

Using Loops

2021.1

Abstract

This lab demonstrates the usage of loops in a VHDL design. This lab should take approximately 60 minutes.

Objectives

After completing this lab, you will be able to:

- Construct a combination of a generic and generate statement(s) to produce code that will select at elaboration time 8 bits from one of two devices.
- Construct a collection of output buffers using a looping generate statement.
- Build a process in a testbench that serializes data.

Introduction

This design is intended for use in the `wave_gen.vhd` module and will steer either data from the UART receiver or the sample generator to the LEDs on the development board. You built the multiplexer version of the `LED_manager` in the "Using Concurrent Statements" lab; however, product specifications have changed and the selection of Channel1 or Channel2 occurs after coding, but before customer use. To save device resources you are being directed to replace the multiplexer with some capability to make this selection during synthesis (without changing the source).

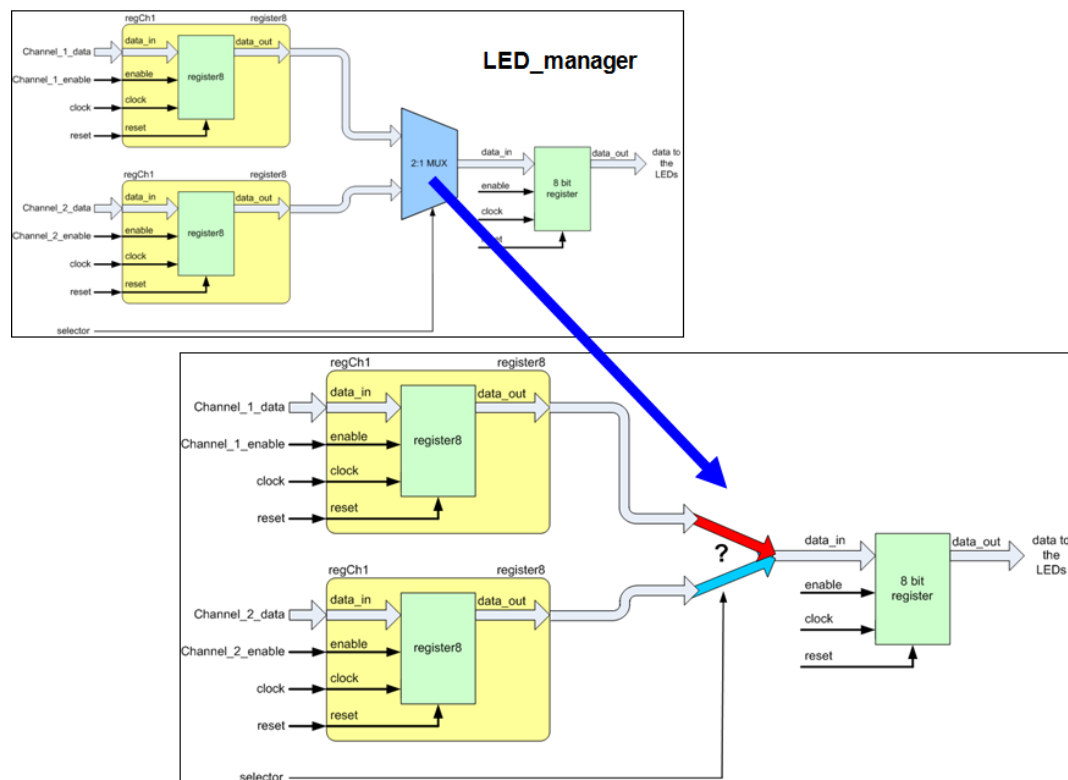


Figure 5-1: Block Diagram for the First Portion of This Lab

You will build upon the "Using Concurrent Statements" lab in which the `LED_manager` module was constructed for the first portion of the lab. You will modify this module by replacing the 2:1 combinatorial multiplexer with a conditional generate statement.

The second portion of the lab requires that the output buffers for the `LED_out` signals be instantiated. You will use a looping generate statement to create these buffers.

Finally, you will create a module to replace the serializer that you will use in an upcoming testbench lab.

Understanding the Lab Environment

The labs and demos provided in this course are designed to run on a Linux platform. Many of the labs and demos can be successfully executed in the Windows environment or a native Linux environment as well.

The instructions found in this lab are expressed using the Linux notation. This includes the forward slash ('/') as the hierarchy separator instead of the Windows backslash ('\'). Students who want to run the labs directly under Windows must use the correct hierarchy separator.

Customizable environment variables enable you to tailor your environment for specific machine configurations. The only environment variable (shown below) used in the customer training environment (CustEd_VM) points to the training directory where all the lab files are located. This reduces the amount of typing you need to do when entering directory paths.

This environment variable can be customized according to your specific location and can be set for Linux systems in the `/etc/profile` file and for Windows systems by entering "env" from the search bar.

