Toyon Research Corporation

Lab 4: DC Offset Correction

Chilipepper Tutorial Projects

Version 0.4

4/22/2013

Table of Contents

Lab 4: DC Offset Correction

Introduction

This lab will show you how to recover your QPSK Waveform generated in Lab 3 using a second Xilinx Zed Board FPGA and Toyon Chilipepper FMC. 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. In future labs, we will increase this verification gradually using 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 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:

- Generate HDL code from a MATLAB algorithm
- 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:

- Implement DC Offset correction for a received QPSK Waveform
- Create a Simulink model to implement a basic signal receiver
- Receive a QPSK Waveform using the Chilipepper FMC
- Create a basic C Application Project to test your design
- Verify your results in ChipScope

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 Function

Your MATLAB function 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. The code required for this lab will be used in future labs as the receiver core is expanded.

The MATLAB Function used is called `dc_offset_correction.m`. Its primary functionality is to correct any DC offset that occurs when receiving the signal from the ADC, however it also performs some related RSSI estimation and gain correction routines required for proper dc offset correction. The MATLAB code for this function is shown in Appendix A.

1. Create a directory for the project under C:\QPSK_Projects\Project_4.
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 `dc_offset_correction.m` inside the MATLAB project directory.

1.2 MATLAB Test Bench

Now that you have created the code needed to correct the DC offset, we also need to create a test bench script to test the algorithm. This is done by observing the output graph of the result, which in this case is the signal I and q channels with their corrected means, as well as a plot of the RSSI output. To accomplish this, it is necessary to have a “test” signal from the ADC to use for the analysis; Therefore in addition to this script, you will need a signal exported from ChipScope with the ADC output for both the i and q channels. You can either use the ChipScope data obtained from Lab 2, or download the DC.prn file provided on the GitHub Repo.

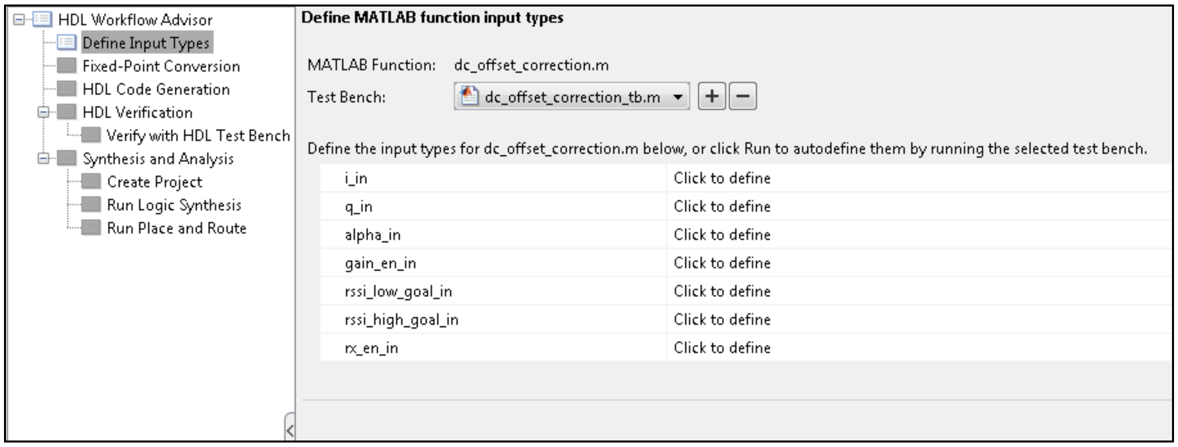
Note	If you use the ChipScope data exported from Lab 2 as your ADC signal input you may need to modify the Test Bench script file to correctly load the variables you exported. Run “help textscan” to get more information on how to change this line to load data from your ChipScope Export.

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

1.3 DC Offset Correction HDL Coder Project

Using the same steps outlined in the previous labs, create a new HDL coder project called `dc_offset`. Add your MATLAB function and test bench script 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 1-1 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 in this lab, the auto-defined types are fine.



Define MATLAB function input types

MATLAB Function: dc_offset_correction.m

Test Bench: dc_offset_correction_tb.m

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

i_in	Click to define
q_in	Click to define
alpha_in	Click to define
gain_en_in	Click to define
rssi_low_goal_in	Click to define
rssi_high_goal_in	Click to define
rx_en_in	Click to define

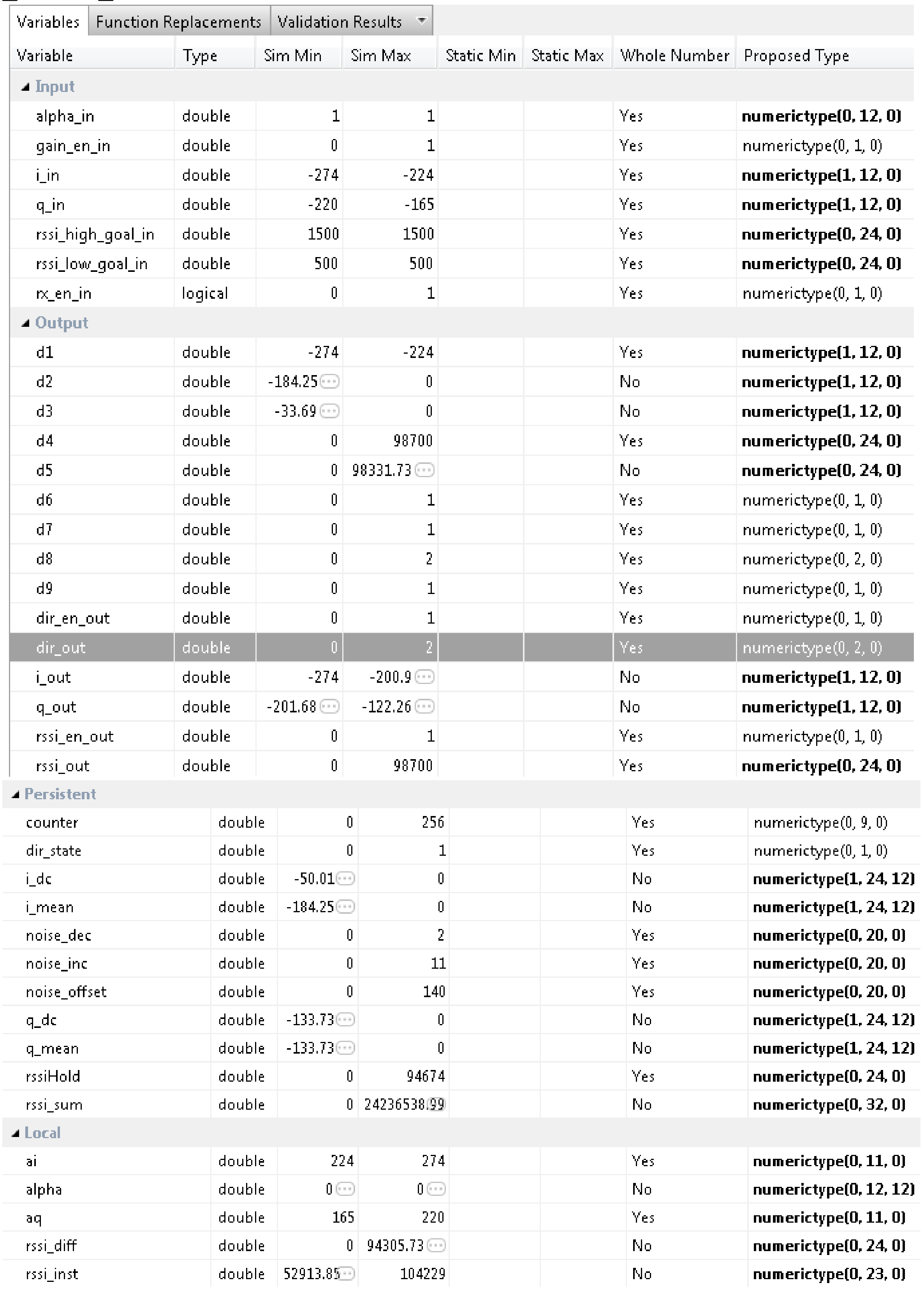
Figure 1-1: HDL Code Generation Workflow Advisor

