DSP Builder

User Guide

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

Software Version: 8.1
Document Date: November 2008

UG-DSPBUILDER-8.1

Contents

1. About DSP Builder

Release Information

Table 1–1 provides information about this release of DSP Builder.

Table 1–1. DSP Builder Release Information

Item	Description
Version	8.1
Release Date	November 2008
Ordering Code	IPT-DSPBUILDER

Device Family Support

DSP Builder supports the following target Altera® device families: Stratix®, Stratix GX, Stratix II, Stratix II GX, Stratix III, Stratix IV, 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 1–2 shows the device families which support each memory block type.

Table 1–2. Supported Memory Block Types

Memory Block Type	Device Family
M144K	Stratix IV, Stratix III
M9K	Stratix IV, Stratix III, Cyclone III
MLAB	Stratix IV, Stratix III
M-RAM	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.

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.
- Automatic VHDL testbench generation and control of Quartus II compilation.
- Provides a variety of fixed-point arithmetic and logical operators for use with the Simulink software.
- Enables rapid prototyping using Altera DSP development boards.
- 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 workspace to facilitate visual analysis.
- HDL import of VHDL or Verilog HDL design entities and HDL defined in a Quartus II project file.
- Hardware in the Loop (HIL) support to enable FPGA hardware accelerated co-simulation with Simulink.
- Support for Avalon® Memory-Mapped (Avalon-MM) interfaces including user configurable blocks that you can use to build custom logic that works with the Nios® II processor and other SOPC Builder designs.
- Support for Avalon Streaming (Avalon-ST) interfaces including an Packet Format Converter block and configurable Avalon-ST Sink and Avalon-ST Source blocks.
- Altera DSP or Video and Image Processing Suite MegaCore® functions can be directly instanced in a DSP Builder design model.
- Supports cycle-accurate or fast functional (bit-accurate) simulation of the Video and Image Processing Suite MegaCore functions.
- Support for tabular and graphical state machine editing.

For information about new features and errata in this release, refer to the *DSP Builder Release Notes and Errata*.

Installing DSP Builder

You can choose to optionally install DSP Builder when you install the Quartus II software.

For specific information about installing and licensing DSP Builder, refer to *DSP Builder Installation and Licensing*.

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 achieve the better performance and 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 contain other DSP Builder blocks and MegaCore functions. `Signal Compiler` then generates the 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. You can think of programmable logic as an array of elements, each of which you can configure as a complex processor routine.

You can link these routines together in serial (the same way that a digital signal processor would execute them), or connect them in parallel. When connected in parallel, they give many times better performance than standard digital signal processors by executing hundreds of instructions at the same time.

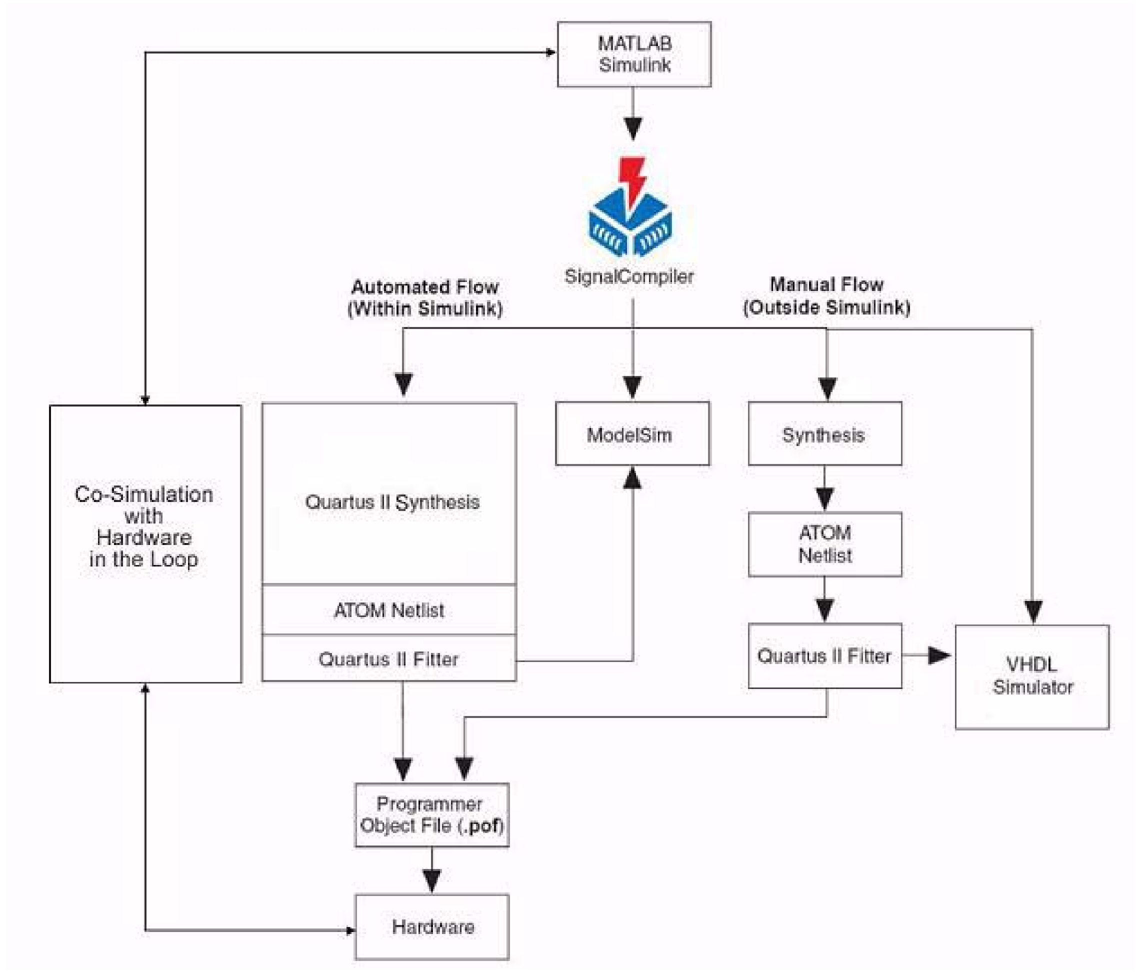
Algorithms that benefit from this improved performance include forward-error correction (FEC), modulation/demodulation, and encryption.

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.

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

Figure 1–1. System-Level Design Flow



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.

   The DSP Builder part of your design should be separated from the Simulink blocks by Input and Output blocks from the DSP Builder IO and Bus library.

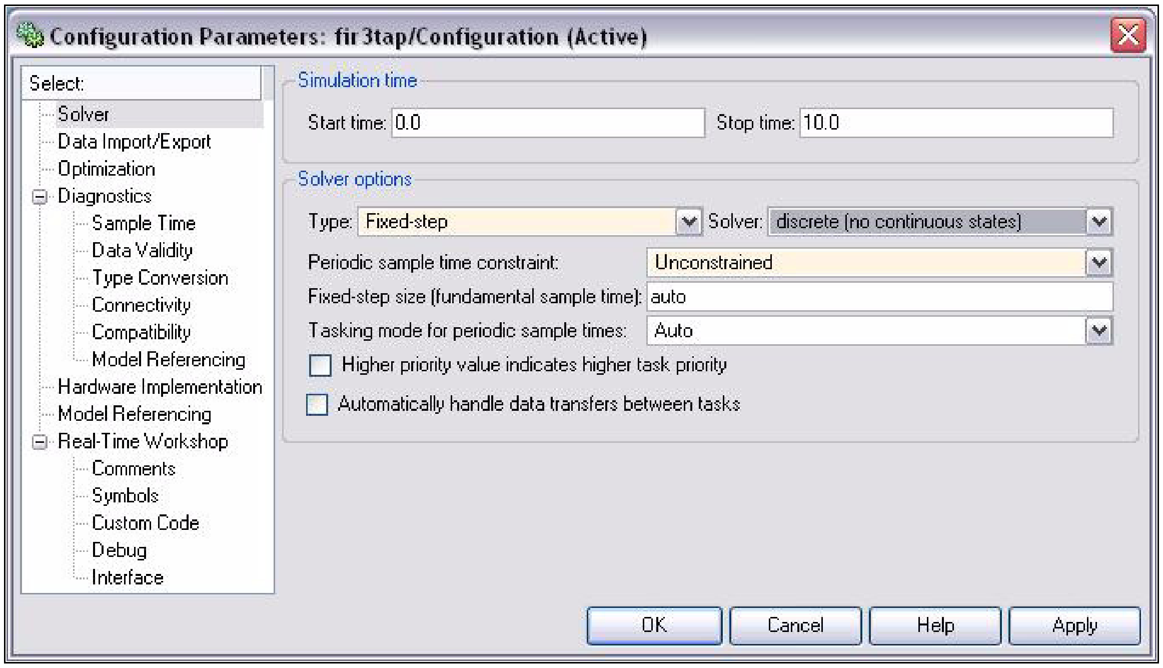
2. Include a `Clock` block from the DSP Builder AltLab library to specify the base clock for your design which must have a period greater than 1ps but less than 2.1 ms.

   If no base clock exists in the design, DSP Builder creates a default clock with a 20ns real-world period and a Simulink sample time of 1. You can derive additional clocks from the base clock by adding `Clock_Derived` blocks.

3. Set a **discrete (no continuous states)** solver in Simulink. Choose a **Fixed-step** solver type if you are using a single clock domain or a **Variable-step** type if you are using multiple clock domains.

   To set the solver options, click **Configuration Parameters** on the Simulation menu to open the **Configuration Parameters** dialog box and select the **Solver** page (Figure 1–2).

Figure 1–2. Configuration Parameters for Simulation


Configuration Parameters: fir3tap/Configuration (Active)

Simulation time

Start time: 0.0
Stop time: 10.0

Solver options

Type: Fixed-step
Solver: discrete (no continuous states)
Periodic sample time constraint: Unconstrained
Fixed-step size (fundamental sample time): auto
Tasking mode for periodic sample times: Auto
☐ Higher priority value indicates higher task priority
☐ Automatically handle data transfers between tasks

Refer to the description of the “Solver Pane” in the Simulink Help for detailed information about solver options.

4. Simulate the model in Simulink using a `Scope` block to monitor the results.
5. Run `Signal Compiler` to setup RTL simulation and synthesis.
6. 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.
7. 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.
8. Compile your design in the Quartus II software.
9. Download to a hardware development board and test.

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.

for information about controlling the DSP Builder design flow using `Signal Compiler`, refer to "Design Flows for Synthesis, Compilation and Simulation" on page 3–18. For detailed information about the blocks in the DSP Builder blockset, refer to the *DSP Builder Reference Manual*.

Interoperability with the Advanced Blockset

This release of DSP Builder includes an optional advanced blockset that is described by separate documentation.

For information about the advanced blockset refer to the *DSP Builder Advanced Blockset Reference Manual* and the *DSP Builder Advanced Blockset User Guide.* For information about the differences between the standard and advanced blocksets and about design flows that combine both blocksets, refer to the *DSP Design Flow User Guide.*

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 the following:

- 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.
- 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. On the File menu, point to **New** and click **Model** to create a new model window.
3. Click **Save** on the 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 Start. Point to **Simulink** and click **Library Browser**.

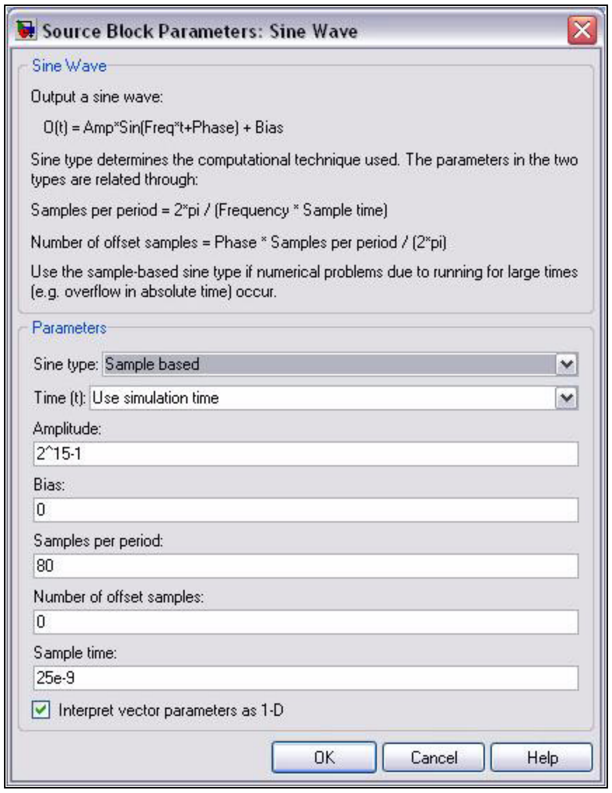
☞ You can also open Simulink by using the toolbar icon.

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** to view the blocks in the Sources library.
2. Drag and drop a `Sine Wave` block into your model.
3. Double-click the `Sine Wave` block in your model to display the **Block Parameters** dialog box (Figure 2–1).

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


Source Block Parameters: Sine Wave

Sine Wave

Output a sine wave:

O(t) = Amp*Sin(Freq*t+Phase) + Bias

Sine type determines the computational technique used. The parameters in the two types are related through:

Samples per period = 2*pi / (Frequency * Sample time)

Number of offset samples = Phase * Samples per period / (2*pi)

Use the sample-based sine type if numerical problems due to running for large times (e.g. overflow in absolute time) occur.

Parameters

Sine type: Sample based

Time (t): Use simulation time

Amplitude: 2^15-1

Bias: 0

Samples per period: 80

Number of offset samples: 0

Sample time: 25e-9

✓ Interpret vector parameters as 1-D

4. Set the `Sine Wave` block parameters as shown in Table 2–1.

Table 2–1. Parameters for the Sine Wave Block

Parameter	Value
Sine type	Sample based
Time	simulation time
Amplitude	2^15–1
Bias	0
Samples per period	80
Number of offset examples	0
Sample time	25e-9
Interpret vector parameters a 1-D	On

5. Click **OK**.

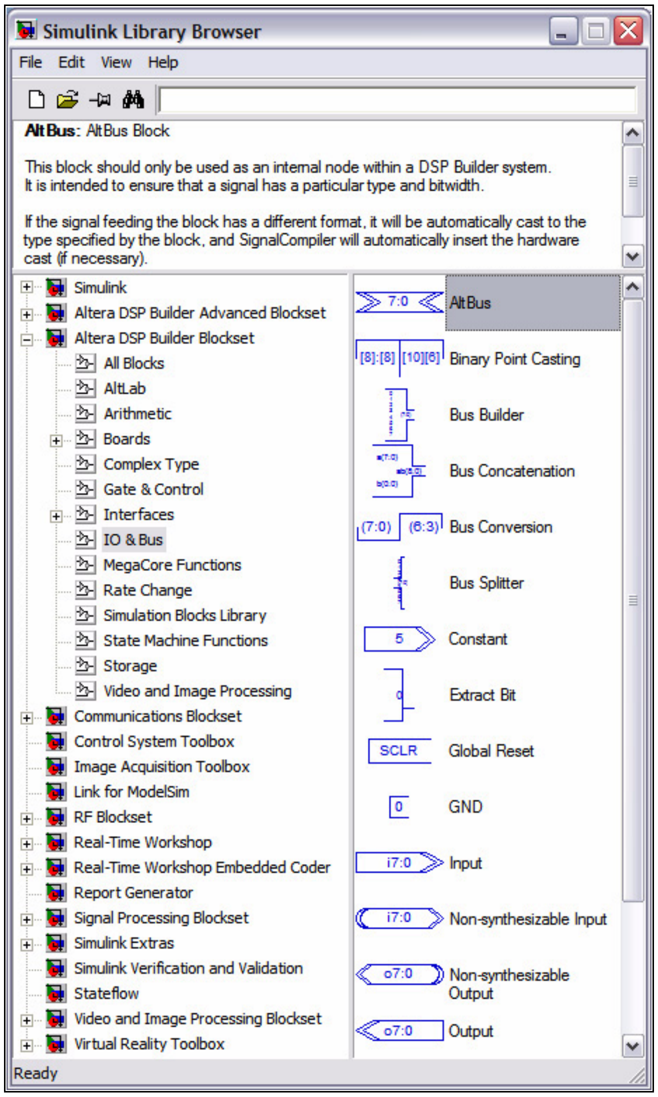
☞ See the equation in "Frequency Design Rules" on page 3–7 for information on how you can calculate the frequency.

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).

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



AltBus: AltBus Block

This block should only be used as an internal node within a DSP Builder system.
It is intended to ensure that a signal has a particular type and bitwidth.

If the signal feeding the block has a different format, it will be automatically cast to the type specified by the block, and SignalCompiler will automatically insert the hardware cast (if necessary).

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.

