching the regular signals *on the edges of the clock*. This is done with a flip flop, whose inputs are not equal. (They are equal in case of the regular gate.) The notation may look bizarre. This notation is how the HDL is coding for the flip flop. For clarity this design is using both clock edges, which may not be supported by the FPGA circuitry. If this was the case (as signaled by the compile time error) then the designer should resort to the DDR techniques discussed by P.P.Chu on page 293 of reference **1c**.

```
– VHDL; the Verilog equivalent is left as an exercise
process (clk) begin
    if raising_edge (clk) then
      WRn <= WRlong;
    else if falling_edge (clk) then
      WRn <= NOT WRlong;
    end if;
end process;
```

7 How the timing requirements are met in practice?

Now we can elaborate on how the timing requirements are actually met. Conceptually this is done in the following steps.

1. Arrange the “bucket brigade” using the registered signals. Every registered signal is a flip flop by definition. VHDL does not use a keyword for such signals. Verilog is using the keyword *reg*. When one *reg* is assigned the value of another *reg*, then the data transfers only happen at the clock edges. The chain of assignments establishes the “bucket brigade”. Remember that both in VHDL and in Verilog you use the <= operator within the timing text block to perform the clocked data transfer. Use the construct *PROCESS (clk)* in VHDL. In Verilog it is *always @ (posedge(clk))*.

2. Write the combinatorial expressions to be executed between the registered assignments. These expressions are *logic functions* composed of logic operators.

3. Introduce the clock period constraint into your UCF file (in ISE) or XDC file (in Vivado). In plain English: You need to tell the synthesis how fast you want to run the design.

4. Run the synthesis and examine the timing reports. Look for the “timing violations”. If the software cannot make the combinatorial functions propagate from one flip flop to the next one within the clock period, then it will inform you of the fact. It will also tell you, which logic function is taking too long.

5. Now you need to think. What is more important to you? Is the long logic expression so valuable that it cannot be calculated in pieces? Then you need to increase the clock period and drop the clock frequency. Conversely, is the clock frequency important? Then you need to break the logic functions into pieces, introduce new intermediate stages into the bucket brigade, and assign partial results to the intermediate flip flops.

In practice you will need to learn the timing constrain syntax, which is very murky and not obvious with ISE and perhaps a bit easier with Vivado. Keeping in mind the preceding explanations should provide a reasonable guideline for further practice.

8 Controlling the FPGA with a register

We just said that the *boolean logic* is represented with values 0 and 1, meaning LOW and HIGH voltage levels. It is true in the logic textbooks, but it is not how the electrical connections work with FPGAs. There is one more electrical state that we need to use, and that is OFF, coded as 'Z' in the VHDL language. The device that is always *actively driving* the connection either LOW or HIGH cannot be be put on a *bus* where it may cause the *contention* with other devices. We need to be able to turn the device OFF, and this is what the high impedance state 'Z' is doing for us. We are now ready to see the example circuit coded with the following VHDL snippet.

```
IOBUS <= local_signal WHEN read_enable = '1' ELSE (others => 'Z');
```

This line of VHDL should be read as follows. The IOBUS is declared in the PORT. It is assigned to the off-chip bus in the UCF file. The bus can be *driven* by the `local_signal` when the `read_enable` is requesting the FPGA to do so. Otherwise the FPGA cuts off the output drivers. The VHDL construct `others` is a handy notation meaning “all the other bits of IOBUS”. This shortcut evaluates to “all the bits” when used as shown. We will now show the entire VHDL “program” surrounding the snippet.

```
1. -- CTRL_reg_CPU_writes.vhd.  A register CPU --> FPGA.
2. -- (C) Wojtek Skulski 2003-2011.
3. -- This register can be written and read back by the CPU.
4. -- It does not read data from the FPGA fabric.
5. -- It can only read back the data previously written to it.
6. -- Input: CLK, CS, WR, RD active HIGH, the usual r/w controls.
7. --   CS should be tied to the address decoder at the higher level.
8. -- Output:  regout is static data to drive the controlled circuitry.
9.       library IEEE;
10.      use IEEE.std_logic_1164.all;
11.
12.      ENTITY CTRL_reg_CPU_writes IS
13.        GENERIC (regWdt:   INTEGER := 16);
14.        PORT (
15.            CLK   : in STD_LOGIC;   --- system clock from BF
16.            CS    : in STD_LOGIC;   --- circuit select
17.            WR    : in STD_LOGIC;   --- write enable
18.            RD    : in STD_LOGIC;   --- read enable
19.            IOBUS : inout STD_LOGIC_VECTOR (regWdt-1 downto 0);
20.            regout: out STD_LOGIC_VECTOR   (regWdt-1 downto 0)
21.          );
22.      END CTRL_reg_CPU_writes;
23.
24.      ARCHITECTURE CTRL_reg_behavior OF CTRL_reg_CPU_writes IS
25.      SIGNAL rena, wrena: STD_LOGIC;
26.      SIGNAL local: STD_LOGIC_VECTOR(regWdt-1 downto 0);
27.
28.      BEGIN      -- ARCHITECTURE IMPLEMENTATION
29.      wrena <= '1' WHEN ((CS='1') AND (WR='1') AND (RD= '0')) ELSE '0';
30.      rena  <= '1' WHEN ((CS='1') AND (WR='0') AND (RD= '1')) ELSE '0';
31.
32.      IOBUS <= local WHEN rena = '1' ELSE (others => 'Z');
33.
34.      register: PROCESS (CLK, wrena) BEGIN
35.          IF rising_edge(CLK) THEN
36.            IF (wrena = '1') THEN
```

```
37.              local <= IOBUS;      – from CPU to fabric
38.            END IF;
39.          END IF;  -- CLK
40.      END PROCESS register;
41.
42.      regout <= local; -- static output from the register to controlled logic
43.      END CTRL_reg_behavior;
```

Now we will explain what this circuit is doing. First of all, this register is a *component* declared in its own file. The component file is only a template that can be *instantiated* in the main VHDL file as many times as needed. In this respect the register is like an integrated circuit that gets purchased from a store in several copies. In the electronic industry all the copies are identical, but here they are not because the template is *parametrized* with the keyword GENERIC meaning that the bit width is specified while instantiating the template. The default width "16" is only a placeholder. The component has several inputs: the clock, the active-high read/write strobes RD and WR, and the active-high chip select CS. The IOBUS will be connected to the system bus when instantiating the component. Finally, `regout` is a *static array of bits* that can be connected to anything inside the FPGA. If FPGA had LEDs on top, we could connect `regout` to these LEDs and turn them on and off by writing to the register.

The register is *static* in the following sense: imagine that we write a "1" to one of the bits. This "1" will stay permanently there until we decide to write a "0" to the same bit. Therefore, this component is intended to *control* some circuitry in the FPGA. The register bits can turn some options ON and OFF. Alternatively, the value written to the register can be used as a calculation coefficient.

The register is read/write in the limited sense. Any value written to the register can be read back by the processor. We say that the value has been *latched*. However, the register cannot be used to read any output calculated by the FPGA. The register is *unidirectional*. It can transfer the bit pattern from the processor to the FPGA, and it can remember its value, but the other direction does not work. The reason for this restriction is that any circuit within the FPGA can have only one driver. Since the driver in this circuit is driving towards the FPGA, adding the opposite driver is not possible.

We said that the latched value can be read back, what implies that some sort of *memory* is implemented in the section beginning with the keyword PROCESS. The word "register" is a hint for the reader, but it is a purely decorative *label*. Most often such labels are omitted from the VHDL program. The memory is always implied when the keyword PROCESS is used *together with a clock*, which is the case in this example. The clock is not indicated by the name CLK, but rather by invoking the predeclared function `rising_edge`. It is the combination of the PROCESS with the statement `IF rising_edge`, that makes the PROCESS into a *flip-flop*. A flip-flop is a memory element that can stay ON or OFF. The names "register" and "flip-flop" are synonymous.

If this sounds confusing, then it really is. The reason for the confusion is that the VHDL programs are not being literally translated into the FPGA circuitry. The VHDL compilers (and the Verilog as well) are using a method called "inference". The compilers do not translate the text, but rather they scan the text looking for familiar patterns. When the compiler sees the pattern, it will "infer" the circuitry. Another name for infer is "guess". Yes, it is right. The compiler is guessing what the programmer meant to say. It is in the programmer's best interest that the compiler is guessing right. The programmer is advised to follow the code patterns that were suggested by the compiler vendor. The patterns are available from the ISE Toolbar after pressing the rightmost icon that looks like a bulb. Navigate to *VHDL → Synthesis Constructs* and use only these snippets in order to help the compiler guess right.

