Introduction

Power amplifiers (PAs) for third-generation (3G) wireless communication systems need high linearity at the PA output, to achieve high adjacent channel leakage ratio (ACLR) and low error vector magnitude (EVM). In addition, high efficiency is desirable. However when operating with high efficiency, PA’s are the most nonlinear. Digital predistortion (DPD) is an efficient cost-effective means of compensating for PA nonlinearity and retaining high efficiency.

The DPD reference design implements an adaptive lookup table (LUT) based algorithm—commonly used for linearization of power amplifiers (PA). The design constantly applies correction values from a LUT to the incoming stream of samples. It also compares the measured output with the input, and uses this measurement to update the LUT, making the system adaptive.

For 3G systems the DPD reference design can operate on up to four universal mobile telecommunications system (UMTS) channels and corrects 3rd and 5th order intermodulation products.

The reference design comprises a MATLAB system simulation environment, including a graphical user interface (GUI). This environment allows you to custom configure the reference design algorithm, experiment with other algorithms, and apply stimuli. The MATLAB model of the design is a bit-accurate representation of the Verilog HDL solution.

Altera supplies the reference design as Verilog HDL source code. The reference design includes a Nios® processor subsystem, and testbenches that allow you to test the Verilog HDL with data generated by the MATLAB system simulation environment.

Features

- MATLAB GUI that allows configuration of the design and test conditions, and displays graphs of the results
- Bit-accurate MATLAB model of a DPD design
- Nios processor subsystem using hardware acceleration
- Verilog HDL testbench that compares the MATLAB-generated results with the results from the RTL simulation
- Verilog HDL source code for Nios processor, co-ordinate rotation digital computer (CORDIC) reference design, and DPD logic
Purpose

The purpose of this reference design is to demonstrate that Altera® Stratix™ and Cyclone™ devices are suitable in performance and capacity to implement DPD solutions. The MATLAB simulation approach allows you to see that the DPD algorithm works, before you simulate and then synthesize the RTL for Stratix and Cyclone devices.

The DPD reference design is not a complete solution but provides a useful evaluation platform for you to start developing a complete solution for a real system.

Background

DPD is commonly used to linearize PAs. Ideal PAs are perfectly linear. Their response can be described with the equation, $V_{out} = kV_{in}$ (see Figure 1).

\[ V_{OUT} = kV_{IN} \]

Figure 1. Ideal PA

However, real PAs as used in wireless systems exhibit some nonlinearities and eventually reach saturation. This nonlinearity can be expressed by adding the term $f_{nl}$ into the equation, where $f_{nl}$ is used to describe the nonlinearity, $V_{out} = f_{nl}kV_{in}$ (see Figure 2).

\[ V_{OUT} = f_{NL} k V_{IN} \]

Figure 2. Real PA
The nonlinearity adversely affects the overall performance of a wireless system. It causes in-band distortion, which degrades the performance of the receiver and out-of-band distortion, which degrades the performance of receivers in adjacent channels.

The task of the predistorter is to add predistortion before the power amplifier, which is exactly the inverse of the distortion caused by the power amplifier (see Figure 3). When combining the predistorter with the power amplifier, the terms $f_{nl}$ and $f^{-1}_{nl}$ cancel out, and the overall system can be described by the ideal PA equation $V_{in} = kV_{out}$.

**Figure 3. DPD**

The nonlinearity of the PA is affected by ageing and changes in the operating environment, in particular the temperature. For this reason, the nonlinearity changes over time, and the solution should be made adaptive such that the predistorter tracks the changes in behavior of the PA.

**Figure 4** describes the basic algorithm implemented in the reference design. The incoming complex samples, in I and Q, have correction factors applied from the LUT and then sent to the radio frequency (RF) module. The addresses for the LUT are derived from the input power. The LUT must contain two values for each location—the real part I and the imaginary part Q.
In the RF module, samples are upconverted and sent to the PA. The PA output is downconverted, which allows you to measure the error, i.e., the difference between the input phase and magnitude, and the measured phase and magnitude. Obviously, the delay blocks ensure that the input is compared to the correct output value. The error signal is then used to update the values currently stored in the LUT.

The LUT address is derived from the input power. Hence this algorithm is only able to correct for phase and magnitude errors that are purely a function of the current input power.

**Functional Description**

**Figure 5** shows the block diagram of the DPD reference design, a PA, an upconverter, and a downconverter.
The reference design is an adaptive LUT-based DPD solution.
Input data for the PA is fed into the predistorter, which reads an appropriate correction value (LUT value) from the LUT and uses it to modify the input data. The resultant modified input data is referred to as predistorted data, which is converted into RF to be fed into the PA.

The feedback processing block performs the adaptive processing to update the values in the LUT. The updated values in the LUT reflect changes in the PA behavior over time from ageing and temperature, and at startup to determine the PA behavior. To do this, the PA output (in digital form), PA input, and the address of the LUT value that is used to modify (i.e., predistort) the PA input are fed into this block.

**Predistorter**

The input data (in cartesian form) is fed into the index calculation block, which determines the index (address) of the LUT value. This LUT value modifies the delayed input data. For the reference design, only power indexing is used i.e., the power of the input data determines the LUT index. You can implement other indexing schemes such as magnitude or a FIR filter using a number of previous input magnitudes.

The input data is delayed by the delay blocks, prior to undergoing a complex multiplication with the cartesian LUT value read from the LUT, to compensate for the delay through the index calculation block and the reading out of data from the LUT.

The input data and LUT indices (LUT read addresses) are delayed before being output from the predistorter to be fed into the feedback processing block. This delay compensates for the delay of the predistorted signal traveling to the PA and then the PA output making its way back to the feedback processing path. In the reference design, as no real PA is being used and everything remains in the digital domain, a fixed delay (integral number of clock cycles) is used. For a hardware implementation using a real PA, a more elaborate delay matching scheme will be required to synchronize the feedback PA output with the PA input.

**Feedback Processing**

This section is an overview. For a more detailed description on the design’s software, see “Software Description” on page 34.

The feedback processing block comprises the following components:

- CORDIC block
- Nios processor
- General purpose RAM
The feedback processing block uses a Nios processor and hardware acceleration. The Nios processor offers you the flexibility to change the algorithm or to implement your own algorithm, without having to make significant changes to the RTL. The hardware acceleration ensures that the feedback path exhibits reasonable performance.