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

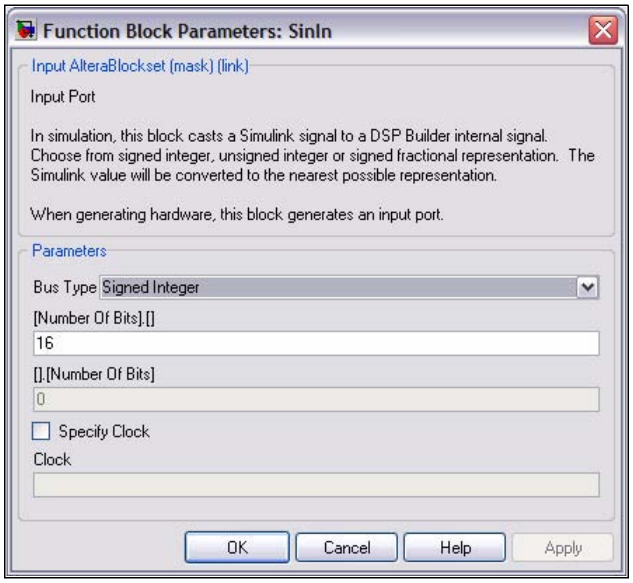
☞ You can use the Up, Down, Right, and Left arrow keys to adjust the position of a block while it is selected.

4. Click the text 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 (Figure 2–3).
6. Set the `SinIn` block parameters as shown in Table 2–2.

Table 2–2. Parameters for the SinIn Block

Parameter	Value
Bus Type	Signed Integer
[number of bits].[]	16
Specify Clock	Off

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



Function Block Parameters: SinIn

Input AlteraBlockset (mask) (link)

Input Port

In simulation, this block casts a Simulink signal to a DSP Builder internal signal. Choose from signed integer, unsigned integer or signed fractional representation. The Simulink value will be converted to the nearest possible representation.

When generating hardware, this block generates an input port.

Parameters

Bus Type Signed Integer

[Number Of Bits].[]

16

[].[Number Of Bits]

0

☐ Specify Clock

Clock

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.

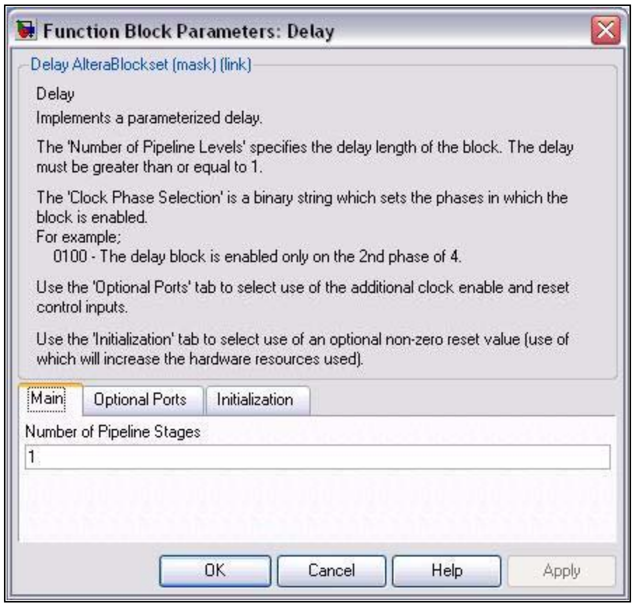
☞ Alternatively, you can select a block, hold down the Ctrl key and click the destination block to automatically make a connection between the two blocks.

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. Type `1` as the **Number of Pipeline Stages** for the `Delay` block.

Figure 2–4. Setting the Downsampling Delay



Function Block Parameters: Delay

Delay AlteraBlockset (mask) (link)

Delay

Implements a parameterized delay.

The 'Number of Pipeline Levels' specifies the delay length of the block. The delay must be greater than or equal to 1.

The 'Clock Phase Selection' is a binary string which sets the phases in which the block is enabled.
For example;
0100 - The delay block is enabled only on the 2nd phase of 4.

Use the 'Optional Ports' tab to select use of the additional clock enable and reset control inputs.

Use the 'Initialization' tab to select use of an optional non-zero reset value (use of which will increase the hardware resources used).

Main | Optional Ports | Initialization

Number of Pipeline Stages: 1

OK | Cancel | Help | Apply

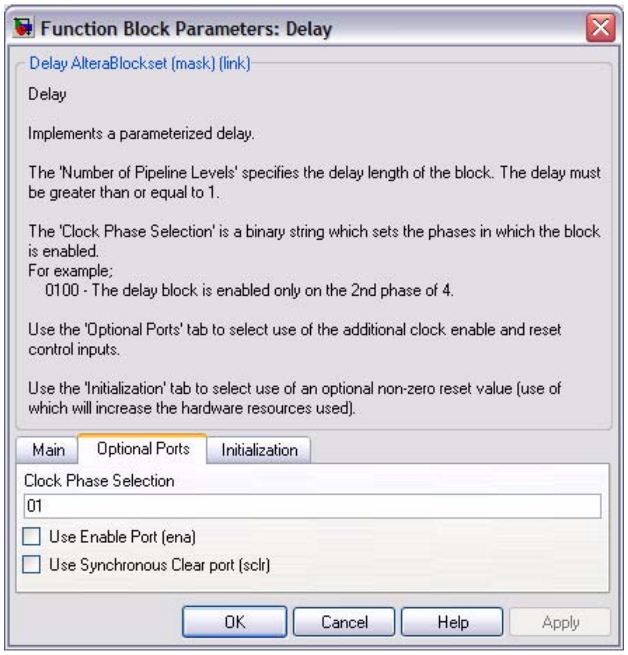
5. Click the **Optional Ports** tab and set the parameters shown in Table 2–3.

Table 2–3. Parameters for the Delay Block.

Parameter	Value
Clock Phase Selection	01
Use Enable Port	Off
Use Synchronous Clear port	Off

The completed dialog box is shown in Figure 2–4 on page 2–5.

Figure 2–5. Delay Block Optional Ports Tab


Function Block Parameters: Delay
Delay AlteraBlockset (mask) (link)
Main
Optional Ports
Initialization
OK
Cancel
Help
Apply

Delay

Implements a parameterized delay.

The 'Number of Pipeline Levels' specifies the delay length of the block. The delay must be greater than or equal to 1.

The 'Clock Phase Selection' is a binary string which sets the phases in which the block is enabled.
For example;
0100 - The delay block is enabled only on the 2nd phase of 4.

Use the 'Optional Ports' tab to select use of the additional clock enable and reset control inputs.

Use the 'Initialization' tab to select use of an optional non-zero reset value (use of which will increase the hardware resources used).

Clock Phase Selection
01
☐ Use Enable Port (ena)
☐ Use Synchronous Clear port (sclr)

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:

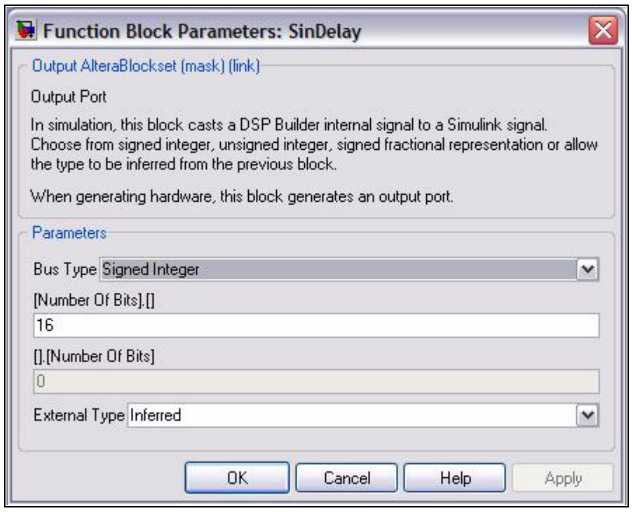
1. Select the **IO & Bus** library from the **Altera DSP Builder Blockset** folder in the Simulink Library Browser.
2. Drag and drop two `Output` blocks into your model, positioning them to the right of the `Delay` block.
3. Click the text under the block symbols in your model. Change the block instance 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 shown in Table 2–4.

Table 2–4. Parameters for the SinDelay Block

Parameter	Value
Bus Type	Signed Integer
[number of bits].[]	16
External Type	Inferred

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

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


Function Block Parameters: SinDelay
Output AlteraBlockset (mask) (link)

Output Port

In simulation, this block casts a DSP Builder internal signal to a Simulink signal. Choose from signed integer, unsigned integer, signed fractional representation or allow the type to be inferred from the previous block.

When generating hardware, this block generates an output port.

Parameters

Bus Type: Signed Integer

[Number Of Bits].[]: 16

[].[Number Of Bits]: 0

External Type: Inferred

6. Click **OK**.
7. Repeat steps 4 to 6 for the `SinIn2` block setting the parameters as shown in Table 2–5.

Table 2–5. Parameters for the SinIn2 Block

Parameter	**Value**
Bus Type	Signed Integer
[number of bits].[]	16
External Type	Inferred

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:

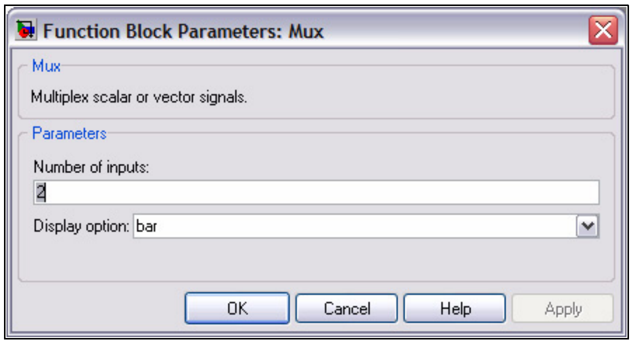
1. Select the Simulink **Signal Routing** library in the Simulink Library Browser.
2. Drag and drop a `Mux` block into your design, positioning it to the right of the `SinDelay` block.
3. Double-click the `Mux` block in your model to display the **Block Parameters** dialog box.
4. Set the `Mux` block parameters as shown in Table 2–6.

Table 2–6. Parameters for the Mux Block

Parameter	**Value**
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


Function Block Parameters: Mux

Mux

Multiplex scalar or vector signals.

Parameters

Number of inputs: 2

Display option: bar

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:

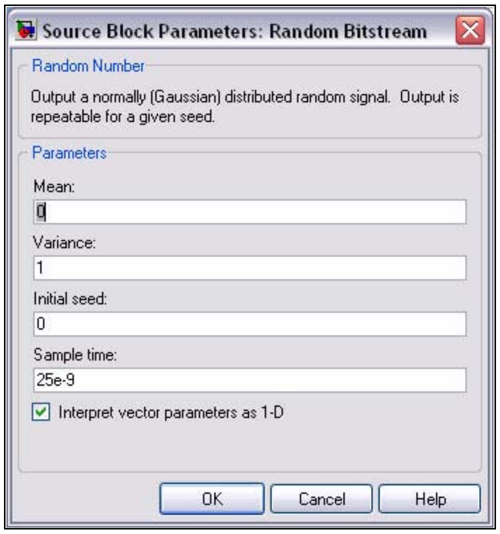
1. Select the Simulink **Sources** library in the Simulink Library Browser.
2. Drag and drop a `Random Number` block into your model, positioning it underneath the `Sine Wave` block.
3. Double-click the `Random Number` block in your model to display the **Block Parameters** dialog box.
4. Set the `Random Number` block parameters as shown in Table 2–7.

Table 2–7. Parameters for the Random number Block

Parameter	Value
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



Source Block Parameters: Random Bitstream

Random Number

Output a normally (Gaussian) distributed random signal. Output is repeatable for a given seed.

Parameters

Mean:
0

Variance:
1

Initial seed:
0

Sample time:
25e-9

✓ Interpret vector parameters as 1-D

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 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. Set the `Noise` block parameters as shown in Table 2–8.

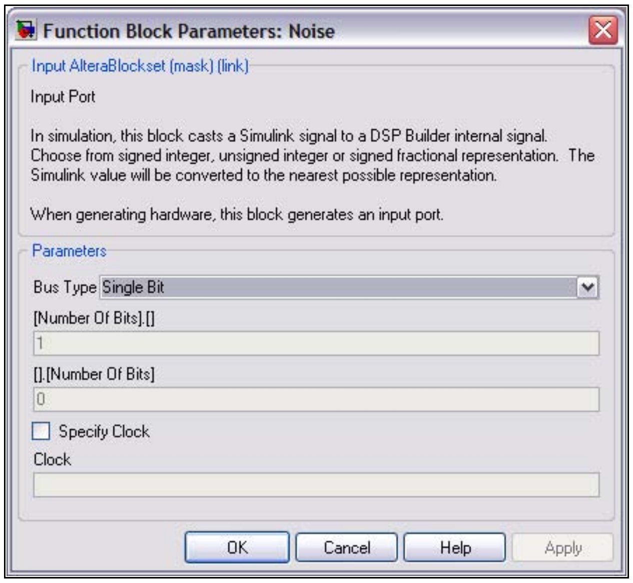
Table 2–8. Parameters for the Noise Block

Parameter	Value
Bus Type	Single Bit
Specify Clock	Off

☞ 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


Function Block Parameters: Noise

Input AlteraBlockset (mask) (link)

Input Port

In simulation, this block casts a Simulink signal to a DSP Builder internal signal. Choose from signed integer, unsigned integer or signed fractional representation. The Simulink value will be converted to the nearest possible representation.

When generating hardware, this block generates an input port.

Parameters

Bus Type: Single Bit

[Number Of Bits].[]: 1

[].[Number Of Bits]: 0

☐ Specify Clock

Clock:

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:

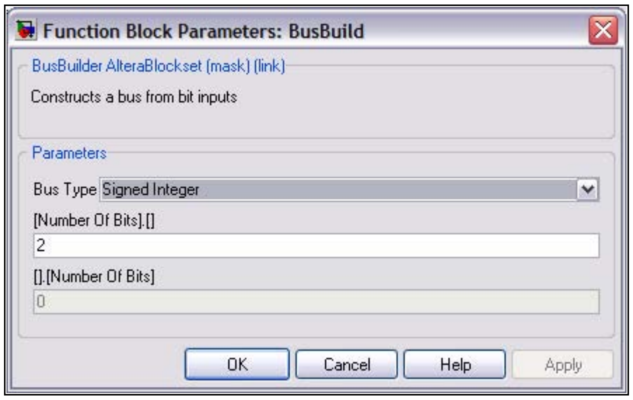
1. Select the **IO & Bus** library from the **Altera DSP Builder Blockset** folder in the Simulink Library Browser.
2. Drag and drop a `Bus Builder` block into your model, positioning it to the right of the `Noise` block.
3. Double-click the `Bus Builder` block in your model to display the **Block Parameters** dialog box.
4. Set the `Bus Builder` block parameters as shown in Table 2–9.

Table 2–9. Parameters for the Bus Builder Block

Parameter	Value
Bus Type	Signer Integer
[number of bits].[]	2

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

Figure 2–10. Build a 2-Bit Signed Bus


Function Block Parameters: BusBuild

BusBuilder AlteraBlockset (mask) (link)

Constructs a bus from bit inputs

Parameters

Bus Type: Signed Integer

[Number Of Bits].[]: 2

[].[Number Of Bits]: 0

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 `Product` block parameters as shown in Table 2–10.

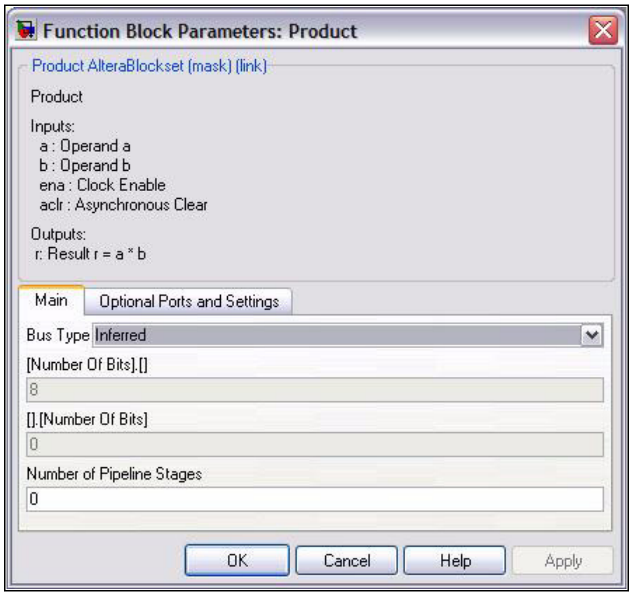
Table 2–10. Parameters for the Product Block

Parameter	Value
Bus Type	Inferred
Number of Pipeline Stages	0

The bit width parameters are set automatically when you select **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.

Figure 2–11. Product Block Parameters



Function Block Parameters: Product

Product AlteraBlockset (mask) (link)

Product

Inputs:
a : Operand a
b : Operand b
ena : Clock Enable
aclr : Asynchronous Clear

Outputs:
r: Result r = a * b

Bus Type Inferred

[Number Of Bits].[]
8

[].[Number Of Bits]
0

Number of Pipeline Stages
0

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.