1. Open Workflow Advisor and select “Run” to auto-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. Modify your HDL Coder design to match the following Fixed-Point conversions

dc_offset_correction



Variables | Function Replacements | Validation Results

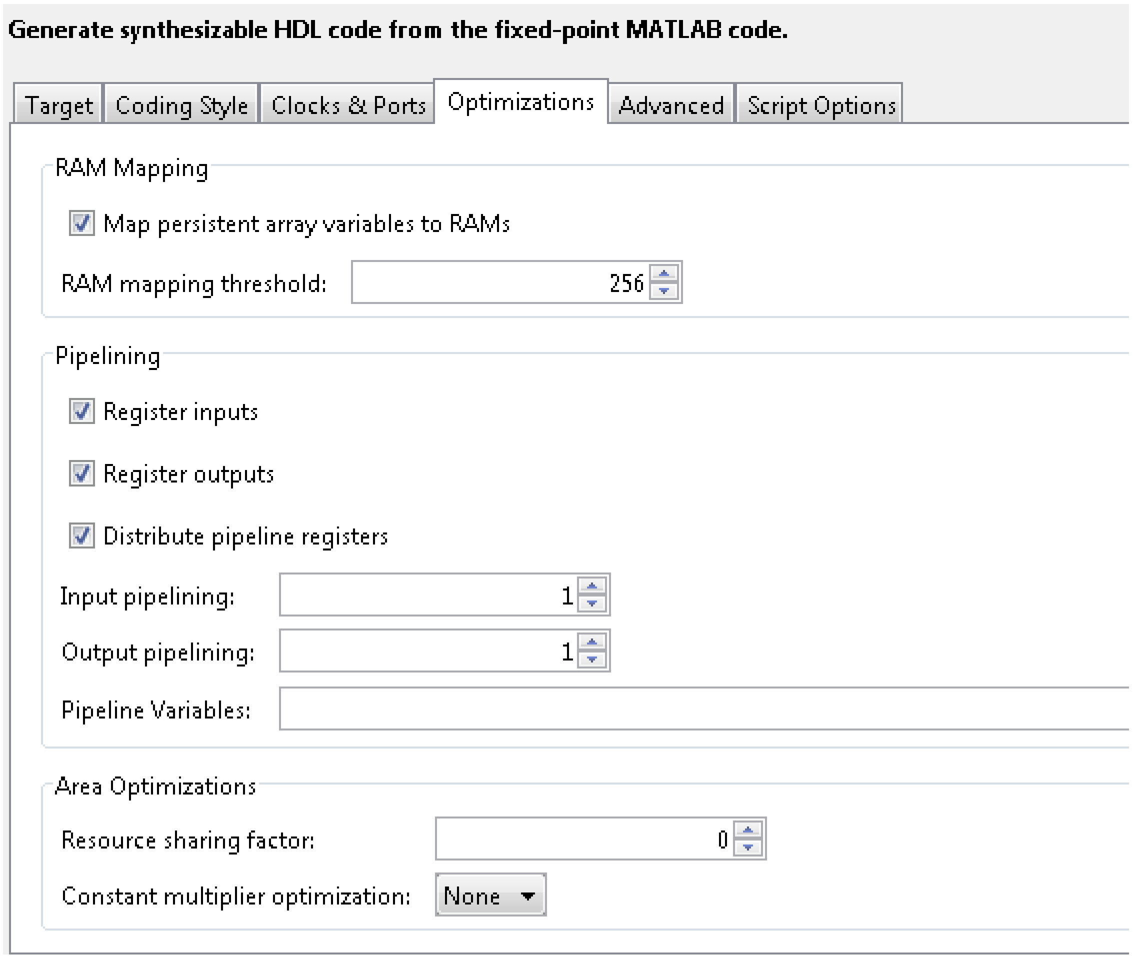
Variable	Type	Sim Min	Sim Max	Static Min	Static Max	Whole Number	Proposed Type
◢ Input							
alpha_in	double	1	1			Yes	**numerictype(0, 12, 0)**
gain_en_in	double	0	1			Yes	numerictype(0, 1, 0)
i_in	double	-274	-224			Yes	**numerictype(1, 12, 0)**
q_in	double	-220	-165			Yes	**numerictype(1, 12, 0)**
rssi_high_goal_in	double	1500	1500			Yes	**numerictype(0, 24, 0)**
rssi_low_goal_in	double	500	500			Yes	**numerictype(0, 24, 0)**
rx_en_in	logical	0	1			Yes	numerictype(0, 1, 0)
◢ Output							
d1	double	-274	-224			Yes	**numerictype(1, 12, 0)**
d2	double	-184.25…	0			No	**numerictype(1, 12, 0)**
d3	double	-33.69…	0			No	**numerictype(1, 12, 0)**
d4	double	0	98700			Yes	**numerictype(0, 24, 0)**
d5	double	0	98331.73…			No	**numerictype(0, 24, 0)**
d6	double	0	1			Yes	numerictype(0, 1, 0)
d7	double	0	1			Yes	numerictype(0, 1, 0)
d8	double	0	2			Yes	numerictype(0, 2, 0)
d9	double	0	1			Yes	numerictype(0, 1, 0)
dir_en_out	double	0	1			Yes	numerictype(0, 1, 0)
dir_out	double	0	2			Yes	numerictype(0, 2, 0)
i_out	double	-274	-200.9…			No	**numerictype(1, 12, 0)**
q_out	double	-201.68…	-122.26…			No	**numerictype(1, 12, 0)**
rssi_en_out	double	0	1			Yes	numerictype(0, 1, 0)
rssi_out	double	0	98700			Yes	**numerictype(0, 24, 0)**
◢ Persistent							
counter	double	0	256			Yes	numerictype(0, 9, 0)
dir_state	double	0	1			Yes	numerictype(0, 1, 0)
i_dc	double	-50.01…	0			No	**numerictype(1, 24, 12)**
i_mean	double	-184.25…	0			No	**numerictype(1, 24, 12)**
noise_dec	double	0	2			Yes	**numerictype(0, 20, 0)**
noise_inc	double	0	11			Yes	**numerictype(0, 20, 0)**
noise_offset	double	0	140			Yes	**numerictype(0, 20, 0)**
q_dc	double	-133.73…	0			No	**numerictype(1, 24, 12)**
q_mean	double	-133.73…	0			No	**numerictype(1, 24, 12)**
rssiHold	double	0	94674			Yes	**numerictype(0, 24, 0)**
rssi_sum	double	0	24236538.99			No	**numerictype(0, 32, 0)**
◢ Local							
ai	double	224	274			Yes	**numerictype(0, 11, 0)**
alpha	double	0…	0…			No	**numerictype(0, 12, 12)**
aq	double	165	220			Yes	**numerictype(0, 11, 0)**
rssi_diff	double	0	94305.73…			No	**numerictype(0, 24, 0)**
rssi_inst	double	52913.85…	104229			No	**numerictype(0, 23, 0)**

Once all modifications have been made, you will need to let MATLAB validate the new conversion values for your new Fixed-Point conversions. Again, once the process is complete, you should get a message saying Validation Succeeded.

4. Select Validate Types in the top right corner of the screen to verify the new design.

5. Click on HDL Code Generation and modify the settings according to the previous labs.

Due to the size and delay required for the internal registers of this MATLAB function, it is required to add some additional pipelining to these registers to meet timing constraints. This can be done by checking the pipelining options under Optimizations in the HDL Code Generation settings. Refer to Figure 1-6 below.


Generate synthesizable HDL code from the fixed-point MATLAB code.

RAM Mapping

[✓] Map persistent array variables to RAMs

RAM mapping threshold: 256

Pipelining

[✓] Register inputs

[✓] Register outputs

[✓] Distribute pipeline registers

Input pipelining: 1

Output pipelining: 1

Pipeline Variables:

Area Optimizations

Resource sharing factor: 0

Constant multiplier optimization: None

1-2: Modified optimization settings to allow for register pipelining

6. Right Click and select run this task to generate your Xilinx Block Box Design. Your default design should look similar to Figure 1-6 below.

