Toyon Research Corporation

Lab 8: Processor Integration

Chilipepper Tutorial Projects

Version 0.1
5/24/2013

Table of Contents

Introduction 3
Procedure 3
Objectives 3
Generate HDL Code 4
1.1 MATLAB Functions 4
1.2 MATLAB Test Bench 5
1.3 RX HDL Coder Project 5
Export Simulink models 10
2.1 Create MCU Simulink Design 10
2.2 Create Receiver Simulink Design 11
2.3 Create ADC driver Simulink Design 13
2.4 DC Offset Core 15
Configure Cores and Export Design 17
3.1 Needed IP Cores 17
3.2 Configuring the Pcore Ports 18
3.3 Pin Assignments 18
Create software project 20
4.1 Creating a new C Project 20
4.2 Adding Supporting files 21
4.3 Loading Hardware Platform with iMPACT 22
4.4 Debugging with SDK 24
Testing and Design Verification 25
5.1 Verification with Terminal 25
Appendix A Core MATLAB rx Function 26
Appendix B MATLAB Frequency Offset Correction 28
Appendix C MATLAB Timing Offset Correction 30
Appendix D MATLAB Correlation Function 32
Appendix E main.c 36

Lab 8: Processor Integration

Introduction

This lab will extend the previous labs and allow you to interface your receive core to the Xilinx MicroBlaze Processor. The Focus of this integration will be on sending a message via button press on the FPGA using Lab 3, receiving the transmitted QPSK waveform on a second FPGA using lab 7, and outputting this message directly to a Terminal by extending the lab 6 SDK project. The lab will assume the message was received using a second Chilipepper board however if you do not have two boards, you can extend lab 3 by allowing a user to input a message to the Tx core via terminal, export the transmitted QPSK waveform using ChipScope, and use MATLAB to analyze the transmitted message. Both the transmit and receive core created in the previous labs will be used in this lab, and prior knowledge of the workings of HDL Coder as well as the Xilinx EDK environment is assumed. It is recommended that you complete the previous labs before completing this lab.

This lab is created using:

- MATLAB 2013a
- Xilinx ISE Design Suite 14.4 with EDK and System Generator
- Windows 7, 64-bit

Procedure

This lab is organized into a series of steps, each including general instructions and supplementary steps, allowing you to take advantage of the lab according to your experience level.

This lab consists of the following basic steps:

- Create and export Simulink models using System Generator
- Configure your created PCores and export the design into SDK
- Create software to run your design
- Test and verify your results

Objectives

After completing this lab, you will be able to:

- Create a Simulink model to interface your PCore to the MicroBlaze Processor
- Send and Receive a QPSK Waveform using the Chilipepper FMC
- Send output to the MicroBlaze Processor Serial Port
- Create a software application to test your design
- Verify your results in ChipScope and analyze them using MATLAB

Generate HDL Code Step 1

This section will show you how to create your MATLAB function and test bench files as well as the process for generating the HDL code used in the Simulink model.

1.1 MATLAB Functions

Your MATLAB functions will eventually become a core that will be synthesized into hardware. The algorithm describes the operations in each clock cycle, and processes data on a sample-by-sample basis. This lab builds on the MATLAB algorithm used in Lab 7 and adds processor integration with the FPGA MicroBlaze. This is an important step towards allowing for the FPGA code created in SDK to have full control over the underlying workings of the transmit and receive algorithms written in MATLAB. To add this functionality to the existing receiver algorithm, we need to make several small changes to the MATLAB files used in the previous labs. The first function we must modify is the qpsk_rx function used in the previous labs. This function was at the core of our qpsk receiver and allowed us to call other important functions such as those used for timing and frequency correction. The modified version of this function is shown in appendix A. This function is almost identical to the one used in the previous lab, we have simply added several signals which allow for more control of the inner working of each function.

1. Create a directory for the project under C:\QPSK_Projects\Project_8.
2. Create a MATLAB directory within the main project directory.
3. Create a new **MATLAB function** with the contents of Appendix A.
4. **Save** this function as `qpsk_rx.m` inside the project directory.

The first function called by `rx_qpsk` which needs to be modified is `qpsk_rx_foc`. This function simply needs to add an input parameter which can be used to reset the frequency offset to zero after a packet has been fully received. The new function is shown in Appendix B.

5. Create a new **MATLAB function** with the contents of Appendix B.
6. Save this function as `qpsk_rx_foc.m` inside the MATLAB project directory

The next function we will modify is `qpsk_rx_toc`. Just like the previous modification, the timing estimate needs to be reset after a packet has been received successfully. The new function is shown in Appendix C

7. Create a new **MATLAB function** with the contents of Appendix C.
8. Save this function as `qpsk_rx_foc.m` inside the MATLAB project directory

The last function called by `rx_qpsk` that needs a slight modification is the `qpsk_rx_correlator` function created in Lab 7. Like the previous two modification, we will add an input to allow for more control over the functionality of the algorithm. In this case however, it is used primarily to let the correlator know when it can start decoding and sending bits, as the MCU is ready to receive them and transfer them out the serial port. If the MCU is not ready, the correlator should not attempt to decode the signal to find a packet. In addition, there a few signals added to allow the correlator to communicate to the processor when it has finished its byte translation, and also how many bytes are ready to be transferred into the buffer. The new function is shown in Appendix D.

9. Create a new **MATLAB function** with the contents of Appendix D.
10. Save this function as `qpsk_rx_correlator.m` inside the MATLAB project directory.

Note	Optionally you can download any of these MATLAB files from the GitHub[1] Repo.

1.2 MATLAB Test Bench

Now that you have made the modifications to the algorithm needed for processor integration, we need to modify our test bench script slightly to send and receive these new signals.

1. Create a new **MATLAB function** with the contents of Appendix E.
2. Save this function as `qpsk_tb.m` inside the MATLAB project directory.

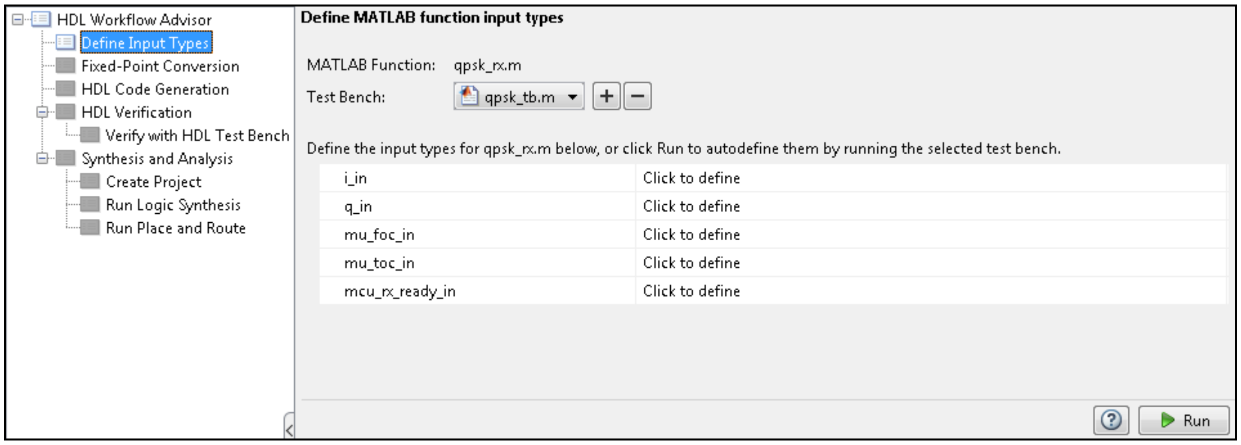
This function doesn't do much in terms of testing the integration portion of the design, but running it should at least verify that there are no errors in your MATLAB functions. Once you verify that you're able to decode the simulated message, you're ready to compile your design in HDL coder.

1.3 RX HDL Coder Project

Using the same steps outlined in the previous labs, create a new HDL coder project called rx_qpsk. Add your MATLAB function `qpsk_rx.m` and your test bench script `qpsk_tb.m` to the **MATLAB Function** and **MATLAB Test Bench** categories respectively.

Once you open your Workflow Advisor, you should be greeted with a screen similar to Figure 1-4 which allows you to define input types for your function. You can also allow them to be auto-defined by simply selecting run, and letting MATLAB analyze your design. For the inputs listed for the qpsk_rx function, the auto-defined types are fine.

[1] The MATLAB files can be found at https://github.com/Toyon/Chilipepper/tree/master/Labs/Lab_8/MATLAB



Define MATLAB function input types

MATLAB Function: qpsk_rx.m

Test Bench: qpsk_tb.m

Define the input types for qpsk_rx.m below, or click Run to autodefine them by running the selected test bench.

i_in	Click to define
q_in	Click to define
mu_foc_in	Click to define
mu_toc_in	Click to define
mcu_rx_ready_in	Click to define