The following is the environment variable used in the customer training VM:

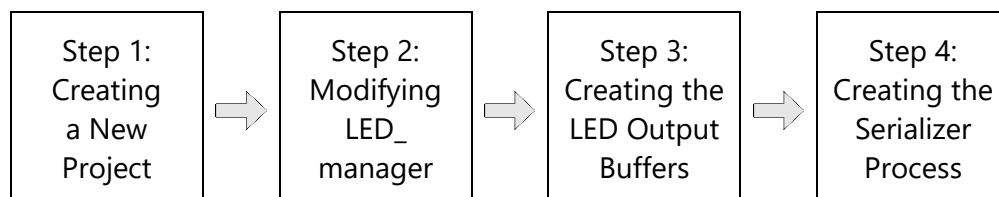
Environment Variable Name	Description
<code>\$TRAINING_PATH</code>	<p>Points to the space allocated for students to work through the labs. This directory includes prebuilt images and starting points for the labs and demos.</p> <p>The customer training VM sets <code>\$TRAINING_PATH</code> to the <code>/home/xilinx/training</code> directory.</p> <p>Typically, Windows users will install the training directory under C: to keep the path names as short as possible.</p>

Note: Environment variables are not supported from the Vitis IDE GUI. When using this tool, you must manually replace `$TRAINING_PATH` with the value of the variable, which in the customer training virtual machine, is `/home/xilinx/training`. Other tools, such as the Vivado Design Suite, will properly expand the environment variable.

Additional note about environments: Both the Vivado Design Suite and Vitis platform offer a Tcl environment. The contents of this environment are NOT preserved with the project. When the tools launch, they start with a pristine Tcl environment with none of the procs or variables remaining from a previous launch of the tools.

This means that if you sourced a Tcl script or manually set any Tcl variables and you closed the tool, when you reopen the tool (and perhaps even reopen the last project), you will need to source the Tcl script again and set any variables that the lab requires.

General Flow



Creating a New Project

Step 1

1-1. Launch the Vivado Design Suite.

If you do not recall how to perform this task, refer to the "Launching the Vivado Design Suite" section under Vivado Design Suite Operations in the *Lab Reference Guide*.

1-2. Create a new Vivado Design Suite project named *genLoops* and locate it in the following directory.

Browse to the `$TRAINING_PATH/genLoops/lab/KCU105` directory.

Target the project for the Kintex UltraScale KCU105 Evaluation Platform evaluation board.

If you do not recall how to perform this task, refer to the "Creating a Blank Vivado Design Suite Project" section in the *Lab Reference Guide*.

From the settings in the Flow Navigator, change the target language to VHDL.

Modifying LED_manager

Step 2

In this lab, you will import the LED_manager design file that was used in the "Using Concurrent Statements" lab. This design has a multiplexer which selects between two channels. This multiplexer is replaced by a conditional generated here. When this is complete, you will synthesize the design and compare the device resources with the original device resources as well as reviewing the RTL code.

Refer to the design block diagram in the Introduction section of this lab.

Now you will add the LED_manager and register8 modules (provided in the current working directory) into the project.

2-1. Add an HDL source file to the design.

2-1-1. Click **Add Sources** under Project Manager in the Flow Navigator.

The Add Sources Wizard opens.

2-1-2. Select **Add or create design sources**.

2-1-3. Click **Next**.

2-1-4. Click the **Plus (+)** icon and select **Add Files**.

2-1-5. Browse to the \$TRAINING_PATH/genLoops/support directory if it is not open already.

2-1-6. Select **LED_manager.vhd** and **register8.vhd**.

2-1-7. Click **OK**.

2-1-8. Ensure that the **Copy sources into project** option is selected.

2-1-9. Click **Finish** in the Add or Create Design Sources dialog box to add the HDL sources to the project.

2-2. Examine the multiplexer.

2-2-1. Select the **Sources** view.

2-2-2. Double-click **LED_manager** to open the source code in the text editor.

2-2-3. Examine the code.

2-3. Run synthesis.

2-3-1. Click **Run Synthesis** in the Flow Navigator under Synthesis.

Alternatively, you can also select **Flow > Run Synthesis** or press <F11>.

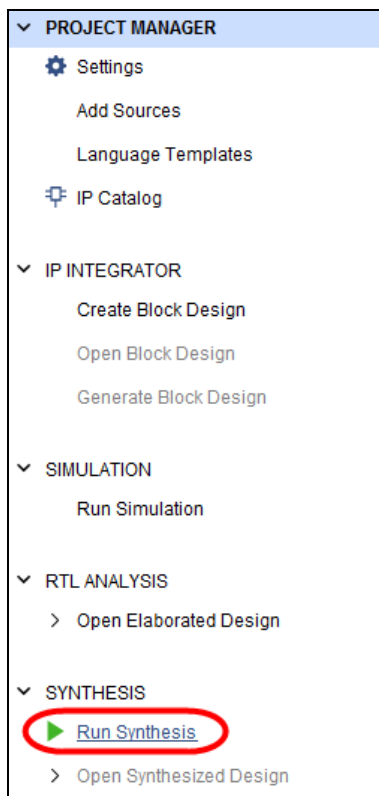


Figure 5-2: Selecting Run Synthesis

The Launch Runs dialog box opens.

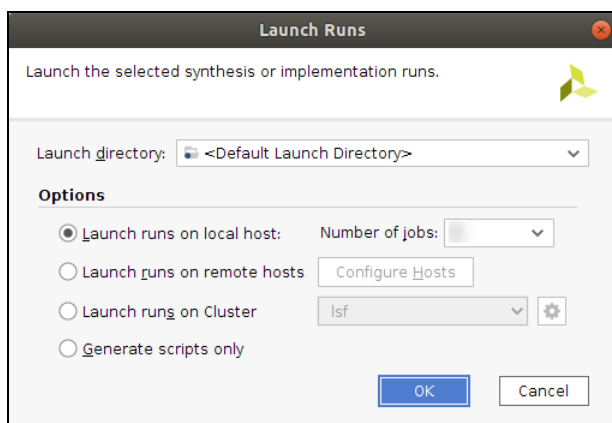


Figure 5-3: Setting the Launch Run Configuration

Hint: When launching the runs on a local host, it is common to set the number of jobs to the maximum value as this recruits the largest number of processors for the task and typically results in the least amount of time spent in synthesis.