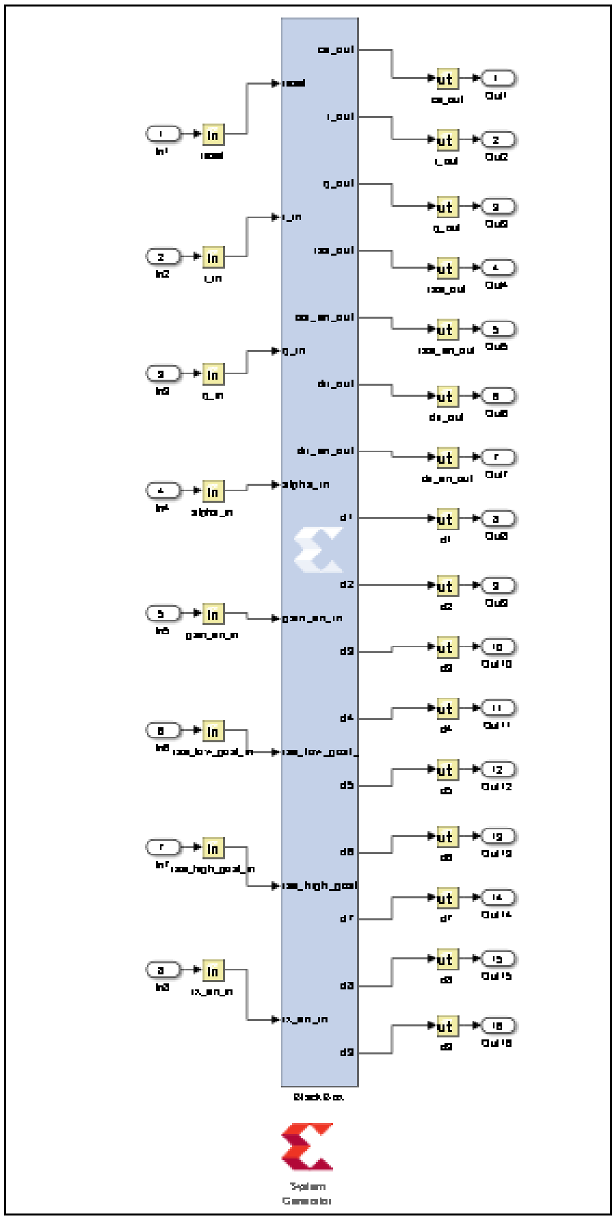


Figure 1-7: Xilinx Black Box generated by HDL Coder

7. Copy the black box and System generator Blocks to a new Model just as in the previous labs.
8. Save the new model as dc_offset.slx into the sysgen directory “Lab_4\sysgen”.
9. Copy the `dc_offset_correction_FixPt_xsgbbxcfg.m` file into your Sysgen folder just as in the previous labs.

10. Create the hdl folder inside the Sysgen folder and copy your vhd file into this directory. Make sure you modify the previously copied m file to point to the new location of the vhd files. Refer to previous labs for the process required for these modifications.

Create and export Simulink models Step 2

This section will show you how to create and customize your Simulink models to properly receive a signal and control the Chilipepper MCU and ADC. For additional information on this process, see lab 2.

2.1 Create MCU Simulink Design

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

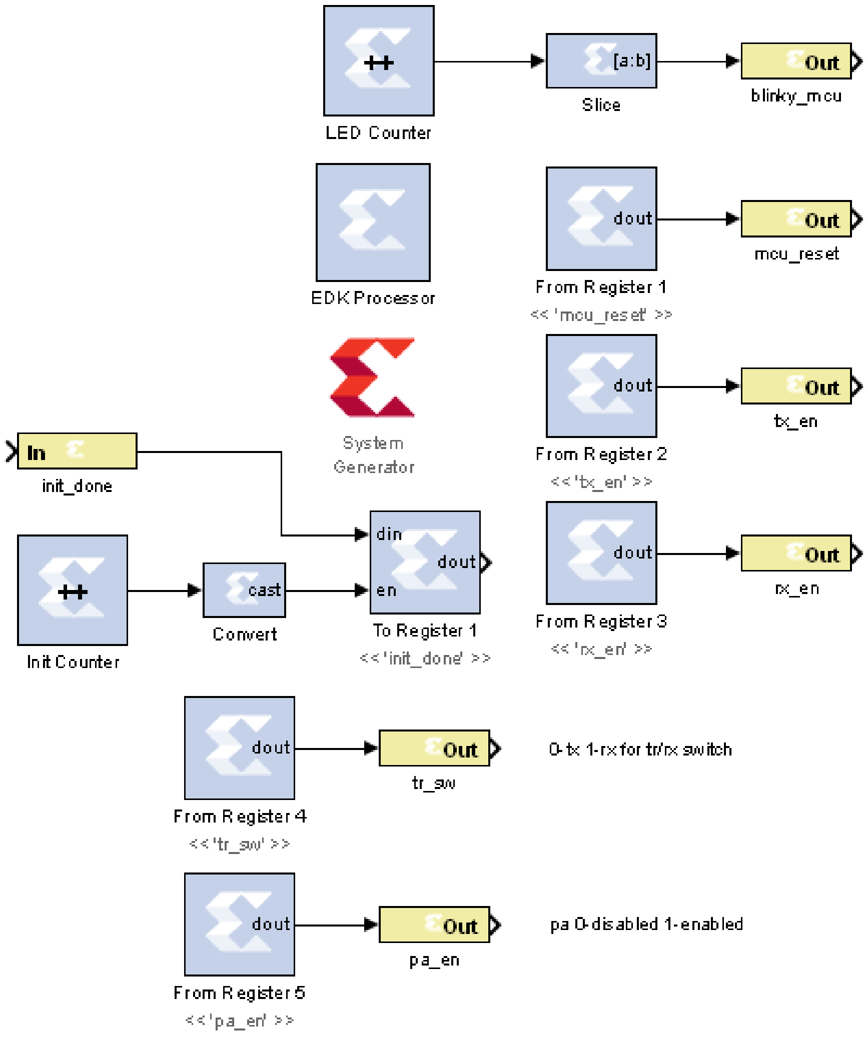


Figure 2-1: Simulink model for MCU control

[1] This model can be downloaded from https://github.com/Toyon/Chilipepper/tree/master/Labs/Lab_4/sysgen

1. To create a new Simulink model, open MATLAB and click on the **Simulink Library** button in the Home menu.

2. Select **File → New → Model**

3. **Configure** this model and the system generator the same as in Lab 1, and **save** the design into the Sysgen folder. **Be sure to change the cfg file as well to find the files in your new directory structure.** Name the file **mcu.slx** or something similar.