Add the StreamMod and StreamBit Blocks

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

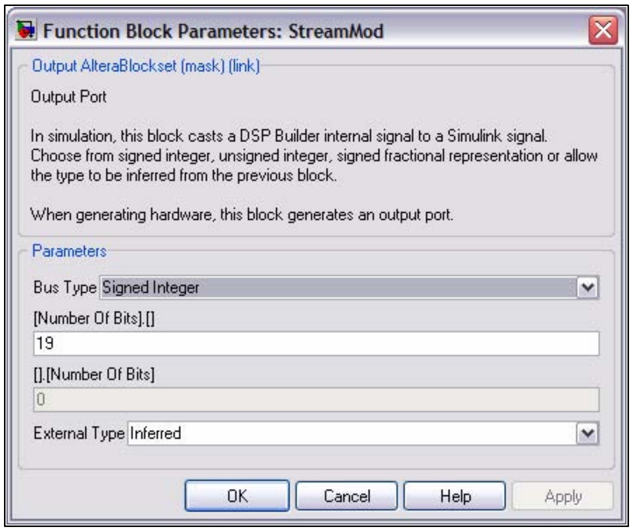
1. Select the **IO & Bus** library from the **Altera DSP Builder Blockset** folder in the Simulink Library Browser.
2. Drag and drop two `Output` blocks into your model, positioning them to the right of the `Product` block.
3. Click the text under the block symbols in your model. Change the block instance names from `Output` and `Output1` to `StreamMod` and `StreamBit`.
4. Double-click the `StreamMod` block to display the **Block Parameters** dialog box.
5. Set the `StreamMod` block parameters as shown in Table 2–11.

Table 2–11. Parameters for the StreamMod Block

Parameter	Value
Bus Type	Signed Integer
[number of bits].[]	19
External Type	Inferred

The completed dialog box is shown in Figure 2–12.

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


Function Block Parameters: StreamMod

Output AlteraBlockset (mask) (link)

Output Port

In simulation, this block casts a DSP Builder internal signal to a Simulink signal. Choose from signed integer, unsigned integer, signed fractional representation or allow the type to be inferred from the previous block.

When generating hardware, this block generates an output port.

Parameters

Bus Type: Signed Integer

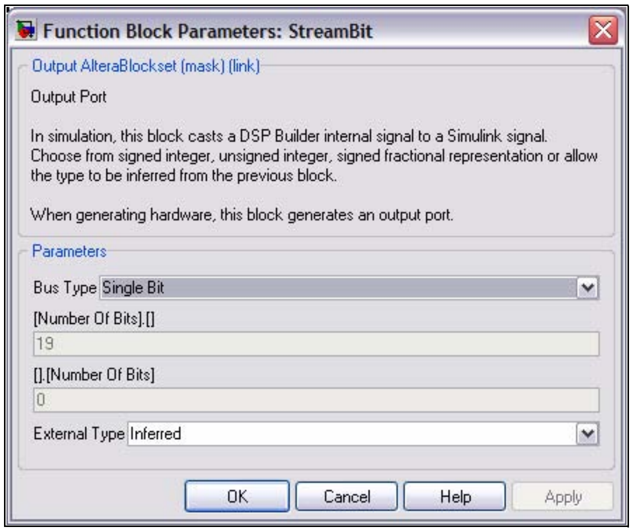
[Number Of Bits].[]: 19

[].[Number Of Bits]: 0

External Type: Inferred

6. Click **OK**.
7. Double-click the `StreamBit` block to display the **Block Parameters** dialog box (Figure 2–13).

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


Function Block Parameters: StreamBit

Output AlteraBlockset (mask) (link)

Output Port

In simulation, this block casts a DSP Builder internal signal to a Simulink signal. Choose from signed integer, unsigned integer, signed fractional representation or allow the type to be inferred from the previous block.

When generating hardware, this block generates an output port.

Parameters

Bus Type: Single Bit

[Number Of Bits].[]: 19

[].[Number Of Bits]: 0

External Type: Inferred

8. Set the `StreamMod` block parameters as shown in Table 2–12.

Table 2–12. Parameters for the StreamMod Block

Parameter	Value
Bus Type	Single Bit
External Type	Inferred

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:

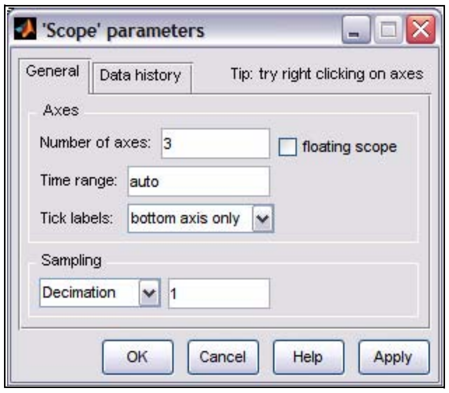
1. Select the Simulink **Sinks** library in the Simulink Library Browser.
2. Drag and drop a `Scope` block into your model and position it to the right of the `StreamMod` block.
3. Double-click the `Scope` block and click the **Parameters** icon to display the **'Scope' parameters** dialog box.
4. Set the `Scope` parameters as shown in Table 2–13.

Table 2–13. Parameters for the Scope Block

Parameter	Value
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


'Scope' parameters

General | Data history | Tip: try right clicking on axes

Axes
Number of axes: 3 ☐ floating scope
Time range: auto
Tick labels: bottom axis only

Sampling
Decimation 1

OK | Cancel | Help | Apply

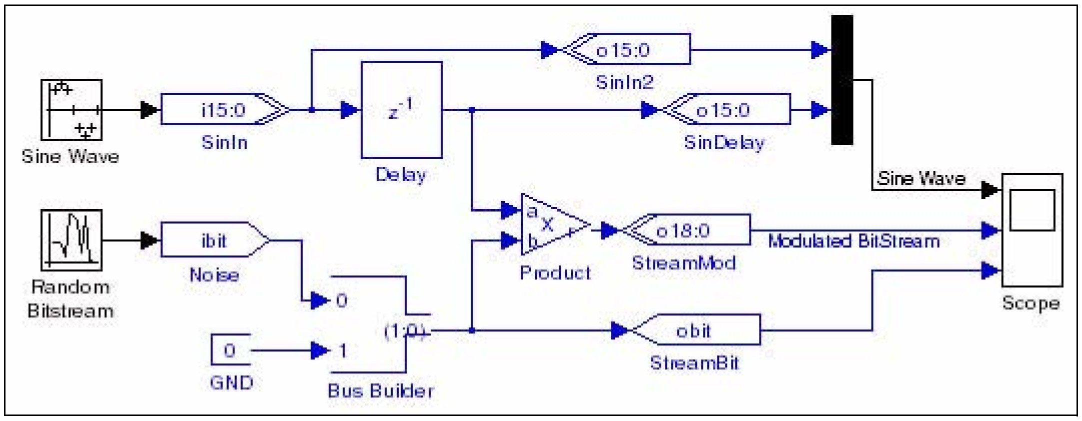
5. Click **OK**.
6. Close the Scope.
7. Make connections to connect the complete your design as follows:
   a. From the right side of the `Mux` block to the top left side of the `Scope` block.
   b. From the right side of the `StreamMod` block to the middle left side of the `Scope` block.

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 on page 2–15).
4. Set the Clock parameters as shown in Table 2–14.

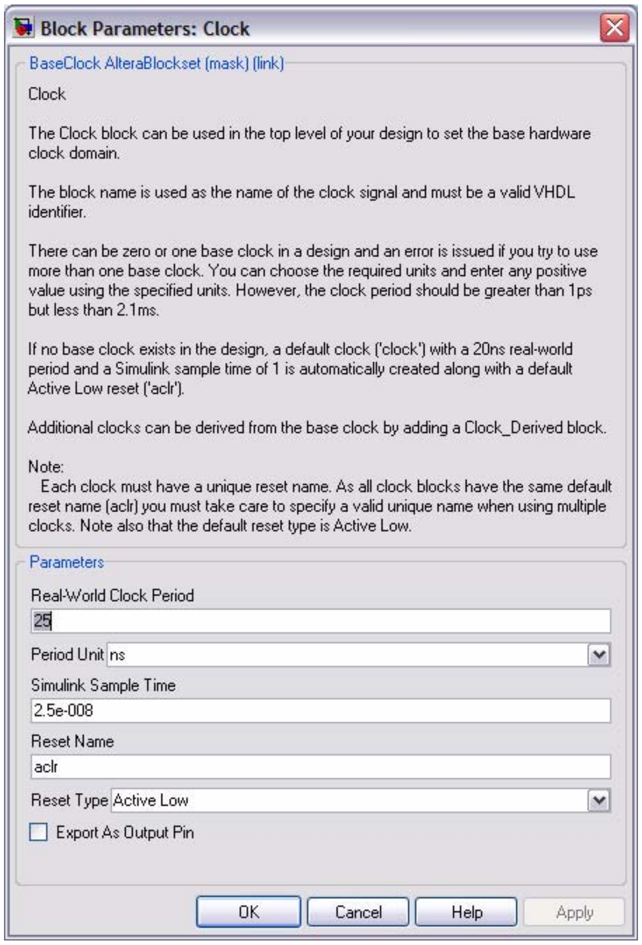
Table 2–14. Parameters for the Clock Block

Parameter	**Value**
Real-World Clock Period	20
Period Unit:	ns
Simulink Sample Time	2.5e–008
Reset Name	aclr
Reset Type	Active Low
Export As Output Pin	Off

A `clock` block is required to set a Simulink sample time that matches the sample time specified on the `Sine Wave` and `Random Bitstream` blocks. If no base clock exists in the design, a default clock with a 20ns real-world period and a Simulink sample time of 1 is automatically created.

5. Save your model.

Figure 2–16. Clock Block Parameters Dialog Box



Block Parameters: Clock

BaseClock AlteraBlockset (mask) (link)

Clock

The Clock block can be used in the top level of your design to set the base hardware clock domain.

The block name is used as the name of the clock signal and must be a valid VHDL identifier.

There can be zero or one base clock in a design and an error is issued if you try to use more than one base clock. You can choose the required units and enter any positive value using the specified units. However, the clock period should be greater than 1ps but less than 2.1ms.

If no base clock exists in the design, a default clock ('clock') with a 20ns real-world period and a Simulink sample time of 1 is automatically created along with a default Active Low reset ('aclr').

Additional clocks can be derived from the base clock by adding a Clock_Derived block.

Note:
Each clock must have a unique reset name. As all clock blocks have the same default reset name (aclr) you must take care to specify a valid unique name when using multiple clocks. Note also that the default reset type is Active Low.

Parameters

Real-World Clock Period: 25

Period Unit: ns

Simulink Sample Time: 2.5e-008

Reset Name: aclr

Reset Type: Active Low

☐ Export As Output Pin

Simulate Your Model in Simulink

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

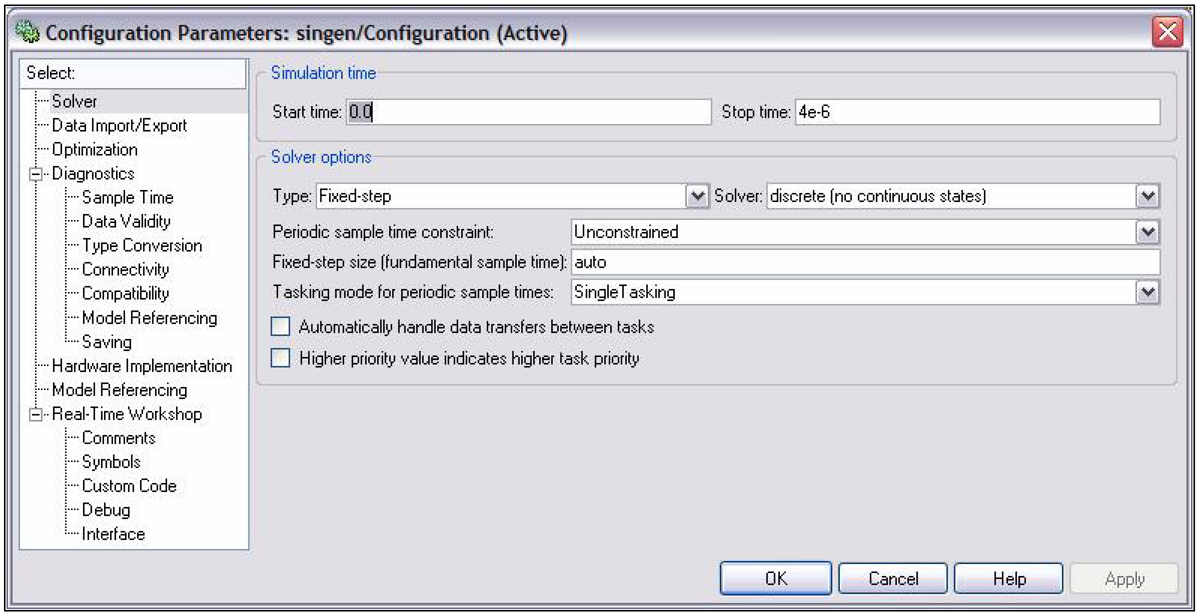
1. Click **Configuration Parameters** on the Simulation menu to display the **Configuration Parameters** dialog box and select the **Solver** page (Figure 2–17 on page 2–17).
2. Set the parameters shown in Table 2–15.

Table 2–15. Configuration Parameters for the singen Model

Parameter	Value
Start time	0.0
Stop time	4e–6
Type	Fixed-step
Solver	discrete (no continuous states)

Refer to the description of the “Solver Pane” in the Simulink Help for detailed information about solver options.

Figure 2–17. Configuration Parameters



Configuration Parameters: singen/Configuration (Active)

Simulation time

Start time: 0.0 Stop time: 4e-6

Solver options

Type: Fixed-step Solver: discrete (no continuous states)

Periodic sample time constraint: Unconstrained

Fixed-step size (fundamental sample time): auto

Tasking mode for periodic sample times: SingleTasking

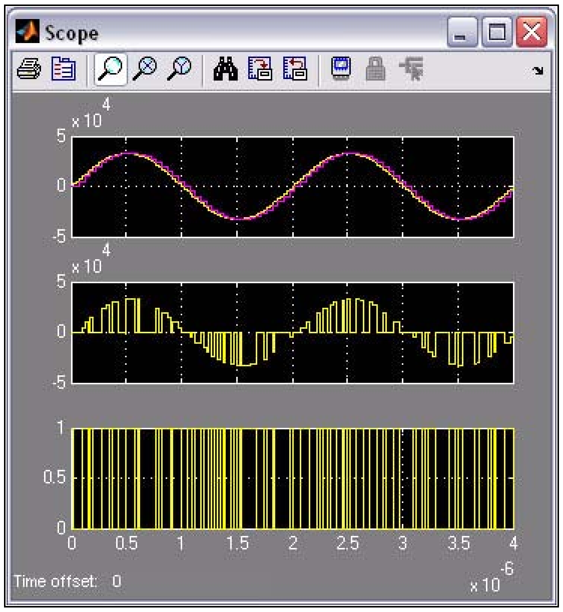
☐ Automatically handle data transfers between tasks

☐ Higher priority value indicates higher task priority

3. Click **OK**.
4. Start simulation by clicking **Start** on the Simulation menu.
5. Double-click the `Scope` block to view the simulation results.
6. Click the **Autoscale** icon (binoculars) to auto-scale the waveforms.

   Figure 2–18 shows the scaled waveforms.

Figure 2–18. Scope Simulation Results



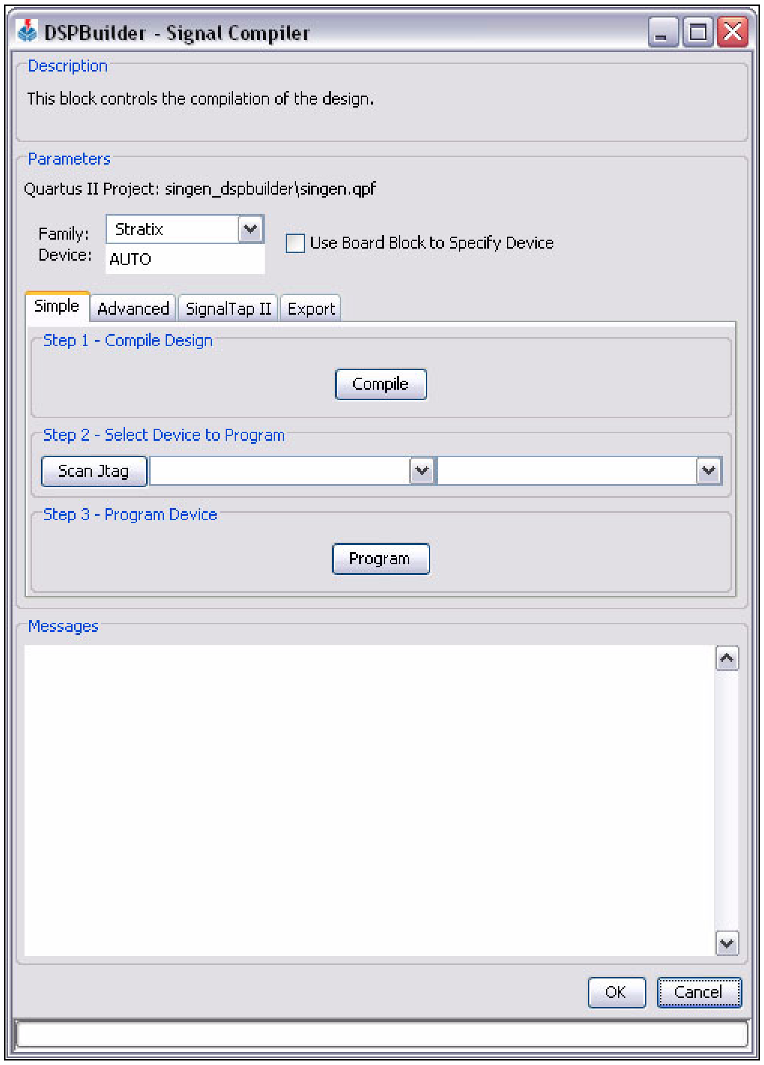
Compiling the Design

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.
3. Double-click the `Signal Compiler` block in your model to display the **Signal Compiler** dialog box (Figure 2–19).

