DSP Builder

User Guide

101 Innovation Drive
San Jose, CA 95134
www.altera.com

Software Version: 7.2 SP1
Document Date: December 2007

UG-DSPBUILDER-7.2.1

Contents

Chapter 4. Using MegaCore Functions

Chapter 5. Using Hardware in the Loop (HIL)

Chapter 6. Performing SignalTap II Logic Analysis

Chapter 7. Using the Interfaces Library

Chapter 12. Adding a Board Library

Chapter 13. Troubleshooting

Additional Information

1. About DSP Builder

Features

DSP Builder supports the following features:

- Links The MathWorks MATLAB (Signal Processing ToolBox and Filter Design Toolbox) and Simulink software with the Altera® Quartus® II software
- Supports the following Altera device families:
  - Stratix®, Stratix GX, Stratix II, Stratix II GX, and Stratix III devices
  - Cyclone®, Cyclone II, and Cyclone III devices
  - Arria™ GX devices
  - APEX™ II, APEX 20KC, and APEX 20KE devices
  - ACEX® 1K devices
  - FLEX 10K® and FLEX® 6000 devices
- Enables rapid prototyping using Altera DSP development boards
- Supports a unified representation of the algorithm and implementation of a DSP system
- Supports the SignalTap® II logic analyzer, an embedded signal analyzer that probes signals from the Altera device on the DSP board and imports the data into the MATLAB work space to facilitate visual analysis
- HDL Import block in AltLib library supports:
  - Import of VHDL or Verilog HDL design entities
  - Import HDL defined in a Quartus project file.
- Hardware in the Loop (HIL) block to enable FPGA hardware accelerated co-simulation with Simulink (AltLab library)
- Avalon® Memory-Mapped (Avalon-MM) blockset in the Interfaces library includes user configurable blocks that you can use to build custom logic that works with Nios® II and other SOPC Builder designs:
  - Low level Avalon-MM Master and Slave interface blocks
  - Wrapped blocks for Avalon-MM Read FIFO and Write FIFO
- Avalon Streaming (Avalon-ST) interface blockset in the Interfaces library supports:
  - Avalon-ST Packet Format Converter block
  - Avalon-ST Sink and Source interfaces
- State Machine Table block in the State Machine Functions library
- Simulation only External RAM block in the Simulation library
- Automatically generates a VHDL testbench
- Automatically starts Quartus II compilation
- Enables bit- and cycle-accurate design simulation

- Supports fast functional simulation of the Video and Image Processing Suite MegaCore functions
- Provides a variety of fixed-point arithmetic and logical operators for use with the Simulink software
- Automatic propagation of signal names to generated HDL
- You can specify most values in the block parameter dialog boxes using MATLAB workspace or masked subsystem variables

General Description

Digital signal processing (DSP) system design in Altera programmable logic devices (PLDs) requires both high-level algorithm and hardware description language (HDL) development tools.

The Altera DSP Builder integrates these tools by combining the algorithm development, simulation, and verification capabilities of The MathWorks MATLAB and Simulink system-level design tools with VHDL and Verilog HDL design flows, including the Altera Quartus II software.

DSP Builder shortens DSP design cycles by helping you create the hardware representation of a DSP design in an algorithm-friendly development environment. You can combine existing MATLAB functions and Simulink blocks with Altera DSP Builder blocks and Altera intellectual property (IP) MegaCore® functions to link system-level design and implementation with DSP algorithm development. In this way, DSP Builder allows system, algorithm, and hardware designers to share a common development platform.

You can use the blocks in DSP Builder to create a hardware implementation of a system modeled in Simulink in sampled time. DSP Builder contains bit- and cycle-accurate Simulink blocks—which cover basic operations such as arithmetic or storage functions—and takes advantage of key device features such as built-in PLLs, DSP blocks, or embedded memory.

You can integrate complex functions by using MegaCore functions in your DSP Builder model. You can also experience the faster performance and richer instrumentation of hardware co-simulation by implementing parts of your design in an FPGA.

The DSP Builder `Signal Compiler` block reads Simulink Model Files (**.mdl**) that are built using DSP Builder and MegaCore functions and generates VHDL files and Tcl scripts for synthesis, hardware implementation, and simulation.

High-Speed DSP with Programmable Logic

Programmable logic offers compelling performance advantages over dedicated digital signal processors.

Programmable logic can be thought of as an array of elements, each of which can be configured as a complex processor routine.

These processor routines can then be linked together in serial (the same way digital signal processor would execute them), or they can be connected in parallel.

In parallel, they offer many times the performance of standard digital signal processors by executing hundreds of instructions at the same time.

Algorithms that benefit from this improved performance include forward-error correction (FEC), modulation/demodulation, and encryption.

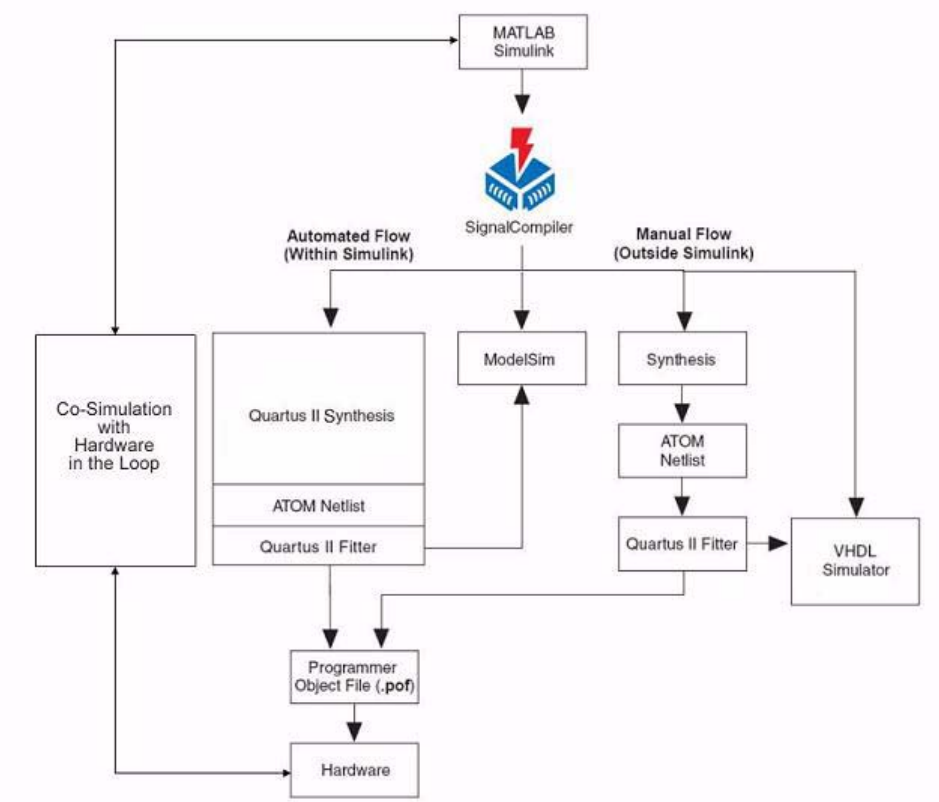
Design Flow

When using DSP Builder, you start by creating a design model in the MATLAB/Simulink software. After you have created your model, you can output VHDL files for synthesis and Quartus II compilation or generate files for VHDL or Verilog HDL simulation. The design flow involves the following steps:

1. Create a model with a combination of Simulink and DSP Builder blocks using the MATLAB/Simulink software.
2. Use the `Signal Compiler` block to analyze your design.
3. Simulate the model in Simulink using a `Scope` block to monitor the results.
4. Run `Signal Compiler` to setup RTL simulation and synthesis.
5. Perform RTL simulation. DSP Builder supports an automated flow for the ModelSim software (using the `TestBench` block). You can also use the generated VHDL for manual simulation in other simulation tools.
6. Use the output files generated by the DSP Builder `Signal Compiler` block to perform RTL synthesis. Alternatively, you can synthesize the VHDL files manually using other synthesis tools.
7. Compile your design in the Quartus II software.
8. Download to a hardware development board and test.

Figure 1–1 shows the system-level design flow using DSP Builder.

Figure 1–1. System-Level Design Flow



For an automated design flow, the `Signal Compiler` block generates VHDL and Tcl scripts for synthesis in the Quartus II software. The Tcl scripts let you perform synthesis and compilation automatically from within the MATLAB and Simulink environment. You can synthesize and simulate the output files in other software tools without the Tcl scripts. In addition, the `Signal Compiler` block generates models and a testbench for VHDL simulation.

Refer to "Design Flows for Synthesis, Compilation & Simulation" on page 3–14 for more information about controlling the DSP Builder design flow using `Signal Compiler`.

2. Getting Started Tutorial

Introduction

This tutorial uses an example amplitude modulation design, **singen.mdl**, to demonstrate the DSP Builder design flow.

The amplitude modulation design example is a modulator that has a sine wave generator, a quadrature multiplier, and a delay element. Each block in the model is parameterizable. When you double-click a block in the model, a dialog box is displayed where you can enter the parameters for the block. Click the **Help** button in these dialog boxes to view on-line help for a specific block.

The instructions in this tutorial assume that:

- You are using a PC running Windows XP.
- You are familiar with the MATLAB, Simulink, Quartus® II, and ModelSim® software and the software is installed on your PC in the default locations.
- The instructions in this tutorial assume that you have basic knowledge of the Simulink software. For information on using the Simulink software, see the Simulink Help.

You can perform a walkthrough by using the **singen.mdl** model file that is provided in the <*DSP Builder install path*>**DesignExamples\Tutorials\GettingStartedSinMdl** directory or you can create your own amplitude modulation model.

Creating the Amplitude Modulation Model

To create the amplitude modulation model, follow the instructions in the following sections.

Create a New Model

To create a new model, perform the following steps:

1. Start the MATLAB software.
2. Choose the **New > Model** command (File menu) to create a new model file.
3. Choose **Save** (File menu) in the new model window.

4. Browse to the directory in which you want to save the file. This directory becomes your working directory. This tutorial uses the working directory *<DSP Builder install path>***DesignExamples****Tutorials****GettingStartedSinMdl****my_SinMdl**.

5. Type the file name into the **File name** box. This tutorial uses the name **singen.mdl**.

6. Click **Save**.

7. Click the MATLAB **Start** button  in the lower left corner (this replaces the Launch Pad of earlier versions of the MATLAB software).

   You can also open Simulink by using the  toolbar icon.

8. Choose **Simulink**, then **Library Browser**.

The following sections describe how to add blocks to your model and simulate the model in Simulink.

Add the Sine Wave Block

Perform the following steps to add the `Sine Wave` block:

1. In the Simulink Library Browser, click **Simulink** and **Sources** library to view the blocks in the Sources library.

2. Drag and drop a `Sine Wave` block into your model (the **singen** window).

3. Double-click the `Sine Wave` block in your model to display the Block Parameters dialog box.

4. Set the `Sine Wave` block parameters as follows:

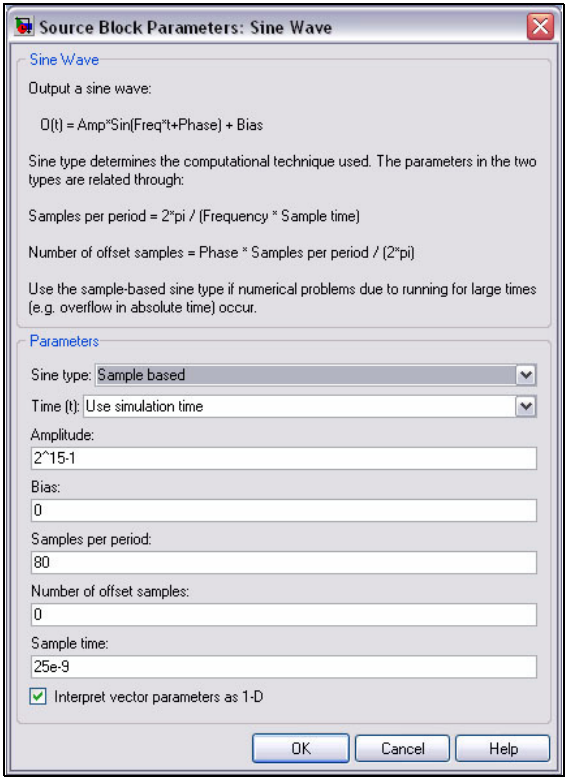
   - **Sine type**: `Sample based`
   - **Time**: `Use simulation time`
   - **Amplitude**: `2^15-1`
   - **Bias**: `0`
   - **Samples per period**: `80`
   - **Number of offset examples**: `0`
   - **Sample time**: `25e-9`
   - **Interpret vector parameters as 1-D**: Turn on

   The completed dialog box is shown in Figure 2–1 on page 2–3.

5. Click **OK**.

See the equation in "Frequency Design Rule" on page 3–4 for information on how you can calculate the frequency.

Figure 2–1. 500-kHz, 16-Bit Sine Wave Specified in the Sine Wave Dialog Box



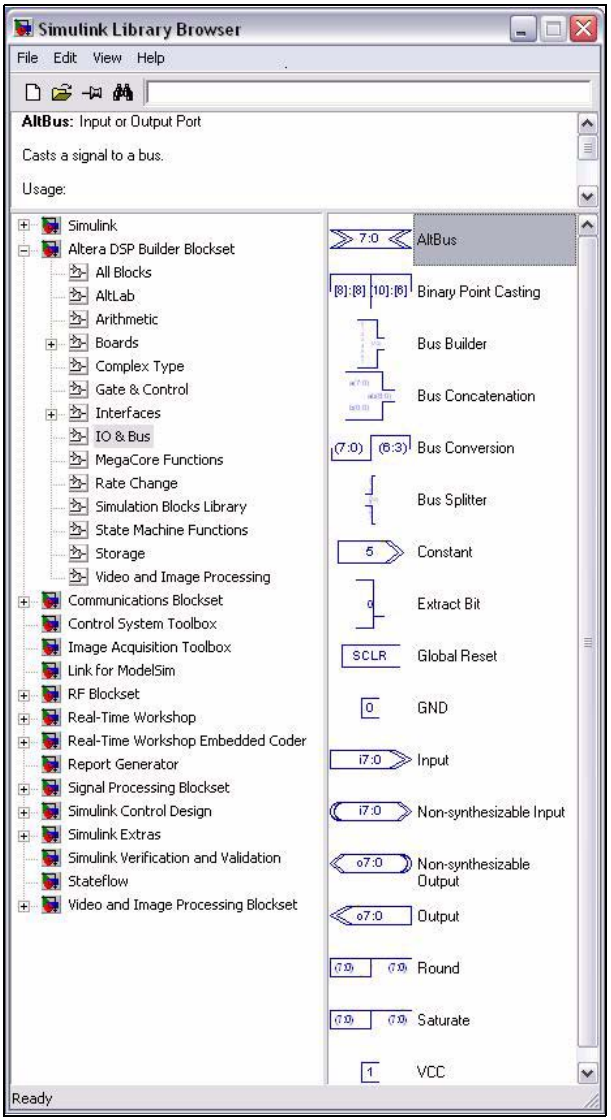
Add the SinIn Block

Perform the following steps to add the `SinIn` block:

1. In the Simulink Library Browser expand the **Altera DSP Builder Blockset** folder to display the DSP Builder libraries (Figure 2–2 on page 2–4).

Leave the Altera DSP Builder folder expanded for the rest of the tutorial as you will add several more blocks from this folder.

Figure 2–2. Altera DSP Builder Folder in the Simulink Library Browser



2. Select the IO & Bus library.

3. Drag and drop the `Input` block from the Simulink Library Browser into your model. Position the block to the right of the `Sine Wave` block.

You can use the Up, Down, Right, and Left arrow keys to adjust the position of a block while it is selected.

If you are unsure how to position the blocks or draw connection lines, see the completed design shown in Figure 2–15 on page 2–17.

4. Click the text `Input` under the block icon in your model. Delete the text `Input` and type the text `SinIn` to change the name of the block instance.

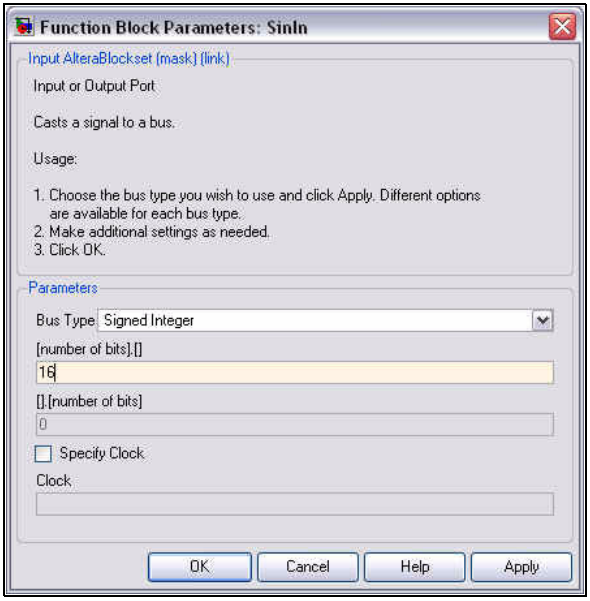
5. Double-click the `SinIn` block in your model to display the Block Parameters dialog box.

6. Set the `SinIn` block parameters as follows:

   - **Bus Type**: `Signed Integer`
   - **[number of bits].[]**: `16`
   - **Specify Clock**: `Off`

   The completed dialog box is shown in Figure 2–3.

Figure 2–3. Setting the 16-Bit Signed Integer Input



7. Click **OK**.

8. Draw a connection line from the right side of the `Sine Wave` block to the left side of the `SinIn` block by holding down the left mouse button and dragging the cursor between the blocks.

Alternatively, you can select a block, hold down the **Ctrl** key and click the destination block to automatically make a connection between the two blocks.

Add the Delay Block

Perform the following steps to add the `Delay` block:

1. Select the Storage library from the **Altera DSP Builder Blockset** folder in the Simulink Library Browser.

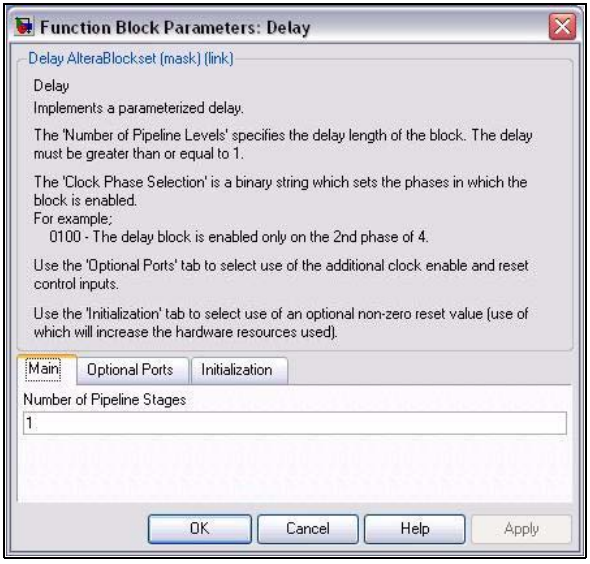
2. Drag and drop the `Delay` block into your model and position it to the right of the `SinIn` block.

3. Double-click the `Delay` block in your model to display the Block Parameters dialog box (Figure 2–4).

4. Set the `Delay` block parameters as follows:

    - **Number of Pipeline Stages**: `1`

Figure 2–4. Setting the Downsampling Delay

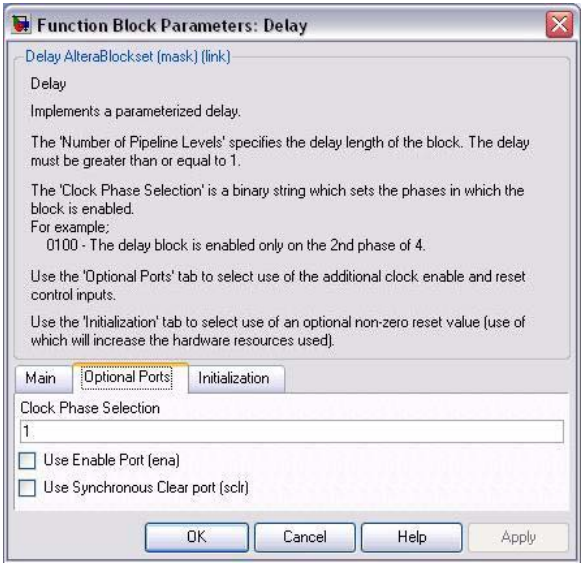


5. Click the **Optional Ports** tab and set the following parameters:

   - **Clock Phase Selection**: `1`
   - **Use Enable Ports**: `Off`
   - **Use Synchronous Clear Port**: `Off`

   The completed dialog box is shown in

Figure 2–5. Delay Block Optional Ports Tab



6. Click **OK**.

7. Draw a connection line from the right side of the `SinIn` block to the left side of the `Delay` block.

Add the SinDelay and SinIn2 Blocks

Perform the following steps to add the `SinDelay` and `SinIn2` blocks:

1. Select the IO & Bus library from the **Altera DSP Builder Blockset** folder in the Simulink Library Browser.

2. Drag and drop two `Output` blocks into your model, positioning them to the right of the `Delay` block.

3. Click the text under the block symbols in your model. Change the block instance names from `Output` and `Output1` to `SinDelay` and `SinIn2`.

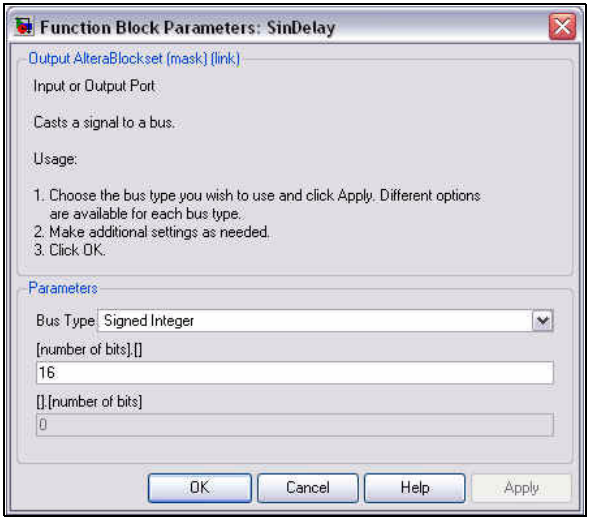
4. Double-click the `SinDelay` block in your model to display the Block Parameters dialog box.

5. Set the `SinDelay` block parameters as follows:

   - **Bus Type**: `Signed Integer`
   - **[number of bits].[]:** `16`.

   The completed dialog box is shown in Figure 2–6.

Figure 2–6. Setting the 16-Bit Signed Output Bus



6. Click **OK**.

7. Repeat steps 4 to 6 for the `SinIn2` block setting the parameters as follows:

   - **Bus Type**: `Signed Integer`
   - **[number of bits].[]:** `16`.

8. Draw a connection line from the right side of the `Delay` block to the left side of the `SinDelay` block.

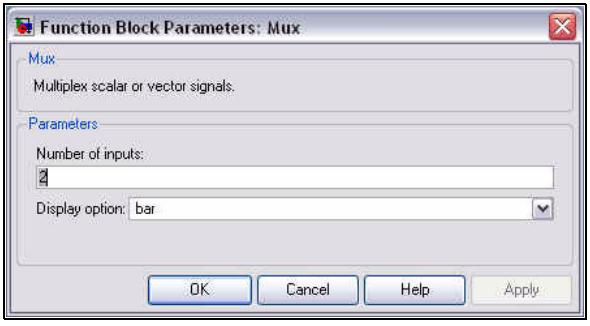
Add the Mux Block

Perform the following steps to add the Mux block:

1. Select the Simulink Signal Routing library in the Simulink Library Browser.
2. Drag and drop a Mux block into your design, positioning it to the right of the SinDelay block.
3. Double-click the Mux block in your model to display the Block Parameters dialog box.
4. Set the Mux block parameters as follows:
   - **Number of inputs**: 2
   - **Display Options**: bar

   The completed dialog box is shown in Figure 2–7.

Figure 2–7. Setting the 2-to-1 Multiplexer



5. Click **OK**.
6. Draw a connection line from the bottom left of the Mux block to the right side of the SinDelay block.
7. Draw a connection line from the top left of the Mux block to the line between the SinIn2 block.
8. Draw a connection line from the SinIn2 block to the line between the SinIn and Delay blocks.

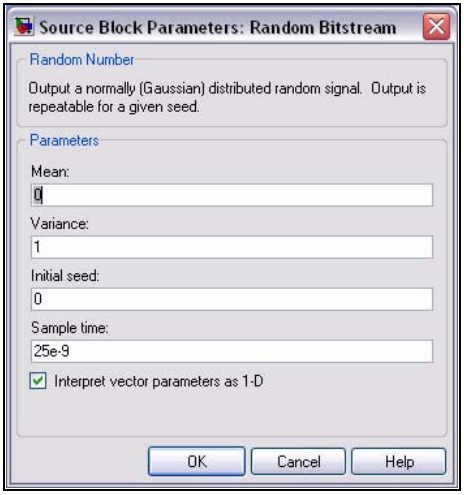
Add the Random Bitstream Block

Perform the following steps to add the `Random Bitstream` block:

1. Select the Simulink Sources library in the Simulink Library Browser.
2. Drag and drop a `Random Number` block into your model, positioning it underneath the `Sine Wave` block.
3. Double-click the `Random Number` block in your model to display the Block Parameters dialog box.
4. Set the `Random Number` block parameters as follows:
   - **Mean**: `0`
   - **Variance**: `1`
   - **Initial seed**: `0`
   - **Sample time**: `25e-9`
   - **Interpret vector parameters as 1-D**: On

   The completed dialog box is shown in Figure 2–8.

Figure 2–8. Setting Up the Random Number Generator



5. Click **OK**.
6. Rename the `Random Noise` block `Random Bitstream`.

Add the Noise Block

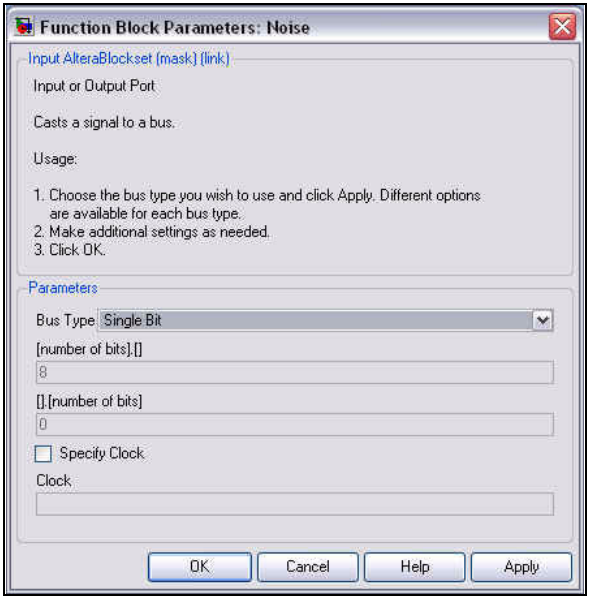
Perform the following steps to add the `Noise` block:

1. Select the IO & Bus library from the **Altera DSP Builder Blockset** folder in the Simulink Library Browser.
2. Drag and drop an `Input` block into your model, positioning it to the right of the Random Bitstream block.
3. Click the text `Input` under the block icon in your model. Rename the block `Noise`.
4. Double-click the `Noise` block to display the Block Parameters dialog box.
5. Choose the `Single Bit` option from the **Bus Type** list.

   The dialog box options change to display only the relevant options when you select a new bus type.

   The completed dialog box is shown in Figure 2–9.

Figure 2–9. Setting the 1-Bit Noise Input Port



6. Click **OK**.

7. Draw a connection line from the right side of the `Random Bitstream` block to the left side of the Noise block.

Add the Bus Builder Block

The `Bus Builder` block converts a bit to a signed bus. Perform the following steps to add the `Bus Builder` block:

1. Select the IO & Bus library from the **Altera DSP Builder Blockset** folder in the Simulink Library Browser.

2. Drag and drop a `Bus Builder` block into your model, positioning it to the right of the `Noise` block.

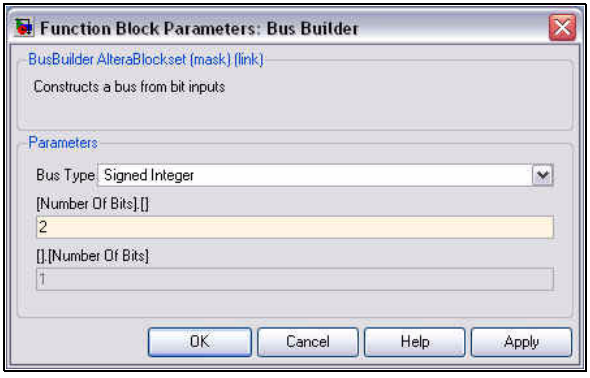
3. Double-click the `Bus Builder` block in your model to display the Block Parameters dialog box.

4. Set the `Bus Builder` block parameters as follows:

   - **Bus Type**: `Signed Integer`
   - **[number of bits].[]**: 2

   The completed dialog box is shown in Figure 2–10.

Figure 2–10. Build a 2-Bit Signed Bus



5. Click **OK**.

6. Draw a connection line from the right side of the `Noise` block to the top left side of the `Bus Builder` block.

Add the GND Block

Perform the following steps to add the GND block:

1. Select the IO & Bus library from the **Altera DSP Builder Blockset** folder in the Simulink Library Browser.
2. Drag and drop a GND block into your model, positioning it underneath the Noise block.
3. Draw a connection line from the right side of the GND block to the bottom left side of the Bus Builder block.

Add the Product Block

Perform the following steps to add the Product block:

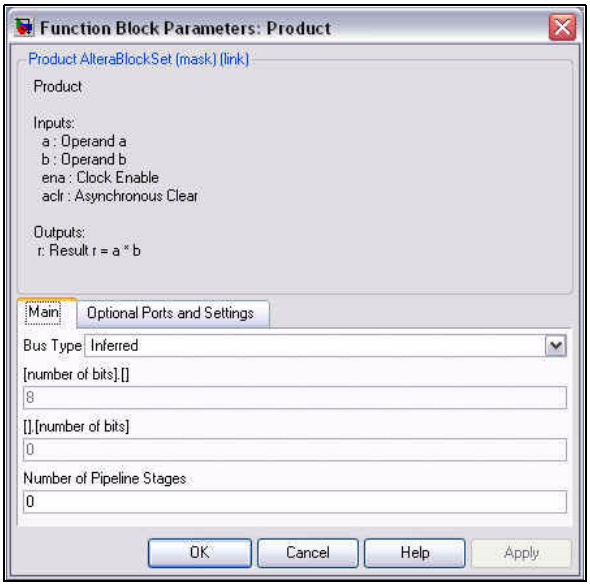
1. Select the Arithmetic library from the **Altera DSP Builder Blockset** folder in the Simulink Library Browser.
2. Drag and drop a Product block into your model, positioning it to the right of the Bus Builder block and slightly above it. Leave enough space so that you can draw a connection line under the Product block.
3. Double-click the Product block to display the Block Parameters dialog box.
4. Set the following Product block parameters:
   - **Bus Type**: Inferred
   - **Number of Pipeline Stages**: 0

The bit width parameters are set automatically when you choose Inferred bus type. The parameters in the **Optional Ports and Settings** tab of this dialog box can be left with their default values.

The completed dialog box is shown in Figure 2–11 on page 2–14.

5. Click **OK**.
6. Draw a connection line from the top left of the Product block to the line between the Delay and SinDelay blocks.

Figure 2–11. Product Block Parameters



Add the StreamMod and StreamBit Blocks

Perform the following steps to add the `StreamMod` and `StreamBit` blocks:

1. Select the IO & Bus library from the **Altera DSP Builder Blockset** folder in the Simulink Library Browser.

2. Drag and drop two `Output` blocks into your model, positioning them to the right of the `Product` block.

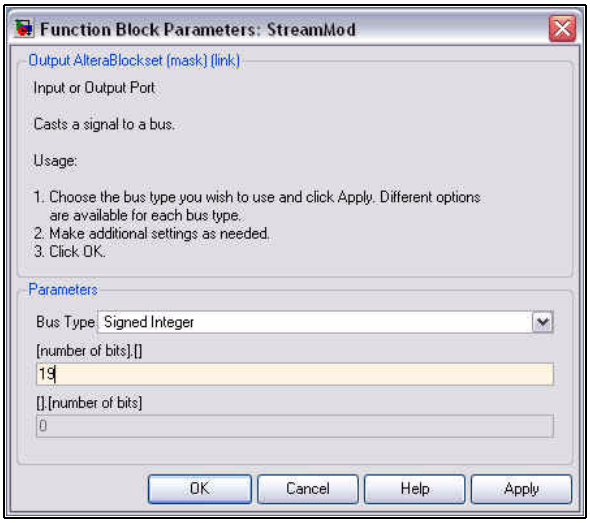
3. Click the text under the block symbols in your model. Change the block instance names from `Output` and `Output1` to `StreamMod` and `StreamBit`.

4. Double-click the `StreamMod` block to display the Block Parameters dialog box (Figure 2–12 on page 2–15).

5. Set the `StreamMod block parameters` as follows:

    - **Bus Type**: `Signed Integer`
    - **[number of bits].[]**: `19`

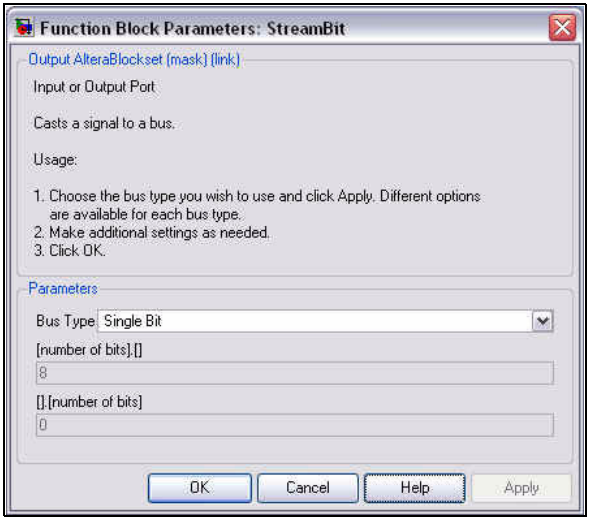
Figure 2–12. Set a 19-Bit Signed Output Bus



6. Click **OK**.

7. Double-click the `StreamBit` block to display the Block Parameters dialog box.

Figure 2–13. Set a Single-Bit Output Bus



8. Set the `StreamMod` block parameters as follows:

   - **Bus Type**: `Single Bit`

   The completed dialog box is shown in Figure 2–13 on page 2–15.

9. Draw connection lines from the right side of the `Product` block to the left side of the `StreamMod` block, and from the right side of the `Bus Builder` block to the left side of the `StreamBit` block.

Add the Scope Block

Perform the following steps to add the `Scope` block:

1. Select the Simulink Sinks library in the Simulink Library Browser.

2. Drag and drop a `Scope` block into your model and position it to the right of the `StreamMod` block.

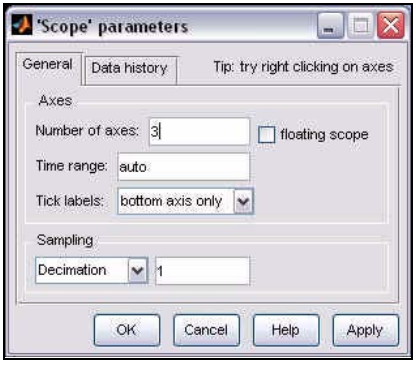
3. Double-click the `Scope` block and click the Parameters icon to display the 'Scope' Parameters dialog box.

4. Set the Scope parameters as follows:

   - **Number of axes:** `3`
   - **Time range**: `auto`
   - **Tick labels**: `bottom axis only`
   - **Sampling**: `Decimation 1`

   The completed dialog box is shown in Figure 2–14.

Figure 2–14. Display Three Signals in Time



5. Click **OK**.

6. Close the Scope.

7. Make connections to connect the complete your design as follows:

   a. From the right side of the `Mux` block to the top left side of the `Scope` block.

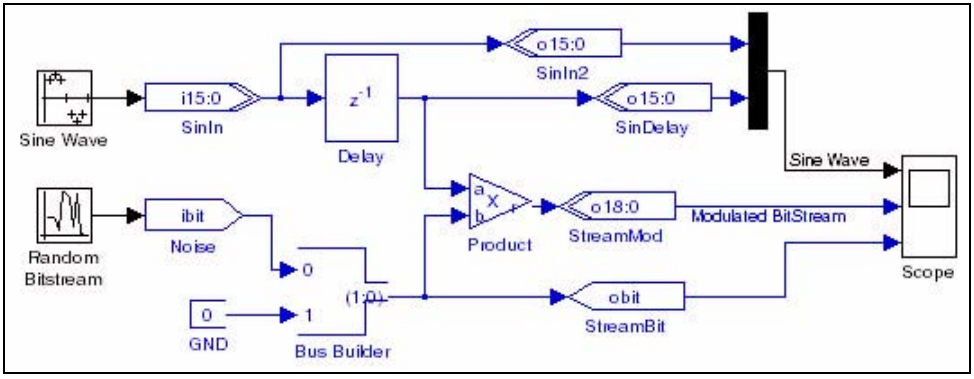
   b. From the right side of the `StreamMod` block to the middle left side of the `Scope` block.

   c. From the right side of the `StreamBit` block to the bottom left of the `Scope` block.

   d. From the bottom left of the `Product` block to the line between the `Bus Builder` block and the `StreamBit` block.

Figure 2–15 shows the required connections.

Figure 2–15. Amplitude Modulation Design Example



Add a Clock Block

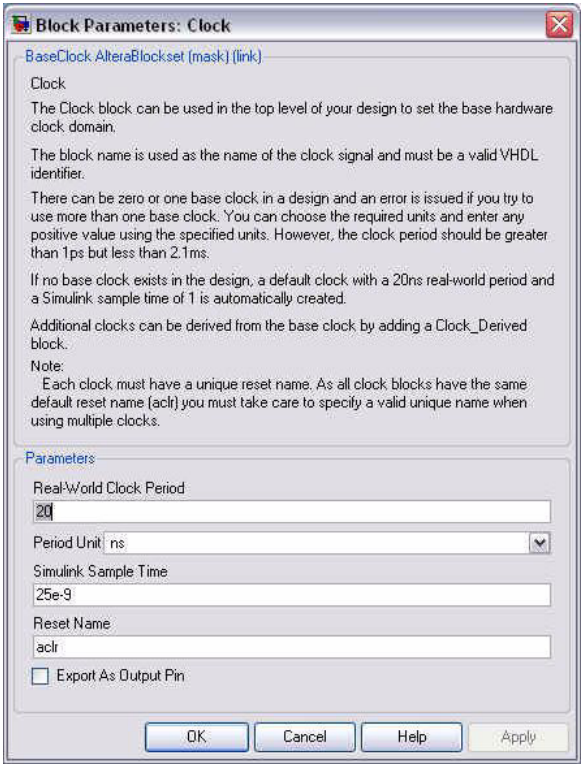
Perform the following steps to add a `Clock` block:

1. Select the AltLab library from the **Altera DSP Builder Blockset** folder in the Simulink Library Browser.

2. Drag and drop a `Clock` block into your model.

3. Double-click on the `Clock` block to display the block parameters dialog box (Figure 2–15).

Figure 2–16. Clock Block Parameters Dialog Box



4. Set the Clock parameters as follows:

   - **Real-World Clock Period:** `20`
   - **Period Unit**: `ns`
   - **Simulink Sample Time**: `25e-9`
   - **Reset Name**: `aclr`
   - **Export As Output Pin**: `Off`

A `clock` block is required in this design to set a Simulink sample time that matches the sample time specified on the `Sine Wave` and `Random Bitstream` blocks.

5. Save your model.

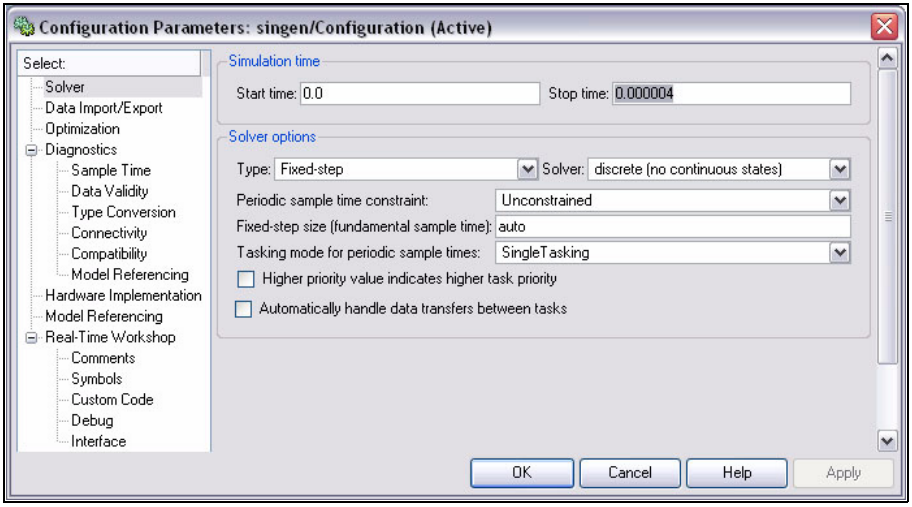
Simulate Your Model in Simulink

To simulate your model in the Simulink software, perform the following steps:

1. Choose **Configuration parameters** (Simulation menu) to display the Configuration dialog box (Figure 2–17).

   - Type `0.000004` in the **Stop time** box to display 200 samples.
   - Choose `Fixed-step` in the Solver options **Type** box.
   - Choose `discrete (no continuous states)` in the **Solver** box.

Figure 2–17. Configuration Parameters

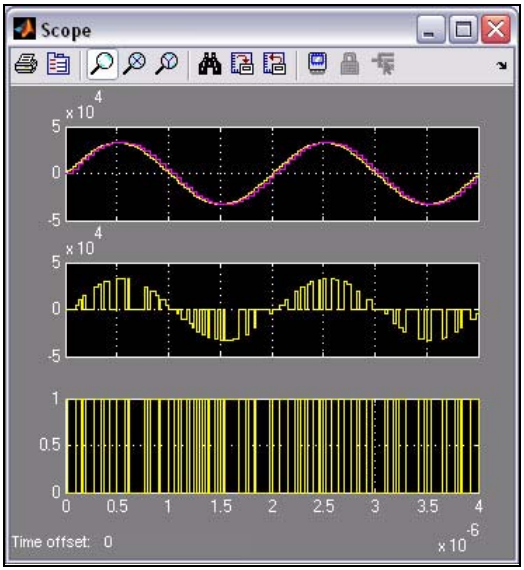


2. Click **OK**.
3. Start simulation by choosing **Start** (Simulation menu) or by pressing Ctrl+T.
4. Double-click the `Scope` block to view the simulation results.

5. Click the binoculars icon to auto-scale the waveforms.

   Figure 2–18 shows the scaled waveforms.

Figure 2–18. Scope Simulation Results



Compiling the Design

To create and compile a Quartus II project for your DSP Builder design, and to program the design onto an Altera FPGA, you need to add the `Signal Compiler` block.

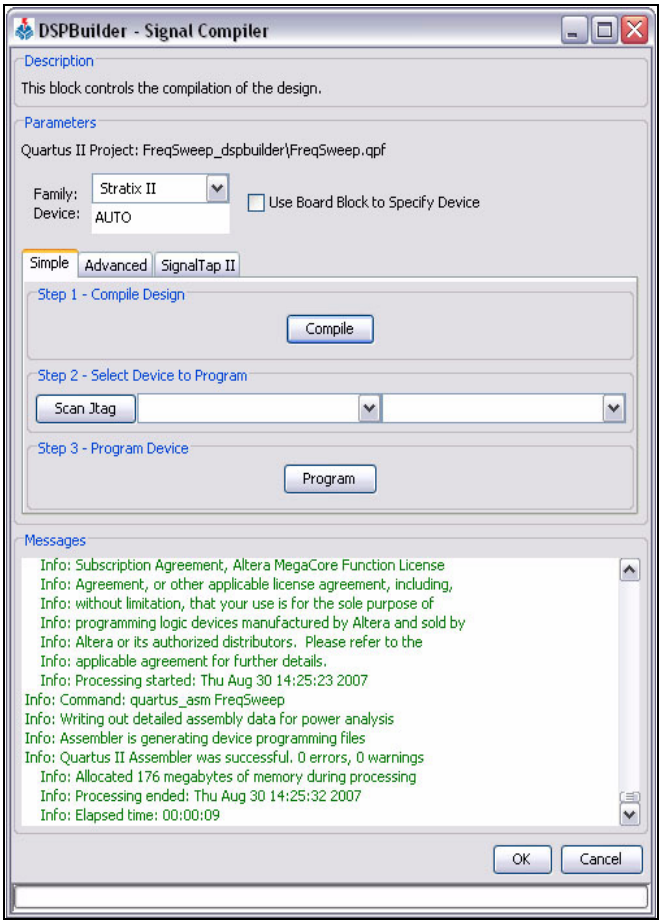
Perform the following steps:

1. Select the AltLab library from the **Altera DSP Builder Blockset** folder in the Simulink Library Browser.

2. Drag and drop a `Signal Compiler` block into your model.

1. Double-click the `Signal Compiler` block in your model to display the Signal Compiler dialog box (Figure 2–19 on page 2–21).