2.2 Create ADC driver Simulink Design

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

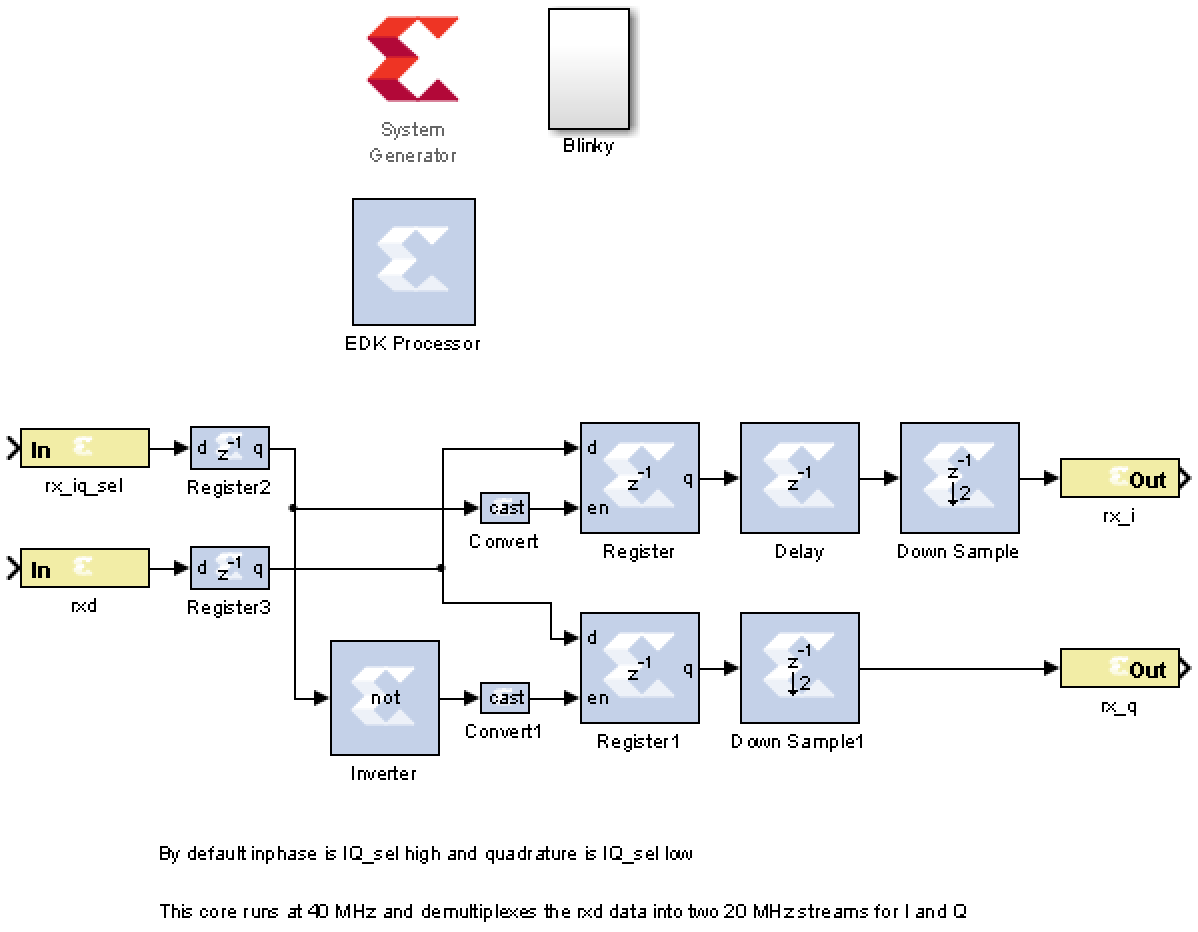


Figure 2-2: Simulink model for ADC control

[2] This model can be downloaded from https://github.com/Toyon/Chilipepper/tree/master/Labs/Lab_4/sysgen

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 2-3. Configure the Blinky subsystem identically to the other LED out systems in the previous labs.

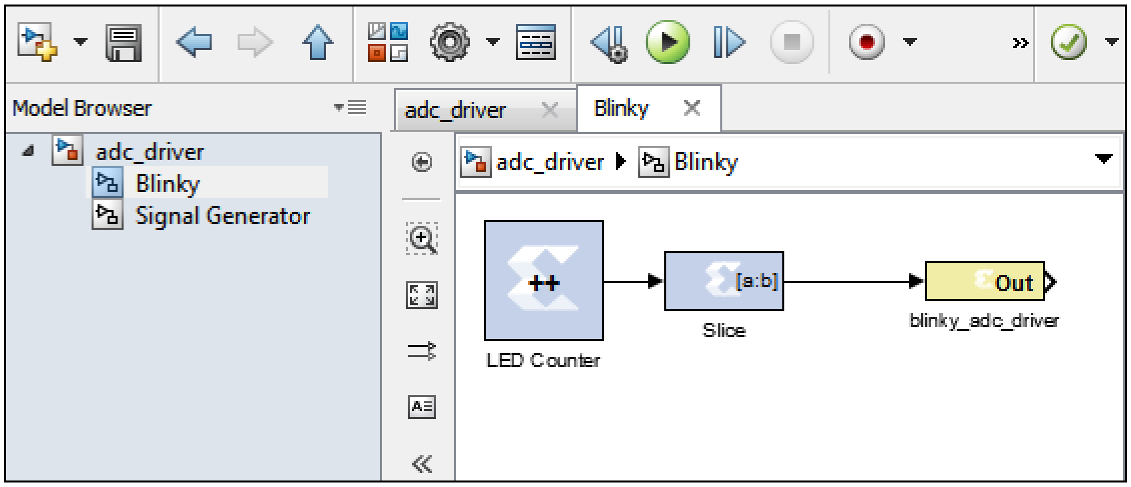


Figure 2-3: Blinky Subsystem

2.3 Create Receiver Simulink Design

Now we will modify the previously created dc_offset.slx model created earlier. Your design should resemble the **Simulink model**[3] shown in Figure 2-4. This model will be used for receiving the ADC output and sending the resultant data to ChipScope for verification. You should notice that several of the outputs of the design (d1-d9) are not connected. These outputs are debug outputs, and only need to be used for finding problems within your design. They have not been included in this lab, however if you have problems with your design, you can simply add these output to your ChipScope input block for help in finding problems with your MATLAB design.

1. **Modify** the Simulink model dc_offset.slx created earlier to look similar to Figure 2.4.

[3] This model can be downloaded from https://github.com/Toyon/Chilipepper/tree/master/Labs/Lab_4/sysgen

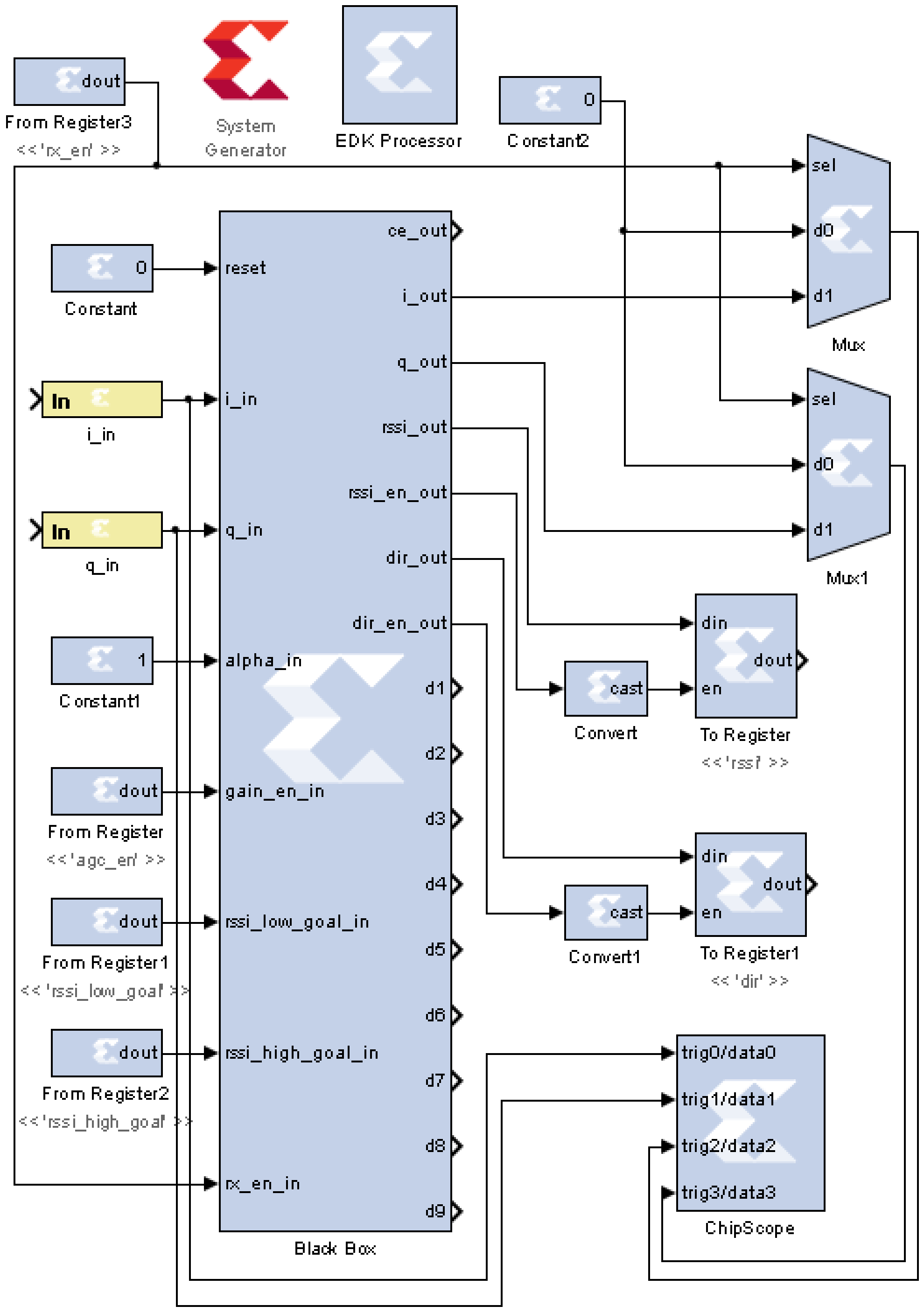


Figure 2-4: Simulink model for receiving ADC output and applying DC Offset Correction

2. Both `i_in` and `q_in` should be signed 12 bit (0 decimal bits) inputs.

3. The constant for `alpha_in` should be a 12 bit 0 decimal fixed-point unsigned number, and its constant value should be 0.