One has to get used to such patterns. After a while they will become the second nature. The patterns may look confusingly similar to one another. Here is an example of an apparent similarity.

```
1.  PROCESS (CLK, RST) BEGIN
2.     IF rising_edge(CLK) THEN
3.         IF (RST = '1') THEN
4.             local <= (others => '0');
5.         ELSE
6.             local <= previous;
7.         END IF;
8.     END IF;  -- CLK
9.  END PROCESS;
```

This snippet may look confusingly similar to the previous one. It translates to a one-clock delay. The bit pattern `local` will be an exact copy of the `previous`, but delayed by a single clock cycle. The signal RST implements a *reset* that holds the bit pattern at “00..0”. We did not use labels in this example.

9 Reading from the FPGA device using a register

We need a complementary operation to the one discussed previously. The following code will allow to read a single word from the FPGA. This register is again unidirectional for the reason discussed earlier.

```
1. -- CTRL_reg_CPU_reads.vhd.
2. -- (C) Wojtek Skulski 2003-2011.
3. -- This register cannot be written by the CPU. There is no WR control.
4. -- Data either comes from another clock domain or it is static.
5. -- If the other domain is not stopped then local will be complete mess.
6. -- Input from CPU: CLK, CS, RD active HIGH.
7. --   CS should be tied to the address decoder at the higher level.
8. -- Input from the fabric:  data to be read by the CPU from the FPGA fabric.
9.       library IEEE;
10.use IEEE.std_logic_1164.all;
11.
12.ENTITY CTRL_reg_CPU_reads IS
13.  GENERIC (regWdt:   INTEGER := 16);
14.  PORT (
15.       CLK    : in STD_LOGIC;     --- system clock from CPU
16.       CS     : in STD_LOGIC;     --- circuit select
17.       RD     : in STD_LOGIC;     --- read enable
18.       IOBUS  : inout      STD_LOGIC_VECTOR( regWdt-1 downto 0);
19.       fabric: in          STD_LOGIC_VECTOR (regWdt-1 downto 0)
20.    );
21.END CTRL_reg_CPU_reads;
22.
23.ARCHITECTURE CTRL_reg_read_behavior OF CTRL_reg_CPU_reads IS
24.SIGNAL local: STD_LOGIC_VECTOR(regWdt-1 downto 0);
25.
26.BEGIN      -- ARCHITECTURE IMPLEMENTATION
27.IOBUS <= local WHEN (CS='1') AND (RD= '1') ELSE (others => 'Z');
28.
29.PROCESS (CLK) BEGIN
30.    IF (rising_edge(CLK)) THEN
31.        local <= fabric; -- the crucial modification is here
32.    END IF;  -- CLK
33.END PROCESS;
34.END CTRL_reg_read_behavior;
```

This code is very similar to the previous one, but the signal drivers are reversed. The FPGA fabric is now driving the `local` register upon the rising edge of the clock. Previously it was the IOBUS whose value was latched, but now it is the `fabric` that is coming from within the FPGA. The modification is easy to overlook when looking at the text.

The lesson from the examples is that a seemingly minor modification of the source code can lead to profoundly different circuitry. In one case we described a register controlling the FPGA, in the other case we described a readout register, and in the meantime we described a delay by a single clock.

10 Dealing with clock domains

The previous example included a warning *"if the other domain is not stopped then local will be complete mess"*. It refers to the fact that the FPGA operates under two different clocks. One is the "main clock" that is driving the daughter card. The other is the CPU clock that is driving the readout bus. The circuitry that is synchronized to a particular clock is called the *clock domain* of this clock.

The two clocks are completely independent. Imagine that data words are changing under both clocks. In the ADC clock domain the fresh ADC samples are being written to the word named `fabric` in the previous example. At the same time the Blackfin is reading the same word using the `local` register that is controlled by the CPU clock. It is easy to imagine that `fabric` may change in the middle of the `local` being read out. The result will be a "complete mess" because some newly arrived `fabric` bits will make it through the `fabric` → `local` transition, and some perhaps not. One should remember that the circuitry is never perfectly balanced along all data paths. Some bits will be delayed by a fraction of a nanosecond more than other bits, and the slower bits will not make it.

The comment was warning against such a situation and at the same time it provided a hint that "the other domain should be stopped". It does not mean that the clock itself should get stopped. Stopping the clock will throw all the phase lock loops (PLL) out of synch. Clocks should never get stopped. However, the data acquisition *can* be stopped. And it should, because there is little point in reading the sample that is changing during the readout. Once the data words in the ADC clock domain get static, the other clock domain can safely read them out.

The simplest method of dealing with multiple clock domains is to stop changing the data words in one clock domain, while the words are being used in the other domain. There are other methods allowing both domains to keep running, but they are more complicated. For example, the designer may use a first-in, first-out buffer (FIFO) between the two domains. The FIFOs provide means of "crossing the clock boundary" to avoid the bit synchronization problems. The FPGA programmer has to study the relevant chapters describing the particular solutions recommended by Xilinx. (Keep in mind that other FPGA vendors may have implemented the relevant components differently from Xilinx.)

We conclude this section with two recommendations concerning multiple clock domains.

1. Always stop changing the data in the clock domain that is producing the data before using the data in another clock domain. This solution is pretty radical, but it is guaranteed to work.

2. Read the relevant chapters from the FPGA textbooks dealing with the particular FPGA family that you are using.

11 External hard core processor bus

There are two ways of connecting a processor to the FPGA design: (**1**) either an *external hard CPU* or (**2**) an *internal soft CPU*. The third case would be an *internal hard CPU*, like found in Zynq devices. Such hard FPGA/CPU combinations are more like two separate chips, involving a hard interface between the hard silicon CPU and the FPGA fabric, even though they are enclosed under the same lid.

We will now discuss the two-chip solution (1), where an ***external hard silicon*** is added to the FPGA. We will illustrate the bus design with the actual Analog Devices Blackfin BF561 processor interface.

The *microprocessor bus* consists of an address, data, a clock CLK, and control strobes: chip select CS, write enable WR, output enable OE, and read enable RE. The latter two strobes provide essentially the same functionality. They may be merged into one signal. If they are separated, then their timings are different and only one of them is needed. Such details can be found in the processor data sheet.

Most hard silicon chips use an *active LOW* convention for the controls. In such a case they are denoted with an added ”n”, “b” (for “bar”), or “#”. For example, both OE# and OEn mean “OE active LOW”.

```
ENTITY Blackfin_BF561 is PORT (
-- FPGA is connected as BF external memories AMS0 and AMS1
-- BF AMS0 and AMS1 spaces are both configured in 32-bit mode
-- ADDR(1:0) are not active in 32-bit mode.
-- All addresses must be on 32-bit boundaries. ADDR(1:0) are both ignored.
    BF_CLK    : in STD_LOGIC;  -- clock from the BF561 (120 MHz)
    BF_AMS0_b : in STD_LOGIC;  -- AMS0: LOW chip select for 0th 64 MB range
    BF_AMS1_b : in STD_LOGIC   -- AMS1: LOW chip select for 1st 64 MB range
    BF_AWE_b  : in STD_LOGIC;  -- LOW write strobe,  AWE#
    BF_ARE_b  : in STD_LOGIC;  -- LOW read strobe,   ARE#
--  BF_AOE_b  : in STD_LOGIC;  -- LOW out enable, AOE# (use either AOE# or ARE#)
    BF_ADDR   : in    STD_LOGIC_VECTOR(25 downto 2);    -- ADDR increments by 4
    BF_DATA   : inout STD_LOGIC_VECTOR(31 downto 0);    -- BF data bus
);
END Blackfin_BF561;
```

There are a few things to be noted in these declarations. The meaning of *read* and *write* is from the BF point of view. All the signals except BF_DATA are driven by the processor. They are declared with direction “in”. It means that the processor is the *bus master*, while the FPGA is the *bus slave*. It has the benefit that the bus cycle involves neither a handshake nor an acknowledgment from the FPGA. The bus cycle will complete even in the abnormal situation, when the FPGA is not even configured. (Which makes no sense, but it can happen). It means that an abnormal condition at the FPGA side will not automatically deadlock the processor, what could happen, if the bus cycle required a handshake.