The adaptive algorithm works with the PA input and output values in polar form. However, they are available in cartesian form; the conversion of cartesian to polar co-ordinates can be a time-consuming process if it is implemented on a processor. Furthermore, it is not possible to pipeline these tasks on simple processors such as the Nios processor. Thus the conversion is hardware accelerated using a CORDIC.

**CORDIC Block**

The reference design implements an Altera CORDIC reference design, for the cartesian-to-polar and polar-to-cartesian conversions. CORDIC is a hardware-efficient algorithm that allows trigonometric functions to be performed using only shifts and adds. Because of its simplicity, it offers high performance. CORDIC is an iterative algorithm that performs various conversions and uses only additions, subtractions and shift operations. The CORDIC is implemented using logic elements (LEs) that can operate in arithmetic mode, where each LE is configured to contain a full adder/subtractor cell plus associated register.

For more information on the CORDIC reference design, refer to Application Note 263 CORDIC Reference Design.

**Nios Processor**

The Nios processor uses the Avalon™ bus to communicate to its peripherals.

The Nios processor communicates with the CORDIC block via the Avalon bus. The Nios processor reads PA input and output values in polar form (and also the LUT index) from the CORDIC Avalon slave interface. Using these values, the Nios processor implements the algorithm and calculates the new LUT value in polar form. The Nios processor writes this value back to the CORDIC block, and the LUT address that it refers to (this is the same value as the LUT index that it previously read). CORDIC converts the polar form into cartesian. The CORDIC block then writes this value to the main LUT at the LUT address provided by the Nios processor.
Scheduling of Operations—The CORDIC Frame

The DPD reference design performs the following repeating cycle of operations:

1. Cartesian PA input and output values are fed into the CORDIC block.

2. The polar PA input and output values are read from the CORDIC block by the Nios processor.

3. The Nios processor implements the algorithm to calculate the new LUT value in polar form.

4. The Nios processor writes this LUT value to the CORDIC block.

5. The CORDIC block converts the LUT value into cartesian form and then writes it to the LUT.

6. Repeat from step 1.

No deviation from the cycle occurs and the cycle takes a finite number of clock cycles. The cycle is called a CORDIC frame.

Figure 6 on page 8 shows the CORDIC frame in relation to the operations performed.

<table>
<thead>
<tr>
<th>Figure 6. CORDIC Frame Note (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORDIC Frame</td>
</tr>
<tr>
<td>CORDIC Frame</td>
</tr>
<tr>
<td>CORDIC Block Operations</td>
</tr>
<tr>
<td>Nios Processor Operations</td>
</tr>
</tbody>
</table>

Note:
(1) The numbers in the figure refer to the preceding text.

Synchronizing the Nios Processor to the CORDIC Block

For the cycle to work, an input signal is fed into the CORDIC block to signify the start of a CORDIC frame. Also, the Nios processor must then synchronize itself to the CORDIC frame as seen by the CORDIC block.

A small window at the start of the frame is predetermined to be a valid phase. During this phase, if the Nios processor attempts to read from the CORDIC block, a valid value is returned. During the rest of the time
within the CORDIC frame, an invalid value is returned to the Nios processor. So after startup and initialization has completed, the Nios processor repeatedly reads from the CORDIC block until a valid value is returned. At this point the Nios processor (within a margin of the valid window) knows where it is in the CORDIC frame and thus can commence with the DPD algorithm.

It has been assumed that during this synchronizing phase the Nios processor attempts to read the register in the CORDIC block holding the PA input magnitude value; an invalid value would be a value of all ones (this magnitude could not normally occur as magnitudes must always be positive).

**Nios Processor Operations**

During a CORDIC frame, the Nios processor performs the following tasks (also see Figure 7):

1. Sync-up with the CORDIC block by polling the PA input magnitude. The CORDIC block ensures that the INVALID_FLAG is always read for all but the first WINDOW_VALID cycles of a CORDIC frame. When the Nios processor reads a positive value it knows it has read a valid value and moves on to the next stage.

2. Read the following remaining polar form data from the output of the CORDIC block to be used this cycle:
   a. PA input phase
   b. PA output magnitude
   c. PA output phase
   d. LUT index

3. Compute the error in magnitude and phase between PA-out and PA-in and pass through the DPD algorithm to compute the new LUT value(s) around the given index.

4. Update the Nios private polar form LUT with the new LUT value(s).

5. Compensate for the gain of the CORDIC by multiplying the magnitude component of each new LUT entry by CORDIC_COMP_FP.
6. Write the results back to the CORDIC block for conversion to cartesian format (the CORDIC block automatically writes the cartesian-format LUT value(s) to the primary LUT).

**Figure 7. The CORDIC Frame**  
Note (1)

<table>
<thead>
<tr>
<th>PA Input Magnitude</th>
<th>Valid</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORDIC Frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nios Processor Operations</td>
<td>1, 2a, 2c, 2d, 3</td>
<td>4, 5, 6</td>
</tr>
</tbody>
</table>

**Note:**
(1) The numbers in the figure refer to the preceding text.

**Memory Effects**

The memory effect models the distortions due to short-term temperature and electrical variations on the silicon of the PA transistors. The temperature depends on the magnitude of current and previous input samples and hence a FIR filter can be used in the address calculation block to compensate for the memory effect, by providing a weighted sum of previous inputs as index to the look-up table. This solution has not been verified using an actual PA and is suggested if you want to implement your own DPD system.

**MATLAB GUI**

Altera provides a bit-accurate and cycle-accurate (cycle-accurate only at outputs) MATLAB model of the DPD reference design. In addition, Altera provides a MATLAB environment, which, via a user-friendly GUI, allows you to perform the following actions:

- Configure the reference design to your exact requirements—set bit widths, depth of LUT, set learn factor
- Configure the test scenario—which PA model to use, what type and how many input stimuli to apply, whether to run the simulation repeatedly until the design has reached convergence
- Examine graphs of results, write test input and output data, and LUT values to text files for use in RTL simulation. Also write to files the Verilog HDL and C constants that describe your configurations, for use in RTL simulation and synthesis.

You can experiment further in the GUI (if you turn off the reference design configuration), by configuring the following options:

- Change bit widths
- Add interpolation
- Use magnitude indexing (as opposed to power indexing) of the LUT
Change the number of CORDIC iterations (see AN263 CORDIC Reference Design)

• Choose a different DPD algorithm

The MATLAB model is no longer representative of the RTL design, if you turn off the reference design configuration.

These additional options are provided purely for experimentation purposes in the MATLAB environment; if you wish to implement one of these variations in hardware, you will have to modify the hardware source files accordingly.