4. The output of both the `agc_en` and the `rx_en` From Registers should be a 1 bit 0 decimal fixed point value.

5. The output of both the `rssi_low_goal` and the `rssi_high_goal` From Registers should be a 24 bit 0 decimal Fixed Point unsigned number.

6. The Convert blocks are simply Boolean output types.

7. The rssi To Register is a Fixed-Point unsigned number with 24 bits and 0 decimal bits.

8. The dir To Register is a Fixed-Point unsigned number with 2 bits and 0 decimal bits

9. Create a new EDK Project, and export each of the cores to the project one by one

Refer to Lab 0 Step 3 for more information on how to **Create a New Blank EDK Project**. Be sure to follow the directory structure used. Also verify 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.

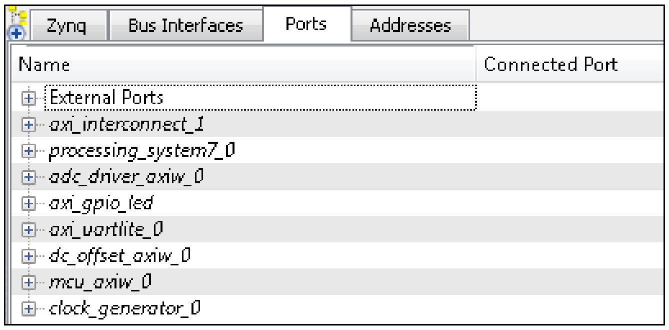
Configure Cores and Export Design Step 3

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
- 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
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	
clock_generator_0	

Figure 3-1: EDK project ports list

3.2 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. The first three are the `rx_iq_sel`, the `rxd` and the `bliky_adc_driver` 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 created in Simulink.

3. The `sysgen_clk` pin can be skipped for now and will be connected later in **section 3.4.**

3.3 Configuring the MCU Port

Expand the **MCU** port. There are 8 individual I/O pins which need to be routed on this port.

1. Configuring this port is very simple as all of the pins with the exception of the `sysgen_clk` are assigned as E**xternal ports**.

3.4 Configuring the dc offset Port

Expand the **dc_offset** port. There are 3 individual I/O pins which need to be routed on this port.

1. Ignore the Sysgen_clik pin for now. Verify that the `i_in` and `q_in` pins are connected to the adc driver as a result of step 3.2.

3.5 Configuring the GPIO Port

This port will be used later in SDK to verify the functionality of the receiver.

1. Add an AXI General Purpose IO to the design. Check the box to Enable Channel 2 and give each channel a width of 1 bit. Name the port `axi_gpio_led` or something similar.

2. Expand the `IO_IF` section of the GPIO, and assign the `GPIO_IO` and `GPIO2_IO` pins to external ports. The other pins can be left blank.

3.6 Configuring the AXI UART (LITE) Core

This UART port will be the serial port that is used for communication with the Chilipepper board.

1. Configure the UART settings with the Following: Baud Rate of 9600, 8 Data Bits per Frame, No parity, and Even Parity Type.

2. Make sure the RX and TX pins are configured properly as External Ports.

3.7 Configuring the Clock Generator IP Core

The Clock Generator is used in this project to distribute the appropriate clock signals to each of the PCores, as well as any external hardware which may require a clock signal. For this project, the Clock Generator is sourced from the 40 MHz `pll_clk_out` on the Chilipepper radio board (as described in the **Chilipepper user's guide**).

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

- Input Clock Frequency of **40Mhz**
- CLKFBIN Required Frequency of **40Mhz** with **no Clock Deskew**
- CLKFBOUT Required Frequency of **40Mhz**, Required Group **PLLE0**, and **Buffered True**
- CLKOUT0 Required Frequency of **20MHz**, 0Phase, **PLLE0** group and **Buffered true**
- CLKOUT1 Required Frequency of **40MHz**, 0Phase, **PLLE0** group and **Buffered true**
- CLKOUT2 Required frequency of **40Mhz**, **90 Phase**, **PLLE0** group and **Buffered true**
- CLKOUT3 Required Frequency of **40MHz**, 0Phase, **PLLE0** group and **Buffered true**

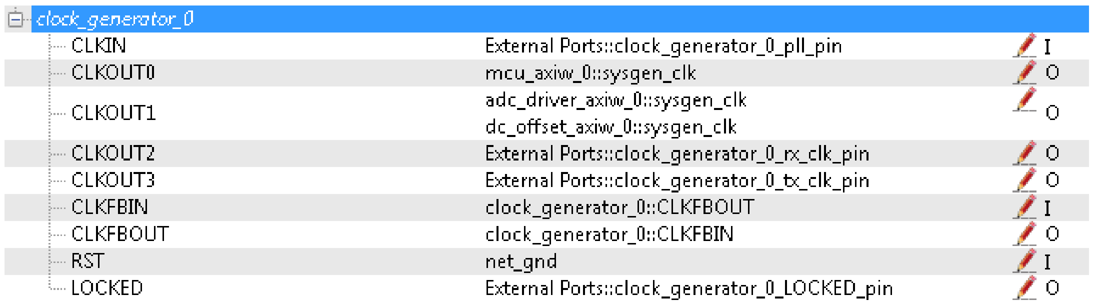
Now that the settings are configured you should have several clocks in your clock generator list.

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

CLKIN	→	External Ports
CLKOUT0	→	mcu::sysgen_clk
CLKOUT1	→	adc_driver::sysgen_clk && dc_offset::sysgen_clk
CLKOUT2	→	External Ports
CLKOUT3	→	External Ports
CLKFBIN	→	CLKFBOUT
RST	→	net_gnd
LOCKED	→	External Port

Note The `CLKOUT3` pin has a 90 degree phase shift as mentioned in the **Chilipepper user's guide**. This line will be used as the `RX_CLK` signal from the FPGA to the radio board.

Your Clock Generator port should look similar to Figure 3-2.



clock_generator_0		
CLKIN	External Ports::clock_generator_0_pll_pin	I
CLKOUT0	mcu_axiw_0::sysgen_clk	O
CLKOUT1	adc_driver_axiw_0::sysgen_clk dc_offset_axiw_0::sysgen_clk	O
CLKOUT2	External Ports::clock_generator_0_rx_clk_pin	O
CLKOUT3	External Ports::clock_generator_0_tx_clk_pin	O
CLKFBIN	clock_generator_0::CLKFBOUT	I
CLKFBOUT	clock_generator_0::CLKFBIN	O
RST	net_gnd	I
LOCKED	External Ports::clock_generator_0_LOCKED_pin	O

Figure 3-2: Clock Generator port configuration

3.8 Pin Assignments

Once the clock generator is configured correctly, the sysgen clock for the other 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 3-3 shows the names used in this Lab, 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 3-3 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[4] 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

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

steps to perform are identical to those in the previous labs; however they must be performed for **both** 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.

```
###################################################### 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"
```

Figure 3-3: 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 receive tone project.

1. Select **File → New → Application Project**.
2. Name the project `dc_offset` or something similar and leave the other settings at their defaults. Be sure to select **Hello World** from the **Select Project Template** section.
3. Click **Finish**. You should now see your project folder, as well as a **board support package** (bsp) folder.
4. If you navigate into the project folder, and into the src folder, you should see a `helloworld.c` file. This is the file we will be using to create our software design. Feel free to give the file a more descriptive name such as `main.c` or something similar.
5. **Double click** the file to open it and **replace** all of its contents with the code in Figure 4-1.

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.

```
#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;

int SetupPeripherals( void );

int main()
{
    int aliveLed = 0;
    static int BlinkCount = 0;
    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)
    {
         Chilipepper_ControlAgc(); //update the Chilipepper AGC
         BlinkCount += 1;
         if (BlinkCount > 1000000)
         {
             if (aliveLed == 0)
                  aliveLed = 1;
             else
                  aliveLed = 0;
             BlinkCount = 1;
             XGpio_DiscreteWrite(&gpio_blinky, 2, aliveLed);  //blink LEDs
             XGpio_DiscreteWrite(&gpio_blinky, 1, ~aliveLed);
         }
    }
    cleanup_platform();
    return 0;
}

int SetupPeripherals( void )
{
    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);
    return XST_SUCCESS;
}
```

Figure 4-1: Main C function for dc_offset project

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 8-12 of the Chilipepper.c file to specify which cores you will be using. Your code should resemble the following, as we will only be using the MCU_UART, MCU_DRIVER and DC_OFFSET cores for this Lab.