Figure 1-4: HDL Code Generation Workflow Advisor

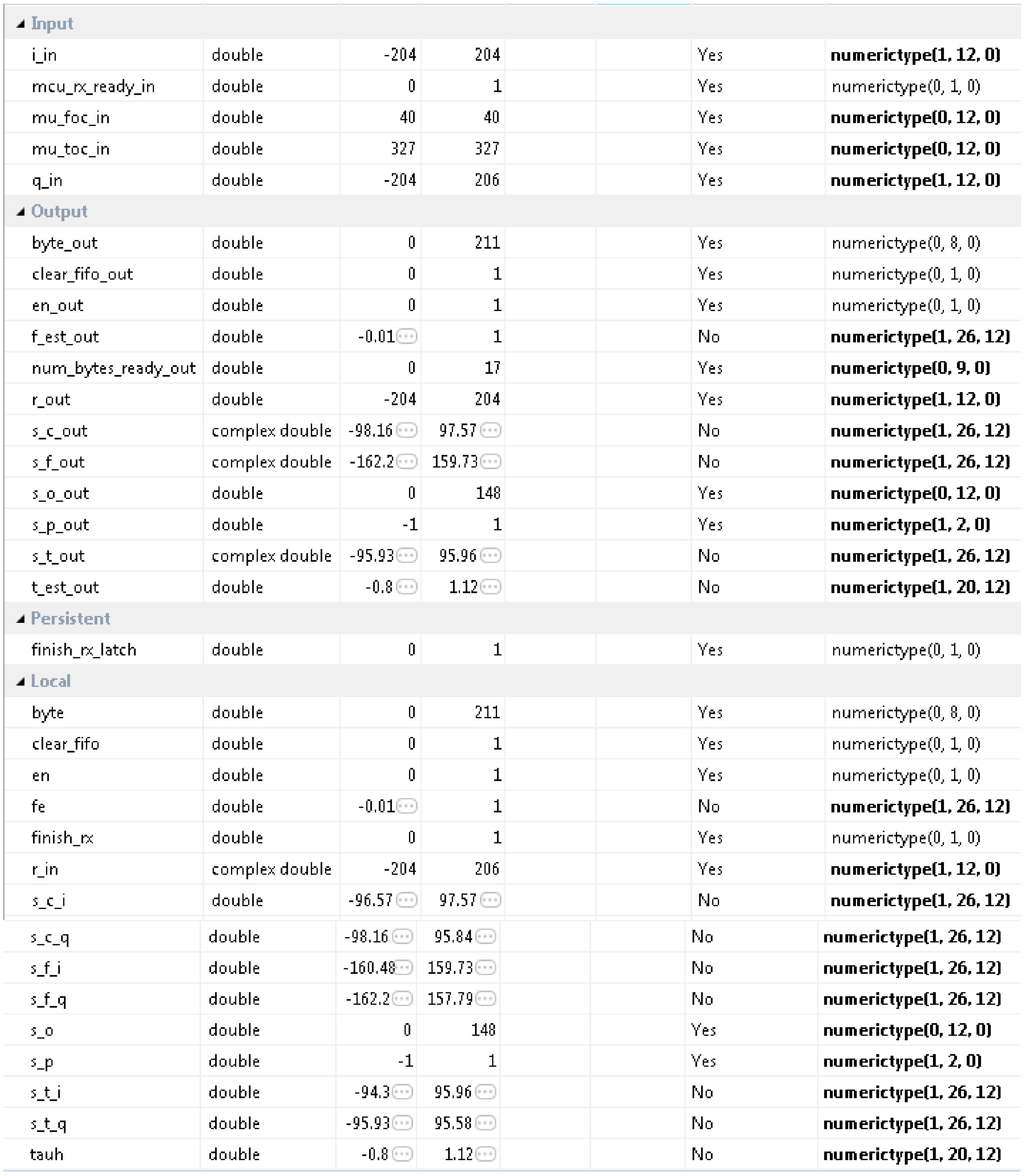
1. Open Workflow Advisor and select “Run” to define the input types.
2. Click on Fixed-Point Conversion and select run this task. This process may take awhile.

Once the process is completed, you should receive a popup that says “Validation succeeded”. This means that MATLAB has successfully analyzed your design and selected fixed point types to replace the floating point arithmetic required in your algorithm. However, not all of the automatic selections are sufficient for our FPGA design; therefore several of the conversions will need to be modified.

3. Using the function dropdown menu at the top of the HDL Code Generation screen, select each of the functions in the design and make the following modifications.

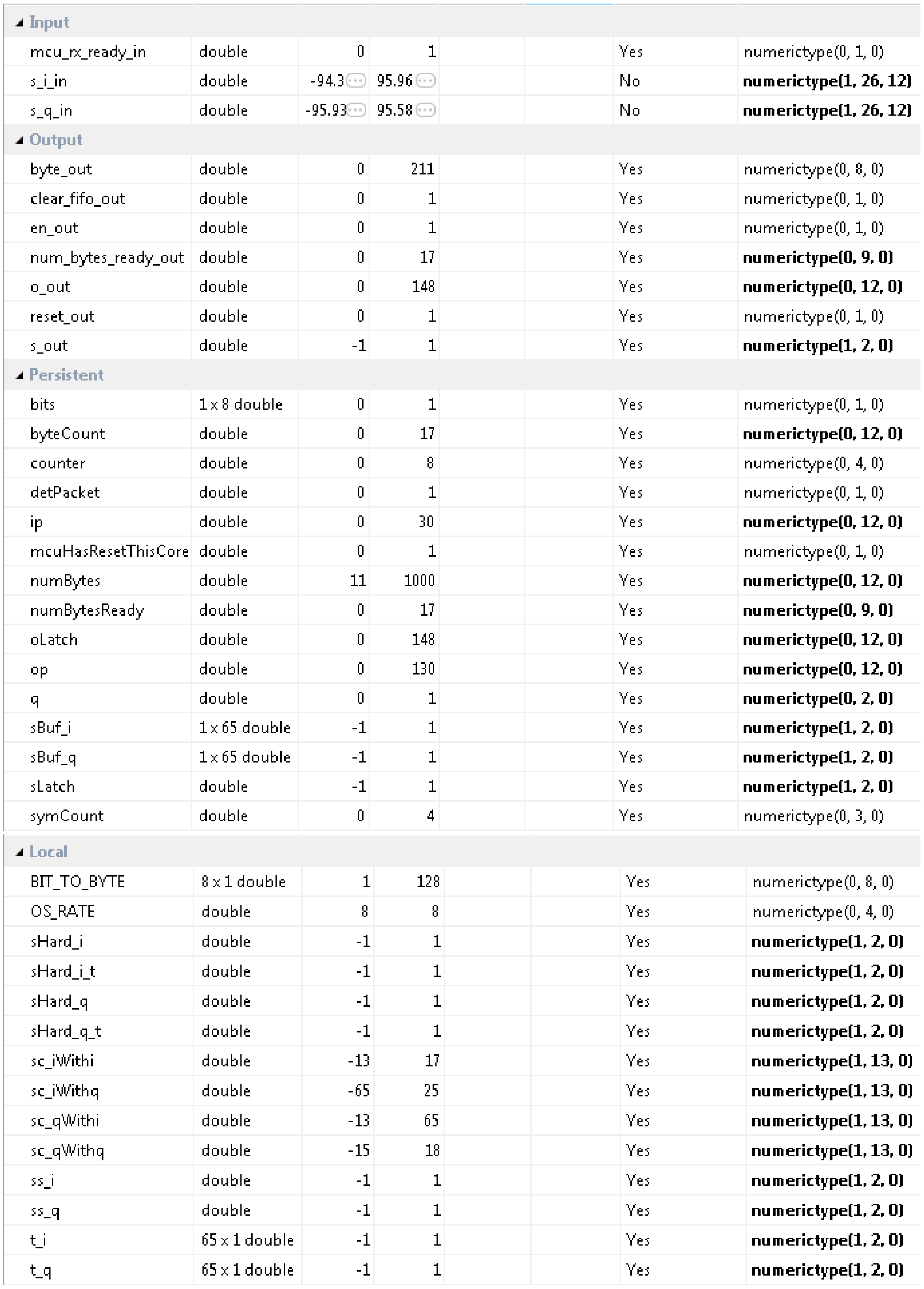
Note	Many of the Fixed-Point conversions are the same as those used in the previous labs. For your convenience, this lab only shows the Functions which have more input/output variables than the previous labs, and whose proposed types need to be modified. Refer to the previous lab for information on all other proposed types.

qpsk_rx



◢ Input							
i_in	double	-204	204			Yes	**numerictype(1, 12, 0)**
mcu_rx_ready_in	double	0	1			Yes	numerictype(0, 1, 0)
mu_foc_in	double	40	40			Yes	**numerictype(0, 12, 0)**
mu_toc_in	double	327	327			Yes	**numerictype(0, 12, 0)**
q_in	double	-204	206			Yes	**numerictype(1, 12, 0)**
◢ Output							
byte_out	double	0	211			Yes	numerictype(0, 8, 0)
clear_fifo_out	double	0	1			Yes	numerictype(0, 1, 0)
en_out	double	0	1			Yes	numerictype(0, 1, 0)
f_est_out	double	-0.01⋯	1			No	**numerictype(1, 26, 12)**
num_bytes_ready_out	double	0	17			Yes	**numerictype(0, 9, 0)**
r_out	double	-204	204			Yes	**numerictype(1, 12, 0)**
s_c_out	complex double	-98.16⋯	97.57⋯			No	**numerictype(1, 26, 12)**
s_f_out	complex double	-162.2⋯	159.73⋯			No	**numerictype(1, 26, 12)**
s_o_out	double	0	148			Yes	**numerictype(0, 12, 0)**
s_p_out	double	-1	1			Yes	**numerictype(1, 2, 0)**
s_t_out	complex double	-95.93⋯	95.96⋯			No	**numerictype(1, 26, 12)**
t_est_out	double	-0.8⋯	1.12⋯			No	**numerictype(1, 20, 12)**
◢ Persistent							
finish_rx_latch	double	0	1			Yes	numerictype(0, 1, 0)
◢ Local							
byte	double	0	211			Yes	numerictype(0, 8, 0)
clear_fifo	double	0	1			Yes	numerictype(0, 1, 0)
en	double	0	1			Yes	numerictype(0, 1, 0)
fe	double	-0.01⋯	1			No	**numerictype(1, 26, 12)**
finish_rx	double	0	1			Yes	numerictype(0, 1, 0)
r_in	complex double	-204	206			Yes	**numerictype(1, 12, 0)**
s_c_i	double	-96.57⋯	97.57⋯			No	**numerictype(1, 26, 12)**
s_c_q	double	-98.16⋯	95.84⋯			No	**numerictype(1, 26, 12)**
s_f_i	double	-160.48⋯	159.73⋯			No	**numerictype(1, 26, 12)**
s_f_q	double	-162.2⋯	157.79⋯			No	**numerictype(1, 26, 12)**
s_o	double	0	148			Yes	**numerictype(0, 12, 0)**
s_p	double	-1	1			Yes	**numerictype(1, 2, 0)**
s_t_i	double	-94.3⋯	95.96⋯			No	**numerictype(1, 26, 12)**
s_t_q	double	-95.93⋯	95.58⋯			No	**numerictype(1, 26, 12)**
tauh	double	-0.8⋯	1.12⋯			No	**numerictype(1, 20, 12)**