**Design Flow**

The design flow involves the following steps:

1. Configure the reference design settings in MATLAB until you are satisfied with the results (displayed by the graphs).
2. Write the Verilog HDL, C constants and test data to files.
3. Build the design using SOPC Builder.
4. Use the test data in RTL simulation to confirm that the hardware provides the same results as the MATLAB model.
5. Synthesize the design.

**Functional Description**

Stimuli (random, constant, linearly changing) is generated in MATLAB in cartesian format \((I + j \times Q)\). The stimuli is interpolated and applied to the predistorter. The resulting predistorted data is also in cartesian format.

In a real situation, the predistorted data is converted to analogue, filtered, upconverted to RF and then applied to the PA.

To keep the situation simple, such that only the performance of the DPD design is being examined, in MATLAB no modelling of these processes is performed. The predistorted digital data is applied directly to a MATLAB PA model that accepts digital input. Also the output from the PA model is applied directly to the reference design (no downconversion or digitizing required).

The predistorter attempts to add 3rd and 5th order intermodulation products to the input signals in a way that cancels out the 3rd and 5th order intermodulation products added by the PA. Thus the bandwidth of the
predistorted signal must be three times greater than the bandwidth of the input signals to be able to represent up to the 5th order intermodulation products. Bearing in mind, that in the real world the predistorted signals are fed into a DAC and then low-pass filtered at the Nyquist rate (half the input sample rate), the predistorted signals must have a sample rate of at least six times that of the original input signals.

Thus, in MATLAB the input signals are interpolated by a factor of six before being fed into the predistorter. The interpolation used is ideal to ensure that only the performance of the DPD design is being examined. For every input sample, five zero-valued samples are added and then the resultant data stream is low-pass filtered using an ideal rectangular filter.

To help evaluate the performance of the DPD reference design, a figure for in-band distortion is calculated. This involves calculating the EVM, which is given by the following equation:

\[
EVM = \frac{\text{rms}(\text{error}_\text{vector})}{\text{rms}(\text{input magnitude})} \times 100
\]

Where:

\[
\text{error}_\text{vector} = |\text{output} - \text{input}|
\]

In MATLAB, the output from the PA is at a rate of six times that of the original input (as the input was interpolated by factor of six before being applied to predistorter). Thus, the output is decimated by a factor of six using an ideal rectangular low-pass filter, before calculating the EVM.

The following two measurements provide the performance of the DPD reference design:

- The EVM is calculated, which indicates in-band distortion. This EVM is compared to the EVM when no DPD is being used (the EVM of the PA) to give a comparative figure of improvement.
- The ACLR is calculated, which indicates out-of-band distortion. This ACLR is the power (in dB) that the adjacent channel is down from the main in-band channel. The ACLR is also calculated for a system with no DPD, to give a comparative figure of improvement.

Also, a visual indication of the results is given by various graphs.

**RTL Testbench**

The RTL testbench sends PA-in and PA-out data stored in MATLAB-generated text files and compares the predistorted result with expected values stored in a text file.
The testbench implements a method of synchronizing up with the Nios processor, so that PA data is not written until the Nios processor has run all its initialization code. When the Nios processor is ready to begin processing CORDIC data, it first writes to the testb_if Avalon slave.

The top-level testbench, located in dpd_nios.v, performs the following tasks:

- Generates the system clock
- Waits for the Nios processor to write to testb_if, which signifies that the Nios processor has completed its initialization and is ready to begin polling the CORDIC block for valid data
- Sends PA in and PA out data
- Checks predistorted data values
- Checks LUT indices are calculated correctly for each PA in data
- Checks each write to the LUT

The system uses a single clock for all components.

**Before You Begin**

This section involves the following steps:

- “Software Requirements”
- “Install the Design”

**Software Requirements**

The reference design requires the following software:

- Quartus® II software version 3.0, or higher
- Nios embedded processor, version 3.02
- MATLAB version 6.5
- ModelSim version 5.7a

**Install the Design**

To install the reference design, run the .exe and follow the installation instructions. Figure 4 shows the directory structure.
The design walkthrough involves the following steps:

- “Parameterize the Design”
- “Auto-generate Verilog HDL Simulation Files from MATLAB”
- “Build the SOPC Builder Project”
- “Simulate the Design”
- “Synthesize the Design”

Parameterize the Design

To parameterize the design or examine a DPD design’s performance, perform the following steps:
1. Start the MATLAB GUI.
   a. Start MATLAB.
   b. Change the current directory to `predistortion/test/matlab/tb`.
   c. Type the following command:
      
      ```
      dpd_gui
      ```
      
      Output messages are still output to the MATLAB command window.

      The Digital Predistortion GUI opens (see Figure 3).

      The white bar at the bottom of the GUI displays recommendations, if you enter unsuitable values.

2. In the Algorithm Configuration part of the digital predistortion GUI, perform the following steps:
a. If you are using the reference design, turn on the Reference Design Configuration, otherwise turn it off.

- Upon starting MATLAB and whenever the Reference Design Configuration is turned on, the Algorithm Configuration section has all the appropriate settings for the default reference design configuration.

- If turned off, then you can only work in the MATLAB domain; the RTL will not match the MATLAB model. Thus, only turn off this option for experimentation purposes of other schemes or variants.

- Many options are greyed out if you turn on the Reference Design Configuration, the rest of this walkthrough assumes that you turn it off to allow the explanation of all the options.

b. In the Algorithm drop-down box, choose the algorithm type. For all algorithms you can enter the LUT size 1 and learn factor. For certain algorithm types you can enter the LUT size 2, and the FIR coefficient bit widths and coefficients values.

   For a description of the algorithms, contact Altera.

c. Enter the bit widths for the following signals:
   - PA IQ signals. The number of bits in PA input and output signals
   - LUT IQ signals. The number of bits in LUT values
   - Phase signals: The number of bits for phase
   - CORDIC XY precision extend: The number of bits to extend XY vectors used in the CORDIC block, which improves the accuracy of the results of polar or cartesian value calculation
   - CORDIC phase precision extend. The number of bits to extend phase signals used in the CORDIC block, which improves the accuracy of the results of cartesian or polar values calculated
   - Nios fixed point fractional bits. The Nios processor uses 32-bit fixed-point format. This field determines the number of fractional bits.

d. In the Feedback Performance section enter the following values to determine the modelling performance of the feedback path:
Input signals are applied every clock cycle, yet it takes a finite number of clock cycles for the feedback path to process one set of values to calculate a LUT value, before being able to process the next set of values.

