Toyon Research Corporation

Lab 1: Output Tone

Chilipepper Tutorial Projects

Version 0.5
12/14/2012

Table of Contents

Lab 1: Output Tone

Introduction

This lab will show you how to output a single frequency tone on an FPGA Mezzanine Card (FMC) radio board using the Xilinx Zed Board FPGA and the Toyon Chilipepper FMC. The components used to generate the tone will be created in hardware on the FPGA using the "black box" approach used in lab 0. The FMC initialization and microcontroller (MCU) signal control will be handled in software using the Xilinx Software Development Kit (SDK). This lab assumes prior knowledge of the workings of HDL Coder as well as the Xilinx EDK environment. It is recommended that you complete lab 0 before completing this lab.

This lab is created using:

- MATLAB 2012b
- Xilinx ISE Design Suite 14.3 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

Objectives

After completing this lab, you will be able to:

- Translate an algorithm from MATLAB code into an FPGA PCore design
- Create a Simulink model to control hardware on an FMC
- Create a software application to test your created FPGA hardware

---

Note Please refer to lab 0 to ensure that you have properly configured **System Generator** with your installation of MATLAB **before** starting this lab.

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 Files

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 function used is shown in Figure 1-1.

```
%#codegen
function [ i_out , q_out ] = output_tone()
%Output a tone at 1MHz

    persistent i_hold q_hold phi

    if (isempty(phi))
        phi=1;
    end

    %ROMs *not* declared persistent
    lSin = SIN;
    lCos = COS;

    q_hold = lCos(phi);
    i_hold = lSin(phi);

    phi = phi + 1;
    if phi > 20
        phi = 1;
    end

    i_out = (2^11)*i_hold;
    q_out = (2^11)*q_hold;

end
```

Figure 1-0-1: MATLAB algorithm for 1 MHz tone generation

In the above algorithm, `phi` is an offset used to grab the correct value from the sine and cosine look up table. The `i_out` and `q_out` outputs are the signals that are sent directly to the DAC. Since the DAC on the Chilipepper is 12 bits, these outputs are multiplied by a scaling factor to allow for the full 12 bit resolution (12 bit 2's Compliment).

1. Create a directory for the project under C:\QPSK_Projects\Project_1.
2. Create a new **MATLAB function** with the contents of Figure 1-1.
3. **Save** this function as `output_tone.m` inside the project directory.

There are two lines in Figure 1-1 which refer to `SIN` and `COS` variables. Since the tone generation will be implemented in hardware, we will pre-define our sine and cosine variables and save them into MATLAB files. These variables will be used as **lookup tables** to correctly output a sine and cosine waveform from within our Simulink model. To do this an **additional MATLAB function** must be created and run prior to creating the core in workflow advisor. The code used for this function is shown in Figure 1-2.

```
function make_trig_lut

    % Generate LUT values
    ii = 0:(20-1);
    c = cos(2*pi*ii/20);
    s = sin(2*pi*ii/20);

    % Create cosine LUT
    fid = fopen('COS.m','w+');
    fprintf(fid,'function y = COS\n');
    fprintf(fid,'%%#codegen\n');
    fprintf(fid,'y = [\n');
    fprintf(fid,'%14.12f\n',c);
    fprintf(fid,'];\n');
    fclose(fid);

    % Create sine LUT
    fid = fopen('SIN.m','w+');
    fprintf(fid,'function y = SIN\n');
    fprintf(fid,'%%#codegen\n');
    fprintf(fid,'y = [\n');
    fprintf(fid,'%14.12f\n',s);
    fprintf(fid,'];\n');
    fclose(fid);

end
```

Figure 1-0-2: MATLAB code for creating sine and cosine lookup tables

4. Create a new **MATLAB function** with the contents of Figure 1-2.
5. **Save** this function as `make_trig_lut` inside the project directory.

Note This function will create **two** more MATLAB files called `SIN.m` and `COS.m`. By default, these files are placed in the MATLAB folder. Place these files in the same directory as your MATLAB function, and be sure this directory is in the MATLAB PATH variables.

Both the `SIN.m` and `COS.m` files are created by sampling a pure sine and cosine wave at some pre-calculated interval. The value of this interval should be proportional to the frequency of the wave we wish to generate, the sampling rate of our tone generator, and the offset (phi) we wish to use in our lookup table (LUT). For this lab, we want to make the LUT only as big as needed for a **1MHz tone**; therefore we want an **offset of 1**. This means our sine and cosine waves should be sampled at a rate of $\frac{Ftone}{Tone\ Generator\ Sampling\ Rate} = \frac{1MHz}{20MHz}$.

The last MATLAB code required for this core is the test bench script. This script is required for HDL generation, but also allows you to test the functionality of the MATLAB algorithm. The code used for this script is shown in Figure 1-3.

```
clear all;

Fs = 20e6;      % Sampling frequency (tone - 20MHz)
L = 500;      % Length of signal
t = (0:L-1)/Fs; % Time vector

make_trig_lut;

for i1 = 1:L
    [I(i1), Q(i1)] = output_tone();
    y(i1) = I(i1) - Q(i1);
end

subplot(2,1,1);
plot(t,y);
title('y(t)')
xlabel('time (milliseconds)')

NFFT = 2^nextpow2(L); % Next power of 2 from length of y
Y = fft(y,NFFT)/L;
f = Fs/2*linspace(0,1,NFFT/2+1);

% Plot single-sided amplitude spectrum.
subplot(2,1,2);
plot(f,2*abs(Y(1:NFFT/2+1)))
title('Single-Sided Amplitude Spectrum of y(t)')
xlabel('Frequency (Hz)')
ylabel('|Y(f)|')
```

Figure 1-0-3: MATLAB code for HDL test bench script

6. Create a new **MATLAB script** with the contents of Figure 1-3.
7. **Save** this script as `output_tone_tb.m` inside the project directory.
8. **Run** this script in MATLAB (be sure the project directory is in the MATLAB PATH variable) to test the algorithm and the lookup table. You may have to run the `make_trig_lut.m` function first to create the lookup tables.
9. Once you have verified that your algorithm is correct, proceed to the next step of the lab.