2-3-2. Click **OK** to launch the runs.

2-3-3. Click **Save** if you are asked to save your files.

Once synthesis completes, you are asked what task you want to perform next: run implementation, open the synthesized design, view reports, or none of the above.

2-3-4. Select **Open Synthesized Design** (1).

2-3-5. [Optional] If you are familiar with accessing these various capabilities, you can disable this dialog box from appearing again by selecting **Don't show this dialog again** (2).

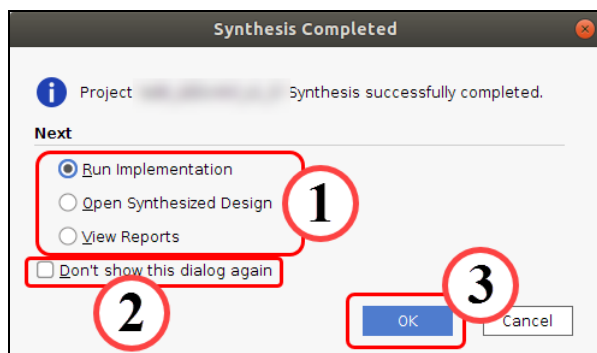


Figure 5-4: Selecting After Synthesis Options

2-3-6. Click **OK** to take the action you just selected (3) or click **Cancel** to simply close the dialog box.

2-4. Generate a Utilization report.

2-4-1. Click **Report Utilization** under Synthesized Design in the Flow Navigator.

Alternatively, you can select **Reports > Report Utilization**.

Make sure that the synthesized design is open at this point in order to access the reports.

2-4-2. Click **OK** for the default name utilization_1 in the pop-up window.

Question 1

Complete the Multiplexer Version column in the table below.

	Multiplexer Version	Other Version
Number of Slice Registers	24	8
Number of Slice LUTs	4	4

Resources Table

Question 2

How will you code the multiplexer so that its selection will be fixed at synthesis?

Don't understand the question

Question 3

Identify the inputs for the multiplexer. Which signals will and will not be used to implement the conditional generate?

Mux data selected

2-4-3. Close the synthesized design.

2-5. In `LED_manager.vhd`, replace the multiplexer with a conditional generate. Create a generic named `CHAN_SEL` of an integer type with a range from 1 to 2 to use as the selector.

Although suggested code solutions can be found in the Answers section, you should make every effort to code these yourself.

2-5-1. Highlight the code and click the **Comment** icon  in the toolbar of the text editor to comment out the multiplexer.

```
--  
-- construct a 2 input mux  
-- with Selector select  
-- mux_data_selected <= Channel_1_registered_data when '0',  
--                      Channel_2_registered_data when '1',  
--                      (others=>'-' ) when others;
```

Figure 5-5: Commenting out the Multiplexer

- 2-5-2.** Create the conditional generate to use the generic CHAN_SEL (which will be defined in a couple of steps) to drive Channel_1_registered_data to the mux_data_selected signal.

```
c1Sel: if (CHAN_SEL = 1) generate
    mux_data_selected <= Channel_1_registered_data;
end generate c1Sel;
```

Figure 5-6: Conditional generate to drive Channel_1_registered_data

- 2-5-3.** Repeat to create a second conditional generate to drive Channel_2_registered_data to the mux_data_selected signal.
- 2-5-4.** Add a generic to the entity named CHAN_SEL which is an integer that can have the value of either 1 or 2 (to serve as the selection mechanism between channels 1 and 2).
- 2-5-5.** Select **File > Text Editor > Save File**.

2-6. Synthesize LED_manager and open the Design Summary.

- 2-6-1.** Click **Run Synthesis** under Synthesis in the Flow Navigator.
- 2-6-2.** Select the **Open Synthesized Design** option in the dialog box and click **OK** if the Synthesis Completed dialog box appears.

2-7. Generate a Utilization report.

- 2-7-1.** Click **Report Utilization** under Synthesized Design in the Flow Navigator.
Alternatively, you can select **Reports > Report Utilization**.
- 2-7-2.** Click **OK** for the default name **utilization_1** in the pop-up window.

Question 4

Complete the Other Version column in the Resources table.

	Multiplexer Version	Other Version
Number of Slice Registers	16	8
Number of Slice LUTs	0	0

Resources Table

Question 5

Compare the results between the two synthesis runs. Explain the differences.

The generate statement uses 16 slice registers and 0 slice LUTs compared to the with select 24 slice registers and 4 slice LUTS(reference above)

2-7-3. Close the synthesized design.

Creating the LED Output Buffers

Step 3

The synthesis tools are capable of recognizing the top-level signals in a design and placing basic I/O buffers on them. However, sometimes you may want to take control from the synthesis tools and define buffers manually. This is typically done when the I/O buffers need special qualities associated with them, such as double-data rate or differential signaling, for example.

You will instantiate the output buffers for the LEDs in the UART_led module in this step.

