Toyon Research Corporation

Lab 7: Correlation Decoding

Chilipepper Tutorial Projects

Version 1.0
1/25/2014

Table of Contents

Lab 7: Correlation Decoding

Introduction

This lab will extend the previous labs and allow you to decode the received signal to extract the original message. The Analog to Digital Conversion (ADC) used to receive the signal will take place on the Chilipepper board. The FMC initialization and microcontroller (MCU) signal control will be handled in software using the Xilinx Software Development Kit (SDK). Finally, verification of the received signal will be done using ChipScope and MATLAB. This lab assumes prior knowledge of the workings of HDL Coder as well as the Xilinx EDK environment. It is recommended that you complete the previous labs before completing this lab.

This lab is created using:

- MATLAB 2014a
- Xilinx ISE Design Suite 14.7
- 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:

- Generate HDL code from MATLAB functions
- Generate an IP core using MATLAB HDL Coder
- 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:

- Implement Correlation Decoding for a received QPSK Waveform
- Receive a QPSK Waveform using the Chilipepper FMC
- 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 which are required to export your design into EDK.

1.1 Supplemental PCores

As in the previous receiver tutorials, this lab will make use of the MCU, ADC and DC Offset PCores designed in earlier labs. Since these cores have already been created, we can copy the core design into our EDK project without having to recreate the HDL Coder project.

1.2 QPSK_RX

The QPSK RX design in this lab adds a fourth component to the previously created three stage receive path. The correlation decoding is required to properly determine the start of the data packet and translate those symbols into bits. This is done by first finding the Kasami sequence sent at the beginning of the transmitted waveform and aligning it properly to find the number of bytes in the payload. Just as in the previous receiver lab, we will split each of these four steps into its own MATLAB function. Each of these functions will then be called sequentially by a central function, called qpsk_rx.m. The contents of the qpsk_rx function are shown in Figure 1-1 below.

```
function [byte, blinky, s_out, o_out] = qpsk_rx(i_in, q_in)

persistent finish_rx_latch
persistent blinky_cnt

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

% frequency offset estimation.
[s_f_i, s_f_q] = qpsk_rx_foc(i_in, q_in, finish_rx_latch);

% Square-root raised-cosine band-limited filtering and time offset estimation
[s_c_i, s_c_q] = qpsk_rx_srrc(s_f_i, s_f_q);
[s_t_i, s_t_q] = qpsk_rx_toc(s_c_i, s_c_q, finish_rx_latch);

% Determine start of packet using front-loaded training sequence
[byte, finish_rx, s_out, o_out] = qpsk_rx_correlator(s_t_i, s_t_q);

blinky_cnt = blinky_cnt + 1;
if blinky_cnt == 20000000
    blinky_cnt = 0;
end
blinky = floor(blinky_cnt/10000000);
finish_rx_latch = finish_rx;
```

Figure 1-1: MATLAB function to analyze received signal.

1. Create a directory for the project under C:\QPSK_Projects\Lab_7.
2. Create a MATLAB directory within the main project directory.
3. Create a new **MATLAB function** with the contents of Figure 1-1.
4. Save this function as `qpsk_rx.m` inside the MATLAB directory.

As you can see from Figure 1 above, there have been some slight modifications to the previously created functions for frequency and timing offset estimation. After correlation decoding has been performed, the algorithm uses the knowledge about when the packet ends to assist the frequency and timing estimation functions by resetting their estimates. This is not required, but can assist in removing false positive hits within these cores. The new qpsk_rx_foc.m and qpsk_rx_toc.m functions are shown in Appendix A and B respectively.

5. Create a new **MATLAB function** with the contents of Figure A.
6. Save this function as `qpsk_rx_foc.m` inside the MATLAB directory.
7. Create a new **MATLAB function** with the contents of Figure B.
8. Save this function as `qpsk_rx_toc.m` inside the MATLAB directory.

The function which performs the correlation is called `qpsk_rx_correlator`.m. The code required to create the function can be found in Appendix C.

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

1.3 MATLAB Test Bench

Now that you have added functionality to the receiver core, we also need to modify the test bench script a bit to accommodate the new output. For this lab, the primary output is the payload bytes. Therefore we should be able to fully observe the transmitted message directly within ChipScope. In addition to the payload bytes, we will observe the correlation magnitude which is used to determine the start of the packet, and the message bits waveform, which is a nrz signal that represents each bit of the message. The only code changes required are to the variable names of the output from qpsk_rx. Just as in the previous labs, a simulated transmit waveform is required to fully test the design. Therefore, this script will require several of the MATLAB functions used in Lab 3 to transmit the QPSK waveform. A quick list of the needed files to create the simulated waveform is shown below. The code for the test bench script can be found in Appendix D.

Required files for creating Simulated QPSK waveform

- `make_srrc_lut.m and make_trig_lut.m`
- `CreateAppend16BitCRC.m`
- `tx_fifo.m`
- `qpsk_tx.m`
- `qpsk_tx_byte2sym.m`
- `qpsk_srrc.m`
- `mybitget.m`
- `TB_i.m and TB_q.m`

1. Create a new **MATLAB script** with the contents of Appendix D.

2. Save this function as qpsk_tb`.m` inside the MATLAB project directory.

Observing the output of the simulation results in Figure 1-2 below shows that the correlation magnitude reaches its maximum value when the Kasami sequence is detected. Additionally, the message bits contain the ASCII representation of the transmitted message `'hello world!'`

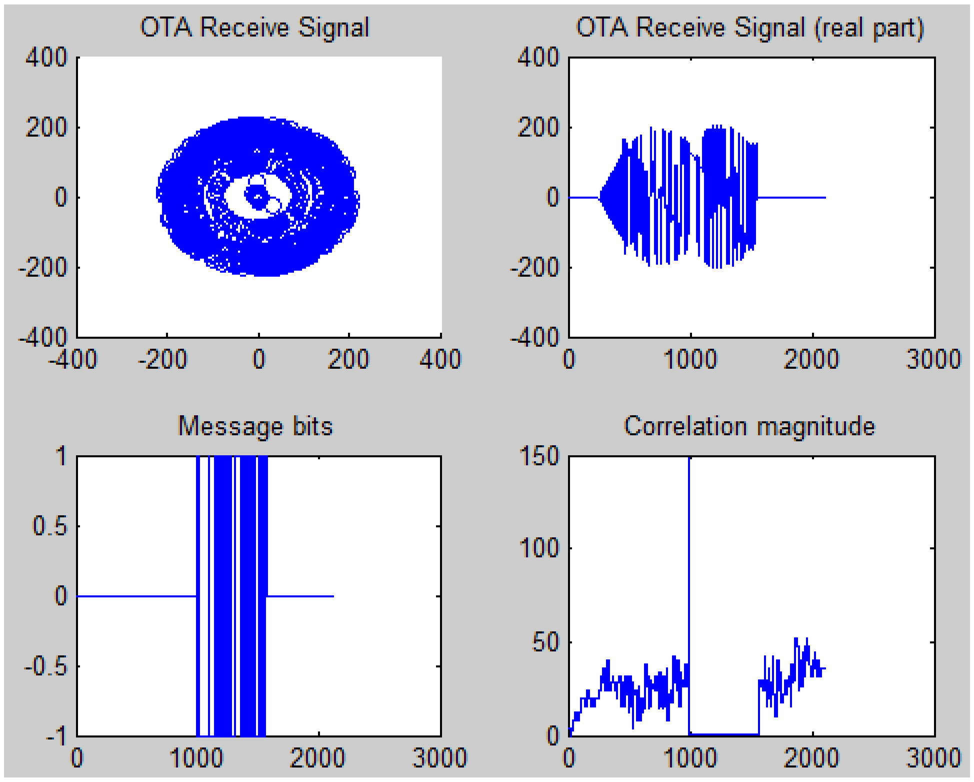


Figure 1-2: Output of QPSK Correlation decoding simulation in MATLAB test bench script