1.2 HDL Coder Project

Using the same steps outlined in Lab 0, create a new HDL coder project called Tone. Add both your `output_tone.m` file and your `output_tone_tb.m` file to the **MATLAB Function** and **MATLAB Test Bench** categories respectively. The two lookup table files will be automatically added to the project later; just be sure they are located in the same directory as your output tone files.

1. Once inside the workflow advisor screen, configure the **Clocks & Ports** settings as well as the **Black Box Generator** settings (See lab 0).
2. Right-click **Propose Fixed-Point Types,** and select **Run to Selected Task.** For this Lab, the values of your "Type" column should resemble the settings below in Figure 1-4.

	Min	Max	IsInt	Type	RoundMode	OverflowMode
i_hold	-1.0	1.0	☐	sfix14_En12	floor	wrap
i_out	-2048.0	2048.0	☐	sfix12_En0	floor	saturate
ICos	-1.0	1.0	☐	sfix14_En12	floor	wrap
ISin	-1.0	1.0	☐	sfix14_En12	floor	wrap
phi	1.0	21.0	☑	ufix5	floor	wrap
q_hold	-1.0	1.0	☐	sfix14_En12	floor	wrap
q_out	-2048.0	2048.0	☐	sfix12_En0	floor	saturate

Figure 1-0-4: Variable types for Tone MATLAB algorithm

As shown in Figure 1-4, the `i_out` and `q_out` variables are set to `sfix12_En0`. This formats these variables as **12 bit signed values** with 0 bits used for a decimal value. You can change this format if you would like your precision to include decimal values, however remember to change the multiplier used in the MATLAB algorithm accordingly to ensure you get full **12 bit resolution**.

3. Set the **Overflow Mode** to **saturate** as shown. In our example we used a multiplier of 2^{11} for our MATLAB algorithm; however due to the fact that 12 bit two's compliment can only represent a positive value up to $2^{11} - 1$, we cannot use the full range output from the algorithm.

4. Once you have corrected the **Type** setting for all your variables, right-click **Code Generation,** and select **Run to Selected Task**. A Simulink model containing your design should appear after the task is completed.

Create and export Simulink models Step 2

This section will show you how to customize your Simulink Models to generate the required sinusoids and control the Chilipepper MCU and DAC.

2.1 Modify Tone Simulink Design

Your Simulink design is a model which will eventually be exported as a PCore into your Xilinx EDK project. Any blocks you add to this model, including the one created from your MATLAB algorithm, will essentially be simulated as physical hardware components on the FPGA.

1. As in Lab 0, you first need to copy and paste the contents of the design into a **new Simulink Model**. Call this new model `tone.slx`, and **save** it in a directory structure similar to the one used in Lab 0 (**Sysgen folder**). Be sure to change the cfg file as well to find the files in your new directory structure.

2. Once you have created your new model, add blocks from the Xilinx blockset library to create a model similar to the one shown in Figure 2-1.

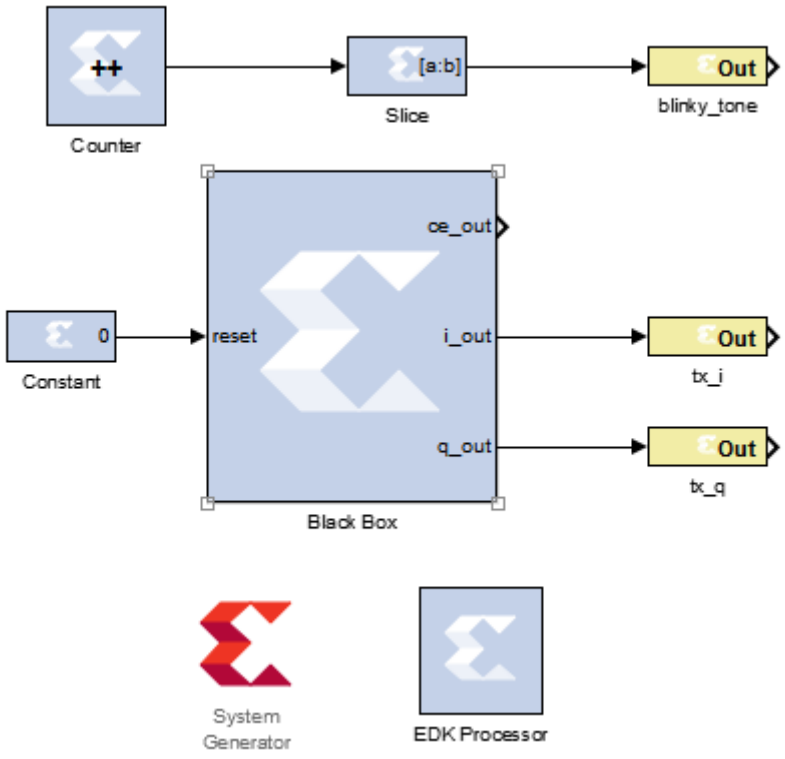


Figure 2-1: Simulink model for output_tone.m

3. Set the following parameters for the Constant block: **Value** is 0; **Output Type** Fixed-point; **Arithmetic type** Unsigned; **Number of bits** 1; **Binary point** 0; **Sampled constant** checked; **Sample period** 1.