3-1. Add the UART_led to the design and its supporting files:

- **uart_led.vhd**
- **meta_harden.vhd**
- **reset_bridge.vhd**
- **uart_rx.vhd**
- **uart_rx_ctl.vhd**
- **uart_baud_gen.vhd**

3-1-1. Click **Add Sources** under Project Manager in the Flow Navigator.

The Add Sources Wizard opens.

3-1-2. Select **Add or create design sources**.

3-1-3. Click **Next**.

3-1-4. Click the **Plus (+)** icon and select **Add Files**.

3-1-5. Press the **<Ctrl>** key and select all the files listed above.

3-1-6. Click **OK** and **Finish** to add the selected files to the project.

3-2. Disable the synthesis tool's ability to automatically insert I/O buffers.

3-2-1. Select **Settings** in the Flow Navigator.

Select **Synthesis** in the settings dialog box.

3-2-2. Enter **-no_iobuf** in the More Options field.

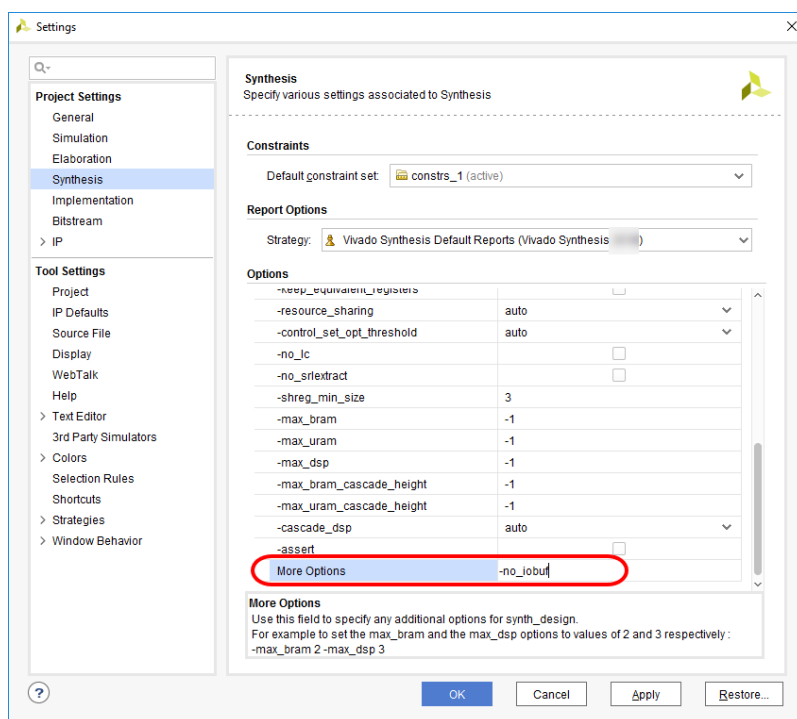


Figure 5-7: Disabling Automatic I/O Buffer Insertion

3-2-3. Click **OK**.

3-3. Add the appropriate generate statement to create eight instances of the OBUF, one for each LED output.

The OBUF is a primitive that is defined in Xilinx's UNISIM library in the VComponents package.

Although suggested code solutions can be found in the Answers section, you should make every effort to code these yourself.

Question 6

How can you determine what primitive component to use and how it should be connected?

Do not know!

- 3-3-1. Select the **Sources** view.
- 3-3-2. Double-click the **uart_led** module to open the file in the text editor.
- 3-3-3. Select **Tools > Language Templates** to open the Language Templates.
 - 1. Expand **VHDL > Device Primitive Instantiation > Kintex UltraScale > I/O > OUTPUT_BUFFER**.
 - 2. Select **Output Buffer (OBUF)**.
- 3-3-4. Add the instantiation template to `uart_led.vhd` at the location indicated by the comment "`—define the buffers for the outgoing data`".
- 3-3-5. Wrap the instantiation with a `for-loop generate` statement.
- 3-3-6. Change the **OBUF_inst** name to **OBUF_led_i**.
- 3-3-7. Change the input signal from 'I' to the output of LED_manager (LED_o).

Remember that this signal is a bus and that OBUF takes a single value.

Hint: You will have to represent the bus as a series of slices.
- 3-3-8. Change the output signal from 'O' to the pin assignment `led_pins`.

As with the previous step you will have to represent this as a slice.
- 3-3-9. Select **File > Text Editor > Save File**.

Question 7

What library needs to be loaded to use primitives? Why was this not a step in this process?

Library Unisim,Vcomponents package was already included with the `uart_LED` file

3-4. Now that the synthesis options have been set, launch synthesis and verify that the output buffers were properly integrated into the design.

3-4-1. Click **Run Synthesis** under Synthesis in the Flow Navigator.

3-4-2. After synthesis finishes, select **Open Synthesized Design** in the Synthesis Completed dialog box and click **OK**.

3-4-3. Select the **Netlist** view.

3-4-4. Right-click **uart_led** and select **Schematic** to launch the schematic viewer.