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

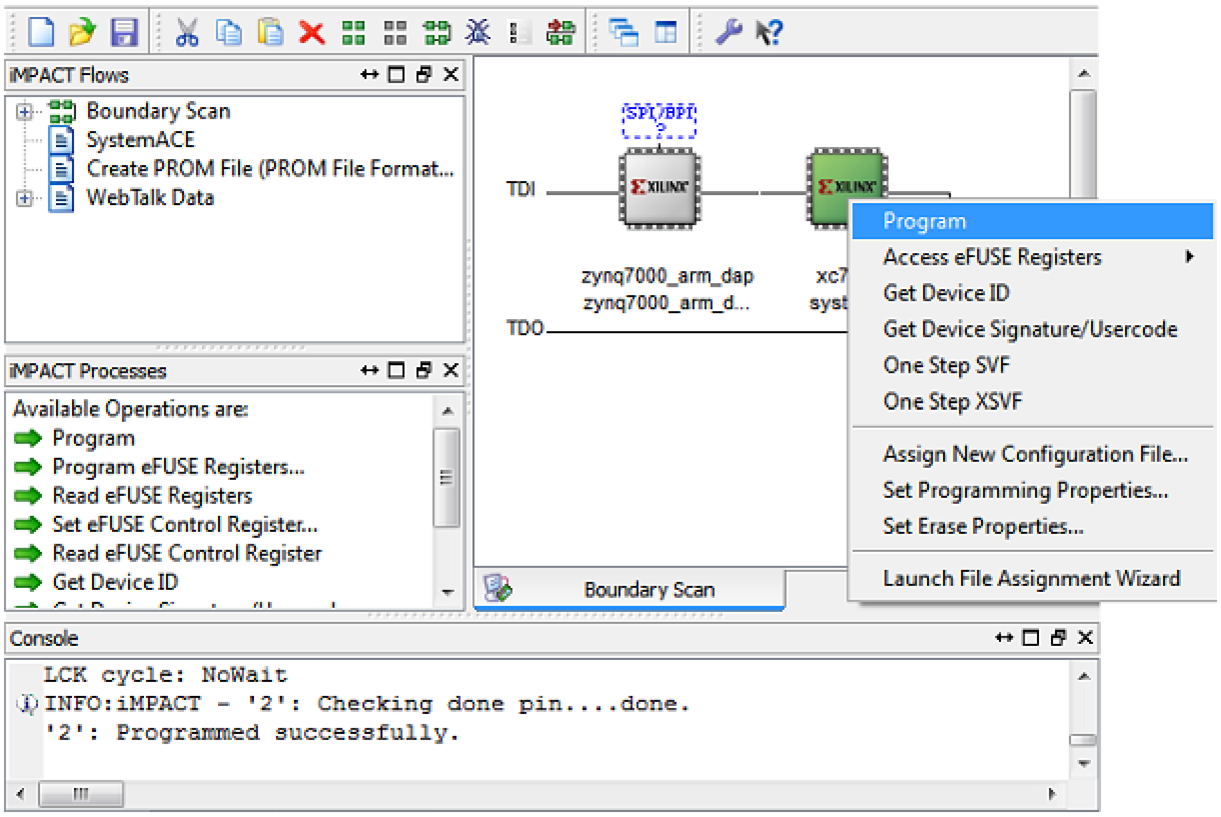
If you are still not able to compile your C design due to include errors, you may need to tell SDK where your PCore drivers are stored. If you click on Xilinx Tools → Repositories, you can specify (in Global Repositories) where the EDK directory of your project is. (This will need to be changed for each 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 or transmitting over the air by connecting an antenna to each board. 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. If you do not have a second board, you can still run the design, however you will likely only see noise from the output of your ADC.

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.

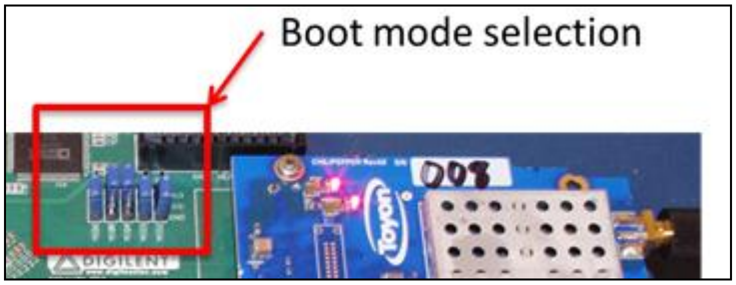


Figure 4-3: Jumper configuration needed to load a project via SD card

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.

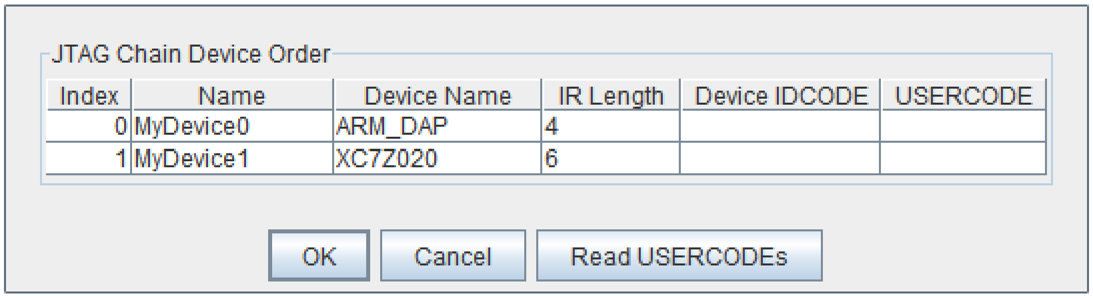
Testing and Design Verification Step 5

5.1 Verification with ChipScope Pro

There are several methods available for verifying the received QPSK transmission. This lab focuses on verification using ChipScope Pro, as well as exporting to MATLAB for further analysis.

1. To verify the received signal, you will need to open **ChipScope Pro Analyzer**. Be sure that the JTAG cable is connected to the FPGA board properly (or the 6 pin output of the Chilipepper).

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		

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

4. Click **Select New File**, and browse to the location of your ChipScope **CDC file**, which is located in the Sysgen/netlist folder of your project directory. This file was created for you when you generated your PCores from your Simulink Model design. It tells the ChipScope program how to interpret the data it is receiving from the JTAG port.

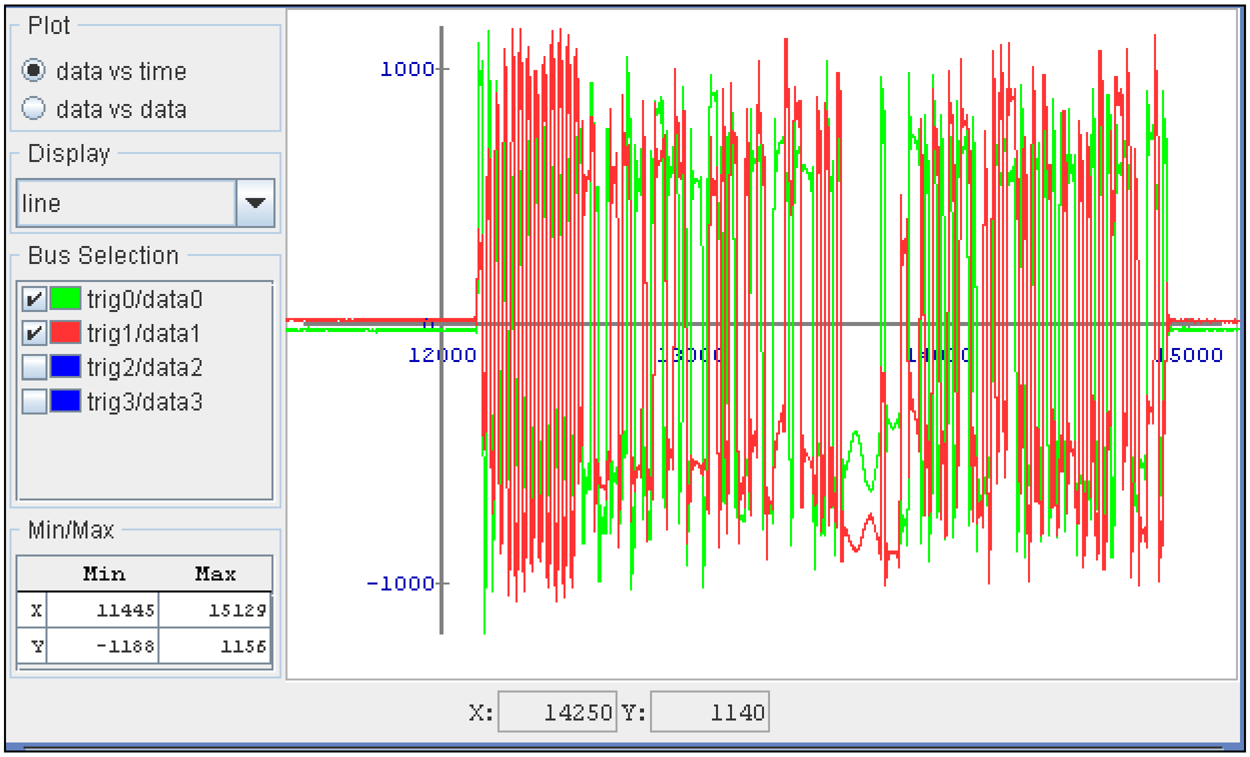
5. Next double click on the **bus plot** option in the New Project menu in the top left hand side of the screen. This will open a window which allows you to view a signal **value vs. time** plot of your waveforms.