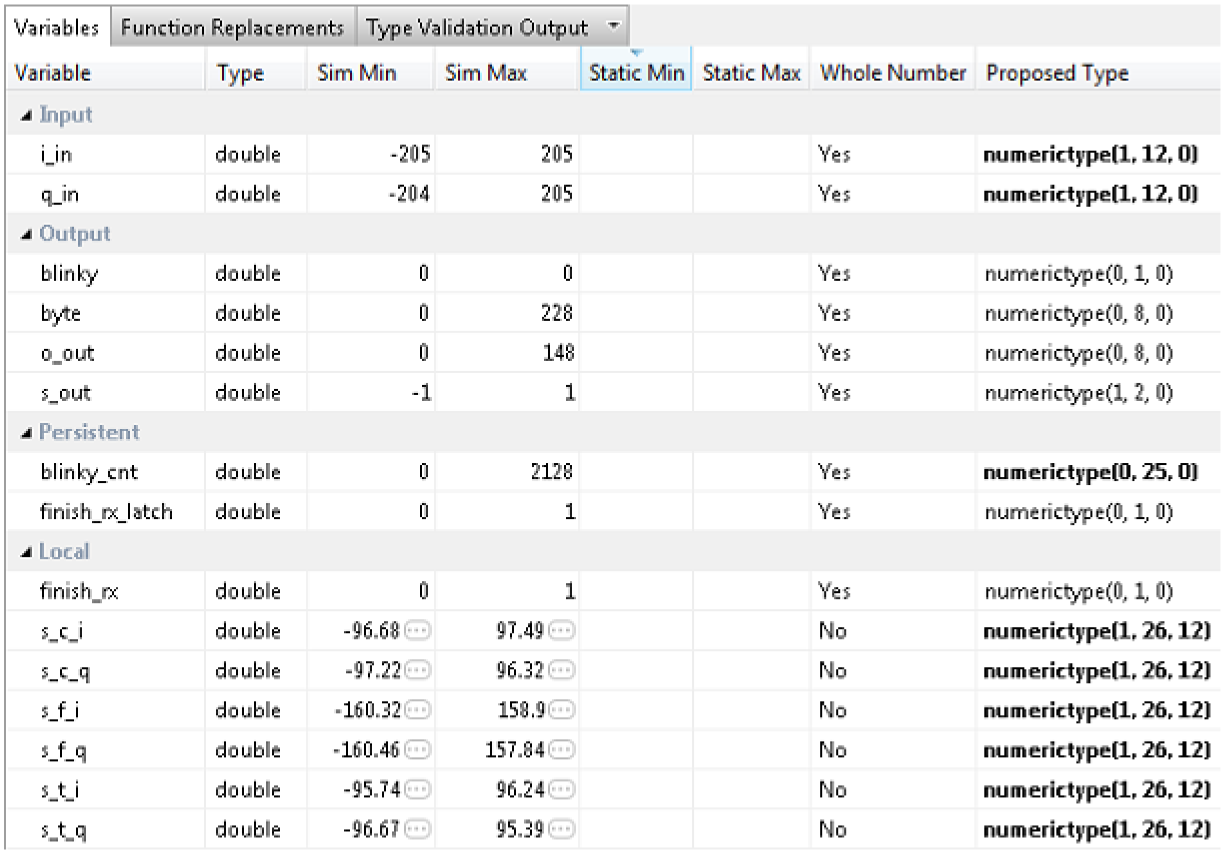
1.4 HDL Coder Project

Now that the MATLAB files have been created, we can turn them into PCores. As mentioned earlier, we will reuse the previously created DC Offset, MCU and ADC Driver PCores, thus the only core we need to create for this lab is the qpsk_rx PCore. Using the same steps outlined in the previous labs, create a new HDL coder project called `qpsk_rx`. Add both your `qpsk_rx.m` file and your `qpsk_rx_tb.m` files to the **MATLAB Function** and **MATLAB Test Bench** categories respectively.

1. Once inside the workflow advisor screen, click on **HDL Code Generation** on the left hand side, and be sure to set the clock to be driven at the **DUT base rate** as in the previous labs.

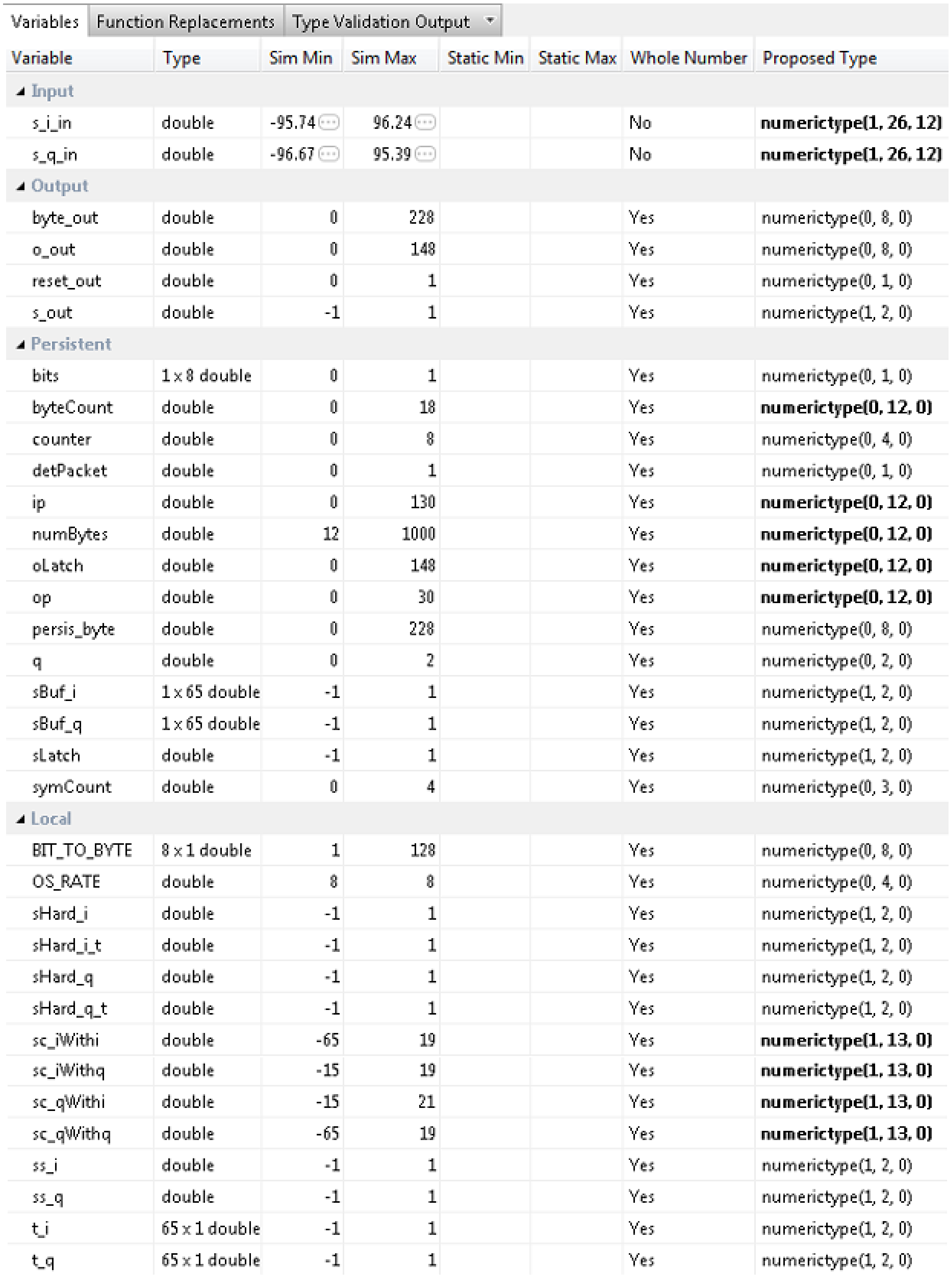
2. Right-click **Fixed-Point Conversion,** and select **Run to Selected Task.**

3. The qpsk_rx.m and qpsk_rx_correlator.m, functions both require modifications to their variable's proposed types. Modify your HDL Coder design to match the following Fixed-Point conversions for each function.



Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
i_in	double	-205	205			Yes	**numerictype(1, 12, 0)**
q_in	double	-204	205			Yes	**numerictype(1, 12, 0)**
Output							
blinky	double	0	0			Yes	numerictype(0, 1, 0)
byte	double	0	228			Yes	numerictype(0, 8, 0)
o_out	double	0	148			Yes	numerictype(0, 8, 0)
s_out	double	-1	1			Yes	numerictype(1, 2, 0)
Persistent							
blinky_cnt	double	0	2128			Yes	**numerictype(0, 25, 0)**
finish_rx_latch	double	0	1			Yes	numerictype(0, 1, 0)
Local							
finish_rx	double	0	1			Yes	numerictype(0, 1, 0)
s_c_i	double	-96.68	97.49			No	**numerictype(1, 26, 12)**
s_c_q	double	-97.22	96.32			No	**numerictype(1, 26, 12)**
s_f_i	double	-160.32	158.9			No	**numerictype(1, 26, 12)**
s_f_q	double	-160.46	157.84			No	**numerictype(1, 26, 12)**
s_t_i	double	-95.74	96.24			No	**numerictype(1, 26, 12)**
s_t_q	double	-96.67	95.39			No	**numerictype(1, 26, 12)**

Figure 1-3: Proposed variable types for qpsk_rx function



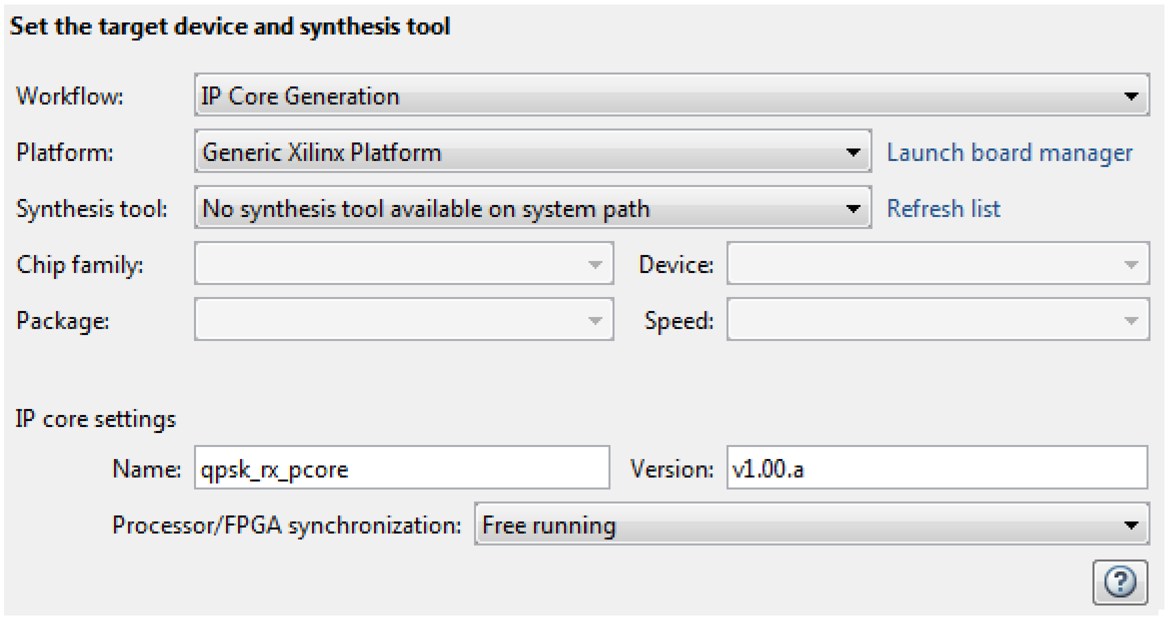
Variables | Function Replacements | Type Validation Output

Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
Input							
s_i_in	double	-95.74	96.24			No	**numerictype(1, 26, 12)**
s_q_in	double	-96.67	95.39			No	**numerictype(1, 26, 12)**
Output							
byte_out	double	0	228			Yes	numerictype(0, 8, 0)
o_out	double	0	148			Yes	numerictype(0, 8, 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	18			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	130			Yes	**numerictype(0, 12, 0)**
numBytes	double	12	1000			Yes	**numerictype(0, 12, 0)**
oLatch	double	0	148			Yes	**numerictype(0, 12, 0)**
op	double	0	30			Yes	**numerictype(0, 12, 0)**
persis_byte	double	0	228			Yes	numerictype(0, 8, 0)
q	double	0	2			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	-65	19			Yes	**numerictype(1, 13, 0)**
sc_iWithq	double	-15	19			Yes	**numerictype(1, 13, 0)**
sc_qWithi	double	-15	21			Yes	**numerictype(1, 13, 0)**
sc_qWithq	double	-65	19			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)

Figure 1-4: Proposed variable types for qpsk_rx_correlator function

The Proposed variable types for `qpsk_rx_foc.m`, `qpsk_rx_toc.m`, and `qpsk_rx_srrc.m` should be set to the same types used in the previous labs. Refer to Lab 6 for help configuring these functions.

4. Once you have corrected the **Type** setting for all your variables, click **Select Code Generation Target**. Here you can select the FPGA you will use for your design. For this Lab, we will not be using any of the built-in Zynq board functionality within our MATLAB PCores. Therefore you can leave the default settings. Ensure your Workflow settings resemble figure 1-5 below


Set the target device and synthesis tool

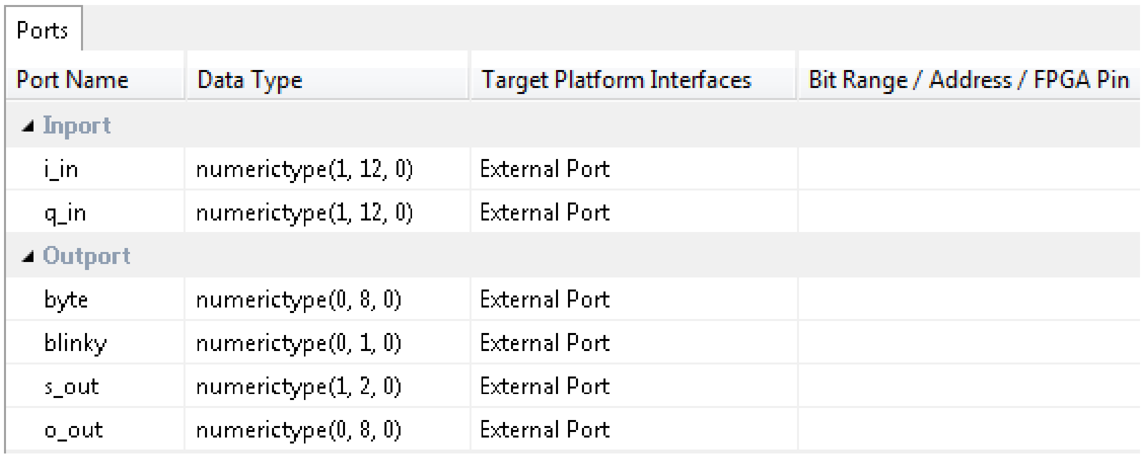
Workflow: IP Core Generation
Platform: Generic Xilinx Platform
Synthesis tool: No synthesis tool available on system path
Chip family:
Device:
Package:
Speed:

IP core settings
Name: qpsk_rx_pcore
Version: v1.00.a
Processor/FPGA synchronization: Free running

1-5: Settings for Xilinx Zed Board HDL Coder Design

5. Just below the synthesis tool settings, **rename your PCore** to `qpsk_rx_pcore` or something similar. This is optional as MATLAB will give its default name for each of your cores, as well as a default version, however it is helpful to rename your core for easier netlist configuration later in the lab.

6. Once the platform and synthesis tool are set, you can click **Set Target Interface** to configure the input and output ports of the design. For this Lab, follow the settings shown in Figure 1-6 below.



Ports

Port Name	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
◢ Inport			
i_in	numerictype(1, 12, 0)	External Port	
q_in	numerictype(1, 12, 0)	External Port	
◢ Outport			
byte	numerictype(0, 8, 0)	External Port	
blinky	numerictype(0, 1, 0)	External Port	
s_out	numerictype(1, 2, 0)	External Port	
o_out	numerictype(0, 8, 0)	External Port	

Figure 1-6: Port Interface settings for the dc offset correction HDL Coder project

7. Once the ports are set, right-click **HDL Code Generation** and select Run This Task. This will create a PCore for your design that can be used directly within Xilinx EDK. By default, the PCore is created in <Project Directory/MATLAB folder/codegen/ipcore>.

8. Once the PCore has been created, make a **new EDK project** using the same method used in the previous lab. Be sure that you **import** the correct system configuration file.

9. Once the project is created, **copy each of the PCore folders** from the MATLAB directory into the PCores folder of your **EDK Project**. Don't forget to also copy any previously created cores you may be reusing as well. Then simply select project -> **rescan user repositories** to show your newly added user PCores within your EDK project.

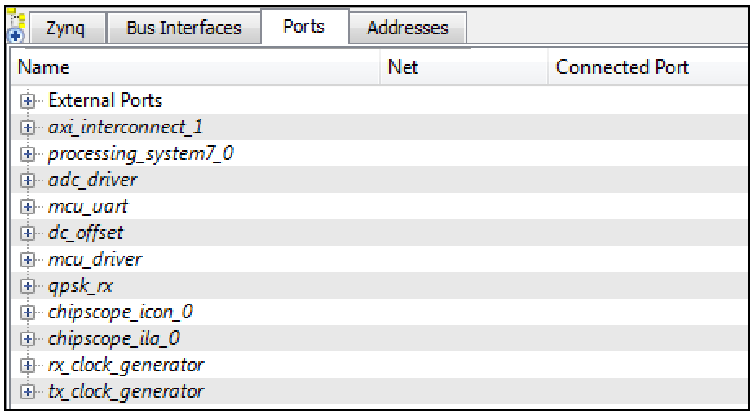
Configure Cores and Export Design Step 2

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.