The BF561 bus is shared among the SDRAM and the Asynchronous Memory Spaces (AMS). The word *asynchronous* is a misnomer, because the AMS protocol is synchronous to the clock BF_CLK. Nevertheless, Analog Devices named it this way. Both the ADDR and the DATA bus carry the signals directed to the FPGA as well as to the SDRAM chips outside the FPGA. It is an old design style, because newer CPU chips use two separate buses for accessing the SDRAM and the (a)synchronous RAM. The internal FPGA circuits are seeing a lot of SDRAM activity not directed to the FPGA. The remedy consists of using the AMS strobes to open the tristate buffers to only the AMS transactions. The SDRAM transactions stay behind the input gates.

The DATA bus between the BF561 and the FPGA is *bidirectional*, indicated with the keyword *inout*. It means that the same 32 wires are sometimes driven by the CPU, and sometimes by the FPGA. The direction is defined by the data strobes AWE#, ARE#, and AOE#. The latter two are functionally equivalent. The logic inside the FPGA should drive the DATA bus only when it is expected to do so. Otherwise it will create *contention*, which means both sides of the DATA bus are driving the same wires. Such an abnormal condition can even burn the chips. A question then arises, how the designer can avoid a possible catastrophe? (Unlike in politics, a catastrophe in electronics benefits nobody, so it is better avoided.)

There are two ways to avoid burning the chips. The hardware method consists of inserting series 50 ohm resistors into the DATA path. There are special quad or octal resistor packs designed for this application. Resistors will limit the current. They will also prevent signals from ringing. A diligent board designer should always use such resistor packs. A diligent firmware designer should reduce the FPGA output drive current to a low value, like 2 mA or 4 mA. There are configuration options for doing this. Low drive current will prevent burning the chips even if the contention happens.

The bidirectional bus protocol is meant to limit the number of wires at the expense of both the increased chip complexity and reduced performance. The former is of little concern because silicon is cheap and powerful these days. The performance would improve if the two directions were assigned to two separate paths routed in parallel. Some very high performance chips are doing just that, but in most cases it is too much of a hassle. Anyone who has routed over fifty Blackfin wires will agree that doubling the number is not desired. One should note that the address, data, and strobes are doubled rather than just the data part of the bus.

12 Internal on-chip soft core processor bus

Despite their lower performance, the soft cores are becoming increasingly popular due to their flexibility. Soft cores exist in many varieties, from very simple 8-bit microcontrollers (PicoBlaze), 32-bit processors (MicroBlaze, Nios-2), all the way up to 64-bit RISC-V. In this section I will focus on a 32-bit soft core named RISC5 authored by Professor Niklaus Wirth. It is different from RISC-V despite a similar name.

The designer of an on-chip soft processor bus does not need to worry about wire routing. The FPGA is providing thousands of pre-routed wires which are ready to use. Using the tristate buffers is not possible inside the FPGA because such circuitry is only available at the chip periphery for the purpose of constructing the external buses. One can use the tristate coding style, which is however translated into gates and multiplexers because internally the tristate buffers are not available.

Due to lack of tristate elements, the on-chip processor buses cannot be bidirectional. They are rather unidirectional, either *in* or *out*. In the example below, the incoming DATA bus is named *inbus*. The outgoing DATA bus is named *outbus*. Both the ADDRESS bus and the strobes are outgoing. The processor is driving the outgoing buses, which means it is a *bus master.* It is a very simple design.

In case of the RISC5 core the design was made slightly more complex because of the memory mapped video *framebuffer* implemented in the external RAM. The video controller was made a secondary master which would occasionally steal the bus access from the main CPU to refresh its internal video pixel buffer. This technique is named *direct memory access* (DMA). In order to make it simple, the secondary master did not ask the main CPU for permission to use the bus (which would have been named Bus Grant Request), but rather it used a special *stallX* signal to temporarily suspend the CPU.

```
// RISC5 soft processor core by Niklaus Wirth  31.8.2018
// with interrupt and floating-point
// https://people.inf.ethz.ch/wirth/ProjectOberon/index.html

module RISC5(
input clk, rst, irq, stallX,   // in:  clock, reset, interrupt, stall the CPU
input  [31:0] inbus,           // in:  data
input  [31:0] codebus,         // in:  program instruction memory
output [23:0] adr,             // in:  address in external RAM
output [31:0] outbus           // out: data
output rd, wr, ben,            // out: memory strobes: read, write, byte enable
);
```

I added comments to the original Verilog module declaration. The interface is similar to the previous Blackfin interface, but it has three unidirectional data buses The *inbus* and the *outbus* are both the DATA buses, *in* or *out* the CPU. The third DATA bus is named *codebus*, which is a syntactic leftover from the original Harvard architecture of this soft core. (Harvard architecture uses separate memories for instruction and data.) Outside the core the *codebus* is multiplexed among the on-chip BRAM containing a bootloader, and the off-chip asynchronous RAM. The RAM pins, which are bidirectional, are multiplexed between the to *outbus* in the outgoing direction, and *inbus* in the incoming direction. The two buses, which are unidirectional on chip, are merged into a single bidirectional bus off chip.

13 Using a processor's bus address

The FPGA is a big device with many resources that can be addressed individually by the CPU. Inside the FPGA the resources are connected to an *address decoder*, which is yet another pattern of the VHDL code. There are several recommended ways of writing the decoder in VHDL. First we will show the code snipped from the ISE Language Templates, and then we will show another equally valid way of coding the same.

```
1. --- Address decoder
2. --- Code pattern from ISE Language Templates -> Synthesis Constructs
3.    process (clock) begin
4.         if ( clock'event and clock ='1') then
5.             if ( reset = '1') then
6.                   output <= "0000";
7.             else
8.                case input is    -- “input” is the address
9.                     when "00"   => output <= "0001";
10.                    when "01"   => output <= "0010";
11.                    when "10"   => output <= "0100";
12.                    when "11"   => output <= "1000";
13.                    when others => output <= "0000";
14.               end case;
15.            end if;     -- reset
16.        end if;         -- clock
17.     end process;
```

First of all we see that VHDL is case insensitive. Second, here we see yet another incarnation of a “process”. Third, Xilinx chose not to use the function `rising_edge(clock)`, which we find more

descriptive than `clock'event and clock ='1'`. The two idioms are exactly equivalent. Either way the VHDL compiler will infer `output` as a register that is synchronized to the clock.

The pattern of a “walking 1” in lines 7 through 10 implements a decoder. Every binary representation of the `input` address gets translated to a “1” in the corresponding position. The `output` bits should be connected to circuit-select CS inputs of the register components shown earlier. Only single register will drive the bus because only a single “1” is active at any given time.

We will now demonstrate another way of writing the address decoder that looks remarkably different. We use this form in our own VHDL code.

```
1.       RawSelect <=
2.       "0001" WHEN ADDR (25 downto 2)="00000000000000000000001" ELSE
3.       "0010" WHEN ADDR (25 downto 2)="00000000000000000000010" ELSE
4.       "0100" WHEN ADDR (25 downto 2)="00000000000000000000011" ELSE
5.       "1000" WHEN ADDR (25 downto 2)="00000000000000000000100" ELSE
6.       (others =>'0');
7.       RegSelect <= RawSelect WHEN (BF_AMS0 = LOW) AND (BF_AMS1 = HIGH)
8.                    ELSE (others => '0');
```

Unlike the previous decoder, the resultant circuit will be combinatorial rather than registered. It will be susceptible to glitches. It is virtually guaranteed that the glitches will occur in `RawSelect` because all 23 address bits cannot be perfectly synchronized. The designer must somehow know that glitches will not be a problem before using this circuit. How can we know that this solution is safe to use?

If our CPU is Blackfin from Analog Devices, then the answer can be found in the Blackfin data sheet. It specifies the timing between the address bits and the read/write strobes implementing the industry-standard memory bus. The WR and RD strobes that are connected to the registers (see previous code) become active many nanoseconds after the transitions of the address bits. Even though the `RawSelect` may not be valid during the address transition, this period of invalidity happens outside the active time window that is defined by the memory strobes. One should note that the strobes originate from a thoroughly tested microprocessor chip and therefore they are safe to use. The conclusion from this example is that the validity of a circuit cannot be established without knowing the context in which this circuit will operate. In principle, glitches are a bad thing. However, their presence may be rendered completely irrelevant by the context of the application.

There are a few other important points to note in this example. First, we are using the address bits of the Blackfin processor BF561. The designer needs to study the processor's data sheet and its Hardware Reference Manual (HRM) before implementing the address decoder. The processor must operate in the proper memory mode (32-bit mode in this case) because we are decoding its address bits downto 2. The two least significant bits (LSB) are not decoded because they are not driven in the 32-bit mode. These details are explained in the HRM.