6. Under Data Port in the Signals Dev menu on the left side of the screen, right click on each of the ports, and change their **bus radix** to **signed decimal**. Click OK to accept the default decimal values.

7. On the Bus Plot screen, you can change the color of each of the signals to get a better view of each individual signal. Click the **check box** next to any of the signals you wish to see on the plot.

8. To correctly view the received the signal in ChipScope, you must catch the signal in the window which is currently being viewed in ChipScope. This is much easier to do if you flip the switch on the Lab 3 demo to allow for continuous packet transmission.

9. Click the **play button** in the top menu bar until you get a full display of the signal. Additionally you can set up triggering options for periodic or continuous playback of the received signal. Your received signal should look similar to Figure 5-1 below.



Plot: [✓ data vs time] data vs data

Display: line

Bus Selection:
- [✓] 🟩 trig0/data0
- [✓] 🟥 trig1/data1
- [ ] 🟦 trig2/data2
- [ ] 🟦 trig3/data3

Min/Max:

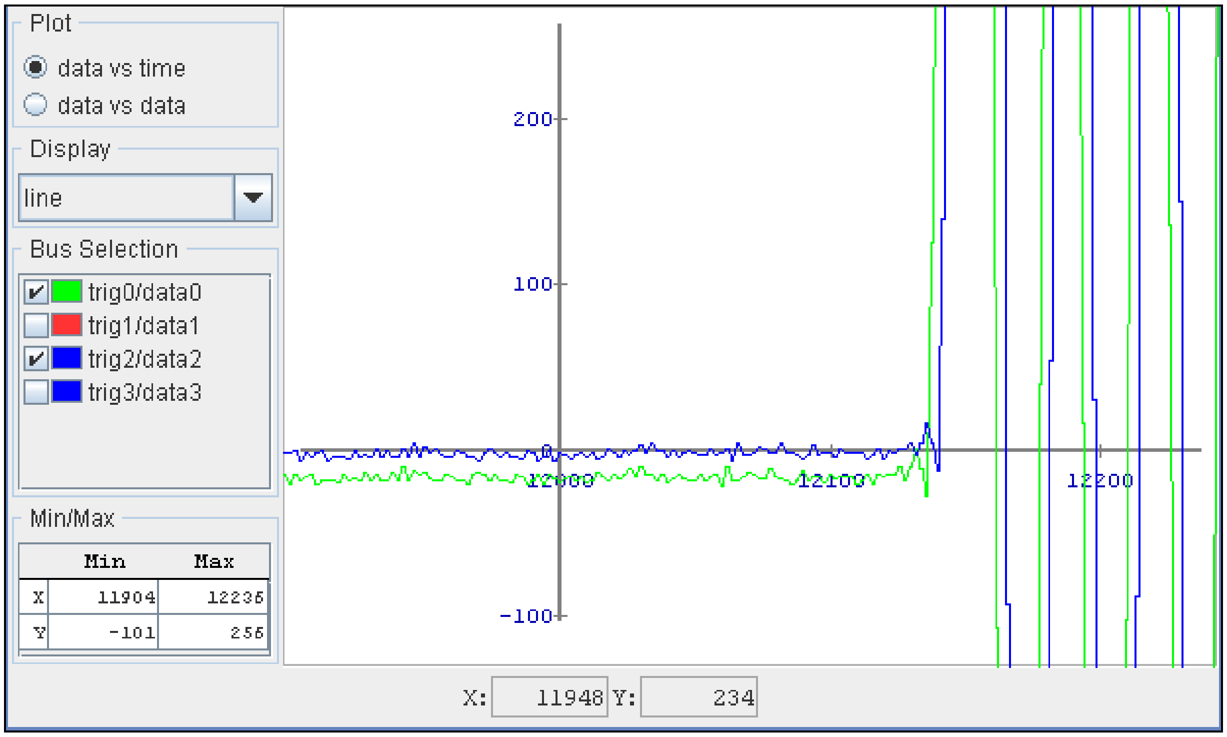
	Min	Max
X	11445	15129
Y	-1188	1156

X: 14250 Y: 1140

5-1: The received QPSK waveform in ChipScope Pro

This is the original received signal from the adc, and is before any of DC Offset Correction is applied. To see the signal post processing, change the bus plot view to see the data2 and data3 values. Be sure to set the correct bus radix for your signal. If you compare the I and q channels respectively pre and post DC correction, your results should resemble Figure 5-2 below.

Note	If you are not able to get the results shown above, the debug outputs d1-d9 can be used to verify the results of the design. If you follow along with the logic of the `dc_offset_correction.m` you should see a corresponding output in ChipScope.



Plot
[✓ data vs time] data vs data

Display
line

Bus Selection

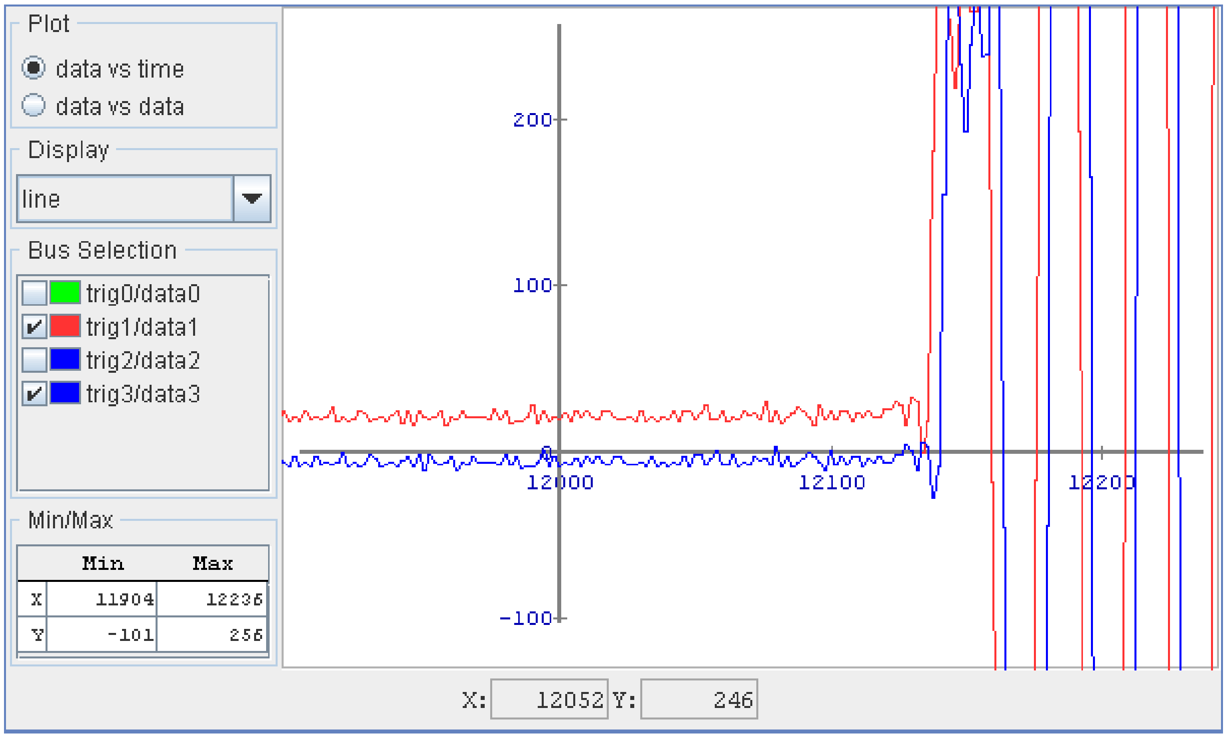
✓	🟩	trig0/data0
	🟥	trig1/data1
✓	🟦	trig2/data2
	🟦	trig3/data3

Min/Max

	Min	Max
X	11904	12236
Y	-101	256

X: 11948 Y: 234

Figure 5-2: Plot of the i channel for pre (green) and post (blue) DC Offset correction.