2.1 Needed IP Cores

- ADC Driver
- MCU Driver
- MCU UART
- DC Offset
- QPSK RX
- Clock Generator (one for RX and one for TX)
- Processing System
- AXI Interconnect

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 2-1 Below.



Name	Net	Connected Port
External Ports		
axi_interconnect_1		
processing_system7_0		
adc_driver		
mcu_uart		
dc_offset		
mcu_driver		
qpsk_rx		
chipscope_icon_0		
chipscope_ila_0		
rx_clock_generator		
tx_clock_generator		

Figure 2-1: EDK project ports list

2.2 Configuring the ADC Driver Port

Expand the **ADC Driver** port. There are 6 individual I/O pins which need to be routed on this port.

1. First we will configure the `rx_iq_sel`, the `rxd` and the `bliky` pins. Each of these pins can be assigned as **External ports**.
2. Next are the `rx_i` and the `rq_q` output pins. Connect these pins to the `i_in` and `q_in` pins of the dc_offset PCore.
3. Connect the IPCORE_RESETN port to the processing_system7 FCLK_RESET0_N port.
4. The `IPCORE_CLK` pin can be skipped for now and will be connected later in **section 2.5**

2.3 Configuring the MCU Driver Port

Expand the **MCU Driver** core. There are 9 individual I/O pins which need to be routed on this core.

1. Configuring this core is very simple as **all of the pins** with the exception of the `IPCORE_CLK` and the `IPCORE_RESETN` are simply **assigned** as **external ports**.
2. Connect the IPCORE_RESETN port to the processing_system7 FCLK_RESET0_N Port and skip the `IPCORE_CLK` for now.

2.4 Configuring the MCU UART

1. Under the Communications Low-Speed section, add the AXI UART (Lite) to your design
2. Name the core mcu_uart as shown in Figure 2-1. Keep all configuration settings as default.
3. This core requires no other customization; just verify the RX and TX pins are set as External ports.

2.5 Configuring the DC Offset

Expand the **DC Offset** core. There are 7 individual I/O pins which need to be routed on this core.

1. If the ADC driver was previously configured correctly, the i_in and q_in pins of the dc_offset core should already be set.
2. The i_out and q_out pins should be connected to the qpsk_rx i_in and q_in pins respectively.
3. Set the blinky pin as an External port.
4. Connect the IPCORE_RESETN port to the processing_system7 FCLK_RESET0_N Port and skip the `IPCORE_CLK` for now.

2.6 Configuring the QPSK RX

Expand the **QPSK RX** core. There are 8 individual I/O pins which need to be routed on this core.

1. If the DC Offset core was previously configured correctly, the i_in and q_in pins of the qpsk_rx core should already be set.
2. Set the blinky pin as an External port.
3. The s_out, o_out and byte pins should be left unconnected for now and will eventually be connected to ChipScope for further analysis.
4. Connect the IPCORE_RESETN port to the processing_system7 FCLK_RESET0_N Port and skip the `IPCORE_CLK` for now.

2.7 Configuring the TX Clock Generator IP Core

The TX Clock Generator is used in this project to distribute the appropriate clock signals to each of the PCores required for Chilipepper initialization, as well as any external hardware which may require a clock signal. For this project, the TX Clock Generator is sourced from the 40 MHz `pll_clk_out` on the Chilipepper radio board (as described in the **Chilipepper user's guide**). This signal is then distributed to 3 other devices; 1 PCore (MCU Driver) and the `TX_CLK` and `RX_CLK` signals. The TX and RX clock signals are used to latch data from the TXD and RXD lines to the DAC and ADC respectively on the radio board. Although no DAC is used within the design, the clock is required for proper initialization of the Chilipepper FMC. For this lab, the Clock Generator has been named tx_clock_generator.

1. **Double click** the Clock Generator PCore and **configure** the settings as follows
   - Input Clock Frequency of **40Mhz**
   - CLKOUT0 Required Frequency of **20MHz**, 0 Phase, **PLLE0** group and **Buffered True**
   - CLKOUT1 Required Frequency of **40MHz**, 180 Phase, **PLLE0** group and **Buffered True**
   - CLKOUT2 Required Frequency of **40Mhz**, 0 Phase, **PLLE0** group and **Buffered True**

Now that the settings are configured you should have several clocks in your clock generator list.

2. **Connect** the pins according to the following.
   - CLKIN $\longrightarrow$ External Ports

- CLKOUT0 $\longrightarrow$ mcu:: IPCORE_CLK
- CLKOUT1 $\longrightarrow$ External Ports
- CLKOUT2 $\longrightarrow$ External Ports
- RST $\longrightarrow$ net_gnd
- LOCKED $\longrightarrow$ External Port

2.8 Configuring the RX Clock Generator IP Core

In addition to the TX Clock Generator, another clock generator is required for this design. As mentioned in Lab 2 and the Chilipepper User's Guide, the receiver chain is to be clocked using the RX return clock on the Chilipepper board to ensure data is latched properly from the ADC. In this design, there are three cores which must be clocked using the RX return clock; therefore a new clock generator called rx_clock_generator is used to distribute the clock signal.

1. **Double click** the Clock Generator PCore and **configure** the settings as follows

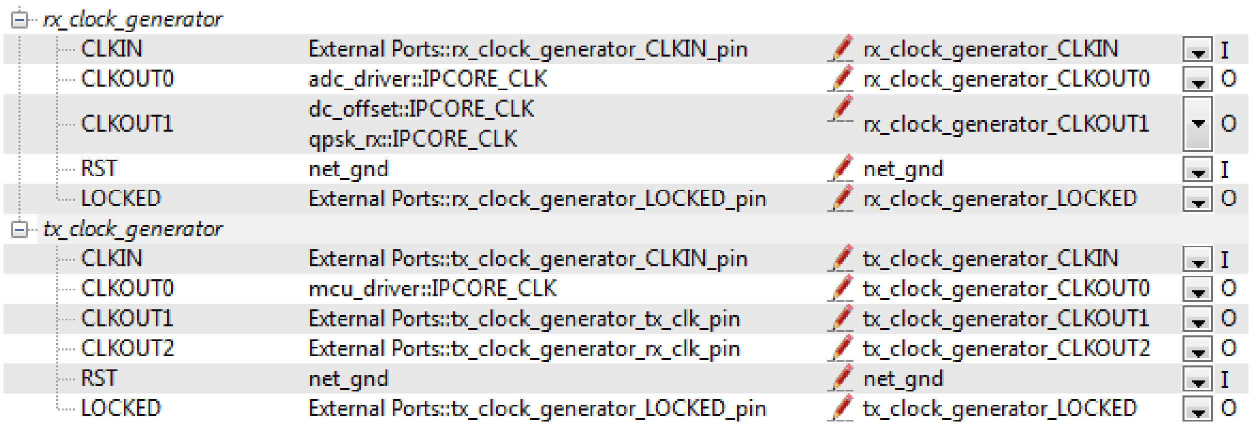
- Input Clock Frequency of **40Mhz**
- CLKOUT0 Required Frequency of **40MHz**, 180 Phase, **PLLE0** group and **Buffered True**
- CLKOUT1 Required Frequency of **20MHz**, 180 Phase, **PLLE0** group and **Buffered True**

Now that the settings are configured you should have several clocks in your clock generator list.

2. **Connect** the pins according to the following.

- CLKIN $\longrightarrow$ External Ports
- CLKOUT0 $\longrightarrow$ adc_driver::IPCORE_CLK
- CLKOUT1 $\longrightarrow$ dc_offset:: IPCORE_CLK and qpsk_rx::IPCORE_CLK
- RST $\longrightarrow$ net_gnd
- LOCKED $\longrightarrow$ External Port

Your Clock Generator ports should look similar to Figure 2-2 below.



rx_clock_generator			
CLKIN	External Ports::rx_clock_generator_CLKIN_pin	rx_clock_generator_CLKIN	I
CLKOUT0	adc_driver::IPCORE_CLK	rx_clock_generator_CLKOUT0	O
CLKOUT1	dc_offset::IPCORE_CLK qpsk_rx::IPCORE_CLK	rx_clock_generator_CLKOUT1	O
RST	net_gnd	net_gnd	I
LOCKED	External Ports::rx_clock_generator_LOCKED_pin	rx_clock_generator_LOCKED	O
tx_clock_generator			
CLKIN	External Ports::tx_clock_generator_CLKIN_pin	tx_clock_generator_CLKIN	I
CLKOUT0	mcu_driver::IPCORE_CLK	tx_clock_generator_CLKOUT0	O
CLKOUT1	External Ports::tx_clock_generator_tx_clk_pin	tx_clock_generator_CLKOUT1	O
CLKOUT2	External Ports::tx_clock_generator_rx_clk_pin	tx_clock_generator_CLKOUT2	O
RST	net_gnd	net_gnd	I
LOCKED	External Ports::tx_clock_generator_LOCKED_pin	tx_clock_generator_LOCKED	O

Figure 2-2: Clock Generator port configurations

Be sure your External Port pins, as well as your PCores match the names shown in the figures above.