Figure 2–19. Signal Compiler Block Dialog Box



This page allows you to set the target device family. For this tutorial, you can use the default `Stratix` device family.

2. Click **Compile**.
3. When the compilation has completed successfully, click **OK**.
4. Choose **Save** (File menu) to save the model.

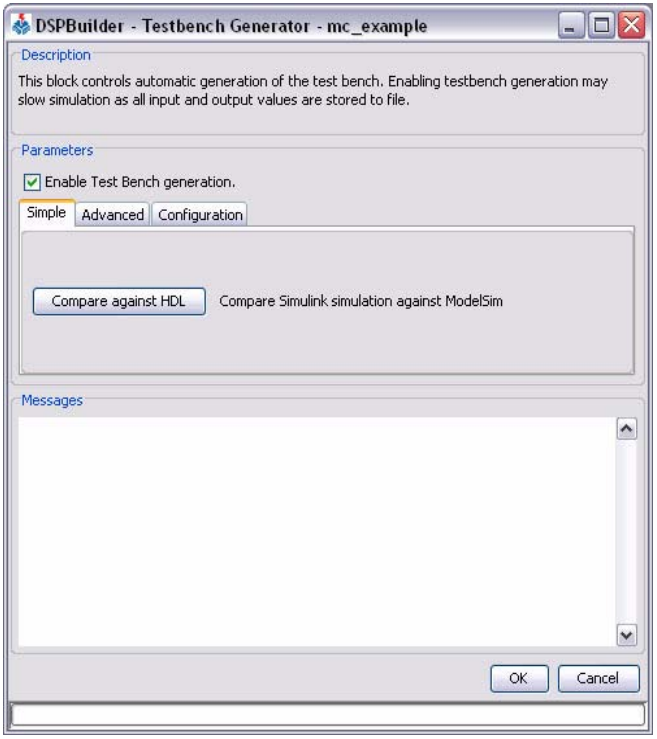
Performing RTL Simulation

To perform RTL simulation with the ModelSim software, you need to add a `TestBench` block.

Perform the following steps:

1. Select the AltLab library from the **Altera DSP Builder BlockSet** folder in the Simulink Library Browser.
2. Drag and drop a `TestBench` block into your model.
3. Double-click on the new `TestBench` block. The Testbench Generator dialog box appears (Figure 2–20).

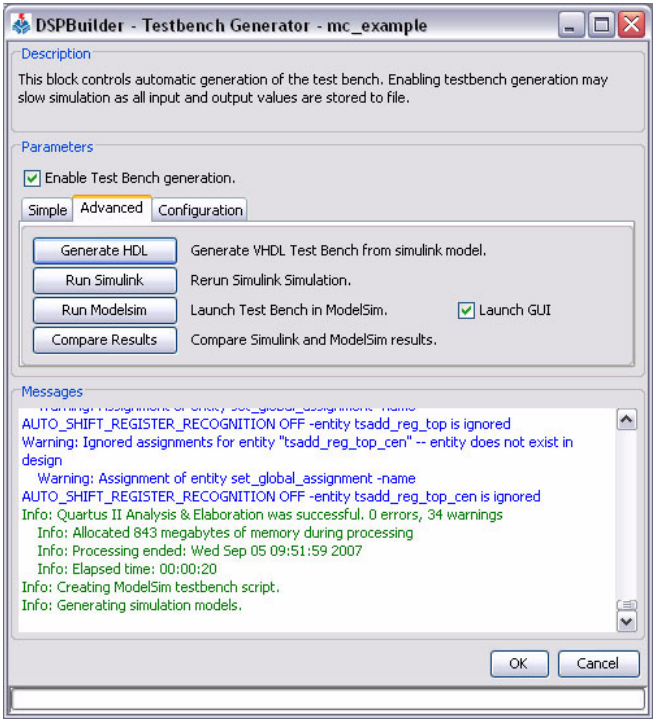
Figure 2–20. Testbench Generator Dialog Box



4. Ensure that **Enable Test Bench generation** is turned on.

5. Select the **Advanced** Tab (Figure 2–21).

Figure 2–21. Testbench Generator Dialog Box Advanced Tab



6. Turn on the **Launch GUI** option. This option causes the ModelSim GUI to be launched when ModelSim simulation is invoked.

7. Click **Generate HDL** to generate a VDHL-based testbench from your model.

8. Click **Run Simulink** to generate Simulink simulation results for the testbench.

9. Click **Run ModelSim** to load the design in ModelSim.

   The design is simulated with the output displayed in the ModelSim Wave window.

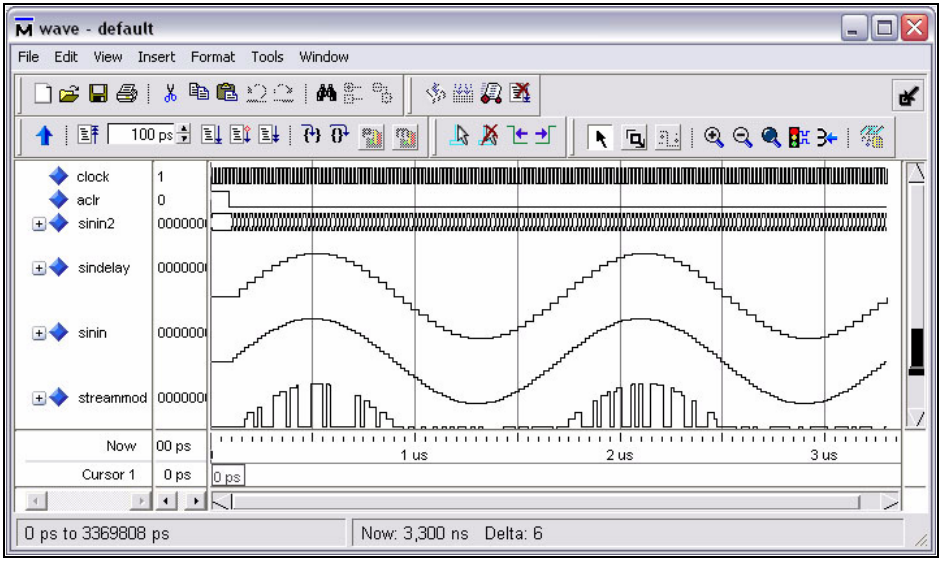
All waveforms are initially shown using digital format in the ModelSim Wave window.

The testbench initializes all of the design registers with a pulse on the `aclr` input signal.

10. Change the format of the `sinin`, `sindelay` and `streammod` signals to analog by right-clicking on the signal names in the waveform and selecting **Properties**. In the **Format** tab, select Analog and specify the height `50` and scale `0.001`.

11. Choose **Zoom Full** from the popup menu in the ModelSim Wave window. The simulation results display as an analog waveform similar to that shown in Figure 2–22.

Figure 2–22. Analog Display



You have now completed the introductory DSP Builder tutorial. The next section shows how you can add a DSP Builder design to a new or existing Quartus II project.

Later chapters in this user guide provide walkthroughs that illustrate some of the additional design features supported by DSP Builder.

Adding the Design to a Quartus II Project

The Quartus II project created by `Signal Compiler` is used internally by DSP Builder. This section describes how to add your design to a new or existing Quartus II project.

Before following these steps, ensure that the design has been compiled using the `Signal Compiler` block as described in "Compiling the Design" on page 2–20.

Creating a Quartus II Project

To create a new Quartus II project, perform the following steps:

1. Start the Quartus II software.
2. Choose **New Project Wizard** from the File menu in the Quartus II software and specify the working directory for your project. For example, **D:\MyQuartusProject**.
3. Specify the name of the project. For example, **NewProject** and the name of the top level design entity for the project.

   The name of the top-level design entity typically has the same name as the project.
4. Click **Next** to display the **New Project Wizard Add Files** page. There are no files to add for this tutorial.
5. Click **Next** to display the **New Project Wizard Family & Device Settings** page and check that the required device family is selected. This should normally be the same device family as specified for `Signal Compiler` in "Compiling the Design" on page 2–20.
6. Click **Finish** to close the wizard and create the new project.

   When you specify a directory that does not already exist, a message asks if the specified directory should be created. Click **Yes** to create the directory.

Add the DSP Builder Design to the Project

To add your DSP Builder design to the project in the Quartus II software:

1. Choose **Tcl Console** from the **Utility Windows** available in the File menu (or use Alt+2) to display the Tcl Console.

2. Run the <*DSP Builder install path*>**DesignExamples\Tutorials\GettingStartedSinMdl\singen_add.tcl** script in the Tcl Console by typing the command:

```
# source <install path>/DesignExamples/Tutorials/GettingStartedSinMdl/singen_add.tcl
```

You must use / separators instead of \ separators in the command path name used in the Tcl console window. You can use a relative path if you organize your design data with the DSP Builder and Quartus II designs in subdirectories of the same design hierarchy.

An example instantiation is added to your Quartus II project.

3. Click the **Files** tab in the Quartus II software.

4. Right click on **singen_example.vhd** and choose **Select Set as Top-Level Entity**.

5. Compile the Quartus II design by choosing **Start Compilation** from the Processing menu (or by using Ctrl+L).

You can copy the component declaration from the example file for your own code.

3. DSP Builder Design Rules

DSP Builder Naming Conventions

DSP Builder generates VHDL files for simulation and synthesis. When there are blocks in the model that share the same VHDL name, the blocks are renamed in the VHDL to avoid name clashes. However, clock and reset ports are never renamed, and an error is issued if they do not have unique names. It is advisable to avoid name clashes on other ports, as this will result in the top level ports of the VHDL being renamed. Therefore, all DSP Builder port names must comply with the following naming conventions:

- VHDL is not case sensitive. For example the input port MyInput and MYINPUT is the same VHDL entity.
- Avoid using VHDL keywords for DSP Builder port names.
- Watch for illegal characters. VHDL identifier names can only contain a - z, 0 - 9, and underscore (_) characters.
- Begin all port names with a letter (a - z). VHDL does not allow identifiers to begin with non-alphabetic characters or end with an underscore.
- Do not use two underscores in succession (__) in port names because it is illegal in VHDL.

White spaces in the names for the blocks/components and signals are converted to an underscore when Signal DSP Builder converts the Simulink model file (**.mdl**) into VHDL.

Using a MATLAB Variable

You can specify many block parameters (such as bit widths and pipeline depth) by entering a MATLAB base workspace or masked subsystem variable. These variables can then be set on the MATLAB command line or from a script. The variable is evaluated and its value passed to the simulation model files. Checks are performed to make sure that the parameters are in the required range. Values that can be set in this way are annotated as “Parameterizable” in the block parameter tables shown in the reference manual.

Although DSP Builder no longer restricts parameters to 51 bits, MATLAB evaluates parameter values to doubles. This restricts the possible values to 51-bit numbers expressible by a double.

For information about which values are parameterizable, refer to the *DSP Builder Reference Manual* or to the block descriptions that can be accessed using the **Help** option in the right mouse menu for each block.

Bit Width Design Rule

You must specify the bit width at the source of the data path. The DSP Builder propagates this bit width from the source to the destination through all intermediate blocks.

Some intermediate DSP Builder blocks must have a bit width specified, while others have specific bit width growth rules which are described in the documentation for each block. Some blocks which allow bit widths to be specified optionally, have an `Inferred` type setting that allows a growth rule to be used. For example, in the amplitude modulation design (see Chapter 2, Getting Started Tutorial), the `SinIn` and `SinDelay` blocks have a bit width of 16. Therefore, a bit width of 16 is automatically assigned to the intermediate `Delay` block.

Fixed-Point Notation

Table 3–1 describes the fixed-point notation used for I/O formats in the DSP Builder block descriptions.

Table 3–1. Fixed-Point Notation

Description	Notation	Simulink-to-HDL Translation (1), (2)
Signed binary: fractional (SBF) representation; a fractional number	[L].[R] where: • [L] is the number of bits to the left of the binary point and the MSB is the sign bit • [R] is the number of bits to the right of the binary point	A Simulink SBF signal $A_{[L].[R]}$ maps in VHDL to STD_LOGIC_VECTOR({L + R - 1} DOWNTO 0)
Signed binary; integer (INT)	[L] where: • [L] is the number of bits of the signed bus and the MSB is the sign bit	A Simulink signed binary signal $A_{[L]}$ maps to STD_LOGIC_VECTOR({L - 1} DOWNTO 0)
Unsigned binary; integer (UINT)	[L] where: • [L] is the number of bits of the unsigned bus	A Simulink unsigned binary signal $A_{[L]}$ maps to STD_LOGIC_VECTOR({L - 1} DOWNTO 0)
Single bit integer (BIT)	[1] where: • the single bit can have values 1 or 0	A Simulink single bit integer signal maps to STD_LOGIC

Notes to Table 3–1:

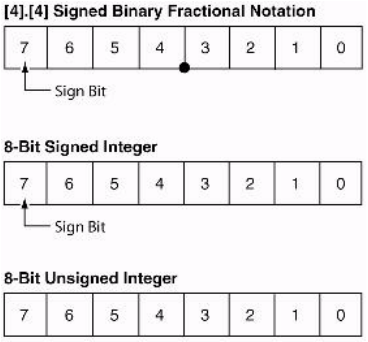
(1) STD_LOGIC_VECTOR and STD_LOGIC are VHDL signal types defined in the IEEE library (**ieee.std_logic_1164.all** and **ieee.std_logic_signed.all** packages).

(2) For designs in which unsigned integer signals are used in Simulink, DSP Builder translates the Simulink unsigned bus type with width w into a VHDL signed bus of width $w + 1$ where the MSB bit is set to 0.

The DSP Builder internal wire type (*SBF_L_R*, *INT_L*, *UINT_L*, or *BIT*) is displayed when **Port Data Types** is turned on in the **Port/Signal Displays** section of the Simulink **Format** menu. For example, *SBF_8_4* for a 12-bit signed binary fractional data type with 4 fractional bits.

Figure 3–1 graphically compares the signed binary fractional, signed binary, and unsigned binary number formats.

Figure 3–1. Number Format Comparison



Binary Point Location in Signed Binary Fractional Format

For hardware implementation, Simulink signals must be cast into the desired hardware bus format. Therefore, floating-point values must be converted to fixed-point values. This conversion is a critical step for hardware implementation because the number of bits required to represent a fixed-point value plus the location of the binary point affects both the amount of the hardware resources used and the system accuracy.

Choosing a large number of bits gives excellent accuracy—the fixed-point result is almost identical to the floating-point result—but consumes a large amount of hardware. The designer's task consists of finding the right size/accuracy trade-off. DSP Builder speeds up the design cycle by enabling simulation with fixed-point and floating-point signals in the same environment.

The `Input` block casts floating-point Simulink signals of type double into fixed-point signals. The fixed-point signals are represented in signed binary fractional (SBF) format as shown below:

- [*number of bits*].[]—Represents the number of bits to the left of the binary point including the sign bit.
- [].[*number of bits*]—Represents the number of bits to the right of the binary point.

In VHDL, the signals are typed as STD_LOGIC_VECTOR (see *Note (1)* in Table 3–1 on page 3–2).

For example, the 4-bit binary number 1101 is represented as:

Simulink This signed integer is interpreted as –3

VHDL This signed STD_LOGIC_VECTOR is interpreted as –3

If you change the location of the binary point to 11.01, that is, two bits on the left side of the binary point and two bits on the right side, the numbers are represented as:

Simulink This signed fraction is interpreted as –0.75

VHDL This signed STD_LOGIC_VECTOR is interpreted as –3

From a system-level analysis standpoint, multiplying a number by –0.75 or –3 is obviously very different, especially when looking at the bit width growth. In the one case the multiplier output bus grows on the most significant bit (MSB), in the other case the multiplier output bus grows on the least significant bit (LSB). In both cases, the binary numbers are identical. However, the location of the binary point affects how a simulator formats the signal representation. For complex systems, you can adjust the binary point location to define the signal range and the area of interest.

For more information on number systems, refer to *AN 83: Binary Numbering Systems*.

Frequency Design Rule

This section describes the frequency design rules for single and multiple clock domains.

Single Clock Domain

If your design does not contain a `PLL` block or `Clock_Derived` block, DSP Builder uses synchronous design rules to convert a Simulink design into hardware. All DSP Builder registered blocks (such as the `Delay` block) operate on the positive edge of the single clock domain, which runs at the system sampling frequency.

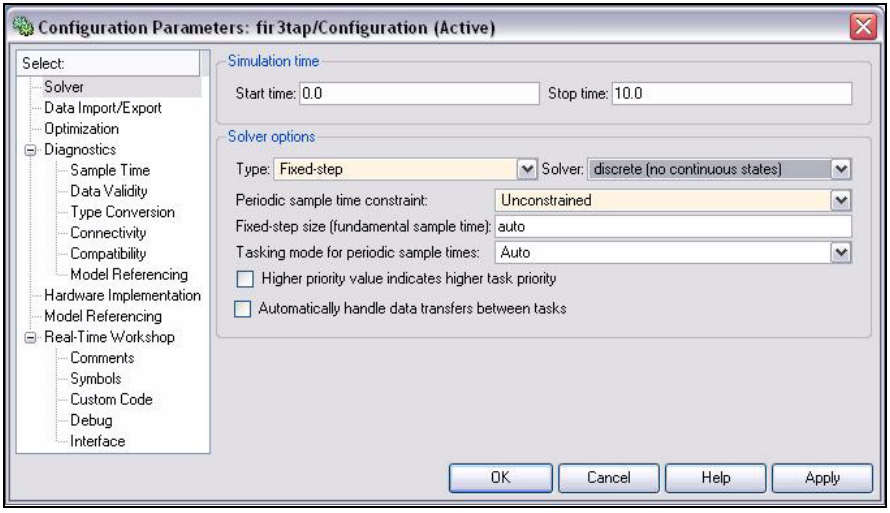
The clock pin is not graphically displayed in Simulink unless the `Clock` block is used. However, when DSP Builder converts the design to VHDL it automatically connects the clock pin of the registered blocks (such as the `Delay` block) to the single clock domain of the system.

The default clock pin is named `clock` and there is also a default active-low reset pin named `aclr`.

By default, Simulink does not graphically display the clock enable and reset input pins of the DSP Builder registered blocks. When DSP Builder converts a design to VHDL, it automatically connects these pins. You can access and drive these optional ports by checking the appropriate option in the parameter dialog box.

Simulink issues a warning if you are using an inappropriate solver for your model. You should set the solver options to fixed-step discrete. To set the solver options, choose **Configuration parameters** (Simulation menu) to open the Configuration Parameters dialog box and choose the Solver page (Figure 3–2). Choose the options for a `Fixed-step` type and `discrete (no continuous states)` solver.

Figure 3–2. Configuration Parameters for Simulation



For Simulink simulation, all DSP Builder blocks (including registered DSP Builder blocks) use the sampling period specified in the `Clock` block. If there is no `Clock` block in the design, the DSP Builder blocks use a sampling frequency of 1. You can use the `Clock` block to change the Simulink sample period and the hardware clock period.

Multiple Clock Domains

A DSP Builder model can operate using multiple Simulink sampling periods. The clock domain can be specified within some DSP Builder block sources, such as the `Counter` block. The clock domain can also be specified within DSP Builder rate change blocks such as `Tsamp`.

When using multiple sampling periods, DSP Builder must associate each sampling period to a physical clock domain that can be available from an FPGA PLL or a clock input pin. Therefore, the DSP Builder model must contain DSP Builder rate change blocks such as `PLL` or `Clock_Derived` at the top level.

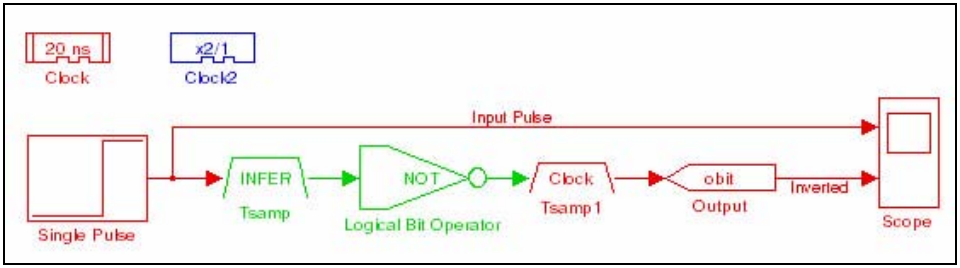
You can use a `PLL` block to synthesize additional clock signals from a reference clock signal. These internal clock signals are multiples of the system clock frequency. Table 3–2 on page 3–8 shows the number of PLL internal clock outputs supported by each device family.

If your design contains the `PLL` block, `Clock` or `Clock_Derived` blocks, the DSP Builder registered blocks operate on the positive edge of one of the block's output clocks. You should set the solver options to variable-step discrete. To set the solver options, choose **Configuration parameters** (Simulation menu) to open the Configuration Parameters dialog box and choose the Solver page. Choose the options for a `Variable-step` type and `discrete (no continuous states)` solver.

To ensure a proper hardware implementation of a DSP Builder design using multiple clock domains, consider the following:

- Do not use DSP Builder combinational blocks for rate transitions to ensure that the behavior of the DSP Builder Simulink model is identical to the generated RTL representation. Figure 3–3 illustrates an incorrect use of the DSP Builder `Logical Bit Operator` (NOT) block.

Figure 3–3. Example of Incorrect Usage: Mixed Sampling Rate on a NOT Block



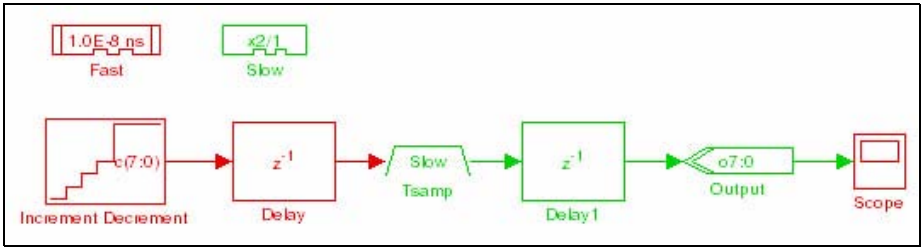
- Two DSP Builder blocks can operate with two different sampling periods. However within most DSP Builder blocks the sampling period of each input port and each output port must be identical.

Although this rule applies to a large majority of DSP Builder blocks, there are some exceptions such as the `Dual-Clock FIFO` block where the sampling period of the read input port is expected to be different than the sampling period of the write input port.

- For a data path using mixed clock domains, additional register decoupling may be required around the register that is between the domains.

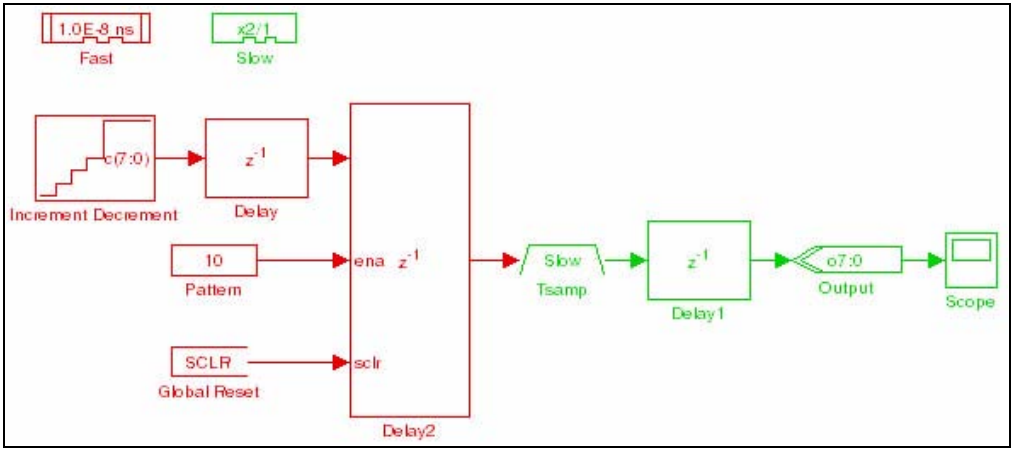
  This is especially true when the source data rate is higher than the destination register, in other words, when the data of a register is toggling at the higher rate than the register's clock pin, as shown in Figure 3–4.

Figure 3–4. Data Toggling Faster than Clock



A stable hardware implementation is shown in Figure 3–5.

Figure 3–5. Stable Hardware Implementation



Using Clock and Clock_Derived Blocks

DSP Builder maps the `Clock` and `Clock_Derived` blocks to two hardware device input pins; one for the clock input, and one for the reset input for the clock domain. A design may contain zero or one `Clock` block and zero or more `Clock_Derived` blocks.

The `Clock` block defines the base clock domain, and `Clock_Derived` blocks define other clock domains whose sample times are specified in terms of the base clock sample time. If there is no `Clock` block, a default base clock is used, with a Simulink sample time of 1, and a hardware clock period of 20us.

This feature is available across all device families supported by DSP Builder. If no `Clock` block is present, a default clock pin named `clock` and a default active-low reset pin named `aclr` are used.

`Signal Compiler` assigns a clock buffer and a dedicated clock-tree-to-clock-signal-input pin automatically in order to maintain minimum clock skew. If the design contains more `Clock` and `Clock_Derived` blocks than there are clock buffers available, non dedicated routing resources are used to route the clock signals.

Using the PLL Block

DSP Builder maps the `PLL` block to the hardware device PLL.

Table 3–2 shows the number of PLL internal clock outputs supported by each device family.

Table 3–2. Device Support for PLL Clocks

Device Family	Number of PLL Clocks
Stratix III	9
Stratix II GX	6
Stratix II	6
Stratix GX	6
Stratix	6
Arria GX	6
Cyclone III	5
Cyclone II	3
Cyclone	2

Figure 3–6 shows an example of multiple-clock domain support using the PLL block.

Figure 3–6. MultipleClockDelay.mdl

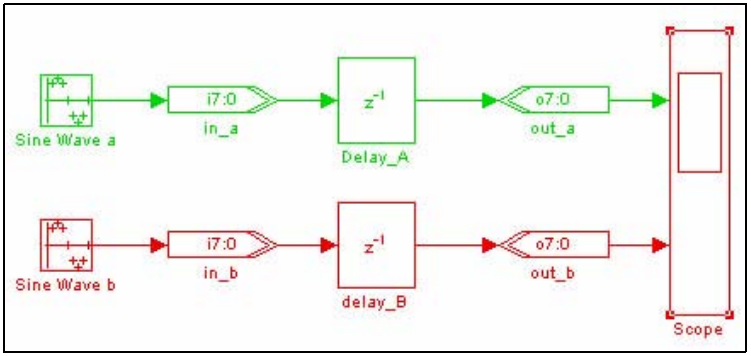


Figure 3–7 shows the clock setting configuration for the PLL block in the **MultipleClockDelay.mdl** design. Output clock *PLL_clk0* is set to 800 ns, and output clock *PLL_clk1* is set to 100 ns.

Figure 3–7. PLL Setting

Data path A (shown in green in Figure 3–6 on page 3–9) operates on output clock *PLL_clk0* and data path B (shown in red in Figure 3–6) operates on output clock *PLL_clk1*. These clocks are specified by setting the **Specify Clock** option and entering the clock name in the block parameter dialog box for each input block.

In this design, the **Sample Time** parameters for the `Sine Wave a` block and `Sine Wave b` block are set explicitly to 1e-006 and 1e-007, so that data is provided to the input blocks at the rate at which they sample.

Using Advanced PLL Features

The DSP Builder `PLL` block supports the fundamental multiplication and division factor for the PLL. If you want to use other PLL features (such as phase shift, duty cycle), do so in a separate Quartus II Project:

- Create a new Quartus II project and use the MegaWizard Plugin Manager to configure the `ALTPLL` block.
- Add the DSP Builder .**mdl** file as a source file.
- You can then create a top level design that instantiates your ALTPLL variation and your DSP Builder design.

Timing Semantics Between Simulink & HDL Simulation

DSP Builder uses the Simulink engine to simulate the behavior of hardware components. However, there are some fundamental differences between the step-based simulation in Simulink and the event-driven simulation used for VHDL and Verilog HDL designs.

DSP Builder bridges this gap by establishing a set of timing semantics for translating between the Simulink and HDL environments.

Simulink Simulation Model

The Simulink timing mode recommended for use with DSP Builder is a discrete fixed-step simulation, to facilitate correlation between HDL and Simulink simulation. This is configured in the **Configuration Parameters** dialog box (Simulation menu) in Simulink (See Figure 3–2 on page 3–5). Each step is a discrete unit of simulation. The clock is quantized in an idealized manner as a cycle counter.

At the beginning of each step, Simulink provides each block with known inputs. Functions are evaluated and the resultant outputs are propagated within the current step. The outputs of the model are the results of all of these computations.

For all steps, Simulink blocks produce output signals. Outputs varying based on inputs received in the same step are referred to as direct feedthrough. Some DSP Builder blocks may include direct feedthrough outputs, depending on the parameterization of each block.

HDL Simulation Models

Hardware simulation is driven by a clock signal and the availability of input stimuli. The testbench script generated by the `TestBench` block has been designed to feed input signals to the HDL simulator in order to maintain correlation between HDL and Simulink simulation.

Simulation models in the DSP Builder libraries evaluate their logic on positive clock edges. To avoid any timing conflicts, external inputs transition on negative clock edges.

Registered outputs are updated on positive clock edges. `TestBench` block-generated inputs arrive on negative clock edges, causing an apparent half-cycle delay in the arrival of output (see Figure 3–8 on page 3–13).

Startup & Initial Conditions

The testbench includes a global reset for each clock domain. All blocks (except the `HDL Import` and MegaCore function blocks) automatically connect any reset on the hardware to the global asynchronous reset for the clock domain.

When a block explicitly declares an asynchronous reset, this reset is `OR`ed with the global reset.

A `Global Reset` block (`SCLR`), which corresponds to this hardware signal is provided in the Altera DSP Builder Blockset IO & Bus library.

The global reset signal is used as a reset prior to meaningful simulation. When converting from the Simulink domain to the hardware domain, the reset period is considered to be before the Simulink simulation begins. Therefore, in Simulink simulation, the `Global Reset` block outputs only a constant zero and has no simulation behavior. This means that the hardware is connected to reset, and thus reset at the start of a ModelSim testbench simulation.

DSP blocks or MegaCore functions may have additional initial conditions or startup states which are not automatically reset by the global reset signal.

DSP Builder Global Reset Circuitry

By default, Simulink does not graphically display the clock enable and reset input pins on DSP Builder registered blocks. When DSP Builder converts a design to HDL, it automatically connects the implied clock enable and reset pins.

If you turn on the optional ports in the parameter dialog box for each of the DSP Builder registered blocks, you can access and drive the clock enable and reset input pins graphically in the Simulink software.

In the HDL domain, an asynchronous reset is used for the registered DSP Builder blocks, as shown in this behavioral VHDL code example:

```
process(CLOCK, RESET)
begin
   if RESET = '1' then
      dout <= (others => '0');
   else if CLOCK'event and CLOCK = '1' then
      dout <= din;
   end if;
end
```

In addition, when targeting a development board, the Block Parameters dialog box for the DSP Board configuration block typically includes a **Global Reset Pin** selection box where you can choose from a list of pins that correspond to the DIP and push button switches.

The reset logic polarity can be either active-high or active-low.

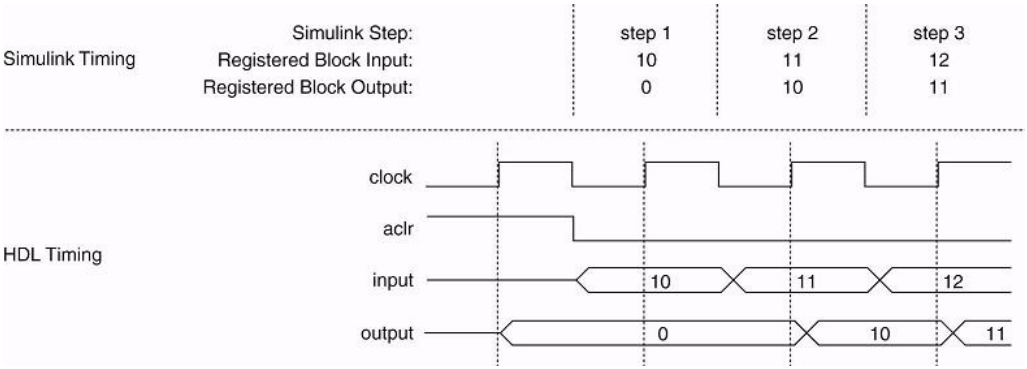
When active-low is selected, the value of the reset signal in Simulink simulation is still 0 for inactive and 1 for active. However, a `NOT` gate is inserted on the input pin in the generated hardware. The value of the reset signal in simulation is therefore the value as it exists across the internal design, rather then the value at the input pin.

The Quartus® II synthesis tool interprets this reset as an asynchronous reset, as discussed above, and uses an input of the logic element look-up table to instantiate this function. The HDL simulation functions properly in this case because the testbench produces the reset input as required.

Reference Timing Diagram

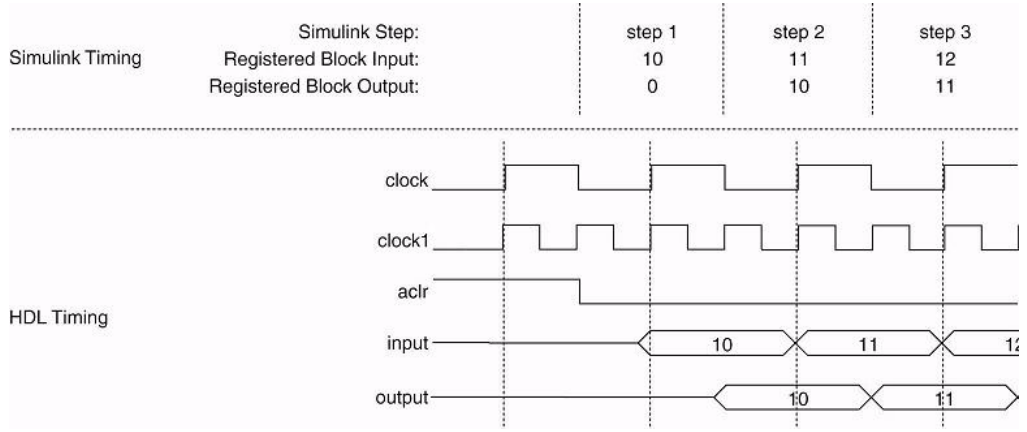
Figure 3–8 shows the timing relationships in a hypothetical case where a register is fed by the output of a counter. The counter output begins at 10, in other words, the value is 10 during the first Simulink clock step.

Figure 3–8. Single-Clock Timing Relationships



This timing is not true when crossing clock domains. For example, Figure 3–9 shows the timing delays introduced in a design with a derived clock that has half the base clock period. In general, DSP Builder is not cycle-accurate when crossing clock domains.

Figure 3–9. Multiple-Clock Timing Relationships



Signal Compiler and TestBench Design Rules

The `Signal Compiler` block uses Quartus II synthesis to convert a Simulink design into synthesizable VHDL including generation of a VHDL testbench and other supporting files for simulation and synthesis.

`Signal Compiler` assumes that the design complies with the Simulink rules and that any variables and inherited variables have been propagated through the whole design.

You should always run a simulation in Simulink before running `Signal Compiler`. This will update all variables in the design (including workspace variables and inherited parameters), setup certain blocks (such as memory blocks, and input from/output to workspace blocks), and trap any design errors that do not comply with Simulink rules.

The `Input` and `Output` blocks map to input and output ports in VHDL and so they mark the edge of the generated system. You would normally connect these blocks to Simulink simulation blocks for your testbench. An `Output` block should not connect to another Altera block. If you connect more Altera blocks (that map to HDL), you get empty ports in your HDL and it does not compile for synthesis.

For more information on the `Input` and `Output` blocks, refer to the *IO & Bus Library* chapter of the *DSP Builder Reference Manual*.

Design Flows for Synthesis, Compilation & Simulation

You can use `Signal Compiler` to control your design flow for synthesis, compilation, and simulation. DSP Builder supports the following flows:

- *Automatic Flow*—The automated flow allows you to control the entire design process from within the MATLAB/Simulink environment using the `Signal Compiler` block. With this flow, the design is compiled inside a temporary Quartus II project. The results of the synthesis and compilation are displayed in the Signal Compiler **Messages** box. You can also use the automated flow to download your design into supported development boards.
- *Manual Flow*—You can also add the .**mdl** file to an existing Quartus II project using the <*model name*>**_add.tcl** script. This script is generated whenever the `Signal Compiler` or `TestBench` block is run and can be used to add the .**mdl** file and any imported HDL to your project. You can then instantiate your design in HDL.
- *Simulation Flow*—Use the `TestBench` block to compile your design for Modelsim simulation. For this to work, ModelSim must be on your path. You can automatically compare the Simulink and Modelsim simulation results.

For an example of using the `Signal Compiler` block, refer to page 2–17 of the "Getting Started Tutorial". For information about setting parameters for the `Signal Compiler` block, refer to the *Signal Compiler Block* section in the *AltLab Library* chapter of the *DSP Builder Reference Manual*.

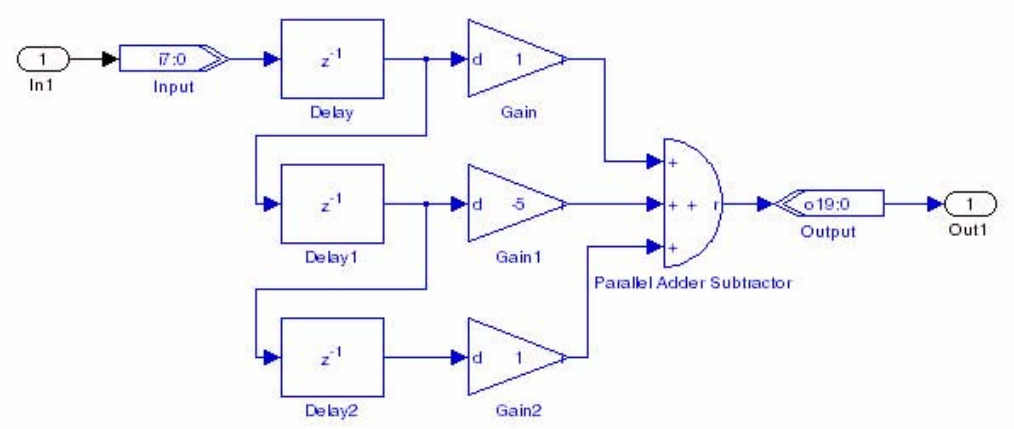
Data Width Propagation

You can specify the bit width of many Altera blocks in the Simulink design. However, you do not need to specify the bit width for all blocks. If the bit width is not explicitly specified, DSP Builder assigns a bit width during the Simulink-to-VHDL conversion by propagating the bit width from the source of a data path to its destination.

Some intermediate DSP Builder blocks must have a bit width specified, while others have specific bit width growth rules which are described in the documentation for each block. Some blocks which allow bit widths to be specified optionally allow a growth rule to be used; this is the `Inferred` type setting.

The following design example illustrates bit-width propagation (Figure 3–10).

Figure 3–10. 3-Tap FIR Filter

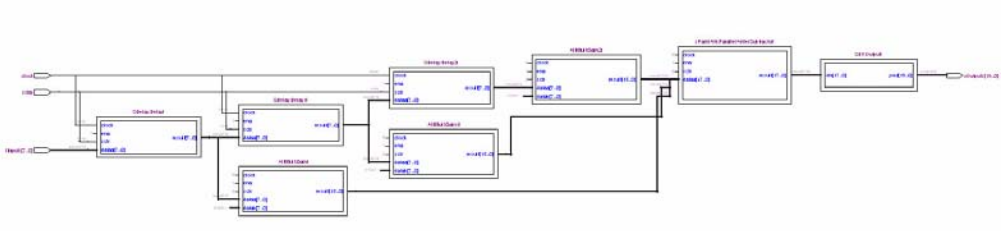


The **fir3tapsub.mdl** design is a 3-tap finite impulse response (FIR) filter and has the following attributes:

- The input data signal is an 8-bit signed integer bus
- The output data signal is a 20-bit signed integer bus
- Three `Delay` blocks are used to build the tapped delay line
- The coefficient values are {1.0000, -5.0000, 1.0000}, a `Gain` block performs the coefficient multiplication

Figure 3–11 shows the RTL representation of **fir3tapsub.mdl** created by `Signal Compiler`.

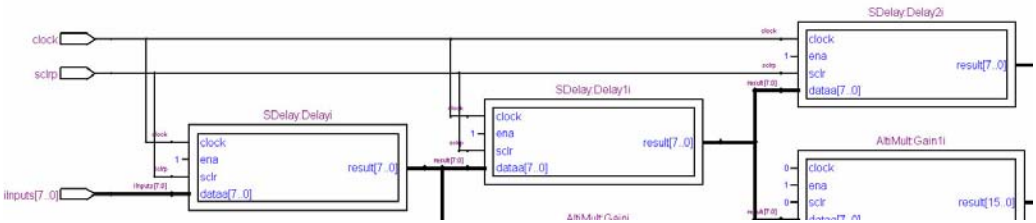
Figure 3–11. 3-Tap FIR Filter in Quartus II RTL View



Tapped Delay Line

The bit width propagation mechanism starts at the source of the data path, in this case at the `Input` block which is an 8-bit input bus. This bus feeds the register `U0`, which feeds `U1`, which feeds `U2`. DSP Builder propagates the 8-bit bus in this register chain where each register is eight bits wide. See Figure 3–12.

Figure 3–12. Tap Delay Line in Quartus II Version RTL Viewer



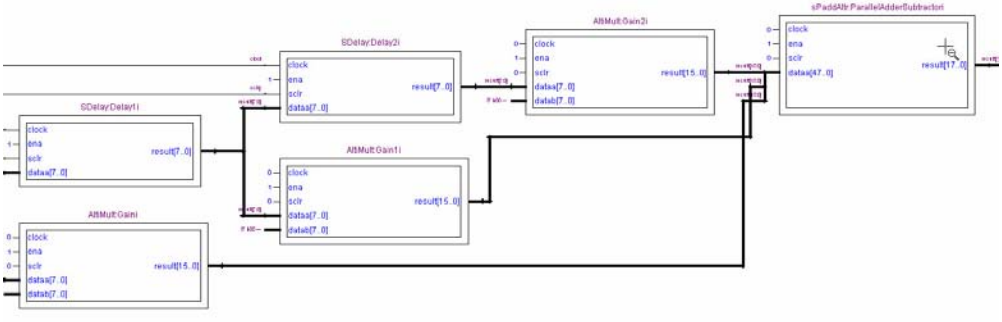
Arithmetic Operation

Figure 3–13 shows the arithmetic section of the filter, which computes the output `yout`:

$$yout[k] = \sum_{i=0}^{2} x[k-i]c[i]$$

Where *c[i]* are the coefficients and *x*[*k* - *i*] are the data.

Figure 3–13. 3-Tap FIR Filter Arithmetic Operation in Quartus II Version RTL Viewer



The design requires three multipliers and one parallel adder. The arithmetic operations increase the bus width in the following ways:

- Multiplying $a \times b$ in SBF format (where *l* is left and *r* is right) is equal to:

  [*la*].[*ra*] × [*lb*].[*rb*]

  The bus width of the resulting signal is:
  ([*la*] + [*lb*]).([*ra*] + [*rb*])

- Adding $a + b + c$ in SBF format (where *l* is left and *r* is right) is equal to:

  [*la*].[*ra*] + [*lb*].[*rb*] + [*lc*].[*rc*]

  The bus width of the resulting signal is:

  (max([*la*], [*lb*], [*lc*]) + 2).(max([*ra*], [*rb*], [*rc*]))

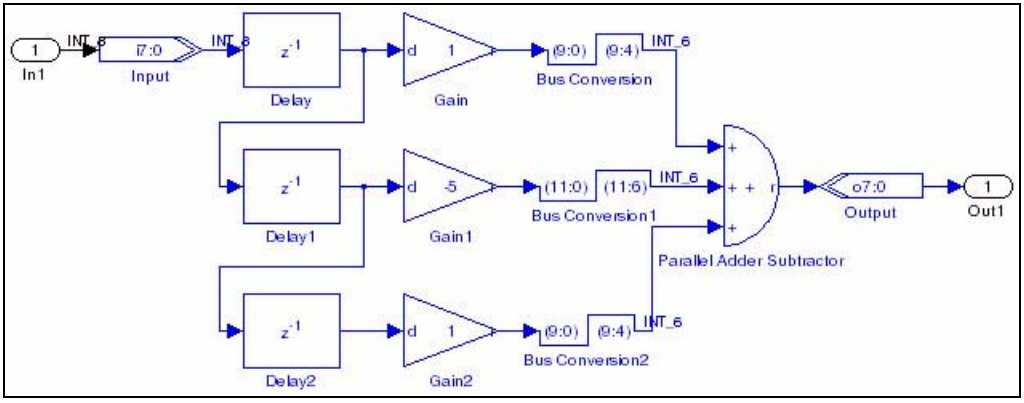
The parallel adder has three input buses of 14, 16, and 14 bits. To perform this addition in binary, DSP Builder automatically sign extends the 14 bit buses to 16 bits. The output bit width of the parallel adder is 18 bits, which covers the full resolution.