Another detail concerns the asynchronous memory space (AMS) strobes. There are four such signals in case of the BF561, AMS0 through AMS3. Each one is addressing a separate 64-megabyte range of memory. Two of these are used to address the Ethernet and the USB chips outside the FPGA. The remaining two, the AMS0 and AMS1, are routed to the FPGA. These two signals are used in our decoder to *qualify* the address bits. According to the code from the previous page, the signal `RegSelect` is active (i.e., non-zero) when the AMS1 is active. This strobe is directed to the AMS1

address range that can be found in the processor's data sheet. Utilizing the AMSx strobes is mandatory because the same ADDR bit combination will occur in four cases, once per AMS memory space. If we neglected to use the AMS strobes, the FPGA could respond to the memory request directed to the Ethernet chip, causing an immediate system crash.

One has to combine all the above information in order to calculate the memory addresses issued by the processor. The following details have to be kept in mind.

1. Two LSBs 0 and 1 are not used by the decoder because the processor is working in the 32-bit mode. Consequently, one should extend the bit patterns to the right by two bits “00” before calculating the addresses to be used in the C header files.
2. The AMSx spaces start at fixed “base addresses” specified in the data sheet. The AMS1 base address has to be added to the bit pattern of the lower 25 address bits (23 bits from our example, extended with two LSBs as explained above).
3. All the memory strobes originally issued by the processor are active-low. The strobes are converted to active-high inside the FPGA. We use the active-high convention in the FPGA designs because several library components expect such active-high signals.

We will now provide the resulting addresses that the Blackfin processor will issue to access the FPGA registers connected to the address decoder. The reader should double check that his/her address calculations yield the same results.

```
0x2400 0000  unused
0x2400 0004  1st register
0x2400 0008  2nd register
0x2400 000c  3rd register
0x2400 0010  4th register
```

14 Making the components visible throughout the project

The two types of registers shown earlier are examples of components that you can use throughout your projects. There are three important things to keep in mind.

1. The component can exist in more than one copy in direct analogy with hardware components.
2. Each time you instantiate the component, you are creating a separate hardware copy of it. Components are built from hardware elements such as gates, flip-flops, etc. These elements are taken from the enormous pool provided by the FPGA fabric and assembled into a local instance of the component that you are instantiating.
3. Unlike electronics bought in a store, the VHDL components do not have to be identical. If the component was parametrized with GENERIC statements, then you can change the parameters each time the component is instantiated. In such a way you can create several registers of different bit widths, for instance.

We will now discuss how to make the best use of the VHDL component files that we presented earlier. First of all, the component instance must match the PORT declaration in the component source file. In principle, it should be sufficient to declare the PORT twice: once in the prototype file, and the second time when instantiating the component. However, the VHDL committee decided otherwise. They

decided that the PORT declaration has to be repeated one more time before the component is instantiated. It is a bit annoying. This (mis)feature of the VHDL can be worked around as follows. We collect these redundant component declarations into a PACKAGE that is used in all our files. The package makes the component declarations visible throughout the entire project.

```
-- BlackVME_types.vhd.  Package collects all our declarations.
-- (C) Wojtek Skulski 2011-2012.
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
PACKAGE BlackVME_types IS

COMPONENT CTRL_reg_CPU_writes                          -- component #1
       GENERIC (regWdt : INTEGER := 16);
       PORT (
       CLK    : in STD_LOGIC;
       CS     : in STD_LOGIC;
       WR     : in STD_LOGIC;
       RD     : in STD_LOGIC;
       IOBUS  : inout      STD_LOGIC_VECTOR    (regWdt-1 downto 0);
       regout : out        STD_LOGIC_VECTOR    (regWdt-1 downto 0)); -- output
END COMPONENT;

COMPONENT CTRL_reg_CPU_reads                           -- component #2
       GENERIC (regWdt : INTEGER := 16);
       PORT (
       CLK    : in STD_LOGIC;
       CS     : in STD_LOGIC;
       RD     : in STD_LOGIC;
       IOBUS  : inout      STD_LOGIC_VECTOR    (regWdt-1 downto 0);
       fabric : in         STD_LOGIC_VECTOR    (regWdt-1 downto 0)); -- input
END COMPONENT;
END BlackVME_types;                                    -- end of the file
```

The PACKAGE declaration is similar to the C language header files that establish “prototypes” of C functions. Just like the header files, VHDL packages are used throughout the project. The VHDL file that is going to make use of the components will start as follows.

```
1. library IEEE;
2. use IEEE.STD_LOGIC_1164.ALL;  -- basic VHDL
3. -- Our own types, constants, and components in file BlackVME_types.vhd
4. use work.BlackVME_types.all;
```

The package declarations are compiled into a standard workhorse library named `work` during project compilation. The library `work` is automatically included by all files in the project. It does not need to be mentioned with the `library` clause. It is sufficient to `use` our package in line 4. From then on all the components collected in the package are known to the project.

15 Making use of the register component

Each time you instantiate the component, you are creating a separate hardware copy of it. Just like any other electronic hardware, the component instance must be connected to some physical wires. The

wires are called “signals” in VHDL. The following code snippet is instantiating a register that allows the Blackfin processor to toggle several control bits inside the FPGA. This snippet should be put in the architecture implementation of the main project file.

```
1. -- Control register CPU --> FPGA
2.    BFcontrol: CTRL_reg_CPU_writes  -- CPU control over FPGA
3.       GENERIC MAP (regWdt => 16)
4.       PORT MAP (
5.             CLK    => BF_CLK,
6.             CS     => RegSelect (0),
7.             WR     => BF_WRENA1,
8.             RD     => BF_rena1,
9.             IOBUS  => BF_DATA      (15 downto 0),
10.            regout => BF_ctrl_Reg  (15 downto 0));
```

Apparently the VHDL committee was very fond of the arrows => and <=. In this code => means “connect”. The inputs of the register are connected to the Blackfin memory signals. The chip select CS is tied to the address decoder shown earlier. The output is wired to the control bits `BF_ctrl_Reg`. These bits turn ON and OFF certain firmware features. For completeness we show how the Blackfin memory strobes are conditioned before being wired to the register. The signals marked with “_b” are active-low signals from the processor. They are converted to active-high signals before being used in the FPGA.

```
1.     -- Convert active-low signals from Blackfin to active-high
2.     BF_AMS1       <= NOT BF_AMS1_b; -- AMS1 memory space
3.     BF_WR         <= NOT BF_AWE_b;
4.     BF_RD         <= NOT BF_ARE_b;
5.     BF_rena1      <= BF_AMS1 AND BF_RD AND (not BF_WR);
6.     BF_WRENA1     <= BF_AMS1 AND BF_WR AND (not BF_RD);
```

16 Utilizing the dual-port memory blocks

The on-board Spartan-6 FPGA contains 603 kilobytes of dual-port block memory (BRAM) partitioned into as many as 268 memory blocks, 18 kilobits each. Making effective use of the BRAM is tremendously important in many applications. There are two ways of achieving this goal.

1. The blocks can be instantiated “by hand” in the way described in the previous section. One needs to use the library named UNISIM that is supplied by Xilinx. Memory components are declared in that library. Using the memory components is described in Spartan-6 Libraries Guide available from ISE Help. One should read the pages titled BRAM in Chapter 2.
2. The BRAM can be configured with Core Generator that is a part of Xilinx ISE. Core Generator will customize many options that are available in BRAM components, as well as provide a netlist file that tiles several BRAM's together into a composite block of the size and aspect ratio that is needed.