2.9 Pin Assignments

Once the clock generator is configured correctly, the `IPCORE_CLK` for the other cores should be set as well. The next step is to setup the **pin assignments** for the external ports.

1. Open the **Project** tab.
2. Double-click on the **UCF File: data\system.ucf** from this panel, to open the constraints file.
3. Fill in the pin out information for your design using Figure 2-3 below as a reference.

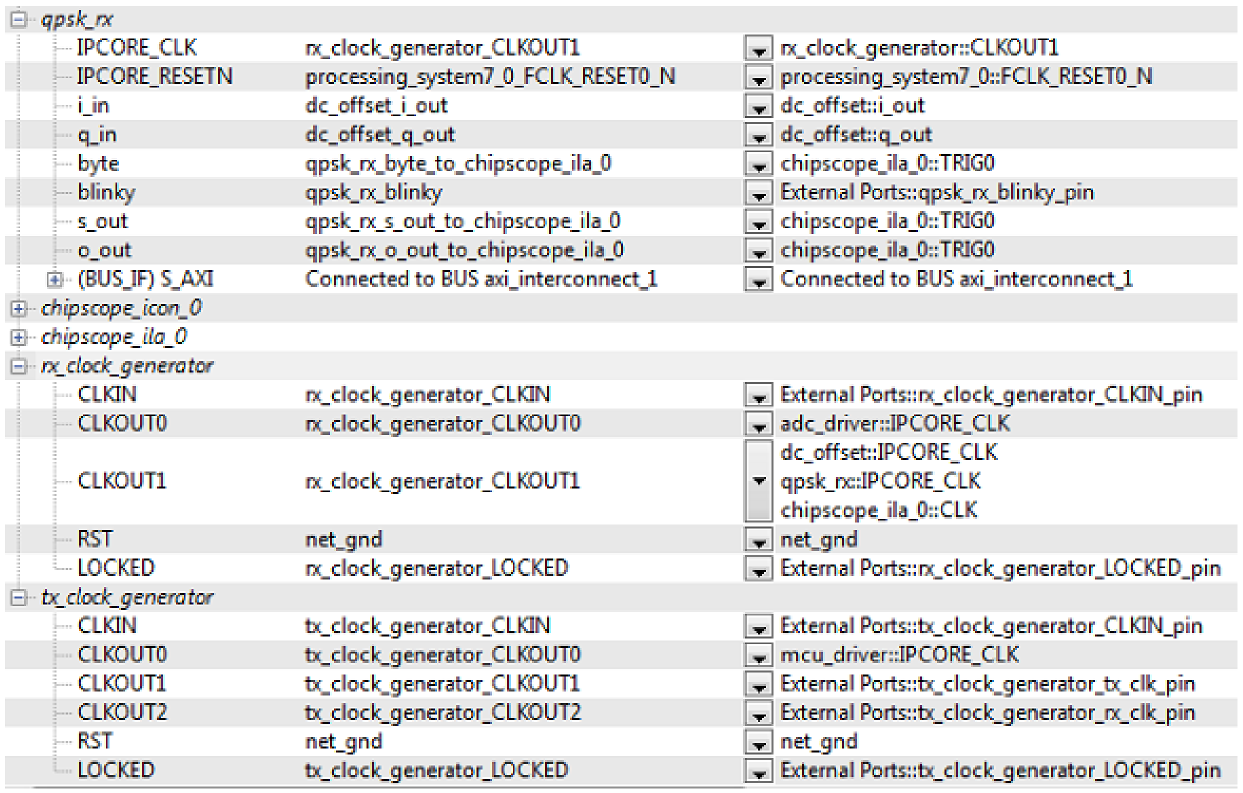
```
################################################## PL clocks and reset ##########################################
NET tx_clock_generator_CLKIN_pin                LOC = D18   | IOSTANDARD = LVCMOS25;
NET tx_clock_generator_CLKIN_pin                TNM_NET = tx_clock_generator_CLKIN;
TIMESPEC TS_tx_clock_generator_CLKIN = PERIOD tx_clock_generator_CLKIN 40.000 MHz;
################################################################################################################
NET rx_clock_generator_CLKIN_pin                LOC = L18   | IOSTANDARD = LVCMOS25;
NET rx_clock_generator_CLKIN_pin                TNM_NET = rx_clock_generator_CLKIN;
TIMESPEC TS_rx_clock_generator_CLKIN = PERIOD rx_clock_generator_CLKIN 40.000 MHz;
###########################################Chilipepper Rx and Tx clock lines######################################
NET tx_clock_generator_tx_clk_pin               LOC = C17          | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET tx_clock_generator_rx_clk_pin               LOC = J18          | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
##############################################Rx – FMC interface at 2.5V ########################################
NET adc_driver_rx_iq_sel_pin                    LOC = N19          | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[0]                       LOC =M21           | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[1]                       LOC = J21          | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[2]                       LOC = M22          | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[3]                       LOC = J22          | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[4]                       LOC = T16          | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[5]                       LOC = P20          | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[6]                       LOC = T17          | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[7]                       LOC = N17          | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[8]                       LOC = J20          | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[9]                       LOC = P21          | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[10]                      LOC = N18          | IOSTANDARD = LVCMOS25;
NET adc_driver_rxd_pin[11]                      LOC = J16          | IOSTANDARD = LVCMOS25;
############################################### MCU Interface ###################################################
NET mcu_uart_RX_pin                             LOC = R19          | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_uart_TX_pin                             LOC = L21          | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_driver_mcu_reset_out_pin                LOC = K20          | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_driver_tx_en_pin                        LOC = D22          | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_driver_tr_sw_pin                        LOC = D20          | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_driver_rx_en_pin                        LOC = C22          | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_driver_pa_en_pin                        LOC = E21          | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET mcu_driver_init_done_pin                    LOC = K19          | IOSTANDARD = LVCMOS25;
################################################## LEDs ########################################################
NET tx_clock_generator_LOCKED_pin               LOC = T22          | IOSTANDARD =LVCMOS33;  # "LD0"
NET rx_clock_generator_LOCKED_pin               LOC = T21          | IOSTANDARD =LVCMOS33;  # "LD1"
NET adc_driver_blinky_pin                       LOC = U22          | IOSTANDARD =LVCMOS33;  # "LD2"
NET mcu_driver_blinky_pin                       LOC = U21          | IOSTANDARD =LVCMOS33;  # "LD3"
NET dc_offset_blinky_pin                        LOC = V22          | IOSTANDARD =LVCMOS33;  # "LD4"
NET qpsk_rx_blinky_pin                          LOC = W22          | IOSTANDARD =LVCMOS33;  # "LD5"
```

Figure 2-3: EDK project pin assignments

2.10 Adding ChipScope Peripheral

The last step is to setup the ChipScope peripheral which will be used to capture the output of the qpsk_rx core for further analysis in MATLAB.