- Enter the delay for Nios to calculate the next LUT value in terms of how many input samples surpass.
- Also there is a delay between the Nios calculating the new LUT value and the LUT being updated with that value (delay due to CORDIC conversion of polar to cartesian). Enter the delay in updating the LUT, in terms of number of input samples.

For each calculated LUT value, the LUT values nearby could also be altered, thereby improving convergence time. To enable this feature, turn on **Interpolation Enable** and enter the number of LUT values to interpolate above and below the new calculated LUT value.

Enter the number of **Cordic Iterations**. The accuracy and latency of the cartesian-to-polar and vice-versa is determined by the number of CORDIC iterations. The more iterations (up to the bit width of the inputs) improves the accuracy of the results.

3. In the Test Environment Configuration part of the digital predistortion GUI, perform the following steps:

   a. In the **PA Model** drop-down box, choose the PA model. For the reference design Altera recommends using **Saleh model**.

   b. To use the previous run’s final LUT, turn on **Use previous run’s LUT**.

   c. To load the LUT from text files, turn on **Load LUT from Text File**. This option is only available for certain DPD algorithms. The LUT files must be in the `build/mat_gen/dat_files` directory. The following files are required:

      - **init_lut_iq_table.txt**. The initial cartesian values of the LUT
      - **init_lut_magn_fp.txt**. The initial magnitude scaled by the CORDIC gain and represented in Nios fixed-point format of the LUT
      - **init_lut_phase_fp.txt**. The initial phase, represented in Nios fixed-point format of the LUT
d. In the **Input Stimuli** drop-down box, choose the input stimuli. For constant input stimuli, you can enter the magnitude and phase.

e. Enter the number of input samples. Remember that these samples are interpolated by a factor of six, thus generating six times the number of samples to be fed into the predistorter.

f. For fixed point input stimuli, turn on **Fixed Point**, otherwise turn it off. If turned off, signal values will be between –1 and +1.

g. To repeatedly run the simulation until convergence has been achieved turn on **Run until converges**, otherwise turn it off. Turning it on can result in long simulation times.

h. If you turned on **Run until converges**, you have to define convergence. Enter the **number of times the EVM is greater than previous EVM** defines attaining convergence. Thus for each simulation run (assuming the number of input batches is 1), the EVM is calculated and compared to the previous run’s EVM. Simulation runs are repeated until the current EVM has exceeded the previous ones by the number entered in the box. At this point the DPD reference design has achieved convergence and so simulation stops.

i. Enter the number of input batches to run, with each simulation run comprising the number of input stimuli specified in **Number of Input Samples**. If **Run until converges** is turned on, this number of input batches are run before comparing the EVM with the previous (average EVM over previous batch run) EVM.

4. Click **Run**, to run the simulation.

The MATLAB command window prints out the EVM and ACLR figures for the run.

5. Select the type of graph that you want MATLAB to plot. The options include examining final LUT values, magnitude and phase errors, cartesian errors and frequency plots.

6. When you have viewed the graph, click **Close Plots**.
Auto-generate Reference Design Configuration and Test Data For Verilog HDL Simulation from MATLAB

Once MATLAB testing reaches a conclusion with the configuration of a suitable reference design, to auto-generate test data for Verilog simulation from MATLAB, perform the following steps:

You can only auto-generate Verilog HDL simulation files, if you turn on Reference Design Configuration.

1. Enter the number of data values that you want to write in Number of Data Values to Write.

2. Enter a value in Start offset to write LUT table and data from. If you want to use the files for RTL simulation you must enter 0.

3. Click Write Data & LUT to Files to write the data and LUT to predistortion/test/dat_files

4. To write data values midway through the simulation, perform the following steps:
   a. Run the simulation once.
   b. Turn on Use Final LUT from Previous Run.
   c. Click Run, to run the simulation for a second time.
   d. Click Write Data & LUT to Files to write the data and LUT to predistortion/test/dat_files.

5. Click Write Verilog & C Constants, to write the required Verilog HDL parameters and C constants to the following files:
   - predistort/source/verilog/dpd_inc_p2.v
   - predistort/ip/cordic/source/verilog/cordic_inc_p2.v
   - predistort/source/verilog/cordic_convert_inc.v
   - predistort/source/software/predistort.h.

MATLAB Results for Default Configuration

For the default parameters of the DPD Reference Design, turn on Reference Design Configuration and see the settings in the MATLAB GUI.
The Saleh power amplifier model was used and random input stimuli was selected with 500 input samples (MATLAB interpolates to give 3000 samples).

**Run until convergence** was turned on and convergence was defined as:

- EVM > prev EVM = 10.
- The Number of input batches = 10.

The final results were EVM reduction over EVM when no DPD implemented = 95.838%

**No Predistortion**

3rd order > Average sideband magnitude = –55.943 dB, maximum sideband magnitude = –30.945 dB

5th order > Average sideband magnitude = –83.768 dB, maximum sideband magnitude = –43.624 dB

**Predistortion**

3rd order > Average sideband magnitude = –110.121 dB, maximum sideband magnitude = –86.145 dB

5th order > Average sideband magnitude = –114.752 dB, maximum sideband magnitude = –90.715 dB

*Figure 10 through Figure 13* show the various MATLAB plots for the default configuration.
**Figure 10. Frequency Plot**

![Frequency Plot Image](image-url)
Figure 11. Magnitude and Phase Errors
Figure 12. Cartesian Errors

Comparison of error with predistorted and non-predistorted PA system. Normalized plots of fractional and imaginary error parts over time (samples).
Build the SOPC Builder Project

At this point you have used the MATLAB GUI, and should be confident of the performance of the reference design. You should also have written the Verilog HDL and C constants that configure the reference design and the test data to apply in the RTL simulation. You can now build the RTL for the reference design. When you have finished this section, the RTL, software, and simulation files for your reference design will be in the \predistortion\build\dpd_nios directory.

To build the SOPC Builder project involves the following steps:

- Run the Perl Script
- Open the Quartus II Project
Launch SOPC Builder
Generate the System

Run the Perl Script

The perl script performs the following actions:

- Creates the `predistortion\build\dpd_nios` directory
- Creates a directory structure for your design in the `predistortion\build\dpd_nios` directory
- Copies the required files into the `predistortion\build\dpd_nios` directory

Once you have run the perl script you can use SOPC Builder to generate the project files and perform simulation and synthesis. To run the perl script perform the following steps:

1. Open the command prompt (Windows Start menu).
2. Use the `cd` command to change the `predistortion\build` directory.
3. Type the following command:

   ```bash
   perl gen_dpd_frame.pl -rel
   ```

   As it runs, the perl script informs you of the files that are being copied.