4. Set the following parameters for the Counter block**: Counter type** Free running; **Count direction** Up; **Initial value** 0; **Step** 1; **Output type** Unisgned; **Number of bits** 25; **Binary point** 0; **Optional Ports** all unchecked; **Explicit period** 1.

5. Set the following parameters for the Slice block: **Width of slice** 1; **Boolean output** unchecked; **Specify range as** Upper bit location + width; **Offset of top bit** 0; **Relative to** MSB of input.

6. Add the **gateway out** pins, and name them similarly to the names given in Figure 2-1.

Note Before changing the settings in the System Generator, you should allow the system to connect to the **EDK Processor** using a method similar to that used in Lab 0. Add a **From Register** to the design to sync the EDK processor to, then remove it and **re-sync the processor** without it.

---

7. Now configure the **System Generator** as shown in Figure 2-2 below.

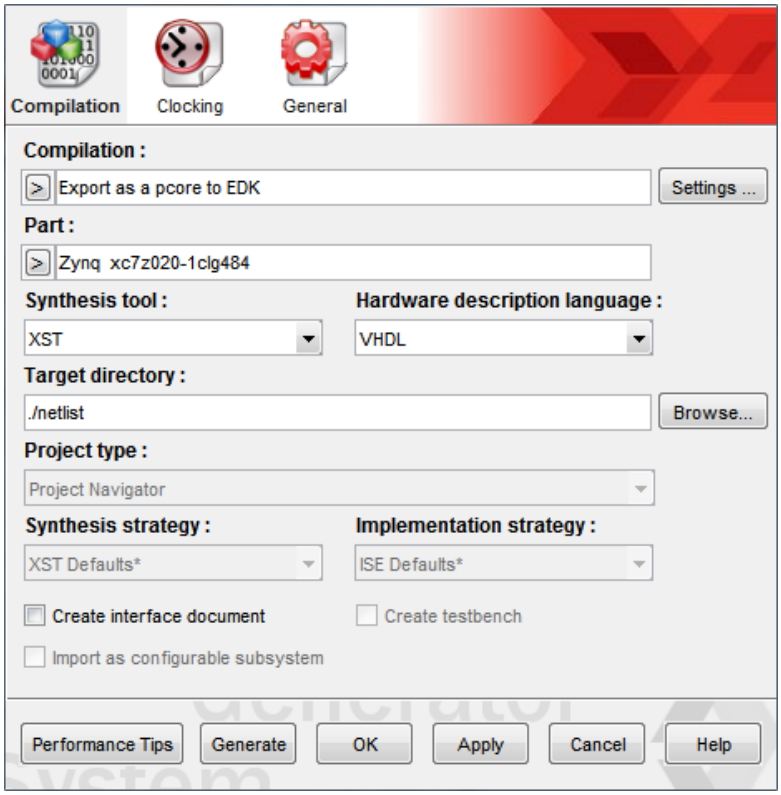


Figure 2-2: Settings for System generator

Don't worry about changing the settings of the compilation yet, as we must first create a new **EDK project** to export the PCore to. Simply apply the current changes, select ok and **save** the model.

2.2 Create MCU Simulink Design

This **Simulink model** will be used for the control signals to and from the **MCU**. The Model is shown in Figure 2-3 below.