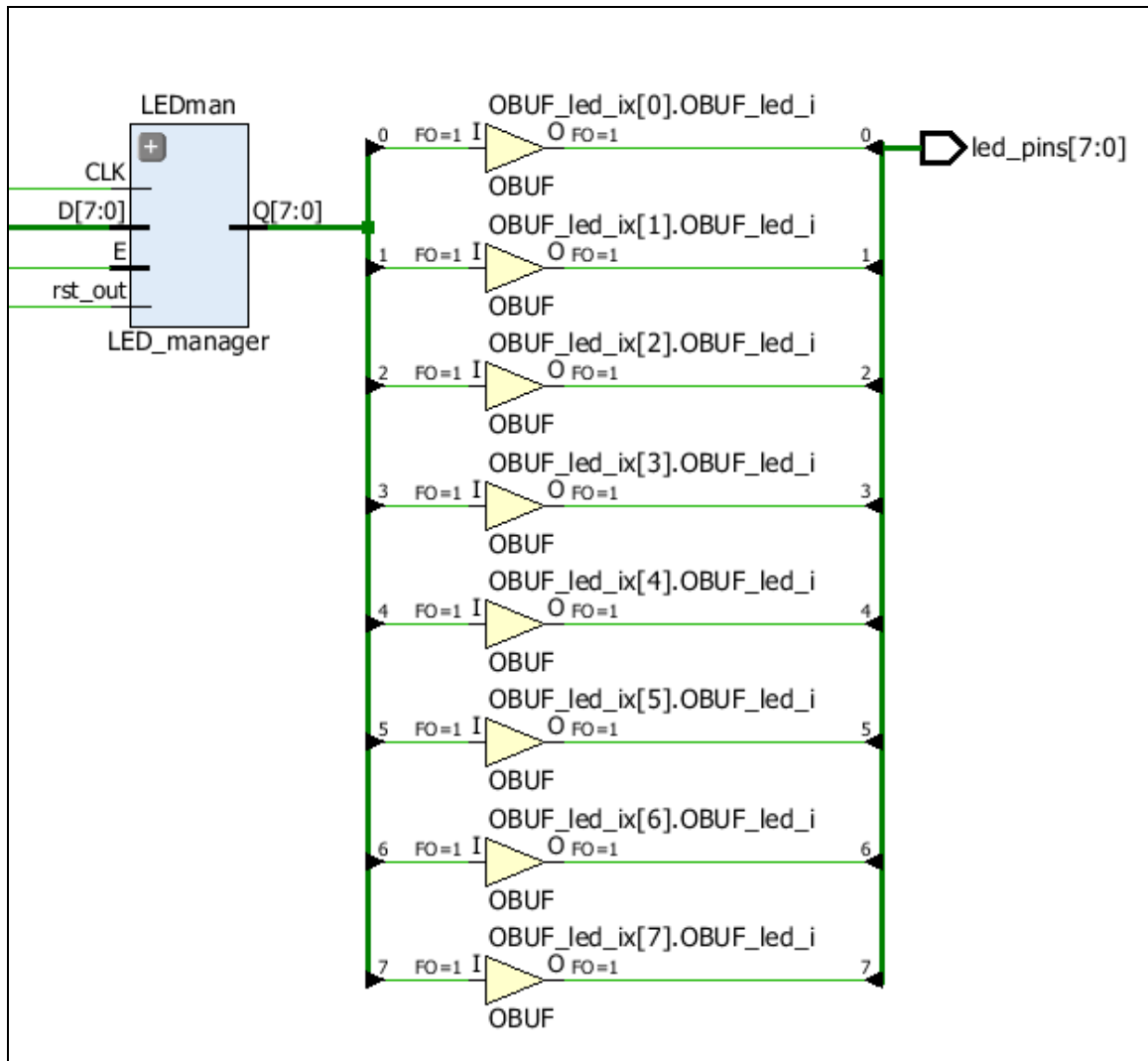


Figure 5-8: Schematic Showing the Eight OBUFs

3-4-5. Close the viewer windows.

Creating the Serializer Process

Step 4

The serializer process takes a provided string, breaks the string into characters, and converts the characters into a binary serial usable for testing an RS-232 receiver. The parameters for this process are 115,200 baud, 8 data bits, 1 stop bit, no parity, and no flow control.

The outline of the module has been provided for you. You are only required to fill in part of the process.

For your reference, here is a generic RS-232 frame.

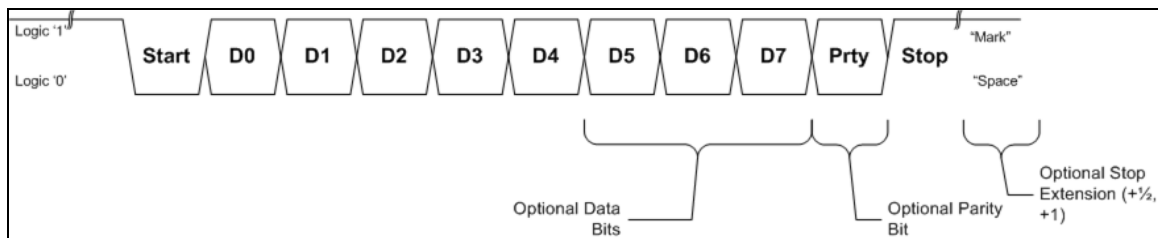


Figure 5-9: Generic RS-232 Frame

4-1. Load `tb_uart_driver.vhd` for simulation only.

This is a "helper" module for the final testbench that you will assemble in the last lab of this course.

4-1-1. Click **Add Sources** under Project Manager in the Flow Navigator.

The Add Sources Wizard opens.

4-1-2. Select **Add or create simulation sources**.

4-1-3. Click **Next**.

4-1-4. Click the **Plus (+)** icon and select **Add Files**.

4-1-5. Select `tb_uart_driver.vhd` and click **OK**.