1. Select Debug -> **Debug Configuration** from the top menu
2. Click the **Add ChipScope Peripheral** button on the bottom left hand side of the screen
3. Select To **monitor arbitrary system level signals** (middle option) from the list.
4. Add the s_out, o_out and byte pins from the qpsk_rx Port. Additionally, you should set the clock to the same clock used for the core, which for this design is rx_clock_generator_clockout_1.
5. Click ok to finish configuration of your ChipScope peripheral. Your new port list should look similar to Figure 2-4 below. Be sure your Clock and qpsk_rx ports have the ChipScope peripherals in the correct locations.



qpsk_rx		
IPCORE_CLK	rx_clock_generator_CLKOUT1	rx_clock_generator::CLKOUT1
IPCORE_RESETN	processing_system7_0_FCLK_RESET0_N	processing_system7_0::FCLK_RESET0_N
i_in	dc_offset_i_out	dc_offset::i_out
q_in	dc_offset_q_out	dc_offset::q_out
byte	qpsk_rx_byte_to_chipscope_ila_0	chipscope_ila_0::TRIG0
blinky	qpsk_rx_blinky	External Ports::qpsk_rx_blinky_pin
s_out	qpsk_rx_s_out_to_chipscope_ila_0	chipscope_ila_0::TRIG0
o_out	qpsk_rx_o_out_to_chipscope_ila_0	chipscope_ila_0::TRIG0
(BUS_IF) S_AXI	Connected to BUS axi_interconnect_1	Connected to BUS axi_interconnect_1
chipscope_icon_0		
chipscope_ila_0		
rx_clock_generator		
CLKIN	rx_clock_generator_CLKIN	External Ports::rx_clock_generator_CLKIN_pin
CLKOUT0	rx_clock_generator_CLKOUT0	adc_driver::IPCORE_CLK
CLKOUT1	rx_clock_generator_CLKOUT1	dc_offset::IPCORE_CLK qpsk_rx::IPCORE_CLK chipscope_ila_0::CLK
RST	net_gnd	net_gnd
LOCKED	rx_clock_generator_LOCKED	External Ports::rx_clock_generator_LOCKED_pin
tx_clock_generator		
CLKIN	tx_clock_generator_CLKIN	External Ports::tx_clock_generator_CLKIN_pin
CLKOUT0	tx_clock_generator_CLKOUT0	mcu_driver::IPCORE_CLK
CLKOUT1	tx_clock_generator_CLKOUT1	External Ports::tx_clock_generator_tx_clk_pin
CLKOUT2	tx_clock_generator_CLKOUT2	External Ports::tx_clock_generator_rx_clk_pin
RST	net_gnd	net_gnd
LOCKED	tx_clock_generator_LOCKED	External Ports::tx_clock_generator_LOCKED_pin

Figure 2-4: Ports list after adding ChipScope peripheral to monitor qpsk_rx signals

Once completed, you're ready to generate your bitstream file! Select the Export Design button from the navigator window on the left. Click the Export and Launch SDK button. This process may take awhile.

Create software project Step 3

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 SDK folder in the same directory as your MATLAB and EDK folders. Doing this will keep all relevant files in the same location.

3.1 Creating a new C Project

This section will show you how to create a C program to test your QPSK RX project.

1. Select **File → New → Application Project**.
2. Name the project "qpsk_rx" or something similar and leave the other settings at their defaults. Click next.
3. On the next screen, be sure to select **Hello World** from the list of Available Templates.
4. Click **Finish**. You should now see your qpsk_rx project folder, as well as a **board support package** (bsp) folder.
5. If you navigate into the qpsk_rx project folder, and into the src folder, you should see a `helloworld.c` file. 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 Figure 3-1.
7. **Download** the **Chilipepper.c** and **Chilipepper.h** files from the GitHub repository[1] if you don't already have them. Copy them into the source directory with your main.c file.
8. Open the Chilipepper.c file and modify it for this lab. The only PCores that should be defined at the top of the file are MCU_DRIVER, DC_OFFSET, and MCU_UART.

Note	You may be required to add the Math Library to the project to define the pow function used in the Chilipepper.c Library file. If so, follow the optional step 9 listed below.

9. (Optional) Click on **Project →Properties.** Open the **C/C++ Build** arrow and click the settings option. Under **ARM gcc linker**, click the Libraries folder. Click the button, type the letter **m** into the prompt and select ok. **Apply** and hit ok.

[1] Can be found at https://github.com/Toyon/Chilipepper/tree/QPSK_pcore/ChilipepperSupport/Library%20Files

```
#include <stdio.h>
#include "platform.h"
#include "chilipepper.h"
#include "xuartps.h"

XUartPs uartPs;
XUartPs_Config *pUartPsConfig;

int main()
{

	init_platform();

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

	Chilipepper_SetPA(0);
	Chilipepper_SetTxRxSw(1);   // 0- transmit, 1-receive
	Chilipepper_SetDCOC(1);     // enable dc offset correction

	while (1)
	{
		Chilipepper_ControlAgc(); //update the Chilipepper AGC
	}
	cleanup_platform();
	return 0;
}
```

Figure 3-1: main.c file for DC Offset Correction SDK Project

3.2 Programming the Board

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.

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 Getting Started Guide*[2] 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.

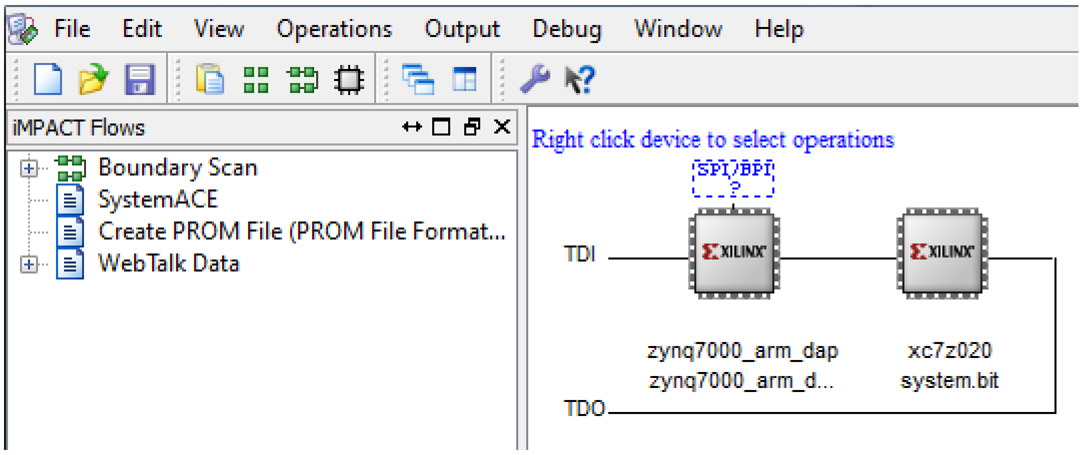
5. Select yes to allow iMPACT to automatically create a new project for you. If you receive any connection errors, verify your USB or JTAG programmer cables are connected properly.

[2] Can be found at https://github.com/Toyon/Chilipepper/tree/master/QPSK_Radio/DemoFilesAndDocumentation

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 3-2 below.



3-2: configuration for Zed Board System.bit file

9. Right click on the xc7z020 board icon (should be on the right), select program and hit ok.

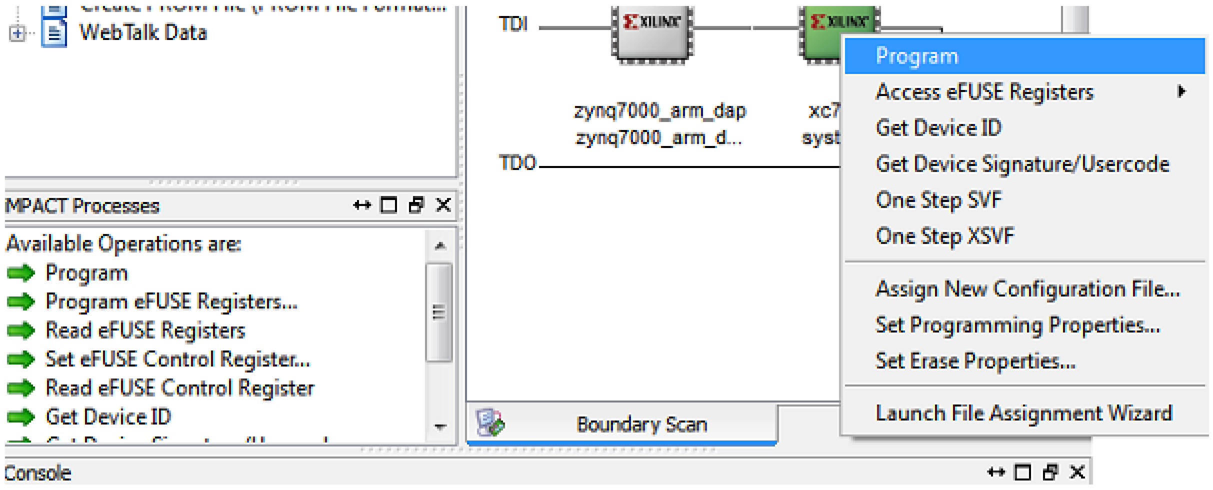

WebTalk Data
TDI
TDO
zynq7000_arm_dap
zynq7000_arm_d...
xc7
syst
MPACT Processes
Available Operations are:
Program
Program eFUSE Registers...
Read eFUSE Registers
Set eFUSE Control Register...
Read eFUSE Control Register
Get Device ID
Boundary Scan
Console
Program
Access eFUSE Registers
Get Device ID
Get Device Signature/Usercode
One Step SVF
One Step XSVF
Assign New Configuration File...
Set Programming Properties...
Set Erase Properties...
Launch File Assignment Wizard

Figure 3-3: iMPACT configuration screen

3.3 Debugging with SDK

If the hardware design is correct, you should see a blue light on the ZED Board indicating the program was successful. You can now return to the SDK project screen to test your software.