We recommend using the Core Generator because manual instantiation is tedious. Extensive help is available from within the Core Generator. The Generator will provide a set of files as well as a comprehensive Data Sheet that explains all the relevant details. We will not attempt to duplicate this information in this manual. We only point out that the Core Generator will wrap the BRAM blocks into

a component whose declaration should be placed into the package BlackVME_types as follows.

```
-- EXAMPLE: Dual port "simple memory" generated with Core Generator
-- Port A (BF write)   N/2  cells each 32 bit wide
-- Port B (FPGA read)  N    cells each 16 bit wide
-- This component is in three files named "EXAMPLE_BRAM_8k"
-- .ngc, .xco, .vho. The .vho contains instructions.
-- FPGA side: 8k 16-bit samples.  BF side: 4k 32-bite words.
      -- BF:   Port A addr (11 : 0), but BF uses address (13 : 2)
      -- FPGA: Port B addr (12 : 0)
COMPONENT EXAMPLE_BRAM_8k
      PORT (
      -- port A; BF side
      clka   : IN std_logic;
      ena    : IN std_logic;                      -- read enable
      wea    : IN  std_logic_VECTOR(0 downto 0);  -- write enable
      addra  : IN  std_logic_VECTOR(11 downto 0); -- (13 : 2) = 12 bits
      dina   : IN  std_logic_VECTOR(31 downto 0); -- Blackfin writes directly to this
      douta  : OUT std_logic_VECTOR(31 downto 0); -- connect this to a multiplexer
      -- port B; FPGA side
      clkb   : IN std_logic;
      enb    : IN std_logic;                     -- read enable
      web    : IN std_logic_VECTOR(0 downto 0);  -- write enable
      addrb  : IN std_logic_VECTOR(12 downto 0); -- (12 : 0) = 13 bits
      dinb   : IN std_logic_VECTOR(15 downto 0);
      doutb  : OUT std_logic_VECTOR(15 downto 0));
END COMPONENT;
```

This particular component has been tailored to our needs indicated with comments. We will refrain from explaining the details because your requirements may be different. The memory component named EXAMPLE_BRAM_8k can be instantiated in the main VHDL file according to the rules explained in the earlier sections.

17 Addressing blocks of on-chip memory

The dual-port BRAM is addressed from two sides. The FPGA side can use a simple counter to step through the memory range. The counter wraps around when all its bits are '1'. Waveform acquisition can be restarted from the start of the memory block. Collecting the samples is performed when enabled, otherwise the memory counter is not running. Lines 1 through 4 should be put in the declaration section. The rest of the code should be put into the architecture implementation.

```
1.  library IEEE;
2.  use IEEE.STD_LOGIC_1164.ALL;      -- basic VHDL
3.  use IEEE.STD_LOGIC_ARITH.ALL;     -- basic arithmetics on std vectors
4.  SIGNAL MemAddr_ctr: STD_LOGIC_VECTOR (Nbit-1 downto 0); -- declaration
5.
6.  PROCESS (CLK, restart, enable) BEGIN
7.       IF rising_edge (CLK) THEN
8.          IF restart = '1' THEN
9.             MemAddr_ctr <= (others => '0');
10.         ELSIF enable = '1' THEN
11.            MemAddr_ctr <= MemAddr_ctr + 1; -- this counter wraps around
```

```
12.         END IF;
13.      END IF; -- CLK
14.  END PROCESS;
```

The Blackfin side is a bit more complicated. The Blackfin address is split into two parts. The lower-order bits (13 downto 2) are tied to the BRAM address. These bits do not include the two lowest order bits 0 and 1 because the Blackfin address jumps by four, when the processor is working in the 32-bit mode. The upper part (25 downto 15) of the Blackfin address is used to select among multiple BRAM blocks, using a variant of the address decoder shown below.

```
-- At BF side, 4k-word needs 12-bit address, 13:2. The lowest select bit is 14.
MemSelect  <=                                                -- result addr
"0001" WHEN BF_AMS0& BF_ADDR(25 downto 14)="1000000000000" ELSE -- 0x20000000
"0010" WHEN BF_AMS0& BF_ADDR(25 downto 14)="1000001000000" ELSE -- 0x20100000
"0100" WHEN BF_AMS0& BF_ADDR(25 downto 14)="1000010000000" ELSE -- 0x20200000
"1000" WHEN BF_AMS0& BF_ADDR(25 downto 14)="1000011000000" ELSE -- 0x20300000
(others=>'0');
```

We used a notation trick in the above example. The AMS0 select was prepended to the address bits using the VHDL operator `&` that concatenates bits together. The leftmost “1” in the address bit patterns corresponds to the AMS0 bit. In such a way we did not have to write a separate line of the VHDL code that would AND the select vector with the AMS0 bit. The ANDing was implicitly coded into the address bit pattern used in the decoder. This kind of a notation shortcut is very handy, but it may be a bit confusing at the first sight. Nevertheless, it belongs to the war chest of VHDL programming.

The MemSelect bit is then tied to the respective “enable” bit of the Blackfin side of the BRAM. This bit will place the BRAM at the address range indicated in the comment. For example, if we use the 0-th bit, the memory will be mapped starting at address `0x20000000` in hex. The reader should verify that the Blackfin addresses are indeed as indicated in the comments. Please read in the BF561 data sheet what addresses correspond to which AMSx space when you perform this exercise.

18 Multiplexing BRAM memories for CPU readout

The outputs of the dual-port BRAM are not equipped with tristate buffers, which are avoided on-chip. As a consequence, the BRAM outputs cannot be directly wired to the readout bus because they would create electrical contention. Previously we did not encounter the contention because we declared the register outputs using the tri-state notation 'Z', which was implemented with multiplexers behind the scenes. Now we have to deal with the problem one more time.

Lacking the on-chip high-impedance 'Z' buffers, the BRAM outputs have to be multiplexed. We will compare two different ways to infer the multiplexer. The first example is provided by the ISE Synthesis Constructs examples. It has been edited for clarity.

```
   --- Multiplexer
   --- Code pattern from ISE Language Templates -> Synthesis Constructs
1. process (select,input1,input2,input3,input4)
2. begin
3.    case select is
```

```
4.       when "00"   => output <= input1;
5.       when "01"   => output <= input2;
6.       when "10"   => output <= input3;
7.       when "11"   => output <= input4;
8.       when others => output <= input1;    -- looks redundant?
9.    end case;
10. end process;
```

The multiplexer circuit was coded as a `process` that does not reference the clock. Consequently, the combinatorial circuitry will be inferred by the compiler. Here we see that the `process` does not necessarily imply that the circuit is clocked. If there is no reference to the clock, then the circuit will be combinatorial. (As a reminder, the clock is implied by a `rising_edge`, `falling_edge`, or a construct `signal'event and signal='1'`.) Line 8 covers non-boolean signal values such as 'X' or 'Z'.

We now present another version of a multiplexer that makes the reference to the Blackfin AMS space.

```
1.   TYPE BF_MemBus_t IS ARRAY (3 downto 0) OF STD_LOGIC_VECTOR (31 DOWNTO 0);
2.
3.   -- array of BF memory outputs, 32 bits each.
4.   SIGNAL BF_MemBus : BF_MemBus_t;
5.
6.   MemorySpace0  <=
7.       BF_MemBus  (0) WHEN select="00" ELSE
8.       BF_MemBus  (1) WHEN select="01" ELSE
9.       BF_MemBus  (2) WHEN select="10" ELSE
10.      BF_MemBus  (3) WHEN select="11" ELSE
11.      (others => 'Z');
12.
13.  BF_DATA <= MemorySpace0 WHEN BF_rena0 = '1' ELSE (others=>'Z');
```

The type declaration from line 1 should be put into the central repository *BlackVME_types*. The output `MemorySpace0` is switched among bit vectors `BF_MemBus` according to the selection bit pattern. In line 13 the output from the multiplexer is wired to the Blackfin data bus through a tristate buffer controlled by the read strobe. One should note that in order to complete the circuit the individual MemBus vectors are connected to the ports `douta` of the BRAM memories shown earlier. The other ports `dina` can be wired directly to the Blackfin BF_DATA because being inputs they cannot cause a contention.
All the tri-state 'Z' constructs are implemented with multiplexers behind the scenes. It is good that Xilinx compiler is kind enough to let us use the 'Z' notation which is very convenient.

19 Synchronizing an external signal to the FPGA clock

We are now turning our attention to bit processing. We start with a very simple example. Let us make sure that an external signal starts on an FPGA clock boundary. This example can be applied to the front panel NIM inputs. But first let's ask why do we care?

The answer is that an unsynchronized signal is potentially a very bad thing. Let's imagine that we use an external NIM pulse to zero a time stamping register. OK, so we connect it to a `reset` of a free-running counter similar to the one shown in Section 13. We will find out that we indeed reset the counter most of the time. But sometimes we do not. Sometimes the counter starts from a seemingly random bit pattern. What is going on?