This will keep the `tb_uart_driver` associated only with simulation and will not appear in the implementation sources.

4-1-6. Click **Finish** to close the Add and Create Simulation Sources dialog box.

4-2. Set the `tb_uart_driver.vhd` as the top-level file for simulation.

4-2-1. Expand **Simulation Sources** in the Sources view.

4-2-2. Right-click `tb_uart_driver.vhd` under **Simulation Sources** > **sim_1** and select **Set as Top**.

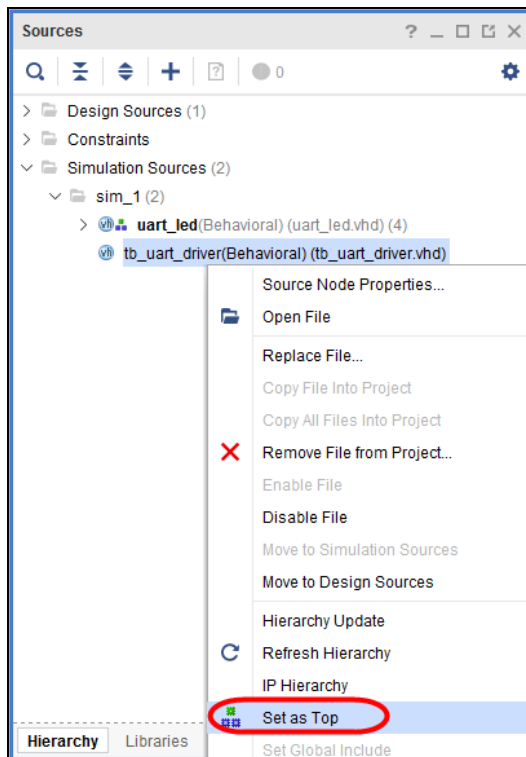


Figure 5-10: Selecting `tb_uart_driver.vhd` as the Top-Level File

4-3. Open `tb_uart_driver` and write the code to generate RS-232 style packets based on the provided "MESSAGE".

Although suggested code solutions can be found in the Answers section, you should make every effort to code these yourself.

4-3-1. Select the **Sources** view.

4-3-2. Expand **Simulation Sources** and double-click `tb_uart_driver.vhd`.

4-3-3. Locate the comment "-- loop through the entire message" in the source code.

4-3-4. Write the code to break the string described in the generic "MESSAGE" into characters.

Hint: Try using a for loop.

4-3-5. Write the code to convert the character into an 8-bit `std_logic_vector` named `slv_char_to_send`.

Hint: The attribute `'pos` can be used in the synthesis environment to return the ASCII equivalent of a character. The use in this case is `character'pos(<character you want to send>)`.

4-3-6. Write the code to send the start bit.

Hint: The start bit is a logic 0 that is presented to the serial line for one baud period.

Hint: There is a pre-defined constant that will help you.

4-3-7. Write the code to send all 8 bits of the character to send with the most significant bit sent last.

Hint: Use a nested for loop. Be sure not to name this loop identifier the same as the outer loop identifier.

4-3-8. Write the code to generate a 1-bit period high signal to indicate the stop bit.

Hint: This is exactly like the start bit, but with a different value.

4-3-9. Write the code to create a 10-ns pulse to indicate that the character was sent

4-3-10. Write the code to wait 5 bit periods before sending the next packet.

4-3-11. Select **File > Text Editor > Save File**.


4-4. Simulate the design. Load the provided configuration file genLoops.wcfg. Set the run time for 2 ms and examine the waveform.

4-4-1. Select **Simulation > Run Simulation > Run Behavioral Simulation** in the Sources view.

This will launch the Vivado simulator.

4-4-2. Select **File > Simulation Waveform > Open Configuration**.

4-4-3. Select **genLoops.wcfg** and click **OK**.

4-4-4. Select **Run > Restart** (or click the  icon) to restart the simulation.

4-4-5. Select **Run > Run All** (or click the  icon) to run the simulation.

4-4-6. Click the **Zoom to Fit**  icon to zoom to full screen.

Question 8

Is the design working? How can you tell?

4-5. Close the simulation.

4-5-1. Select **File > Close Simulation**.

4-5-2. Click **OK** in the Confirm Close window.

4-5-3. Click **Discard** in the pop-up dialog box.

4-6. Close the Vivado Design Suite.

4-6-1. Select **File > Exit**.

The Exit Vivado dialog box opens.



Figure 5-11: Exit Vivado Dialog Box

4-6-2. Click **OK**.

Some systems (particularly VMs) may be memory constrained. Removing the workspace frees a portion of the disk space, allowing other labs to be performed.

You can delete the directory containing the lab you just ran by using the graphical interface or the command line interface. You can choose either mechanism. Both processes will recursively delete all the files in the `$TRAINING_PATH/genLoops` directory.

4-7. [Optional] [Only for local VMs—not for CloudShare] Clean up the file system.

Using the GUI:

4-7-1. Using the graphical browser (Windows: press the **<Windows>** key + **<E>**; Linux: press **<Ctrl + N>**), navigate to `$TRAINING_PATH/genLoops`.

4-7-2. Select **genLoops**.

4-7-3. Press **<Delete>**.

-- OR --

Using the command line:

4-7-4. Open a terminal window (Windows: press the **<Windows>** key + **<R>**, then enter **cmd**; Linux: press **<Ctrl + Alt + T>**).