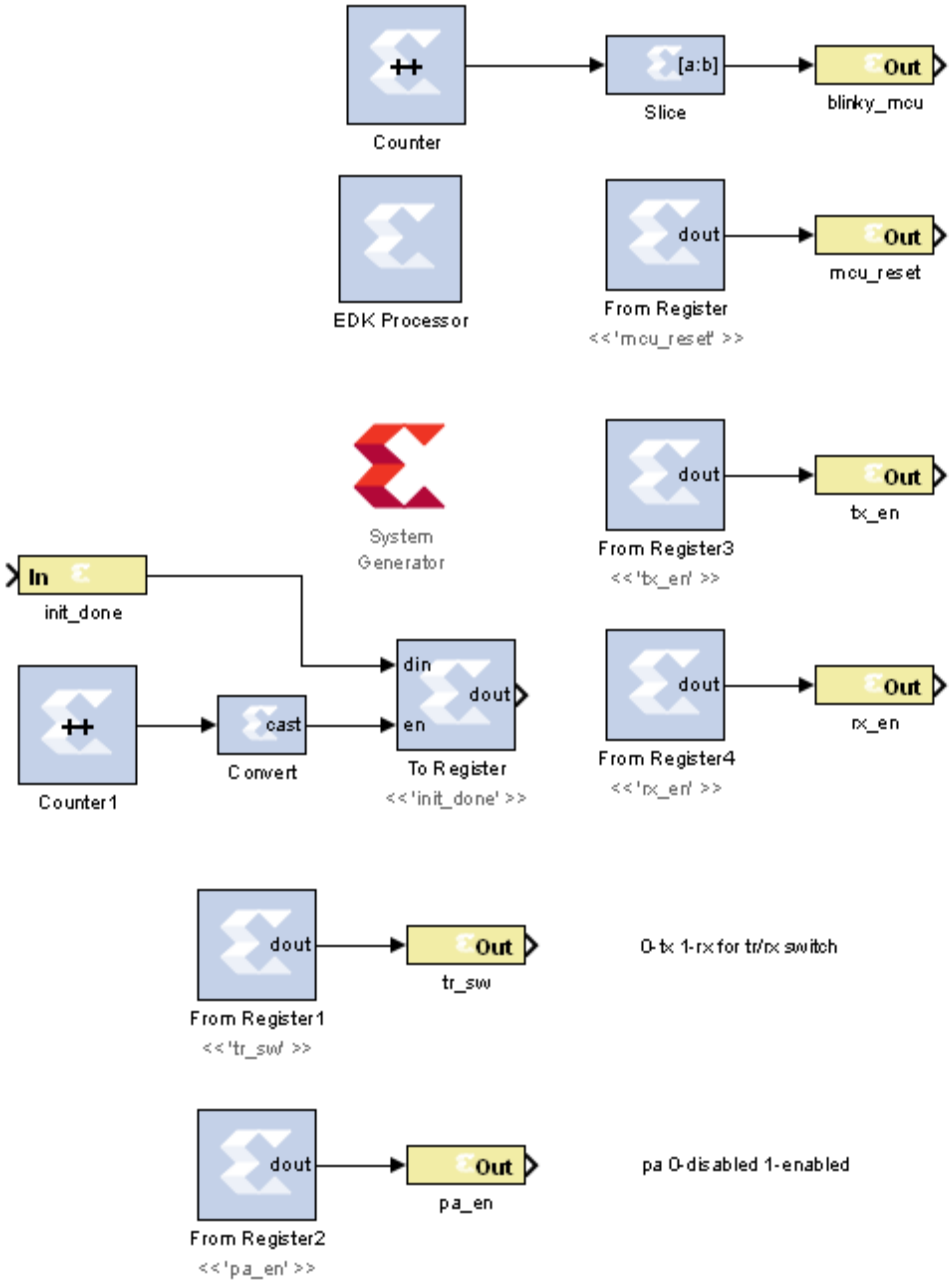


Figure 2-3: Simulink model for MCU control

1. Create the model shown in Figure 2-3, name it `mcu.slx` for convenience, and **save** it into the **Sysgen folder**. The info in the model about `pa_en` and `tx/rx` switch orientation is **not required** for the design.

2. Set the following parameters for the registers (to and from): **Basic Tab → Sample period** 1; **Output Tab → Output Type** Fixed-Point; **Arithmetic type** Unsigned; **Number of bits** 1; **Binary point** 0; In addition, the `init_done` register should have an **Initial value** of 0.

3. Configure the counter and slice blocks the same way as in the previous model (**section 2.1**).

4. Set the following parameters for Counter 1:**Counter type** Free running; **Count direction** Up; **Initial value** 0; **Step** 1; **Output type** Unsigned; **Number of bits** 1; **Binary point** 0; **Optional Ports** all unchecked; **Explicit period** 1.

5. Set the **Output type** of the Convert block to boolean. No other settings are required.

6. Set the following parameters for the `init_done` gateway in: **Output type** Fixed-point; **Arithmetic type** Unsigned; **Number of bits** 1; **Binary point** 0; **Quantization** Round; **Overflow** Saturate; **Sample period** 1.

7. Now, **sync** the registers to the EDK processor. Double click your EDK block, and **verify** it looks similar to Figure 2-4 below.

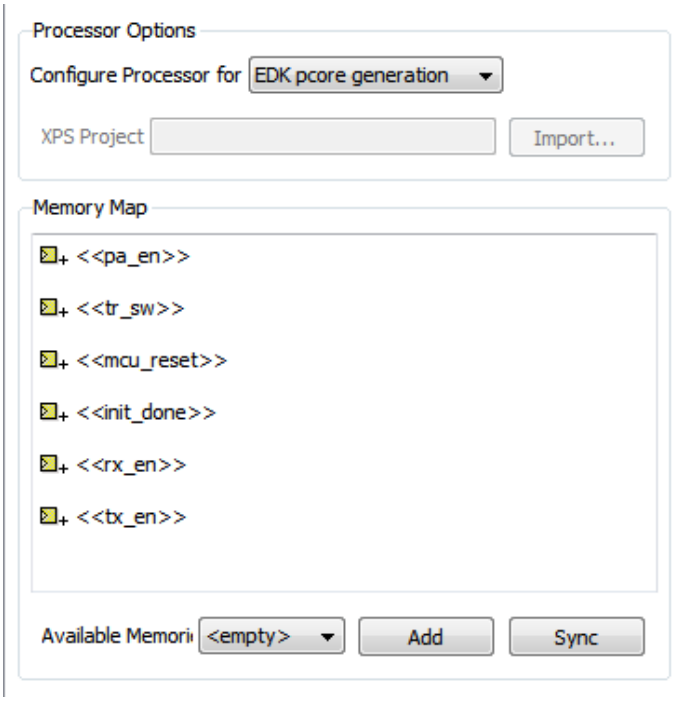


Figure 2-4: EDK Processor Configuration

Be sure all the registers are shown as memory mapped in the EDK processor. **Configure** the system generator the same as Figure 2-2, and **save** the design.

2.3 Create DAC driver Simulink Design

This **Simulink model** will be used for creating the signals which interface to the **DAC driver**. The model is shown in Figure 2-5 below.