There are several options that can change the internal bit width resolution and therefore change the size of the hardware required to perform the function described in Simulink:

- Change the bit width of the input data.
- Change the bit width of the output data. The VHDL synthesis tool will remove the unused logic.
- Insert a `Bus Conversion` block to change the internal signal bit width.

Figure 3–14 shows how `Bus Conversion` blocks can be used to control internal bit widths.

Figure 3–14. 3-Tap Filter with BusConversion to Control Bit Widths

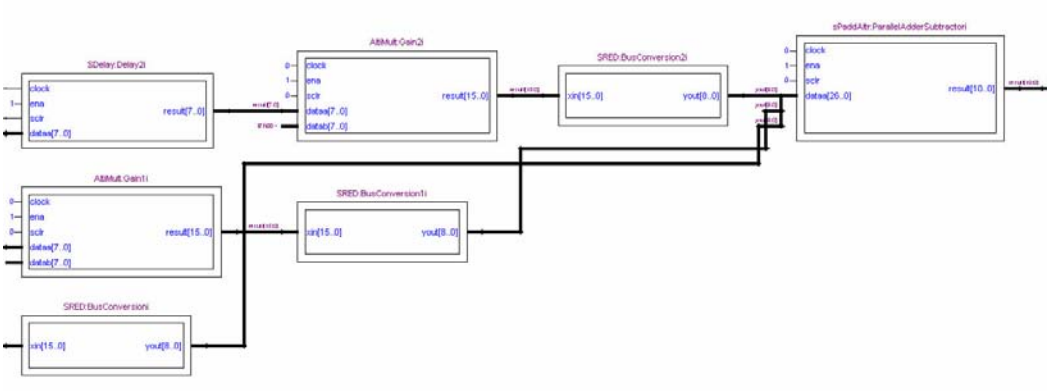


In this example, the output of the `Gain` block has 4 bits removed. (Port data type display is enabled in this example and shows that the inputs to the `Delay` blocks are of type `INT_8` but the outputs from the `Bus Conversion` blocks are of type `INT_6`.)

Bus conversion can also be achieved by inserting an `AltBus`, `Round`, or `Saturate` block.

The RTL view illustrates the effect of this truncation. The parallel adder required has a smaller bit width and the synthesis tool reduces the size of the multiplier to have a 9-bit output. See Figure 3–15.

Figure 3–15. 3-Tap Filter with BusConversion to Control Bit Widths in Quartus II RTL Viewer



Refer to "Fixed-Point Notation" on page 3–2 for more information.

Clock Assignment

DSP Builder identifies registered DSP Builder blocks such as the `Delay` block and implicitly connects the clock, clock enable, and reset signals in the VHDL design for synthesis. When the design does not contain a `Clock` block, `Clock_Derived` block, or `PLL` block, all of the registered DSP Builder block clock pins are implicitly connected to a single clock domain (signal 'clock' in VHDL).

Clock domains are defined by the clock source blocks: the `Clock` block, the `Clock_Derived` block and the `PLL` block.

The `Clock` block defines the base clock domain. You can specify its Simulink sample time and hardware clock period directly. If no `Clock` block is used, there is a default base clock with a Simulink sample time of 1. You can use the `Clock_Derived` block to define clock domains in terms of the base clock. The sample time of a derived clock is specified as a multiple and divisor of the base clock sample time.

The `PLL` block maps to a hardware PLL. You can use it to define multiple clock domains with sample times specified in terms of the PLL input clock. The PLL input clock may be either the base clock or a derived clock.

Each clock domain has an associated reset pin. The `Clock` block and each of the `Clock_Derived` blocks have their own reset pin, which is named in the block's parameter dialog. The clock domains of the `PLL` block share the reset pin of the `PLL` block's input clock.

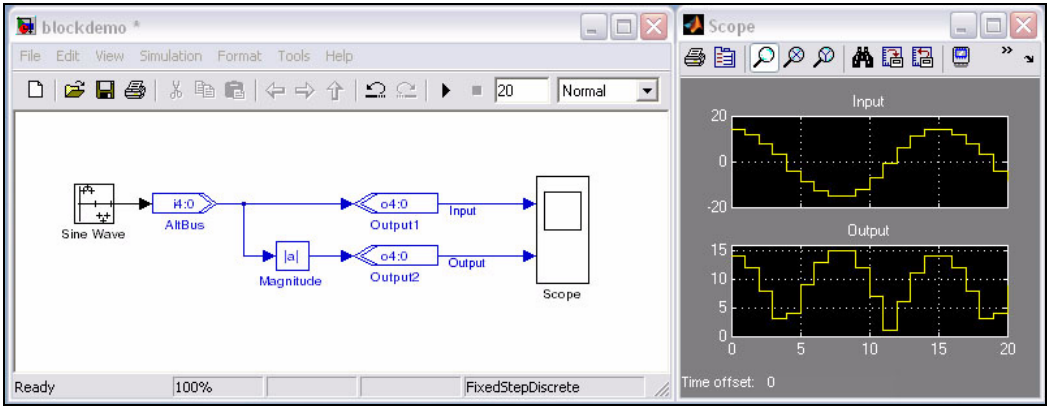
When the design contains clock source blocks, DSP Builder implicitly connects the clock pins of all the registered blocks to the appropriate clock pin or PLL output. DSP Builder also connects the reset pins of the registered blocks to the top-level reset port for the block's clock domain.

DSP Builder blocks fall into two clocking categories:

- *Combinational blocks*—The output always changes at the same sample time slot as the input.
- *Registered blocks*—The output changes after a variable number of sample time slots.

Figure 3–16 illustrates DSP Builder block combinational behavior.

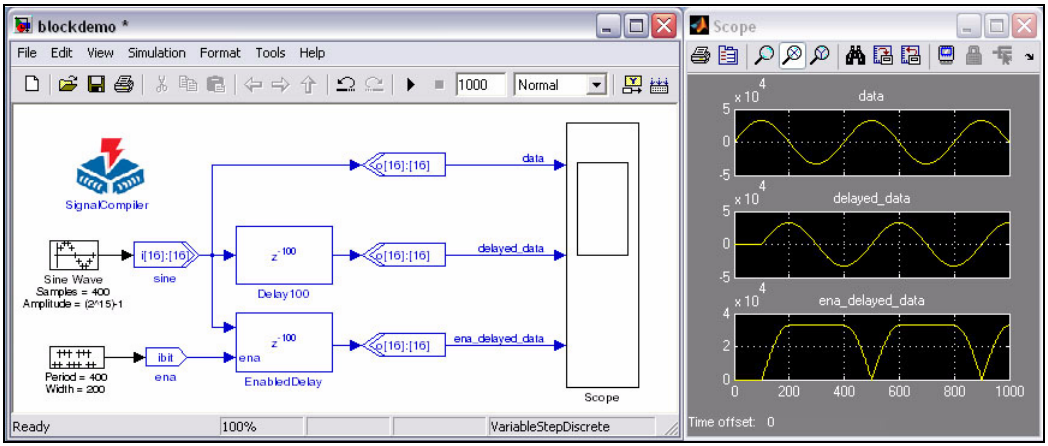
Figure 3–16. Magnitude Block: Combinational Behavior



The `Magnitude` block translates as a combinational signal in VHDL. DSP Builder does not add clock pins to this function.

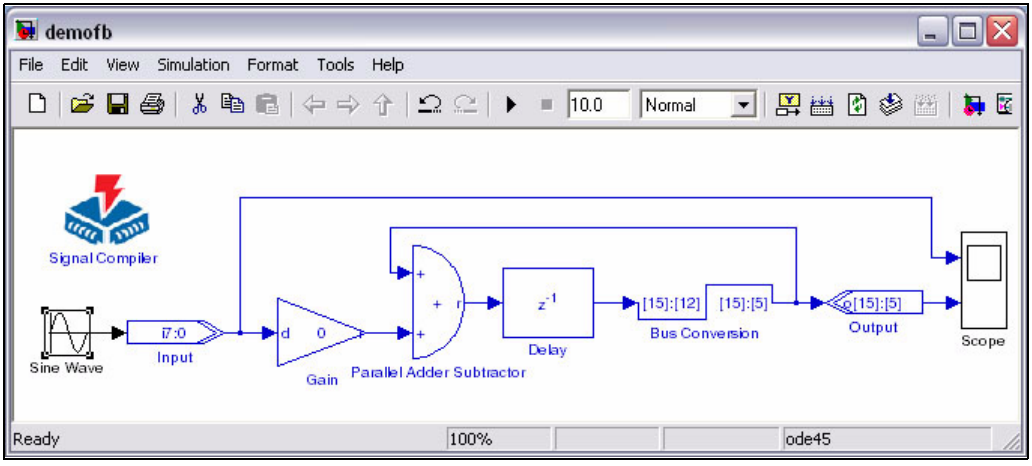
Figure 3–17 on page 3–21 illustrates the behavior of a registered DSP block. In the VHDL netlist, DSP Builder adds clock pin inputs to this function. The `Delay` block, with the **Clock Phase Selection** parameter equal to 100, is converted into a VHDL shift register with a decimation of three and an initial value of zero.

Figure 3–17. Delay Block: Registered Behavior



For feedback circuitry (that is, the output of a block fed back into the input of a block), a registered block must be in the feedback loop. Otherwise, an unresolved combinational loop is created. See Figure 3–18.

Figure 3–18. Feedback Loop



You can design multi-rate designs by using the PLL block and assigning different sampling periods on registered DSP Builder blocks.

Alternatively, you can design multi-rate designs without the DSP Builder PLL block by using a single clock domain with clock enable and the following design rules:

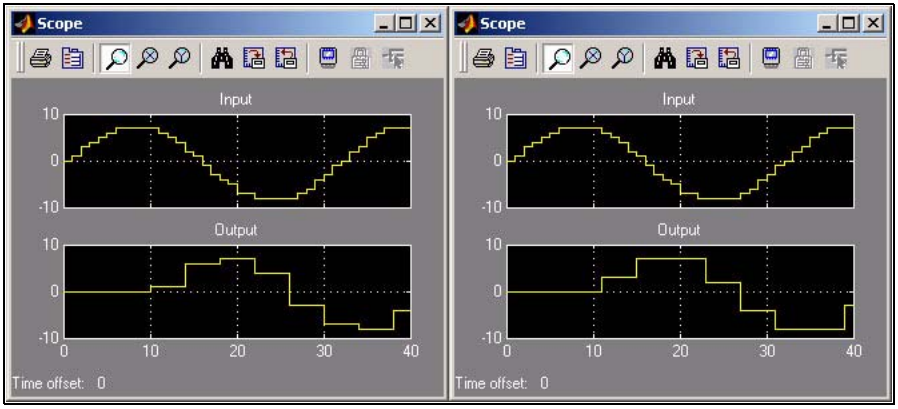
- The fastest sample rate is an integer multiple of the slower sample rates. The values are specified in the **Clock Phase Selection** field in the Block Parameters dialog box for the Delay block.
- The **Clock Phase Selection** box accepts a binary pattern string to describe the clock phase selection. Each digit or bit of this string is processed sequentially on every cycle of the fastest clock. When a bit is equal to one, the block is enabled; when a bit is equal to zero, the block is disabled. For example, see Table 3–3.

Table 3–3. Clock Phase Selection Example

Phase	Description
1	The Delay block is always enabled and captures all data passing through the block (sampled at the rate 1).
10	The Delay block is enabled every other phase and every other data (sampled at the rate 1) passes through.
0100	The Delay block is enabled on the 2nd phase out of 4 and only the 2nd data out of 4 (sampled at the rate 1) passes through. In other words, the data on phases 1, 3, and 4 do not pass through the Delay block.

Figure 3–19 compares the scopes for the Delay block operating at a one quarter rate on the 1000 and 0100 phases, respectively.

Figure 3–19. 1000 as Opposed to 0100 Phase Delay



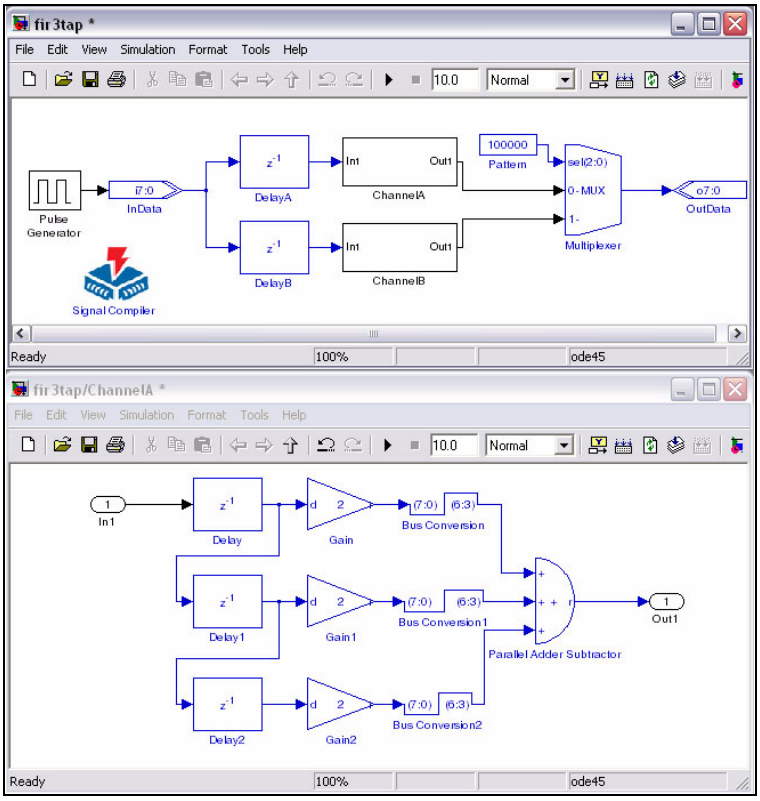
Hierarchical Design

DSP Builder supports hierarchical design using the Simulink `Subsystem` block.

DSP Builder preserves the hierarchy structure in a VHDL design and each Simulink model file (**.mdl**) hierarchical level translates into one VHDL file.

For example, Figure 3–20 illustrates a hierarchy for a design **fir3tap.mdl**, which implements two FIR filters.

Figure 3–20. Hierarchical Design Example



For information on naming the `Subsystem` block instances, refer to "DSP Builder Naming Conventions" on page 3–1.

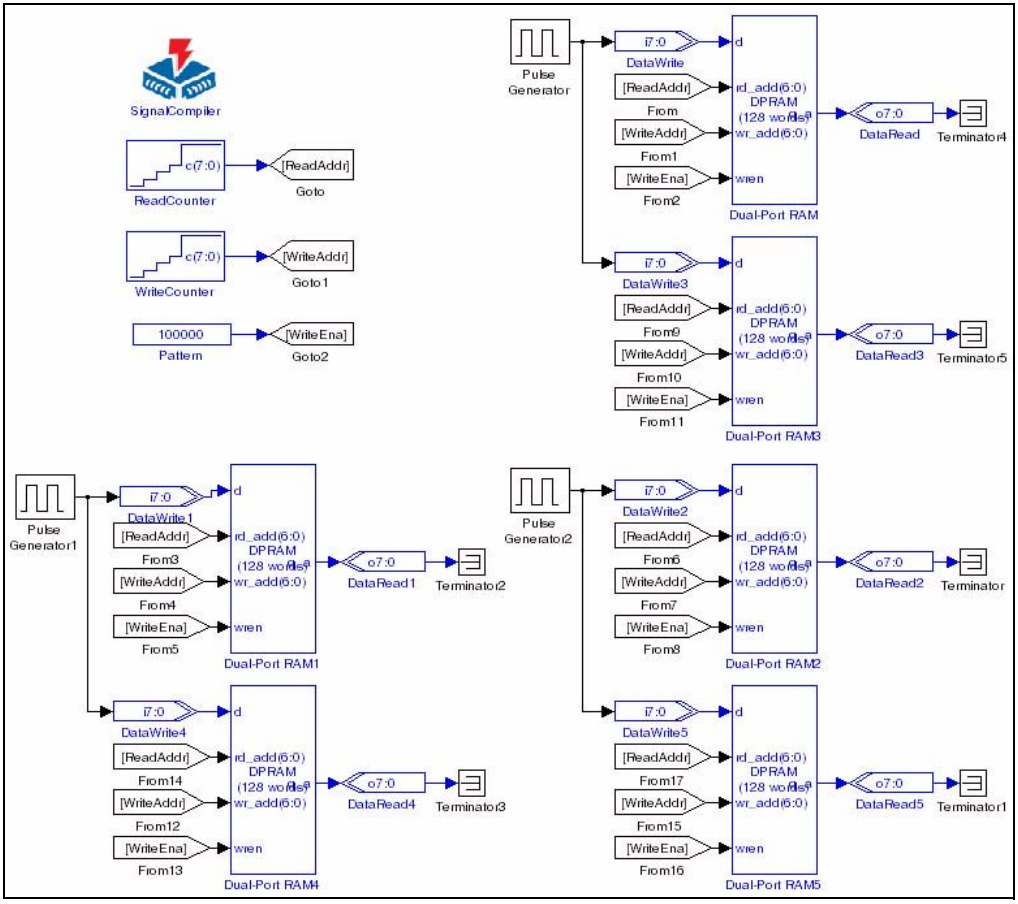
Goto & From Block Support

DSP Builder supports the `Goto` and `From` blocks from the Signal Routing folder in the generic Simulink library.

You can use these blocks for large fan-out signals and to enhance the diagram clarity.

Figure 3–21 shows an example of the `Goto` and `From` blocks. The Goto blocks ([`ReadAddr`], [`WriteAddr`], and [`WriteEna`] are used with the `From` blocks ([`ReadAddr`], [`WriteAddr`], and [`WriteEna`], which are connected to the dual-port RAM blocks.

Figure 3–21. Goto & From Block Example



Black Boxing & HDL Import

You can add your own VHDL or Verilog HDL code to the design and specify which subsystem block(s) should be translated into VHDL by DSP Builder. This process, called black boxing, can be implemented implicitly or explicitly.

An explicit black box uses the `HDL Input`, `HDL Output`, `HDL Entity`, and `Subsystem Builder` blocks.

For information on using these blocks to create an explicit black box, refer to "Subsystem Builder Walkthrough" in Chapter 9.

An implicit black box uses the `HDL Import` block to instantiate the black box subsystem.

For information on implicit black boxing using your own HDL code, refer to the "HDL Import Walkthrough" in Chapter 9.

Using a MATLAB Array or HEX File to Initialize a Block

You can use a MATLAB array to specify the values entered in the `LUT` block or to initialize the `Dual-Port RAM`, `Single-Port RAM`, `True Dual-Port RAM`, or `ROM` blocks. You can also use an Intel format HEX file to initialize a `RAM` or `ROM` block.

If the data values specified by the MATLAB array or HEX file are not exactly representable in the selected data type, they are rounded and a warning is issued. The values are rounded by expressing the number in binary format, then truncating to the specified width. This results in rounding towards minus infinity.

For example, if the input value is `-0.25` (minimally expressed in signed binary fractional two's compliment format as `111`) and the selected target data format is signed fractional `[1].[1]`, then the value is truncated to `11` = `-0.5`. The value is rounded towards minus infinity to the nearest representable number.

Similarly, if you select unsigned integer data type and the value is 1.9, this is rounded down to 1.

Device Support

DSP Builder supports the following target Altera device families which can be selected in `Signal Compiler`: Stratix®, Stratix GX, Stratix II, Stratix II GX, Stratix III, Arria™ GX, Cyclone®, Cyclone II, Cyclone III, APEX 20K, APEX 20KE, APEX 20KC, APEX II, FLEX® 10KE, FLEX 6000, and ACEX® 1K.

Memory Options

A number of the blocks in the Storage library allow you to choose the required memory block type. In general, all supported memory block types are listed as options although some may not be available for all device families.

Table 3–4 shows the device families which support each memory block type.

Table 3–4. Supported Memory Block Types

Memory Block Type	Device Family
M144K	Stratix III
M9K	Stratix III, Cyclone III
MLAB	Stratix III
M-RAM	Stratix III, Stratix II GX, Stratix II, Stratix GX, Stratix, Arria GX
M4K	Stratix II GX, Stratix II, Stratix GX, Stratix, Arria GX, Cyclone II, Cyclone
M512	Stratix II GX, Stratix II, Stratix GX, Stratix, Arria GX

For more information about each memory block type, refer to the Quartus II online help.

Comparison Utility

DSP Builder provides a simple utility that can be used to run simulation comparison between Simulink and ModelSim from the command line:

```
alt_dspbuilder_verifymodel('modelname.mdl', 'logfile.txt')↵
```

A testbench GUI is temporarily displayed displaying messages as the comparison is performed. The command returns true (1) or false (0) according to whether the simulation results match and the output is recorded in the specified log file.

For more information on running a comparison between Simulink and ModelSim, refer to "Performing RTL Simulation" in Chapter 2.

Adding Quartus II Constraints

You can add additional Quartus II assignments or constraints that are not supported within DSP Builder by creating a Tcl script in your design directory. Any file named **<*model name*>_add_user.tcl** is automatically sourced when you run `Signal Compiler`.

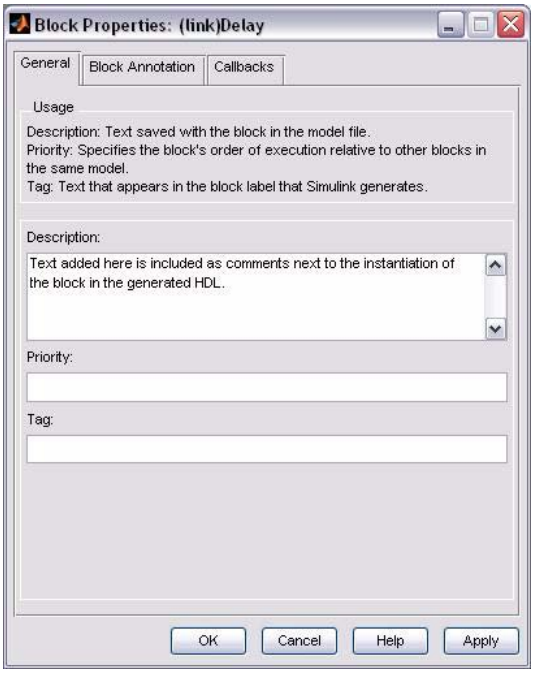
Loading Additional ModelSim Commands

When you have imported HDL as a black box, DSP Builder creates a **DSPBuilder<model name>_import** subdirectory. Any Tcl script named ***_add_msim.tcl** in this subdirectory is automatically sourced when you launch ModelSim. You should not modify the generated scripts, but you can create you own scripts such as **<*user name*>_add_msim.tcl** which contain additional ModelSim commands that you want to load into ModelSim.

Adding Comments to Blocks

You can add comments to any DSP Builder block by right -clicking on the block to display the Block Properties dialog box and entering text in the Description field of the dialog box as shown in Figure 3–22.

Figure 3–22. Adding Comments to a Block



The text will be included as comments next to the instantiation of the block in the generated HDL.

4. Using MegaCore Functions

Introduction

Altera offers a large selection of off-the-shelf MegaCore® functions. Designers can implement these parameterized blocks of intellectual property (IP) easily, reducing design and test time.

The OpenCore Plus evaluation feature allows the user to download and evaluate Altera® MegaCore functions in hardware and simulation prior to licensing.

You can add a wide variety of Altera DSP MegaCore functions to your Simulink model. In Simulink, these MegaCore functions are represented by blocks in the **Altera DSP Builder Blockset** in the Simulink Library Browser.

MegaCore Function Libraries

There are two separate libraries that can contain MegaCore functions in DSP Builder:

The MegaCore Functions library contains:

- CIC
- FFT
- FIR Compiler
- NCO
- Reed Solomon Compiler
- Viterbi Compiler

These MegaCore function variations must be generated after you add one of these blocks to your model. The "MegaCore Function Walkthrough" on page 4–9 illustrates the design flow using these MegaCore functions.

The Video and Image Processing library contains:

- Alpha Blending Mixer
- Chroma Resampler
- Color Space Converter (CSC)
- Deinterlacer
- 2D FIR Filter
- 2D Median Filter
- Gamma Corrector
- Scaler

These MegaCore functions do not need to be generated before you can connect the blocks to your design.

Any required VHDL or simulation files are created only when they are required by your design flow. The "Fast Functional Simulation Walkthrough" on page 11–2 illustrates simulation using one of these MegaCore functions.

Installing MegaCore Functions

Altera DSP MegaCore functions can be installed with the Quartus® II software or can be downloaded individually from the Altera website. See the user guide for the MegaCore function or functions for instructions on how to install the MegaCore functions on your system.

It is important to run the DSP Builder MegaCore function setup script after the installation of new MegaCore functions. This will update DSP Builder for all newly installed or upgraded MegaCore functions.

To run this setup script, follow these steps:

1. Start the MATLAB/Simulink software.
2. Use the `cd` command at the MATLAB prompt to change directory to the directory where DSP Builder was installed.
3. Run the setup script by typing the following at the MATLAB prompt:

```
alt_dspbuilder_setup_megacore ↵
```

Running this script, creates a **MegaCore Functions** subfolder below the **Altera DSP Builder Blockset** in the Simulink Library Browser.

Within this folder, there should be two or more blocks representing each of the installed MegaCore functions:

- One or more blue blocks with the name of the current version of the MegaCore function they represent. It is recommended that these blocks should be used in new designs.
- Dimmed blocks which represent older versions of the MegaCore functions. These are supported for backwards compatibility.

If you have installed any of the MegaCore functions in the Altera Video and Image Processing Suite, a separate **Video and Image Processing** folder is created which contains these MegaCore functions. All blocks in this library are versioned.

Updating MegaCore Function Variation Blocks

Although a DSP Builder design using MegaCore function blocks from the MegaCore Functions library can be translated by `Signal Compiler` into a VHDL or Verilog HDL model, a MegaCore function variation block always uses an intermediate VHDL file to record parameters.

These blocks may revert to their unconfigured appearance if the VHDL file which details the function variation is available but the (**.simdb**) file is not. A block may also require updating if you have changed the version of the MegaCore function you are using.

In these cases, you can update the MegaCore function variation blocks in your design using the **alt_dspbuilder_refresh_megacore** script.

This will recreate the simulation files based on the VHDL file for each MegaCore function block in the current Simulink model.

A Quartus II license must be available on the machine for the script to execute without errors.

To run the script, perform the following steps:

1. Start the MATLAB/Simulink software.
2. If necessary, use the `cd` command at the MATLAB prompt to change directory to your project directory.
3. Run the script by typing the following at the MATLAB prompt:

```
alt_dspbuilder_refresh_megacore ↵
```

This procedure is not required for the MegaCore functions in the Video and Image Processing library.

Design Flow Using MegaCore Functions

Using MegaCore functions in the MATLAB/Simulink environment is a five-step process.

1. Add the MegaCore function to the Simulink model and give the block a unique name.
2. Parameterize the MegaCore function variation.
3. Generate the MegaCore function variation.

This step is not required for the MegaCore functions in the Video and Image Processing library.

4. Connect your MegaCore function variation to the other blocks in your model.

5. Simulate the MegaCore function variation in your model.

Refer to the appropriate user guide for information about the design flow used for each MegaCore function.

Place the MegaCore Function in the Simulink Model

You can add a MegaCore function to a Simulink model by dragging a copy of the block from the Simulink Library Browser to the design workspace like any other Simulink block.

The default name of a MegaCore function block includes its version number. If you add more than one copy of a block in the same model, this number is automatically incremented to make the name unique. (The correct version number is still shown on the body of the block.) However, you are recommended to rename all blocks representing MegaCore functions with a name describing their use in your design. This will ensure that all generated entities for the same MegaCore function in a hierarchical design have unique names.

After adding the block and before parameterization, save the model file.

Parameterize the MegaCore Function Variation

Double-click the MegaCore function block to open the IP Toolbench or MegaWizard interface.

You can also double-click on a block to re-open and modify a previously parameterized MegaCore function variation.

If you are using one of the MegaCore functions in the Video and Image Processing library, click **Finish** in the MegaWizard interface to update your block with the required input and output ports.

Generate the MegaCore Function Variation

If you are using one of the MegaCore functions in MegaCore function library, you must generate a MegaCore function variation after you have parameterized the MegaCore function before you can connect the block to your design.

Click **Generate** in IP Toolbench (or **Finish** in the MegaWizard interface) to generate the necessary files for your MegaCore function variation.

DSP Builder also performs an additional step of optimizing the model for use in Simulink. This process may take up to a few minutes to complete depending on the complexity of the MegaCore function variation.

Connect Your MegaCore Function Variation Block to Your Design

The Simulink block now has the required input and output ports as parameterized in IP Toolbench or the MegaWizard interface. These ports can be connected to other Altera DSP Builder blocks in your Simulink design.

Simulate the MegaCore Function Variation in Your Model

The Simulink block representing the MegaCore function variation can be simulated like any other block from the Simulink Library Browser.

Ensure that the Simulink simulation engine is set to use the discrete solver by choosing **fixed-step** for Type under Solver Options in the **Configuration Parameters** dialog box.

You should reset the MegaCore function at the start of the simulation to avoid any functional discrepancy between RTL simulation and Simulink simulation, as described in "Startup & Initial Conditions" on page 3–11.

Design Issues When Using MegaCore Functions

This section describes some of the design issues that must be considered when using MegaCore functions in a DSP Builder design.

Simulink Files Associated with a MegaCore Function

The files necessary to support the configuration and simulation of a MegaCore function variation generated from the MegaCore Functions library are stored in a subdirectory of the directory containing your Simulink MDL file named **DSPBuilder_**<*design name*>**_import**. When copying a design from one location to another, make sure that you also copy this subdirectory.

Two specific files are needed to simulate a MegaCore function variation. If your MegaCore function variation is named `my_function`, and it is generated in VHDL, there is a **my_function.vhd** file that defines the variation in the design directory. If the design is named `my_design`, there is a file **DSPBuilder_my_design_import/my_function.vo.simdb** that contains the simulation information.

These files do not exist for a MegaCore function variation generated from the Video and Image processing library.

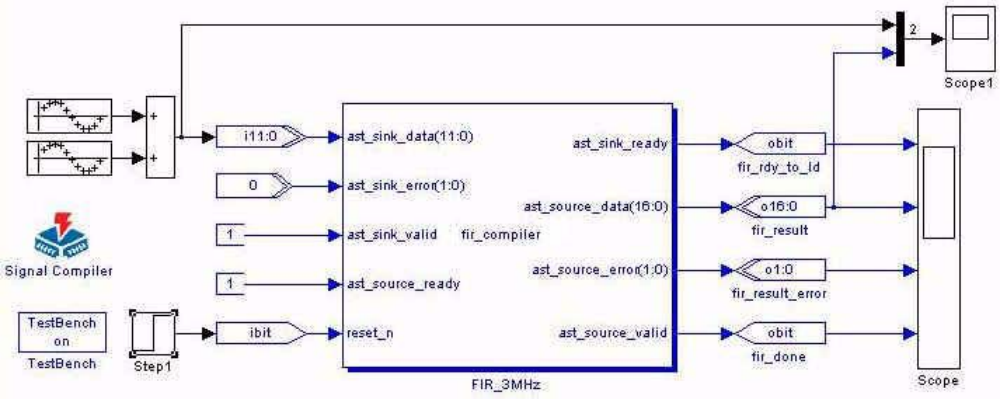
Simulating MegaCore Functions That Have a Reset Port

MegaCores functions that have a reset port need to have a reset cycle in Simulink simulation at the start in order to produce correct simulation results. This reset cycle must be of sufficient length, depending on the particular MegaCore function and parameterization.

For example in Figure 4–1, the reset cannot be tied to a constant because the simulation would not match hardware. You have to simulate an initial reset cycle (with the step input) to replicate hardware behavior. As in hardware, this reset cycle has to be sufficiently long to propagate through the core, which may be 50 clock cycles or more for some MegaCore functions such as the FIR Compiler.

Additional adjustment of the reset cycles may be necessary when a MegaCore function receives data from other MegaCore functions, to ensure that the blocks leave the reset state in the correct order and are delayed by the appropriate number of cycles.

Figure 4–1. MegaCore Function Design With a Reset Port



Using Feedback Between MegaCore Functions

If you connect a feedback loop between two MegaCore function variation blocks, you should set a higher value priority on the sink block than on the source block. This will ensure that the sink block is not executed before the source block has created valid data for it.

Lower values of the priority setting ensure that a block is invoked before blocks with higher settings.

You can set the priority for a block by choosing **Block Properties** from the popup menu and setting a Priority value as shown in Figure 4–2.

Figure 4–2. Setting the Priority for a Block

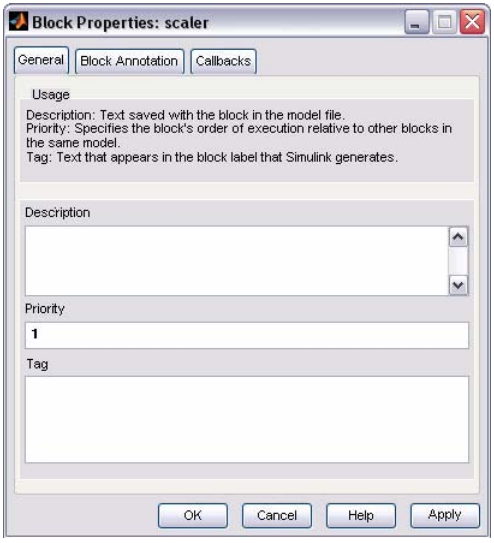
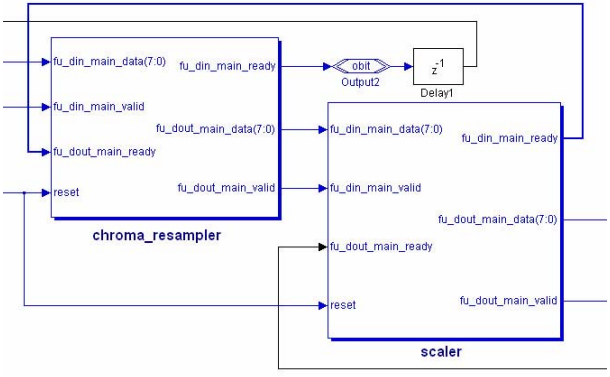


Figure 4–3 shows an example design with a feedback loop from the `scaler` block to an input on the `chroma_resampler` block. Setting a higher value priority on the `scaler` block will ensure that it is executed after the `chroma_resampler` block.

Figure 4–3. Example Design with Feedback between MegaCore Functions



Setting the Device Family for MegaCore Functions

There are a number of MegaCore functions available in DSP Builder.

Some of these use the IP Toolbench interface (for example, the FIR Compiler MegaCore function used in the walkthrough later in this chapter).

More recently introduced MegaCore functions (such as the Video and Image Processing Suite used in Chapter 11, Using the Simulation Accelerator) use a simplified MegaWizard user interface.

The newer MegaCore functions always inherit the device family setting from the `Signal Compiler` block. If there is no `Signal Compiler` block in your design, the `Stratix` device family is chosen by default. The following MegaCore functions have this behavior:

- CIC
- Alpha Blending Mixer
- Chroma Resampler
- Color Space Converter (CSC)
- Deinterlacer
- 2D FIR Filter
- 2D Median Filter
- Gamma Corrector
- Scaler

Older MegaCore functions (using the IP Toolbench user interface) allow you to modify the device family setting in the IP Toolbench interface.

The following MegaCore functions have this behavior:

- FFT
- FIR Compiler
- NCO
- Reed Solomon Compiler
- Viterbi Compiler

If you change the device family in `Signal Compiler`, you must check that any IP Toolbench MegaCore Functions have the correct device family set to ensure that the simulation models and generated hardware are consistent.

You can then run **alt_dspbuilder_refresh_megacore** as described in "Updating MegaCore Function Variation Blocks" on page 4–3 to ensure that all the MegaCore functions are up-to-date and consistent.

MegaCore Function Walkthrough

This walkthrough shows how to create a custom low-pass FIR filter MegaCore function variation using the IP Toolbench interface. For an example of a MegaCore function that uses the MegaWizard interface, refer to "Using the Simulation Accelerator" on page 11–1.

This walkthrough assumes that the Altera MegaCore IP Library is installed.

Create a New Simulink Model

Create a new Simulink workspace by performing the following steps:

1. Start the MATLAB/Simulink software.
2. Choose the **New > Model** command (File menu) to create a new model file.
3. Choose **Save** (File menu) in the new model window.
4. Browse to the directory in which you want to save the file. This directory becomes your working directory. This walkthrough creates and uses the working directory **<***DSP Builder install path***>\DesignExamples\Tutorials\MegaCore**
5. Type the file name into the **File name** box. This walkthrough uses the name **mc_example.mdl**.
6. Click **Save**.

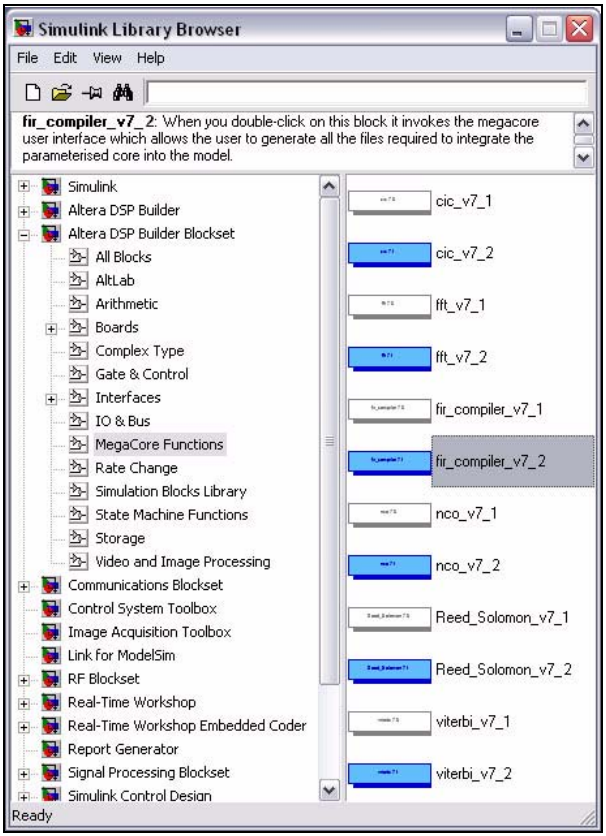
Add the FIR Compiler Function to Your Model

To place a FIR Compiler MegaCore function block in your design, perform the following steps:

1. In your Simulink model window, select **Library Browser** (View menu). The Simulink Library Browser is displayed.
2. Select the MegaCore Functions library from the **Altera DSP Builder Blockset** folder in the Simulink Library Browser (Figure 4–4 on page 4–10).

If the MegaCore function blocks do not appear as shown in Figure 4–4, make sure that the MegaCore IP library and the FIR Compiler MegaCore function is installed and you have run the `alt_dspbuilder_setup_megacore` script. For instructions on running this script, see "Installing MegaCore Functions" on page 4–2.

Figure 4–4. MegaCore Functions Library



3. Drag and drop a blue versioned `fir_compiler_v7.2` block into your model as shown in Figure 4–5.

Figure 4–5. FIR Compiler Block Placed in Simulink Model

The older versions of MegaCore function blocks (shown dimmed in the Simulink library browser) are provided for back compatibility and should not be used in a new design.

The block is added with a default instance name which includes the version string. This name will be automatically made unique if you add more than one instance of the same block. However, you may want to change the name to be more meaningful within your design.

4. For this tutorial, rename the block to `my_fir_compiler`. To rename the block, click the default name (the text outside of the block itself) and edit the text. Naming conventions are described in "DSP Builder Naming Conventions" on page 3–1.

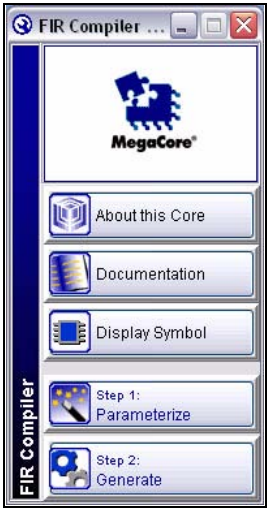
Always give blocks representing your MegaCore function variations unique names, to avoid issues caused by two or more entities in a hierarchical design.

Parameterize the FIR Compiler Function

To use FIR Compiler to create a MegaCore function variation that fits the specific needs of your design, perform the following steps:

1. Double-click the `my_fir_compiler` block to start IP Toolbench (Figure 4–6).

Figure 4–6. IP Toolbench-Parameterize

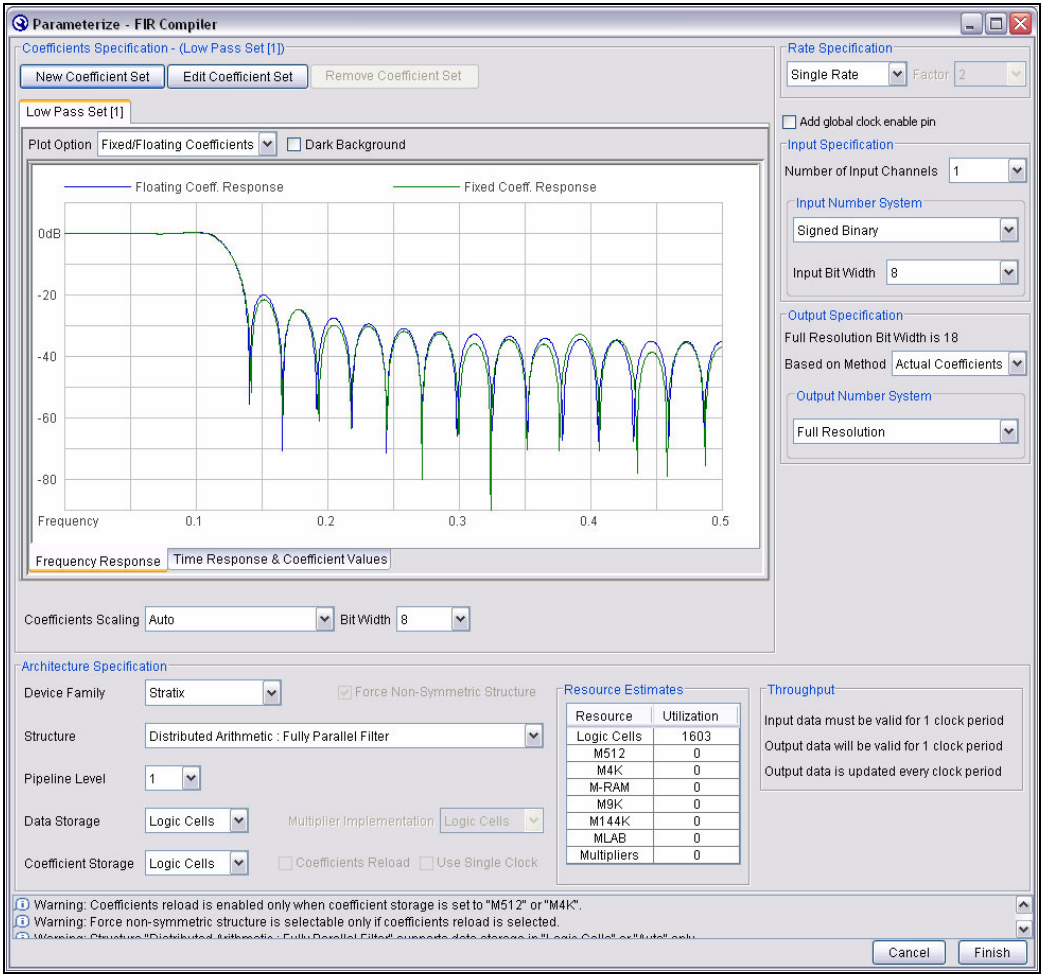


2. Click **Step 1: Parameterize** to specify how the FIR filter should operate.

The **Parameterize - FIR Compiler MegaCore function** dialog box is displayed (Figure 4–7 on page 4–12).