Figure 2–19. Signal Compiler Block Dialog Box


DSPBuilder - Signal Compiler

Description

This block controls the compilation of the design.

Parameters

Quartus II Project: singen_dspbuilder\singen.qpf

Family: Stratix
Device: AUTO

Use Board Block to Specify Device

The dialog box allows you to set the target device family. For this tutorial, you can use the default **Stratix** device family.

4. Click **Compile**.

5. When the compilation has completed successfully, click **OK**.
6. Click **Save** on the File menu to save the model.

Performing RTL Simulation

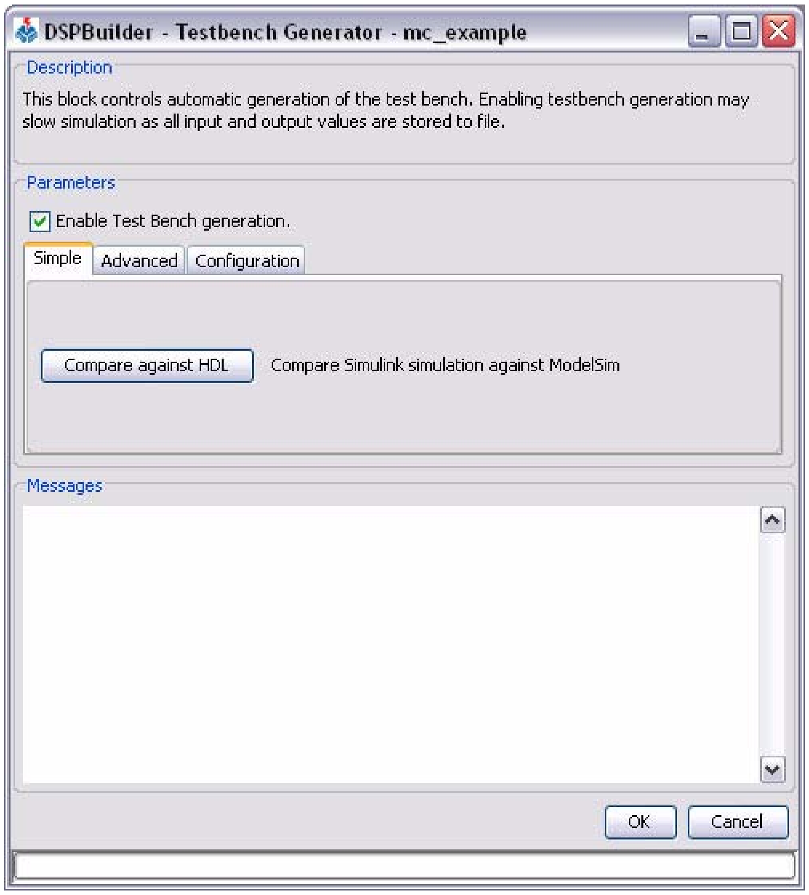
To perform RTL simulation with the ModelSim software, you need to add a `TestBench` block.

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


DSPBuilder - Testbench Generator - mc_example

Description

This block controls automatic generation of the test bench. Enabling testbench generation may slow simulation as all input and output values are stored to file.

Parameters

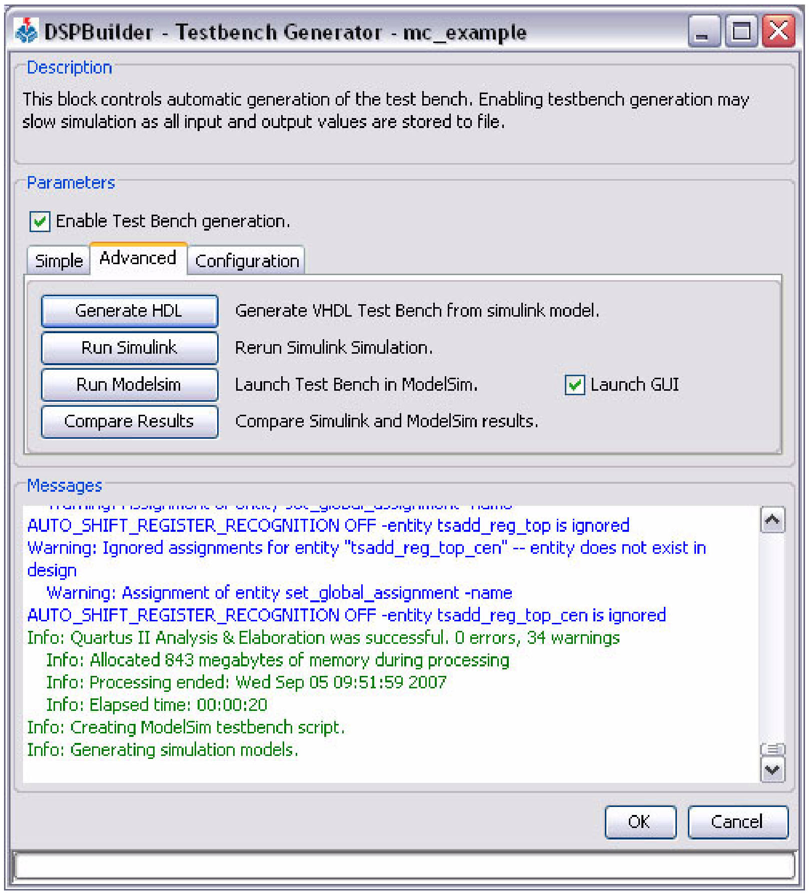
☑ Enable Test Bench generation.

Compare Simulink simulation against ModelSim

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

5. Click the **Advanced** tab (Figure 2–21).

Figure 2–21. Testbench Generator Dialog Box Advanced Tab



This block controls automatic generation of the test bench. Enabling testbench generation may slow simulation as all input and output values are stored to file.

[✓] Enable Test Bench generation.

Generate HDL — Generate VHDL Test Bench from simulink model.
Run Simulink — Rerun Simulink Simulation.
Run Modelsim — Launch Test Bench in ModelSim. [✓] Launch GUI
Compare Results — Compare Simulink and ModelSim results.

```
AUTO_SHIFT_REGISTER_RECOGNITION OFF -entity tsadd_reg_top is ignored
Warning: Ignored assignments for entity "tsadd_reg_top_cen" -- entity does not exist in
design
   Warning: Assignment of entity set_global_assignment -name
AUTO_SHIFT_REGISTER_RECOGNITION OFF -entity tsadd_reg_top_cen is ignored
Info: Quartus II Analysis & Elaboration was successful. 0 errors, 34 warnings
   Info: Allocated 843 megabytes of memory during processing
   Info: Processing ended: Wed Sep 05 09:51:59 2007
   Info: Elapsed time: 00:00:20
Info: Creating ModelSim testbench script.
Info: Generating simulation models.
```

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.

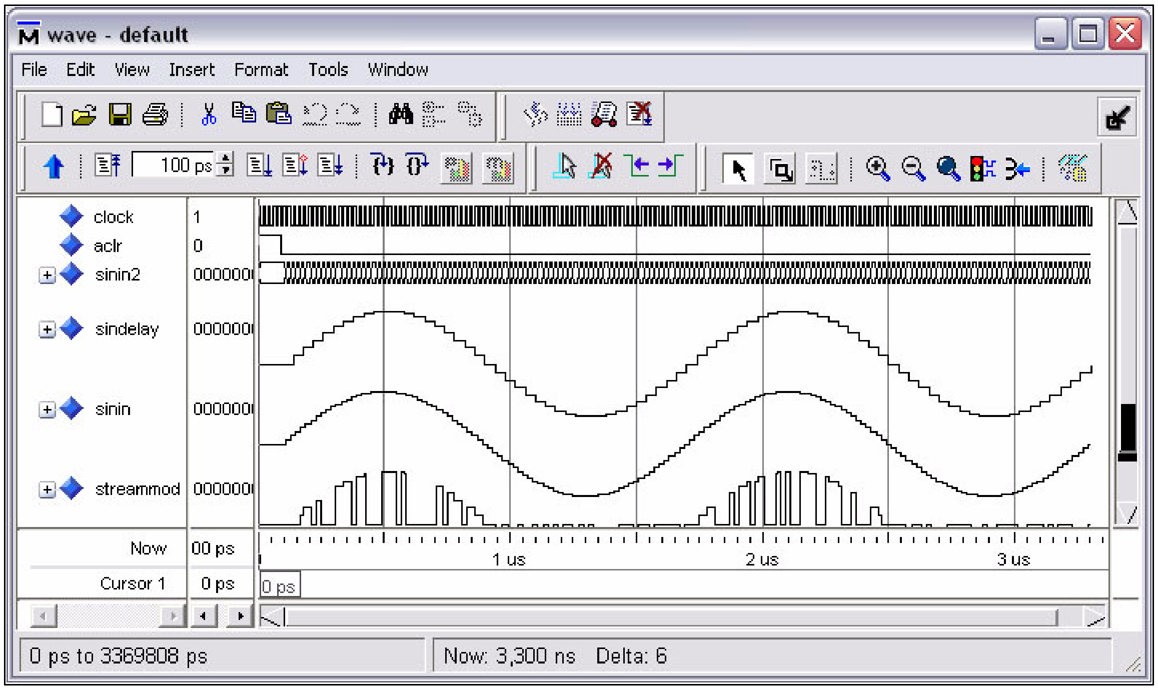
   All waveforms are initially shown using digital format in the ModelSim Wave window.

   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 selecting the signal name in the Wave window and right-clicking on **Properties**. In the **Format** tab, select **Analog**, and specify height **50**, scale **0.001**.

11. Click **Zoom Full** on the right button pop-up 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



0 ps to 3369808 ps

Now: 3,300 ns Delta: 6

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.

Subsequent 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 the `Signal Compiler` block 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 2–18.

Creating a Quartus II Project

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

1. Start the Quartus II software.
2. Click **New Project Wizard** on 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 **Add Files** page. There are no files to add for this tutorial.
5. Click **Next** to display the **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 2–18.
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. On the View menu in the Quartus II software, point to **Utility Windows** and click **Tcl Console** 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 following 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 **singen.mdl** and click **Select Set as Top-Level Entity**.
5. Compile the Quartus II design by clicking **Start Compilation** on the Processing menu.

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

3. Design Rules and Procedures

DSP Builder Naming Conventions

DSP Builder generates VHDL files for simulation and synthesis. When there are blocks or ports in the model that share the same VHDL name, they are given unique names 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. Try to avoid name clashes on other ports, as this results in renaming of the top level ports in the VHDL.

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.
- Do not use 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** command in the right button pop-up menu for each 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

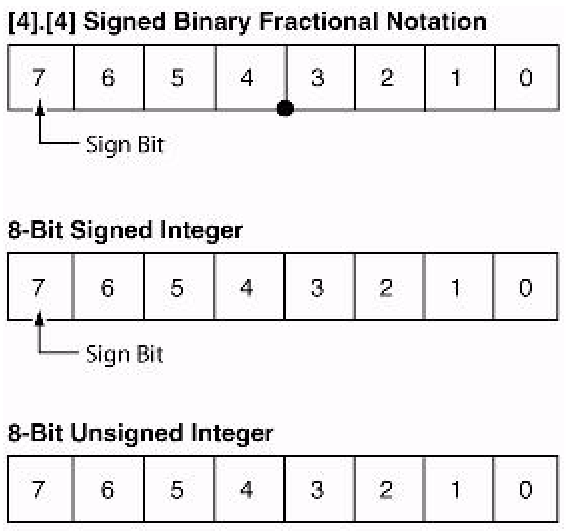
Notes to Table 3–1:

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

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 point of view, multiplying a number by –0.75 or –3 is very different, especially when looking at the bit width growth. In the first case, the multiplier output bus grows on the most significant bit (MSB), in the second 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 representation of the signal. 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*.

Bit Width Design Rule

You must specify the bit width at the source of the data path. 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 tutorial 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.

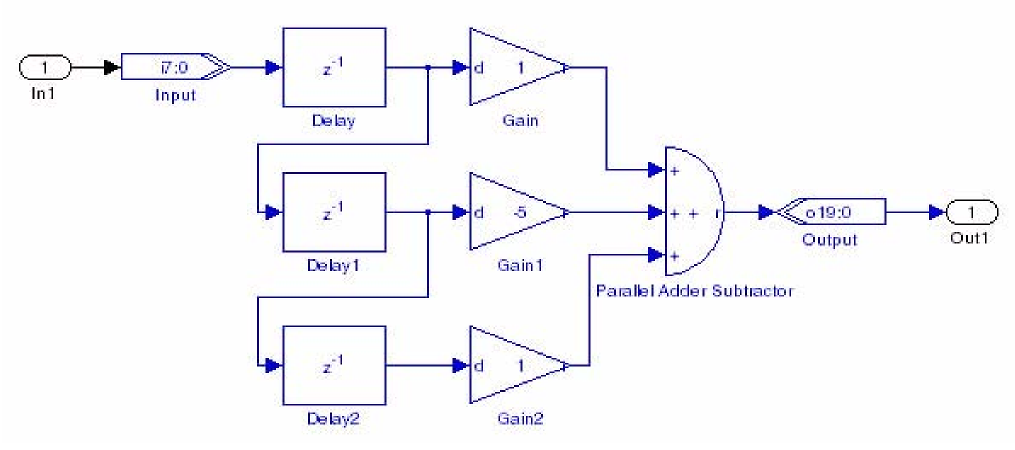
Data Width Propagation

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 3–2).

Figure 3–2. 3-Tap FIR Filter

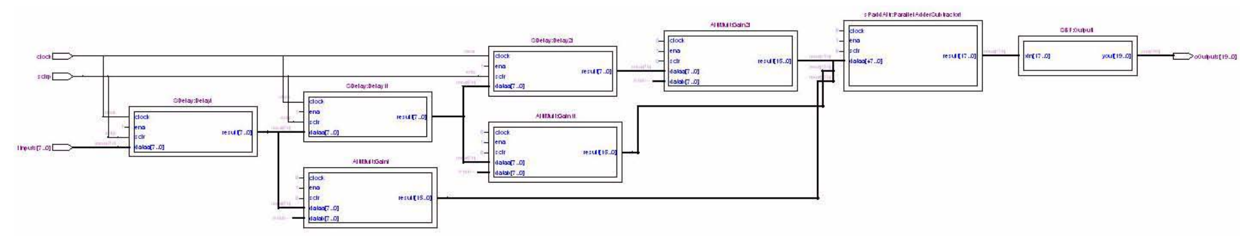


The **fir3tapsub.mdl** design is a 3-tap finite impulse response (FIR) filter and has the following attributes:

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

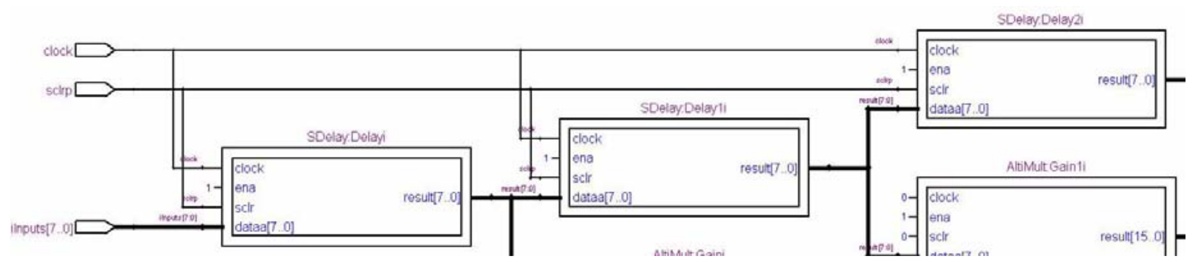
Figure 3–3. 3-Tap FIR Filter in Quartus II RTL View



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 3–4.