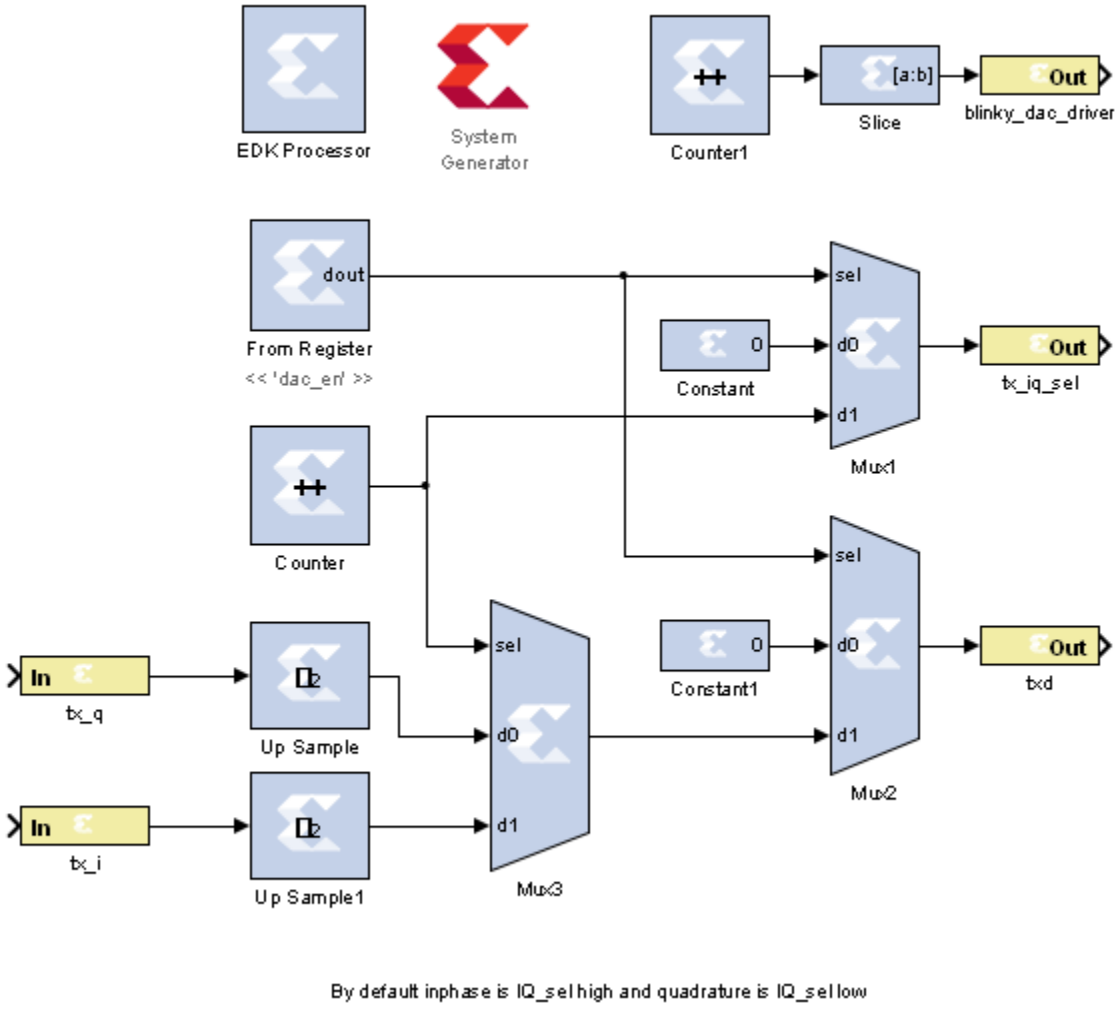


Figure 2-5: Simulink model for DAC control

1. Setup the **Counter** as a simple 1 bit **free running up counter** just like the one used in the MCU model.

2. **Counter1** and the **Slice** should also be configured identically to the ones used in the MCU model: a 25 bit **free running up counter**. As we will see later, this model has a faster system clock then the other 2 models, and therefore we should expect to see this LED blink faster than the other 2.

3. Configure the 2 **constants** as 1 bit, **fixed-point** with **no binary point** and a constant **sample period** of 1.

4. The **MUX's** are standard with **no changes required** from the default.

5. Configure the **up samplers** to have a **sampling rate** of 2, and make sure to check the box for **Copy Samples**. The **latency** should be left at 0.

6. Setup both **input gateways** to be **fixed-point 2's Compliment** numbers. They should have **12 bits** each and a **sample period** of 2. **Round** and **saturate** are fine for the **Quantization** and **Overflow** settings.

7. Once the design is ready, you must **sync** the `dac_en` from register to the **EDK processor**. **Double click** your EDK block, and **verify** the register is listed. **Sync** and **apply** changes.

8. Configure the System Generator the same as Figure 2-2, and **save** the design. You are now ready to create your EDK project and Export your models as PCore IP!

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

Configure cores and export design Step 3

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

3.1 Needed IP Cores

- DAC Driver PCore created in Simulink
- MCU PCore created in Simulink
- Tone PCore created In Simulink
- Clock Generator IP Core
- Processing System IP Core
- AXI Interconnect IP 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.