3. For this walkthrough, use the default values, specifying a low-pass filter. Click **Finish**.

Figure 4–7. Parameterize - FIR Compiler MegaCore Function Dialog Box



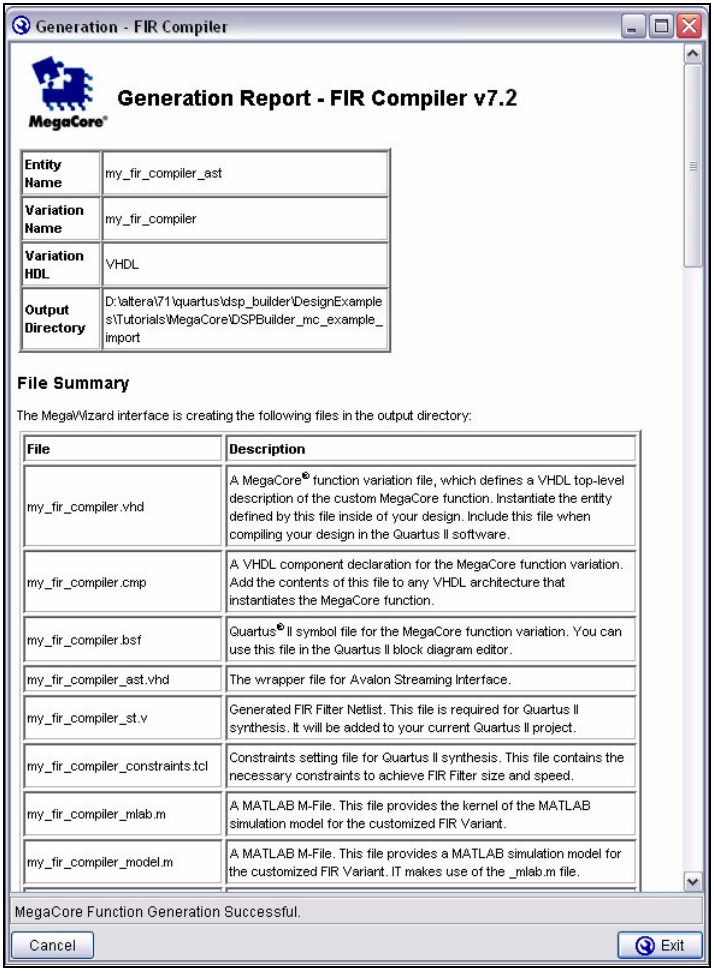
Generate the FIR Compiler Function Variation

After you parameterize the MegaCore function, to generate the files required for inclusion in the Simulink model and simulation, perform the following steps:

1. Click **Step 2: Generate** in IP Toolbench (Figure 4–6 on page 4–11).

2. The generation report lists the design files that IP Toolbench creates (Figure 4–8).

Figure 4–8. Generation Report



For more information about the FIR Compiler including a complete description of the generated files, refer to the *FIR Compiler User Guide*.

3. Click **Exit**.

The `my_fir_compiler` block in the Simulink model is updated to show the input and output ports for the given configuration (Figure 4–9).

The FIR filter is ready to be connected to the rest of your Simulink design.

Figure 4–9. FIR Compiler Block in Simulink Model After Generation

Add Stimulus and Scope Blocks to Your Model

In this section, you create a sample design to test the low-pass filter by feeding the filter two sine waves as shown in Figure 4–10 on page 4–16.

For information on how to add blocks to a design and modify their parameters, see Chapter 2, Getting Started Tutorial.

Perform the following steps:

1. Add two `Sine Wave` blocks (from the Simulink Sources library) and connect their outputs to an `Add` block (from the Simulink Math Operations library).

2. Double-click on the first `Sine Wave` block to display the Block Parameters dialog box and set the following parameters:

   - Choose a `Sample based` sine type
   - Specify the Amplitude to be `64`
   - Specify the Samples per period to be `200`
   - Specify the **Sample time** to be `1`

3. Double-click on the second `Sine Wave` block and use the Block Parameters dialog box to set the following parameters:
   - Choose a `Sample based` sine type
   - Specify the Amplitude to be `64`
   - Specify the Samples per period to be `7`
   - Specify the **Sample time** to be `1`
4. Add an `Input` block (from the IO & Bus library in the **Altera DSP Builder Blockset**) and connect it between the `Add` block and the `ast_sink_data` pin on the `my_fir_compiler` block.
5. Add a `Constant` block (from the IO & Bus library) and connect this block to both the `ast_sink_valid` and `ast_source_ready` pins on the `my_fir_compiler` block.
6. Double-click on the `Constant` block and use the Block Parameters dialog box to set the following parameters:
   - Specify the constant value to be `1`
   - Choose a `Single Bit` bus type
7. Add another `Constant` block (from the IO & Bus library) and connect this block to the `ast_sink_error` pin on the `my_fir_compiler` block.
8. Double-click on the second `Constant` block and use the Block Parameters dialog box to set the following parameters:
   - Specify the constant value to be `0`
   - Choose a `Signed Integer` bus type
   - Specify the number of bits to be `2`
9. Add a `Single Pulse` block (from the Gate & Control library in the **Altera DSP Builder Blockset**) and connect this block to the `reset_n` pin on the `my_fir_compiler` block.
10. Double-click on the `Single Pulse` block and use the Block Parameters dialog box to set the following parameters:
    - Choose `Step Up` for the signal generation type
    - Specify the **Delay** to be `50`
11. Add an `Output` block (from the IO & Bus library in the **Altera DSP Builder Blockset**) to the design and connect it to the `ast_source_data` pin on the `my_fir_compiler` block.

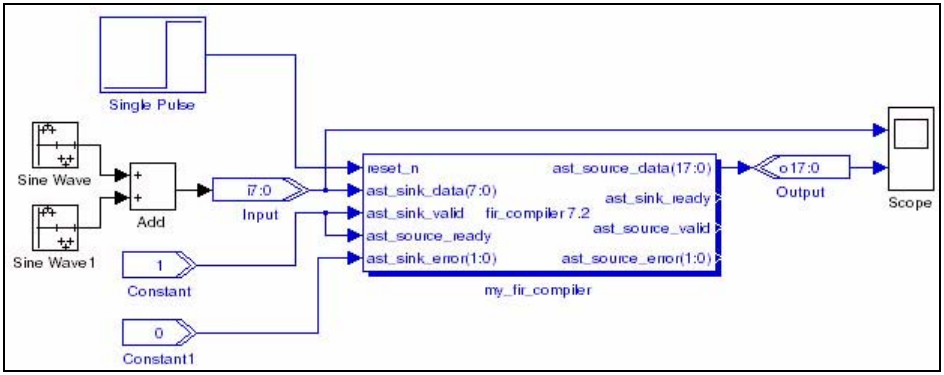
12. Double-click on the `Output` block and use the Block Parameters dialog box to change the number of bits to `18`.

13. Add a `Scope` block (from the Simulink Sinks library). Use the Scope Parameters dialog box to configure the `Scope` block as a 2-input scope.

14. Connect the `Scope` block to the `Input` and `Output` blocks to monitor the source noise data as well as the filtered output.

    Your model should look similar to that shown in Figure 4–10.

Figure 4–10. Connecting Blocks to the Low-Pass Filter



Simulate Your Design in Simulink

To simulate the design, perform the following steps:

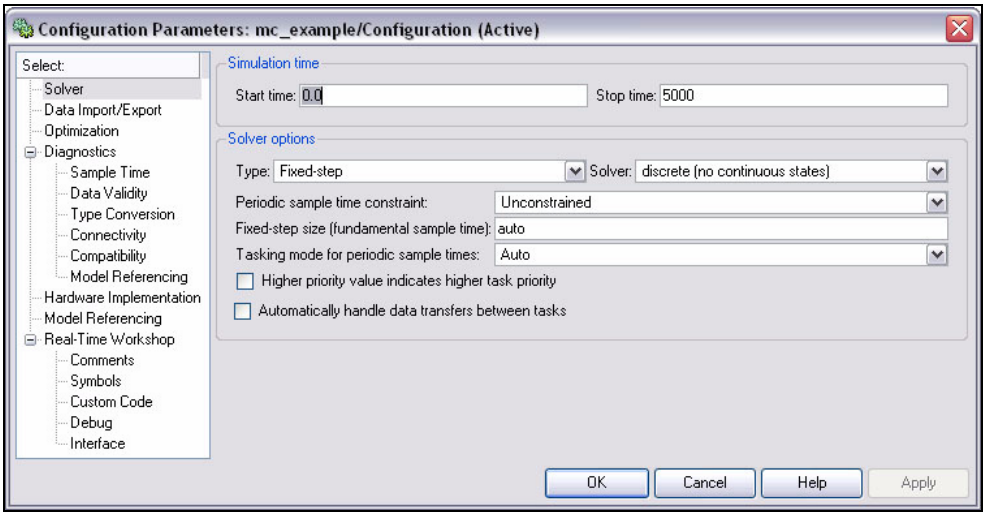
1. In your model, choose **Configuration Parameters** (Simulation menu) to display the **Configuration** dialog box (Figure 4–11 on page 4–17).

2. Specify the following options in the **Configuration** dialog box:

   - Under Simulation Time, for **Stop Time**, specify `5000`.
   - Under Solver Options, in the **Type** list choose `Fixed-step`.
   - In the Solver list, choose `discrete (no continuous states)`.

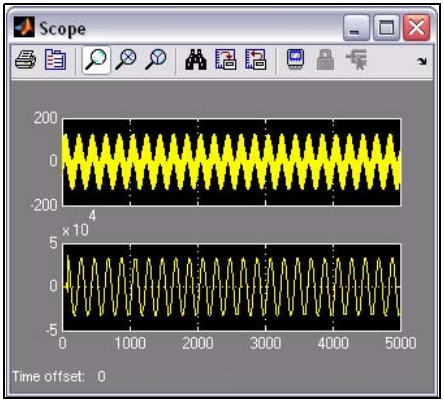
3. Click **OK**.

Figure 4–11. Configuration Parameters: mc_example/Configuration Dialog Box



4. In the simulink model, choose **Start** (Simulation menu). The scope output shows the effect of the low-pass filter in the bottom window, as shown in Figure 4–12. Check that the FIR filter block behaves as expected, filtering high-frequency data as a low-pass filter.

Figure 4–12. Simulation Output



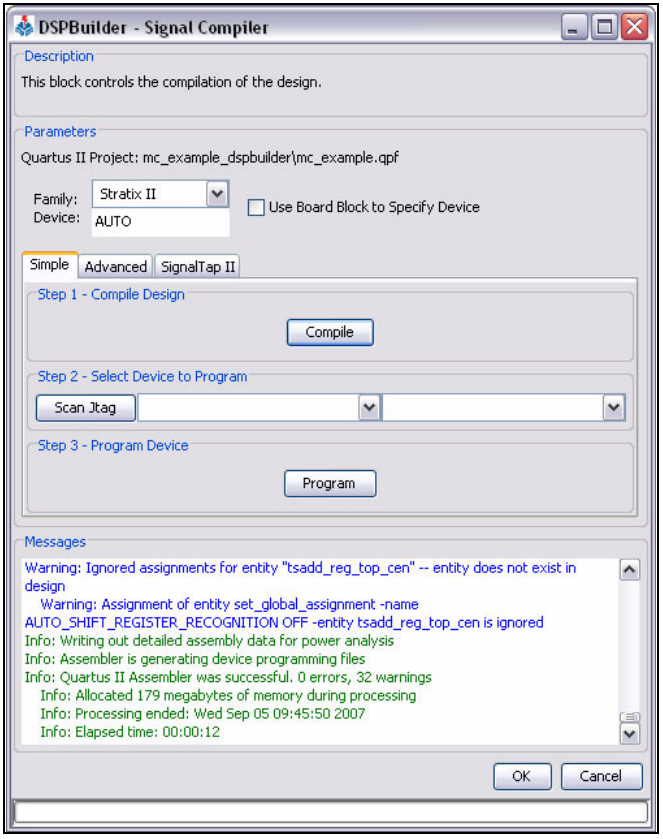
You may need to use the **Autoscale** command in the Scope display to view the complete waveforms.

Compile the Design

To create and compile a Quartus II project for your DSP Builder design, and to program the design onto an Altera FPGA, you need to add the `Signal Compiler` block. Perform the following steps:

1. Select the AltLab library from the Altera DSP Builder Blockset folder in the Simulink Library Browser.
2. Drag and drop a `Signal Compiler` block into your model.
3. Double-click the new `Signal Compiler` block in your model. The Signal Compiler dialog box appears (Figure 4–13).
4. Click **Compile**.

Figure 4–13. Signal Compiler Dialog Box



5. When the compilation has completed successfully, click **OK**.

Perform RTL Simulation

To perform RTL simulation with the ModelSim software, you need to add a `TestBench` block.

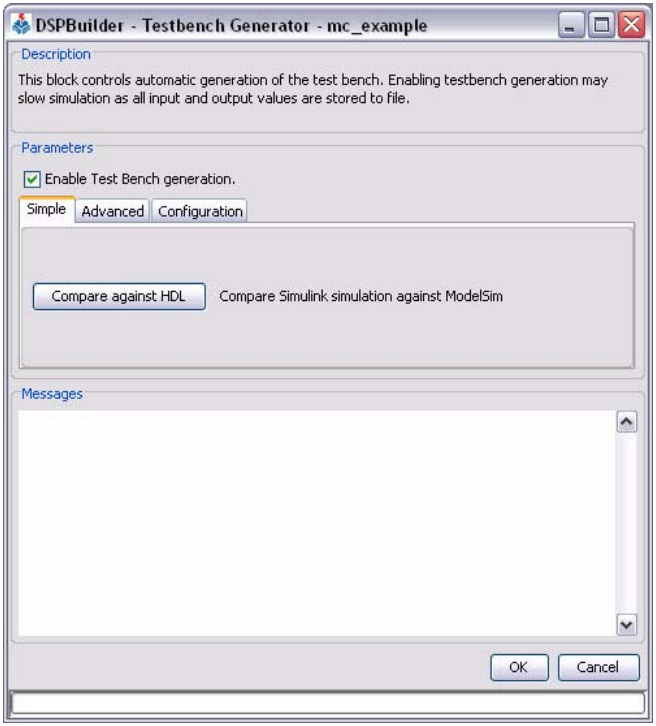
Perform the following steps:

1. Select the AltLab library from the **Altera DSP Builder BlockSet** folder in the Simulink Library Browser.

2. Drag and drop a `TestBench` block into your model.

3. Double-click on the new `TestBench` block. The TestBench Generator dialog box appears (Figure 4–14).

Figure 4–14. TestBench Generator Dialog Box



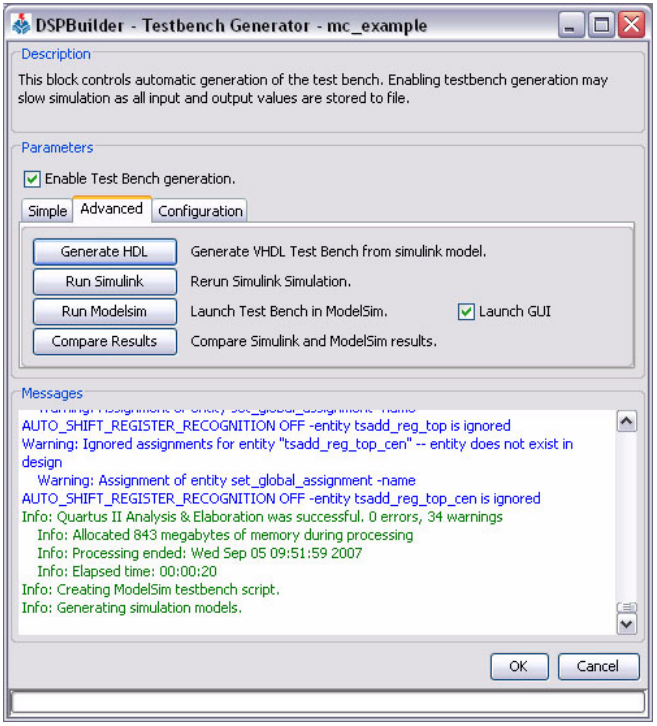
4. Ensure that **Enable Test Bench generation** is turned on.

5. Select the **Advanced** Tab (Figure 4–15).

6. Turn on the **Launch GUI** option. This will cause the ModelSim GUI to be launched if ModelSim simulation is invoked.

7. Click **Generate HDL** to generate a VDHL-based Testbench from your model.

Figure 4–15. TestBench Generator Dialog Box Advanced Tab



8. Click **Run Simulink** to generate Simulink simulation results for the testbench.

9. Click **Run ModelSim** to simulate the design in ModelSim.

   The design is loaded into ModelSim and simulated with the output displayed in the Wave window.

All waveforms are initially shown using digital format in the ModelSim Wave window.

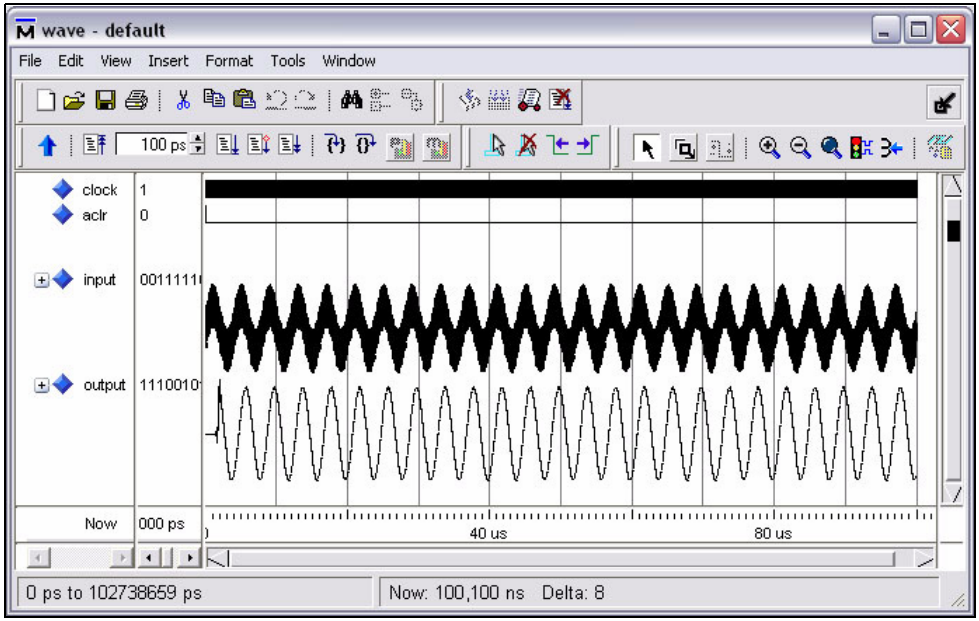
10. Select the `input` signal in the ModelSim Wave window and choose **Properties** from the popup menu to display the Wave Properties dialog box. Click the **Format** tab and change the format to Analog with height `75` and scale `0.25`.

11. Select the `output` signal in the ModelSim Wave window and use the Wave Properties dialog box to change the format to Analog with height `75` and scale `0.001`.

12. Choose **Zoom Full** from the popup menu in the ModelSim Wave window.

The ModelSim simulator now displays the input and output waveforms in analog format as shown in Figure 4–16.

Figure 4–16. Generated HDL for mc_example Simulated in ModelSim Simulator *Note (1)*



Note to Figure 4–16:
(1) The waveform display shown here has been formatted to show the input and output signals as analog waveforms.

13. Click on **Compare Results** to compare the simulink results with those generated by ModelSim. The message `Exact Match` should be issued indicating that the results are identical.

14. Click OK to close the TestBench dialog box when you have finished.

5. Using Hardware in the Loop (HIL)

Introduction

Adding the Hardware in the Loop (HIL) block to your Simulink model allows you to co-simulate a Quartus® II software design with a physical FPGA board implementing a portion of that design.

You define the contents and function of the FPGA by creating and compiling a Quartus II project. A simple JTAG interface between Simulink and the FPGA board links the two.

The main benefits of using the HIL block are faster simulation and richer instrumentation. The Quartus II project you embed in an FPGA will run faster than a software-only simulation. To further increase simulation speed, the HIL block offers frame and burst modes of data transfer that are significantly faster than single-step mode when used with suitable designs.

The HIL block also makes available to the hardware a large Simulink library of sinks and sources, such as channel models and spectrum analyzers, which can give you greater control and observability.

This chapter explains the HIL design flow, walks through an example using the HIL block, and discusses the optional burst and frame data transfer modes.

HIL Design Flow

The HIL block in AltLab library of the **Altera DSP Builder Blockset** enables the Hardware in the Loop functionality. It represents the functions implemented on your FPGA, and works smoothly with the normal DSP Builder/Simulink work flow.

Follow these steps to use the HIL block:

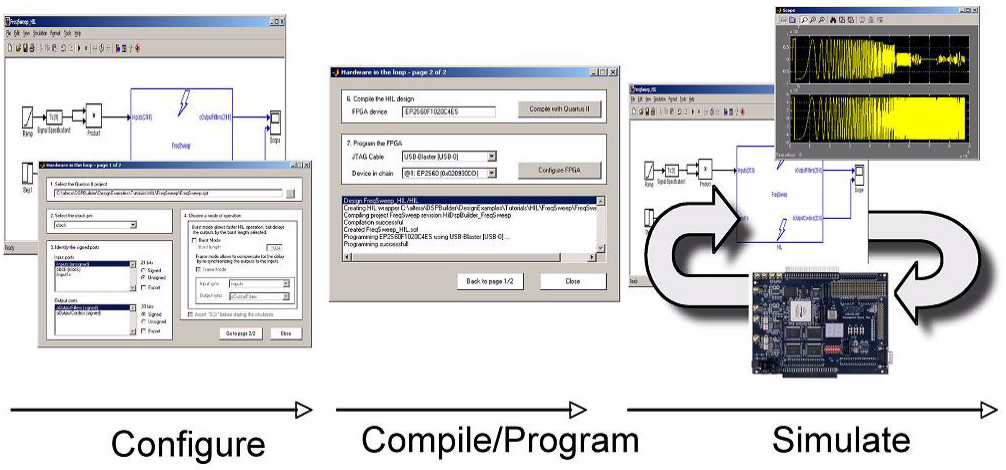
1. Create a Quartus II project that defines the functions you want to co-simulate in hardware and compile the Quartus II project through the Quartus II Fitter step.

2. Add the HIL block to your Simulink model and import the compiled Quartus II project into the HIL block. You might also connect instrumentation to your HIL block by adding additional blocks from the Simulink Sinks and Sources libraries.

3. Specify the various parameters of the `HIL` block, including:
   - the Quartus II project you compiled to define its functionality,
   - the input and output pin characteristics, and
   - the use of single-step versus burst and frame mode.
4. Compile the `HIL` block to create a programming object file that will be used for hardware co-simulation.
5. Program the board that contains your target FPGA.
6. Simulate the combined software and hardware system in Simulink. When using a `HIL` block in a Simulink model, set the simulation parameters for the model as follows:
   - **Solver option**: `Fixed-step`
   - **Mode**: `Single-tasking`

Figure 5–1 shows this system-level design flow using DSP Builder.

Figure 5–1. System-Level Design Flow



HIL Requirements

You need the following to use the `HIL` block:

- An FPGA board with a JTAG interface (Stratix® II, Stratix, Cyclone® II, or Cyclone device)
- A valid Quartus II project that contains a single clock domain driven from Simulink. (An internal Quartus II project is created when you run `Signal Compiler`.)
- A JTAG download cable (for example, a ByteBlasterMV™, ByteBlaster™ II, ByteBlaster, MasterBlaster™, or USB-Blaster™ cable)
- A maximum of one `HIL` block for each JTAG download cable

HIL Walkthrough

DSP Builder includes several design examples in the <*DSP Builder install path*>**\DesignExamples\Tutorials\HIL** directory that demonstrate the use and effectiveness of HIL:

- Imaging Edge Detection
- Export Example
- Fast Fourier Transform (FFT)
- Frequency Sweep

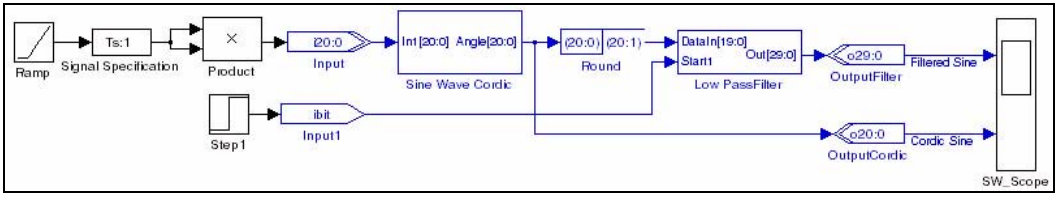
This section walks through the Frequency Sweep design. It assumes you have completed Chapter 2, Getting Started Tutorial, and are familiar with using DSP Builder, MATLAB, Simulink, and the Quartus II software.

You will also need an FPGA board connected to your computer with a JTAG download cable.

This walkthrough uses a Quartus II project which is created using DSP Builder and a Stratix II device on the Altera® Stratix II EP2S60 DSP Development Board. However, you could also use a Quartus II project created within the Quartus II software with any other supported device and board.

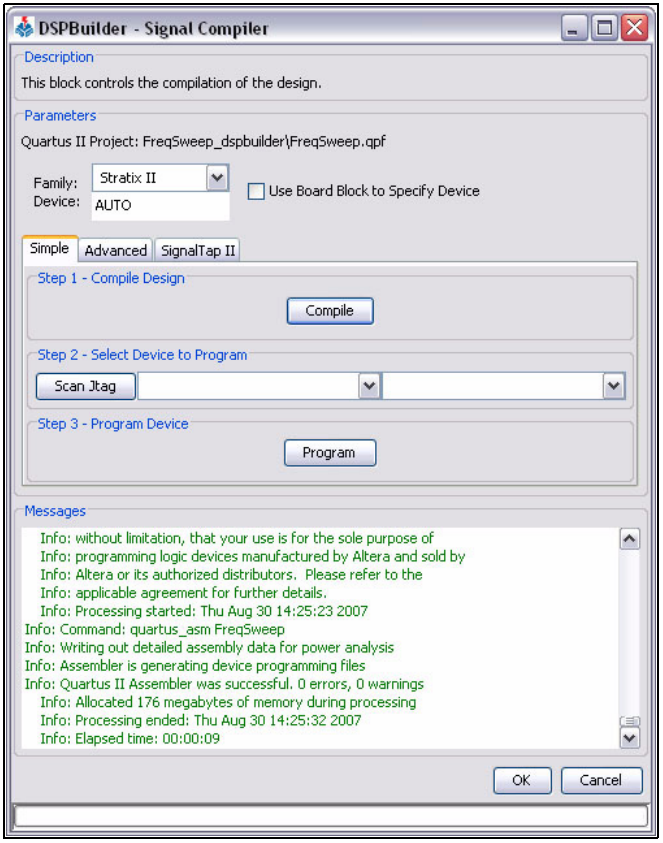
1. Run MATLAB, and open the model **FreqSweep.mdl** in the <*DSP Builder install path*>**\DesignExamples\Tutorials\HIL\FreqSweep** directory. Figure 5–2 on page 5–3 shows the loaded model.

Figure 5–2. Frequency Sweep Model



2. Double-click the `Signal Compiler` block. In the dialog box that appears (Figure 5–3), click **Compile**.

Figure 5–3. Signal Compiler Dialog Box, Simple Tab



This action creates a Quartus II project, **FreqSweep.qpf**, compiles the model for synthesis and runs the Quartus II fitter.

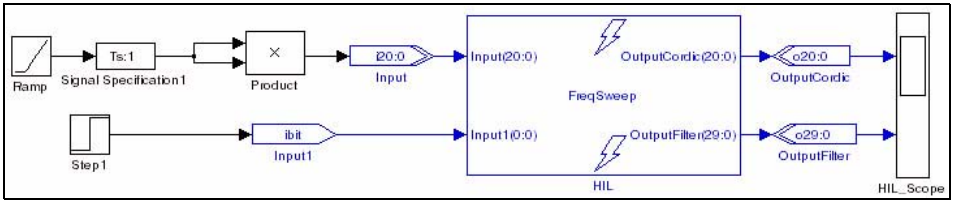
Progress is indicated by status messages and a scrolling bar at the bottom of the dialog box.

3. Review the Messages, then click **OK** to close `Signal Compiler`.

4. Replace the internal functions of the Frequency Sweep model with an `HIL` block. For this walkthrough, do this by opening the prepared model **FreqSweep_HIL.mdl** from the same directory you used earlier.

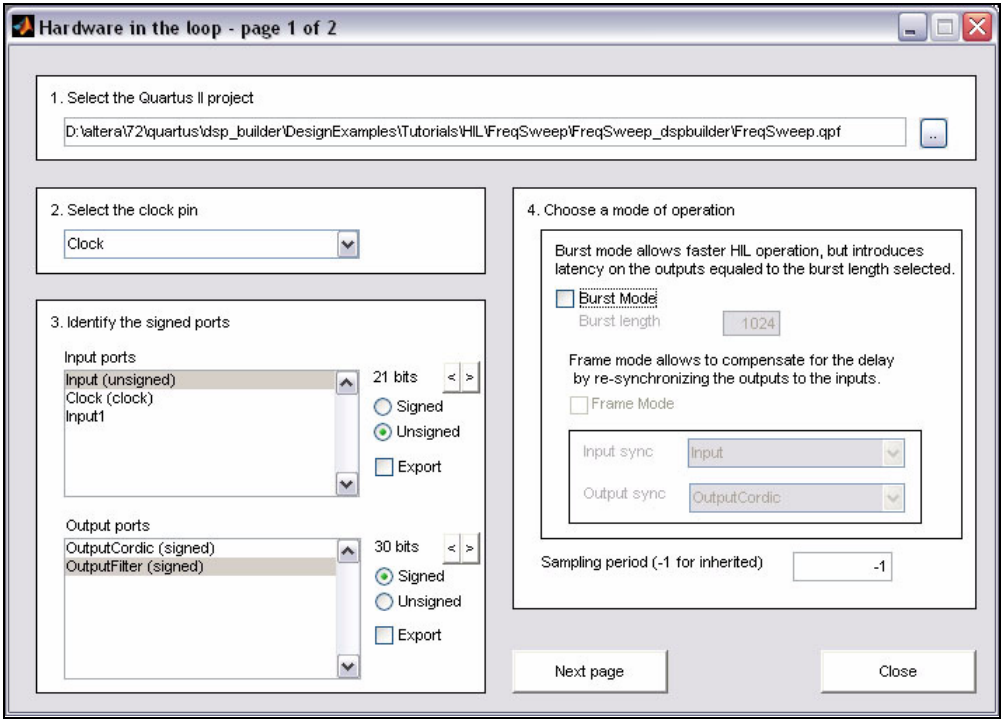
Figure 5–4 shows this model, with the HIL block in place.

Figure 5–4. Frequency Sweep Design Model Using the HIL Block



5. Double-click the Frequency Sweep HIL block to display the Hardware in the loop dialog box (Figure 5–5).

Figure 5–5. Setting HIL Block Parameters, page 1 of 2



a. Select the Quartus II project **FreqSweep.qpf** by browsing into the **FreqSweep_dspbuilder** directory to locate the file.

The full path to this file should be visible in the dialog box when this file is selected.

b. Identify the signed ports by selecting each port and using the **Signed**/**Unsigned** buttons:

- `Input`: `unsigned`
- `OutputCordic`: `signed`
- `OutputFilter`: `signed`.

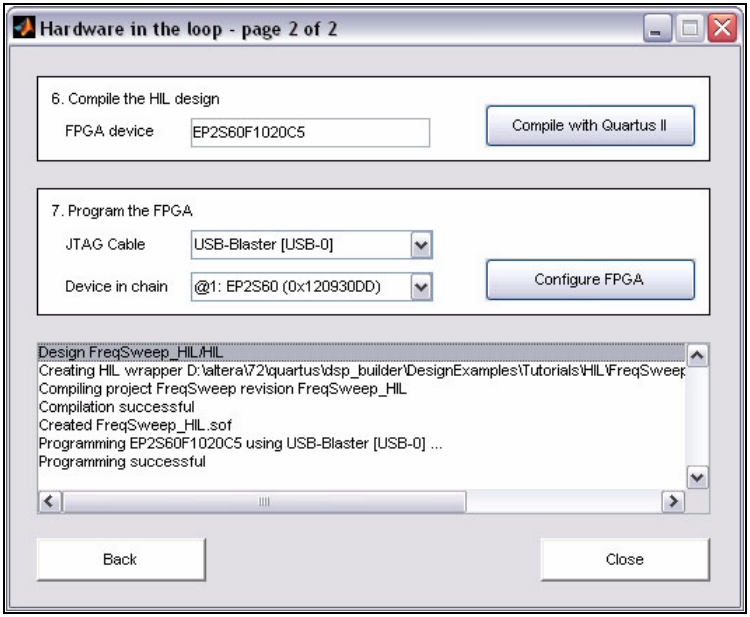
c. Choose the mode of operation by turning off **Burst Mode**.

6. Click **Next page**. to display the second page of the dialog box (Figure 5–6).

a. Select a device and click **Compile with Quartus II** to compile the HIL design.

b. Specify a JTAG download cable, and click **Configure FPGA** to program the FPGA on the board.

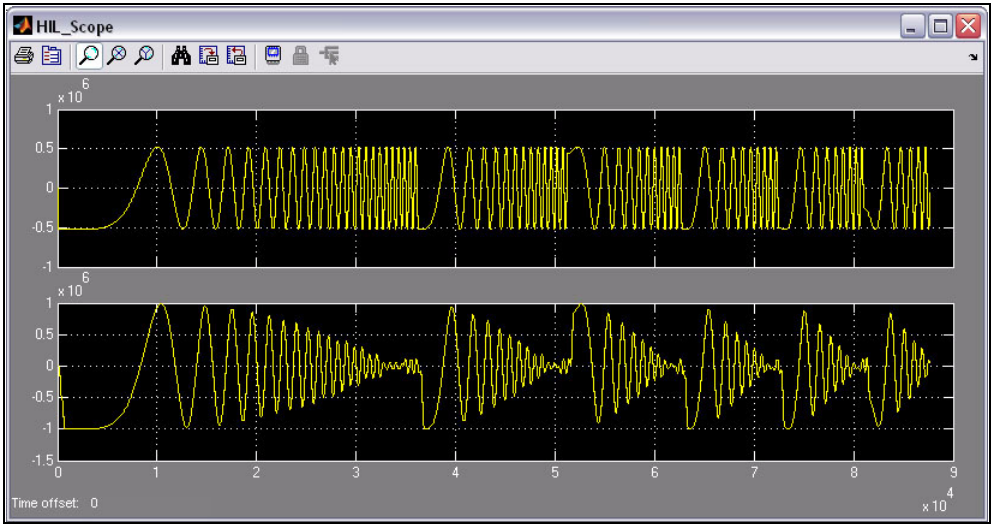
Figure 5–6. Setting HIL Block Parameters, page 2 of 2



7. Click **Close**.

8. Simulate the design in Simulink. Figure 5–7 shows the scope display from the finished design.

Figure 5–7. Scope Output from the FrequencySweep Model with HIL Block



Burst & Frame Modes

The Quartus II software infrastructure that communicates with the FPGA through JTAG—known as system-level debug (SLD)—uses a serial data transfer protocol. To maximize the throughput of this data transfer, the `HIL` block offers a burst mode that buffers the stimulus data and presents it in bursts to the hardware. Burst mode also allows a frame mode to be used for certain types of designs.

Table 5–1 shows the advantages and disadvantages of using burst mode compared with the normal single-step mode.

Table 5–1. Comparing Single-Step and Burst Modes

	Single-Step Mode	Burst Mode
Advantages	Cycle accurate simulation. Feedback is possible outside of the `HIL` block.	Low SLD overhead. Fast HIL results. Frame mode possible.
Disadvantages	High SLD overhead. No frame mode.	A latency is introduced on the output signals of the `HIL` block making feedback loop difficult outside the FPGA device.

Using Burst Mode

You can activate burst mode by turning on the **Burst Mode** checkbox in the Hardware in the loop block parameters as shown in Figure 5–10 on page 5–10. When this option is set, you can specify the required number of data packets as the **Burst Length**. The `HIL` block will then send data to the hardware in bursts of the size you have specified.

The size of the packet is determined by the larger of the total input data width or the total output data width. If the packet size multiplied by the **Burst Length** exceeds the preset data array, the **Burst Length** is set to 1.

Simulation using burst mode works the same as single clock mode, but a latency of the specific packet size is introduced on the output signals of the `HIL` blocks. As a consequence, feedback-loops may not work properly unless they are enclosed within the `HIL` block and some intervention may be necessary when comparing or visualizing HIL simulation results.

The `HIL` block uses software buffers to send and receive from the hardware, so you can change these buffer sizes without recompiling the HIL function.

Using Frame Mode

You can activate frame mode by turning on the **Frame Mode** checkbox in the Hardware in the loop block parameters as shown in Figure 5–10 on page 5–10. Frame mode builds on the burst functionality and provides a way to partially compensate for the burst mode output delay.

To use frame mode, the following conditions must be true:

- The `HIL` block works with the concept of blocks of data ("frames").
- The data frames are provided at regular intervals.
- There is one Input sync and one Output sync signal available.
- The latency between the Input sync and Output sync signals is constant.

In frame mode, the `HIL` block monitors the Input sync and Output sync signals and increases the output delay to align the output data frames with the input data frames. For example, if the burst length is 1024 and the latency 3, the delay would be 1027 (1024 + 3) without frame mode or 2048 (aligned to the next frame) with frame mode turned on.

The burst packet size in frame mode needs to be a multiple of the frame packet interval. For example, if packets arrive every 100 clocks, you can use a frame burst size of $N \times 100$ clocks (N positive integer).

Figure 5–8 illustrates a DSP Builder design using a FFT MegaCore® function which has been configured with the following parameters:

- Target Device Family: `Stratix II`
- Transform Length: `64 points`
- Data Precision: `16 bits`
- Twiddle Precision `16 bits`.

Figure 5–8. DSP Builder Design Using the FFT MegaCore Function

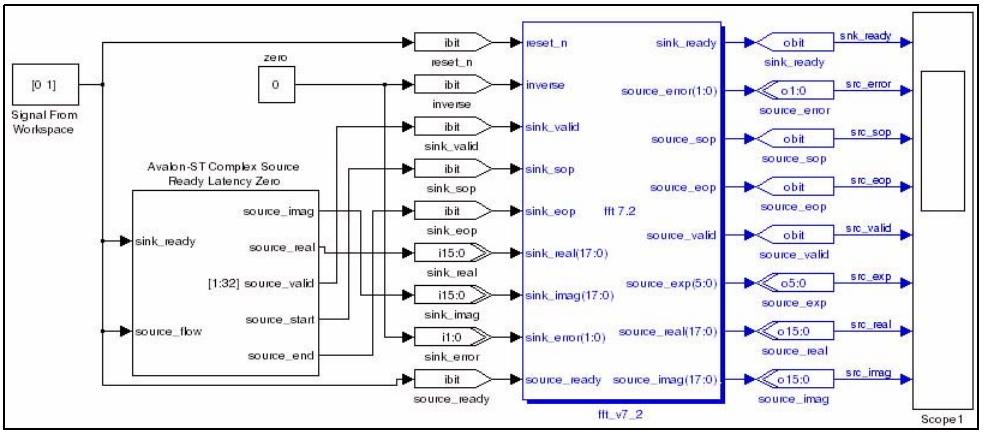
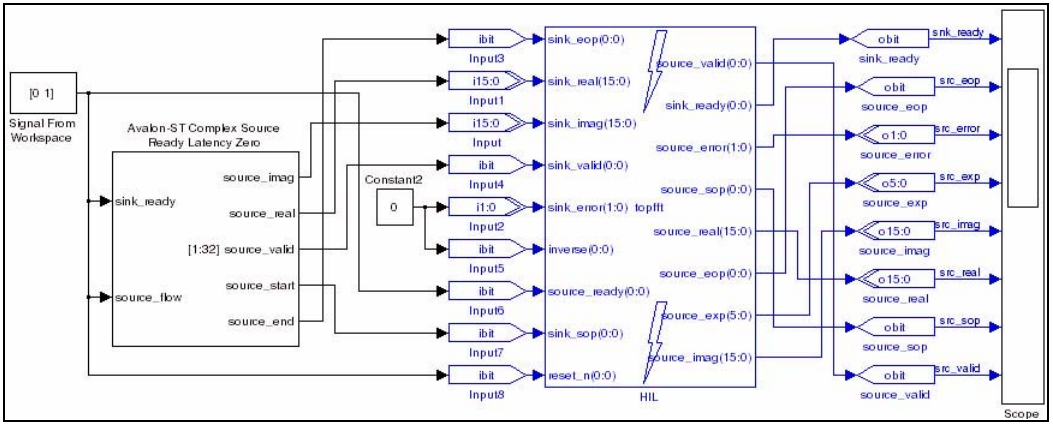


Figure 5–9 shows the FFT design implemented using a `HIL` block (with the parameters shown in Figure 5–10 on page 5–10).

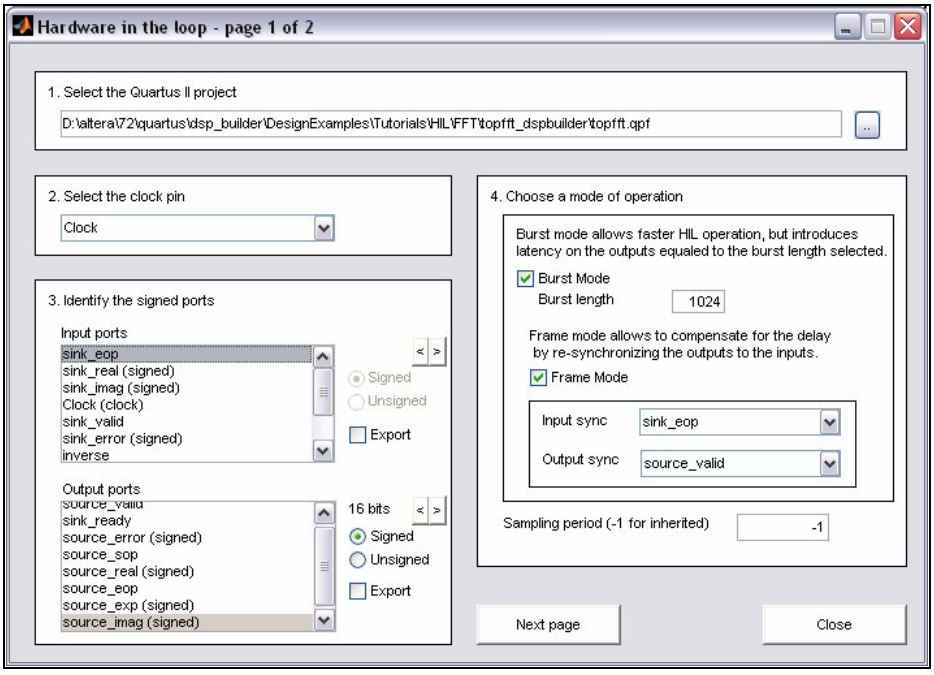
Figure 5–9. Using the FFT Design With an HIL Block



The Avalon® Streaming interface signals `sink_eop` and `source_valid` on the FFT MegaCore function are respectively used in the `HIL` block as the Input sync and Output sync.

Refer to the *FFT MegaCore Function User Guide* for additional information on the input and output port signal timing.

Figure 5–10. Setting Parameters for the HIL Block in Figure 5–9



Troubleshooting HIL Designs

This section describes various issues which may be encountered when you are using HIL designs.

Failed to Load the Specified Quartus II Project

HIL gets the design information, such as clock, reset, and input and output ports, from the specified Quartus II project. However, it could fail to load the project if the project was not compiled through the Quartus II Fitter, there is a Quartus II version mismatch or the Quartus II design project file is not up-to-date.

Project Not Compiled Through the Quartus II Fitter

This occurs when the specified Quartus II project has not been compiled successfully through the Quartus II Fitter.

Action:
Compile the project through Quartus II Fitter before running HIL.

Quartus II Version Mismatch

This occurs when the specified Quartus II project is compiled using a different version of the Quartus II software than the one that is registered.

Action:
Compile the specified project using the registered Quartus II software before running HIL.

Quartus II Design Project File (.qpf) is Not Up-to-Date

This occurs when the specified Quartus II project is older than the design model file. It is possible that the design model has changed or been saved after going through the Quartus II compilation process.

Action:
Recompile the specified the project again before running HIL.

No Clock Found From the Specified Quartus II Project

This could happen if the Quartus II project specified was not compiled successfully through the Quartus II Fitter.

Action:
Ensure that the specified Quartus II project compiles successfully through the Quartus II Fitter.

Multiple Clocks Found From the Quartus II Project

This problem occurs if the DSP Builder model file contains multiple clock domains because HIL does not support multiple clocked designs.

Action:
Only the user selected clock can be used as the HIL clock signal with any other clocks treated as data inputs. HIL simulation of multiple clock designs may differ from Simulink simulation.

No Inputs or Outputs Found From the Quartus II Project

This could occur if the DSP Builder model file contains only the internally induced signals, such as from a counter, and also does not produce any outputs. However, HIL simulation works correctly.

Action:
None required.

Possible Unregistered Paths

This could occur if the DSP Builder model file contains resets associated with clocks but not recognized by the Quartus II Fitter as resets in the Quartus II project. Hence the possibility of clocked but not registered paths inside the design model file

Action:
Check the design model file to make sure that there are no unregistered paths in the design. HIL simulation of designs containing unregistered paths may not match Simulink simulation.

HIL Design Stays in Reset During Simulation

An asynchronous reset is permanently asserted for a Hardware-in-the-Loop (HIL) design.

Action:
You can use the *alt_dspb_hil_reset* workspace variable to specify an active-low (0) or active-high (1) reset that matches the `aclr` settings in the original model.

HIL Compilation Appears to be Hung

After clicking **Compile with Quartus II** in the HIL Block Parameters dialog box, no output is written to the MATLAB command window. This can occur if the original Quartus II project was out-of-date or compiled by a different version of the Quartus II software.