Open the Quartus II Project

To open the `dpd_nios_top.quartus` project, perform the following steps:

1. Open the Quartus II software.
2. Choose Open Project (File menu).

The top-level block diagram file (BDF) for this project, `dpd_nios_top.bdf`, appears (see Figure 14).

The BDF gives a schematic view of the top level input/outputs of the reference design.
Launch SOPC Builder

To launch SOPC Builder, double click on the **dpd_nios** symbol within the BDF. The SOPC builder system contents page for the **dpd_nios** appears (see Figure 15).

For more information on customizing the Nios processor, refer to the **SOPC Builder User Guide** and the documentation that is included with the NDK.
**Figure 15. SOPC Builder System Contents Page**

Generate the System

To generate the `dpd_nios` system, perform the following steps:

1. Click the **System Generation** tab in SOPC Builder.
2. Click **Generate**.

This action generates the NDK, the Nios CPU and the custom instruction macros in `excalibur.h`. It also compiles and links the source files and places the result in the `general_purpose_RAM.srec` file. When the generation is complete a message is displayed (see Figure 16).
Simulate the Design

The Run Modelsim button in the System Generation tab is enabled after your system has finished generating. Click Run Modelsim to run the Modelsim software automatically and load the project file generated for your system.

You must specify the correct path to the ModelSim software by choosing SOPC Builder Setup (File menu).

For more information on generating your design, refer to the SOPC Builder Data Sheet.
When the system generation completes, the simulation directory contains all of the files necessary for simulation. Table 1 describes the files that are created.

<table>
<thead>
<tr>
<th>File Extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.mpf</td>
<td>ModelSim Project file. This file is created if SOPC Builder finds the ModelSim path. If you can open this file in ModelSim, the directories and paths are set for simulation and the simulation macros are initialized (see Table 2).</td>
</tr>
<tr>
<td>.do</td>
<td>ModelSim macro execution scripts.</td>
</tr>
<tr>
<td></td>
<td>The setup_sim.do file initializes the simulation macros (see Table 2).</td>
</tr>
<tr>
<td></td>
<td>The wave_presets.do file holds a list of default signals that are displayed in the waveform window.</td>
</tr>
<tr>
<td></td>
<td>The virtuals.do file sets up a virtual signal that translates Nios operation codes (opcodes) into instructions, allowing you to view assembly instructions during simulation.</td>
</tr>
</tbody>
</table>

You may wish to replace the generated wave_presets.do file with a more detailed version that comes with the design. The file wave_presets.do_keep has been preset to give a view of the important CORDIC block signals, and the Nios processor.

In `build\dpd_nios\dpd-nios.v`, you must set the IP_NO parameter for the number of input data to test.

**Getting Started in ModelSim**

This section describes some basic steps to get you started running your simulation with the ModelSim simulator.
To begin simulating using the SOPC Builder generated ModelSim project file (.mpf) located in the simulation directory, perform the following steps:

1. Click Run ModelSim, to run the ModelSim software and automatically invoke the setup_sim.do.

2. Execute the s macro to load and compile the design.

3. Execute the w macro to display the ModelSim waveform window.

The signals shown are only a starting-point for your analysis of the system behavior. You can add or remove signals by using the options in the view menu in the main ModelSim window.

**Analyzing the Simulation Results**

After you have loaded and compiled the design, you can simulate and analyze the operation of the DPD system with the Nios processor. To start a simulation use commands in the run menu in the main ModelSim window, or execute the run command at the ModelSim prompt. For example type the following command to run the simulation through 100 μs:

```
run 100 us
```

**Figure 17** shows a ModelSim wave window that shows a single Nios CORDIC frame. The waveforms between the two yellow markers represent a single CORDIC frame, and in this instance, shows the Nios data master issuing Avalon read and writes to memory and peripherals. The first two reads after the first marker represent the synchronization to the CORDIC block.

The Nios data master handles reading and writing data to and from memory and memory-mapped peripherals in the Nios subsystem. The read, write, address, and byte-enable signals operate according to the Avalon bus specification.
The testbench is self-checking. It generates output to the screen and the build\dpd_nios\dpd_nios_sim\log_op_err.txt log file, to indicate the outcome of the simulation.

**Synthesize the Design**

Before you synthesize the reference design, you must first generate the project (see “Generate the System” on page 27). To synthesize the design, perform the following steps:

1. Open the Quartus II software.
2. Choose Open Project (File menu).
3. Browse to the installed predistort\build\dpd_nios/ directory. Choose the dpd_nios_top.quartus project file.
5. Under category expand Timing Settings and choose Clocks, to set the synthesis constrained clock frequency. You can either set the same frequency for all clocks in the system or constrain individual clocks. The following three clock domains are in the system and all are assumed to be synchronous:

   - clk_forward—predistorter clock domain
   - clk_nios—Nios clock domain
   - clk_cordic—CORDIC clock domain

For the reference design assume that the input is four UMTS channels, i.e, all clock frequencies are the same.
6. Under category expand Compiler Settings and choose Device. Choose a Stratix device or select Auto device selected by the Fitter.

7. Choose Start Compilation (Processing menu).

Performance

Table 3 shows the performance for the default configuration of the reference design, using Quartus II version 3.0 with push-button synthesis for Stratix devices. The clock frequency was constrained to 120 MHz for all clock domains.

<table>
<thead>
<tr>
<th>Forward Path Clock $f_{\text{MAX}}$ (MHz)</th>
<th>Nios Clock $f_{\text{MAX}}$ (MHz)</th>
<th>CORDIC Clock $f_{\text{MAX}}$ (MHz)</th>
<th>Logic Elements (LEs)</th>
<th>Memory (Bits)</th>
<th>9-bit Multipliers</th>
</tr>
</thead>
<tbody>
<tr>
<td>268</td>
<td>126</td>
<td>155</td>
<td>4,133</td>
<td>143,360</td>
<td>22</td>
</tr>
</tbody>
</table>

A single UMTS channel has a bandwidth of 5 MHz. Thus four channels added together have a total bandwidth of $5 \times 4 = 20$ MHz. To be able to represent $3^{rd}$ and $5^{th}$ order intermodulation products the input signal bandwidth must be at least $3 \times 20 = 60$ MHz. Due to the Nyquist sampling rate theorem, this translates to a sample rate (clock speed) in the forward path of at least $2 \times 60$ MHz = 120 MHz.