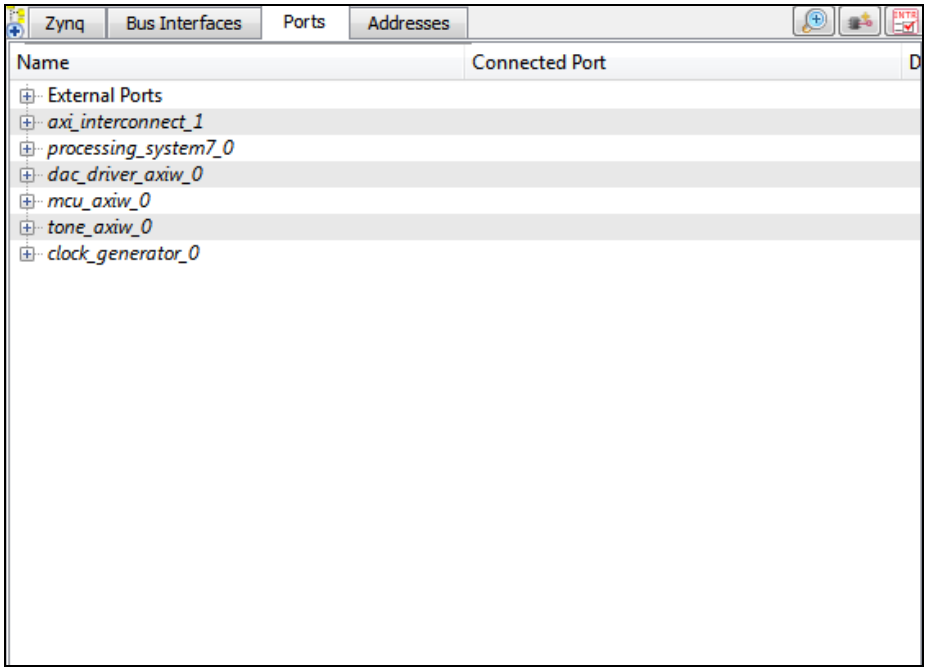


Figure 3-1: EDK project ports list

3.2 Configuring the DAC Driver Port

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

1. The first two are the `tx_i` and `tx_q` pins. These are input pins which are used to create the **TXD output** by interleaving an I and Q channel. The signals come directly from the lookup table created by the MATLAB algorithm. **Assign** these two pins to the `tx_i` and `tx_q` output pins from the **Tone PCore**.

2. Next are the `txd`, the `tx_iq_sel`, and the `blinky_dac_driver` output pins. These three pins are all signals which are **routed** directly to physical components on the FPGA as **external pins**. The `blinky_dac_driver` is sent to an LED on the FPGA, and the other two are sent to the **FMC** connector port and into the radio board. **Assign** these three pins as **external ports.**

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

3.3 Configuring the Tone Port

Expand the **Tone** port. There are 4 individual I/O pins which need to be routed on this port.

1. If the DAC Driver port was configured correctly in section 3.2, the Tone port should already have its `tx_i` and `tx_q` pins assigned. The only other pin to be assigned for this port is thus the `blinky_tone pin`, which can be **assigned** as an **external port**.

2. Again, skip the `sysgen_clk` for now as it is assigned later in **section 3.4**

3.4 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 simply **assigned** as **external ports**.

3.5 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**). This signal is then distributed to 4 other devices; 3 PCores (MCU, DAC and TONE) and the `TX_CLK` signal which latches data from the TXD line to the DAC on the radio board.

1. **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 **40MHz**, 0 Phase, **PLLE0** group and **Buffered true**
   - CLKOUT1 Required Frequency of **20MHz**, 0Phase, **PLLE0** group and **Buffered true**
   - CLKOUT2 Required frequency of **40Mhz**, **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 → External Ports
   - CLKOUT0 → dac_driver::sysgen_clk
   - CLKOUT1 → Tone::sysgen_clk **and** mcu::sysgen
   - CLKOUT2 → External Ports
   - CLKFBIN → CLKFBOUT
   - RST → net_gnd
   - LOCKED → External Ports

Note Although there are 4 sources which need to be clocked, **two** of the PCores use the same clock frequency, and can thus **share** the same clock signal from the Clock Generator. In addition, notice that the `CLKOUT2` pin has a 180 degree phase difference. This line will be used as the `TX_CLK`, and the phase shift will allow for data to be latched (rising edge by default) into the DAC **midway between** when the `TXD` value is changing. Also, it can be seen that the Tone and MCU clocks are only half the clock speeds of the DAC and the CLKIN pin. This is necessary for the creation of the `TXD` signal as both the I and Q channels must be **interleaved** into the signal at a rate **equal to** that of the `TXD_CLK` rate.