Action:
Recompile the original project using the matching version of the Quartus II software.

6. Performing SignalTap II Logic Analysis

Introduction

This chapter describes how to set up and run the SignalTap® II embedded logic analyzer. In this walkthrough, you analyze three internal nodes in a simple switch controller design named **switch_control.mdl**. The design flow described in this example works for any of the Altera® development boards that DSP Builder supports.

For detailed information on the supported development boards, refer to the *Boards Library* chapter in the *DSP Builder Reference Manual*.

In this design, an LED on the DSP development board turns on or off depending on the state of user-controlled switches and the value of the incrementer. The design consists of an incrementer function feeding a comparator, and four switches fed into two `AND` gates. The comparator and `AND` gate outputs feed an `OR` gate, which feeds an LED on supported DSP development boards.

The SignalTap II embedded logic analyzer captures the signal activity at the output of the two `AND` gates and the incrementer of the design loaded into the Altera device on DSP Builder-supported development boards. The logic analyzer retrieves the values and displays them in the MATLAB work space.

For more information on using the SignalTap II embedded logic analyzer with the Quartus II software, refer to the Quartus II Help.

DSP Builder and the `SignalTap II Logic Analyzer` block create a SignalTap II embedded logic analyzer that:

- Has a simple, easy-to-use interface
- Analyzes signals in the top-level design file
- Uses a single clock source
- Captures data around a trigger point: 88% of the data is pre-trigger and 12% of the data is post-trigger

You can also use the Quartus II software to insert an instance of the SignalTap II embedded logic analyzer in your design. The Quartus II software supports additional features such as using multiple clock domains, and adjusting what percentage of data is captured around the trigger point.

SignalTap II Design Flow

Working with the SignalTap II embedded logic analyzer in DSP Builder involves the following flow:

1. Add a `SignalTap II Logic Analyzer` block to your design.
2. Specify the signals (nodes) that you want to analyze by inserting `SignalTap II Node` blocks.
3. Turn on the **Enable SignalTap** option in `Signal Compiler`.
4. Choose one of the JTAG cable ports in `Signal Compiler` or the SignalTap II Logic Analyzer.
5. Using `Signal Compiler`, synthesize your model, perform Quartus II compilation, and download the design into the DSP development board (starter or professional).
6. Specify the required trigger conditions in the `SignalTap II Logic Analyzer` block.

For more information on the `SignalTap II Logic Analyzer` and `SignalTap II Node` blocks, refer to the descriptions of these blocks in the *AltLab Library* chapter of the *DSP Builder Reference Manual*.

SignalTap II Nodes

By definition, a node represents a wire carrying a signal that travels between different logical components of a design file. The SignalTap II embedded logic analyzer can capture signals from any internal device node in a design file, including I/O pins.

The SignalTap II embedded logic analyzer can analyze up to 128 internal nodes or I/O elements. As more signals are captured, more logic elements (LEs) and embedded system blocks (ESBs) are used.

Before capturing signals, each node to be analyzed must be assigned to a SignalTap II embedded logic analyzer input channel. To assign a node to an input channel, you must connect it to a SignalTap II Node block.

SignalTap II Trigger Conditions

The trigger pattern describes a logic event in terms of logic levels and/or edges. It is a comparison register used by the SignalTap II embedded logic analyzer to recognize the moment when the input signals match the data specified in the trigger pattern.

The trigger pattern is composed of a logic condition for each input signal. By default, all signal conditions for the trigger pattern are set to "Don't Care," masking them from trigger recognition. You can select one of the following logic conditions for each input signal in the trigger pattern:

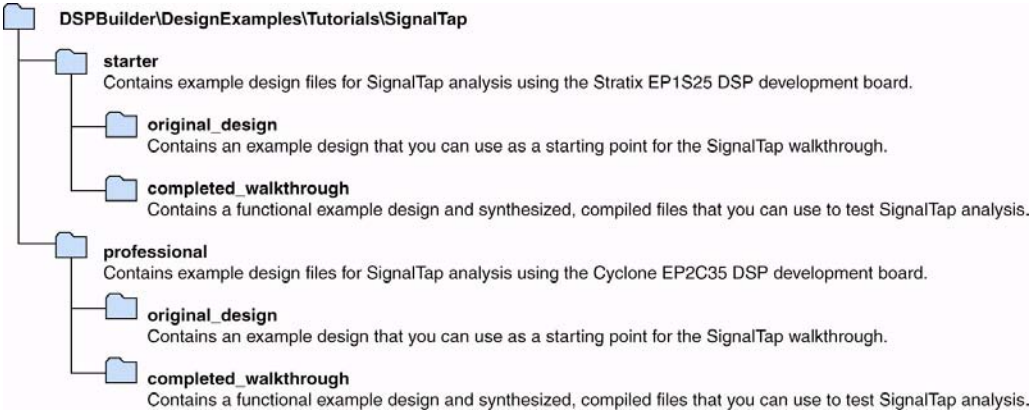
- Don't Care
- Low
- High
- Rising Edge
- Falling Edge
- Either Edge

The SignalTap II embedded logic analyzer is triggered when it detects the trigger pattern on the input signals.

SignalTap II Walkthrough

Altera provides several design files for this walkthrough in the directory structure shown in Figure 6–1.

Figure 6–1. SignalTap II Design Example Directory Structure



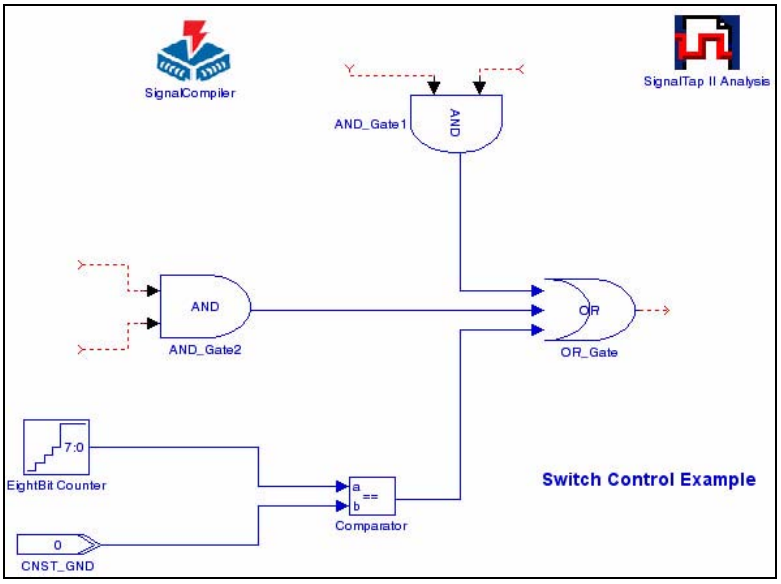
You can start from the design in the **original_design** directory and go through the complete walkthrough. Alternatively, you can use the design in the **completed_walkthrough** directory and go directly to "Turn On the SignalTap II Option in Signal Compiler" on page 6–8.

Open the Walkthrough Example Design

To open the example design:

1. Open the **switch_control.mdl** design in the *<DSP Builder install path>***DesignExamples\Tutorials\SignalTap\professional\original_design** directory. (Figure 6–2).

Figure 6–2. Starting Point for the SignalTap II Walkthrough



Add the Configuration and Connector Blocks

You must add the board configuration block and connector blocks for the board that you want to use. This walkthrough uses the Cyclone EP2C35 development board.

1. Select the Boards library from the **Altera DSP Builder Blockset** folder in the Simulink library browser. See Chapter 2, Getting Started Tutorial for instructions on accessing libraries.

2. Open the CycloneIIEP2C35 folder. Drag and drop the `Cyclone II EP2C35 DSP Development Board` configuration block into your model.

3. Drag and drop the SW2 and SW3 blocks close to the AND_Gate2 block in your model. Connect these switch blocks to the AND_Gate2 inputs.

4. Drag and drop the SW4 and SW5 blocks close to the AND_Gate1 block in your model. Connect these switch blocks to the AND_Gate1 inputs.

   You can rotate the SW5 block to make the connection easier by clicking the block with the right mouse button and choosing **Rotate Block** from the Format menu.

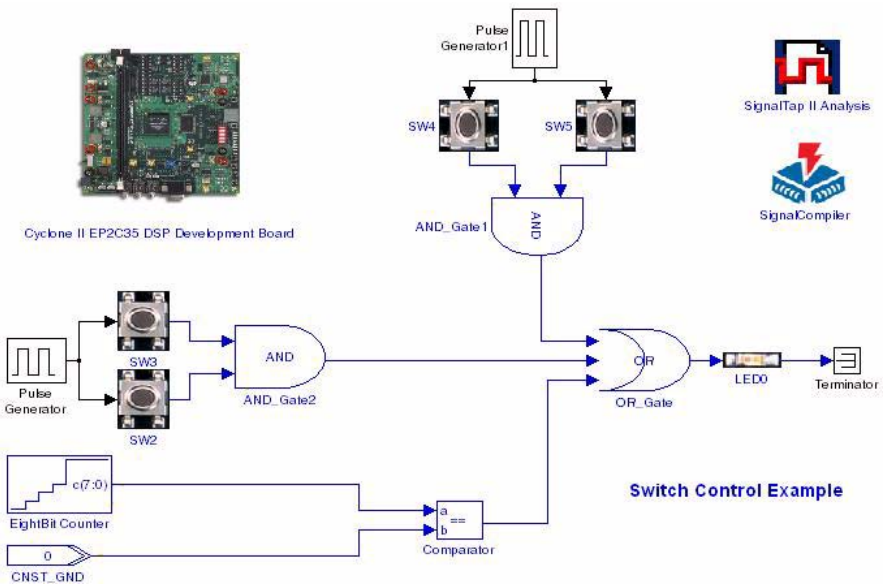
5. Drag and drop the LED0 block close to the OR_Gate block in your model. Connect this blocks to the OR_Gate output.

6. Select the Simulink Sources library. Drag and drop a Pulse Generator block near the SW2 and SW3 blocks and connect it to these blocks.

7. Drag and drop another Pulse Generator block near the SW4 and SW5 blocks and connect it to these blocks.

Your model should look similar to Figure 6–3.

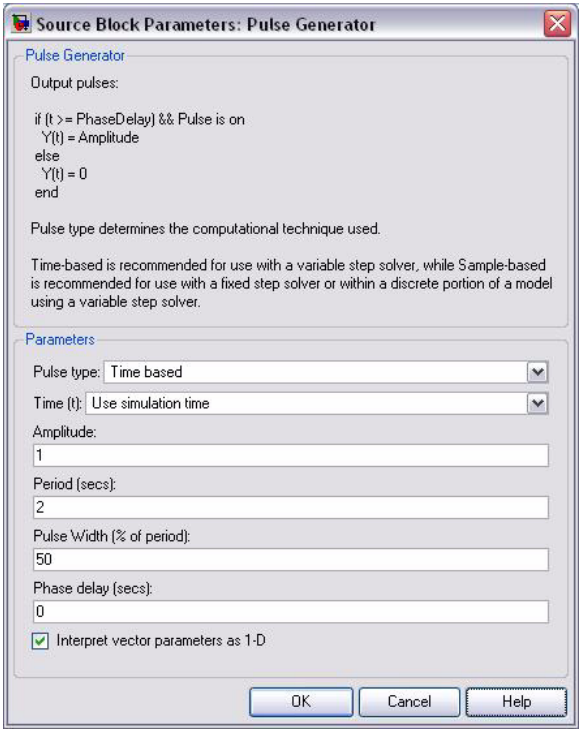
Figure 6–3. Switch Control Example with Board, Pulse Generator and Terminator Blocks



8. Set the following parameters in the Block Parameters dialog box for both pulse generator blocks (Figure 6–4):

   - **Pulse Type**: `Time based`
   - **Time**: `Use simulation time`
   - **Amplitude**: `1`
   - **Period**: `2`
   - **Pulse Width**: `50`
   - **Phase delay**: `0`
   - **Interpret vector parameters as 1-D**: `On`

Figure 6–4. Pulse Generator Dialog Box



9. Select the Simulink Sinks library. Drag and drop a `Terminator` block near the `OR_Gate` block and connect it to this block.

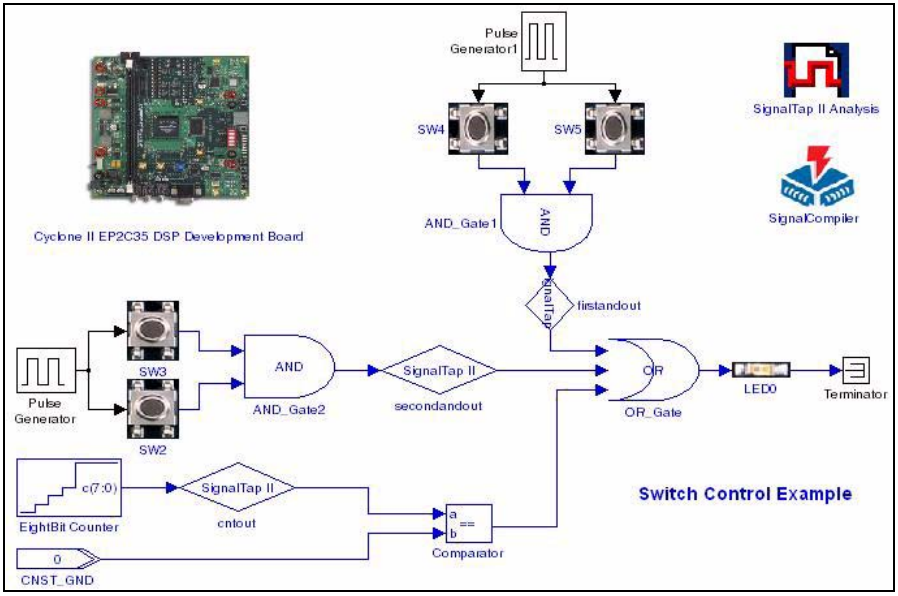
Specify the Nodes to Analyze

In the following steps, you add `SignalTap II Node` blocks to the signals (also called nodes) that you want to analyze; in this walkthrough they are the output of each `AND` gate and the output of the incrementer. Perform the following steps:

1. Open the AltLab library in the Simulink Library Browser. Drag a `SignalTap II Node` block into your design. Position the block so that it is on top of the connection line between the `AND_Gate1` block and the `OR_Gate` block. See Figure 6–5 if you are unsure of the positioning.

   If you position the block using this method, the Simulink software inserts the block and joins connection lines on both sides.

Figure 6–5. Completed SignalTap II Design



2. Click the text under the block icon in your model and change the block instance name by deleting the text and typing in the new text `firstandout`.

3. Add a `SignalTap II Node` block between the `AND_Gate2` block and the `OR_Gate` block and name it `secondandout`.

4. Add a `SignalTap II Node` block between the `Eightbit Counter` block and the `Comparator` block and name it `cntout`.

5. Choose **Save** (File menu).

Turn On the SignalTap II Option in Signal Compiler

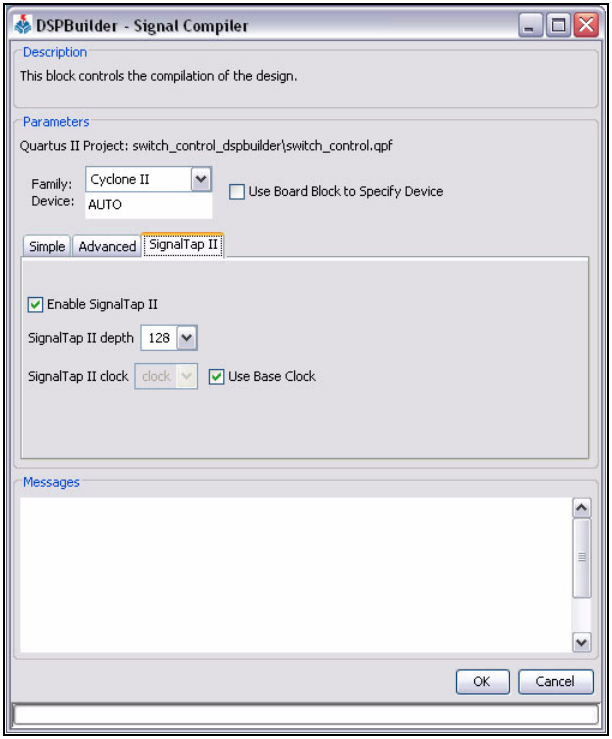
When you add node blocks to signals, each block is implicitly connected to the SignalTap II embedded logic analyzer.

This is a functional change to the model and you must re-compile the design before you can use the SignalTap II embedded logic analyzer.

To compile the design, perform the following steps:

1. Double-click the `Signal Compiler` block and choose the **SignalTap II** tab in the Signal Compiler dialog box (Figure 6–6).

Figure 6–6. SignalTap II Tab in Signal Compiler



2. Check that the **Enable SignalTap II** option is turned on.

   When this option is turned on, `Signal Compiler` inserts an instance of the SignalTap II embedded logic analyzer into the design.

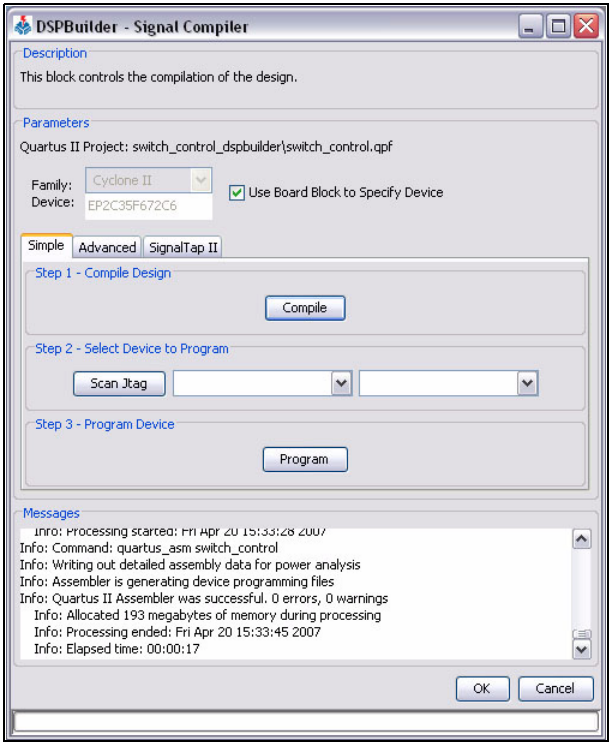
3. Select a depth of `128` for the SignalTap II sample buffer (that is, the number of samples stored for each input signal) in the **Depth** list.

4. Check that the **Use Base Clock** option is turned on.

5. Choose the **Simple** tab and check that the **Use Board Block to Specify Device** option is turned on (Figure 6–7).

6. Click the **Compile** button.

Figure 6–7. Simple Tab in Signal Compiler



When the conversion is complete, the dialog box messages display the memory allocated during processing.

You must compile the design before you open the `SignalTap II Analyzer` block because the block relies on data files that are created during compilation.

7. Click **Scan Jtag** and select the appropriate download cable and device (for example, `USB-Blaster` cable and `EP2C35` device).

8. Click **Program** to download your design to the development board.

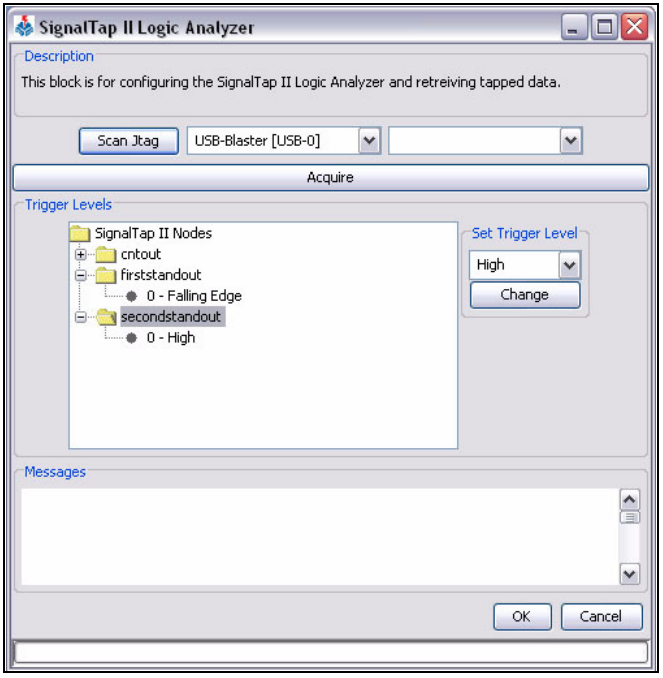
9. Click **OK**.

Specify the Trigger Levels

To specify the trigger levels, perform the following steps:

1. Double-click the `SignalTap II Logic Analyzer` block. The dialog box displays all of the nodes connected to `SignalTap II Node` blocks as signals to be analyzed (Figure 6–8).

Figure 6–8. SignalTap II Logic Analyzer



2. Specify the trigger condition for the `firstandout` node:
   a. Click `firstandout` under `Signal Tap II Nodes`.
   b. Select `Falling Edge` in the **Set Trigger Level** list.
   c. Click **Change**. The condition is updated.
3. Repeat these steps to specify the trigger condition `High` for the `secondandout` node.

The SignalTap II embedded logic analyzer captures data for analysis when it simultaneously detects all trigger patterns on the input signals. For example, because you specified `Falling Edge` for `firstandout` and `High` for `secondandout`, the SignalTap II embedded logic analyzer is only triggered when it detects a falling edge on `firstandout` *and* a logic level high on `secondandout`.

Perform SignalTap II Analysis

You are ready to run the analyzer and display the results in a MATLAB plot. After you click **Acquire**, the SignalTap II embedded logic analyzer begins analyzing the data and waits for the trigger conditions to occur.

Perform the following steps:

1. Click **Scan Jtag** in the SignalTap II Analyzer dialog box and select the appropriate download cable and device.
2. Click **Acquire**.
3. Press switch SW4 on the DSP development board to trigger the SignalTap II embedded logic analyzer.

Note that if switch SW2 or SW 3 is pressed and held while pressing switch SW4, the trigger condition is not met and acquisition does not occur.

4. Click **OK** in the SignalTap II Analysis dialog box when you are finished.

   The captured data are interpreted as unsigned values and displayed in MATLAB plots. The values are also stored in MATLAB **.mat** files in the working directory.

Figure 6–9 shows the MATLAB plot for the SignalTap II node `firstandout`.

Figure 6–9. MATLAB Plot for SignalTap II Node firstandout

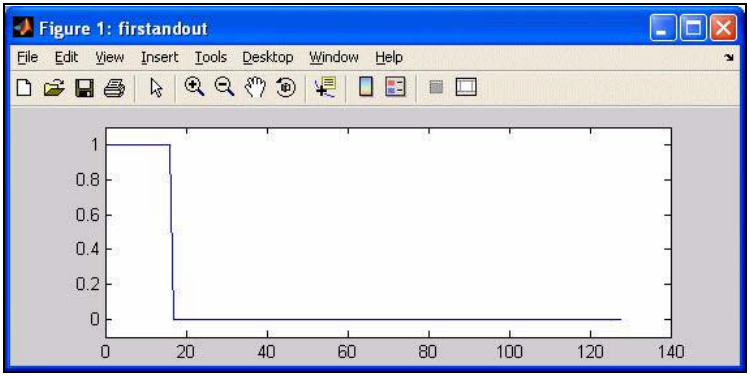


Figure 6–10 shows the MATLAB plot for the SignalTap II node `secondandout`.

Figure 6–10. MATLAB Plot for SignalTap II Node secondandout

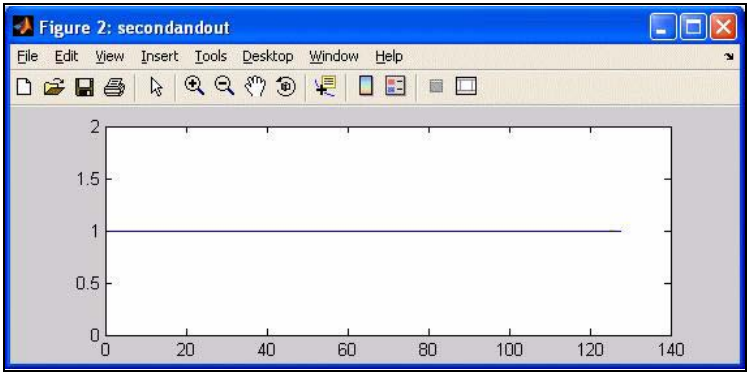
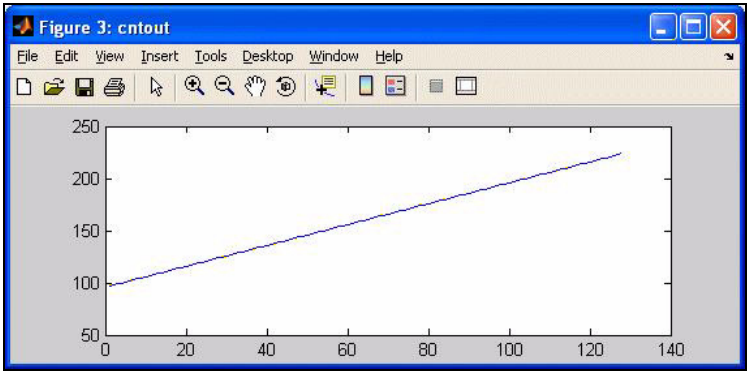


Figure 6–10 shows the MATLAB plot for the SignalTap II node `secondandout`.

Figure 6–11. MATLAB Plot for SignalTap II Node secondandout



For more information on the `SignalTap II Logic Analyzer` block, refer to the *SignalTap II Logic Analyzer* block description in the *AltLib Library* chapter in the *DSP Builder Reference Manual*.

7. Using the Interfaces Library

Introduction

This chapter describes how to use the Avalon® Memory-Mapped (Avalon-MM) blocks in the Interfaces library to create a design which functions as a custom peripheral to SOPC Builder.

SOPC Builder is a system development tool for creating systems based on processors, peripherals, and memories. SOPC Builder automates the task of integrating hardware components into a larger system.

To integrate a DSP Builder design into your SOPC Builder system, your peripheral must meet the Avalon-MM interface or Avalon-ST interface specification and qualify as a SOPC Builder-ready component.

The Interfaces library supports peripherals that use the Avalon-MM and Avalon Streaming (Avalon-ST) interface specifications.

The correct version of MATLAB with DSP Builder installed must be available on your system path to integrate DSP Builder MDL files in SOPC Builder.

Avalon-MM Interface

The *Avalon Memory-Mapped Interface Specification* provides peripheral designers with a basis for describing the address-based read/write interface found on master (for example, a microprocessor or DMA controller) and slave peripherals (for example, a memory, UART, or timer).

The `Avalon-MM Master` and `Avalon-MM Slave` blocks in DSP Builder provide a seamless flow for creating a DSP Builder block as a custom peripheral and integrating the block into your SOPC Builder system. These blocks provide you the following benefits:

- Automates the process of specifying Avalon-MM ports that are compatible with the Avalon-MM bus
- Supports multiple Avalon-MM Master and Avalon-MM Slave instantiations
- Saves time spent hand coding glue logic that connects Avalon-MM ports to DSP blocks

For more information on SOPC Builder, refer to the *Quartus II Handbook Volume 4: SOPC Builder*. For more information on the Avalon-MM Interface, refer to the *Avalon Memory-Mapped Interface Specification*.

Avalon-MM Interface Blocks

A SOPC Builder component is a design module that SOPC Builder recognizes and can automatically integrate into a system.

SOPC Builder can recognize a DSP Builder design model provided that it is in the same working directory as the SOPC Builder project.

With the Avalon-MM blocks provided in the DSP Builder library, you can design the DSP function and add an Avalon-MM block which makes it a custom peripheral within Simulink environment.

Each Avalon-MM block can be instantiated multiple times in a design to implement an SOPC component with multiple master and/or slave ports.

Avalon-MM Slave Block

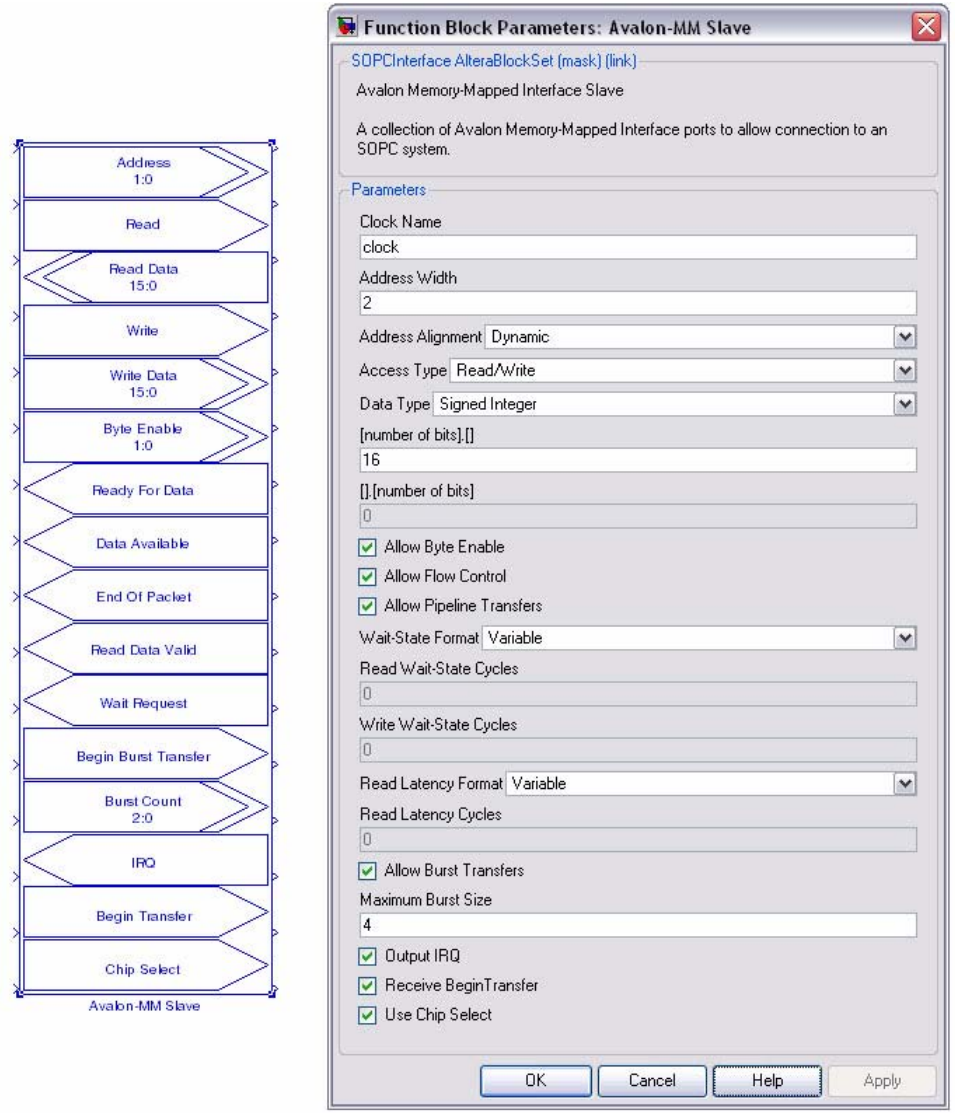
The `Avalon-MM Slave` block supports the following signals:

- `clock`
- `address`
- `read`
- `readdata`
- `write`
- `writedata`
- `byteenable`
- `readyfordata`
- `dataavailable`
- `endofpacket`
- `readdatavalid`
- `waitrequest`
- `beginbursttransfer`
- `burst count`
- `irq`
- `begintransfer`
- `chipselect`

Refer to the *DSP Builder Reference Manual* for more information about these signals.

The block shown in Figure 7–1 describes an Avalon-MM Slave interface where all of the Avalon-MM signals have been enabled.

Figure 7–1. Avalon-MM Slave Block Signals



Each of the input and output ports of the block correspond to the input and output ports of the pin or bus shown between the ports.

Inputs to the DSP Builder core are displayed as right pointing bus/pins; outputs from the core are displayed as left pointing pins/buses.

The opposite end of any pins can be used to provide "pass-through" test data from the Simulink domain.

Avalon-MM Master Block

You may want to use an `Avalon-MM Master` block (for example, to design a DMA controller) in a design which functions as an Avalon-MM Master in your SOPC Builder system.

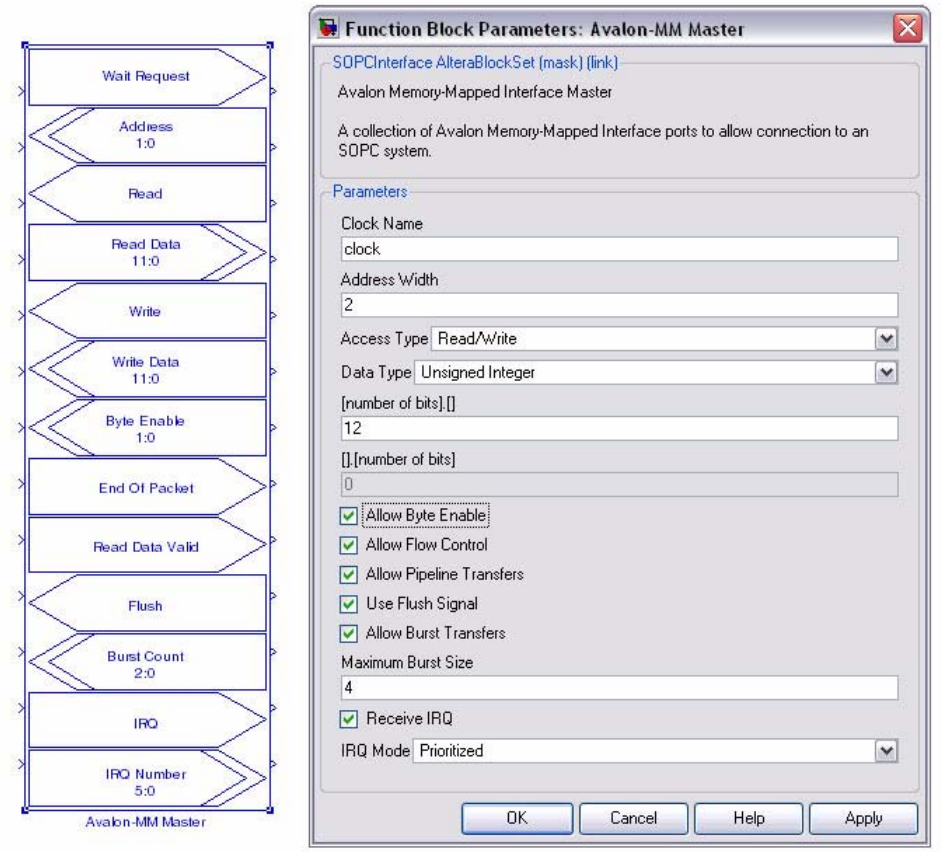
The `Avalon-MM Master` block is similar to the `Avalon-MM Slave` block and supports the following signals:

- `clock`
- `waitrequest`
- `address`
- `read`
- `readdata`
- `write`
- `writedata`
- `byteenable`
- `endofpacket`
- `readdatavalid`
- `flush`
- `burstcount`
- `irq`
- `irqnumber`

Refer to the *DSP Builder Reference Manual* for more information about these signals.

The block shown in Figure 7–2 on page 7–5 describes an Avalon-MM Master interface where all of the Avalon-MM signals have been enabled.

Figure 7–2. Avalon-MM Master Block Signals



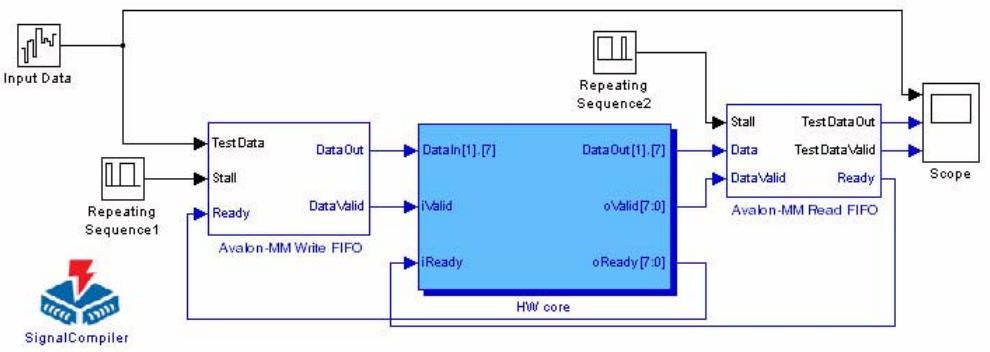
Wrapped Blocks

While the `Avalon-MM Master` and `Avalon-MM Slave` interface blocks allow you to generate a SOPC component in DSP Builder, they do little to mask the complexities of the interface. The Avalon-MM read and write FIFO blocks in the Interfaces library provide a higher level of abstraction.

You can implement a typical DSP core to handle data in a streaming manner, using the signals `Data`, `Valid` and `Ready`. To provide a high level view, configurable `Avalon-MM Write FIFO` and `Avalon-MM Read FIFO` blocks are provided for you to map Avalon-MM interface signals to this protocol.

Figure 7–3 shows an example system with Avalon-MM Write FIFO and Avalon-MM Read FIFO blocks.

Figure 7–3. Example System with Avalon-MM Write FIFO and Avalon-MM Read FIFO Blocks



Avalon-MM Write FIFO

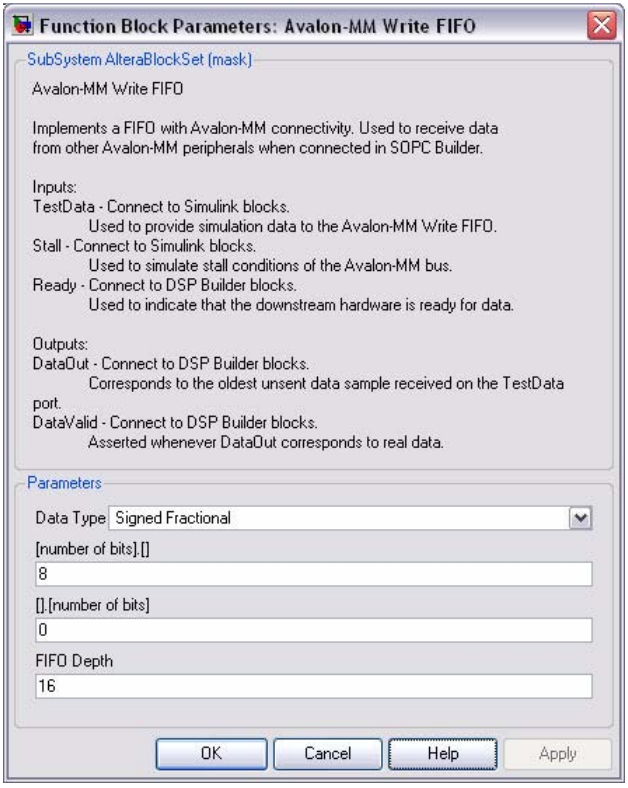
An Avalon-MM Write FIFO has the following ports:

- **TestData** (input): This port should connect to a Simulink block that provides simulation data to the Avalon-MM Write FIFO. The data is passed to the DataOut port one cycle after the Ready input port is asserted,
- **Stall** (input): This port should be connected to Simulink blocks and is used to simulate stall conditions of the Avalon-MM bus and hence underflow to the SOPC component. For any simulation cycle where Stall is asserted, the test data is cached by the Avalon-MM Write Test Converter and released in order, one sample per clock, when stall is de-asserted.
- **Ready** (input): This port should be connected to DSP Builder blocks and is used to indicate that the downstream hardware is ready for data.
- **DataOut** (output): This port should be connected to DSP Builder blocks and corresponds to the oldest unsent data sample received on the TestData port.
- **DataValid** (output): This port should be connected to DSP Builder blocks and is asserted whenever DataOut corresponds to real data.

Double-clicking on an Avalon-MM Write FIFO block brings up a Block Parameters dialog box which can be used to set parameters for the data type, data width and FIFO depth.

Figure 7–4 shows the Avalon-MM Write FIFO dialog box.

Figure 7–4. Figure 4: Avalon-MM Write FIFO Block Parameters



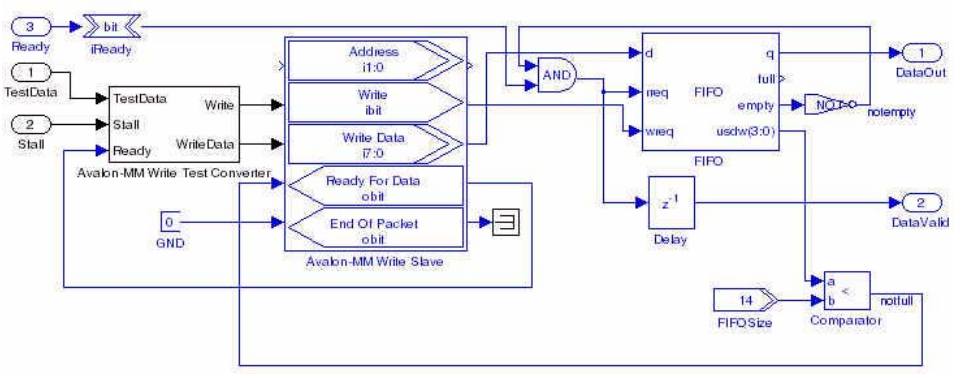
Refer to the *DSP Builder Reference Manual* for information about these parameters.

You can open the hierarchy below the `Avalon-MM Write FIFO` block by clicking on the block with the right mouse button and choosing **Look Under Mask** from the popup menu.

You can use this design as a template to design new functionality if required (for example, when an Avalon-MM address input is used to split incoming streams).

The internal content of an Avalon-MM Write FIFO is shown in Figure Figure 7–5.

Figure 7–5. Avalon-MM Write FIFO Content



The Avalon-MM Write Test Converter handles caching and conversion of Simulink/MATLAB data into accesses over the Avalon-MM interface and can be used to test the functionality of the user design. The Avalon-MM Write Test Convertor is simulation only and does not synthesize to HDL.

Avalon-MM Read FIFO

An Avalon-MM Read FIFO has the following ports:

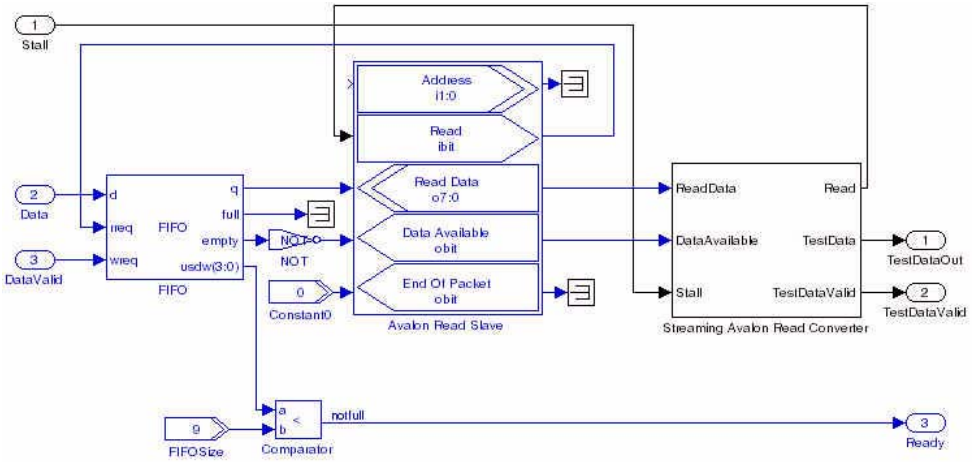
- **Stall** (input): This port should be connected a Simulink block that is used to simulate stall conditions of the Avalon-MM bus and hence back pressure to the SOPC component. For any simulation cycle where `Stall` is asserted, no Avalon-MM reads take place and the internal FIFO fills. When full, the `Ready` output is de-asserted so that no data is lost.
- **Data** (input): This port should be connected to DSP Builder blocks and should be connected to outgoing data from the user design.
- **DataValid** (input): This port should be connected to DSP Builder blocks and should be asserted whenever the signal on the `Data` port corresponds to real data.
- **TestDataOut** (output): This port should be connected to Simulink blocks and corresponds to data received over the Avalon-MM bus.
- **TestDataValid** (output): This port should be connected to Simulink blocks and is asserted whenever `TestDataOut` corresponds to real data.
- **Ready** (output): When asserted, indicates that the block is ready to receive data.

Double-clicking on an `Avalon-MM Write FIFO` block brings up a Block Parameters dialog box which can be used to set parameters for the data type, data width and FIFO depth.

Refer to the *DSP Builder Reference Manual* for information about these parameters.

You can open the hierarchy below the `Avalon-MM Read FIFO` block by clicking on the block with the right mouse button and choosing **Look Under Mask** from the popup menu. The internal content of an Avalon-MM Read FIFO is shown in Figure Figure 7–6.

Figure 7–6. Avalon-MM Read FIFO Content



The Avalon-MM Read Data Converter handles caching and conversion of Simulink/MATLAB data into accesses over the Avalon-MM interface and can be used to test the functionality of the user design. The Avalon-MM Read Data Convertor is simulation only and does not synthesize to HDL.