`qpsk_rx_correlator`



Input							
mcu_rx_ready_in	double	0	1			Yes	numerictype(0, 1, 0)
s_i_in	double	-94.3	95.96			No	**numerictype(1, 26, 12)**
s_q_in	double	-95.93	95.58			No	**numerictype(1, 26, 12)**
Output							
byte_out	double	0	211			Yes	numerictype(0, 8, 0)
clear_fifo_out	double	0	1			Yes	numerictype(0, 1, 0)
en_out	double	0	1			Yes	numerictype(0, 1, 0)
num_bytes_ready_out	double	0	17			Yes	**numerictype(0, 9, 0)**
o_out	double	0	148			Yes	**numerictype(0, 12, 0)**
reset_out	double	0	1			Yes	numerictype(0, 1, 0)
s_out	double	-1	1			Yes	**numerictype(1, 2, 0)**
Persistent							
bits	1 x 8 double	0	1			Yes	numerictype(0, 1, 0)
byteCount	double	0	17			Yes	**numerictype(0, 12, 0)**
counter	double	0	8			Yes	numerictype(0, 4, 0)
detPacket	double	0	1			Yes	numerictype(0, 1, 0)
ip	double	0	30			Yes	**numerictype(0, 12, 0)**
mcuHasResetThisCore	double	0	1			Yes	numerictype(0, 1, 0)
numBytes	double	11	1000			Yes	**numerictype(0, 12, 0)**
numBytesReady	double	0	17			Yes	**numerictype(0, 9, 0)**
oLatch	double	0	148			Yes	**numerictype(0, 12, 0)**
op	double	0	130			Yes	**numerictype(0, 12, 0)**
q	double	0	1			Yes	**numerictype(0, 2, 0)**
sBuf_i	1 x 65 double	-1	1			Yes	**numerictype(1, 2, 0)**
sBuf_q	1 x 65 double	-1	1			Yes	**numerictype(1, 2, 0)**
sLatch	double	-1	1			Yes	**numerictype(1, 2, 0)**
symCount	double	0	4			Yes	numerictype(0, 3, 0)
Local							
BIT_TO_BYTE	8 x 1 double	1	128			Yes	numerictype(0, 8, 0)
OS_RATE	double	8	8			Yes	numerictype(0, 4, 0)
sHard_i	double	-1	1			Yes	**numerictype(1, 2, 0)**
sHard_i_t	double	-1	1			Yes	**numerictype(1, 2, 0)**
sHard_q	double	-1	1			Yes	**numerictype(1, 2, 0)**
sHard_q_t	double	-1	1			Yes	**numerictype(1, 2, 0)**
sc_iWithi	double	-13	17			Yes	**numerictype(1, 13, 0)**
sc_iWithq	double	-65	25			Yes	**numerictype(1, 13, 0)**
sc_qWithi	double	-13	65			Yes	**numerictype(1, 13, 0)**
sc_qWithq	double	-15	18			Yes	**numerictype(1, 13, 0)**
ss_i	double	-1	1			Yes	**numerictype(1, 2, 0)**
ss_q	double	-1	1			Yes	**numerictype(1, 2, 0)**
t_i	65 x 1 double	-1	1			Yes	**numerictype(1, 2, 0)**
t_q	65 x 1 double	-1	1			Yes	**numerictype(1, 2, 0)**

Once all modifications have been made, select “Validate Types” in the top right area of the top toolbar to verify the design for your modified Fixed-Point conversions. Again, once the process is complete, you should get a message saying Validation Succeeded.

4. Select Validate Types to verify the new design
5. Click on HDL Code Generation and modify the settings according to the previous labs. There is no pipelining required for this project. Right Click and select run this task to generate your Xilinx Block Box Design.
6. Once created, copy the black box and System generator Blocks to a new Model just as in the previous labs.
7. Save the new model as rx.slx into the sysgen directory “Lab_7\sysgen”.
8. Copy the `qpsk_rx_FixPt_xsgbbxcfg.m` file into your Sysgen folder just as in the previous labs.
9. Create the hdl folder inside the Sysgen folder and copy your vhd files into this directory. Make sure you modify the previously copied m file to point to the new location of the vhd files.

Export Simulink models Step 2

This section will show you which of the previously created Simulink Models will be used for this lab. It is assumed you have completed the previous labs and have already created the needed Simulink models.

2.1 Create MCU Simulink Design

The **Simulink model**[2] in Figure 2-1 will be used for the control signals to and from the **MCU**.

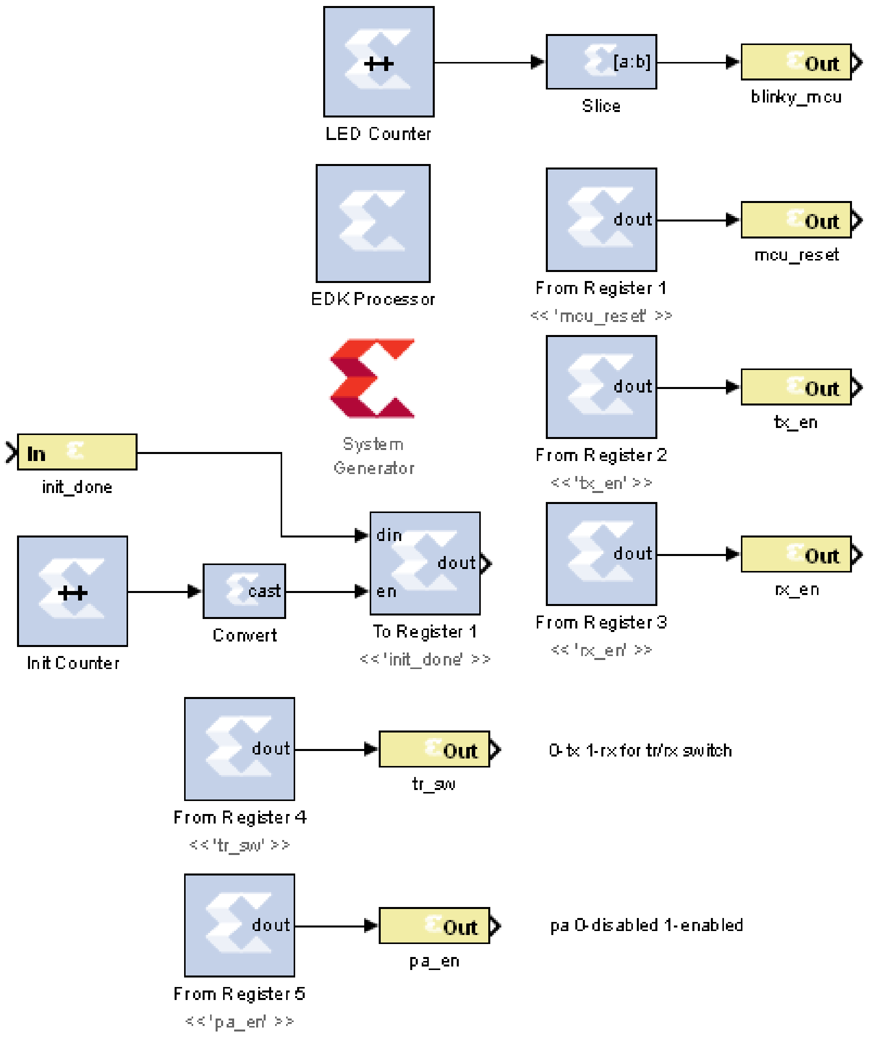


Figure 2-1: Simulink model for MCU control

[2] This model can be downloaded from https://github.com/Toyon/Chilipepper/tree/master/Labs/Lab_8/sysgen

1. **Configure** this model and the system generator the same as in Lab 1, and **save** the design into the Sysgen folder. Name the file **mcu.slx** or something similar.

2.2 Create Receiver Simulink Design

The **Simulink model**[3] In Figure 2-2 will be used for receiving the ADC output and decoding the QPSK waveform.