Figure 3–4. Tap Delay Line in Quartus II Version RTL Viewer



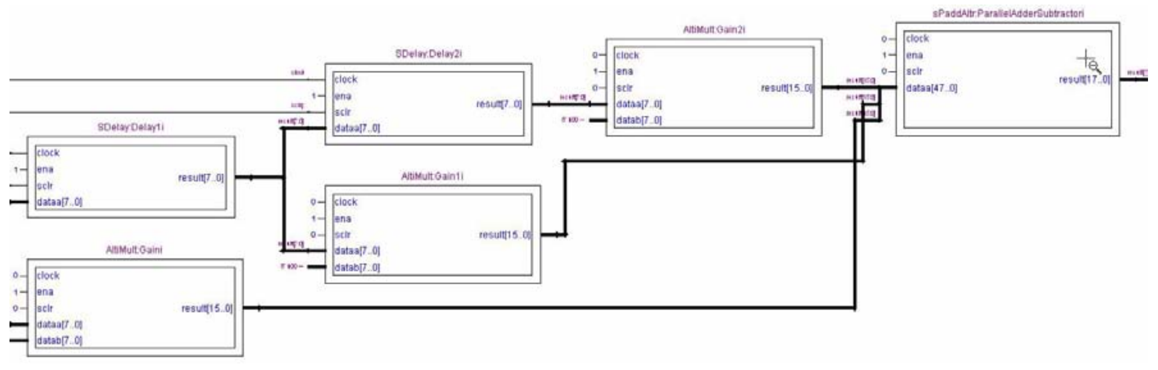
Arithmetic Operation

Figure 3–5 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 3–5. 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:

  $[la].[ra] \times [lb].[rb]$

  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:

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

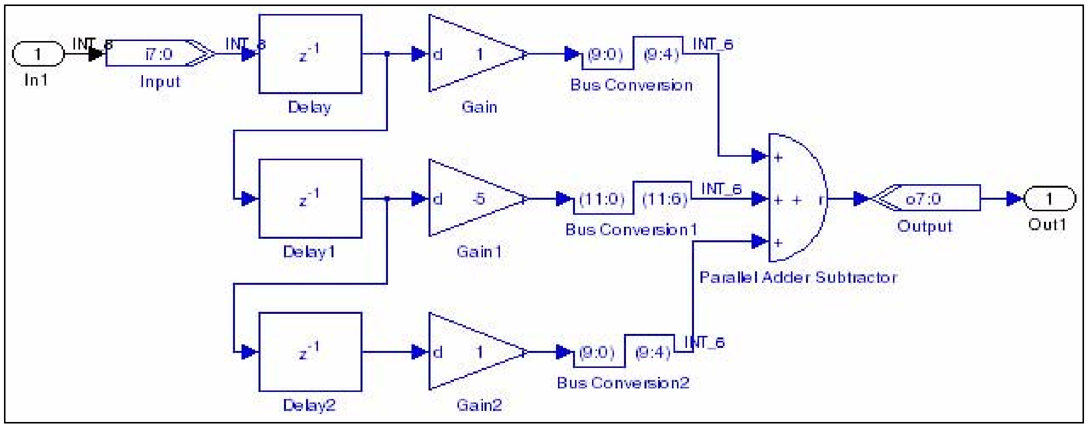
The parallel adder has three input buses of 14, 16, and 14 bits. To perform this addition in binary, DSP Builder automatically sign extends the 14 bit buses to 16 bits. The output bit width of the parallel adder is 18 bits, which covers the full resolution.

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 removes any unused logic.
- Insert a `Bus Conversion` block to change the internal signal bit width.

Figure 3–6 shows how `Bus Conversion` blocks can be used to control internal bit widths.

Figure 3–6. 3-Tap Filter with BusConversion to Control Bit Widths

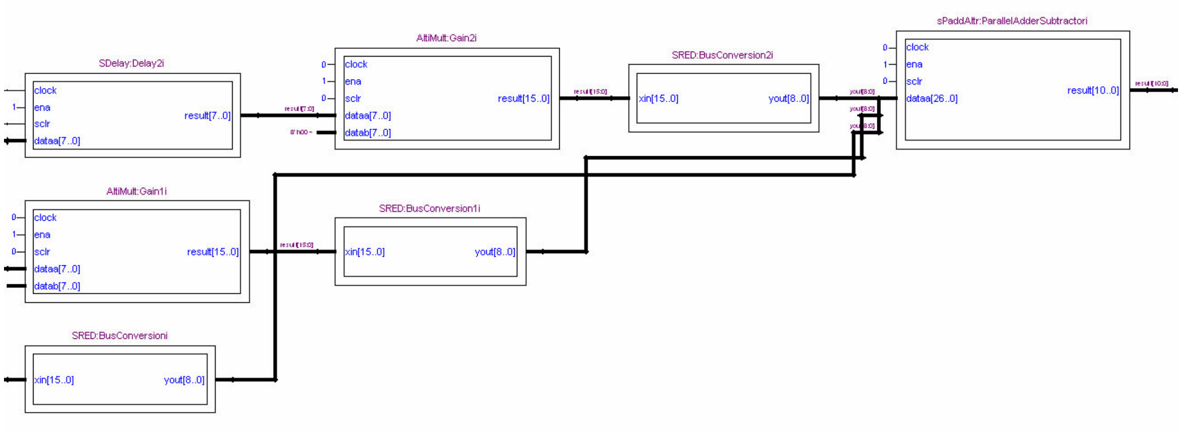


In this example, the output of the `Gain` block has 4 bits removed. (Port data type display is enabled in this example and shows that the inputs to the `Delay` blocks are of type `INT_8` but the outputs from the `Bus Conversion` blocks are of type `INT_6`.)

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 3–7.

Figure 3–7. 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.

Frequency Design Rules

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 **Block Parameters** 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 when you are using a single clock domain.

Refer to "Design Flow" on page 1–3 for information about setting the simulink solver.

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 3–3 on page 3–13 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 must set a variable-step discrete solver in Simulink when you are using multiple clock domains.

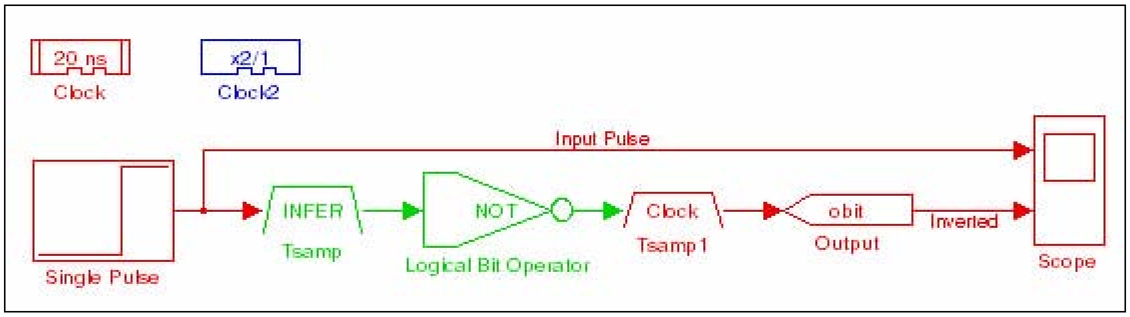
Refer to "Design Flow" on page 1–3 for information about setting the Simulink solver.

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 3–8 illustrates an incorrect use of the DSP Builder `Logical Bit Operator` (NOT) block.

Figure 3–8. Example of Incorrect Usage: Mixed Sampling Rate on a NOT Block



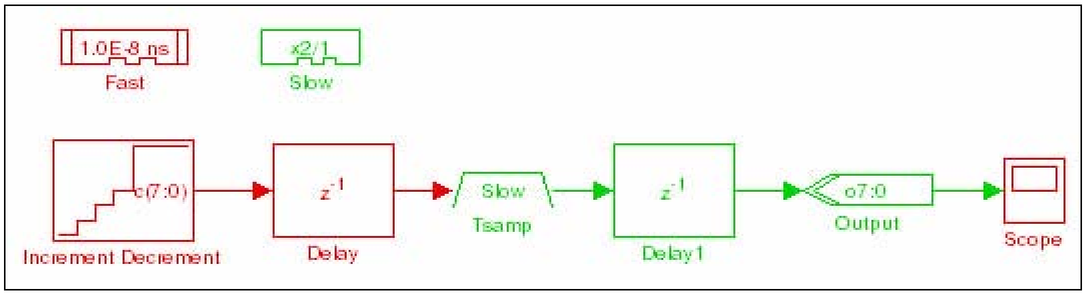
- Two DSP Builder blocks can operate with two different sampling periods. However within most DSP Builder blocks the sampling period of each input port and each output port must be identical.

  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.

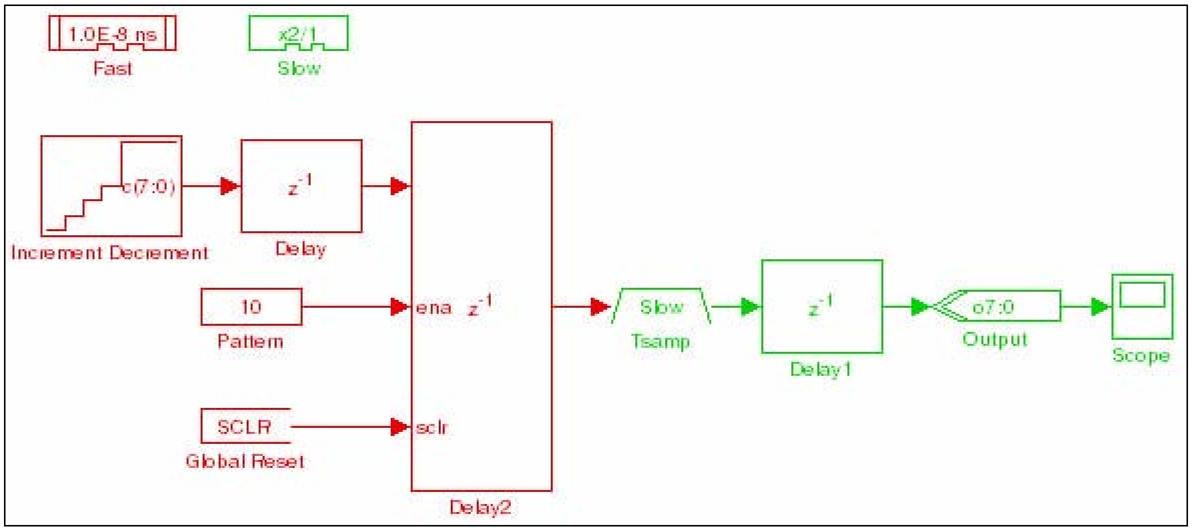
  This requirement is especially true when the source data rate is higher than the destination register, in other words, when the data of a register is toggling at the higher rate than the register's clock pin, as shown in Figure 3–9.

Figure 3–9. Data Toggling Faster than Clock



A stable hardware implementation is shown in Figure 3–10.

Figure 3–10. 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.

If you use `Clock_Derived` blocks, and there is only one system clock, you must generate an appropriate clock signal for connection to the hardware device input pins for the derived clocks.

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.

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 box. The clock domains of the `PLL` block share the reset pin of the `PLL` block's input clock.

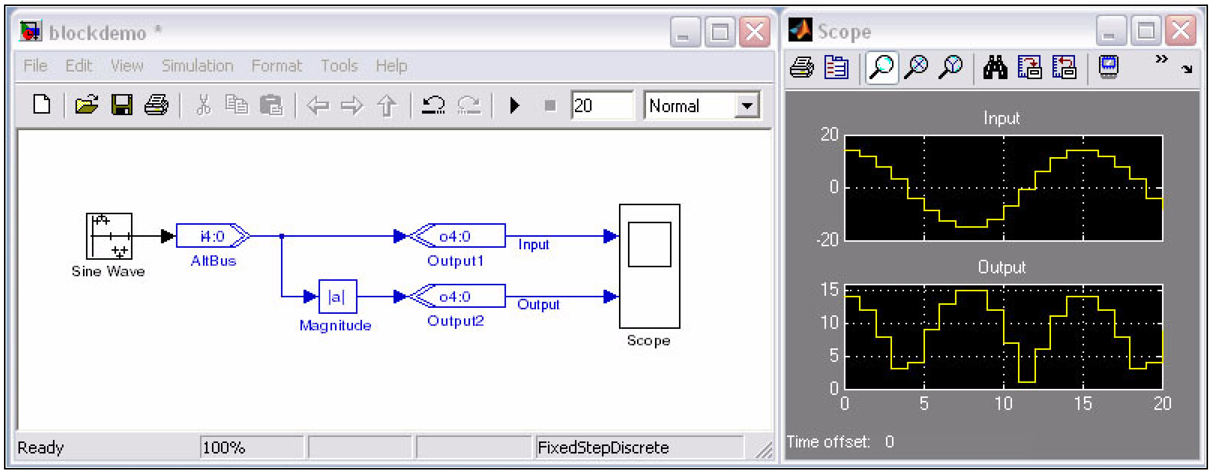
When the design contains clock source blocks, DSP Builder implicitly connects the clock pins of all the registered blocks to the appropriate clock pin or PLL output. DSP Builder also connects the reset pins of the registered blocks to the top-level reset port for the block's clock domain.

DSP Builder blocks fall into the following 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 3–11 illustrates DSP Builder block combinational behavior.

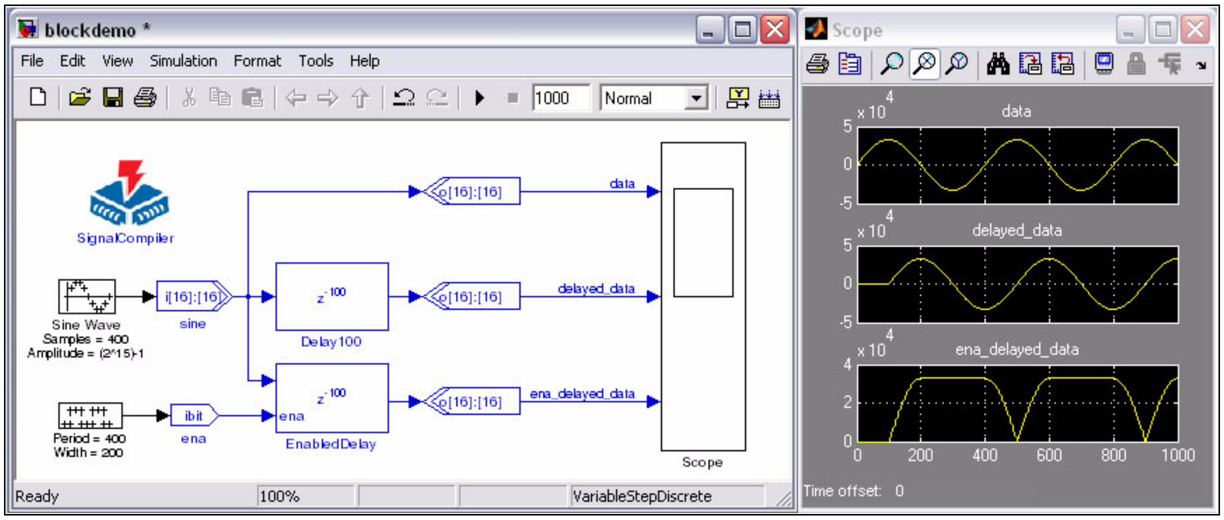
Figure 3–11. Magnitude Block: Combinational Behavior



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

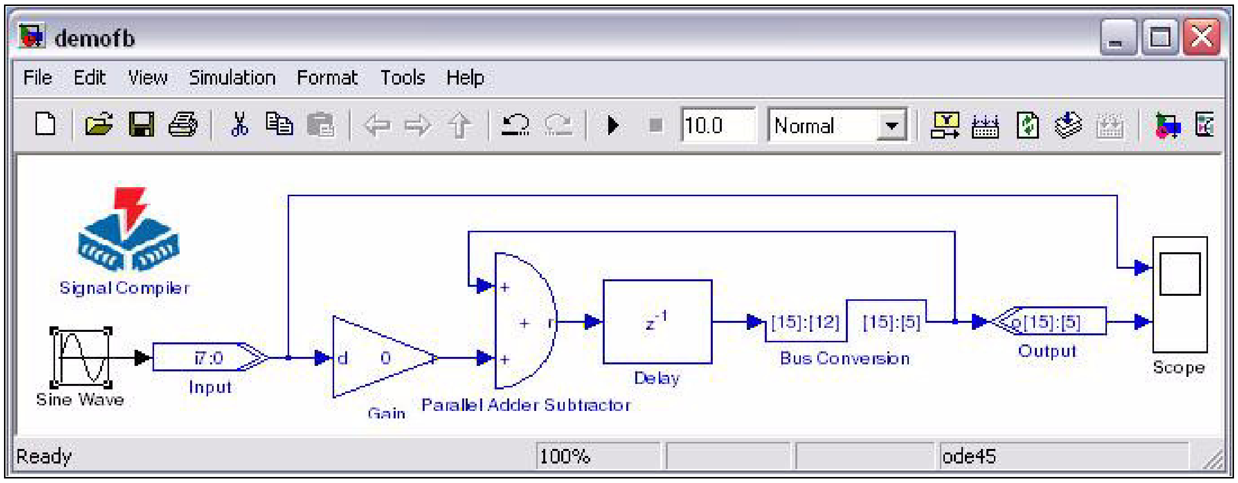
Figure 3–12 illustrates the behavior of a registered DSP block. In the VHDL netlist, DSP Builder adds clock pin inputs to this function. The `Delay` block, with the **Clock Phase Selection** parameter equal to 100, is converted into a VHDL shift register with a decimation of three and an initial value of zero.