Avalon-MM Interface Blocks Walkthrough

This walkthrough describes how to interface a design built using the Avalon-MM Blocks as a custom peripheral to the Nios® II embedded processor in SOPC Builder.

The design consists of a 4-tap FIR filter with variable coefficients. The coefficients are loaded using the Nios embedded processor while the input data is supplied by an off-chip source through an analog-to-digital converter. The filtered output data is sent off-chip through a digital-to-analog converter.

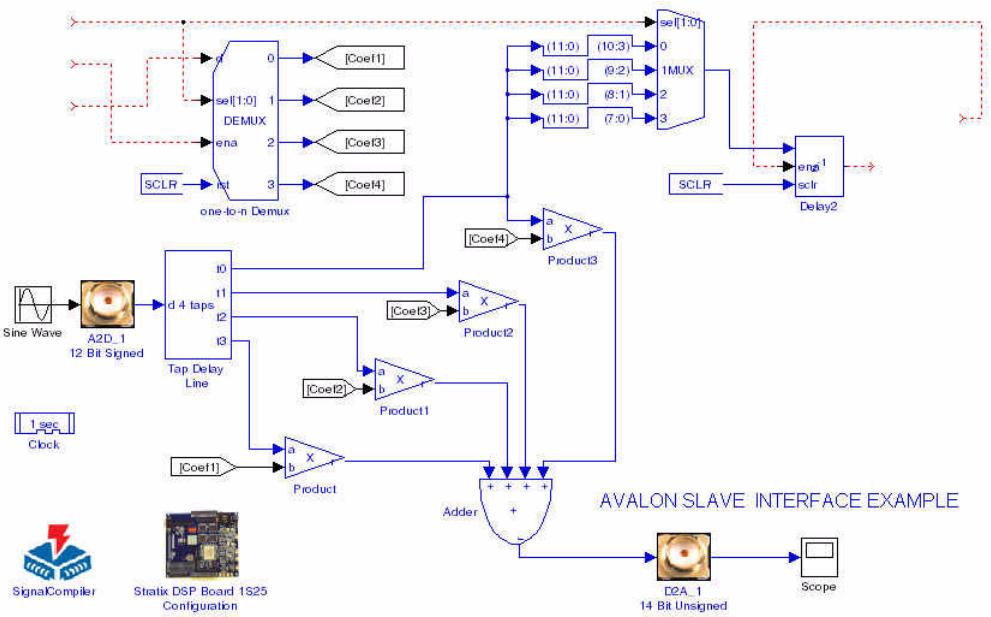
Add Avalon-MM Blocks to the Example Design

To complete the example design, perform the following steps:

1. Choose **Open** (File menu) in the MATLAB software.
2. Browse to the *<DSP Builder install path>***DesignExamples****Tutorials****SOPCBuilder****SOPCBlock** directory.
3. Select the **new_topavalon.mdl** file and click **Open**.

Figure 7–7 shows **new_topavalon.mdl**.

Figure 7–7. new_topavalon.mdl Example Design



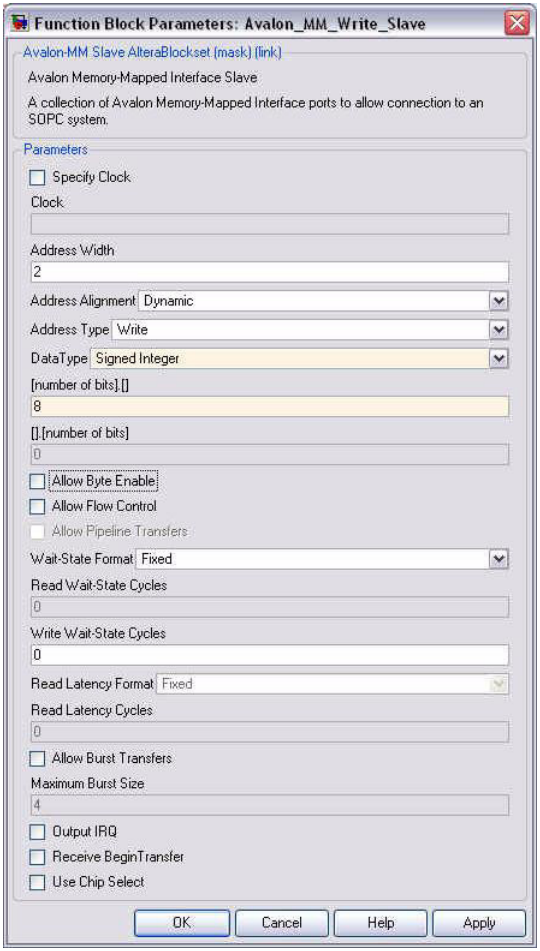
4. Rename the file by choosing **Save As** (File menu). Create a new folder called **MySystem** and save your new MDL file as **topavalon.mdl** in this folder.
5. Open the Simulink Library Browser by clicking on the icon or by typing `simulink` at the MATLAB command prompt. Expand the **Altera DSP Builder Blockset** in the Simulink Library Browser and select **Avalon Memory-Mapped** in the Interfaces library.

6. Drag and drop an `Avalon-MM Slave` block into the top left of the model. Change the block name to `Avalon_MM_Write_Slave`.

7. Double-click on the `Avalon_MM_Write_Slave` block to bring up the Block Parameters dialog box.

8. Choose `Write` in the **Address Type** list box. Choose `Signed Integer` in the **Data Type** list box and specify 8 bits for the data width. Turn off the **Allow Byte Enable** check box. (Figure 7–8). Click **OK**.

Figure 7–8. Block Parameters for Avalon_MM_Write_Slave in topavalon.mdl



Notice that the Avalon_MM_Write_Slave block is redrawn with three ports: Address i1:0, Write ibit, and Write Data i7:0.

9. Connect the ports as shown in Figure 7–9.

Note that you can re-size a block by dragging the resize handles at each corner.

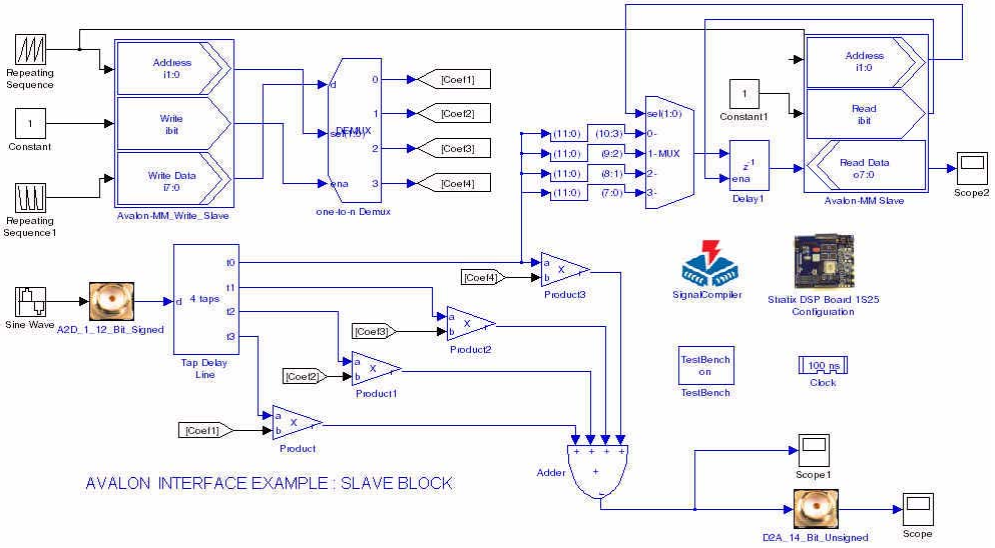
10. Drag and drop another Avalon-MM Slave block into the top right of the model and change the name of this block instance to Avalon_MM_Read_Slave.

11. Double-click on the Avalon_MM_Read_Slave block to bring up the Block Parameters dialog box.

12. Choose Read in the **Address Type** list box. Choose Signed Integer in the **Data Type** list box and specify 8 bits for the data width. Click **OK** and notice that the Avalon_MM_Read_Slave block is redrawn with three ports: Address i1:0, Read ibit, and Read Data o7:0.

13. Connect the Avalon_MM_Read_Slave ports as shown in Figure 7–9.

Figure 7–9. topavalon.mdl Example Design



14. Choose **Save** (File menu) in the model window to save your model.

Verify Your Design

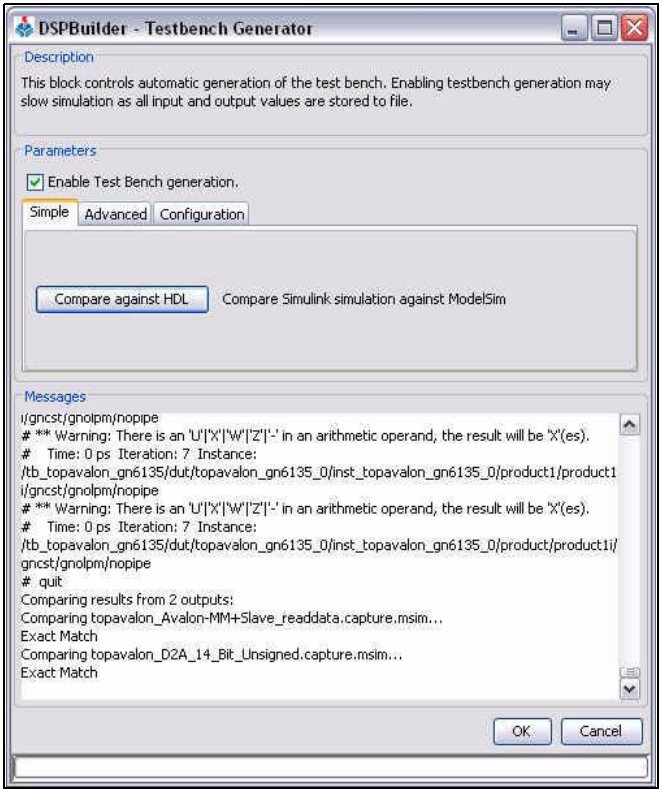
Before using your design in SOPC Builder, you should use the `TestBench` block to verify your design.

Perform the following steps:

1. Double-click the `TestBench` block to display the TestBench dialog box.

2. Click **Compare against HDL** (Figure 7–10).

Figure 7–10. TestBench Dialog Box



This generates HDL, runs Simulink and ModelSim then compares the simulation results. Progress is shown in the message window which should complete with a message "`Exact Match`".

3. Click **OK**.

Instantiate Your Design in SOPC Builder

To instantiate your design as a custom peripheral to the Nios II embedded processor in SOPC Builder, perform the following steps:

1. Start the Quartus® II software.
2. In the Quartus II software, choose **New Project Wizard** (File menu).
   a. Specify the working directory for your project by browsing to the *<DSP Builder install path>***DesignExamples****Tutorials****SOPCBuilder****SOPCBlock****MySystem** directory.
   b. Specify a name for your project. This walkthrough uses `SOPC` for the project name.

The Quartus II software automatically specifies a top-level design entity that has the same name as the project. This walkthrough assumes that the names are the same.

   c. Click **Finish** to create the Quartus II project.
3. Choose **Tcl Scripts** (Tools menu). Select `topavalon_add` in the Project folder and click **Run** to load your MDL file and other required files into the Quartus II project.
4. Choose **SOPC Builder** (Tools menu) to display the Create New System dialog box. Enter `nios32` as your **System Name**, select `VHDL` for your HDL Language and click **OK** (Figure 7–11).

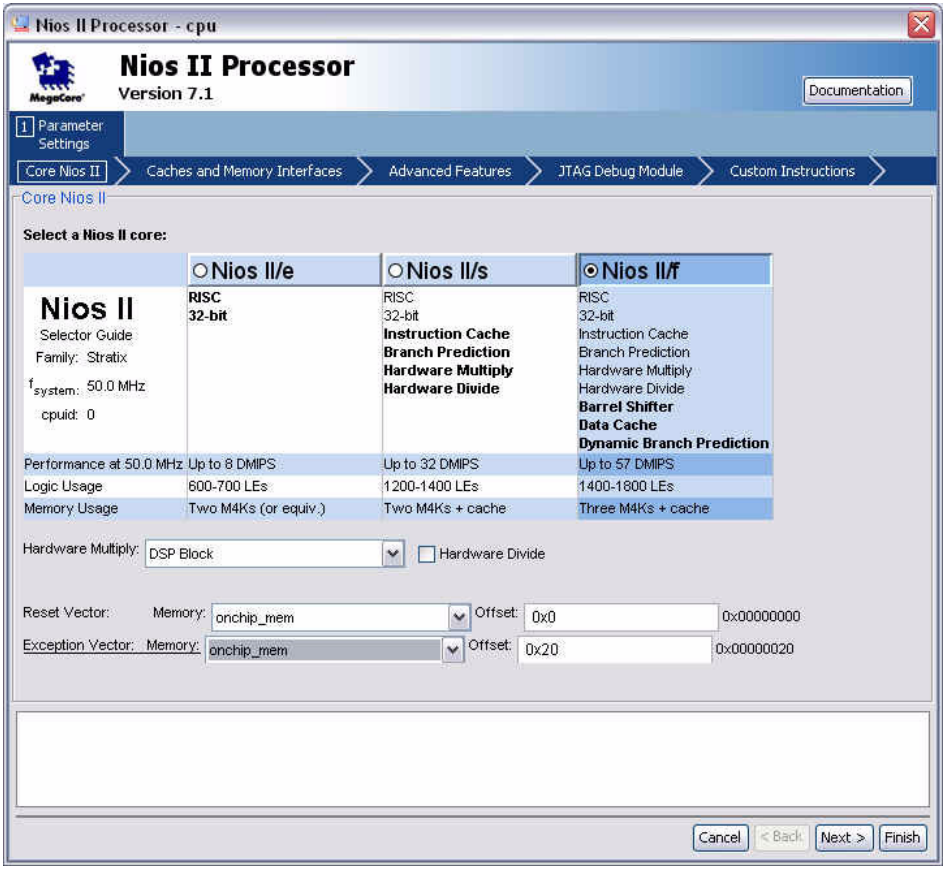
Figure 7–11. SOPC Builder Create New System Dialog Box

5. Click the **System Contents** tab in SOPC Builder and expand **Memories and Memory Controllers**. Expand **On-Chip** and double-click `On Chip Memory (RAM or ROM)`. Click **Finish** to add an on-chip RAM device with default parameters.

6. Double-click the `Nios II Processor` module in the **System Contents** tab to display the MegaWizard interface.

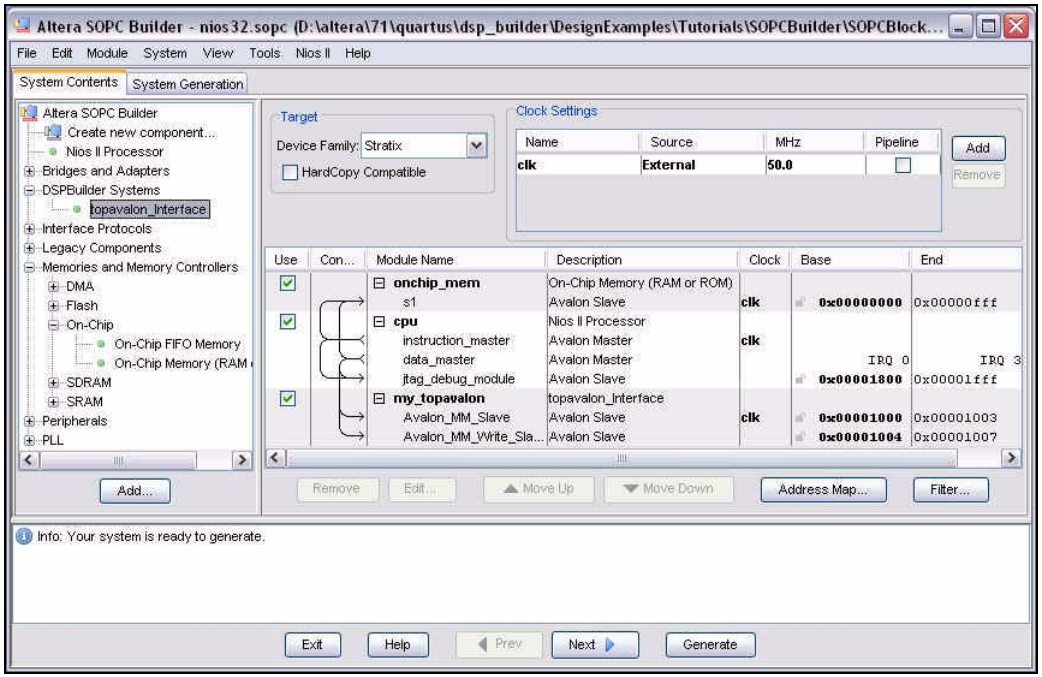
7. Set the reset and exception vectors to use `onchip_mem` and click **Finish** to add the processor to your system with all other parameters set to their default values (Figure 7–12).

Figure 7–12. Nios II Processor Configuration



8. Expand **DSPBuilder Systems** in the **System Contents** tab and double-click the `topavalon_interface` module to include it in your Nios II system (Figure 7–13).

Figure 7–13. Including Your DSP Builder Design Module in SOPC Builder



If the memory device, Nios II processor, and DSP Builder system are added in this order, you should not need to set a base address. However, you can choose Auto-Assign Base Addresses from the System menu to automatically add a base address if necessary.

You can now design the rest of your Nios II embedded processor system using the standard Nios II embedded processor design flow.

For information on using SOPC Builder to create a custom Nios II embedded processor, see *AN 351: Simulating Nios II Embedded Processor Designs*.

A completed version of the `topavalon.mdl` design is available in the *<DSP Builder install path>***DesignExamples****Tutorials****SOPCBuilder****SOPCBlock****Finished Example** directory.

Avalon-MM FIFO Walkthrough

This walkthrough describes how to interface a design built using the `Avalon-MM FIFO` block as a custom peripheral to the Nios® II embedded processor in SOPC Builder.

The design consists of a Prewitt edge detector with one Avalon-MM Write FIFO and one Avalon-MM Read FIFO. An additional slave port is used as a control port.

Refer to *AN364: Edge Detection Reference Design* for a full description of the Prewitt edge detector design.

For the hardware implementation described in the application note, the image is stored in the compact flash and loaded via DMA using the Nios II embedded processor. The edge detected image is output through a VGA controller. This is modelled in DSP Builder by using Simulink to read in the original image, and to capture the edge detected result.

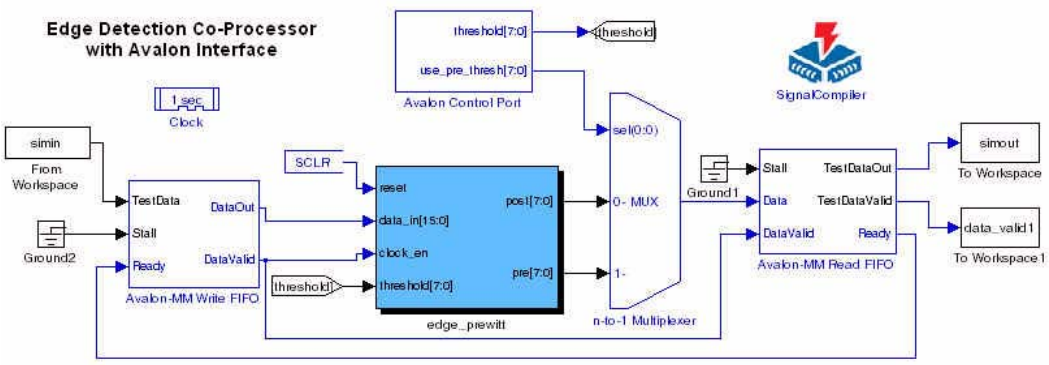
Open the Walkthrough Example Design

To open the example design, perform the following steps:

1. Choose **Open** (File menu) in the MATLAB software.
2. Browse to the *<DSP Builder install path>***DesignExamples****Tutorials****SOPCBuilder****AvalonFIFO** directory.
3. Select the **sopc_edge_detector.mdl** file and click **Open**.

Figure 7–14 shows **sopc_edge_detector.mdl**.

Figure 7–14. sopc_edge_detector.mdl Example Design



Compile the Design

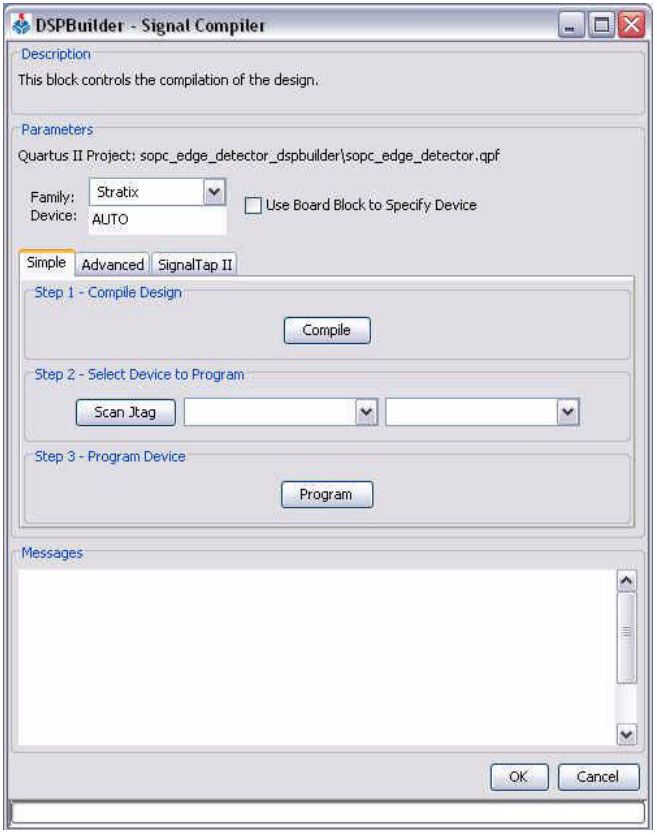
Compile the design to ensure that it produces valid HDL.

Alternatively, you could use the `TestBench` block as described for the "Avalon-MM Interface Blocks Walkthrough" in "Verify Your Design" on page 7–13.

Perform the following steps:

1. Double-click the `Signal Compiler` block.
2. Select the **Device Family** for the DSP Development board you are using. The Walkthrough design is configured for a `Stratix 1S25` board (Figure 7–15).

Figure 7–15. Signal Compiler Dialog Box



3. Click **Compile**.

4. When the compilation has completed successfully, click **OK**.

☞ The Avalon-MM Read/Write Converter is simulation only and does not synthesize to HDL.

Instantiate Your Design in SOPC Builder

To instantiate your design as a custom peripheral to the NIOS II embedded processor in SOPC Builder, perform the following steps:

1. Start the Quartus II software.

2. In the Quartus II software, choose **New Project Wizard** (File menu).

   a. Specify the working directory for your project by browsing to the *<DSP Builder install path>***DesignExamples****Tutorials****SOPCBuilder****AvalonFIFO** directory.

   b. Specify a name for your project. This walkthrough uses `FIFO` for the project name.

☞ The Quartus II software automatically specifies a top-level design entity that has the same name as the project. This walkthrough assumes that the names are the same.

   c. Click **Finish** to create the Quartus II project.

3. Choose **Tcl Scripts** (Tools menu). Load your design by selecting `sopc_edge_detector_add` in the Project folder and clicking **Run**.

4. Choose **SOPC Builder** (Tools menu) to display the Create New System dialog box. Enter `AvalonFIFO` as your **System Name**, select `VHDL` for your HDL Language and click **OK**.

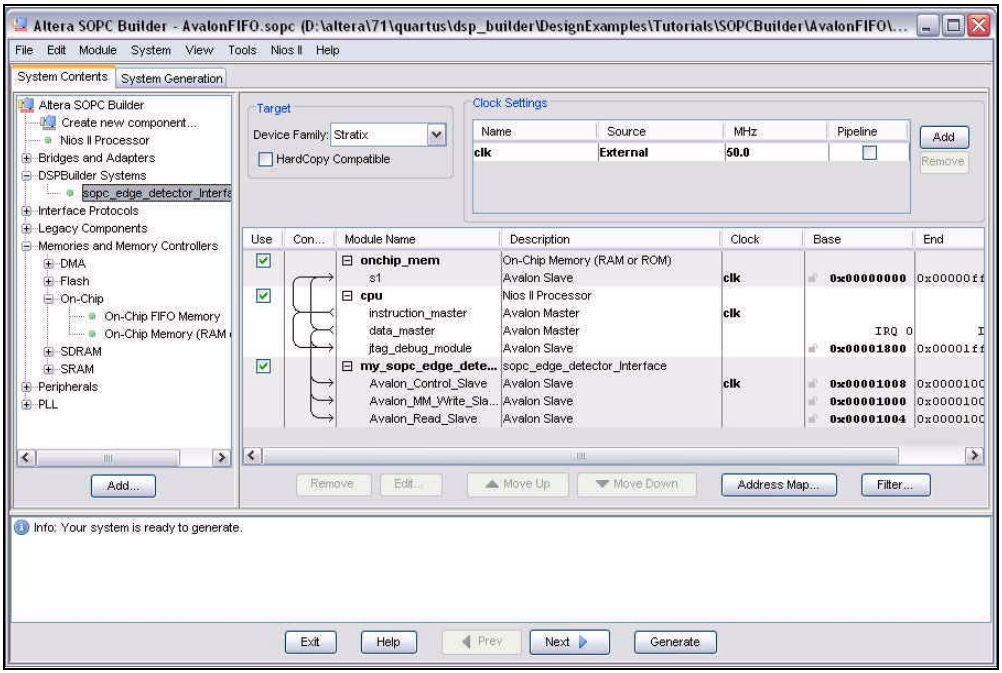
5. Click the **System Contents** tab in SOPC Builder and expand **Memories and Memory Controllers**. Expand **On-Chip** and double-click `On Chip Memory (RAM or ROM)`. Click **Finish** to add an on-chip RAM device with default parameters.

6. Double-click the `Nios II Processor` module in the **System Contents** tab to display the MegaWizard interface.

7. Set the reset and exception vectors to use `onchip_mem` and click **Finish** to add the processor to your system with all other parameters set to their default values.

8. Expand **DSPBuilder Systems** in the **System Contents** tab and double-click the `sopc_edge_detector_interface` module to include it in your Nios II system (Figure 7–16).

Figure 7–16. Including Your DSP Builder Avalon-MM FIFO Design Module in SOPC Builder



You can now design the rest of your NIOS embedded processor using the standard SOPC Builder design flow.

See "Instantiate Your Design in SOPC Builder" on page 7–19 in the "Avalon-MM Interface Blocks Walkthrough" for more detailed instructions.

8. Using the State Machine Library

Introduction

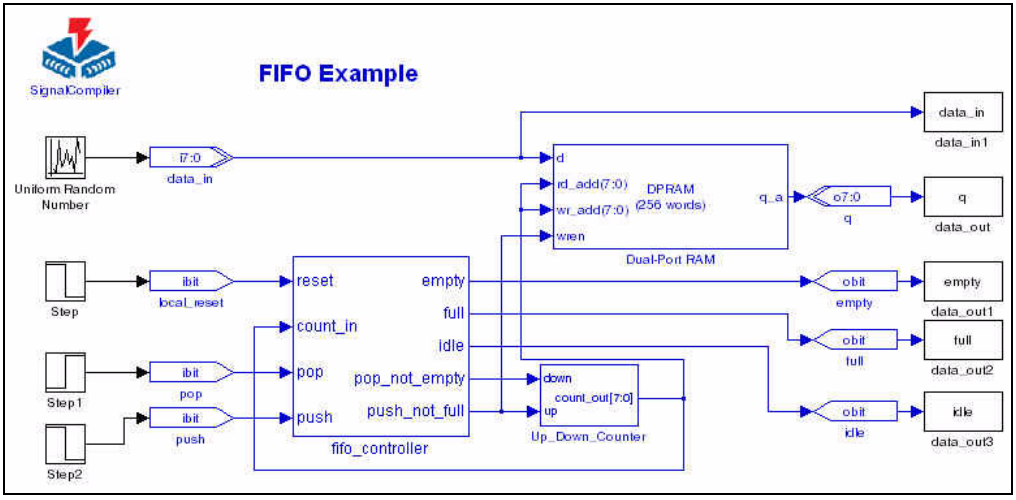
This chapter describes the design flow used to implement a state machine in DSP Builder.

The example design, **fifo_control_logic.mdl**, contains a simple state machine used to implement the control logic for a first-in first-out (FIFO) memory structure.

The design files for this example are installed in *<DSP Builder install path>***DesignExamples\Tutorials\StateMachine\StateMachineTable** directory.

Figure 8–1 shows the top-level schematic for the FIFO design example.

Figure 8–1. FIFO Design Example Top-Level Schematic



The state machine in this design example feeds the control inputs of a Dual-Port RAM block and the inputs of an address counter.

The operation of the state machine is as follows:

- When the push input is asserted and the address counter is less than 250, the address counter is incremented and a byte of data is written to memory.
- When the pop input is asserted and the address counter is greater than 0, the address counter is decremented and a byte of data is read from memory.
- When the address counter is equal to 0, the empty flag is asserted.
- When the address counter is equal to 250, the full flag is asserted.

Table 8–1 shows the state transition table for the FIFO controller state machine.

Table 8–1. FIFO Controller State Machine

Current State	Condition	Next State
empty	(push=1)&(count_in!=250)	push_not_full
empty	(push=0)&(pop=0)	idle
full	(push=0)&(pop=0)	idle
full	(pop=1)	pop_not_empty
idle	(pop=1)&(count_in=0)	empty
idle	push=1	push_not_full
idle	(pop=1)&(count_in!=0)	pop_not_empty
idle	(push=1)&(count_in=250)	full
pop_not_empty	(push=0)&(pop=0)	idle
pop_not_empty	(pop=1)&(count_in=0)	empty
pop_not_empty	(push=1)&(count_in!=250)	push_not_full
pop_not_empty	(pop=1)&(count_in!=0)	pop_not_empty
pop_not_empty	(push=1)&(count_in=250)	full
push_not_full	(push=0)&(pop=0)	idle
push_not_full	(pop=1)&(count_in=0)	empty
push_not_full	(push=1)&(count_in!=250)	push_not_full
push_not_full	(push=1)&(count_in=250)	full
push_not_full	(pop=1)&(count_in!=0)	pop_not_empty

State Machine Walkthrough

The design flow using the `State Machine Table` block involves the following steps:

1. Add a `State Machine Table` block to your Simulink design and assign it a new name. Figure 8–2 shows the default `State Machine Table` block. In this example, the block is named **fifo_controller**.

Figure 8–2. fifo_controller State Machine Block

You must save you model and change the default name of the `State Machine Table` block before you define the state machine properties.

2. Double-click the `fifo_controller` block to define the state machine properties.

   The State Machine Builder dialog box appears with the **Inputs** tab selected. The **Inputs** tab displays the input names defined for your state machine and provides an interface to allow you to add, and delete input names.

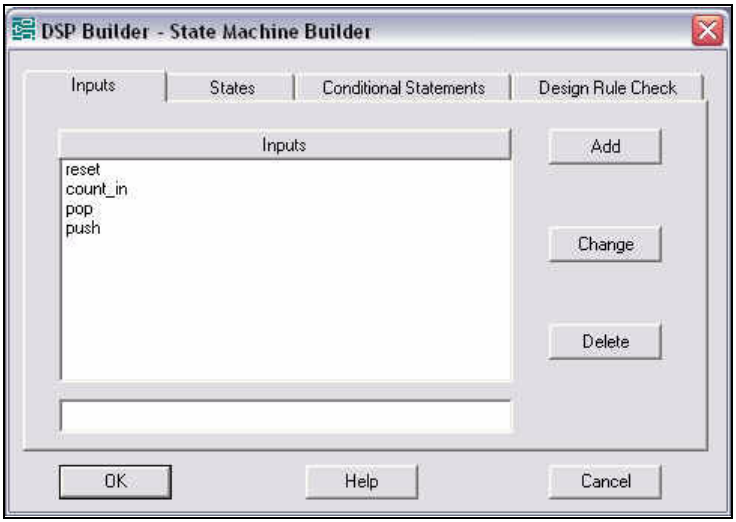
3. Delete the default input names `In2`, `In3`, `In4`, and `In5` and enter the following new input names:

   - `count_in`
   - `pop`
   - `push`

You can add or delete inputs but you cannot change an existing input name directly. You cannot delete or change the `reset` input.

Figure 8–3 on page 8–4 shows the **Inputs** tab after the inputs have been defined for the FIFO design example.

Figure 8–3. State Machine Builder Inputs Tab



4. Click the **States** tab.

   The **States** tab displays the state names defined for your state machine and provides an interface to allow you to add, change, and delete state names.

   The **States** tab also allows you to select the reset state for your state machine. The reset state is the state to which the state machine transitions when the reset input is asserted.

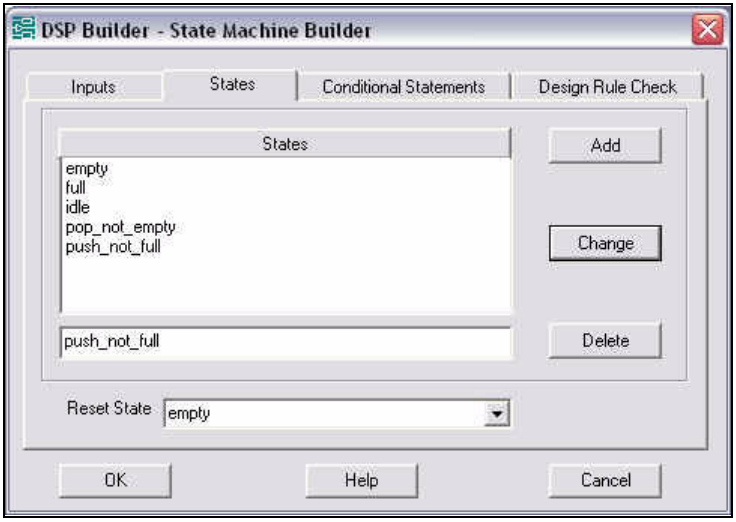
   You must define at least two states for the state machine. You cannot delete or change the name of a state while it is selected as the reset state.

5. Use the **Add**, **Change**, and **Delete** buttons to replace the default states `S1`, `S2`, `S3`, `S4`, and `S5` with the following states:

   - `empty` (reset state)
   - `full`
   - `idle`
   - `pop_not_empty`
   - `push_not_full`

Figure 8–4 shows the State Machine Builder **States** tab after the states have been edited for the FIFO design example.

Figure 8–4. State Machine Builder States Tab



6. After specifying the input and state names, click the **Conditional Statements** tab and use it to describe the behavior of your state machine by adding the statements shown in Table 8–1 on page 8–2.

   The **Conditional Statements** tab displays the state transition table, which contains the conditional statements that define your state machine.

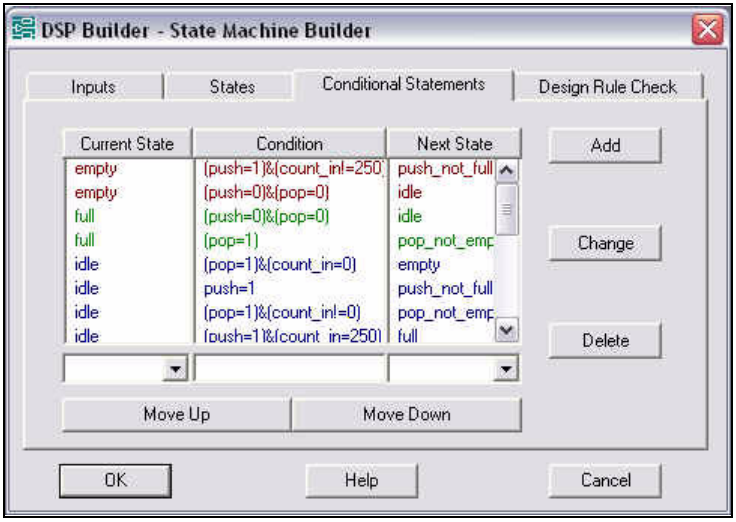
   There must be at least one conditional statement defined in the **Conditional Statements** tab.

   A conditional statement consists of a current state, a condition that causes a transition to take place, and the next state to which the state machine transitions. The current state and next state values must be state names defined in the **States** tab and can be selected from a drop down list in the dialog box.

   To indicate in a conditional statement that a state machine always transitions from the current state to the next state, specify the conditional expression to be one.

Figure 8–5 on page 8–6 shows the **Conditional Statements** tab, after defining the conditional statements for the FIFO controller.

Figure 8–5. State Machine Builder Conditional Statements Tab



When a state machine is in a particular state, it may have to evaluate more than one condition to determine the next state to which it transitions. If the conditions contain a single operator, the priority is determined by the priority of the conditional operator.

Table 8–2 shows the conditional operators that can be used to define a conditional expression.

Table 8–2. Comparison Operators Supported in Conditional Expressions

Operator	Description	Priority	Example
- (unary)	Negative	1	-1
(...)	Brackets	1	(1)
=	Numeric equality	2	in1=5
!=	Not equal to	2	in1!=5
>	Greater than	2	in1>in2
>=	Greater than or equal to	2	in1>=in2
<	Less than	2	in1<in2
<=	Less than or equal to	2	in1<=in2

Table 8–2. Comparison Operators Supported in Conditional Expressions			
Operator	**Description**	**Priority**	**Example**
&	AND	2	(in1=in2)&(in3>=4)
|	OR	2	(in1=in2)|(in1>in2)

If the conditions contain multiple operators they are evaluated in the order that you list them in the conditional statements table.

Table 8–3 shows the conditional statements when the current state is `idle`.

The condition `(pop=1) & (count_in=0)` is higher in the table than the condition `(push=1) & (count_in=250),` therefore it has higher priority.

The condition `(pop=1) & (count_in!=0)` has the next highest priority and the condition `(push=1) & (count_in=250)` has the lowest priority.

Table 8–3. Idle State Condition Priority		
Current State	**Condition**	**Next State**
idle	(pop=1)&(count_in=0)	empty
idle	push=1	push_not_full
idle	(pop=1)&(count_in!=0)	pop_not_empty
idle	(push=1)&(count_in=250)	full

The states in Table 8–3 generate the following VHDL code:

```
IF ((pop_sig=1) AND (count_in_sig=0)) THEN
   next_state <= empty_st;
ELSIF (push_sig=1) THEN
   next_state <= push_not_full_st;
ELSIF ((pop_sig=1) AND (count_in_sig/=0)) THEN
   next_state <= pop_not_empty_st;
ELSIF ((push_sig=1) AND (count_in_sig=250)) THEN
   next_state <= full_st;
ELSE
   next_state <= idle_st;
END IF;
```

The extension `_sig` is automatically added to the input names in the VHDL file.

7. Use the **Move Up** and **Move Down** buttons to change the order of the conditional statements, as shown in Table 8–4.

Table 8–4. Idle State Condition Priority (Reordered)

Current State	Condition	Next State
idle	(pop=1)&(count_in=0)	empty
idle	(push=1)&(count_in=250)	full
idle	(pop=1)&(count_in!=0)	pop_not_empty
idle	push=1	push_not_full

If you move the condition `push=1` down, the State Machine Builder generates the VHDL code shown below:

```
IF ((pop_sig=1) AND (count_in_sig=0)) THEN
   next_state <= empty_st;
ELSIF ((pop_sig=1) AND (count_in_sig/=0)) THEN
   next_state <= pop_not_empty_st;
ELSIF ((push_sig=1) AND (count_in_sig=250)) THEN
   next_state <= full_st;
ELSIF (push_sig=1) THEN
   next_state <= push_not_full_st;
ELSE
   next_state <= idle_st;
END IF;
```

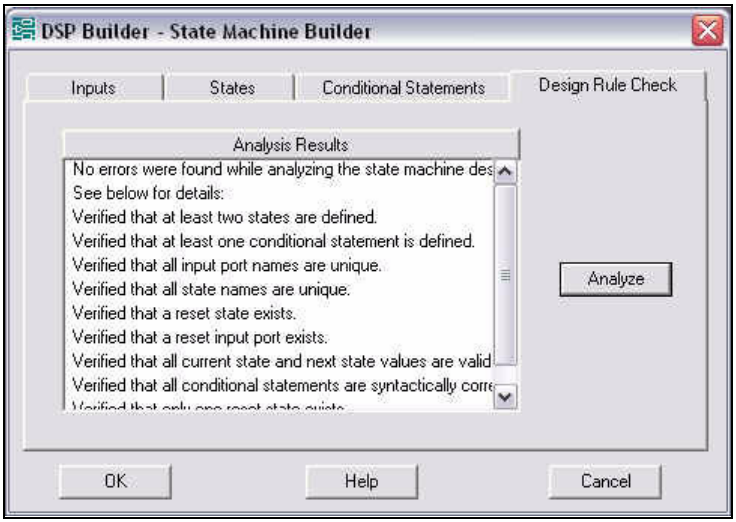
8. Click the **Design Rule Check** tab. You can use this tab to verify that the state machine you defined in the previous steps does not violate any of the design rules. Click **Analyze** to evaluate the design rules for your state machine.

   If a design rule is violated, an error message, highlighted in red, is listed in the **Analysis Results** box.

   If error messages appear in the analysis results, fix the errors and re-run the analysis until no error messages appear before simulating and generating VHDL for your design.

   Figure 8–6 on page 8–9 shows the **Design Rule Check** tab after clicking **Analyze**.

Figure 8–6. State Machine Builder Design Rule Check Tab



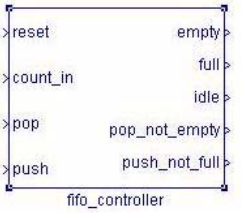
9. To save the changes made to your state machine, click **OK**.

   The State Machine Builder dialog box closes and returns you to your Simulink design file. The design file is automatically updated with the input and output names defined in the previous steps.

You may need to resize the block to ensure that the input and state names do not overlap and are displayed correctly.

Figure 8–7 shows the updated `fifo_controller` block for the FIFO design example.

Figure 8–7. fifo_controller Block After Specifying the State Machine



10. Connect the `fifo_controller` block to the rest of your design.

9. Using Black Boxes for Non-DSP Builder Subsystems

Introduction

The `Signal Compiler` block converts subsystems described using blocks from the DSP Builder block libraries into HDL code. Non-DSP Builder blocks, such as encapsulations of your own pre-existing HDL code, require the `Signal Compiler` block to recognize them as black boxes so that they are not altered by the conversion process.

There are two types of black box interface in DSP Builder: implicit and explicit.

Implicit Black Box Interface

The implicit black box interface can be inferred by using the `HDL Import` block.

The `Signal Compiler` block recognizes the `HDL Import` block as a black box and bypasses this block during the HDL translation.

For information on the `HDL Import` block, refer to the block description in the *AltLab Library* chapter of the *DSP Builder Reference Manual*.

Explicit Black Box Interface

The explicit black box interface is specified using the `HDL Input`, `HDL Output`, `HDL Entity`, and `Subsystem Builder` blocks.

Using the `HDL Input`, `HDL Output`, and `HDL Entity` blocks prevents `Signal Compiler` from translating the subsystem into HDL. You can also use a `Subsystem Builder` block to create a new subsystem and then automatically populate its ports using the specified HDL.

You would typically use the explicit black box interface to encapsulate non-DSP Builder blocks from the main Simulink blocksets.

For information on the `HDL Input`, `HDL Output`, `HDL Entity`, and `Subsystem Builder` blocks, refer to the block descriptions in the *AltLab Library* chapter of the *DSP Builder Reference Manual*.

HDL Import Walkthrough

The `HDL Import` block provides an interface to import a HDL module into your DSP Builder design.

Imported VHDL must be defined using *std_logic_1164* types. If your design uses any other VHDL type definitions (such as arithmetic or numeric types), you should write a wrapper which converts them to *std_logic* or *std_logic_vector*.

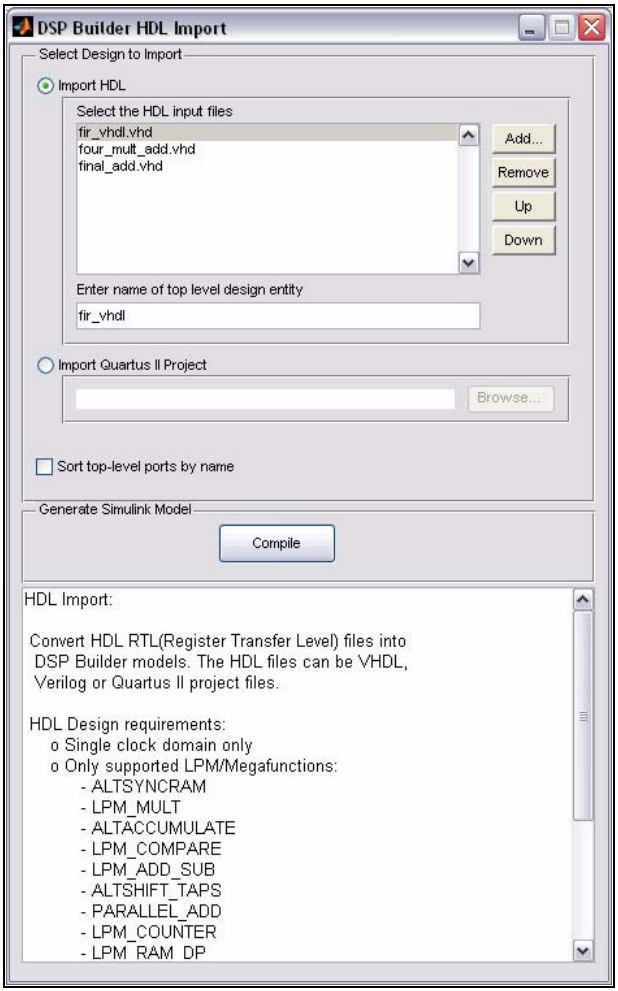
The following sections show an example of importing an existing HDL design written in VHDL into the DSP Builder environment using the `HDL Import` block.