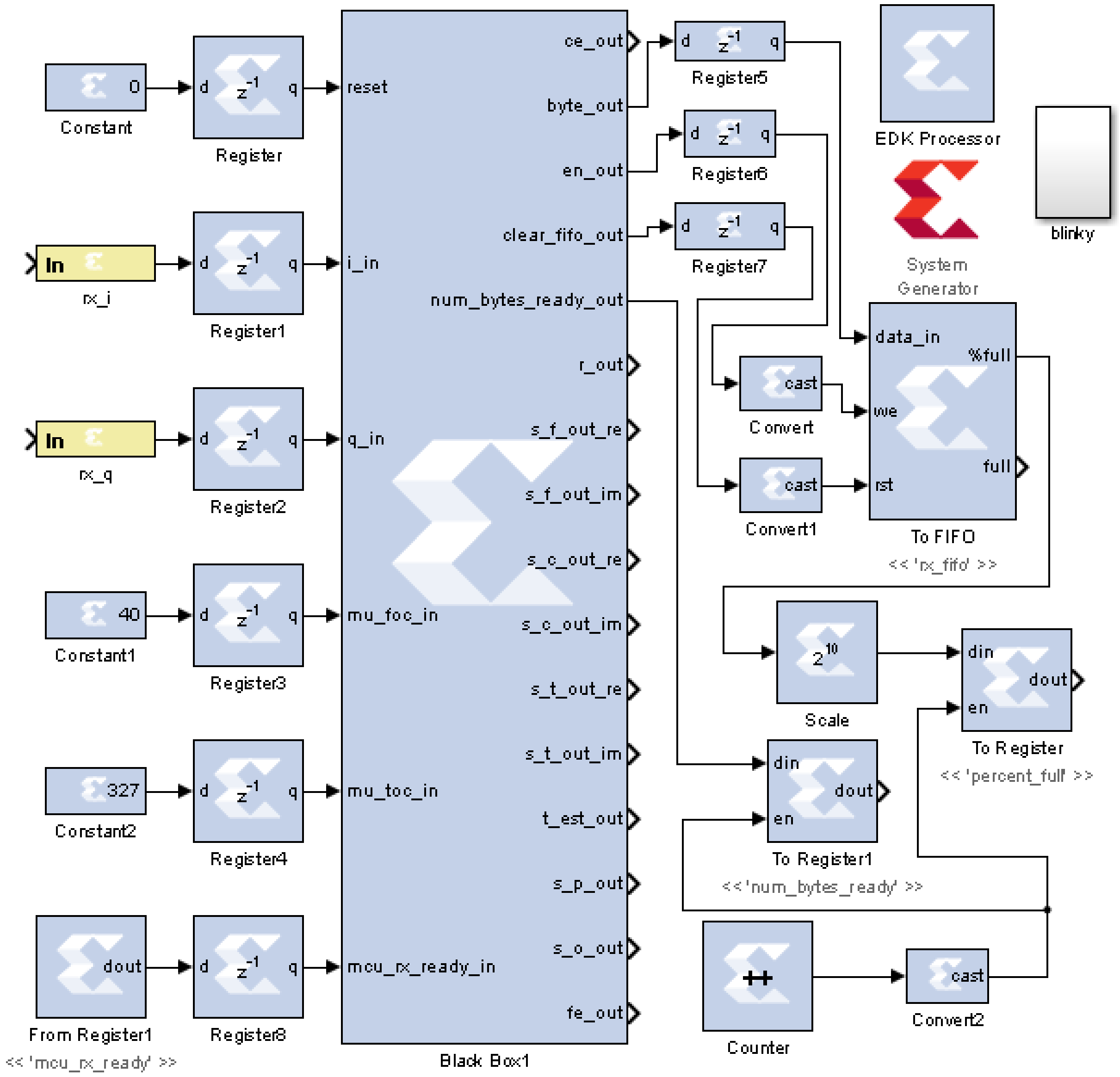


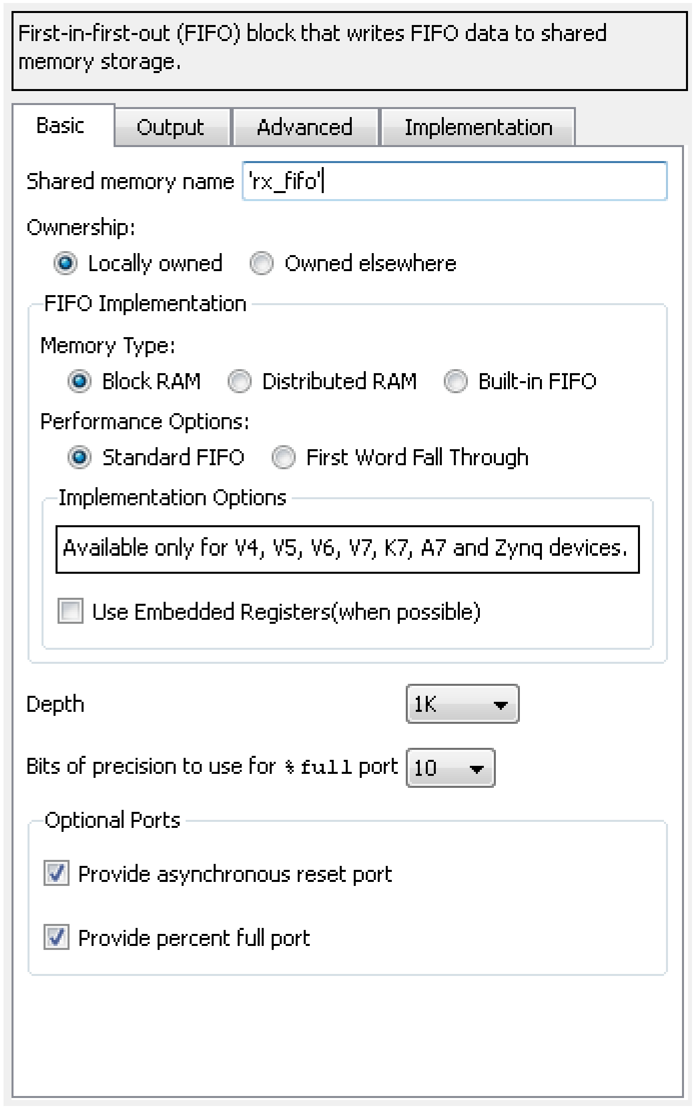
Figure 2-2: Simulink model for receiving DC Offset output

[3] This model can be downloaded from https://github.com/Toyon/Chilipepper/tree/master/Labs/Lab_8/sysgen

1. **Modify** the Simulink model `rx.slx` created earlier to look similar to Figure 2.2. The model is very similar to the one used in the previous labs.

2. The Convert blocks should all be configured as a Boolean output type.

3. The To FIFO should have the settings shown in Figure 2-3.


First-in-first-out (FIFO) block that writes FIFO data to shared memory storage.

Shared memory name 'rx_fifo'

Ownership: [✓ Locally owned] Owned elsewhere

FIFO Implementation

Memory Type: [✓ Block RAM] Distributed RAM Built-in FIFO

Performance Options: [✓ Standard FIFO] First Word Fall Through

Implementation Options

Available only for V4, V5, V6, V7, K7, A7 and Zynq devices.

☐ Use Embedded Registers(when possible)

Depth 1K

Bits of precision to use for % full port 10

Optional Ports

☑ Provide asynchronous reset port

☑ Provide percent full port

Figure 2-3: Settings for the To FIFO block

4. The scale factor block should be set to 10.

5. Setup the counter as an unsigned 1 bit single step size free running up counter so that it will simply toggle on and off.

6. **Save** the design into the Sysgen folder. **Be sure to move and modify the cfg file as well to find the files in your new directory structure.** Name the file **rx.slx** or something similar.

2.3 Create ADC driver Simulink Design

The **Simulink model**[4] In Figure 2-4 will be used for creating the signals which drive the **ADC** on Chilipepper.

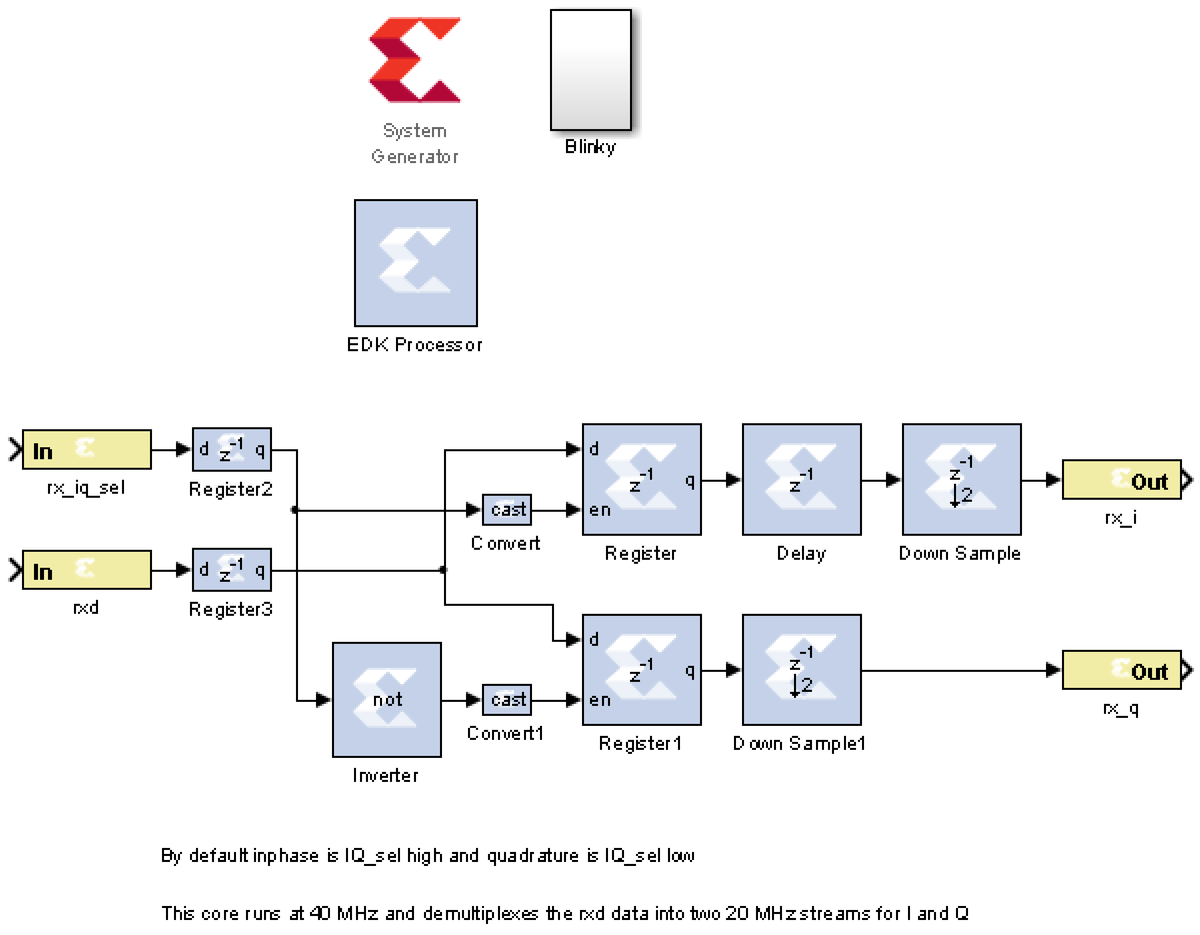


Figure 2-4: Simulink model for ADC control

1. **Create** a new Simulink model and add the components from the Simulink blockset.
2. The white box labeled “Blinky” is simply a subsystem of the **Counter Slice** and LED **Gateway Out** blocks. The Blocks used for this subsystem are shown in Figure 2-5. Configure the Blinky subsystem identically to the other LED out systems in the previous labs.