4-7-5. Enter the following command to delete the contents of the workspace:

[Windows users]: `rd /s /q $TRAINING_PATH/genLoops`

[Linux users]: `rm -rf $TRAINING_PATH/genLoops`

Summary

This lab walked you through three common activities when coding:

- Making a resource versus functionality trade-off while still preserving some amount of flexibility
- Using generate loops to reduce the time it takes to code redundant activities (and to keep the code more concise)
- Using loop statements to ease the repetition when creating stimulus

Answers

Answers listed represent sample solutions only. Your results may differ depending on the version of the software, service pack, or operating system that you are using.

1. Complete the Multiplexer Version column in the table below.

	Multiplexer Version	Other Version
Number of Slice Registers	24	
Number of Slice LUTs	4	

Resources Table

2. How will you code the multiplexer so that its selection will be fixed at synthesis?

A conditional generate based on a generic is one possibility for solving this problem.

3. Identify the inputs for the multiplexer. Which signals will and will not be used to implement the conditional generate?

The multiplexer takes Channel_1_registered_data and Channel_2_registered_data as inputs. This will not change. The output remains mux_data_selected. The remaining signal "Selector" is currently a signal which can change during the operation of the design which will cause mux_data_selected to dynamically switch input channels. Because the selector will be fixed at synthesis, this selector signal will go unused.

This implies the need for some mechanism to select which channel will be enabled during synthesis. Use a generic to make this selection.

4. Complete the Other Version column in the Resources table.

	Multiplexer Version	Other Version
Number of Slice Registers	24	16
Number of Slice LUTs	4	0

Resources Table

5. Compare the results between the two synthesis runs. Explain the differences.

The conditional generate version of the code used fewer resources than the multiplexer version. This is not surprising as all of the code for the multiplexer itself (which was implemented as LUTs) no longer needs to be built. The implementation becomes only

connections. Because only one channel needs to be registered, the eight flip-flops used to register the other channel have been optimized away.

This is a clear example between choosing flexibility or minimal resources.

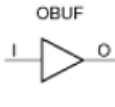
6. How can you determine what primitive component to use and how it should be connected?

One method is to refer to document known as the *<FPGA family name> Libraries Guide for HDL Designs*. These documents, one per FPGA family, describe the behavior of all of the primitive components and their interface.

Here is a small portion of the manual for the OBUF.

OBUF

Primitive: Output Buffer



Introduction

This design element is a simple output buffer used to drive output signals to the FPGA device pins that do not need to be 3-stated (constantly driven). Either an OBUF, OBUFT, OBUFDS, or OBUFTDS must be connected to every output port in the design.

This element isolates the internal circuit and provides drive current for signals leaving a chip. It exists in input/output blocks (IOB). Its output (O) is connected to an OPAD or an IOPAD. The interface standard used by this element is LVTTTL. Also, this element has selectable drive and slew rates using the DRIVE and SLOW or FAST constraints. The defaults are DRIVE=12 mA and SLOW slew.

Port Descriptions

Name	Direction	Width	Function
O	Output	1-bit	Output of OBUF to be connected directly to top-level output port.
I	Input	1-bit	Input of OBUF. Connect to the logic driving the output port.

Instantiation	Yes
Inference	Recommended
Coregen and wizards	No
Macro support	No

Figure 5-12: Snippet from the Spartan-3E FPGA Libraries Guide for HDL Designs

Another quicker method is to use the Language Templates. The templates provide the instantiation template that can be cut-and-pasted directly into your code (however, there is little or no description about how the component works).

Another advantage to the templates is that the primitives are sorted by function, not by name as they are in the Libraries Guides. This makes finding the right component easier if you do not know the exact name.

7. What library needs to be loaded to use primitives? Why was this not a step in this process?

The VComponents package within the UNISIM library contains the declarations for all of the primitive components and associated macros. This library was already loaded to support the other input and output buffers.

8. Is the design working? How can you tell?

The design seems to be working.

The first item to test is the baud rate: measure from one data_sent pulse to the next. This can be done using the click-and-drag markers or placing two cursors and computing the delta between them.

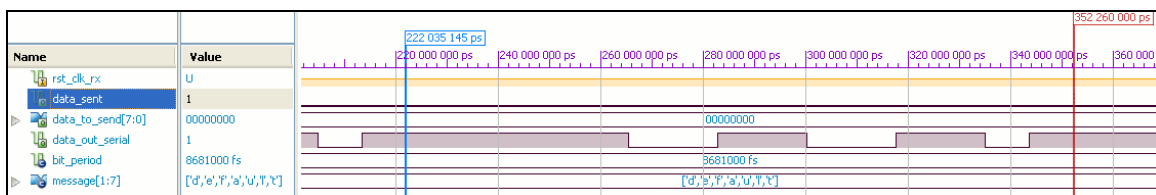


Figure 5-13: Measuring Baud Rate

The difference between the two data_sent pulses is 130.225 μ s. This includes 5 bit periods for the delay between frames and the start, stop, and data bits for a total of 15 bit periods or 8.68 μ s per pulse which results in 115,200 baud. So, this seems to be correct.

Next, you can pick one of the frames and manually decode it. Because you measured the third frame in time, you might as well decode it.

Notice the one shorter bit just prior to the stop bit? Start there and work backwards.