The answer is “metastability”. Type this word to Google and several articles will pop up, documenting the importance of the problem. The metastability occurs when the signal edge hits the flip-flop at *just the wrong moment*, violating the timing requirements. The *just wrong* is guaranteed to happen if we repeatedly use a signal whose edge is unrelated to the clock. So we need to make sure that before using the external NIM pulse inside the FPGA, we force its edge to be synchronized to the FPGA clock.

```
1.  -- ISE VHDL templates --> Synthesis Constructs --> Misc --> Debounce
2.  --**Insert the following between the 'architecture' and 'begin' keywords**
3.  signal Q1, Q2, Q3 : std_logic;
4.
5.  --**Insert the following after the 'begin' keyword**
6.  process(CLK) begin
7.       if rising_edge(CLK) then
8.           Q1 <= external;     -- time = 1. Can be metastable.
9.           Q2 <= Q1;           -- time = 2. Good to use.
10.          Q3 <= Q2;           -- time = 3. Perfect to use.
11.      end if;
12.    end process;
```

The code describes a clocked circuit built with flip-flops. That's what we need. The three signals Q1, Q2, and Q3 are copies of the `external`, but delayed by one, two, and three clock cycles. Strictly speaking, Q1 is delayed by anything between zero and the full clock because the `external` can hit the flip-flop any time within the clock cycle. The Q2 and Q3 are in the fixed time relationship to Q1. The Q2 should be good enough. The Q3 is guaranteed to be perfect for the age of the Universe.

There can be two questions concerning this code snippet. (a) How does it follow that Q1 is a delayed copy of `external`, and so on with Q2 and Q3? (b) How does it work anyway?

The answers: (a) Don't even ask. This kind of code is an idiom. If there is a rising edge, then there are flip-flops. If there is a string of assignments, then flip-flops are strung one after another. That's what we need. (b) The circuit works through black magic based on probabilities. The topic is not an easy one.

20 Delaying a signal by a fixed number of clock cycles

The same circuit can be used to delay any bit by a fixed number of clocks, three in this case. Just replace the `external` with `any_signal` internal to the FPGA. The `any_signal` must belong to the same clock domain. If it does not then we are in square one. A signal belonging to another clock domain is an external signal from the point of view of the target domain.

21 Generating a short pulse lasting one clock cycle

The same circuit can be used to manufacture a pulse lasting one clock cycle. Such a pulse can be useful because it corresponds to a leading edge. In other words, you have a long pulse of unknown duration, perhaps milliseconds. But you need a short pulse to reset your counter and then start counting right away without waiting till the long pulse goes away. Here is the solution. Apply the following to the Q signals from the previous section on synchronizing.

```
pulse1 <= Q1 and (not Q2);    -- use with internal signal
pulse2 <= Q2 and (not Q3);    -- use with external signal
```

One of these lines should be applied to the Q signals from the previous circuit. Version 1 produces the pulse one clock earlier than version 2, but it is susceptible to metastability. Version 2 should be used with external signals. There are some caveats worth putting in writing.

1. Somewhat surprisingly, this circuit seems to annihilate a single-clock pulse applied to its input. One can say that applying the leading-edge detection twice makes no sense, so maybe there is no problem. Nevertheless, annihilating single-clock inputs was not expected. Beware.

2. A similar circuit described in the book by Chu on pages 114, 117 did not work well. Most often it produced double-clock output, but sometimes it was a single clock. The behavior was bizarre. We recommend our solution described in this section because it has been tested in Spartan-6.

22 Delaying a one-bit signal by a variable number of clock cycles

Now we are going to do some heavy lifting. We want to delay a signal by a variable number of clock cycles, from 1 to 32 clocks. The signal will be one bit wide (for example, the NIM input after it was synchronized). We want that Blackfin controls the delay.

```
1.   --**Insert the following at the beginning of the file
2.   library unisim;  -- Xilinx library components
3.   use unisim.vcomponents.all;
4.
5.   --**Insert the following between the 'architecture' and 'begin'
6.   signal prgm : STD_LOGIC_VECTOR (4 downto 0);
7.
8.   --**Insert the following after the 'begin' keyword**
9.   shift_register_32_taps : SRLC32E
10.      generic map (INIT => X"00000000")
11.      port map (
12.      CLK => CLK,     -- Clock input
13.      D   => din,     -- shift register bit input
14.      Q31 => open,    -- SRL cascade output (connect only to next SRL)
15.      A   => prgm,    -- 5-bit shift depth static select input
16.      CE  => '1',     -- Clock enable input, always enabled
17.      Q   => dout     -- output bit from shift register
18.      );
```

We used a shift register SRL32 that Xilinx built into Spartan-6. The number of delay “taps” can be controlled with a register of the type `CTRL_reg_CPU_writes` that was described earlier. The register should be instantiated with 5 bits, because SRL32 needs this number of control bits. The register will be connected to the bit vector `prgm`, which in turn will be wired to the shift register control. The register will be memory-mapped on the Blackfin bus in the way elaborated earlier. The chain of connections is sketched below. Writing down the VHDL statements is left as an exercise for the reader.

```
Blackfin bus →  CTRL_reg_CPU_writes →  prgm →  shift_register_32_taps
```

In this way we demonstrated how the register component described earlier can be utilized to implement functionality that is both needed and not easy to implement without a war chest of proven components.

23 Delaying a multi-bit signal by a variable number of clock cycles

Now we want to delay a multi-bit signal by a variable number of clock cycles, from 1 to 32 clocks. The signal will be of any width. For example, we can delay a stream of ADC samples in order to match a time-of-flight difference among detector subsystems. We want that Blackfin controls the delay.

```
-- Delay_Line_L32.vhd. (C) Wojtek Skulski 2003-2012.
-- Input: prgm 0..31 means delay by 1..32 clocks
library IEEE;
use IEEE.std_logic_1164.all;
library unisim;                    -- Xilinx library components
use unisim.vcomponents.all;

ENTITY Delay_Line_L32 IS
  GENERIC (PipeWdt: INTEGER := 14);  -- default bit width of the stream
  PORT (
    CLK   : in  STD_LOGIC;
    CE    : in  STD_LOGIC;                      -- clock enable
    prgm  : in  STD_LOGIC_VECTOR (4 downto 0); -- delay 0..31, input
    din   : in  STD_LOGIC_VECTOR (PipeWdt-1 downto 0);
    dout  : out STD_LOGIC_VECTOR (PipeWdt-1 downto 0)
    );
end Delay_Line_L32;

ARCHITECTURE SRL32 OF Delay_Line_L32 IS BEGIN

pipe: FOR i IN 0 TO PipeWdt-1 GENERATE

slice : SRLC32E
GENERIC MAP (INIT => X"00000000")
PORT MAP (
      CLK => CLK,       -- Clock input
      D   => din(i),    -- SRL data input
      Q31 => open,      -- SRL cascade output pin (connect only to next SRL)
      A   => prgm,      -- 5-bit shift depth select input
      CE  => CE,        -- Clock enable can freeze the pipe
      Q   => dout(i));  -- SRL data output
END GENERATE pipe;
END SRL32;             -- end of the component file
```

The component should be declared in the central repository *BlackVME_types*. It can be instantiated in the design, connected to the register `CTRL_reg_CPU_writes`, and controlled by the Blackfin. One can change on the fly by how many clock ticks the data stream will be delayed, from 1 to 32 clock ticks. The chain of connections is similar to the previous example.

```
Blackfin bus →  CTRL_reg_CPU_writes →  prgm → Delay_Line_L32
```

The clock enable CE is a handy method of temporarily freezing the shift registers in their current state. It can be used to capture a stretch of the data stream in the pipe. The captured samples can be then clocked one by one from the pipe by pulsing the CE. Each time the CE is pulsed HIGH for the duration of one clock, one sample will be clocked out from the end of the pipe, and one sample will be clocked

into it at the beginning. (Use the previous one-shot to make single-clock CE pulses.) One should note that the pipe cannot be accessed in parallel. The only method of retrieving the samples is to clock them out from the end of the pipe.

We will now present a code snippet that implements a similar functionality, but this time the compiler is allowed to infer the circuit.

```
1.   -- Language templates --> Synthesis constructs --> Shift registers.
2.
3.   TYPE Tpipe IS ARRAY(PipeLen-1 downto 0) OF
4.       STD_LOGIC_VECTOR (PipeWdt-1 downto 0);
5.   SIGNAL pipe: Tpipe;
6.   -- The "dynamic shift register" should be inferred by synthesis.
7.    process (CLK) begin
8.       if rising_edge (CLK) then
9.          pipe <= pipe(PipeLen-2 DOWNTO 0) & din;
10.      end if;
11.   end process;
12.   dout <= pipe(conv_integer(prgm)); -- select the dynamic length
```

The difference between this snippet and the previous one is three fold. First, there is no clock enable CE in the snippet. Adding the CE is left as an exercise. Second, the length of the pipe is not declared explicitly. The code is parametrized with the generic named `PipeLen` that can be any number. The compiler can infer pipes of any length and implement them any way it chooses. Most likely, the compiler will string together as many SRL16 or SRL32 blocks as needed and it will connect the `prgm` bits to implement the requested functionality. It sounds more convenient than the component shown earlier where the maximum delay was imposed by using the library component SRL32 whose maximum capacity is rigidly defined by its internal structure.