[4] This model can be downloaded from https://github.com/Toyon/Chilipepper/tree/master/Labs/Lab_8/sysgen

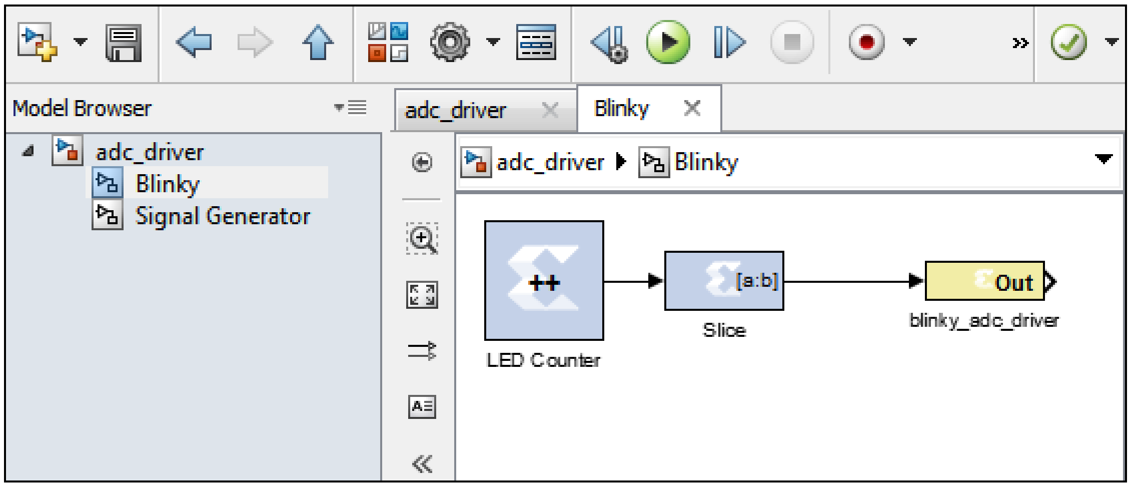


Figure 2-5: Blinky Subsystem

2.4 DC Offset Core

The last **Simulink model** [5]needed for this FPGA design is the DC Offset model created in a previous lab. For your reference this model is shown in Figure 2-6 below.

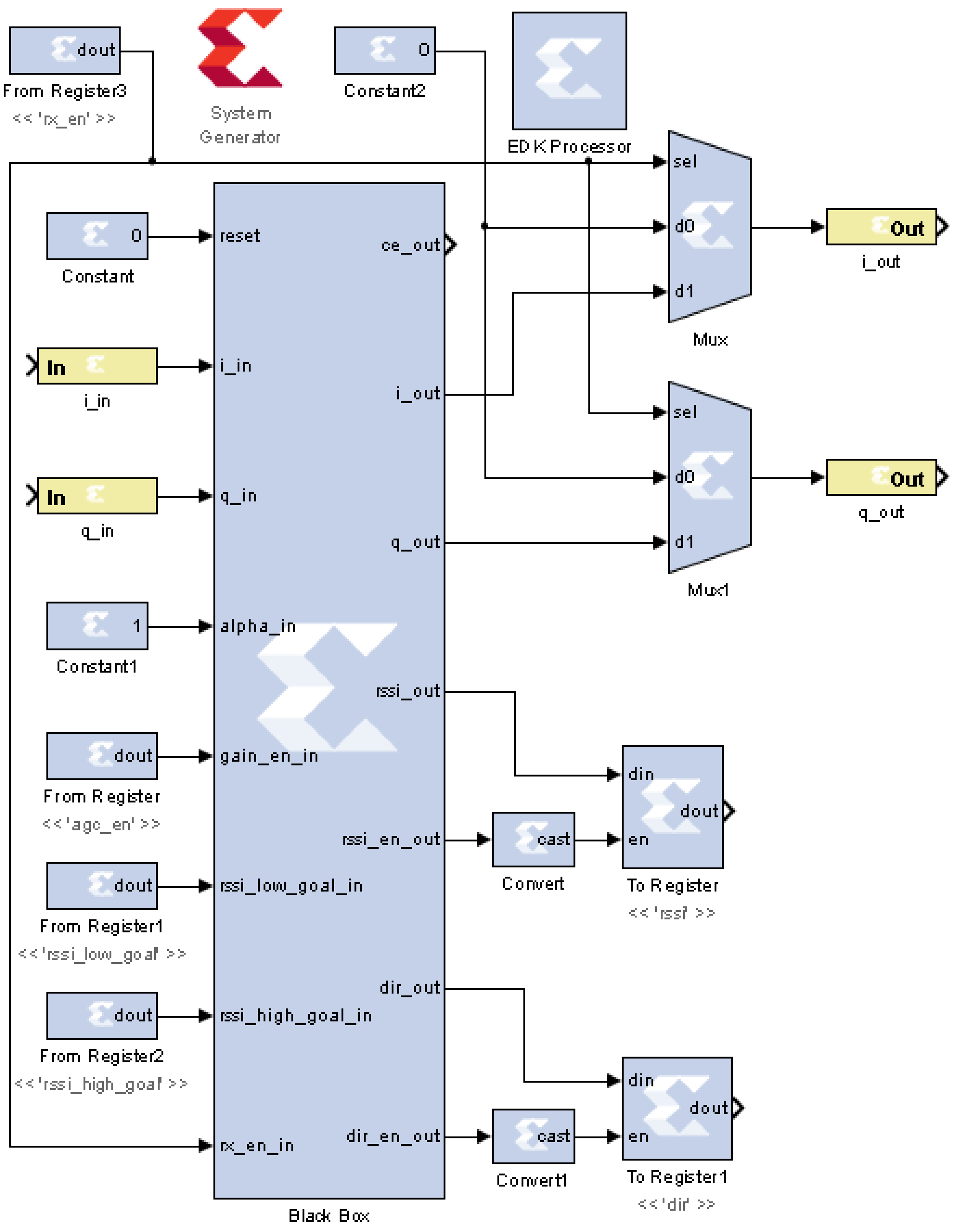


Figure 2-6: Simulink model for receiving ADC output and applying DC Offset Correction

[5] This model can be downloaded from https://github.com/Toyon/Chilipepper/tree/master/Labs/Lab_8/sysgen

1. Copy the `DC_Offset` Simulink files into your new project directory.

Note	Be sure to copy the vhd and config files to the new directory. You may use the Simulink files on the GitHub Repo as a reference. Additionally, you can open the old `DC_Offset` Simulink model and reconfigure its Pcore settings to point to the new EDK project directory, then recreate it.

Refer to Lab 0 Step 3 to **Create a New Blank EDK Project**. Be sure to follow the directory structure used. Once your project is created, **export** each model 1 by 1 into the newly created EDK project. Be sure your **Compilation Settings** are correct as shown in Lab 0 Section 4.1. Once each Simulink model has been exported successfully, you're ready to configure your FPGA design.

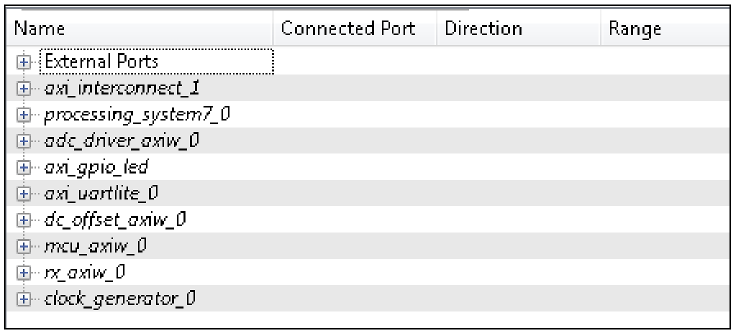
Configure Cores and Export Design Step 3

This section will show you how to integrate your PCores into your FPGA design using EDK. There are several components that must be configured for the design of this project. A quick list of the cores needed is given below. Refer to lab 0 sections 4.3 and 5.1 for information on how to add cores to the design.

3.1 Needed IP Cores

- ADC Driver PCore created in Simulink
- MCU PCore created in Simulink
- rx PCore created in Simulink
- DC Offset PCore created in Simulink
- Clock Generator IP Core
- Processing System IP Core
- AXI Interconnect IP Core
- GPIO Cores for LEDs
- AXI_UART (Lite) Core

In addition, several of these cores will require external ports. Be sure that you have access to modifying the external port settings. Refer to Figure 3-1 Below.



Name	Connected Port	Direction	Range
External Ports			
axi_interconnect_1			
processing_system7_0			
adc_driver_axiw_0			
axi_gpio_led			
axi_uartlite_0			
dc_offset_axiw_0			
mcu_axiw_0			
rx_axiw_0			
clock_generator_0			

Figure 3-1: EDK project ports list

3.2 Configuring the Pcore Ports

Each of the ports used in this lab can be configured exactly as the previous labs design. Refer to Lab 5 for more information on individual port configuration.

3.3 Pin Assignments

Once the ports are configured correctly, the sysgen clock for the cores should be set as well. The last step is to setup the **pin assignments** for the external ports.

1. Rename the pins of the external ports so they are easily identifiable. Figure 3-2 shows the names used in this demo, however you don't have to use the same naming convention.
2. Open the **Project** tab.
3. Double-click on the **UCF File: data\system.ucf** from this panel, to open the constraints file.
4. Fill in the pin out information for your design using Figure 3-2 below as a reference.

Be sure that the **orientation** of the `RXD` pins is set correctly. If you follow the pin list in the figure above, you must **reverse** the `RXD` pins in the external ports assignment section. This is done using the same method used in Lab 0 Section 5.2 for the LEDs.

5. Prior to EDK version 14.4, Xilinx had a documented issue[6] with AXI-bus generation for Simulink PCores targeting the Zynq FPGA. Refer to this issue for more information. As in Lab 0 section 5.2, this bug must be corrected if your **EDK version** is **14.3 or lower**. The steps to perform are identical to those in the previous labs; however they must be performed for **all** of the PCores used in this lab.
6. Select the **Export Design** button from the navigator window on the left. Click the **Export and Launch SDK** button. This process may take awhile.