Plot
[✓ data vs time] data vs data

Display
line

Bus Selection

	🟩	trig0/data0
✓	🟥	trig1/data1
	🟦	trig2/data2
✓	🟦	trig3/data3

Min/Max

	Min	Max
X	11904	12236
Y	-101	256

X: 12052 Y: 246

5-3: Plot of the q channel for pre (red) and post (blue) DC Offset correction.

Appendix A MATLAB Test Bench

MATLAB function `dc_offset_correction.m`

```
function [i_out, q_out, rssi_out, rssi_en_out, dir_out, dir_en_out, ...
    d1, d2, d3, d4, d5, d6, d7, d8, d9] = ...
    dc_offset_correction(i_in, q_in, alpha_in, gain_en_in, ...
        rssi_low_goal_in, rssi_high_goal_in, rx_en_in)

persistent i_dc q_dc i_mean q_mean
persistent counter rssi_sum
persistent dir_state
persistent rssiHold
persistent noise_offset noise_inc noise_dec

alpha = alpha_in/2^12;

if isempty(i_dc)
    i_dc = 0;
    q_dc = 0;
    i_mean = 0;
    q_mean = 0;
    counter = 0;
    noise_inc = 0;
    noise_dec = 0;
    noise_offset = 0;
    rssi_sum = 0;
    dir_state = 0;
    rssiHold = 0;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DC Correction section
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if rx_en_in == 1
    i_mean = (1-alpha)*i_mean + alpha*i_in;
    q_mean = (1-alpha)*q_mean + alpha*q_in;

    i_dc = (1-alpha)*i_dc + alpha*i_in; %update the i dc offset
    q_dc = (1-alpha)*q_dc + alpha*q_in; %update the q dc offset

    if abs(i_mean) > (50 + noise_offset)  % too much noise, raise cieling.
        noise_inc = noise_inc + 1;
        i_dc = 0;
    else
        noise_dec = noise_dec + 1;
    end
    if abs(q_mean) > (50 + noise_offset)  % too much noise, raise cieling.
        noise_inc = noise_inc + 1;
        q_dc = 0;
    else
        noise_dec = noise_dec +1;
    end
    if (noise_inc > 10)
        %there is a high dc_offset value that needs to be corrected
        noise_offset = noise_offset + 10;
        noise_inc = 0;
    end
    if (noise_dec > 100000)
        %dc offset threshold is higher than needed
        noise_offset = noise_offset - 10;
        noise_dec = 0;
    end
end
i_out = i_in - i_dc;
q_out = q_in - q_dc;
%correct false positive/nagatives
if (abs(i_mean) < 50)
    noise_inc = 0;
end
if (abs(i_mean) > noise_offset - 10)
    noise_dec = 0;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% RSSI Estimation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
rssi_inst = i_out*i_out + q_out*q_out;
rssi_en_out = 0;
rssi_out = 0;

if rx_en_in == 1
    if counter == 0 && rssi_inst > 2*50*50
        counter = 1;
        rssi_sum = 0;
    end

    if counter ~= 0
        if rssi_inst < 2*50*50
            counter = 0;
        else
            counter = counter + 1;
            rssi_sum = rssi_sum + rssi_inst;

            if counter >= 2^8
                counter = 0;
                rssi_out = round(rssi_sum/2^8);
                rssiHold = rssi_out;
                rssi_en_out = 1;
            end
        end
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Gain Correction
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
dir_out = 0;
dir_en_out = 0;

% dir_out = 0 - do nothing
% dir_out = 1 - increase
% dir_out = 2 - decrease

ai = abs(i_in);
aq = abs(q_in);
% only increase power if the rssi is away from the mean
rssi_diff = abs(rssiHold-(i_mean*i_mean+q_mean*q_mean));
if rx_en_in == 1
    switch dir_state
        case 0 % wait for some action and the processor is done
            if gain_en_in == 0
                if rssi_en_out == 1
                    if rssi_diff < rssi_low_goal_in %too low - increase
                        dir_out = 1;
                        dir_en_out = 1;
                        dir_state = 1;
                    end
                    if rssi_diff > rssi_high_goal_in %too high - decrease
                        dir_out = 2;
                        dir_en_out = 1;
                        dir_state = 1;
                    end
                end
                % we're saturating the ADC so decrease gain
                % this overrides anything else
                if (ai > 1500) || (aq > 1500)
                    dir_out = 2; % decrease
                    dir_en_out = 1;
                    dir_state = 1;
                end
            end
        case 1 % see if the MCU has done something and if so reset
            if gain_en_in == 1
                dir_out = 0;
                dir_en_out = 1;
                dir_state = 0;
            end
        otherwise
            dir_state = 0;
    end
end
d1 = i_in;
d2 = i_mean;
d3 = i_dc;
d4 = rssiHold;
d5 = rssi_diff;
d6 = dir_state;
d7 = gain_en_in;
d8 = dir_out;
d9 = dir_en_out;
```

Appendix B MATLAB DC Offset Correction

MATLAB script `dc_offset_correction_tb.m`

```
clear all

fid = fopen('dc.prn');
M = textscan(fid,'%d %d %d %d %d %d %d %d %d %d %d %d %d %d %d
%d','Headerlines',1);
%M = textscan(fid,'%d %d %d %d','Headerlines',1);
fclose(fid);
is = double(M{3});
qs = double(M{4});

l = zeros(1,length(is));

figure(1)
clf
subplot(2,1,1)
plot(is)
hold on
plot(l,'g');
title('Original: Inphase');
subplot(2,1,2)
plot(qs)
hold on
plot(l,'g');
title('Original: Quadrature');

disp(['Original Mean I: ',num2str(mean(is)),' Mean Q: ',num2str(mean(qs))]);


% L is the recovery time of the filter, i.e., it takes about that many
% samples for it to recover.
% Becuase an averaging COMB filter is essentially a high pass filter it is
% best to have an offset in frequency above DC in order to be able to keep
% the filter length as short as possible
Ns = length(is);
io = zeros(1,Ns);
qo = zeros(1,Ns);
rssi = zeros(1,Ns);
ff_in = 2^3/2^12*2^12;
rssiH = 0;
for i1 = 1:length(is)
    i_in = is(i1);
    q_in = qs(i1);

    [i_out, q_out, rssi_out, rssi_en_out, dir_out, dir_en_out, ...
    d1, d2, d3, d4, d5, d6, d7, d8, d9] = ...
        dc_offset_correction(i_in, q_in, 1, mod(i1,2), 500, 1500, i1>3000);
    io(i1) = i_out;
    qo(i1) = q_out;
    if rssi_en_out
        rssiH = rssi_out;
    end
    rssi(i1) = rssiH;
end

figure(2)
clf
subplot(2,1,1)
plot(io)
hold on
plot(l,'g');
title('Corrected: Inphase');
subplot(2,1,2)
plot(qo)
hold on
plot(l,'g');
title('Corrected: Quadrature');

disp(['Corrected Mean I: ',num2str(mean(io)),' Mean Q: ',num2str(mean(qo))]);

figure(3)
clf
s = complex(io,qo);
plot(s.*conj(s),'r')
hold on
plot(rssi,'b');
title('rssi');
```