Figure 3–12. 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 3–13.

Figure 3–13. 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.

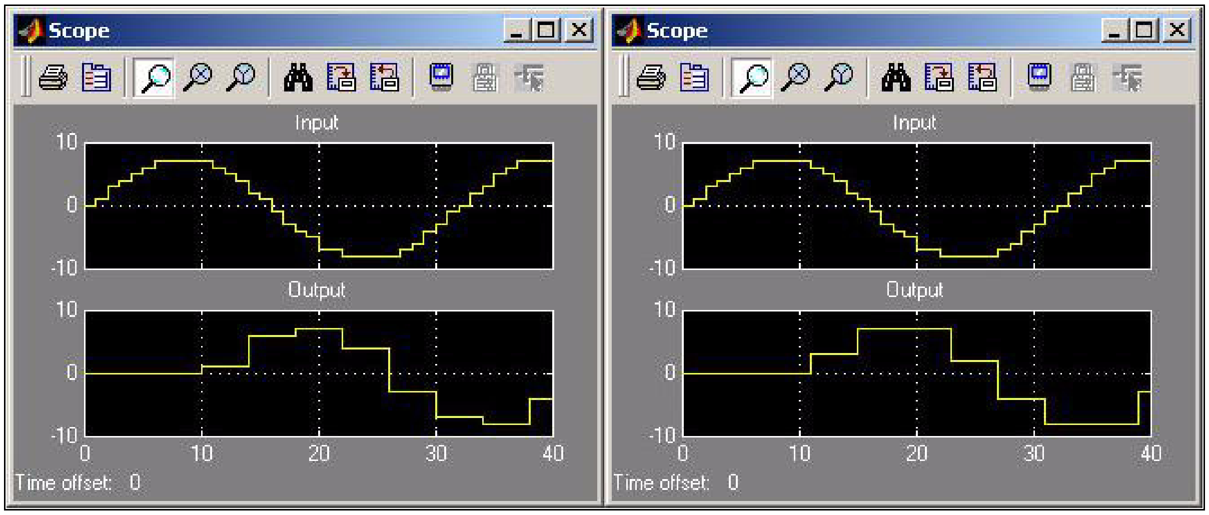
  For example, see Table 3–2.

Table 3–2. 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 3–14 compares the scopes for the `Delay` block operating at a one quarter rate on the `1000` and `0100` phases, respectively.

Figure 3–14. 1000 as Opposed to 0100 Phase Delay



Using the PLL Block

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

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

Table 3–3. Device Support for PLL Clocks

Device Family	Number of PLL Clocks
Stratix IV	9
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 3–15 shows an example of multiple-clock domain support using the `PLL` block.

Figure 3–15. MultipleClockDelay.mdl

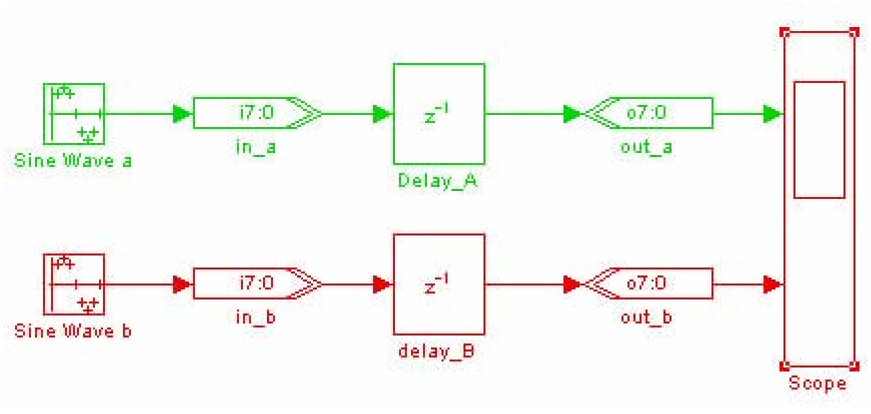
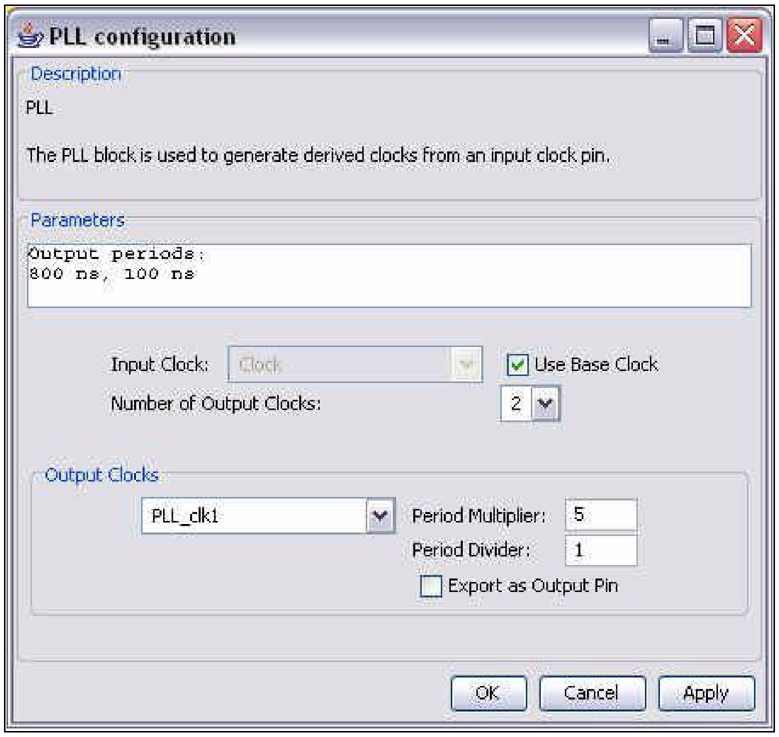


Figure 3–16 shows the clock setting configuration for the `PLL` block in the example design **MultipleClockDelay.mdl**. Output clock `PLL_clk0` is set to 800 ns, and output clock `PLL_clk1` is set to 100 ns.

Figure 3–16. PLL Setting



PLL

The PLL block is used to generate derived clocks from an input clock pin.

```
Output periods:
800 ns, 100 ns
```

Input Clock: Clock
[✓] Use Base Clock
Number of Output Clocks: 2
PLL_clk1
Period Multiplier: 5
Period Divider: 1
[ ] Export as Output Pin

Data path A (shown in green in Figure 3–15) operates on output clock *PLL_clk0* and data path B (shown in red in Figure 3–15) 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 using the following method:

- Create a new Quartus II project and use the MegaWizard Plugin Manager to configure the `ALTPLL` block.
- Add the DSP Builder **.mdl** file to the Quartus II project 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 and 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 1–2 on page 1–5). 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 3–17 on page 3–17).

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 **Block Parameters** 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.

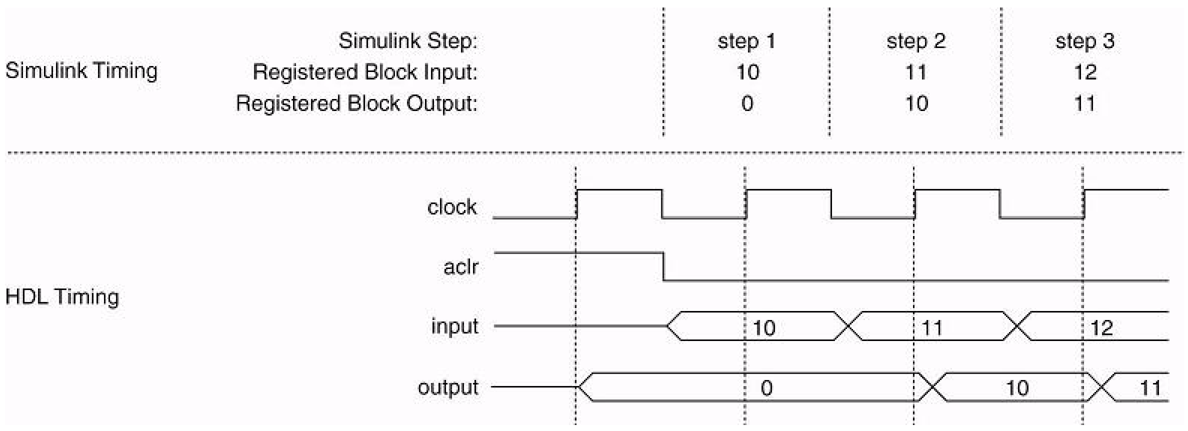
The reset logic polarity can be either active-high or active-low. When active-low is selected, the value of the reset signal in Simulink simulation is still 0 for inactive and 1 for active. However, a `NOT` gate is inserted on the input pin in the hardware that is generated. The value of the reset signal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

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

Reference Timing Diagram

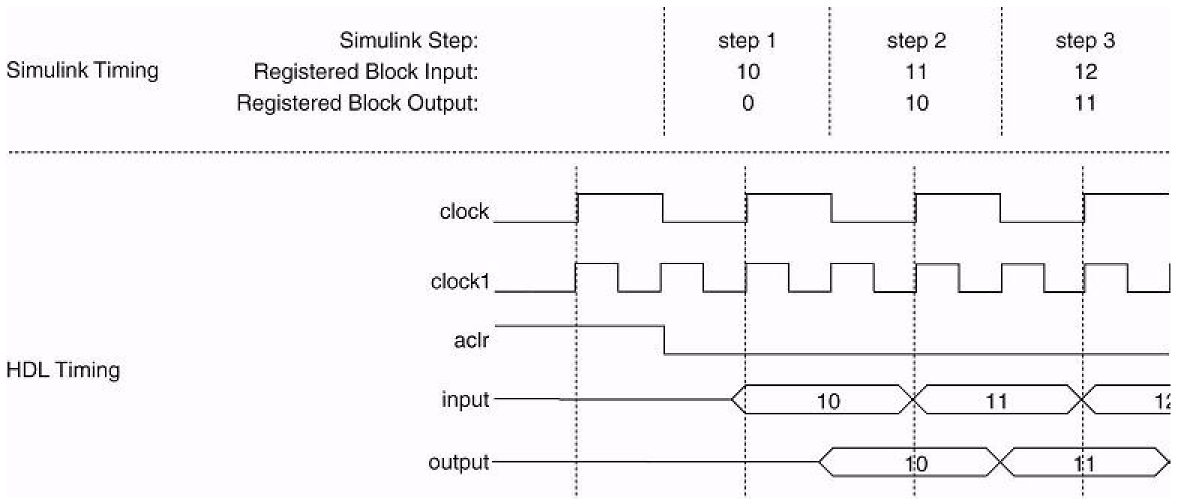
Figure 3–17 shows the timing relationships in a hypothetical case where a register is fed by the output of a counter. The counter output begins at 10, in other words, the value is 10 during the first Simulink clock step.

Figure 3–17. Single-Clock Timing Relationships



This timing is not true when crossing clock domains. For example, Figure 3–18 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 3–18. Multiple-Clock Timing Relationships



Signal Compiler and TestBench Blocks

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`. The simulation updates all variables in the design (including workspace variables and inherited parameters), sets up certain blocks (such as the memory blocks, and inputs from and outputs to workspace blocks), and also traps any design errors that do not comply with Simulink rules.

The `Input` and `Output` blocks map to input and output ports in VHDL and mark the edge of the generated system. Typically, you connect these blocks to the 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), empty ports are created and the HDL 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 and 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.

Refer to page 2–15 of the “Getting Started Tutorial” for an example that uses the `Signal Compiler` block. For information about setting parameters for the `Signal Compiler` block, refer to the *Signal Compiler* section in the *AltLab Library* chapter of the *DSP Builder Reference Manual*.

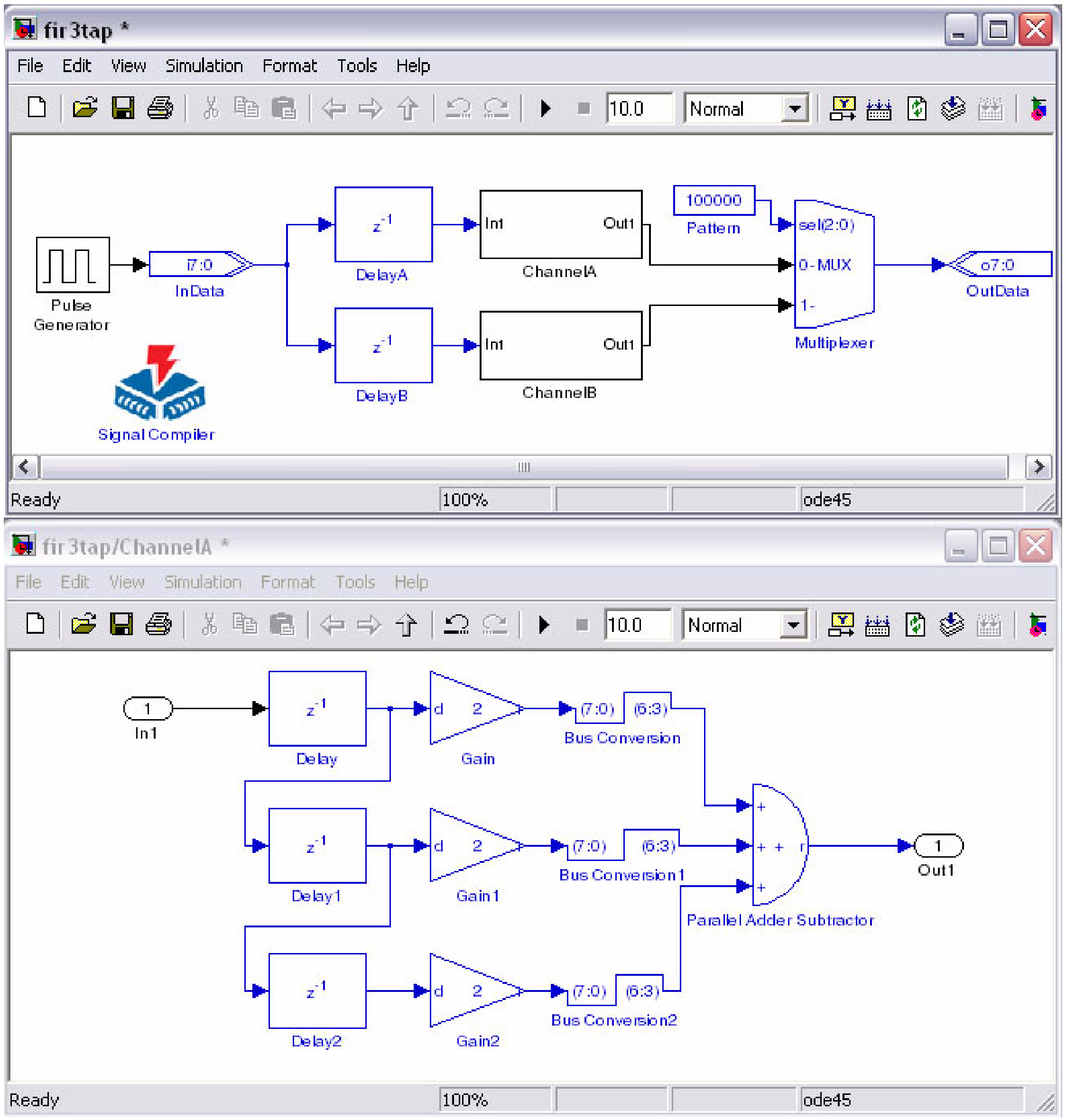
Hierarchical Design

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

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

For example, Figure 3–19 illustrates a hierarchy for a design **fir3tap.mdl**, which implements two FIR filters.