Table 3 shows that the forward path has an $f_{\text{MAX}}$ of 237 MHz and thus can easily run at 120 MHz.

As the Nios clock $f_{\text{MAX}}$ is 126 MHz it cannot operate at a multiple of the forward path and so must also operate at 120 MHz (the CORDIC block also operates at 120 MHz).

Supporting less than 4 channels, means that the forward path clock rate decreases, and so it may be possible to operate the feedback path (Nios and CORDIC) at a multiple of this rate.

Use LEs in Cyclone devices, where an implementation in Stratix devices uses DSP blocks.

Change Bit Widths

You can change the PA input and output and/or the LUT bit widths from the reference design settings, however you must manually edit two SOPC Builder files. To change the bit widths, perform the following steps:

1. In the MATLAB GUI turn off Reference Design Configuration (see “Parameterize the Design” on page 14).
2. Select the appropriate bit widths for the PA IQ signals and the LUT IQ signals.

3. Simulate in MATLAB and modify (if necessary) until you are satisfied with the settings.

4. Write out the design configuration parameters and test data (see “Auto-generate Reference Design Configuration and Test Data For Verilog HDL Simulation from MATLAB” on page 19).

5. Build the design in the build/dpd_nios directory (see “Build the SOPC Builder Project” on page 24) but do not generate the system. Before you generate the system, you must edit the following two SOPC Builder files with a suitable text editor:

   ● build/dpd_nios/dpd_nios.ptf
   ● build/dpd_nios/predistort_cordic/class.ptf

Both these files contain similar sections defining the vector widths of the PA input, PA output, and predistorted signals. The following code is a relevant portion from dpd_nios.ptf:

```plaintext
PORT pa_in_i
{
  width = "14";
  direction = "input";
  type = "export";
}
PORT pa_in_q
{
  width = "14";
  direction = "input";
  type = "export";
}
PORT pred_pa_in_i
{
  width = "14";
  direction = "output";
  type = "export";
}
PORT pred_pa_in_q
{
  width = "14";
  direction = "output";
  type = "export";
}
PORT pa_out_i
```
Software Description

This section describes the software part of the DPD reference design.

The feedback processing block contains a 32-bit Nios processor with advanced features enabled (i.e. custom instructions) (see Figure 18).

Figure 18. Nios Processor Architecture

6. Change the widths to the correct values.

7. Generate the project in SOPC Builder (see “Generate the System” on page 27).

8. Simulate and synthesize (see “Simulate the Design” on page 28 and “Synthesize the Design” on page 31).
Table 2 shows the DPD reference design Nios processor parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction cache</td>
<td>Off</td>
</tr>
<tr>
<td>Data Cache</td>
<td>Off</td>
</tr>
<tr>
<td>Register file size</td>
<td>128</td>
</tr>
<tr>
<td>Multiplier</td>
<td>MUL</td>
</tr>
<tr>
<td>Pipeline Optimization</td>
<td>Fewer stalls</td>
</tr>
<tr>
<td>Instruction decoder</td>
<td>ROM</td>
</tr>
<tr>
<td>RLC/RRC</td>
<td>Disabled</td>
</tr>
<tr>
<td>Interrupts and traps</td>
<td>Disabled</td>
</tr>
<tr>
<td>Catch spurious interrupts</td>
<td>Disabled</td>
</tr>
<tr>
<td>Call C++ constructors</td>
<td>Disabled</td>
</tr>
<tr>
<td>Use the CWP manager</td>
<td>Disabled</td>
</tr>
<tr>
<td>Use fast multiply</td>
<td>On</td>
</tr>
<tr>
<td>Use small printf</td>
<td>On</td>
</tr>
</tbody>
</table>

For more information on editing these parameters, refer to the documentation that is supplied with the Nios processor.

Figure 19 and Figure 20 on page 36 show the Nios processor settings.
Figure 19. Nios Processor Hardware

Figure 20. Nios Processor Software
This design does not require the CWP manager as the maximum subroutine depth is only 2 (see “Software Description” on page 34), whereas the theoretical maximum is 6 given a register window depth of 128. This also reduces the code footprint of the compiled software.

The system does not use the Nios debug features (see Figure 21).

**Figure 21. Nios Processor Debug**

![Nios Processor Debug](image)

Figure 22 shows the custom instructions used by this design (see “Software Description” on page 34).
The boot ROM represents 4 K of ROM and is only relevant to synthesized designs where it will be typically used to hold the Nios GERMS monitor program.

The general purpose RAM represents 16 K (programmable) of RAM and in this instance is used to hold the executables .code and .bss segments (see “Generate the System” on page 27).

The design includes a general purpose 32-bit counter useful for performance measuring on synthesized designs. The counter is not used during simulation.

The design includes a general purpose UART running at 155 K baud. The UART is not used during simulation.

The Nios subsystem comprises the following source and header files:

- **Source files:**
  - nios_startup.s
  - dpd.c

- **Header files**
  - excalibur.h
  - predistort.h
**nios_startup.s**

The *nios_startup.s* file is a cut-down version of the file provided in the Nios development kits (NDKs), and performs the usual Nios startup and initialization tasks, followed by a call to `main()`.

For reduced simulation start-up time, the *nios_startup.s* file does not clear the `.bss` segment, unlike the NDK version. The clearing of the `.bss` segment is unnecessary—the only variables that get stored there are initialized within the main code segment.

**dpd.c**

The primary source file *dpd.c* comprises two functions:

- `main()`
- `digital_pre_distort()`

The `main()` procedure is the entry point from *nios_startup.s* and calls up the `digital_pre_distort()` function.

The `digital_pre_distort` function is written as a continuous loop, where each pass through the loop represents one single CORDIC frame (see “Nios Processor Operations” on page 9).

**excalibur.h**

The *excalibur.h* file is generated by SOPC builder. This file contains `#define` statements that describe the memory map and custom instruction macros. This file also contains C language structures for accessing peripheral registers if required.

Because the Nios processor architecture and memory map are configurable, you must only reference the hardware using the symbols contained within this file.