The last difference between the previous code and this one is that we can be reasonably certain what the previous code was doing by just looking at it and reading the SRL32 description in the Spartan-6 Libraries Guide. The margin for mistakes was reasonably narrow in the former case. In the latter case we think that we understand the function of the circuit based on the behavioral VHDL code. But there can be a difference between what we think and what the compiler is thinking, even though the code snippet was copied from ISE Help. It happens now and then that the code snippets do not yield the desired circuitry. The developer has to test the behavioral code snippets piece by piece.

24 Driving the clock off-chip

The next two sections are based on our experience with sending a data stream from the FPGA to a device that required double data rate (DDR). The outlined solutions are applicable to about 100 MHz, that is 100 megabits-per-second (Mbps) per IO pin for single data rate (SDR) and 200 Mbps for DDR.

DDR transmission is a common technique where data bits are latched on both rising and falling clock edges. It is commonly used together with source-synchronous transmission (SST) where both the data and the associated clock are sent together from the source to the destination along a multiwire cable or a bunch of printed circuit board traces with closely matched lengths. The idea is that as the clock and

the data travel together, their phase relationship is not distorted and the clock can be used to latch the data at the destination. The clock that is transmitted to another device is called a *forwarded clock*.

One should note that the received clock is defining its own clock domain at the destination, even in those cases where it toggles at precisely the same frequency as some other clock in the destination chip. The captured bits must be transferred to the other clock domains with proper techniques described in Xilinx application notes. The reader should search the following web page for the word *capture* or a phrase *data capture*. The relevant information can be found in XAPP225, XAPP709, XAPP802, XAPP855, XAPP860, and others. The application notes can be downloaded from the following page:

http://www.xilinx.com/support/documentation/application_notes.htm

The referenced information is crucial in order to design reliable data links between two FPGAs. The problem is at the receiving end where the received clock has no fixed relationship to the on-chip clock, even if both toggle at the same frequency. The transmitting end has no such problem because the data and the forwarded clock are in phase with the on-chip clock, which is in fact identical with the forwarded clock.

The above considerations will be important if the reader is planning to send the data from one FPGA to another one at a high data rate, or along a cable of an appreciable length. The problem becomes less severe in case of the on-board FIFO connection between the Spartan-6 FPGA and the Blackfin processor, especially when the FPGA is the transmitter and Blackfin is the receiver of the FIFO data. The Blackfin end of the FIFO is named a *Parallel Peripheral Interface* (PPI). Its timing is described on page 28 of the BF561 Data Sheet revision D. There are three important observations concerning the Blackfin PPI.

1. The PPI uses a single data rate (SDR) rather than DDR. The data bits are latched on the rising edge of the forwarded clock. The falling edge is not utilized.
2. The PPI clock is an input-only pin at the Blackfin end. This is fine when Blackfin is receiving the data. It may lead to some timing complications if Blackfin is sending the data, because the data bits and the clock will then travel in the opposite directions.
3. The bit capture was implemented by the designers of Blackfin silicon. We do not need to bother how the bits are captured at the Blackfin end.
4. The PPI is running at ½ of the Blackfin peripheral clock that is f_{sclk} = 120 MHz on the BlackVME board. It means that PPI is running at 60 MHz, that is a period of 16.7 ns, and half period of 8.3 ns. Establishing a proper timing margin within such a wide window should not be too problematic. We only need to supply the bits with proper bit-to-clock alignment described on page 28 of the BF561 Data Sheet.

Now we will tackle the first part of the topic. We want to drive the clock off-chip in order to generate the forwarded clock either for the Blackfin, or for some other destination chip. Lets try this:

```
output_pin <= CLK;
```

The assignment is perfectly legal and it worked in our previous design based on Spartan3A-DSP. Unfortunately, upon seeing this assignment the ISE compiler violently complained about illegal clock forwarding techniques in Spartan-6. We were advised to read a paragraph from the Synthesis Constructs → Coding Examples → Misc → Output Clock Forwarding Using DDR → Info (Clock Forwarding). A part of the paragraph is reproduced below.

```
The basic technique is to supply the input clock to an output DDR register where one
value is tied to a logic 0 and the other is tied to a logic 1.  A clock can be made with
the same phase relationship (plus the added offset delay) or 180 degrees out of phase by
changing the 1 and 0 values to the inputs to the DDR register.
```

An example code for Spartan-6 was also supplied in the help. We worked out the example as follows.

```
-- In Spartan6 we cannot simply drive the IOB with a clock signal.
-- We rather have to use "clock forwarding techniques" with ODDR2 primitive.
-- The register output can be routed only to ILOGIC, IODELAY, or IOB.
 ODDR2_clock : ODDR2
 GENERIC MAP(
   DDR_ALIGNMENT => "NONE",   -- Output alignment to "NONE", "C0", "C1"
   INIT => '0',               -- Initial state of the Q output to '0' or '1'
   SRTYPE => "ASYNC")         -- Specifies "SYNC" or "ASYNC" set/reset
   port map (
      Q  => CLK_RAW,    -- output data (clock in this case) to output pad
      C0 => CLK180,     -- clock0 input to be forwarded
      C1 => CLK,        -- clock1 input inverted
      CE => HIGH,       -- clock enable input
      D0 => HIGH,       -- data input associated with Clock0
      D1 => LOW,        -- data input associated with Clock1
      R  => LOW,        -- reset input
      S  => LOW);       -- set input
```

According to the quoted paragraph, plus other recommended reading (Libraries Guide, the page about ODDR2) the two signals CLK and CLK180 are the two versions of the same clock out of phase by 180 degrees. We generated the inverted clock with brute force:

```
CLK180 <= NOT CLK;
```

This assignment was expected to bring another wave of complaints, but surprisingly there were none. Should the compiler have complained, we would have used the Digital Clock Manager (DCM) to create the inverted clock.

So how does the circuit work in the first place? The idea is simple: there are two clocks C0 and C1. One is an inverted copy of the other. Both clocks are using their *leading edges* to send out two constant signals, one HIGH and one LOW. The constants are connected to D0 and D1, which are associated with the two clocks. The D0 is wired to clock C0, the D1 is wired to clock C1. On each leading edge the associated signal is sent out. That's how the DDR register is doing its job. (Read the Libraries Guide.)

The remaining issue is how to forward an SDR clock if one is planning to drive the Blackfin's PPI? Actually, there is no such thing as the SDR clock. Only the data can be either DDR or SDR in the relation to the clock. Clock forwarding is handled the same way in both the SDR and DDR cases.

In our application we faced an additional difficulty, because the receiver expected the clock edge to fall in the middle of the bit “data eye”. It means that the clock and the data should be 90º out of phase, while both were precisely in phase on-chip. We had to shift the output phase of the forwarded clock to give it just the right timing margin relative to the data bits. (If you are planning to drive the Blackfin PPI then look at page 28 of BF561 Data Sheet.) We delayed the clock with the IODELAY2 that is built into every Spartan-6 pin. The relevant code example can be found onscreen in Device Primitive Instantiation → Spartan-6 → I/O Components → Input. For explanations please read about