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

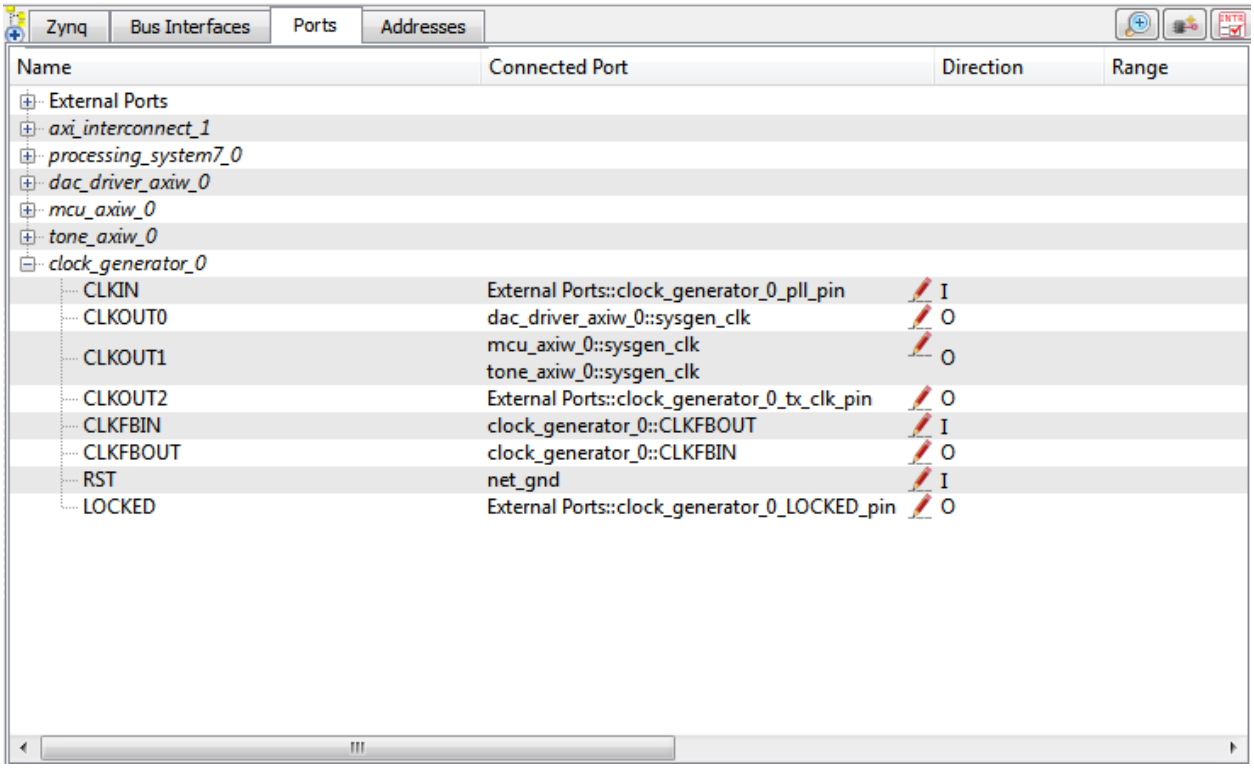


Figure 3-2: Clock Generator port configuration

3.6 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 below shows the pin assignments for this project.
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.

```
############################################## 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;
##############################################Tx – FMC interface at 2.5V ####################################
NET clock_generator_0_tx_clk_pin                LOC = C17       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_axiw_0_tx_iq_sel_pin             LOC = B16       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_axiw_0_txd_pin[0]                LOC = A18       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_axiw_0_txd_pin[1]                LOC = A19       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_axiw_0_txd_pin[2]                LOC = E20       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_axiw_0_txd_pin[3]                LOC = G21       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_axiw_0_txd_pin[4]                LOC = F19       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_axiw_0_txd_pin[5]                LOC = G15       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_axiw_0_txd_pin[6]                LOC = E19       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_axiw_0_txd_pin[7]                LOC = G16       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_axiw_0_txd_pin[8]                LOC = G19       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_axiw_0_txd_pin[9]                LOC = A16       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_axiw_0_txd_pin[10]               LOC = A17       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
NET dac_driver_axiw_0_txd_pin[11]               LOC = C18       | IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = FAST;
################################################## MCU Interface ########################################
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 clock_generator_0_LOCKED_pin                LOC = T22       | IOSTANDARD=LVCMOS33;  # "LD0"
NET mcu_axiw_0_blinky_mcu_pin                   LOC = T21       | IOSTANDARD=LVCMOS33;  # "LD1"
NET tone_axiw_0_blinky_tone_pin                 LOC = U22       | IOSTANDARD=LVCMOS33;  # "LD2"
NET dac_driver_axiw_0_blinky_dac_driver_pin     LOC = U21       | IOSTANDARD=LVCMOS33;  # "LD3"
```

Figure 3-3: EDK project pin assignments

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

At the time of this tutorial, Xilinx has a documented issue 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 for our project. The steps to perform are identical to those in the previous lab; however they must be performed for **all three** of the PCores used in this lab. Once the fix is applied, 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 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 tone generation project. Since our algorithms for the tone generation are written in MATLAB, and the mixing is handled by PCores, all we need to do in software is initialize the hardware.

Note It would be helpful if you have completed the Embedded System Design tutorial in the *ZedBoard AP SoC Concepts Tools and Techniques Guide*. In addition, you should be familiar with the *Chilipepper user guide* to ensure proper MCU Control.

---

1. Select **File → New → Project**.
2. Select **Xilinx C Project,** and hit next.
3. Name the project hello_world and leave the other settings at their defaults. Be sure to select **Hello World** from the **Select Project Template** section.
4. Click **Finish**. You should now see your hello world project folder, as well as a **board support package** (bsp) folder.
5. If you navigate into the hello world 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.
6. **Double click** the file to open it and **replace** all of its contents with the code in Figure 4-1.

```
#include <stdio.h>

int main()
 {



    while (1);

    return 0;
}
```

Figure 4-1: Code outline for SDK project

7. Next, we will write our Chilipepper initialization routine. The first thing we will do is create pointers to all the memory **registers** we created within our PCores. These can be defined as integer pointers and the names can be anything you like, but it's helpful if they are descriptive. An example of defining these pointers is shown in Figure 4-2.

```
// mcu registers
int *chili_init_done, *chili_pa_en, *chili_tr_sw;
int *chili_mcu_reset, *chili_rx_en, *chili_tx_en;

// DAC registers
int *dac_en;
```

Figure 4-2: Create pointers to PCore registers

8. These pointers must be initialized to the values of the actual memory address assigned in EDK. To find these values, select the **ports tab** of your previously compiled EDK project design. **Double click** on one of the PCores created earlier. In the From/To registers section, you will find the **base address** and **offset** of each of the registers used for that core. Use these values to correctly link each register to its memory location. For this project, this must be done for both the MCU and the DAC Driver PCores. An example of what the pointer initializations should look like is shown in Figure 4-3 below. Note that your addresses may be different.

```
  // MCU core
chili_init_done =    (int *)(0x61600000 + 0x800);
chili_pa_en =        (int *)(0x61600000 + 0x800);
chili_tr_sw =        (int *)(0x61600000 + 0x804);
chili_mcu_reset =    (int *)(0x61600000 + 0x808);
chili_rx_en =        (int *)(0x61600000 + 0x80C);
chili_tx_en =        (int *)(0x61600000 + 0x810);

// DAC core
dac_en =             (int *)(0x69800000 + 0x800);
```