1. Test it by **right clicking** the `qpsk_rx` project folder and selecting **Debug As → Launch on Hardware (GDB)**.
2. You should now be taken to a screen which shows the `init_platform()` function 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 the Blinking LEDs on the FPGA from the PCore blinky pins.

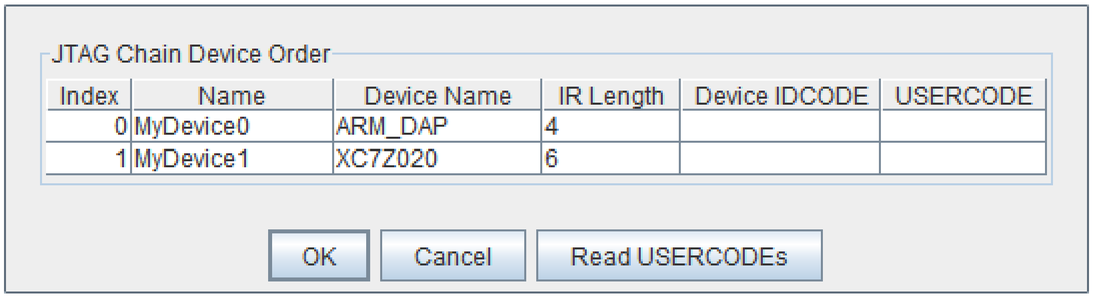
Testing and Design Verification Step 4

4.1 Verification with ChipScope Pro

There are several methods available for verifying the MATLAB functions. For verification of the qpsk_rx_correlator design, ChipScope is recommended as it provides the most useful view of the signal correlation magnitude and an output of the resulting bytes.

1. To verify the qpsk_rx signals, you will need to open **ChipScope Pro Analyzer**. Be sure that the JTAG cable is connected to the FPGA board properly.

2. Once the program opens, click the (open cable) button to open your JTAG connection to the board. If your jumpers are configured correctly, you should see the following devices on the cable.


JTAG Chain Device Order

Index	Name	Device Name	IR Length	Device IDCODE	USERCODE
0	MyDevice0	ARM_DAP	4		
1	MyDevice1	XC7Z020	6		

Note	If you receive an error from ChipScope stating that you either cannot detect or cannot open the cable, try using the optional Step 3 to configure your cable setup correctly.

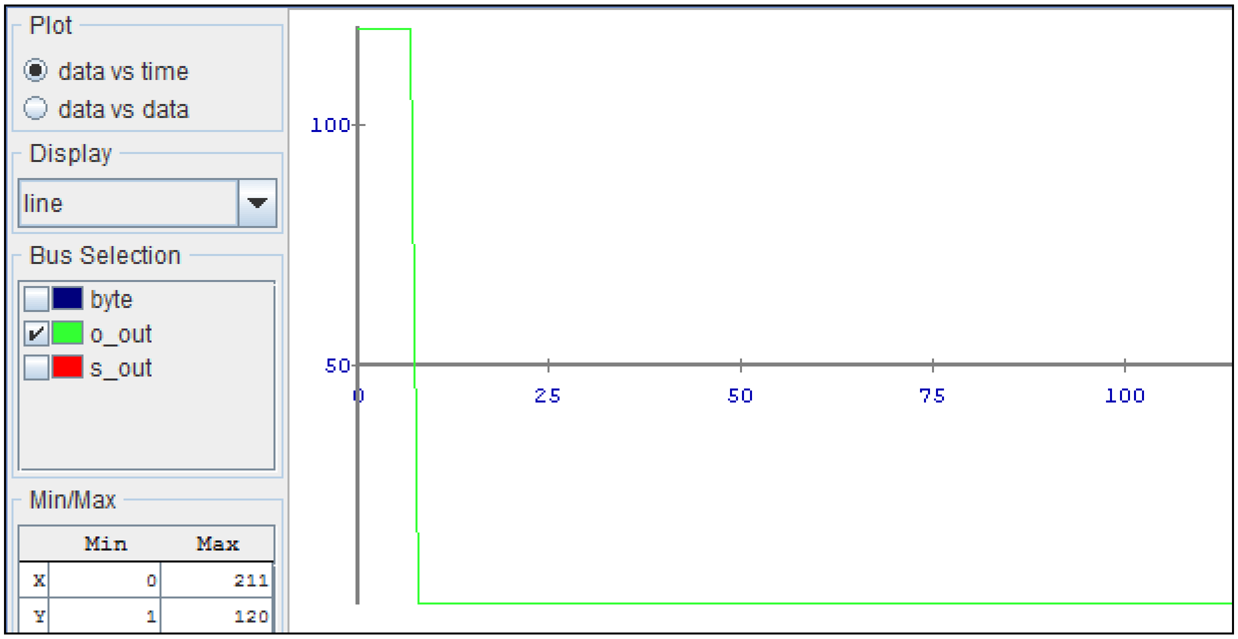
3. (**Optional**)Click JTAG Chain in the top menu selection. Select the option for **Open Plug-in**... You will be greeted with a Plug-in Parameters screen. Enter the following in the box, and hit ok. "**xilinx_tcf URL=tcp::3121**". Then click the open cable button and proceed as usual.

4. Select ok to get to the Analyzer main screen. Open the **file menu** and select **Import**.

5. Click **Select New File**, and browse to the location of your ChipScope **CDC file**, which is located in the <EDK/implementation/chipscope_ila_0_wrapper> folder of your project directory. This file was created for you when you generated your bit file in EDK, assuming you added the ChipScope peripheral appropriately. It tells the ChipScope program how to interpret the data it is receiving from the JTAG port.

6. On the Bus Plot screen, you can view correlation and NRZ signals that you connected to your ChipScope peripheral previously. Right click on a signal to change its features such as bus radix, name or color. For this Lab, both signals should be set to the signed decimal bus radix.

7. Click the **play button** in the top menu bar to display the signal. Additionally you can set up triggering options for periodic or continuous playback of the received signal. Your signals should look similar Figures 4-1 and 4-2.



Plot: [✓ data vs time] data vs data

Display: line

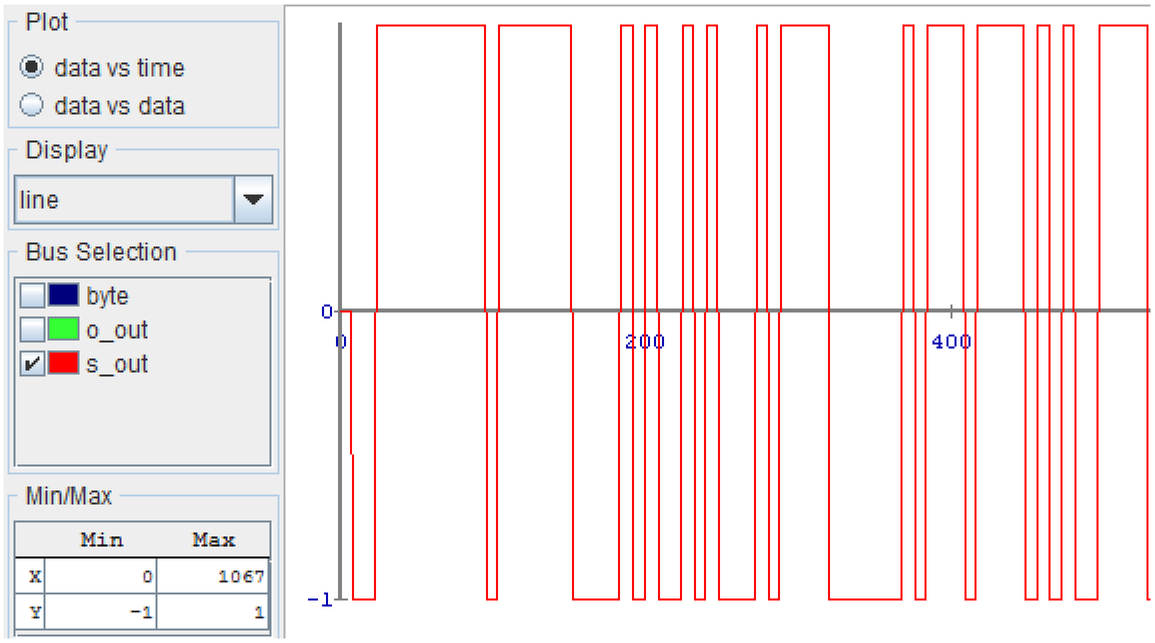
Bus Selection:

	🟦	byte
✓	🟩	o_out
	🟥	s_out

Min/Max:

	Min	Max
X	0	211
Y	1	120

4-1: Correlation magnitude when a new QPSK signal is received. Notice it spikes once the training sequence is detected.