IODELAY2 in the Libraries Guide. We also note that the “7” family of Xilinx devices disposed of the output delay elements. Only the input delays remain in Artix-7 or Kintex-7. Every new generation of Xilinx devices makes our life even more interesting than it used to be under the previous generation.

```
-- It is not clear from documentation what a tap delay really is.
-- Use  the scope to examine bit-to-clock timing.
 IODELAY2_clock : IODELAY2
 GENERIC MAP (
      COUNTER_WRAPAROUND=> "WRAPAROUND", -- STAY_AT_LIMIT or WRAPAROUND
      DATA_RATE         => "SDR",        -- SDR or DDR
      DELAY_SRC         => "ODATAIN",    -- IO, ODATAIN or IDATAIN
      IDELAY_MODE       => "NORMAL",     -- Unsupported (NORMAL or PCI)
      IDELAY_TYPE       => "FIXED",      -- FIXED, DEFAULT, VARIABLE_FROM_ZERO,...
      IDELAY_VALUE      => 0,            -- Input Delay (0-255)
      IDELAY2_VALUE     => 0,            -- Input Delay (0-255); only for PCI
      ODELAY_VALUE      => XXX,          -- Output delay (0-255). Use proper value
      SERDES_MODE       => "NONE",       -- NONE, MASTER or SLAVE
      SIM_TAPDELAY_VALUE => 45)          -- Used for simulation in ps
 PORT MAP ( -- all ports are 1-bit. FIXED mode does not use RST or clocks.
      ODATAIN     => CLK_RAW,       -- in;  Data input from OLOGIC or OSERDES.
      IDATAIN     => LOW,           -- in;  Data input from IOB
      T           => LOW,           -- in;  Tristate input. LOW=output, HIGH=input.
      CLK         => LOW,           -- in;  Clock input from the fabric
      DATAOUT     => open,          -- out; Delayed output to ISERDES/Input reg
      DATAOUT2    => open,          -- out; Delayed output to general FPGA fabric
      DOUT        => FORWARDED_CLK, -- out; Delayed Data Output to output pin
      TOUT        => open,          -- out; Delayed Tristate Out
      IOCLK0      => LOW,           -- in;  Primary   I/O Clock input
      IOCLK1      => LOW,           -- in;  Secondary I/O Clock input
      RST         => LOW,           -- in;  Reset to zero or 1/2 of total period
      CE          => LOW,           -- in;  Enable increment/decrement
      INC         => LOW,           -- in;  Increment / Decrement input
      CAL         => LOW,           -- in;  Initiate calibration input
      BUSY        => open           -- out; Busy after calibration CAL
      );                            -- end of IODELAY2
      OUT_PIN <=  FORWARDED_CLK;    -- OUT_PIN is routed to the receiver
```

The IODELAY2 is inserted between the ODDR2 from the previous page and the output pin. In our case we wanted a *fixed* output delay for the signal. The delay is thus FIXED, what automatically renders most options irrelevant. The number of *delay taps* needs to be set to **XXX** after you look at the scope. The not-so-funny feature of Spartan-6 is that the value of the delay tap is not guaranteed by Xilinx, and therefore one has to use the scope to measure the actual delay. Try 50 picoseconds per tap for a good start and use the scope to make sure. It will delay the clock signal by XXX*50 ps, if the delay tap is 50 picoseconds. But this value is only the first guess. You will need to use the scope.

The very fact of inserting the `IODELAY2` is adding about 2.5 ns to the signal path (check Spartan-6 Data Sheet for exact value). It means that a rough alignment of the clock to the bit “data eye” may be achieved by just inserting the `IODELAY2` into the clock path without any further work, if the `IODELAY2` blocks are not inserted into the bit paths.

25 Driving the data bits off-chip

We will now discuss how to drive the data using the SDR or DDR. Concerning the Single Data Rate, there are two solutions. The first solution is simple. Just write

```
output_pin <= data_bit;
```

We recommend this solution because of its simplicity. Note that the IODELAY2 is omitted in the data path, and therefore one can easily create a 2.5 ns clock-to-bit offset by inserting the IODELAY2 into the clock path, while the data bits are routed directly to the output pins. Such an offset may be required by the receiver (check the data sheet).

Another solution is to use the `ODDR2` as explained in the previous section, but make the two DDR data streams identical. The falling edge data will be the same as the rising edge data, what effectively means there will be no bit transition on the falling clock edge. And this is precisely what SDR means.

```
-- Double data rate register used for SDR transmission.
ODDR2_bit : ODDR2
   GENERIC MAP(
   DDR_ALIGNMENT => "NONE",   -- Output alignment to "NONE", "C0", "C1"
   INIT => '0',               -- Initial state of the Q output to '0' or '1'
   SRTYPE => "ASYNC")         -- Specifies "SYNC" or "ASYNC" set/reset
   port map (
      Q  => output_signal,    -- connect to either output pad or to IODELAY2
      C0 => CLK180,           -- clock0 input associated with D0
      C1 => CLK,              -- clock1 input associated with D1
      D0 => same_data,        -- data input associated with Clock0
      D1 => same_data,        -- data input associated with Clock1
      CE => HIGH,             -- clock enable input
      R  => LOW,              -- reset input
      S  => LOW               -- set input
   );
```

Now we turn our attention to the DDR data stream. Actually, it is very simple. Replace the “same data” in the previous example with `data_1` and `data_2` and you have created a DDR data stream, where `data_1` is coded on one edge, and `data_2` on the opposite edge of the clock.

The solutions outlined in this section are good to about 100 MHz, that is 100 Mbps per IO pin for SDR and 200 Mbps for DDR. The data rate can be increased about 5x using the SERDES blocks that are built into every IO pin. The reader is advised to study Xilinx application notes concerning SERDES.

26 Conclusion and outlook

We walked the reader among several examples of VHDL programs relevant to the BlackVME projects. We are aware that the examples barely scratch the surface. There are whole areas not covered with our discussion, such as high-speed signal transmission over LVDS links, or digital signal processing. It was not possible to dive into these topics in an introductory tutorial whose scope and size has to stay limited. Nevertheless, we hope that even the limited material presented in this tutorial will be useful for jump starting your BlackVME work.

In this concluding section we want to collect some observations and recommendations not mentioned elsewhere. We suspect that the reader may feel a bit overwhelmed and intimidated after the visit to the whole new world of FPGA development. We want to attest that it is not as difficult as it might seem. There are a few principles to keep in mind.

The FPGA work has to be much more rigidly structured than it is customary with computer programming. Let's face the reality. In principle, computer programs should be neatly written and well documented. However, as most researchers probably noticed, it is rarely the case. The computer code is usually quickly cobbled together under the pressure of time. Any kind of crappy spaghetti code that is doing the work is acceptable in research-grade computer code. The kind of programming habits, that work for computers, unfortunately will spell disaster in FPGA programming.

The reason behind the difference between the computer and the FPGA programs is the way the FPGA compilers work. As we repeatedly stressed in the tutorial, FPGA compilers do not translate the programs into the FPGA code line-by-line. The compilers rather look for *patterns* and apply a process called *inferring*. The programmer is advised to make the compiler's life as easy as possible by using only the well-defined patterns that the compiler is able to recognize. The spaghetti kind of programming may confuse the compiler and lead to inferring unintended circuitry. The FPGA netlist will be produced by the compiler, but the output from such circuits may surprise the developer.

The FPGA program patterns can be put directly into the source file. It is an acceptable practice in small and medium projects. Larger projects should be divided into a set of *components*, each one implemented in its own source file. The component interfaces can be conveniently collected in a *package* that we named BlackVME_types in our examples. One can use more than one package, if the number of components grows beyond what is suitable for a single file.

Testing is tremendously important in FPGA projects. The tests need to be conducted while the project is developed. An alternative to incremental testing is a test plan executed after the coding is finished. In any case, significant thought and effort should be devoted to testing the responses of the FPGA circuits. Testing can be conducted in software simulations of the FPGA designs under the development environment. All development systems such as ISE provide tools for testing the netlist responses. The acid test can be conducted only with the actual hardware through looking at the FPGA outputs. LEDs are tremendously useful, as well as the logic outputs that we provide with the BlackVME board. We also provided the reconstruction DAC on our ADC card. The waveform reconstruction proved to be a very useful tool for developing the signal processing algorithms.

SkuTek Instrumentation is offering FPGA development services to our customers. Using our services may be the most cost-effective method to implement your application in the FPGA chip.

27 Important Notice

SkuTek Instrumentation reserves the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. SkuTek does not warrant or represent that any license, either express or implied, is granted under any SkuTek patent right, copyright, or other SkuTek intellectual property right relating to any combination, machine, or process in which SkuTek products or services are used. Information published by SkuTek regarding third-party products or services does not constitute a license from SkuTek to use such products or services or a warranty or endorsement thereof. SkuTek products are not authorized for use in safety-critical applications (such as life support) where a failure of the SkuTek product would reasonably be expected to cause severe personal injury or death. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products or applications and any use of SkuTek products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by SkuTek. Further, Buyers must fully indemnify SkuTek and its representatives against any damages arising out of the use of SkuTek products in such safety-critical applications. SkuTek products are neither designed nor intended for use in military/aerospace applications or environments. SkuTek products are neither designed nor intended for use in automotive applications or environments.

Linux software programs distributed by SkuTek for our Linux-based products are open-source software; you can redistribute it and/or modify these programs under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. The software programs are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with our Linux programs; if not, see the files named COPYING, or write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.

You can incorporate FPGA firmware source code examples developed and distributed by SkuTek in your projects either in their original form or modified to suit your needs. The examples remain our intellectual property. All rights are reserved by SkuTek to the extent permitted by law. Please retain our copyright statement in all copies that you make. The examples are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Source code examples developed by third parties remain the property of the respective owners.