Import Existing HDL Files

To import existing HDL files into a DSP Builder design, perform the following steps in the Simulink software:

1. In MATLAB, change the current directory setting to: <*DSP Builder install path*>**\DesignExamples\Tutorials\BlackBox\HDLImport**
2. Choose **Open** (File menu) and select **empty_MyFilter.mdl**. This design file has some of the peripheral blocks instantiated including the input/output ports and source blocks that provide appropriate stimulus for simulation. It is missing the main filter function which you will import as HDL.
3. Rename the file by choosing **Save As** (File menu). Name your new MDL file **MyFilter.mdl**.
4. Open the Simulink Library Browser by clicking on the  icon or by typing `simulink` at the MATLAB command prompt.
5. In the Simulink Library Browser, expand the **Altera DSP Builder Blockset** and select the AltLab library.
6. Drag and drop a `HDL Import` block into the model.
7. Double-click on the `HDL Import` block to bring up the DSP Builder HDL Import dialog box (Figure 9–5 on page 9–8).
8. In the HDL Import dialog box, enable the **Import HDL** radio button and click on the **Add** button to select the HDL input files.
9. From the **VHDL Black Box File** dialog box, select **fir_vhdl.vhd**, **four_mult_add.vhd**, and **final_add.vhd** files then click on **Open**.

Figure 9–1. HDL Import Dialog Box



10. Ensure that `fir_vhdl` is specified as the name of the top-level design entity. The **fir_vhdl.vhd** file describes the top level entity which implements an 8-tap low-pass FIR filter design.

11. Turn on the option to **Sort top-level ports by name**.

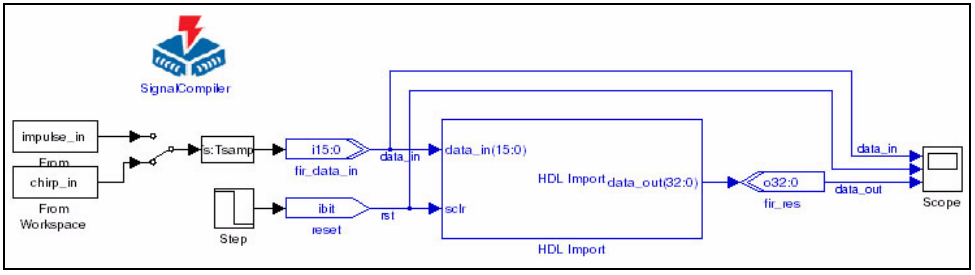
12. In the Generate Simulink Model section, click on **Compile** to generate a Simulink simulation model for the imported HDL design.

13. Once generation is completed, a window should appear with the message `HDL Import complete`. Click **OK** to close this window.

    The HDL Import window is closed automatically and the `HDL Import` block in the **MyFilter** window is updated to show the ports defined in the imported HDL.

14. Connect the input and output ports to the symbol, as shown in Figure 9–2. The code generated for the `HDL Import` block is automatically black boxed.

Figure 9–2. Completed Design



15. Choose **Save** (File menu) to save the **MyFilter.mdl** file.

Simulate the HDL Import Model using Simulink

Perform the following steps to run simulation in Simulink:

1. Double-click on the manual switch connected to the `Tsamp` block which feeds into the `fir_data_in` input port.

   This toggles the switch and sets the `impulse_in` stimulus which is used to verify the impulse response of the low-pass filter.

2. Click on the Start Simulation icon or select **Start** from the Simulation menu in the model window.

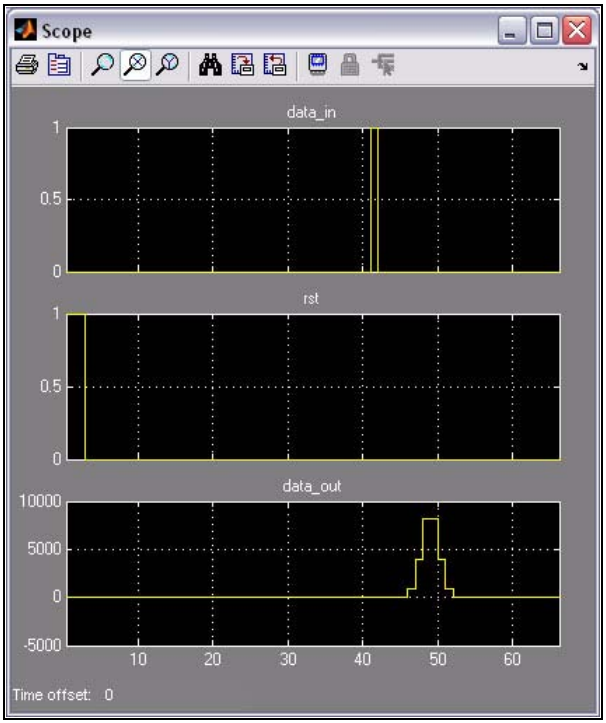
3. Double-click on the `Scope` block to view the simulation results.

4. Click the Autoscale icon to resize the scope. This scales both axes to display all stored simulation data until the end of the simulation (which is set to 500*Tsamp for this model).

5. Click the Zoom X-axis icon and drag the cursor to zoom in on the first 70 X-axis time units.

The simulation results should look similar to Figure 9–3.

Figure 9–3. Simulink Simulation Results for the Impulse Stimulus



6. Double-click on the manual switch connected to the `Tsamp` block to select the *chirp_in* stimulus. (This is a sinusoidal signal whose frequency increases at a linear rate with time.)

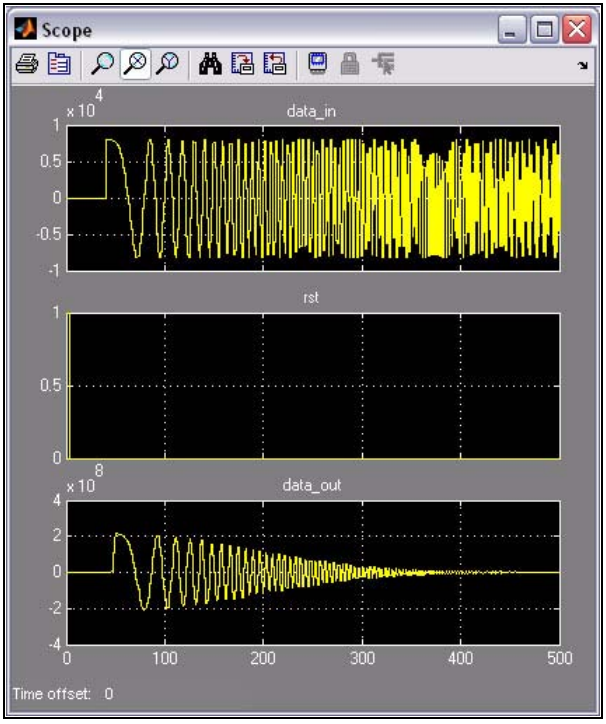
7. Click on the Start Simulation icon or select **Start** from the Simulation menu in the model window.

8. Double-click on the `Scope` block to view the simulation results.

9. Press the Autoscale icon to resize the scope.

The simulation results should look similar to Figure 9–4.

Figure 9–4. Simulink Simulation Results for the Chirp Stimulus



This completes the HDL Import walkthrough. You can optionally compile your model for synthesis or perform RTL simulation on your design by following similar procedures to those described in the "Getting Started Tutorial".

Subsystem Builder Walkthrough

The `Subsystem Builder` block makes it easy for you to import the input and output signals for a VHDL or Verilog HDL design into a Simulink subsystem.

If your HDL design contains any LPM or Megafunctions that are not supported by the `HDL Import` block, you should use the `Subsystem Builder` block. The `Subsystem Builder` block also allows you to create your own Simulink simulation model from non-DSP Builder blocks for faster simulation speed.

Unlike the `HDL Import` block described in the previous section, the `Subsystem Builder` block does not create a Simulink simulation model for the imported HDL design.

For more information on the `Subsystem Builder` block, refer to the block description in the *AltLab Library* chapter in the *DSP Builder Reference Manual*.

In addition to porting the HDL design to a Simulink subsystem, you need to create the Simulink simulation model for the block. The simulation models describes the functionality of the particular HDL subsystem. The following list shows the options available to create Simulink simulation models:

- Simulink generic library
- Simulink Blocksets (such as the DSP and Communications blocksets)
- DSP Builder blockset
- MATLAB functions
- S-functions

You need to add a `Non-synthesizable Input` block and a `Non-synthesizable Output` block around any DSP Builder blocks in the subsystem.

The following section shows an example which uses an S-function to describe the simulation models of the HDL code.

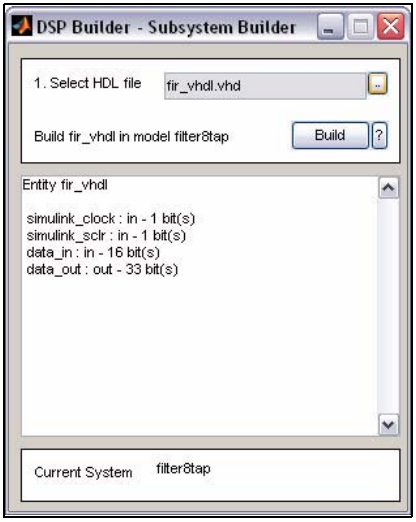
Create a Black Box System

To create a black box system, perform the following steps:

1. In MATLAB, change the current directory to: <*DSP Builder install path*>**\DesignExamples\Tutorials\BlackBox\SubSystemBuilder**
2. Choose **Open** (File menu). Select the **filter8tap.mdl** file and click **OK**.
3. Open the Simulink Library Browser by clicking the Simulink icon or typing `simulink` at the MATLAB command prompt.
4. In the Simulink Library Browser, expand the AltLab library under the **Altera DSP Builder blockset**.
5. Drag a `Subsystem Builder` block into your model.

6. Double-click the `Subsystem Builder` block. The Subsystem Builder dialog box is displayed (Figure 9–5).

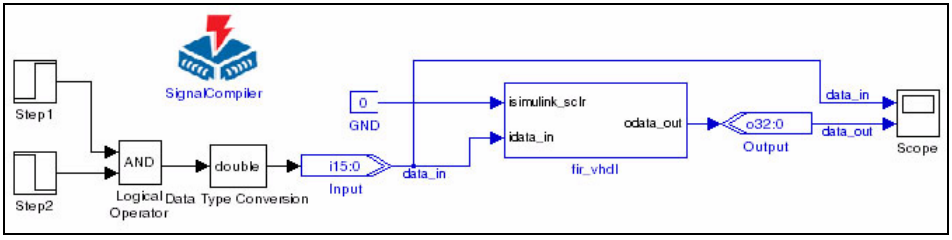
Figure 9–5. Subsystem Builder Dialog Box



7. In the dialog box, browse for the **fir_vhdl.vhd** file and click on the **Build** button. This builds the subsystem and adds the signals for the `fir_vhdl` subsystem to the symbol in your **filter8tap.mdl** model. The Subsystem Builder dialog box is automatically closed.

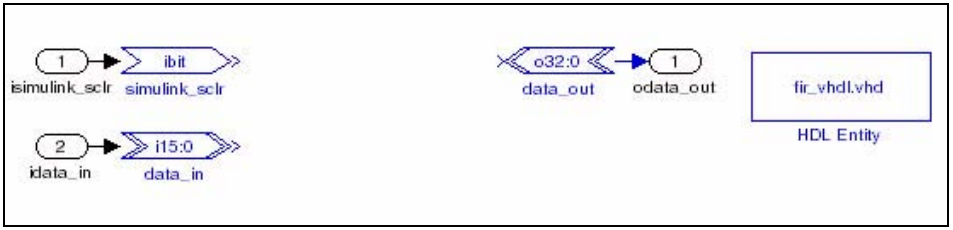
8. Connect the ports as shown in Figure 9–6.

Figure 9–6. filter8tap Design



9. Double-click on the `fir_vhdl` symbol. The **filter8tap/fir_vhdl** subsystem is opened (Figure 9–7).

Figure 9–7. Library: filter8tap/fir_vhdl Window



The **filter8tap/fir_vhdl** subsystem contains two `HDL Input` blocks (`simulink_sclr` and `data_in`) and a `HDL Output` block (`data_out`). Each of these blocks is in turn connected to a subsystem input or output. A `HDL Entity` block is also created to store the name of the HDL file and the names of the clock and reset ports.

No port is created in the subsystem for the clock since this is handled implicitly.

10. Leave the **filter8tap/fir_vhdl** window open for use in the next section.

    In the next section, you build the simulation model that represents the functionality of this block in your Simulink simulations.

Build the Black Box SubSystem Simulation Model

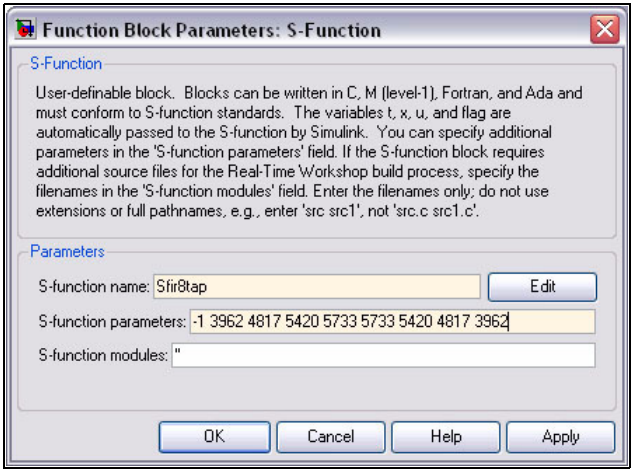
For this example, you use a S-function C++ simulation model to represent the 8-tap FIR filter block created in the previous section.

To create the model, perform the following steps:

1. In the Simulink Library Browser, expand the Simulink folder.
2. From the User-Defined Functions library, drag and drop a S-Function block into the **MyFilter/fir_vhdl** window.

3. Double-click the S-function block to display the Function Block Parameters: S-function dialog box (Figure 9–8).

Figure 9–8. Block Parameters: S-Function Dialog Box



4. In the Block Parameters dialog box, change the S-Function name to `Sfir8tap` and enter the parameters `-1 3962 4817 5420 5733 5733 5420 4817 3962`.

   The `Sfir8tap` function is a C++ Simulink S-Function simulation model written for the 8-tap Fir filter block. The first parameter refers to the sampling rate (-1 indicates it inherits the sampling rate from the preceding block) and the rest of the parameters represent the eight filter coefficients.

   The **S-function modules** parameter should be left with its default value.

5. Click the **Edit** button to view the code that describes the S-Function.

   If the code does not appear automatically, click **Browse** and select the **Sfir8tap.CPP** file.

6. Scroll down in the **Sfir8tap.CPP** file to the S-function methods section.

The following is an excerpt of the Simulink C++ S-Mex function code which can be used to design a Simulink filter simulation model:

```
/*====================*
 * S-function methods *
 *====================*/

/* Function: mdlInitializeSizes ===============================================
 * Abstract:
 *    The sizes information is used by Simulink to determine the S-function
 *    block's characteristics (number of inputs, outputs, states, etc.).
 */

static void mdlInitializeSizes(SimStruct *S)
{
    /* See sfuntmpl.doc for more details on the macros below */
    ssSetNumSFcnParams(S, 9);  /* Number of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        /* Return if number of expected != number of actual parameters */
        return;
    }
    // Set DialogParameters not tunable
    const int iMaxssGetSFcnParamsCount = ssGetSFcnParamsCount(S);
    for (int p=0;p<iMaxssGetSFcnParamsCount;p++){ssSetSFcnParamTunable(S, p,  0);}

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, 0, 1);
    ssSetInputPortDataType(S,  0, SS_DOUBLE);


    if (!ssSetNumOutputPorts(S, 1)) return;
    ssSetOutputPortWidth(S, 0, 1);
    ssSetOutputPortDataType(S, 0, SS_DOUBLE);


    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, 1);

    ssSetNumSampleTimes(S, 1);
    ssSetNumRWork(S, 0);
    ssSetNumIWork(S, 0);
    ssSetNumDWork(S, DYNAMICALLY_SIZED); // reserve element in the pointers vector
    ssSetNumModes(S, 0); // to store a C++ object
    ssSetNumNonsampledZCs(S, 0);
    ssSetOptions(S, 0);
}
```

During simulation, Simulink invokes certain callback methods from the S-function. The callback methods are sub-functions that initialize, update discrete states, and calculate output.

The callback methods used in the example design are described in Table 9–1.

Table 9–1. Callback Methods Used in the S-Function

Callback Method	Description
`mdlInitializeSizes`	Specify the number of inputs, outputs, states, parameters, and other characteristics of the S-function.
`mdlInitializeSampleTimes`	Specify the sample rates at which this S-function operates.
`mdlStart`	Initialize the vectors of this S-function.
`mdlOutputs`	Compute the signals that this block emits.
`mdlUpdate`	Update the states of the block.
`mdlTerminate`	Perform any actions required at termination of simulation.

7. At the MATLAB command prompt, type:

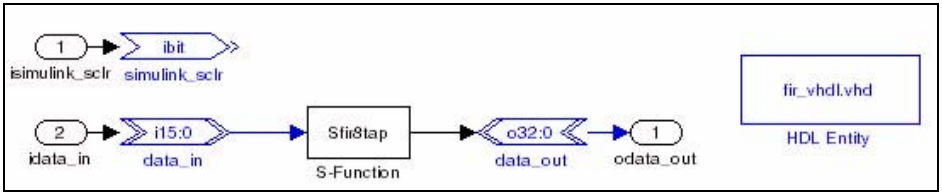
```
mex Sfir8tap.CPP
```

The `mex` command compiles and links the source file into a shared library executable from within MATLAB called **Sfir8tap.mexw32**. (The extension is specific to 32-bit version of MATLAB run in Windows).

8. Close the editor window and click on **OK** to close the Function Block Parameters dialog box.

9. In the **filter8tap/fir_vhdl** window, connect the input port of the S-function block to the `data_in` block, and connect the output port of the **S-function** block to the `data_out` block as shown in Figure 9–9.

Figure 9–9. S-Function Block Connection



You do not have to connect the `simulink_sclr` block. The `HDL Entity` block automatically maps any input ports named `simulink_clock` in the VHDL entity to the global clock signal, and any input ports named `simulink_sclr` in the VHDL entity to the global synchronous clear signal.

10. Choose **Save** (File menu) to save the **filter8tap.mdl** file.

Simulate the Subsystem Builder Model

Perform the following steps to run the Simulink simulation:

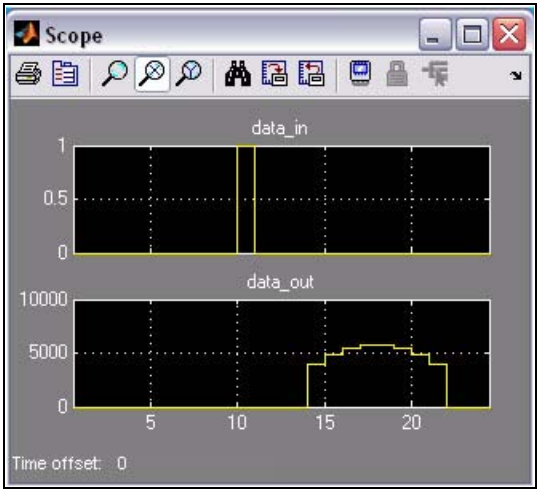
1. Click the Start Simulation icon or choose **Start** (Simulation menu) in the **filter8tap.mdl** window to begin the simulation.

2. Double-click the `Scope` block to view the simulation results. Click **Autoscale** to resize the scope.

3. Click the Zoom X-axis icon and use the cursor to zoom in on the first 22 x-axis time units.

   The simulation results should look similar to Figure 9–10.

Figure 9–10. Simulink Simulation Results of 8-Tap FIR Filter, Scope Window



Because the input is a pulse, the simulation results show the impulse response of the 8-tap FIR filter, which translates to the eight coefficient values. You can change the input stimulus to verify the step and random response of the filter.

Add VHDL Dependencies to the Quartus II Project and ModelSim

The VHDL file used in this example depends on two further VHDL files. As it currently stands, these two files are not examined by the Quartus II software or ModelSim and compilation with either will fail or give unexpected results. To resolve this, perform the following steps:

1. Either:
    a. Double-click on the `Signal Compiler` block and click **Compile**. (Ignore the result for now.)

       This creates a directory called **DSPBuilder_filter8tap_import** in the directory containing the design.

    b. Create the directory **DSPBuilder_filter8tap_import** directly.
2. Copy the **extra_add.tcl** and **extra_add_msim.tcl** files to the **DSPBuilder_filter8tap_import** directory. These can be found in the same directory as the original design.

The **extra_add.tcl** file adds **final_add.vhd** and **four_mult_add.vhd** to the Quartus II project, while **extra_add_msim.tcl** forces ModelSim to compile them when the design is run using the `TestBench` block. Any files ending with **_add.tcl** are executed by the Quartus II software when the project is created. Files ending with **_add_msim.tcl** are executed by ModelSim when it compiles the design testbench.

Simulate the Design in ModelSim

Perform the following steps to test the simulation model against the HDL in ModelSim:

1. In the Simulink Library Browser, expand AltLab library under **Altera DSP Builder Blockset**.
2. Drag a `TestBench` block into your model.

3. Double-click on the `TestBench` block and click **Compare against HDL** (Figure 9–11).

Figure 9–11. Testbench Generator Dialog Box for the filter8tap Design

When the comparison has completed, you should see an `Exact Match` message at the end.

If you want to use ModelSim directly, click on the **Advanced** tab, turn on the **Launch GUI** option, and then click **Run ModelSim**.

This completes the Subsystem Builder walkthrough. You can optionally compile your model for synthesis by following similar procedures to those described in the "Getting Started Tutorial".

10. Creating Custom Library Blocks

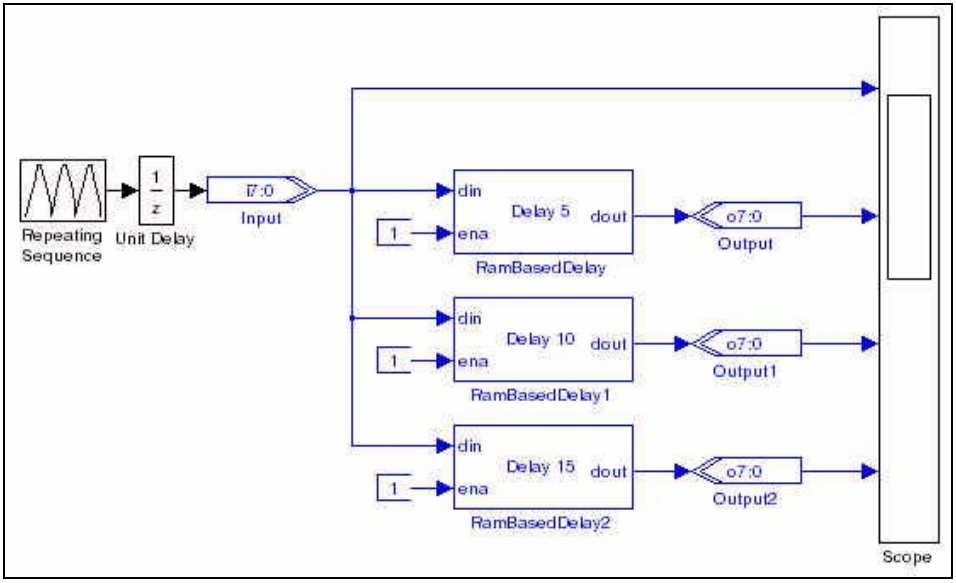
Introduction

This chapter describes how to build a parameterizable custom library block for use with Simulink and DSP Builder.

Custom blocks are Simulink subsystems in which the block functionality is described using DSP Builder primitives. This design flow also supports parameterizable hierarchical subsystem structures.

A completed example is provided in <*DSP Builder install path*>\ **DesignExamples\Tutorials\BuildingCustomLibrary\top.mdl**. (Figure 10–1).

Figure 10–1. top.mdl Example



The `RamBasedDelay` block used in **top.mdl**, is an example of a custom parameterizable Simulink block and is defined in the library file **MyLib.mdl**.

The `RamBasedDelay` block has one parameter, *Delay*.

Creating a Custom Library Block

To create your own custom block, perform the following steps:

Create a Library Model File

Create a Library Model File for your custom block by performing the following steps in the Simulink software:

1. In MATLAB, change the current directory setting to: *<DSP Builder install path>***DesignExamples****Tutorials****BuildingCustomLibrary**
2. Open the Simulink Library Browser by clicking the Simulink  icon or typing `simulink` at the MATLAB command prompt.
3. In the Simulink Library Browser, choose **New > Library** (File menu) to open a new library model window.
4. Expand the Simulink Ports & Subsystems library in the Simulink Library Browser and drag a `Subsystem` block into your model.
5. Click on the `Subsystem` text below the block and rename the block `DelayFIFO`.

   You should always rename a block representing an HDL Subsystem to ensure that all the generated entities in a hierarchical design are unique.
6. Choose **Save** (File menu) and save the library file as **NewLib.mdl**.

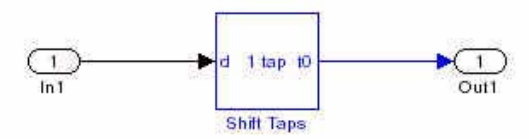
Build the HDL Subsystem Functionality

To add functionality to the `DelayFIFO` block, perform the following steps:

1. Double-click on the `DelayFIFO` block to open the **NewLib/DelayFIFO** subsystem window.

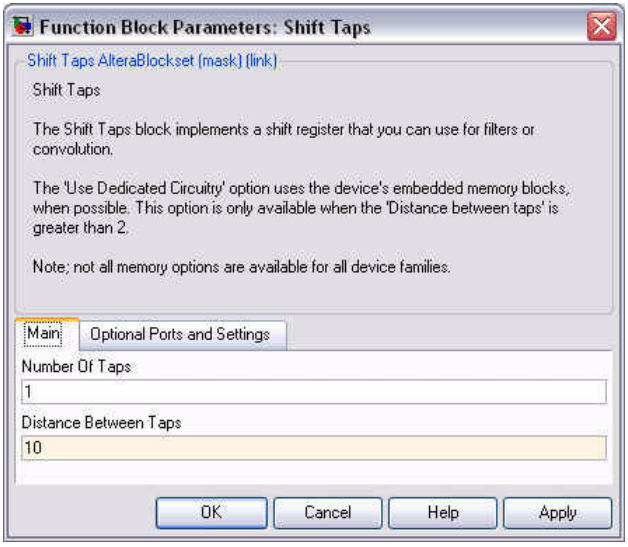
2. Drag and drop a `Shift Taps` block from the Storage library in the **Altera DSP Builder Blockset** into the **NewLib/DelayFIFO** window. Insert the `Shift Taps` block between the input and output blocks (Figure 10–2).

Figure 10–2. Shift Taps Block



3. Double-click the `Shift Taps` block to open the Block Parameters dialog box and set the following parameters in the **Main** tab (see Figure 10–3):
   - **Number of Taps**: `1`
   - **Distance Between Taps**: `10`

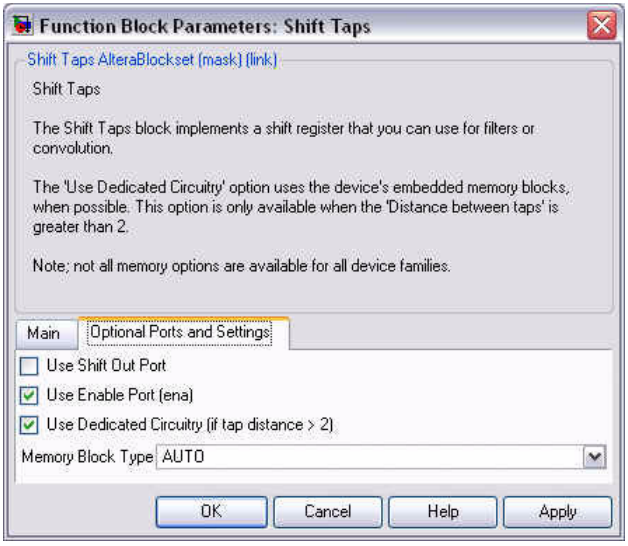
Figure 10–3. Shift Taps Block Parameters



4. Click the **Optional Ports and Settings** tab and set the following parameters (Figure 10–4):

   - **Use Shift Out Port**: `Off`
   - **Use Enable port**: `On`
   - **Use Dedicated Circuitry**: `On`
   - **Memory Block Type**: `Auto`

Figure 10–4. Shift Taps Block Optional Parameters



5. Click **OK** to close the Shift Taps Block Parameters dialog box.

6. Add an `Input` block (`In2`) from the Simulink Ports & Subsystems library and connect it to the `ena` port on the `Shift Taps` block.

7. Rename the blocks as shown below:

Old Name	New Name
`In1`	`InDin`
`In2`	`InEna`
`Shift Taps`	`DRB`
`Out1`	`OutDout`

8. Choose **Save** (File menu).

Figure 10–5 shows the completed DelayFIFO subsystem.

Figure 10–5. DelayFIFO Subsystem

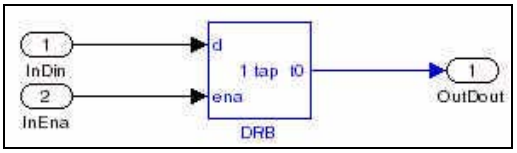


Figure 10–6 shows the NewLib library model which now shows the input and output ports defined in the DelayFIFO subsystem.

Figure 10–6. NewLib Model

Define Parameters Using the Mask Editor

Create parameters for the `DelayFIFO` block using the Mask Editor by performing the following steps:

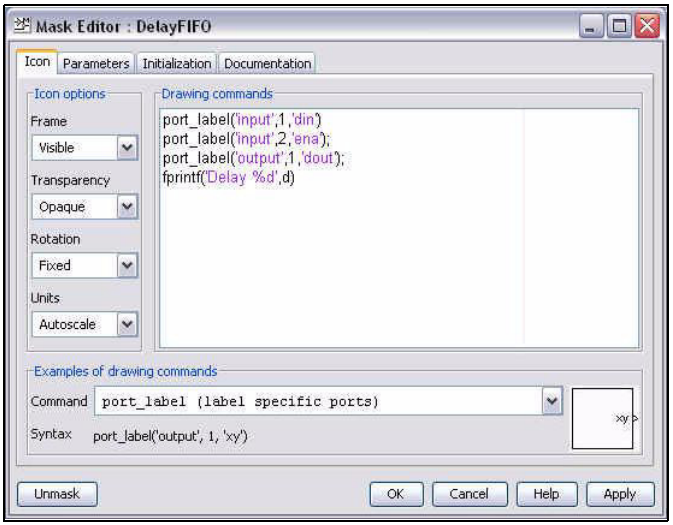
1. Right-click the `DelayFIFO` block in the NewLib model and choose **Mask Subsystem** from the pop-up menu.

2. Click the **Icon** tab in the Mask Editor dialog box (Figure 10–7 on page 10–6) and set the following icon options:

   - **Frame**: `Visible`
   - **Transparency**: `Opaque`
   - **Rotation**: `Fixed`
   - **Units**: `Autoscale`
   - **Drawing commands**:

```
port_label('input',1,'din');
port_label('input',2,'ena');
port_label('output',1,'dout');
fprintf('Delay %d',d)
```

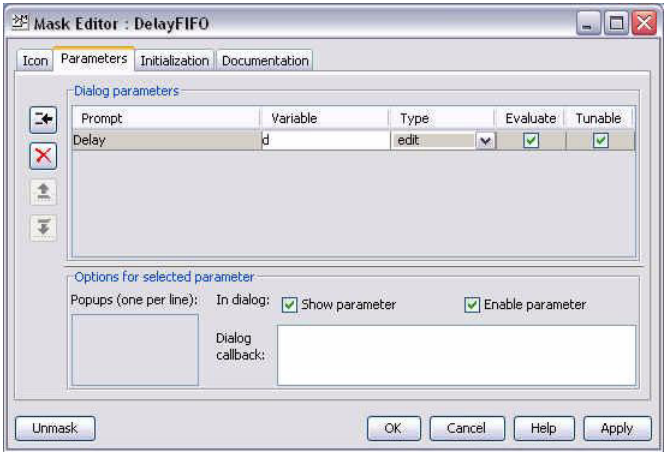
Figure 10–7. Mask Editor Icon Tab



3. Click the **Parameters** tab in the Mask Editor dialog box. Click the button and add the following parameter:

   - **Prompt:** `Delay`
   - **Variable:** `d`

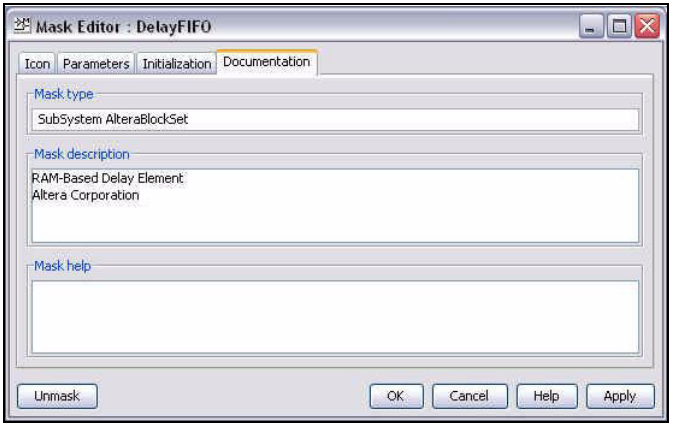
Figure 10–8. Mask Editor Parameters Tab



4. Click the **Documentation** tab (Figure 10–9) and add the following symbol documentation information:

   - **Mask type**:
     ```
     SubSystem AlteraBlockSet
     ```
   - **Mask description:**
     ```
     RAM-Based Delay Element
     Altera Corporation
     ```

Figure 10–9. Documentation Tab

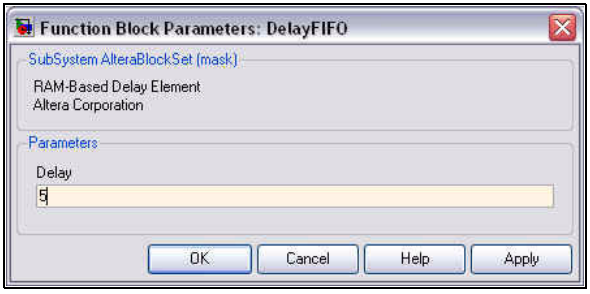


5. Click **OK** in the Mask Editor dialog box.

6. Double-click on the `DelayFIFO` block in your NewLib model to display the Block Parameters dialog box.

7. Set the **Delay** to 5 (Figure 10–10).

Figure 10–10. Delay FIFO Block Parameters



8. Click **OK** in the Block Parameters dialog box.

9. Choose **Save** (File menu) to save your library model.

For more information on the Mask Editor: In MATLAB, select Help > Full Product Help > Simulink > Using Simulink > Creating Masked Subsystems > The Mask Editor.

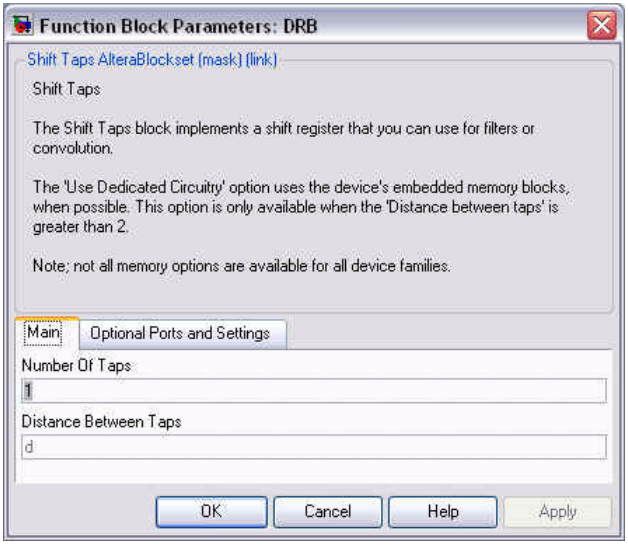
Link the Mask Parameters to the Block Parameters

To pass parameters from the symbol's mask into the block, you use a model workspace variable.

1. Double-click the `Shift Taps` block in the **NewLib/DelayFIFO** window to open the Block Parameters dialog box.

2. Copy the mask parameter variable name (d) from the parameters tab of the Mask Editor into the **Distance Between Taps** field in the Shift Taps Block Parameters dialog box. (Figure 10–11)

Figure 10–11. Shift Taps Block Parameters



3. Click **OK** to close the Shift Taps Block Parameters dialog box.

4. Close the **NewLib/DelayFIFO** window.

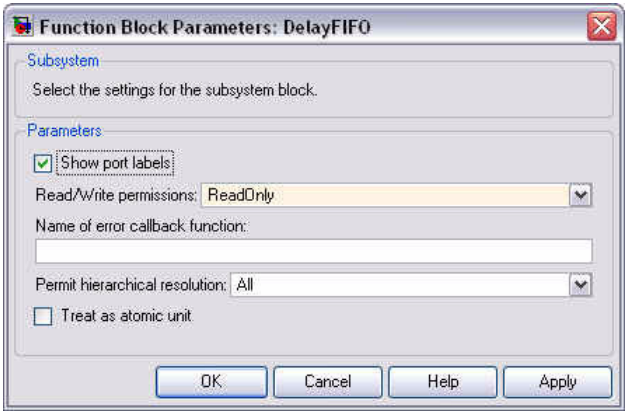
Make the Library Block Read Only

A library block should be made read only so that it is not accidentally edited from within a design model. To set the read/write permissions, perform the following steps:

1. Right-click the `DelayFIFO` block in the NewLib model and choose **SubSystem Parameters** from the pop-up menu.

2. Choose the `ReadOnly` option in the Block Parameters dialog box as shown in Figure 10–12.

Figure 10–12. Delay FIFO Block Parameters



The `ReadWrite` option would allow edits from both the library and the design. The `NoReadOrWrite` option would not allow `Signal Compiler` to generate HDL for the design. If you want to modify a library model, you can open the model, choose **Unlock Library** from the File menu and then change the read/write permissions in the block parameters dialog box. Remember to reset `ReadOnly` after changing the library model. Your changes are automatically propagated to all instances in the design.

3. Click **OK** to close the Block Parameters dialog box.

4. Choose **Save** (File menu) to save your library model.

Add the Library to the Simulink Library Browser

You can add a custom library to the Simulink library browser by creating a file called **slblocks.m**. This file must be in the same location as your library file and both files must be in search path for MATLAB. To create this file, perform the following steps:

1. In MATLAB, choose **New > M-File** (File menu) to open a new editor window.

2. Enter the following text in the editor window:

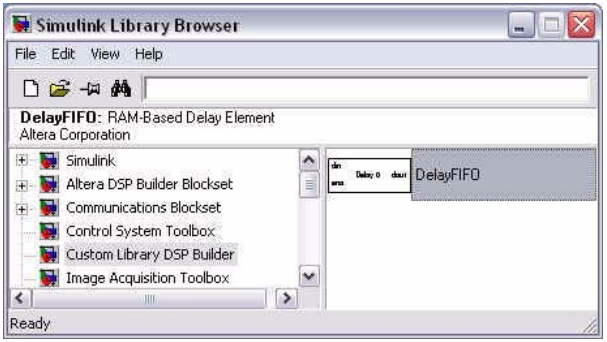
```
function blkStruct = slblocks

blkStruct.Name = ['Custom Library DSP Builder'];
blkStruct.OpenFcn = 'NewLib';
blkStruct.MaskDisplay = '';
% Define the Browser structure array, the first
% element contains the information for the Simulink
% block library and the second for the Simulink
% Extras block library.
Browser(1).Library = 'NewLib';
Browser(1).Name = 'Custom Library DSP Builder';
Browser(1).IsFlat = 0;
blkStruct.Browser = Browser;

% End of slblocks
```

3. Save the M-file with the file name **slblocks.m** in the same directory as **NewLib.mdl**. The next time that you display the Simulink library browser the Custom Library DSP Builder should be available as shown Figure 10–13.

Figure 10–13. Custom library in the Simulink Library Browser



You can drag and drop a block from your custom library in the same way as from any other library in the Simulink library browser.

You can create a custom library with multiple blocks by creating the required blocks in the same library file.

Refer to the MATLAB help for more information about M-files. A template **slblocks.m** file with explanatory comments can be found at *<MATLAB install path>***\toolbox\simulink\blocks\slblocks.m**

11. Using the Simulation Accelerator

Introduction

Fast functional simulation technology in Altera DSP Builder accelerates the simulation of supported Altera MegaCore functions by factors of ten to thirty times. This level of improvement is possible because simulation is performed at the transaction level. The transactions, in this case, are the transport of data over Avalon Streaming (Avalon-ST) and Avalon Memory-Mapped (Avalon-MM) ports.

Exactly the same sequence of Avalon transactions is performed by a fast simulation model and its gate-level simulation counterpart, the only differences lie in the modelling of latency and throughput. Gate level simulation yields fully cycle accurate behavior whereas the fast functional simulation approximates delays in the system to improve simulation speed.

The fast functional simulation flow enables high speed prototyping of complex systems where simulation would ordinarily be prohibitively slow. To obtain useful results from the simulation of a complex video processing system it is necessary to run multiple video frames through it, each of which could require tens of millions of clock cycles of simulated time. Simulating a large design for so long can be impractical. For example, running a single frame of video through the Altera Video and Image Processing Suite Example Design takes only ten minutes with fast functional simulation instead of around four hours using gate-level simulation. This speed increase opens the door to rapid prototyping and experimentation leading to higher image quality.

The following MegaCore functions support fast functional simulation:

- Scaler
- Color Space Converter
- Chroma Resampler
- Gamma Corrector
- Deinterlacer
- Alpha Blending Mixer
- 2D FIR Filter
- 2D Median Filter
- FFT (variable streaming architecture only)

Refer to the appropriate user guide for information about each of these MegaCore functions.

Fast Functional Simulation Design Flow

The `Simulation Accelerator` block in the Simulink AltLab library enables the fast functional simulation technology.

Follow these steps to use the `Simulation Accelerator` block:

1. Create a Simulink model which uses at least one supported MegaCore.
2. Add the `Simulation Accelerator` block to your Simulink model.
3. Run the simulation as normal.

To switch back to gate-level simulation it is not necessary to delete the `Simulation Accelerator` block, just double-click on the block to toggle it to Cycle Accurate mode.

Fast Functional Simulation Walkthrough

DSP Builder includes a design example in the <*DSP Builder install path*>**\DesignExamples\Tutorials\FastFuncSim** directory that demonstrates the use of fast functional simulation.

This section walks through the design, explaining its function and showing how to simulate it with fast functional simulation switched on and off. It assumes you have completed Chapter 2, Getting Started Tutorial, and are familiar with using DSP Builder, MATLAB and Simulink.

You must have MegaCore IP library v7.2 installed.

Load the Design

1. Run MATLAB, change your working directory to <*DSP Builder install path*>**\DesignExamples\Tutorials\FastFuncSim** and open the model **fast_sim_walkthrough.mdl**.