Plot: [✓ data vs time] data vs data

Display: line

Bus Selection:

	🟦	byte
	🟩	o_out
✓	🟥	s_out

Min/Max:

	Min	Max
X	0	1067
Y	-1	1

4-2: NRX signal which represents each bit of the received message 'hello world!'

From Figure 4-1 above, you can also observe that the correlation magnitude spikes when the front loaded Kassami sequence is detected. At this point, the correlator begins working, and sending bytes out of the qpsk_rx core. However, these bytes are not yet valid as the received signal has not yet reached the payload of the transmitted signal. Therefore, this receiver core must now be expanded to let the processor know when valid data is sent out of the QPSK correlator. This will be the focus of lab 8.

Appendix A MATLAB Frequency Offset

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] = qpsk_rx_foc(y_i, y_q, finish_rx)

persistent phi

lSin = SIN;
lCos = COS;

if isempty(phi)
    phi = 0;
end

if finish_rx == 1
    phi = 0;
end

% create the VCO signal
%phi = mod(phi,1);
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

% update with hard coded mu 40/2^12
c = (0.009765625)*e;
phiNew = phi - c;
phi = phiNew;

z_i_out = z_i;
z_q_out = z_q;
```

Appendix B MATLAB Timing Offset

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] = qpsk_rx_toc(r_i, r_q, finish_rx)

persistent counter
persistent tau
persistent rBuf_i rBuf_q
persistent symLatch_i symLatch_q

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;
end

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);
    % basically if we shift out of the window just bail as we're screwed
    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 (with hard coded mu_in) 327/2^12
    tau = tau + (0.079833984375)*oe;

    symLatch_i = zo_i;
    symLatch_q = zo_q;
end

s_i = symLatch_i;
s_q = symLatch_q;

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

Appendix C MATLAB RX Correlator

MATLAB function `qpsk_rx_correlator.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 [byte_out, reset_out, s_out, o_out] = ...
    qpsk_rx_correlator(s_i_in, s_q_in)

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

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;
    persis_byte = 0;
end
```

```
reset_out = 0;
% found a packet, now we're ready to write the data
% out
if counter == 0 && detPacket == 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;
        persis_byte = bits*BIT_TO_BYTE;
        % first byte is number of bytes in payload
        if byteCount == 1
            numBytes = persis_byte;
        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;
            end
        end
    end
end
```

```
% let's see if we can find a packet. only do so if MCU is ok to rcv packet
if counter == 0 && detPacket == 0
    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;
    end
    if op > 100
        if sc_iWithq > 10 && sc_qWithi < 10
            q = 3; % 90 degrees

        else
            q = 1; % 270 degrees;
        end
        detPacket = 1;
    end
    oLatch = ip+op;
    symCount = 0;
    byteCount = 0;
    numBytes = 1000;
end
```

```
s_out = sLatch;
o_out = oLatch;

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

Appendix D MATLAB QPSK RX Test Bench Script

MATLAB script `qpsk_tb.m`

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Model/simulation parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
OS_RATE = 8;
SNR = 100;
fc = 10e3/20e6; % sample rate is 20 MHz, top is 10 kHz offset
sim = 1;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialize LUTs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
make_srrc_lut;
make_trig_lut;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Emulate microprocessor packet creation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% data payload creation
messageASCII = 'hello world!';
message = double(unicode2native(messageASCII));
% add on length of message to the front with four bytes
msgLength = length(message);
messageWithNumBytes =[ ...
    mod(msgLength,2^8) ...
    mod(floor(msgLength/2^8),2^8) ...
    mod(floor(msgLength/2^16),2^8) ...
    1 ... % message ID
    message];
% add two bytes at the end, which is a CRC
messageWithCRC = CreateAppend16BitCRC(messageWithNumBytes);
ml = length(messageWithCRC);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% FPGA radio transmit core
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
data_in = 0;
empty_in = 1;
tx_en_in = 0;
store_byte = 0;
numBytesFromFifo = 0;
num_samp = ml*8*2*2*3;
x = zeros(1,num_samp);
CORE_LATENCY = 4;
data_buf = zeros(1,CORE_LATENCY);
store_byte_buf = zeros(1,CORE_LATENCY);
```

```
clear_buf = zeros(1,CORE_LATENCY);
tx_en_buf = zeros(1,CORE_LATENCY);
re_byte_out(1) = 0;
reset_fifo = 0;
byte_request = 0;
for i1 = 1:num_samp
    % first thing the processor does is clear the internal tx fifo
    if i1 == 1
        clear_fifo_in = 1;
    else
        clear_fifo_in = 0;
    end

    data_buf = [data_buf(2:end) data_in];
    store_byte_buf = [store_byte_buf(2:end) store_byte];
    clear_buf = [clear_buf(2:end) clear_fifo_in];
    tx_en_buf = [tx_en_buf(2:end) tx_en_in];

    [new_data_in, empty_in, byte_recieved, full, percent_full] = ...
    tx_fifo(byte_request, store_byte_buf(1), data_buf(1), reset_fifo);

    [i_out, q_out, tx_done_out, request_byte, clear_fifo_in_done] = ...
        qpsk_tx(new_data_in,empty_in,clear_buf(1),tx_en_buf(1));
    x_out = complex(i_out,q_out)/2^11;
    x(i1) = x_out;
    byte_request = request_byte;

    %%% Emulate write to FIFO interface
    if mod(i1,8) == 1 && numBytesFromFifo < length(messageWithCRC)
        data_in = messageWithCRC(numBytesFromFifo+1);
        numBytesFromFifo = numBytesFromFifo + 1;
    end
    %%% Software lags a but on the handshaking signals %%%
    if (0 < mod(i1,8) && mod(i1,8) < 5) && tx_en_in == 0
        store_byte = 1;
    else
        store_byte = 0;
    end
    % processor loaded all bytes into FIFO so begin transmitting
    if (numBytesFromFifo == length(messageWithCRC) && mod(i1,8) > 5)
        empty_in = 1;
        tx_en_in = 1;
    end
end
if ~sim % load data that was transmitted and captured from chipscope
    if 1
        fid = fopen('tx.prn');
        M = textscan(fid,'%d %d %d %d %d %d %d %d %d %d','Headerlines',1);
        fclose(fid);
        iFile = double(M{3})'/2^11;
        qFile = double(M{4})'/2^11;
    else
        M = load('dac.prn');
        if M(1,end-1) == 0
            iFile = M(1:2:end,end)'/2^11;
            qFile = M(2:2:end,end)'/2^11;
        else
            qFile = M(1:2:end,end)'/2^11;
            iFile = M(2:2:end,end)'/2^11;
        end
    end
    x = complex(iFile,qFile);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Emulate channel
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% pad on either side with zeros
p = complex(zeros(1,100),zeros(1,100));
xp = [p x p]; % pad

% Apply frequency offset and receive/over-the-air AWGN
y = xp.*exp(1i*2*pi*fc*(0:length(xp)-1));
rC = y/max(abs(y))*.1*2^11; % this controls receive gain
%r = awgn(rC,SNR,0,1);
r = rC;
if ~sim
    fid = fopen('rx.prn');
    M = textscan(fid,'%d %d %d %d','Headerlines',1);
    fclose(fid);
    is = double(M{3});
    qs = double(M{4});
    r = complex(is,qs);
    figure(1)
    subplot(2,1,1);
    plot(is);
    subplot(2,1,2);
    plot(qs)
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Main receiver core
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
r_out = zeros(1,length(r));
s_out = zeros(1,length(r));
o_out = zeros(1,length(r));
for i1 = 1:length(r)+200
    if i1 > length(r)
        r_in = 0;
    else
        r_in = r(i1);
    end
    i_in = round(real(r_in));
    q_in = round(imag(r_in));
    r_out(i1) = real(complex(i_in,q_in));
    [dc_i_out, dc_q_out, rssi_out, rssi_en_out, dir_out, dir_en_out] = ...
        dc_offset_correction(i_in, q_in, mod(i1,2), 500, 1500, +(i1>3000));

    [byte(i1), blinky, s_out(i1), o_out(i1)] =...
        qpsk_rx(dc_i_out, dc_q_out);

end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

figure(2)
subplot(2,2,1)
scatter(real(r),imag(r))
title('OTA Receive Signal');
subplot(2,2,2)
plot(real(r_out));
title('OTA Receive Signal (real part)');
subplot(2,2,3)
plot(s_out);
title('Message bits');
subplot(2,2,4)
axis([0 length(s_out) -1.25 1.25]);
plot(o_out);
title('Correlation magnitude');
```