[6] Issue can be found ay http://www.xilinx.com/support/answers/51739.htm

```
################################################## PL clocks and reset ##########################################
NET clock_generator_0_pll_pin                  LOC = D18   | IOSTANDARD = LVCMOS25;
NET clock_generator_0_pll_pin                  TNM_NET = clock_generator_0_pll;
TIMESPEC TS_clock_generator_0_pll = PERIOD clock_generator_0_pll 40.000 MHz;
############################################Rx – FMC interface at 2.5V ##########################################
NET clock_generator_0_rx_clk_pin               LOC = J18          | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET adc_driver_axiw_0_rx_iq_sel_pin            LOC = N19          | IOSTANDARD = LVCMOS25;
NET adc_driver_axiw_0_rxd_pin[0]               LOC =M21           | IOSTANDARD = LVCMOS25;
NET adc_driver_axiw_0_rxd_pin[1]               LOC = J21          | IOSTANDARD = LVCMOS25;
NET adc_driver_axiw_0_rxd_pin[2]               LOC = M22          | IOSTANDARD = LVCMOS25;
NET adc_driver_axiw_0_rxd_pin[3]               LOC = J22          | IOSTANDARD = LVCMOS25;
NET adc_driver_axiw_0_rxd_pin[4]               LOC = T16          | IOSTANDARD = LVCMOS25;
NET adc_driver_axiw_0_rxd_pin[5]               LOC = P20          | IOSTANDARD = LVCMOS25;
NET adc_driver_axiw_0_rxd_pin[6]               LOC = T17          | IOSTANDARD = LVCMOS25;
NET adc_driver_axiw_0_rxd_pin[7]               LOC = N17          | IOSTANDARD = LVCMOS25;
NET adc_driver_axiw_0_rxd_pin[8]               LOC = J20          | IOSTANDARD = LVCMOS25;
NET adc_driver_axiw_0_rxd_pin[9]               LOC = P21          | IOSTANDARD = LVCMOS25;
NET adc_driver_axiw_0_rxd_pin[10]              LOC = N18          | IOSTANDARD = LVCMOS25;
NET adc_driver_axiw_0_rxd_pin[11]              LOC = J16          | IOSTANDARD = LVCMOS25;
############################################### MCU Interface ################################################
NET clock_generator_0_tx_clk_pin               LOC = C17          | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
############################################### MCU Interface ################################################
NET axi_uartlite_0_RX_pin                      LOC = R19          | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET axi_uartlite_0_TX_pin                      LOC = L21          | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_axiw_0_mcu_reset_pin                   LOC = K20          | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_axiw_0_tx_en_pin                       LOC = D22          | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_axiw_0_tr_sw_pin                       LOC = D20          | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_axiw_0_rx_en_pin                       LOC = C22          | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_axiw_0_pa_en_pin                       LOC = E21          | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_axiw_0_init_done_pin                   LOC = K19          | IOSTANDARD = LVCMOS25;
#################################################### LEDs ######################################################
NET axi_gpio_led_GPIO_IO_pin                           LOC = T22  | IOSTANDARD=LVCMOS33;  # "LD0"
NET axi_gpio_led_GPIO2_IO_pin                          LOC = T21  | IOSTANDARD=LVCMOS33;  # "LD1"
NET mcu_axiw_0_blinky_mcu_pin                          LOC = U22  | IOSTANDARD=LVCMOS33;  # "LD2"
NET adc_driver_axiw_0_blinky_adc_driver_pin            LOC = U21  | IOSTANDARD=LVCMOS33;  # "LD3"
NET clock_generator_0_LOCKED_pin                       LOC = V22  | IOSTANDARD=LVCMOS33;  # "LD4"
NET rx_axiw_0_blinky_rx_pin                            LOC = W22  | IOSTANDARD=LVCMOS33;  # "LD5"
```

Figure 3-2: EDK project pin assignments

Create software project Step 4

Once the design is compiled and exported, you'll be greeted with a screen asking you where you would like to store your software project. It is very helpful to create the workspace folder in the same directory as your Sysgen and EDK folders. Doing this will keep all relevant files in the same location.

4.1 Creating a new C Project

This section will show you how to create a C program to test your FPGA design. The primary objectives of this program are to receive a QPSK waveform and communicate through a UART with the Terminal to output the received message.

Note	It would be helpful if you have completed the Embedded System Design tutorial in the *ZedBoard AP SoC Concepts Tools and Techniques Guide*. Refer to Lab 1 for more information on the MCU signal control using C code within SDK.

1. Select **File → New → Project**.
2. Select **Xilinx → Application Project,** and hit next.
3. Name the project qpsk_rx or something similar and leave the other settings at their defaults. Hit next.
4. Select **Hello World** from the **Select Project Template** section.
5. Click **Finish**. You should now see your hello_world.c file in your project src folder, as well as a **board support package** (bsp) folder. Feel free to rename this file to main.c or something more appropriate.
6. **Double click** the file to open it and **replace** all of its contents with the code in Appendix E.

If your SDK project cannot find your include files such as "xuartlite.h" and "xparameters.h" you can add these include files manually by selecting Xilinx Tools from the top menu and selecting Repositories. Under Global Repositories, click New and select your EDK project directory. See here [7]for more information

[7] See http://www.xilinx.com/support/answers/35443.htm

4.2 Adding Supporting files

In addition to the main c file, you need the library files for the Chilipepper board. The 2 required files for this Lab are Chilipepper.c and Chilipepper.h and can be found on the githib repo. Place these files in the src directory of your project workspace.

1. Chilipepper.c – This file is the primary library file for the Chilipepper board. It contains functions for modifying the MCU registers as well as basic helper functions for tasks such as initialization, transmitting, and receiving.
2. Chilipepper.h – This file holds the function prototypes for the Chilipepper.c functions.

Note

In addition to the Library files, you also need to include a Math library which contains the `pow` function that is used when creating the CRC. See Lab 3 section 4.1 for more information on how to add the Math Library to your project.

The Chilipepper.c library file is configured for both TX and RX cores as well as a UART to talk to the on board MCU and configure its settings. To use the library file properly, you must specify which of these features you will use. To do this, modify lines 8-13 of the Chilipepper.c file to define a variable for the cores you will be using. Your code should resemble the following, as we will be using all cores except the TX_DRIVER and DAC_DRIVER cores for this Lab.

```
#define MCU_UART
#define MCU_DRIVER
#define DC_OFFSET
//#define TX_DRIVER
//#define DAC_DRIVER
#define RX_DRIVER
```

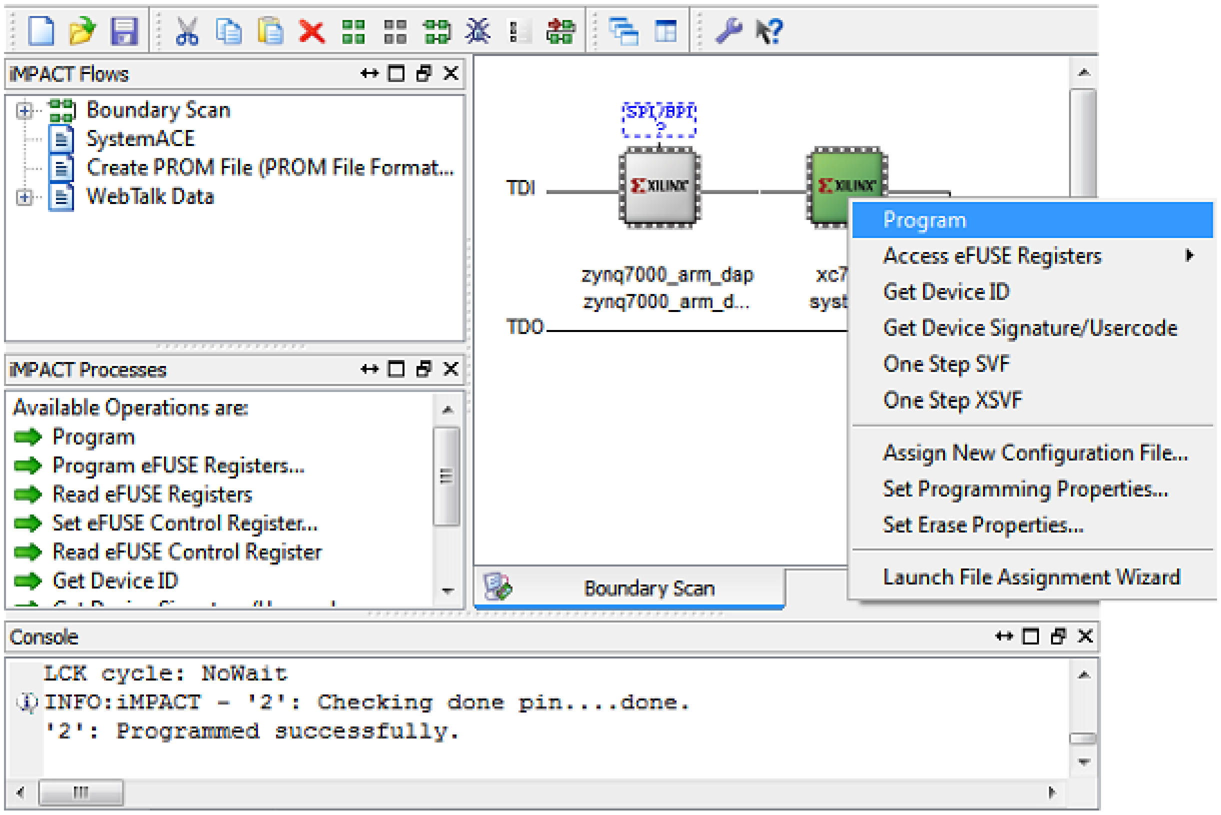
If you are still not able to compile your C design due to include errors, you may need to tell SDK where your PCore drivers are stored. If you click on Xilinx Tools → Repositories, you can specify (in Global Repositories) where the EDK directory of your project is. (This will need to be changed for each new project)

4.3 Loading Hardware Platform with iMPACT

Once your program is written and compiled you are ready to test the design! This is done by programming the FPGA with your hardware descriptions defined in the bit file generated in EDK, and running your software on top of this design. For this lab, you can verify your design by connecting two Chilipepper boards together using an attenuator. On one board, you should run the latest version of Lab 3 which allows you to either send a single packet via button press, or multiple packets using the switch on the FPGA.

1. Connect the Chilipepper to the FPGA board and verify all cables are connected properly and the jumper settings are correct. Verify this by using the *Chilipepper user guide* and the *ZED Board Hardware users guide* as a reference. Also See Lab 0 for details on Jumper Configuration.
2. Once the FPGA and radio board are connected correctly, turn on the board.
3. Open iMPACT in the ISE Design tools.
4. Select no if Impact asks you to load the last saved project.
5. Select yes to allow iMPACT to automatically create a new project for you. If you receive any connection errors, verify your USB or JTAG programmer cables are connected properly.
6. Select the Automatic option for the JTAG boundary scan setting and click ok.
7. Hit yes to assign configuration files. Bypass the first file selection, but for the second selection, browse to the location of your system.bit file. It should be inside the "Implementation" folder of your EDK project folder.
8. Select ok on the next screen verifying that the board displayed is your Zynq xc7z020 board. It should look similar to Figure 4-2 below.
9. Right click on the xc7z020 board icon (should be on the right), select program and hit ok.