**predistort.h**

The *predistort.h* file is generated by MATLAB. This file includes `#define` statements that define the operation of the DPD (see Figure 23).
Figure 23. MATLAB Generated Predistort.h

```c
/* --- C DEFINES auto-generated by MATLAB */
*/
/* Total number of locations in LUT (decimal number) */
#define LUT_SIZE 64
/* Learn factor, expressed in nico fixed point format (hexadecinal) */
#define LEARN_FP 0x00000000
/* CORDIC gain, expressed in nico fixed point format (hexadecimal) */
#define CORDIC_GAIN_FP 0x80000000
/* CORDIC gain compensation (1/cordic_gain^2), expressed in nico fixed point format (hexadecinal) */
#define CORDIC_COMP_FP 0x1793B34
/* Interpolation of LUT (1-enabled 0-disabled) */
#define INTERPOL 0
/* Number of values above and below selected LUT value to interpolate */
#define INTERPOL_NO 3
/* Feedback path performance: Use 1 in every N inputs to calculate new LUT value */
#define Proc_L_IN_SAMPLES 1
/* Feedback path performance: Update main LUT with new value after a further N inputs have passed through system */
#define UAT_LUT_UPDATE 0
/* INVALID FLAG when Nios reads PA_in_msg from CORDIC block at incorrect times */
#define INVALID_FLAGS 0xFFFFF
/* --- */
```

Table 5 shows the definitions.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT_SIZE</td>
<td>Defines the number of elements in the LUT.</td>
</tr>
<tr>
<td>LEARN_31</td>
<td>Defines the learning factor used by the DPD algorithm in both floating point and fixed-point. The learning factor determines the system convergence time. Increasing the learning factor results in a reduced overall convergence time, but at the expense of an increased final error. Conversely, a smaller learning factor results in a slower convergence, but with reduced final error.</td>
</tr>
<tr>
<td>CORDIC_GAIN_FP</td>
<td>Defines the numeric gain of the CORDIC.</td>
</tr>
<tr>
<td>CORDIC_COMP_FP</td>
<td>Defines the reciprocal of the square of CORDIC_GAIN_FP in fixed-point. PA I and Q data always pass through the CORDIC twice. The first time, I and Q data are converted to polar form to be read by the Nios processor. The Nios processor then processes these and writes them back to the CORDIC to be converted back to cartesian form prior to being written to the main LUT. For this reason, the Nios processor must compensate for this gain by multiplying the magnitude component with CORDIC_COMP_FP prior to writing back to the CORDIC.</td>
</tr>
<tr>
<td>INTERPOL_NO</td>
<td>Determines the functionality of the DPD algorithm, set to 1 or 0. The number of interpolations to perform around the current LUT index. Total number of interpolations = (INTERPOL_NO × 2) + 1</td>
</tr>
</tbody>
</table>

Using interpolation significantly increases the running time of the DPD algorithm and therefore the CORDIC frame. For the reference design, interpolation is disabled.
The DPD reference design uses fixed-point format throughout to maximize speed and efficiency. However, the exact format and the number of bits required change as the signal progresses through the system.

The fixed-point format used by the Nios processor is 6.26, (six bits integer and 26 bits fractional), which yields a numerical range of –32.0000 to +31.99999999 and a quantization of 1.4901e–8. This format was chosen for its near floating point precision, while maintaining a usable numeric range to avoid overflow during add and subtract operations.

Numeric overflow during the fixed-point multiply operator is avoided by performing the task in hardware via a Nios custom instruction (see “Hardware Acceleration on Nios Processor” on page 44).

The fixed-point formats at the interface to CORDIC block, as seen by the Nios processor, are specified in the Verilog HDL include file `cordic_convert_inc.v` (see Figure 24). The conversion between the various formats is performed in hardware using the Nios custom instructions.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROC_1_IN_SAMPLES</td>
<td>Used by hardware. Defines the length of the CORDIC frame in the CORDIC clock cycles</td>
</tr>
<tr>
<td>WAIT_LUT_UPDATE</td>
<td>Delay in number of input samples before a LUT value is calculated by the Nios processor. This is because of the delay through the CORDIC block.</td>
</tr>
<tr>
<td>INVALID_FLAG</td>
<td>Used by the Nios processor to sync-up with CORDIC block (see “Nios Processor Operations” on page 9).</td>
</tr>
</tbody>
</table>

**Fixed-Point Format**

The DPD reference design uses fixed-point format throughout to maximize speed and efficiency. However, the exact format and the number of bits required change as the signal progresses through the system.

The fixed-point format used by the Nios processor is 6.26, (six bits integer and 26 bits fractional), which yields a numerical range of –32.0000 to +31.99999999 and a quantization of 1.4901e–8. This format was chosen for its near floating point precision, while maintaining a usable numeric range to avoid overflow during add and subtract operations.

Numeric overflow during the fixed-point multiply operator is avoided by performing the task in hardware via a Nios custom instruction (see “Hardware Acceleration on Nios Processor” on page 44).

The fixed-point formats at the interface to CORDIC block, as seen by the Nios processor, are specified in the Verilog HDL include file `cordic_convert_inc.v` (see Figure 24). The conversion between the various formats is performed in hardware using the Nios custom instructions.
Figure 24. cordic_convert_inc.v

```
-- Verilog Parameters generated by MATLAB
--
parameter NIOS_FRACT_BITS = 26;
-- CORDIC to fixed-point parameters (Polar values to Nios fixed point)
--
parameter CFP_CORDIC_MAGN_INT_BITS = 3;
parameter CFP_CORDIC_MAGN_FRAC_BITS = 13;
parameter CFP_CORDIC_PHASE_INT_BITS = 1;
parameter CFP_CORDIC_PHASE_FRAC_BITS = 15;
-- Fixed-point to CORDIC parameters (LUT Nios fixed point to cordic polar values)
--
parameter FPC_CORDIC_MAGN_INT_BITS = 4;
parameter FPC_CORDIC_MAGN_FRAC_BITS = 14;
parameter FPC_CORDIC_PHASE_INT_BITS = 1;
parameter FPC_CORDIC_PHASE_FRAC_BITS = 15;
```

Table 6 shows the definitions.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFP_CORDIC_MAGN_INT_BITS</td>
<td>Input magnitude integer bits.</td>
</tr>
<tr>
<td>CFP_CORDIC_MAGN_FRAC_BITS</td>
<td>Input magnitude fractional bits.</td>
</tr>
<tr>
<td>CFP_CORDIC_PHASE_INT_BITS</td>
<td>Input phase integer bits.</td>
</tr>
<tr>
<td>CFP_CORDIC_PHASE_FRAC_BITS</td>
<td>Input phase fractional bits.</td>
</tr>
<tr>
<td>FPC_CORDIC_MAGN_INT_BITS</td>
<td>Output magnitude integer bits.</td>
</tr>
<tr>
<td>FPC_CORDIC_MAGN_FRAC_BITS</td>
<td>Output magnitude fractional bits.</td>
</tr>
<tr>
<td>FPC_CORDIC_PHASE_INT_BITS</td>
<td>Output phase integer bits.</td>
</tr>
<tr>
<td>FPC_CORDIC_PHASE_FRAC_BITS</td>
<td>Output phase fractional bits.</td>
</tr>
</tbody>
</table>

**Polar Form**

The Nios processor only operates on polar form (exponential form) data. For this reason, the Nios processor maintains a private LUT that is equivalent to the main cartesian form LUT stored in CORDIC block multiplied by the CORDIC_GAIN_FP.