Figure 3–19. Hierarchical Design Example



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

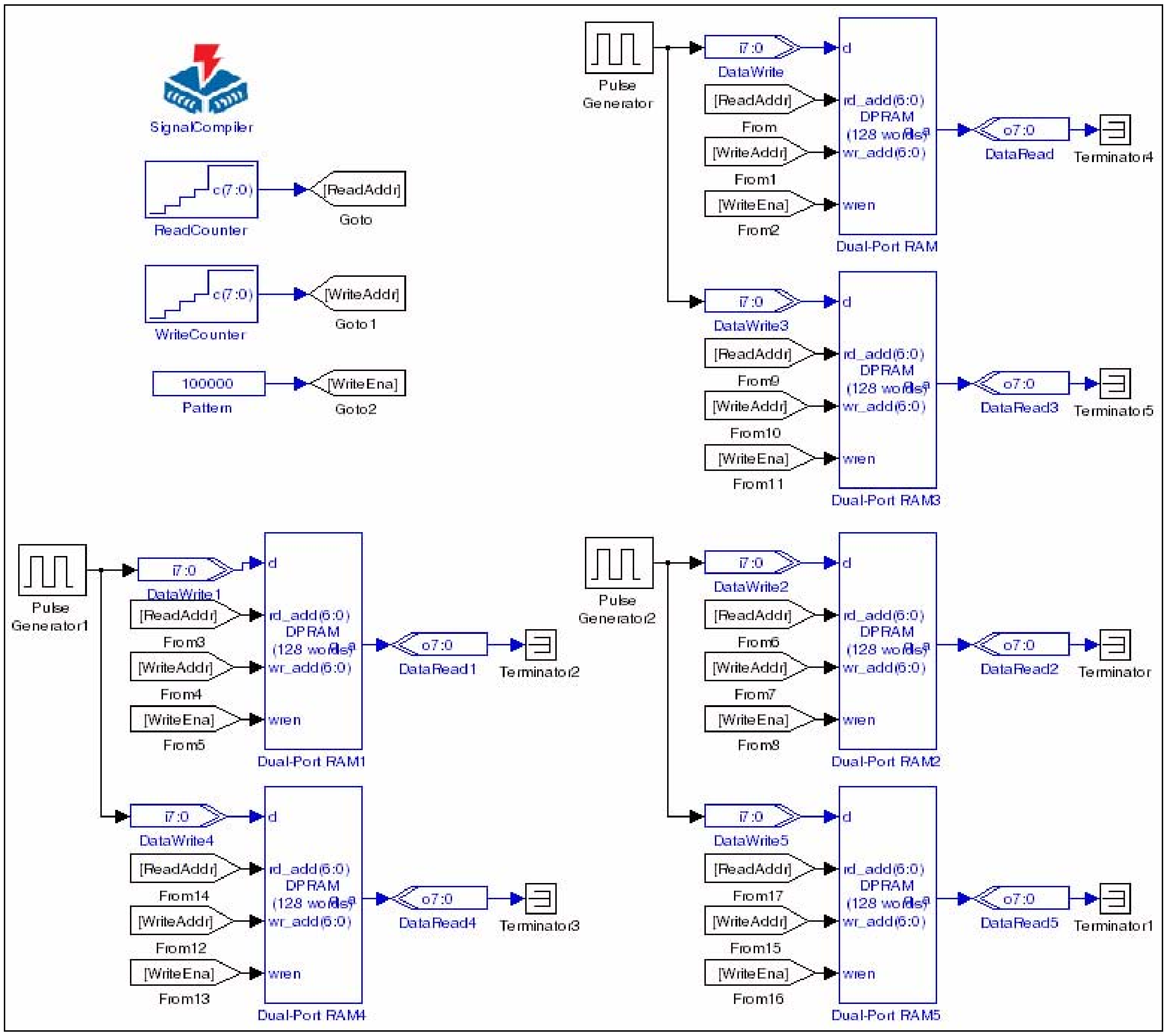
Goto and From Block Support

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 3–20 shows an example of the `Goto` and `From` blocks.

Figure 3–20. Goto & From Block Example



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.

Black Boxing and 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 8.

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 8.

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.

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')↵
```

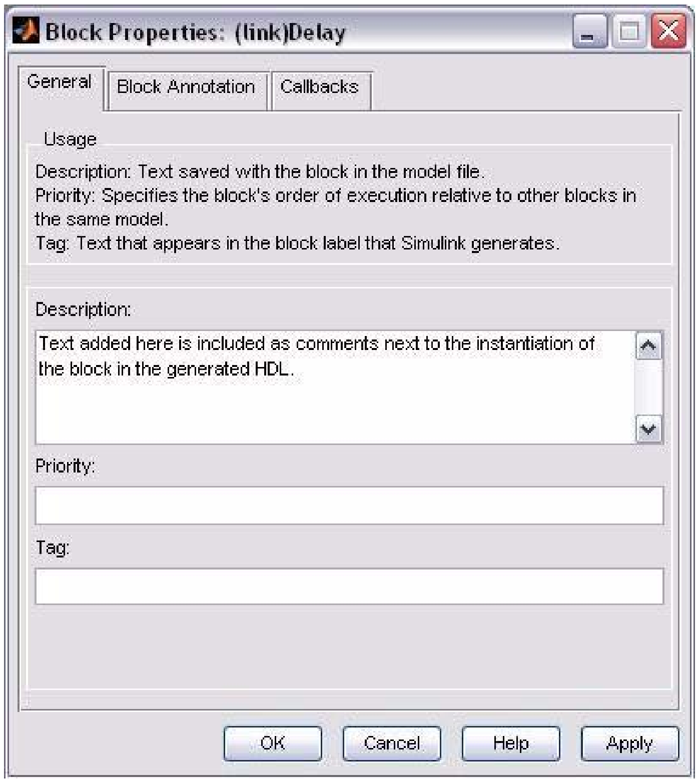
A testbench GUI is displays messages as the comparison is performed. The command returns true (1) or false (0) according to whether the simulation results match and the output is recorded in the specified log file.

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

Adding 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 3–21.

Figure 3–21. Adding Comments to a Block



Description: Text saved with the block in the model file.
Priority: Specifies the block's order of execution relative to other blocks in the same model.
Tag: Text that appears in the block label that Simulink generates.

Description:
Text added here is included as comments next to the instantiation of the block in the generated HDL.

Priority:

Tag:

The comment text is included next to the instantiation of the block in the generated HDL.

Adding Quartus II Constraints

You can set Quartus II global project assignments in your Simulink model by adding `Quartus II Global Project Assignment` blocks from the AltLab library. Each block sets a single global assignment but multiple blocks can be used for multiple assignments. You can use these assignments to set Quartus II compilation directives, such as target device or timing requirements.

For a description of the `Quartus II Global Project Assignment` block, refer to the *DSP Builder Reference Manual*.

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`.

The Tcl file can include any number of Quartus II assignments with the syntax:

```
set_global_assignment -name <assignment> <value>
```

For detailed information about Quartus II assignments, refer to the Quartus II Settings File Reference Manual.

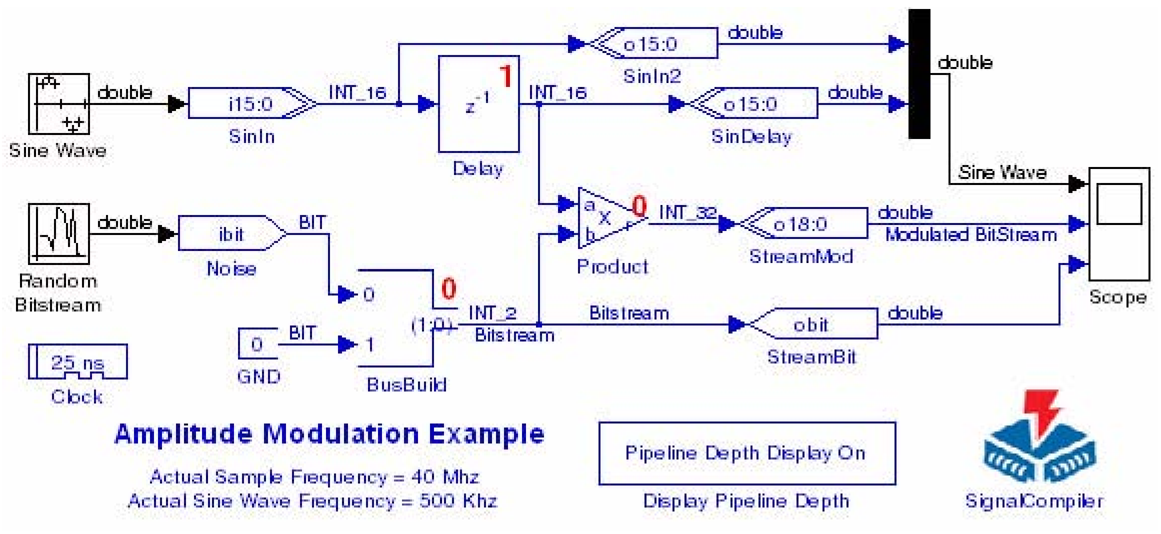
Displaying Port Data Types

You can optionally display the Simulink and DSP Builder port data types for each of the signals in your Simulink model by turning on **Port Data Types** in the **Port/Signal Displays** section of the Simulink Format menu.

When this option is set, the DSP Builder internal signal type (*S*BF_L_R, INT_L, UINT_L, or BIT where L, and R are the number of bit to the left and right of the binary point) is displayed. For example, SBF_8_4 for a 12-bit signed binary fractional data type with 4 fractional bits, or UINT_16 for a 16-bit unsigned integer.

Figure 3–22 shows the Amplitude Modulation example from the Getting Started Tutorial with port data type display enabled.

Figure 3–22. Tutorial Example Showing Port Data Types and Pipeline Depth



For more information about the DSP Builder internal signal types, refer to "Fixed-Point Notation" on page 3–2.

Displaying the Pipeline Depth

You can optionally display the pipeline depth on the primitive blocks (such as the Arithmetic library blocks) in your Simulink model by adding a `Display Pipeline Depth` block from the AltLab library.

You can change the display mode by double-clicking on the block. When set, the current pipeline depth is displayed at the top right corner of each block that adds latency to the design as shown in Figure 3–22. The currently selected mode is shown on the `Display Pipeline Depth` block symbol.

Updating HDL Import Blocks

The `HDL Import` blocks in your design may need updating if you have upgraded from a previous software version or have moved a design to a different workstation.

You can use the `alt_dspbuilder_refresh_hdlimport` command to update these blocks. This command checks that the referenced HDL files (or Quartus II project) exists. If found, the **HDL Import** dialog box is opened and a compilation is automatically invoked to regenerate the Simulink model. If neither is found, but there is an existing simulation netlist, this netlist is used for simulation.

To run the command, perform the following steps:

1. Start the MATLAB/Simulink software.
2. Open a Simulink model that contains imported HDL.
3. Run the command by typing the following at the MATLAB prompt:

   ```
   alt_dspbuilder_refresh_hdlimport ↵
   ```

You can optionally select a `HDL Import` block to run the command on the selected subsystem only.

Analyzing the Hardware Resource Usage

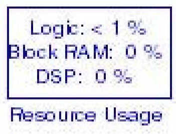
You can analyze the hardware resources required for your design by using a `Resource Usage` block.

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 `Resource Usage` block into your model and double-click on the block to open the **Resource Usage** dialog box.
3. Double-click on the `Signal Compiler` block and click **Compile** to re-compile the design in the Quartus II software.

   The `Resource Usage` block is updated to show a summary of the estimated logic, RAM and DSP block usage (Figure 3–23).

Figure 3–23. Resource Usage Block

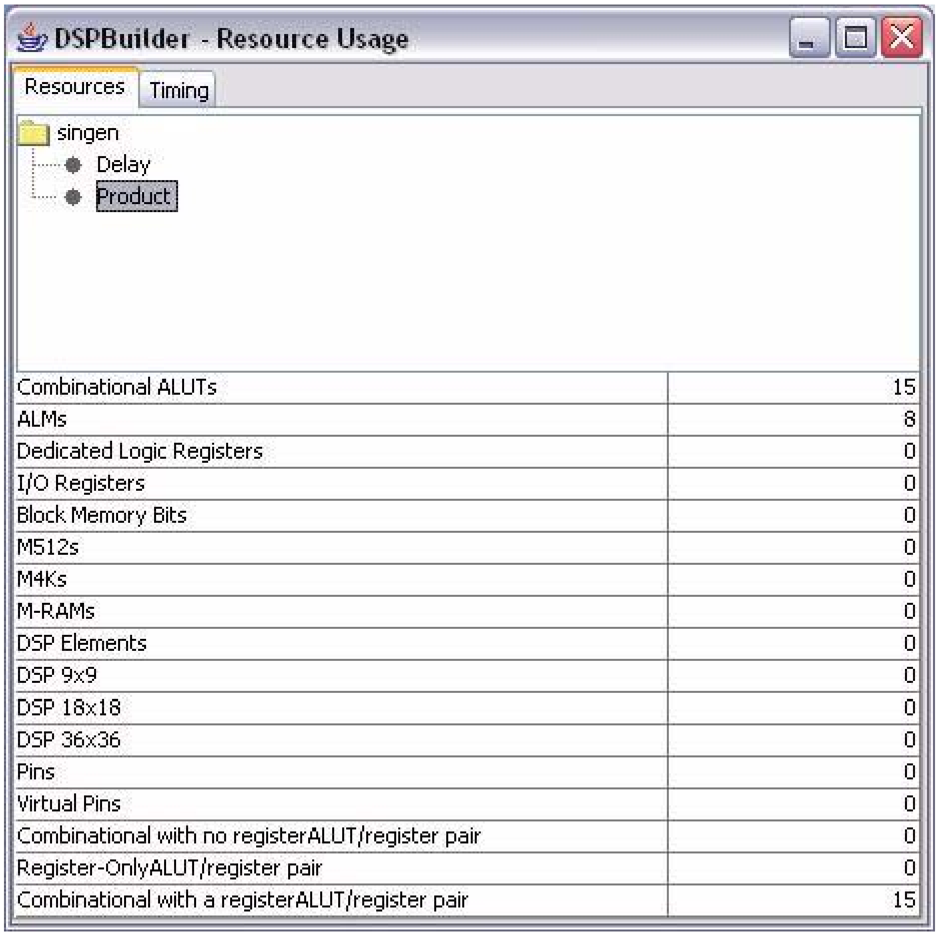


Logic: < 1 %
Block RAM: 0 %
DSP: 0 %

   The **Resource Usage** dialog box is updated to show a detailed report of the resources required by each of the blocks in your model that generate hardware.

For example, Figure 3–24 shows the hardware resources required by the `Product` block in the Amplitude Modulation example from the Getting Started Tutorial.

Figure 3–24. Resource Usage Dialog Box



Combinational ALUTs	15
ALMs	8
Dedicated Logic Registers	0
I/O Registers	0
Block Memory Bits	0
M512s	0
M4Ks	0
M-RAMs	0
DSP Elements	0
DSP 9x9	0
DSP 18x18	0
DSP 36x36	0
Pins	0
Virtual Pins	0
Combinational with no registerALUT/register pair	0
Register-OnlyALUT/register pair	0
Combinational with a registerALUT/register pair	15

The information displayed depends on the selected device family. Refer to the device documentation for more information.

You can also click the **Timing** tab and click **Highlight path** to highlight the critical paths on your design.

When the source and destination shown in the dialog box are the same and a single block is highlighted, the critical path is due to the internal function or a feedback loop. For a more complex example, the entire critical path through the design may be highlighted.

Loading Additional ModelSim Commands

When you import HDL as a black box, DSP Builder creates a subdirectory named **DSPBuilder<***model name***>_import**. 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.

Making Quartus II Assignments to Block Entity Names

The VHDL entity names of the blocks in a DSP Builder design are dependent on the block's parameter values. This means that blocks of the same type and same parameterization share a common VHDL entity.

The entity names have the following format:

```
<block type name>_GN<8 alphanumeric characters>
```

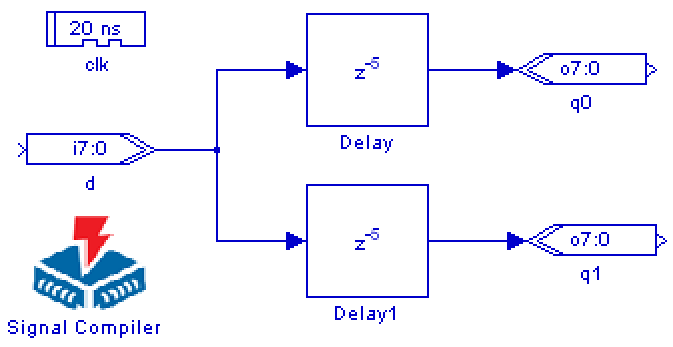
For example, a `Delay` block entity name might be:

```
alt_dspbuilder_delay_GNLVAGVO3B
```

Changing the parameterization of the block causes the entity name to change. If you want to make an assignment to a block in the Quartus II project, and for the assignment to remain when the block parameters are changed, you can use regular expressions in the assignments.

For example, you may want to make a **Preserve Registers** assignment to the `Delay` blocks in Figure 3–25 to prevent them from being merged.

Figure 3–25. Entity Name Assignment Example



Using the Quartus II Assignment Editor and Node Finder tools, you can identify the names of the registers and make the assignments to them. For example, if the model is named `my_model`, the names might be:

```
my_model_GN:auto_inst|alt_dspbuilder_delay_GNLVAGVO3B:Delay|alt_dsp
builder_SDelay:Delay1i|DelayLine
```