Note	If you are running lab 3 from a second PC, you will need to repeat this process for the second board using the Lab 3 system.bit file. Alternatively, you can run Lab 3 directly from the SD card by loading a standard SD card with the Boot.bin file for lab 3, which can be found on the github repo.



```
LCK cycle: NoWait
INFO:iMPACT - '2': Checking done pin....done.
'2': Programmed successfully.
```

4-2: iMPACT configuration screen

To load Lab 3 via SD card:

1. Place the file on the SD card, and place the card inside the SD slot of the FPGA.
2. Configure the jumpers on the FPGA as shown in Figure 4-3.
3. Turn on the board, and the program should load after about 30 seconds. Check for the blue light, indicated the load was successful.

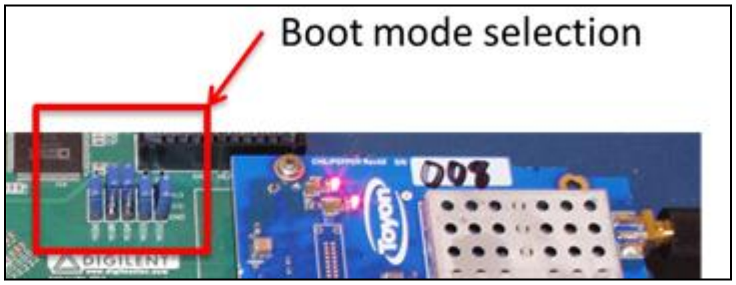


Figure 4-3: Jumper configuration needed to load a project via SD card

4.4 Debugging with SDK

If the hardware design is correct, you should see the LEDs start blinking on the board, as well as a blue light indicating the program was successful. You should also see the LED blinking on the second FPGA indicating the Lab 3 project is working properly. You can now return to the SDK project screen to test your software.

1. Test it by **right clicking** the project name folder and selecting **Debug As → Launch on Hardware**.
2. You should now be taken to a screen which shows the first pointer initialization as highlighted. You can now start the software program by clicking the ▶ (play) button in the top menu.

If the software initialization worked, you should see a green light on the Chilipepper as well as LED0 and LED1 blinking alternatively.

Testing and Design Verification Step 5

5.1 Verification with Terminal

Once you have both labs running successfully, the next step is to verify functionality by connecting the FPGA which is running your Lab 8 design to a terminal to view the received QPSK packet.

1. Connect your FPGA to the PC using a micro USB cable. The cable should be plugged into the UART port on the FPGA, shown in Figure 5-1 below.

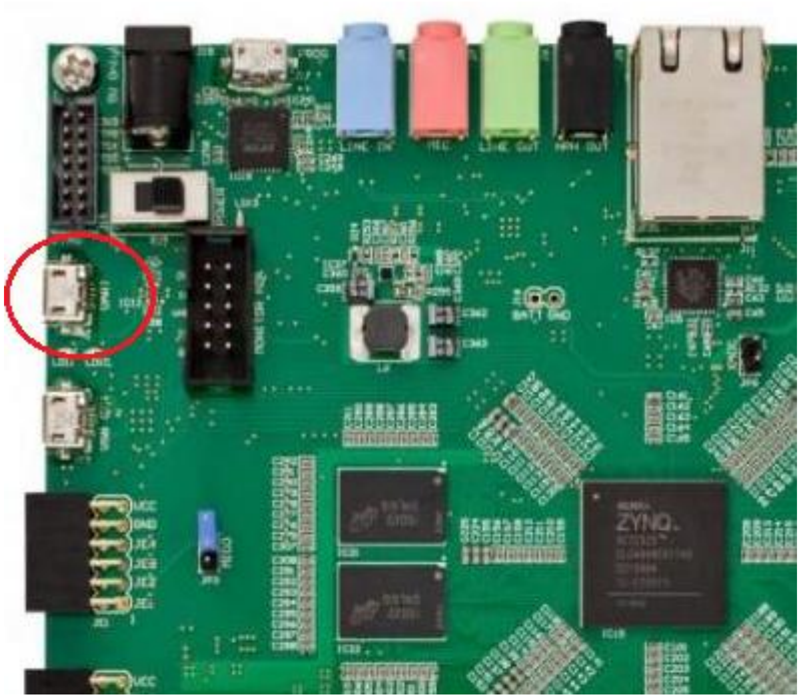


Figure 5-1: Circled in this figure is the UART port of the Xilinx Zed Board FPGA.

2. With the board powered on, open up a hyper terminal window such as Tera Term, and configure it with the following settings

   Baud: 115200; Data: 8 bit; No Parity;1 Stop bit; No Flow Conrol.

   Notice that the baud rate used is configured in the `SetupPeripherals` function in our main.c function created earlier. The other settings are all defaults for the XUartPs port.

3. Once the terminal is configured, you should be able to view your hello world packets by clicking the button on the Lab 3 Demo. In addition, if you flip the switch for continuous packet transmission, you shoud see several hello world packets on your terminal output.

Appendix A Core MATLAB rx Function

MATLAB function `qpsk_rx.m`

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% QPSK demonstration packet-based transceiver for Chilipepper
% Toyon Research Corp.
% http://www.toyon.com/chilipepper.php
% Created 10/17/2012
% embedded@toyon.com
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This file is the top level entry function for the receiver portion of the
% example. The entire receiver is designed to run at Rate=1 (one clock
% cycle per iteration of the core.
% We follow standard receive practice with frequency offset estimation,
% pulse-shape filtering, time estimateion, and correlation to determine
% tart of packet.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%#codegen

function [clear_fifo_out, num_bytes_ready_out, ...
          r_out, s_f_out, s_c_out, s_t_out, t_est_out, f_est_out, ...
          byte_out, en_out, s_p_out, s_o_out ] = ...
    qpsk_rx(i_in, q_in, mu_foc_in, mu_toc_in, mcu_rx_ready_in)

    persistent finish_rx_latch
    if isempty(finish_rx_latch)
        finish_rx_latch = 0; % feedback once packet is received to rest
    end

    % scale input data coming from the Chilipepper ADC to be purely
    % fractional to avoid scaling issues
    r_in = complex(i_in, q_in);

    % frequency offset estimation. Time constant is input as integer
    [s_f_i, s_f_q, fe] = qpsk_rx_foc(i_in, q_in, mu_foc_in, finish_rx_latch);

    % Square-root raised-cosine band-limited filtering
    [s_c_i, s_c_q] = qpsk_rx_srrc(s_f_i, s_f_q);

    % Time offset estimation. Output data changes at the symbol rate.
    [s_t_i, s_t_q, tauh] = qpsk_rx_toc(s_c_i, s_c_q, mu_toc_in, ...
        finish_rx_latch);

    % Determine start of packet using front-loaded training sequence
    [finish_rx, clear_fifo, num_bytes_ready_out, ...
     byte, en, s_p, s_o] = qpsk_rx_correlator(s_t_i, s_t_q, mcu_rx_ready_in);

    %correlator output. en notifies byte changes (change is @ OSRATE)
    byte_out = byte;
    en_out = en;
    % estimation and correlation values
    t_est_out = tauh;
    f_est_out = fe;
    s_p_out = s_p;
    s_o_out = s_o;

    % original signal out (real version)
    r_out = real(r_in);

    % incremental signal outputs after frequency estimation, filtering, and
    % timining estimation
    s_f_out = complex(s_f_i,s_f_q);
    s_c_out = complex(s_c_i,s_c_q);
    s_t_out = complex(s_t_i,s_t_q);

    %processor integration
    finish_rx_latch = finish_rx;
    clear_fifo_out = clear_fifo;
end
```

Appendix B MATLAB Frequency Offset Correction

MATLAB function `qpsk_rx_foc.m`

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% QPSK demonstration packet-based transceiver for Chilipepper
% Toyon Research Corp.
% http://www.toyon.com/chilipepper.php
% Created 10/17/2012
% embedded@toyon.com
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Demonstration of a Costas Loop. Refer to:
% Telecommunications Breakdown: Concepts of Communication Transmitted via
% Software-Defined Radio C. Richard Johnson
% We employ a hard-decision feedback in order to get rid of the loop
% filters.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%#codegen
function [z_i_out, z_q_out, fe] = qpsk_rx_foc(y_i, y_q, mu_in, finish_rx)

persistent phi

lSin = SIN;
lCos = COS;

if isempty(phi)
    phi = 0;
end

mu = mu_in/2^12;

if finish_rx == 1
    phi = 0;
end

% create the VCO signal
if phi >= 1
    phi = phi - 1;
end
if phi < 0
    phi = phi + 1;
end

phi12 = round(phi*2^12)+1;
if phi12 >= 2^12
    phi12 = 1;
end
if phi12 < 0
    phi12 = 0;
end
```

```
f_i = lCos(phi12+1);
f_q = lSin(phi12+1);
ti1 = y_i*f_i;
ti2 = y_q*f_q;
tq1 = y_q*f_i;
tq2 = -y_i*f_q;
z_i = ti1 + ti2;
z_q = tq1 + tq2;

% generate the error term to drive VCO generateion
if z_q < 0
    tf = -z_i;
else
    tf = z_i;
end
if z_i < 0
    bf = -z_q;
else
    bf = z_q;
end
% using sign of error in order to make it gain invariant
time_diff = tf-bf;
if time_diff < 0
    e = -1;
else
    e = 1;
end

c = mu*e;
phiNew = phi - c;
phi = phiNew;

fe = phiNew;

z_i_out = z_i;
z_q_out = z_q;
```

Appendix C MATLAB Timing Offset Correction

MATLAB function `qpsk_rx_toc.m`

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% QPSK demonstration packet-based transceiver for Chilipepper
% Toyon Research Corp.
% http://www.toyon.com/chilipepper.php
% Created 10/17/2012
% embedded@toyon.com
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%#codegen
function [s_i, s_q, tauh] = qpsk_rx_toc(r_i, r_q, mu_in, finish_rx)

persistent counter
persistent tau
persistent rBuf_i rBuf_q
persistent symLatch_i symLatch_q
persistent tEst