**Memory Map**

Table 7 shows the system memory map for simulation.
To access the system memory map open the SOPC Builder System Contents page, see “Build the SOPC Builder Project” on page 24.

<table>
<thead>
<tr>
<th>Name</th>
<th>Base Address</th>
<th>End Address</th>
<th>Size (Bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gen_purpose_RAM</td>
<td>0</td>
<td>0x3fff</td>
<td>16K</td>
<td>Reset entry point. Holds the executable .code and .bss segments during simulation.</td>
</tr>
<tr>
<td>Testb_if</td>
<td>0x10000300</td>
<td>0x100003ff</td>
<td>256</td>
<td>Testbench interface. The Nios processor writes to the base when it has completed all its initialization tasks and is ready to receive data from the CORDIC block.</td>
</tr>
<tr>
<td>Predistort_cordic_0</td>
<td>0x10000200</td>
<td>0x1000021f</td>
<td>32</td>
<td>Interface to the CORDIC block.</td>
</tr>
</tbody>
</table>

**CORDIC Interface**

The register interface to the CORDIC block is defined by the data structures cordic_out (the interface seen by the Nios processor on a read to the CORDIC block), and cordic_in (the interface seen by the Nios processor on a write to the CORDIC block).

For more information on fixed-point formats, see “Fixed-Point Format” on page 41.

The following code shows the interface on a read from CORDIC block:

```c
typedef struct
{
    int pa_in_magn;  // Fixed-point PA-in magnitude
    int pa_in_phase; // Fixed-point PA-in phase
    int pa_out_magn; // Fixed point PA-out magnitude
    int pa_out_phase; // Fixed-point PA-out phase
    int index;       // LUT index derived from the PA-in magnitude
} cordic_out;
```

The following code shows the interface on a write to CORDIC block:

```c
typedef struct
{
    int magn;
    int phase;
} polar;

typedef struct
{
    int num_valid;  // Number of LUT entries written
```
Hardware Acceleration on Nios Processor

During the course of a CORDIC frame, the Nios processor:

- Performs a number of Avalon reads and writes to and from external memory
- Converts from the defined CORDIC numeric format to the 6.26 fixed-point format used by the DPD algorithm (and back again).
- Performs a significant amount of 32-bit maths within the algorithm

If all these actions were performed purely in software, the time taken to execute a single CORDIC frame would compromise overall performance. For this reason the key operations are performed in hardware via custom instructions.

For information on using custom instructions within the Nios embedded processor, refer to Application Note 188: Custom Instructions for the Nios Processor.

Figure 25 shows the custom-instruction tab listing the instructions used in this design.

Figure 25 shows part of `excalibur.h`—the custom instruction macros generated by SOPC builder.
Figure 25. The Nios Custom Instruction Tab

```
Altera Nios 3.0 - nios_0

Library
- Bit Swap
- Divide
- Endian Converter

Name                      Operation     Cycle Count   Opcode
fpml                      RA = RA sp...3  USR0
ctpnn                      RA = sp(RA)    1            USR1
ctppn                      RA = sp(RA)    1            USR2
ftpnn                      RA = sp(RA)    1            USR3
ftpnn                      RA = sp(RA)    1            USR4

See custom_instruction_ref. In, Application Note 128, and the Custom Instructions Tutorial.

Interrupts and traps have been disabled.
```

Figure 26. Part of Excalibur.h

```
#define nu_fpml(x, y) {\
  int __x = (x), __y = (y);
  asm volatile("usr0 %0,%2; does fpml\n
                           : "z"("x") , "r"("y")\n                              : "x", \\
                              : ");
}

#define nu_cfpn(x) {\
  int __x = (x);
  asm volatile("usr1 %0; does cfpn\n
                           : "r"("x")\n                              : "x", \\
                              : ");
}

#define nu_ctpp(x) {\
  int __x = (x);
  asm volatile("usr2 %0; does ctpp\n
                           : "r"("x")\n                              : "x", \\
                              : ");
}

#define nu_fpcm(x) {\
  int __x = (x);
  asm volatile("usr3 %0; does fpcm\n
                           : "r"("x")\n                              : "x", \\
                              : ");
}

#define nu_fpcp(x) {\
  int __x = (x);
  asm volatile("usr4 %0; does fpcp\n
                           : "r"("x")\n                              : "x", \\
                              : ");
```

Altera Corporation
Table 8 shows the DPD custom instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usr0 – nm_fpml(a, b)</td>
<td>Fixed point multiply operator, which computes ((a \times b) &gt;&gt; 26) in 3 cycles. Uses the Stratix DSP blocks to perform the 36 × 36 multiply to yield a 72-bit result.</td>
</tr>
<tr>
<td>Usr1 – nm_cfpma(a)</td>
<td>Converts CORDIC magnitude fixed-point format to Nios 6.26 format.</td>
</tr>
<tr>
<td>Usr2 – nm_cfpap(a)</td>
<td>Converts CORDIC phase fixed-point format to Nios 6.26 format.</td>
</tr>
<tr>
<td>Usr3 – nm_fpcm(a)</td>
<td>Converts Nios 6.26 fixed-point format to CORDIC magnitude format.</td>
</tr>
<tr>
<td>Usr4 – nm_fpcp(a)</td>
<td>Converts Nios 6.26 fixed-point format to CORDIC phase format.</td>
</tr>
</tbody>
</table>

Note:
(1) The CORDIC magnitude and phase fixed-point formats are defined as parameters in `dpd_inc.v`.

Summary

Congratulations! You have successfully used the DPD reference design. You can now see that the Stratix and Cyclone devices are suitable to implement DPD solutions. You can now start developing a complete solution for a real system.