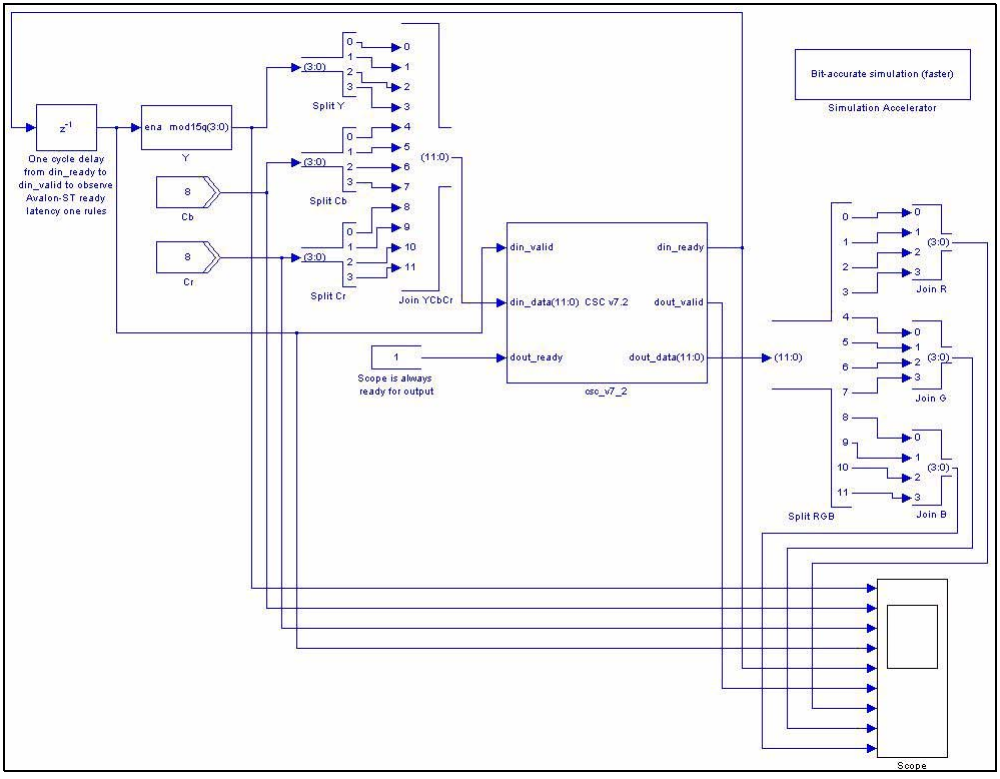
   Figure 11–1 on page 11–3 shows the loaded model.

If you were using the FFT MegaCore function, it would be necessary to run the `alt_dspbuilder_refresh_megacore` command to regenerate the files required for simulation. This is not required when you are using any of the Video and Image Processing Suite MegaCore functions such as the Color Space Converter used in this walkthrough.

Review the Design

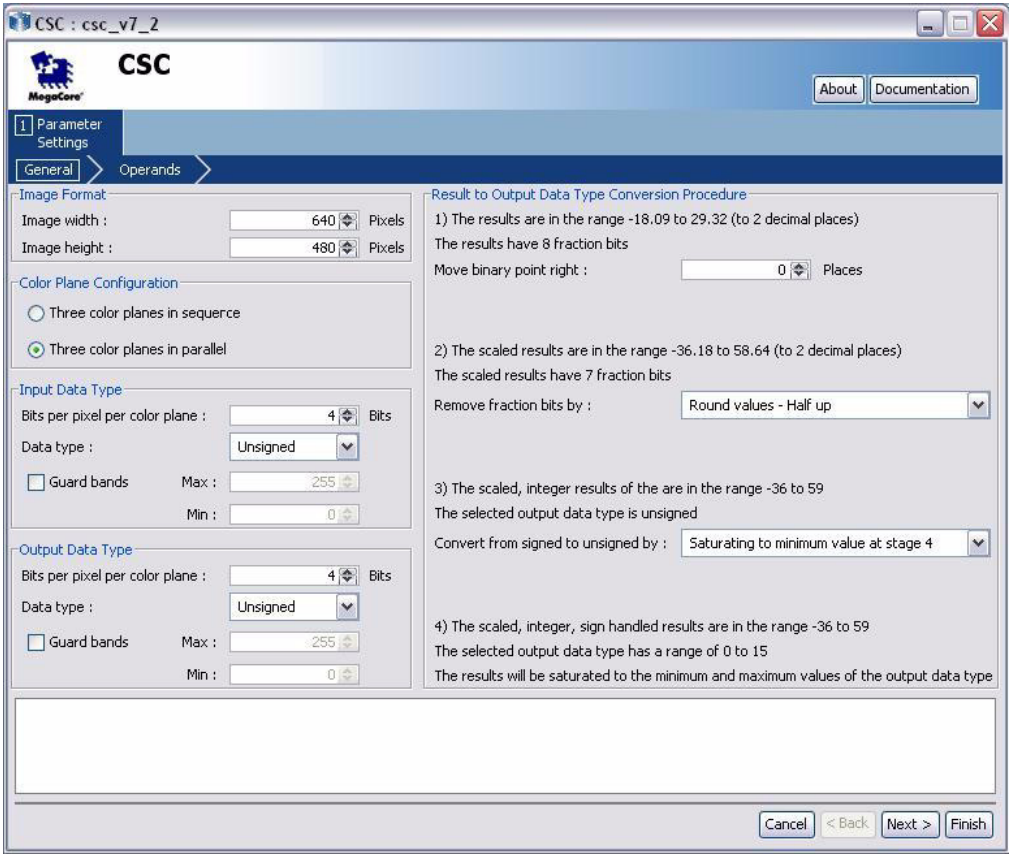
The design consists of a Color Space Converter MegaCore function connected to a simple source which feeds it with varying shades of grey. This design has been chosen for its simplicity and suitability for demonstrating the similarities and differences between fast functional simulation and cycle accurate simulation methods.

Figure 11–1. fast_sim_walkthrough.mdl Example



2. Double-click the csc_7_2 block to open the MegaWizard interface for the Color Space Converter MegaCore function (Figure 11–2 on page 11–4).

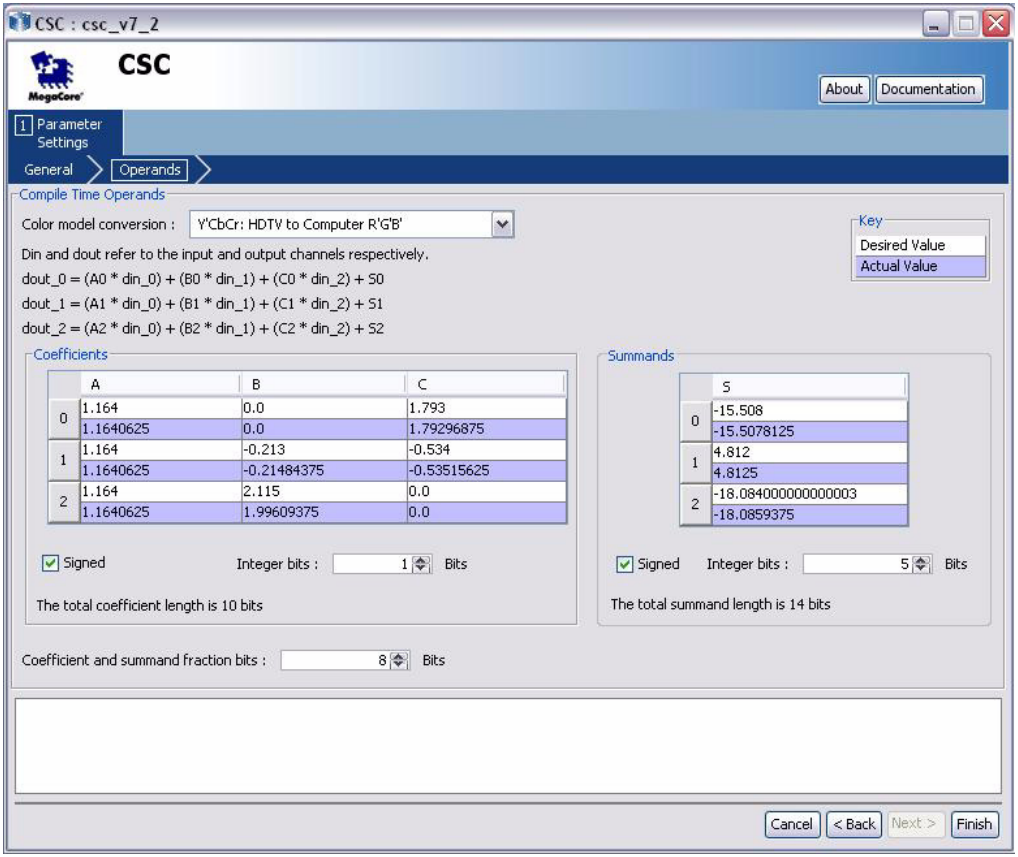
Figure 11–2. General Parameters for the Color Space Converter



Notice that the **Color Plane Configuration** has been set to `Three color planes in parallel` and the **Bits per pixel per color plane** for both input and output have been set to 4 bits. These settings mean that the Color Space Converter input and output ports are each 12 bits wide, accommodating three 4-bit color samples in each input or output word.

3. Click the **Operands** tab to review the second page of the Color Space Converter MegaWizard interface (Figure 11–3 on page 11–5).

Figure 11–3. Operands Tab for the Color Space Converter



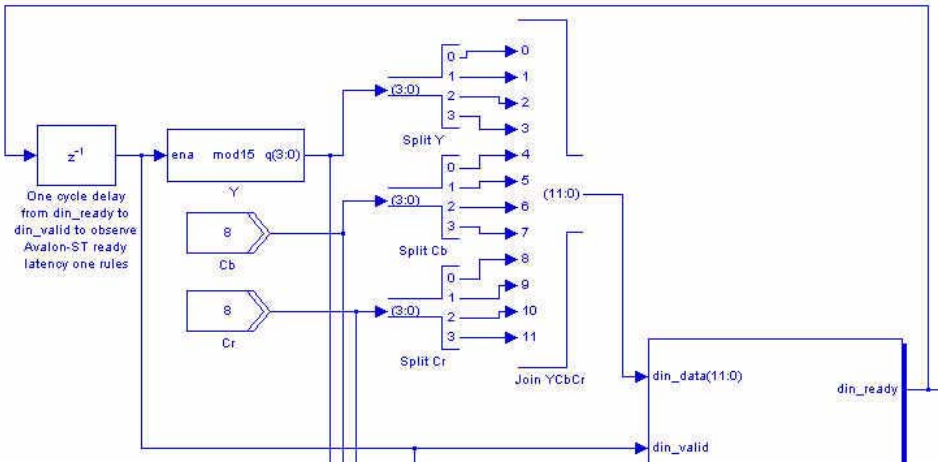
Note that a preset conversion from Y'CbCr to R'G'B has been selected. This means that the three input color planes are Y' (luminescence), Cb and Cr (color difference planes) and the output color planes are Red, Green and Blue.

For full details of the operation of the Color Space Converter MegaCore function, see the *Video and Image Processing Suite User Guide*.

4. Click **Cancel** to close the Color Space Converter MegaWizard interface without making any changes.

Examine the logic driving data into the Color Space Converter (Figure 11–4):

Figure 11–4. Color Space Converter Drive Logic



The MegaCore function has an Avalon-ST input for receiving data. Data is transferred through this port only on clock cycles where `din_valid` is high. The Color Space Converter indicates that it is ready to receive data on cycle *N*+1 by asserting `din_ready` on cycle *N*.

The rules of the Avalon-ST protocol state that it is illegal for `din_valid` to be driven high on cycles where the MegaCore function is not ready. The simple source feeds `din_ready` through one cycle of delay to drive `din_valid` high whenever it is allowed to.

For more information on the Avalon-ST Interface, refer to the *Avalon Streaming Interface Specification*.

Constants are used to drive the value 8 for both Cb and Cr. Each color component consists of four binary bits representing an unsigned number in the range 0-15, so the value 8 is the middle value. In the Y'CbCr color space, middle values of Cb and Cr indicate shades of grey.

Y' is produced by a counter which cycles through the values 0 to 15. This means that the overall effect is to cycle through sixteen shades of grey, starting with black and brightening to white before resetting. The counter has an enable input which means that it only counts on cycles where data is actually being transferred into the Color Space Converter MegaCore function, that is, cycles where `din_valid` is high.

Bus splitting and joining blocks are used to assemble the three four-bit color components into a single 12-bit bus for input to the Color Space Converter MegaCore function.

The remainder of the design splits the 12-bit data output from the Color Space Converter MegaCore function into separate 4-bit R, G and B components and attaches scopes to all relevant signals.

Run a Traditional Cycle-Accurate Simulation

1. Double-click the `Simulation Accelerator` block to switch to standard cycle-accurate simulation.

2. Double-click the `Scope` block to open a Scope window.

3. Click the Start Simulation button to run the simulation for 30 clock cycles. Observe the results in the Scope window (Figure 11–5 on page 11–8).

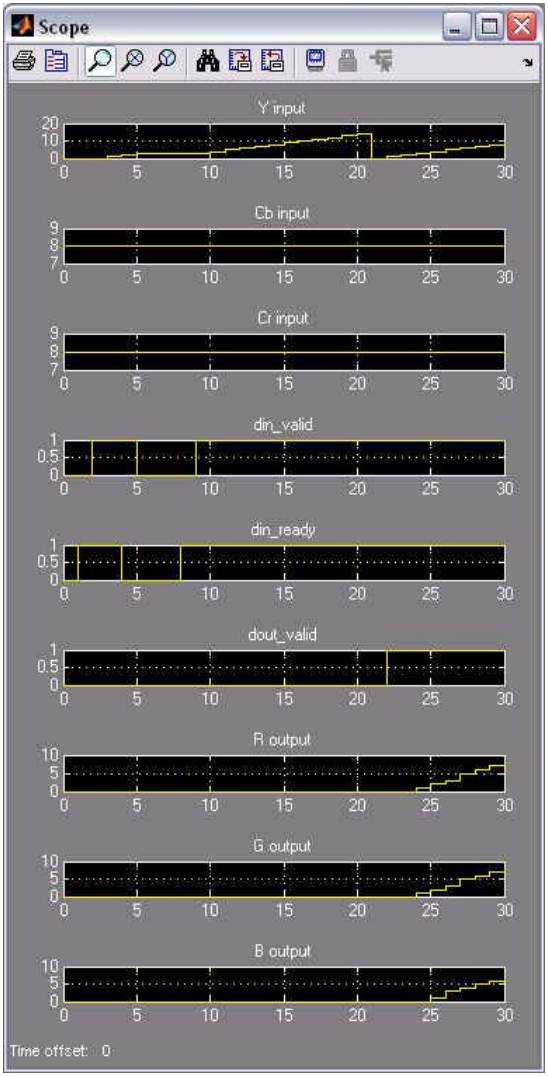
The Color Space Converter takes a few clock cycles to get initialize its internal state before it is ready to process data. Because this simulation is fully cycle accurate, these clock cycles are visible in the Scope:

- `din_ready` starts low. It is high in the second, third and fourth clock cycles, then drops low again. It goes high a second time in the tenth clock cycle and then stays high for the remainder of the simulation.
- `din_valid` follows the rules of the Avalon-ST protocol by shadowing the `din_ready` one clock cycle later. Data is only transferred into the Color Space Converter when `din_valid` is high.
- The counter driving input data Y only counts when `din_valid` is high too. This ensures that however many cycles the Color Space Converter pauses for, the same data is transmitted to it.

There are several clock cycles of latency caused by pipelining within the Color Space Converter. These can also be seen in the Scope:

- `dout_valid` starts low and stays low until the Color Space Converter is ready to start producing data. This happens during the 23rd clock cycle in the cycle accurate simulation.
- Avalon-ST transactions are only being performed when the valid signal is high, so the R, G and B components extracted from `dout_data` are irrelevant unless `dout_valid` is high. In this simulation, the first element of data is transferred out in the 23rd clock cycle and is a zero on all three color planes (black). The second data element is transferred on the 24th clock cycle is also a zero on all three color planes. The third element is transferred on the 25th clock cycle and is a `1` in the R and G planes and a `0` in the B plane.

Figure 11–5. Cycle Accurate Simulation Results

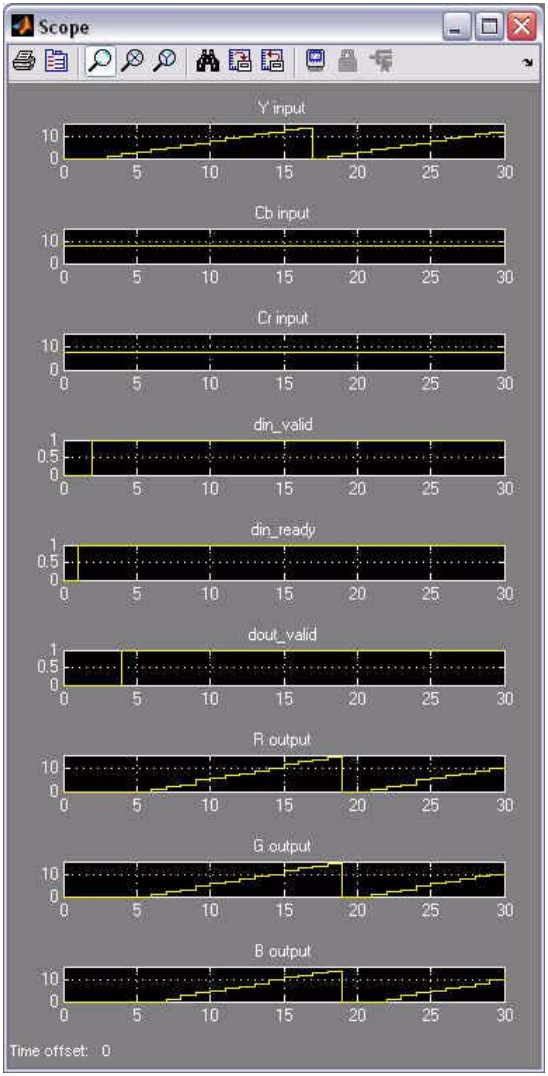


Run a Fast Functional Simulation

1. Double-click the `Simulation Accelerator` block to switch to fast functional (bit-accurate) simulation.

2. Click the Start Simulation button to run the simulation for 30 clock cycles. Observe the results in the Scope window:

Figure 11–6. Bit Accurate Simulation Results



At first glance these results look quite different. However, notice that the Y input is still only changing on cycles where `din_valid` is high, and that the first three valid output cycles are the same.

In fact, if you only examine the data which flows on the Avalon-ST data input and output ports on cycles where the valid lines are high, the behavior of the two simulation models is identical. This is because Avalon-ST transactions are only performed when the valid lines are high.

Because you are running a very simple design for a only handful of clock cycles the speed advantage of functional simulation is not as significant as it would be for a real design.

Video and Image Processing Example Design

You can try a larger design to verify the enhanced performance of fast simulation by downloading v7.1 of the *Video Processing Reference Design* from the Altera website.

You should notice a significant performance difference if you simulate several frames of high-definition video content with and without fast functional simulation.

You can also modify the example design to see the effect of your changes on image quality in just a few minutes.

12. Adding a Board Library

Introduction

This chapter describes how to create and build a customer board library to use inside DSP Builder using built-in board components.

A XML board description file is used to define a new board library. This board description file contains all the board components and their FPGA pin assignments.

The following development boards are already supported in DSP Builder:

- Cyclone II DE2 Starter board
- Cyclone II EP2C35 board
- Cyclone II EP2C70 board
- Stratix EP1S25 board
- Stratix EP1S80 board
- Stratix II EP2S60 board
- Stratix II EP2S180 board
- Stratix II EP2SGX90 PCI Express board

Refer to the *DSP Builder Reference Manual* for information about these boards.

Creating a New Board Description

Additional boards can be added by creating new board description files. You only need to create a board description file for each new board and run a MATLAB command to build it into DSP Builder Library.

The existing components can be used or new components created.

Predefined Components

Predefined components can be found in the following folder:

*<install dir>***quartus****dsp_builder****lib****boardsupport****components**

Each board component is described by a single XML file named *<component_name>*.**component** that defines its data type, direction, bus width, and appearance. The file also contains a brief description of the component.

Component Types

There are three main types of component:

Single Bit Type:
These components have a single bit with one FPGA pin assigned to each component. The components are either inputs or outputs and cannot be changed. Predefined components of this type include:

- Red and Green LEDs (`LED0` to `LED17` and `LEDG0` to `LEDG8`)
- Software switches (`SW0` to `SW17`)
- User push buttons (PB0 to PB3)
- Reset push buttons (`IO_DEV_CLRn` and `USER_RESETN`)
- RS232 receive output and RS232 transmit input pins (`RS232Rout` and `RS232Tin`)

Fix Size Bus Type:
These components have a fixed-sized group of same type (either Input or Output) pins with one FPGA pin assigned to each bit of the bus. Predefined components of this type include:

- 12-bit analog-to-digital converter (`A2D1Bit12` and `A2D2Bit12`)
- 14-bit analog-to-digital converter (`A2D1Bit14` and `A2D2Bit14`)
- 14-bit digital-to-analog converter (`D2A1` and `D2A2`)
- 8-bit dual in-line package switch (`DipSwitch`)
- 7-Segment display with a decimal point (`SevenSegmentDisplay0` to `SevenSegmentDisplay1`)
- Simple 7-Segment display without a decimal point (`Simple7SegmentDisplay0` to `Simple7SegmentDisplay7`)

Selectable Single Bit Type:
These components have a single bit, but the pin can be selected from a group of predefined FPGA pins. Furthermore, the pin can be set as either input or output. Predefined components of this type include:

- Debug pins (`DebugA` and `DebugB`)
- Prototyping pins (`PROTO`, `PROTO1` to `PROTO3`)
- Evaluation input pin (`EvalIoIn`)
- Evaluation output pin (`EvalIoOut`)

Component Description File

You can define a new component by creating a component file named <*component_name*>.**component** in the same folder as the predefined components.

The component description file contains a root element `component` which contains several attributes and sub-elements that define the component. The `component` attributes are defined as follows:

- `displayname=` Specifies the name of the component, which is referenced by the board description file.
- `direction=` Specifies the direction of the signal. It can have the value of `Input` or `Output`. Omit this attribute for the Selectable Single Bit Type, as it will be set later.
- `type=` Specifies the data type of the signal. The type can be `BIT`, `INT`, or `UINT`. followed by the size in square brackets. For example, "`BIT[1,0]`" defines a single bit while "`UINT[12,0]`" is a 12-bit unsigned integer.

The component sub-elements are defined as follows:

- <`documentation`> *text* </`documentation`> This sub-element contains text describing the component and one of the following variable that define how the pin name, or list of pin-names appears in the new board library:
  - `%pinname%` for Single Bit Type
  - `%pinlist%` for Selectable Single Bit Type
  - `%indexedpinliat%` for Fixed Size Bus Type

- <`display [attributes]`> This sub-element has the attributes:
  - `icon=` Specifies the image file name for the component
  - `width=` Specifies the display width for the image file
  - `height=` Specifies the display height for the image file

For components without an image, you can omit the icon attribute and define a visual representation using the `plot` and `fprintf` commands. For example:

```
<display width="90" height="26">
plot([0 19 20 21 22 21 20 19], [0 0 1 0 0 0 -1 0]);
fprintf('EVAL IO OUT \n%pinname% ');
</display>
```

Example Component Description File:

```
<component displayname="EVAL IO OUT" direction="Output"
type="BIT[1,0]">
    <documentation>
Prototyping Area Pin Single Bit Output
%pinlist%
    </documentation>
    <display width="90" height="26">
        plot([0 19 20 21 22 21 20 19],[0 0 1 0 0 0 -1 0]);
        fprintf('EVAL IO OUT \n%pinname% ');
    </display>
</component>
```

Board Description File

The board description file is named <*board_name*>.**board** and should be created in the folder:

<*install dir*>**quartus****dsp_builder****lib****boardsupport****boards**

It consists of several sections as described in the following sections.

Header Section

This section contains a line that defines the XML version and character encoding used in the document:

```
<?xml version="1.0" encoding="UTF-8"?>
```

In this case, the document conforms to the 1.0 specification of XML and uses the ISO-8859-1 (Latin-1/West European) character set.

You should not modify this line.

Board Description Section

The main body of the document is a root element `board` that has several attributes and sub-elements which define the details of the board.

```
<board Attributes>
    <displayname> Text </displayname>
    <component Attributes />
    ...........
    <component Attributes />
    <configuration Attributes>
        <devices> Attributes>
        </devices>
        <option Attributes>
        </option>
    </configursation>
</board>
```

The last line in the file must be a closing tag for the root element board </`board`>.

The `board` attributes are defined as follows:

- `uniquename` = A unique name used to reference the board.
- `family` = Device family of the FPGA on board (assuming only one device is on the board).

The board must contain a `displayname` sub-element containing text that describes the board. For example:

```
<displayname>Cyclone II XYZ Board</displayname>
```

This is followed by `component` sub-elements that declare the components:

Single bit type examples:

```
<component name="LED0" pin="Pin_E5"/>
<component name="LED1" pin="Pin_B3"/>
```

where attribute `name` defines the name of the component on the board and `pin` defines the FPGA pin to which the component is connected. The name must match one of the predefined components and can only be used once per board.

Fixed-size bus type example:

```
<component name="DipSwitch" label="S1">
    <pin location="Pin_AC13"/> <!-- LSB -->
    <pin location="Pin_A19"/>
    <pin location="Pin_C21"/>
    <pin location="Pin_C23"/>
    <pin location="Pin_AE18"/>
    <pin location="Pin_AE19"/> <!-- MSB-->
</component>
```

where attribute `name` defines the name of the component on the board and `label` defines the name of the component as it appears in Simulink. For a component with width *n*, there must be *n* pin sub-elements. The pin location must be a valid FPGA pin name. Note that the pin ordering is listed from LSB to MSB, with LSB on top of the list.

Selectable single bit type example:

```
<component name="PROTO1">
    <pin location="Pin_C3"/>
    <pin location="Pin_D2"/>
    <pin location="Pin_L3"/>
    <pin location="Pin_J7"/>
    <pin location="Pin_J6"/>
    <pin location="Pin_K6"/>
</component>
```

This element has the same format as the fixed-size bus type, but each pin element can be chosen from a specified list of available FPGA pin locations.

The `configuration` element defines the board configuration block. For example:

```
<configuration icon="dspboard2c35.bmp" width="166" height="144">
    <devices jtag-code="0x020B40DD">
        <device name="EP2C35F672C6" />
    </devices>
<!-- Input clock selection list -->
    <option name="ClockPinIn" label="Clock Pin In">
        <pin location="Pin_N2"/>
        <pin location="Pin_N25"/>
        <pin location="Pin_AE14"/>
        <pin location="Pin_AF14"/>
        <pin location="None"/>
    </option>
<!-- Global Reset Pin -->
    <option name="GlobalResetPin" label="Global Reset Pin">
        <pin location="Pin_A14"/>
        <pin location="Pin_AC18"/>
        <pin location="Pin_AE16"/>
        <pin location="Pin_AE22"/>
        <pin location="None"/>
    </option>
</configuration>
```

The configuration attributes are defined as follows:

- `icon` = The image file to be used for the board configuration block
- `width` = The width of the image
- `height` = The height of the image

The `devices` sub-element has the following attributes:

- `jtag-code` = The JTAG code of the FPGA device
- `device name` = The device name of the FPGA used on the board

`Each option` sub-element has the following attributes:

- `name` = The name of the option (clock or reset pin)
- `label` = Labels that identifies the pins on the blocks
- `pin location` = A list of selectable clock or reset pins

Refer to any of the existing board description files for further examples.

Building the Board Library

Restart MATLAB without opening the Simulink library and run the following command in the MATLAB Command Window to create the new board library:

```
alt_dspbuilder_createComponentLibrary
```

13. Troubleshooting

Troubleshooting Issues

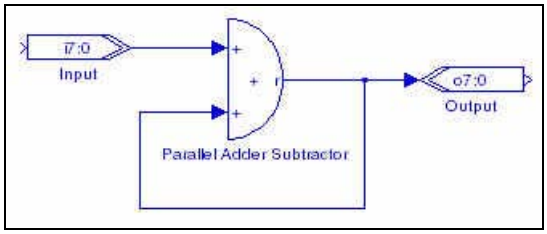
This chapter contains information about resolving the issues and error conditions listed in Table 13–1.

Table 13–1. Troubleshooting Issues

Loop Detected While Propagating Bit Widths

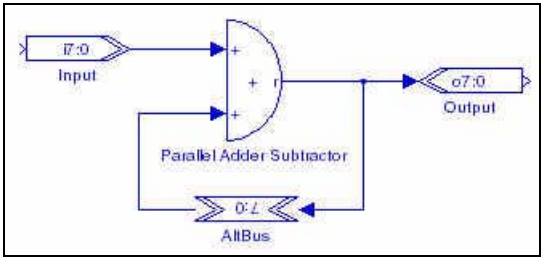
You may get an error if you have a feedback loop in your design and the feedback loop's bit width is not defined explicitly. Figure 13–1 shows this error.

Figure 13–1. Feedback Loop With Unresolved Width Error



To avoid this error, include an `AltBus` block configured as an internal node to specify the bit width in the feedback loop explicitly, as shown in Figure 13–2.

Figure 13–2. Feedback Loop With AltBus Block as an Internal Node



The MegaCore Blocks Folder Does Not Appear in Simulink

The Simulink Library Browser may not display Altera MegaCore® functions if you installed DSP Builder before you installed the Altera MegaCore IP library.

To fix this problem, type the following command after you have installed the Altera MegaCore IP library:

```
alt_dspbuilder_setup_megacore ↵
```

Synthesis Flow Does Not Run Properly

The DSP Builder automated flows allow you to control your entire synthesis and compilation flow from within the MATLAB/Simulink environment using the `Signal Compiler` block. With the automated flow, the `Signal Compiler` block outputs VHDL files and Tcl scripts and then automatically begins synthesis and compilation in the Quartus II software.

If the Quartus II software does not run automatically, check the software paths and if necessary, change the system path settings.

Check the Software Paths

If you have multiple versions of the same software product on your PC (for example, Quartus II Limited Edition and a full version of the Quartus II software), your registry settings may point to the wrong version.

DSP Builder obtains the software path from the QUARTUS_ROOTDIR environment variable.

Change the System Path Settings

If the paths to the software are incorrect, fix them by performing the following steps:

1. Open the Environment Variables dialog box from the **Advanced** tab of the Windows System Properties dialog box. This can be opened by right clicking on My Computer on the Desktop, or by double-clicking on System in Control Panel.

2. Make sure that system variable QUARTUS_ROOTDIR points to the correct version of the Quartus II software. If QUARTUS_ROOTDIR does not appear in the dialog box, create a new system variable and assign a valid value, such as **C:\Altera\71\quartus**.

3. Check that %QUARTUS_ROOTDIR%\bin is included in the Path system variable and located just after the Windows operating system.

4. Click **OK** to exit Environment Variables and System Properties dialog boxes.

DSP Development Board Troubleshooting

If `Signal Compiler` does not appear to have configured the device on the DSP development board, check the following:

- Ensure that the board is set up and connected to your PC and you have installed any necessary drivers. See the DSP development board's getting started user guide for instructions.
- When the board is powered up, the CONF_DONE LED is illuminated. The CONF_DONE LED turns off and then on when configuration completes successfully. If you do not observe the LED operating in this way, configuration was unsuccessful.
- You can configure the DSP board manually using an SRAM Object File (**.sof**), a ByteBlasterMV™, ByteBlaster™ II, ByteBlaster, or USB-Blaster™ download cable, and the Quartus II Programmer in JTAG mode. `Signal Compiler` generates the **.sof** file in your working directory. See any of the white papers included with the Stratix® II or Stratix DSP development kit for instructions on using a **.sof** file to configure the board.

Signal Compiler is Unable to Check a Valid License

You may receive this error message if you try to generate VHDL files and Tcl scripts (or try to generate VHDL stimuli) without having installed a license for DSP Builder. For information on how to obtain a license, see the *DSP Builder Release Notes*.

Verifying That Your DSP Builder Licensing Functions Properly

Type the following command in the MATLAB **Command Window**:

```
dos('lmutil lmdiag C4D5_512A') ↵
```

This command outputs the status of the DSP Builder license as follows:

```
lmutil - Copyright (C) 1989-1999 Globetrotter Software, Inc.
FLEXlm diagnostics on Wed 10/24/2001 14:36
-------------------------------------------------------
License file: c:\qdesigns\license.dat
-------------------------------------------------------
"C4D5_512A" v0000.00, vendor: alterad
uncounted nodelocked license, locked to Vendor-defined
"GUARD_ID=T000001297" no expiration date
```

You receive a message about the `hostid` if you are using an Altera software guard for licensing.

If the command does not work as described above, your license file may not be set up correctly. For information on how to check your system path and registry settings, see "Synthesis Flow Does Not Run Properly" on page 13–3.

If your license file has a SERVER line, type the following command in the MATLAB **Command Window**:

```
dos('lmutil lmstat -a') ↵
```

This command outputs the status of the DSP Builder license as follows:

```
lmutil - Copyright (C) 1989-1999 Globetrotter Software, Inc.
Flexible License Manager status on Fri 8/3/2001 15:36
License server status:
server@company,otherserver@company,thirdserver@company
License file(s) on shama: /usr/licenses/quartus/license.dat:
    shama: license server UP (MASTER) v7.0
    mogul: license server UP v7.0
    newton: license server UP v7.0
        Vendor daemon status (on shama):
        alterad: UP v7.0
        Feature usage info:
        Users of C4D5_512A: (Total of 100 licenses available)
```

If the command does not work as described above, your license file may not be set up correctly.

Verifying That the LM_LICENSE_FILE Variable Is Set Correctly

The LM_LICENSE_FILE system variable must point to your **license.dat** file that includes the DSP Builder FEATURE line for the DSP Builder to operate properly.

If you have multiple versions of software that uses a **license.dat** file (for example, Quartus II Limited Edition and a full version of the Quartus II software), make sure that LM_LICENSE_FILE points to the version of software that you want to use with DSP Builder.

Other software products, such as LeonardoSpectrum, also use the LM_LICENSE_FILE variable to point to a license file. You can combine several **license.dat** files into one or you can specify multiple **license.dat** files in the steps below.

Perform the following steps to set the LM_LICENSE_FILE variable:

1. Choose **Settings > Control Panel** (Windows Start menu).
2. Double-click the **System** icon in the Control Panel window.
3. In the **System Properties** dialog box, click the **Advanced** tab.
4. Click the **Environment Variables** button.
5. Click the **System Variable** list to highlight it, and then click **New**.
6. In the **Variable Name** box, type LM_LICENSE_FILE.
7. In the **Variable Value** box, type <*path to license file*>\license.dat.
8. Click **OK**.

If You Still Cannot Get a License

- Try adding the following paths to your system path:
  - **quartus/bin**
  - **matlab/bin**
- Remove and reinstall DSP Builder. After removing DSP Builder, use the Windows Explorer to delete any DSP Builder files or directories that remain in the file system.

SignalTap II Analysis Appears to be Hung

The SignalTap® II analyzer should terminate successfully after all trigger conditions are met. However, if one or more of the trigger conditions are not met, the SignalTap II analyzer will not terminate and the JTAG node remains locked.

You can either disconnect the USB cable and reconnect it back again; or switch off the board and switch it on again. You will need to program the board again if it is powered off.

Error When Output Block Connected to an Altera Synthesis Block

An Output block maps to output ports in VHDL and marks the edge of the generated system. You should normally use these blocks to connect simulation blocks (that is, Simulink blocks) for your testbench. If you want to use DSP Builder blocks outside your synthesizable system (such as for test bench generation or verification) put Non-synthesizable Input and Non-synthesizable Output blocks around them.

DSP Builder Start Up Dependencies

Before version 6.0, DSP Builder did not have any explicit dependencies on the Quartus II software. `Signal Compiler` could be started in DSP Builder provided there was a version of the Quartus II software registered on the computer where DSP Builder was running.

From the version 6.0 release, DSP Builder is built using the Quartus II libraries to share functionality that exists in the Quartus II software. This, however, places explicit dependencies on the Quartus versions.

DSP Builder is Simulink dependent. After installing DSP Builder, you need to register it inside MATLAB to enable the DSP Builder features. You can then create DSP designs using DSP Builder blocks and run Simulink simulations without any requirements on the Quartus II software.

However, when you want to generate VHDL for the DSP design and to fit the design onto an FPGA, DSP Builder requires the Quartus II synthesis, and place & route tools.

The `Signal Compiler` tool inside DSP Build can only be started with a matching version of the Quartus II software and explicitly depends on the correct version libraries and DLLs from the Quartus II libraries. The second page of the Signal Compiler dialog box does not display without a matching version of the Quartus II software.

If `Signal Compiler` does not run properly, you can follow the steps given below to check whether a compatible version of the Quartus II software is registered when DSP Builder is run.

1. After installing DSP Builder inside MATLAB, type `ver` in the MATLAB command window. The DSP Builder version and build numbers are displayed under **DSP Builder - Altera Corporation**.

2. Open a DOS command prompt and type either `env` or `set` to display the environment settings. Check that the environment variable QUARTUS_ROOTDIR points to the correct Quartus II software installation.

3. Check the PATH environment variable to ensure that the correct version of **Quartus\bin** is in the path.

4. When Cygwin is installed, make sure that it is listed after Quartus in the path. Correct environment settings in Cygwin do not guarantee that `Signal Compiler` will start properly, as DSP Builder relies on DOS settings rather than Cygwin. (When MATLAB is started from a Cygwin command prompt window, `system env` in the MATLAB command window only reflects the Cygwin settings.)

5. If there are any other operation systems, such as WinVar, installed on top of Windows, make sure that they are listed after Quartus in the PATH environment variable.

Warning When Input/Output Blocks Conflict with clock or aclr Ports

A warning is issued if an input or output port has the same name as a clock or reset signal used in the model. For example if your design has an input port named `aclr`, this is the same name as the default system reset and the following warning is issued during analysis:

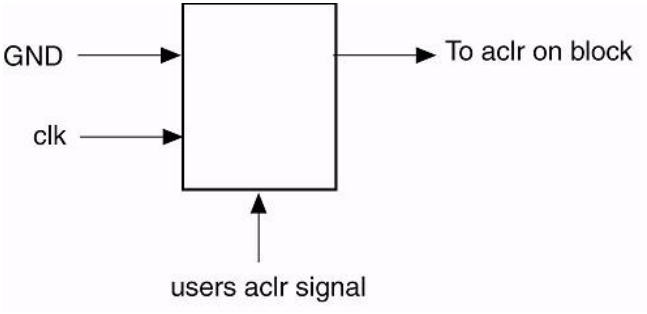
```
Warning: aclrInputPortTest/aclr has been renamed to avoid conflict: aclr has been renamed
to aclr_1:
```

The input port is renamed during HDL conversion. If you want to keep the port called `aclr`, you should add a `Clock` block and use it to rename the name used for the reset port.

Wiring the Asynchronous Clear Signal

The asynchronous clear signal should be wired via a register to make sure that the end of the `aclr` cycle is synchronized with the clock as shown in Figure 13–3.

Figure 13–3. Wiring the Asynchronous Clear Signal



A design may not match the hardware if an asynchronous clear is performed during simulation because the `aclr` cycle may last several clocks - depending on clock speed and the device.

Simulation Mismatch After Changing Signals or Parameters

The simulation results may not match unless you delete the previous testbench directory (**tb_<*model name*>**) before re-running a testbench comparison after changing any signal names or parameters.

Error Issued When a Design Includes Pre-v7.1 Blocks

An error of the following form is issued if you attempt to simulate a design which includes un-upgraded pre-v7.1 blocks:

```
Data type mismatch. Input port 1 of '<old block>' expects a signal of data type 'double'.
However, it is driven by a signal of data type 'DSPB_Type'.
```

Refer to the *DSP Builder Release Notes* for information about upgrading your designs.

Creating an Input Terminator for Debugging a Design

If there is a problem somewhere in a design, it can be useful to disconnect some subsystems so that you can analyze a small portion of the design. This may cause bit width propagation and inheritance problems. You can avoid these problems by inserting a Non-synthesizable Output followed immediately by a Non-synthesizable Input. This combination functions as a temporary input terminator and can be removed after the design has been debugged.

A Specified Path Cannot be Found or a File Name is Too Long

The maximum length for a path is limited to 256 characters in the Windows operating system.

When the file path to a model or the name of the model is very long, DSP Builder may attempt to create a file path exceeding this limit.

If this problem occurs, reduce the length of the file path to the model, and/or the length of its name.

Incorrect Interpretation of Signed Bit in Output from MegaCores

For some MegaCore functions, DSP Builder may be unable to infer whether output signals should be interpreted as signed or unsigned. This can cause problems when visualizing the output (for example, by directly attaching scopes), when the signal waveform may be obscured due to the misinterpretation of the highest bit. This can be corrected by connecting to the output via an `AltBus` or `Non-synthesizable Output` block (as appropriate) with the correct bus type assignment.

Simulation Mismatch For FIR Compiler MegaCore Function

Functional simulation models generated by a FIR Compiler MegaCore function generally do not output valid data until the data storage of these models is clear.

Refer to the *Simulate the Design* section in the *FIR Compiler User Guide* for more information including a formula which can be used to estimate the number of cycles required before relevant samples are available.

Additional Information

Revision History

The table below displays the revision history for the chapters in this user guide.

Date	Version	Changes Made
October 2007	7.2	Minor updates to all chapters. Added a new chapter which describes how to create a custom board library.
June 2007	7.1 SP1	Updated various out-of-date screenshots and other minor corrections.
May 2007	7.1	Major updates to all chapters. New Using the Simulator Accelerator chapter.
March 2007	7.0	Updated for version 7.0 of the Quartus® II software.
December 2006	6.1	SOPC Builder Links library renamed as Interfaces library with the Avalon® blocks renamed as Avalon Memory-Mapped (Avalon-MM) interface blocks. New *tbdiff* comparison utility and updated description of the *dspbuilder_sh* utility. Updated the MegaCore® function walkthrough.
April 2006	6.0	Updates for using MATLAB variables, additional Avalon signal and custom instruction support. Moved the example Tcl script appendix from reference manual.
January 2006	5.1 SP1	Updated the Tutorial, Design Rules, Using Hardware in the Loop, Performing SignalTap II Logic Analysis, Using the State Machine Library chapters, and Creating Custom Library Blocks chapters. Various other minor content and format corrections.
October 2005	5.1.0	Updated the Tutorial, Design Rules, Using MegaCore Functions, Using SOPC Builder Links (new Avalon blocks), Using Black Boxes (new HDL Import block), Creating Custom Library Blocks, and the Troubleshooting chapters.
August 2005	5.0.1	Added support for the Stratix® II EP2S180 DSP Development board.
April 2005	5.0.0	Updated version from 3.0.0 to 5.0.0. Added support for the Cyclone® II DSP board. Removed the "*Supporting Custom Boards with DSP Builder*" chapter.
January 2005	3.0.0	Added support for Hardware in the Loop (HIL). Added additional blocks and design examples.
August 2004	2.2.0	Added support for use of MegaCore® functions and Cyclone II and Stratix® II devices.
July 2003	2.1.3	Split the documentation into two books, the *DSP Builder User Guide*, which provides how-to information, and the *DSP Builder Reference Manual*, which provides design rules and block reference.
April 2003	2.1.2	Added information on the Stratix DSP Board EP1S80 library. Minor additional changes.

Date	Version	Changes Made
February 2003	2.1.1	Added information on using DSP Builder modules in external RTL designs. Added information on creating custom library blocks. Additional minor documentation updates.
December 2002	2.1.0	Added support for Stratix GX devices, Cyclone devices, the state machine, and PLL blocks. Added information and walkthrough for the DSP board, the PLL block, and Simulink v5.0. Updated information on the Signal Compiler block.
June 2002	2.0.0	Updated information on the Signal Compiler block. Added information and walkthrough for the SignalTap® blocks. Added block descriptions for new arithmetic, storage, DSP board, complex signals, and SOPC blocks. Described support for Stratix devices. Updated the tutorial.
October 2001	1.0	First version of the user guide for DSP Builder version 1.0.0.

How to Contact Altera

For the most up-to-date information about Altera® products, refer to the following table.

Information Type	Contact *Note (1)*
Technical support	www.altera.com/mysupport/
Product literature	www.altera.com
Altera literature services	literature@altera.com
FTP site	ftp.altera.com

Note to table:
(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

This document uses the typographic conventions shown below.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, file names, file name extensions, and software utility names are shown in bold type. Examples: $\mathbf{f_{MAX}}$, **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Designs.*
Italic type	Internal timing parameters and variables are shown in italic type. For example: t_{PIA}, $n + 1$. User specified variable names are enclosed in angle brackets (< >) and shown in italic type. For example: <*file name*>, <*project name*>**.pof** file.

Visual Cue	Meaning
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
"Subheading Title"	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: "Typographic Conventions."
`Courier type`	Signal and port names are shown in lowercase Courier type. Examples: `data1`, `tdi`, `input.` Active-low signals are denoted by suffix `n`, for example, `resetn`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\qdesigns\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (for example, the VHDL keyword `BEGIN`), as well as logic function names (for example, `TRI`) are shown in Courier.
1., 2., 3., and a., b., c., and so on	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ● •	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
	The hand points to information that requires special attention.
	The caution calls attention to a condition that could damage the product or design and should be read prior to starting or continuing with the procedure or process.
	The warning calls attention to a condition that could cause injury to the user and should be read prior to starting or continuing the procedure or processes.
↵	The angled arrow indicates you should press the Return key.
	The feet direct you to more information on a particular topic.

Other Documentation

See the *DSP Builder Reference Manual* for a description of the parameters supported by each of the Altera blocks. The block descriptions can also be accessed in HTML format from the right mouse menu in a design model or in the Simulink library browser.

See the *DSP Builder Release Notes and Errata* for information about system requirements, obtaining and installing the software, setting up licensing, upgrading from a pre-v7.1 release, new features, known errata issues and workarounds.

These manuals are available as PDF documents from the **DSP Builder v7.2** menu in the **Altera** section of the programs in the Windows Start menu.

Index

T

U

W