OS_RATE = 8;
if isempty(counter)
    counter = 0;
    tau = 0;
    rBuf_i = zeros(1,4*OS_RATE);
    rBuf_q = zeros(1,4*OS_RATE);
    symLatch_i = 0; symLatch_q = 0;
    tEst = 0;
end

mu = mu_in/2^12;

if finish_rx == 1
    tau = 0;
end

rBuf_i = [rBuf_i(2:end) r_i];
rBuf_q = [rBuf_q(2:end) r_q];

if counter == 0
    taur = round(tau);
    % if we shift out of the window just exit
    if abs(taur) >= OS_RATE
        tau = 0;
        taur = 0;
    end
    % Determine lead/lag values and compute offset error
    zl_i = rBuf_i(2*OS_RATE+taur-1);
    zo_i = rBuf_i(2*OS_RATE+taur);
    ze_i = rBuf_i(2*OS_RATE+taur+1);
    zl_q = rBuf_q(2*OS_RATE+taur-1);
    zo_q = rBuf_q(2*OS_RATE+taur);
    ze_q = rBuf_q(2*OS_RATE+taur+1);
    od_r = ze_i-zl_i;
    od_i = ze_q-zl_q;
    oe_r = zo_i*od_r;
    oe_i = zo_q*od_i;
    % using sign of error in order to make gain invariant
    os = oe_r+oe_i;
    if os < 0
        oe = -1;
    else
        oe = 1;
    end

    % update tau
    tau = tau + mu*oe;

    tEst = tau;

    symLatch_i = zo_i;
    symLatch_q = zo_q;
end

s_i = symLatch_i;
s_q = symLatch_q;
tauh = tEst;

counter = counter + 1;
if counter >= OS_RATE
    counter = 0;
end
```

Appendix D MATLAB Correlation Function

MATLAB function `qpsk_rx_correlation.m`

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% QPSK demonstration packet-based transceiver for Chilipepper
% Toyon Research Corp.
% http://www.toyon.com/chilipepper.php
% Created 10/17/2012
% embedded@toyon.com
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% There are two major goals with this core. The first is to find the peak
% of the training sequence and then to subsequently pull out and pack the
% bits. The number of bytes transmitted is in the packet so we extract this
% to determine how many bytes to pull out.
% The second goal is to send these bytes off to the Microblaze processor.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%#codegen
function [reset_out, clear_fifo_out, num_bytes_ready_out, byte_out, ...
    en_out, s_out, o_out] = ...
    qpsk_rx_correlator(s_i_in, s_q_in, mcu_rx_ready_in)

    persistent counter
    persistent sBuf_i sBuf_q
    persistent oLatch sLatch
    persistent q detPacket
    persistent ip op
    persistent bits symCount byteCount numBytes
    persistent numBytesReady
    persistent mcuHasResetThisCore

    t_i = TB_i;
    t_q = TB_q;
    OS_RATE = 8;
    BIT_TO_BYTE = [1 2 4 8 16 32 64 128]';

    if isempty(counter)
        counter = 0;
        sBuf_i = zeros(1,65);
        sBuf_q = zeros(1,65);
        sLatch = 0;
        oLatch = 0;
        q = 0;
        detPacket = 0;
        ip = 0; op = 0;
        bits = zeros(1,8);
        symCount = 0;
        byteCount = 0;
        numBytes = 1000;
        numBytesReady = 0;
        mcuHasResetThisCore = 0;
    end
    if mcu_rx_ready_in == 0
        numBytesReady = 0;
        mcuHasResetThisCore = 1;
    end

    clear_fifo_out = 0;
    reset_out = 0;
    byte_out = 0;
    en_out = 0;

    % found a packet, now we're ready to write the data out
    if counter == 0 && detPacket == 1
        mcuHasResetThisCore = 0; % don't go high again until MCU 1->0->1
        if s_i_in < 0
            sHard_i_t = -1;
        else
            sHard_i_t = 1;
        end
        if s_q_in < 0
            sHard_q_t = -1;
        else
            sHard_q_t = 1;
        end
        sHard_i = 0; sHard_q = 0;
        switch q
            case 0
                sHard_i = sHard_i_t;
                sHard_q = sHard_q_t;
            case 1
                sHard_i = sHard_q_t;
                sHard_q = -sHard_i_t;
            case 2
                sHard_i = -sHard_i_t;
                sHard_q = -sHard_q_t;
            case 3
                sHard_i = -sHard_q_t;
                sHard_q = sHard_i_t;
        end
        sLatch = sHard_i;
        oLatch = 1;
        bits(symCount*2+1) = (sHard_i+1)/2;
        bits(symCount*2+2) = (sHard_q+1)/2;

        symCount = symCount + 1;
        if symCount >= 4
            byteCount = byteCount + 1;
            symCount = 0;
            byte_out = bits*BIT_TO_BYTE;
            en_out = 1;
            % first byte is number of bytes in payload
            if byteCount == 1
                numBytes = byte_out;
            end
            % if we exceed the packet ID
            if byteCount > 3
                % exit if we've written all the bytes or above reasonable
                % threshold
                if byteCount == numBytes+6 || byteCount > 256
                    detPacket = 0;
                    counter = 1;
                    reset_out = 1;
                    numBytesReady = numBytes+6;
                end
            end
        end
    end

    % let's see if we can find a packet. only if MCU is ok to rcv packet
    if counter == 0 && detPacket == 0 && ...
            mcu_rx_ready_in == 1 && mcuHasResetThisCore == 1
        sLatch = 0;
        if s_i_in < 0
            ss_i = -1;
        else
            ss_i = 1;
        end
        if s_q_in < 0
            ss_q = -1;
        else
            ss_q = 1;
        end

        sBuf_i = [sBuf_i(2:end) ss_i];
        sBuf_q = [sBuf_q(2:end) ss_q];

        sc_iWithi = sBuf_i*t_i;
        sc_iWithq = sBuf_i*t_q;
        sc_qWithi = sBuf_q*t_i;
        sc_qWithq = sBuf_q*t_q;

        ip = abs(sc_iWithi)+abs(sc_qWithq);
        op = abs(sc_iWithq)+abs(sc_qWithi);

        % we found a packet. While we have frequency offset lock we don't
        % know the phase offset. Here we use the inphase and quadrature
        % phasing to determine how to rotate around the circle
        if ip > 100 % 0 or 180 angle
            if sc_iWithi > 10 && sc_qWithq > 10
                q = 0; % 0 degrees
            else
                q = 2; % 180 degrees;
            end
            detPacket = 1;
            % we don't really need to do this as it should be empty, but for
            % good measure...
            clear_fifo_out = 1;
        end
        if op > 100
            if sc_iWithq > 10 && sc_qWithi < 10
                q = 3; % 90 degrees
            else
                q = 1; % 270 degrees;
            end
            detPacket = 1;
            % we don't really need to do this as it should be empty, but for
            % good measure...
            clear_fifo_out = 1;
        end
        oLatch = ip+op;
        symCount = 0;
        byteCount = 0;
        numBytes = 1000;
    end

    num_bytes_ready_out = numBytesReady;
    s_out = sLatch;
    o_out = oLatch;

    % only pull data once every OS_RATE clocks
    counter = counter + 1;
    if counter >= OS_RATE
        counter = 0;
    end
end
```

Appendix E main.c

SDK function `main.c`

```
#include <stdio.h>
#include "platform.h"
#include "xbasic_types.h"
#include "xgpio.h"
#include "xparameters.h"
#include "xstatus.h"
#include "chilipepper.h"
#include "xuartps.h"
#include "xil_printf.h"
#include "xscugic.h"
#include "xil_exception.h"

XGpio gpio_blinky;
XUartPs uartPs;
XUartPs_Config *pUartPsConfig;

int DebouncButton( void );
int SetupPeripherals( void );
void WriteLedGain( int gain );

int main()
{
      int gain;
    int aliveLed = 0;
      int sentCount;
      int numBytes;
    static int BlinkCount = 0;
    unsigned char id;
    unsigned char curValue;
    unsigned char rxBuf[256];

    init_platform();
    if(SetupPeripherals() != XST_SUCCESS)
      return -1;

    if ( Chilipepper_Initialize() != 0 )
      return -1;

    Chilipepper_SetPA( 1 );
    Chilipepper_SetTxRxSw( 1 ); // 0- transmit, 1-receive

    while (1)
    {
      gain = Chilipepper_ControlAgc(); //update the Chilipepper AGC
            // main priority is to parse OTA packets
            numBytes = Chilipepper_ReadPacket( rxBuf, &id );

            if (numBytes > 0)
                  XGpio_DiscreteWrite(&gpio_blinky, 1, 1);
            // This is a normal receive situation.
            // We get a packet, write it to UART.
            if (numBytes > 0)
            {
                  sentCount = 0;
                  while (sentCount < numBytes)
                  {
                        curValue = rxBuf[sentCount+4];
                        sentCount += XUartPs_Send(&uartPs, &curValue, 1);
                  }
            }

            // flip the LED1 so the user knows the processor is alive
         if (BlinkCount > 500000)
         {
             if (aliveLed == 0)
                  aliveLed = 1;
             else
                  aliveLed = 0;
             BlinkCount = 1;
             XGpio_DiscreteWrite(&gpio_blinky, 2, aliveLed);  //blink LED
         }
    }
    cleanup_platform();
    return 0;
}

int SetupPeripherals( void )
{
      int status;

    XGpio_Initialize(&gpio_blinky, XPAR_AXI_GPIO_LED_DEVICE_ID);
    XGpio_SetDataDirection(&gpio_blinky, 2, 0);
    XGpio_SetDataDirection(&gpio_blinky, 1, 0);
      XGpio_DiscreteWrite(&gpio_blinky, 1, 0);
      XGpio_DiscreteWrite(&gpio_blinky, 2, 0);

      //Setup UART for serial port communication
    pUartPsConfig = XUartPs_LookupConfig(XPAR_PS7_UART_1_DEVICE_ID);
      if (NULL == pUartPsConfig) {
            return XST_FAILURE;
      }
      status = XUartPs_CfgInitialize(&uartPs, pUartPsConfig, pUartPsConfig-
>BaseAddress);
      if (status != XST_SUCCESS) {
            return XST_FAILURE;
      }
      XUartPs_SetBaudRate(&uartPs, 115200);

      return XST_SUCCESS;
}
```