Figure 4-3: PCore register Initializations

9. The last step of creating the software routine is to follow the appropriate initialization steps outlined in section 3.1 of the Chilipepper user guide.

   a. Disable the DAC

   b. Set `TX_EN` and `RX_EN` high

c. Bring `MCU_RESET` high then low then high

d. Wait for `MCU_INIT_DONE` to go high

e. Enable the DAC

The code for performing these steps is given in Figure 4-4.

```
// disable data going to DAC
*dac_en = 0;

// put RF front-end in state for transmit
*chili_pa_en = 1;    // active high
*chili_tr_sw = 0;    // 0-transmit, 1-receive

// enable Tr/Rx cores in the RFIC
*chili_rx_en = 1;
*chili_tx_en = 1;

// toggle reset on MCU - active low
*chili_mcu_reset = 1;
*chili_mcu_reset = 0;
*chili_mcu_reset = 1;

// wait for the MCU to finish calibration
while(*chili_init_done == 0)
        ;

// after MCU finished enable the DAC
*dac_en = 1;
```

Figure 4-4: Chilipepper Initialization Routine

4.2 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 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.

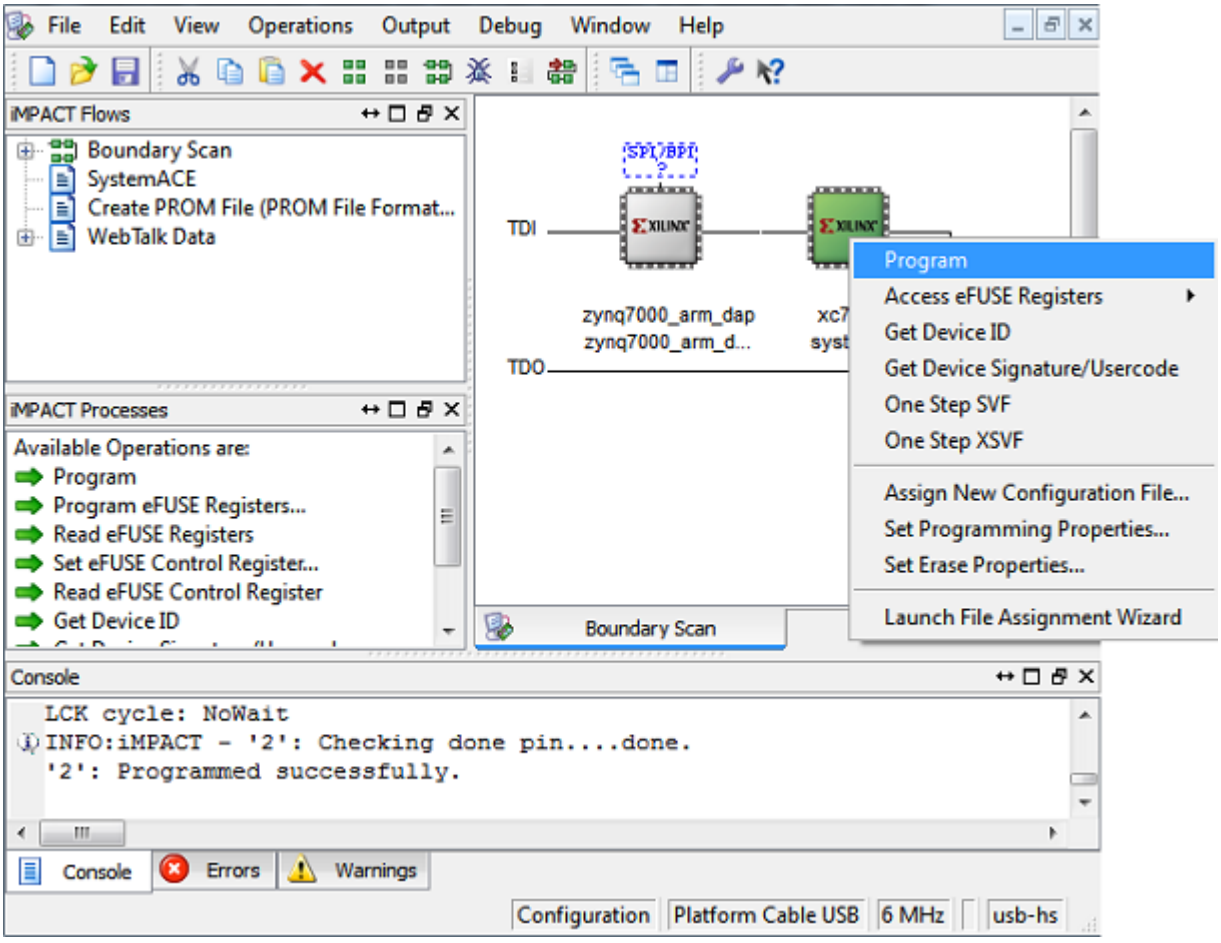
5. Select yes to allow iMPACT to automatically create a new project for you. If you receive any connection errors, verify your USB or JTAG programmer cables are connected properly.

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 4-5 below.

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



4-5: iMPACT configuration screen

4.3 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 can now return to the SDK project screen to test your software.

1. Test it by **right clicking** the `hello_world` project 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 blinking green light on the Chilipepper.

Verify that the output of the antennae is in fact a 2.4 GHz + 1 MHz sine wave using a spectrum analyzer.