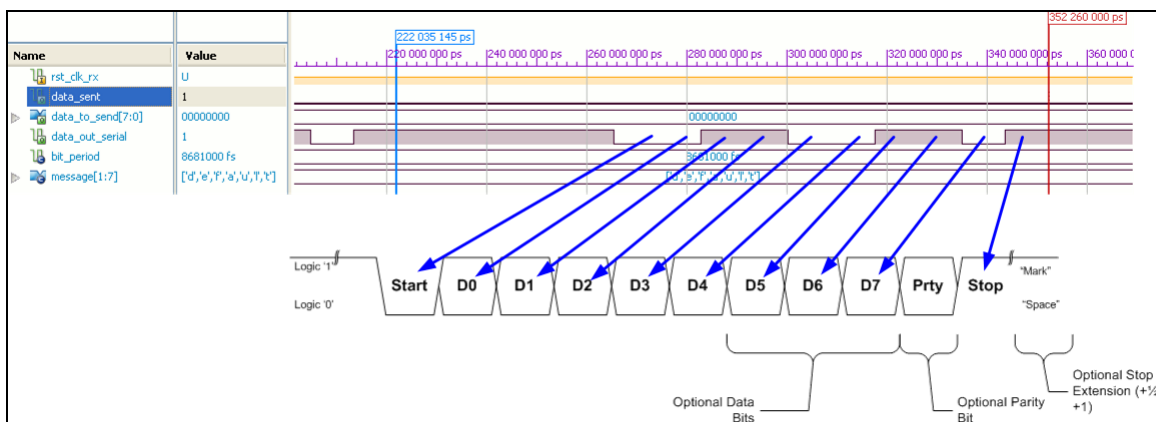


Figure 5-14: Decoding the Waveform

According to this, the result is hexadecimal 66, which is the character code for 'f'. This, not surprisingly, is the third character sent in the message.

Therefore, you can conclude that this design is working correctly.

LED_manager.vhd

Drive Channel_1_registered_data to the mux_data_selected Signal

```
c1se1: if (CHAN_SEL = 1) generate
    mux_data_selected <= Channel_1_registered_data;
end generate c1se1;
```

Drive Channel_2_registered_data to the mux_data_selected Signal

```
c2se1: if (CHAN_SEL = 1) generate
    mux_data_selected <= Channel_2_registered_data;
end generate c2se1;
```

Declaring as generic

```
generic ( CHAN_SEL : integer range 1 to 2 := 1);
```

Final Code

Code to create instances of the OBUF

```
-- define the buffers for the outgoing data
OBUF_led_ix : for i in 0 to 7 generate
    OBUF_led_i: OBUF
        generic map (
            DRIVE => 12,
            IOSTANDARD => "DEFAULT",
            SLEW => "SLOW")
        port map (
            O => led_pins(i),    -- Buffer output (connect directly to top-level port)
            I => LED_o(i)        -- Buffer input
        );
    end generate OBUF_led_ix;
```

Figure 5-15: Creating Instances of the OBUF

Final Code of tb_uart_driver.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
```

```
library std;
use STD.TEXTIO.all;
```

```
entity tb_uart_driver is
    generic (
        MESSAGE      : String := "default";
        BIT_PERIOD    : time := 8681 ns
    );
```

```

port (
    rst_clk_rx      : in  std_logic;
    data_sent       : out std_logic;
    data_to_send    : out std_logic_vector(7 downto 0);
    data_out_serial : out std_logic
);
end tb_uart_driver;

architecture Behavioral of tb_uart_driver is
begin
    send_char_bitper: process
        variable char_to_send      : character := '?';
        variable char_code         : integer range 0 to 255 := 0;
        variable slv_char_to_send  : std_logic_vector(7 downto 0) := (others=>'U');
        variable char_as_string    : string (1 to 1);
        variable hex_as_string     : string (1 to 6);
        variable console_message   : string (1 to 40);

    begin
        data_to_send      <= (others=>'0');
        data_out_serial   <= '1';
        data_sent         <= '0';

        -- wait until (rst_clk_rx = '0');
        wait for 5 us;

        -- loop through the entire message
        sendChars: for i in 1 to MESSAGE'length loop
            -- break the string into characters
            -- convert the character into a std_logic_vector of 8 bit
            char_to_send      := MESSAGE(i); -- convert char to slv8
            char_code         := character'pos(char_to_send); -- char to int
            slv_char_to_send := std_logic_vector(to_unsigned(char_code,8));

            -- send the start bit, then the 8 data bits, then the stop bit
            data_out_serial   <= '0';          -- send start bit
            wait for BIT_PERIOD;              -- and hold for one bit period

            -- loop through the 8 bits of data and send lsb first
            for j in 0 to 7 loop                -- loop through all 8 bits
                data_out_serial <= slv_char_to_send(j);
                wait for BIT_PERIOD;            -- and hold for one
            end loop;

            data_out_serial <= '1';              -- send stop bit
            wait for BIT_PERIOD;                -- delay a period
        end loop;
    end process;
end architecture Behavioral of tb_uart_driver;

```

```
        data_sent    <= '1';
        wait for 10 ns;
        data_sent <= '0';

        wait for BIT_PERIOD*5;                                -- idle 5 bit periods
    end loop; -- sendChars
    wait for 1 ms;
    assert false
    report "***** Simulation Ending due to all data being transmitted! *****"
    severity failure;
    wait;
end process send_char_bitper;
end Behavioral;
```