```
my_model_GN:auto_inst|alt_dspbuilder_delay_GNLVAGVO3B:Delay1|alt_ds
pbuilder_SDelay:Delay1i|DelayLine
```

These assignments prevent merging of the registers. If you change the length of the delay, the assignments are no longer valid. However, you can edit the **To** field of the assignment and use a regular expression that is still valid if the entity name changes due to a parameter change: Replace the eight alphanumeric characters following the `GN` in the block entity name with `.{8}`, which is a regular expression that matches any eight characters. The targets of the assignments then become:

```
my_model_GN:auto_inst|alt_dspbuilder_delay_GN.{8}:Delay|alt_dspbuil
der_SDelay:Delay1i|DelayLine
```

```
my_model_GN:auto_inst|alt_dspbuilder_delay_GN.{8}:Delay1|alt_dspbui
lder_SDelay:Delay1i|DelayLine
```

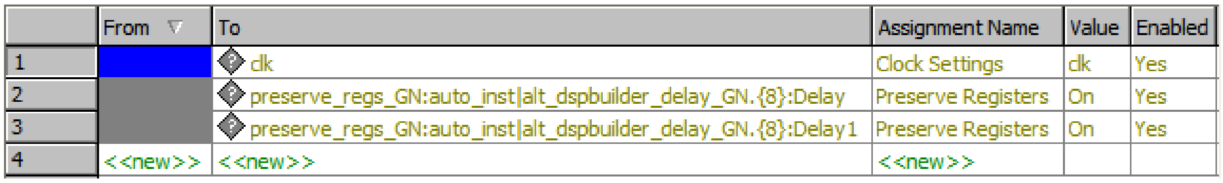
If you want the assignment to apply to the whole block, not just the specific nodes, you can use the following code:

```
my_model_GN:auto_inst|alt_dspbuilder_delay_GN.{8}:Delay
```

```
my_model_GN:auto_inst|alt_dspbuilder_delay_GN.{8}:Delay1
```

Figure 3–26 shows this example in the Quartus II Assignment Editor.

Figure 3–26. Preserve Registers Assignment in the Quartus II Assignment Editor



	From	To	Assignment Name	Value	Enabled
1		clk	Clock Settings	clk	Yes
2		preserve_regs_GN:auto_inst|alt_dspbuilder_delay_GN.{8}:Delay	Preserve Registers	On	Yes
3		preserve_regs_GN:auto_inst|alt_dspbuilder_delay_GN.{8}:Delay1	Preserve Registers	On	Yes
4	<<new>>	<<new>>	<<new>>		

You might want to do this if you are making the assignment to a more complicated block that contains many registers, and you want the assignment to apply to all of them.

Managing Projects and Files

The following files are required to store all the components of a DSP Builder design:

- The top level Simulink model <*top_level_name*>**.mdl**
- The import directory **DSPBuilder_**<*top_level_name*>**_import** and its contents.
- Any source files for imported HDL.
- Any memory initialization (**.hex**) files.
- Any referenced custom library files.

The import directory is required for any design that includes HDL Import, State Machine Editor or MegaCore functions (excluding the Video and Image Processing Suite functions).

Integration with Source Control Systems

Altera recommends that you store Quartus II archive (**.qar**) files rather than individual HDL files for source control purposes.

To create a **.qar** file, perform the following steps in the Quartus II software:

1. Create a Quartus II project that sources the top-level Quartus II IP (**.qip**) file generated by the DSP Builder Export HDL flow as described in "Exporting HDL" on page 3–29.
2. Perform analysis and elaboration to ensure any black box system files are incorporated. (This step may be omitted for simple systems.)
3. Archive the project by clicking **Archive Project** on the project menu in the Quartus II software) to generate the **.qar** file.

Any HDL elements introduced into DSP Builder using custom library blocks may require their own source control. The required files are listed in additional **.qip** files referenced in the "# Imported IP files" section of the top-level **.qip** file.

HDL Import

In general, source files imported using HDL Import are not part of a DSP Builder project but are referenced in projects generated using the Export HDL flow as external files, using absolute paths.

When you move a design to a new version of the tools or to a location on a different computer, run the `alt_dspbuilder_refresh_HDLimport` script to ensure the HDL Import blocks are up-to-date.

When migrating to a new computer, it is necessary to re-import the HDL to enable hardware generation (although simulation in Simulink may be possible without this step).

MegaCore Functions

The MegaCore IP Library is always installed in the same parent directory as the Quartus II installation. Note that this is not a subdirectory of the **quartus** directory but a relative path to an install directory at the same level as the **quartus** directory. The expected directory structure is:

<*install_path*>**<QUARTUS_ROOTDIR>\..\ip**

This allows the Export HDL flow to use relative paths, and improves portability.

Before version 8.0 of the Quartus II software, it was possible to install previous versions of the MegaCore IP Library in any specified location. If an old version of the MegaCore IP Library is used in your design, there may still be absolute paths in the generated Quartus II IP (**.qip)** files which must be modified when you move projects to a different location. The **.qip** file contains all of the assignments and other information required to process the exported HDL in the Quartus II compiler and generate hardware.

When moving a design to a new version of the tools or a different location, run the `alt_dspbuilder_refresh_megacore` script to ensure that the MegaCore function blocks are up-to-date.

Successful migration of designs with MegaCore Functions assumes that the new environment has all the required IP installed. It may be necessary to install the MegaCore IP Library and run the `alt_dspbuilder_setup_megacore` script.

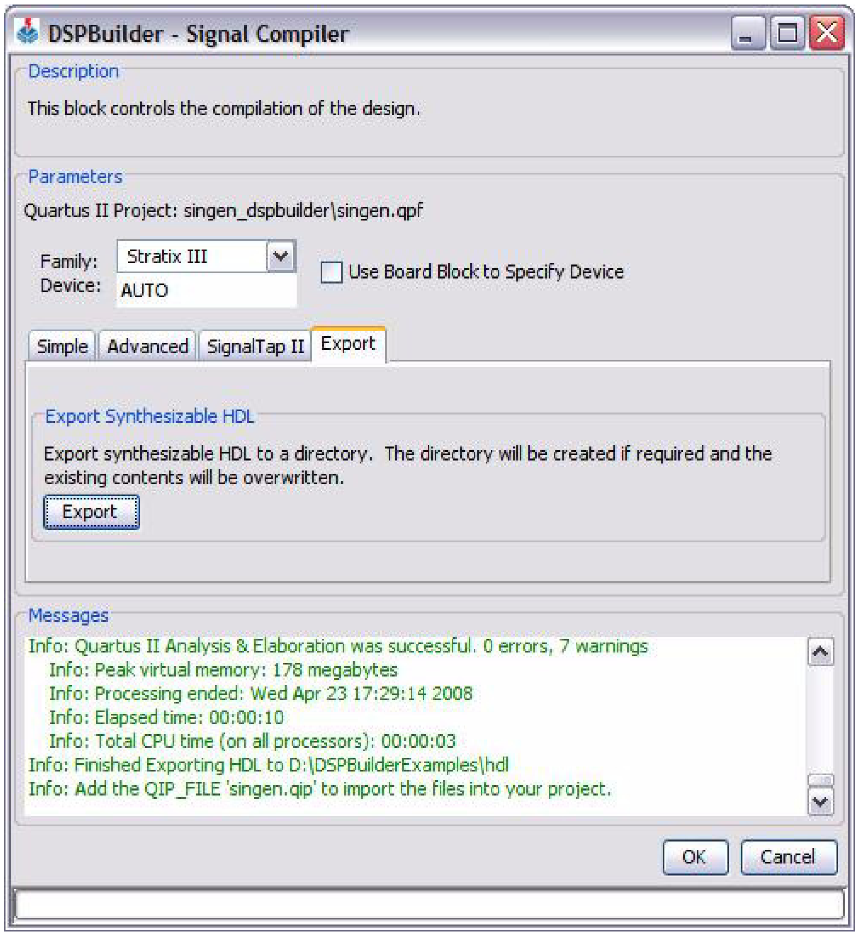
Memory Initialization (.hex) Files

These files are required for simulation and hardware generation purposes. If they are generated by HDL Import or MegaCore function blocks, it is important to ensure that they are located within the import directory. This is generally not the case if the files have been generated using HDL Import.

Exporting HDL

You can export the synthesizable HDL generated using DSP Builder to a Quartus II project can then be used outside DSP Builder by using the **Export** tab in the `Signal Compiler` block (Figure 3–27 on page 3–29).

Figure 3–27. Export HDL Tab in Signal Compiler



This block controls the compilation of the design.

Quartus II Project: singen_dspbuilder\singen.qpf

Family: Stratix III
Device: AUTO

Export synthesizable HDL to a directory. The directory will be created if required and the existing contents will be overwritten.

```
Info: Quartus II Analysis & Elaboration was successful. 0 errors, 7 warnings
   Info: Peak virtual memory: 178 megabytes
   Info: Processing ended: Wed Apr 23 17:29:14 2008
   Info: Elapsed time: 00:00:10
   Info: Total CPU time (on all processors): 00:00:03
Info: Finished Exporting HDL to D:\DSPBuilderExamples\hdl
Info: Add the QIP_FILE 'singen.qip' to import the files into your project.
```

You can also export HDL by executing the `alt_dspbuilder_exportHDL` command in the MATLAB command window.

The syntax for the export HDL command is:

<*exportDir_value*> `alt_dspbuilder_exportHDL(`<*model*>`,` <*exportDir*>`)`

where:

- *model* is the name of the **.mdl** file being exported. This is always the top-level name in the exported Quartus II project.
- *exportDir* is the directory that contains the exported files. (If this optional argument is omitted, the default or previously used export directory is used.)
- *exportedDir_value* is the return string indicating the output directory containing the newly generated files.

Running this flow creates a set of source files in the export directory, including a **.qip** file corresponding to the top-level of the design.

Using Exported HDL

Once the Export HDL flow is complete, you can create a project using the **New Project Wizard** available on the File menu in the Quartus II software. You should enter the top-level name of the exported project and add the corresponding **.qip** file as the single source file for the project. There may be a number of **.qip** files describing the requirements for black box components. These are sourced automatically by the top-level file.

When migrating designs that include MegaCore function blocks to a different location, it may be necessary to edit their corresponding **.qip** files if they include absolute paths to library components.

This project can be archived as required using the **Archive Project** command in the Quartus II software.

☞ You can migrate the files generated using the Export HDL flow on a Windows computer to a Linux-based computer. However, this requires adding an additional file to the project. This additional **alt_dspbuilder_package.vhd** file is located in the **<QUARTUS_ROOTDIR>\libraries\vhdl\altera** directory on a Windows computer which has DSP Builder installed.

4. Using MegaCore Functions

Introduction

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

The OpenCore Plus evaluation feature allows you to download and evaluate 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, and Viterbi Compiler. You must parameterize and generate these MegaCore functions after you add one of these blocks to your model. See "MegaCore Function Walkthrough" on page 4–8 for an example of the design flow using these MegaCore functions.
- The Video and Image Processing Suite library contains Alpha Blending Mixer, Chroma Resampler, Clipper, Color Plane Sequencer, CSC (Color Space Converter), Deinterlacer, 2D FIR Filter, 2D Median Filter, Frame Buffer, Gamma Corrector, Line Buffer Compiler, Scaler, and Test Pattern Generator. You must parameterize these MegaCore functions but you do not need to generate them before you can connect the blocks to your design. All the required VHDL and simulation files are created only when they are required by your design flow.

☞ The use of Video and Image Processing Suite MegaCore functions is deprecated in DSP Builder version 8.1 and this library will not be supported in future releases. However, the Video and Image Processing Suite MegaCore functions continue to be supported in SOPC Builder.

Installing MegaCore Functions

Altera DSP MegaCore functions are installed with the Quartus® II software. Refer to the MegaCore function user guides for information about each MegaCore function.

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

To run this setup command, follow these steps:

1. Start the MATLAB software. (If MATLAB is already running, check that the Simulink library browser is closed.)

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

   `alt_dspbuilder_setup_megacore` ↵

☞ The process of building the MegaCore function blocks can take several minutes. Do not close MATLAB before the process has completed. Any messages of the form "`Cannot find the declaration of element 'entity'`."are expected when installing a new MegaCore library and can be ignored.

Running this command, 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 of the following 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. Use these blocks in all new designs.
- Dimmed blocks which represent older versions of the MegaCore functions. These blocks are provided 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 describes the function variation is available but the simulation database (**.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` command.

This command recreates 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 command to execute without errors.

To run the command, 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 command 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 MegaCore function 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.) Altera recommends that you rename all blocks representing MegaCore functions with a name describing their use in your design. Using unique block names ensures that all the generated entities for the same MegaCore function in a hierarchical design also 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 selecting **fixed-step** 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 3–16.

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 that is 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, the design variation is described in a **my_function.vhd** file in the design directory.

- If the design is named `my_design`, the simulation information is contained in a file **DSPBuilder_my_design_import/my_function.vo.simdb**.

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

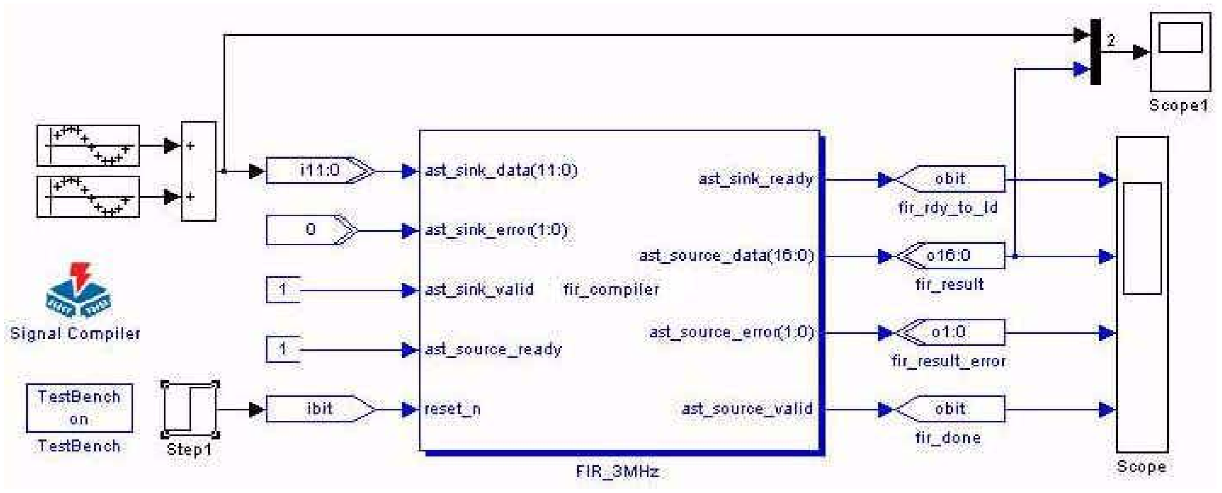
Simulating MegaCore Functions That Have a Reset Port

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 must simulate an initial reset cycle (with the step input) to replicate hardware behavior. As in hardware, this reset cycle must 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



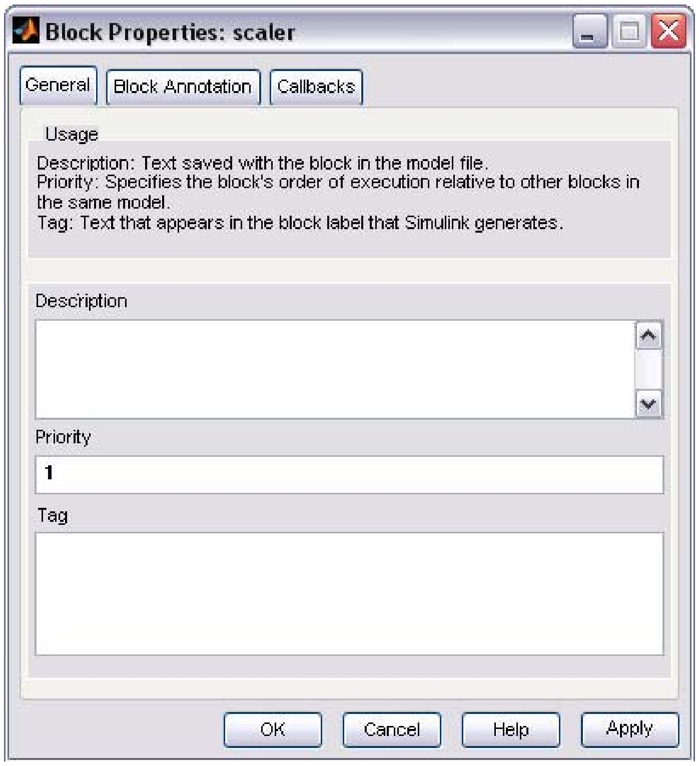
Using Feedback Between MegaCore Functions

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 setting ensures 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 right button pop-up menu and setting a **Priority** value as shown in Figure 4–2.

Figure 4–2. Setting the Priority for a Block


Block Properties: scaler

Usage
Description: Text saved with the block in the model file.
Priority: Specifies the block's order of execution relative to other blocks in the same model.
Tag: Text that appears in the block label that Simulink generates.

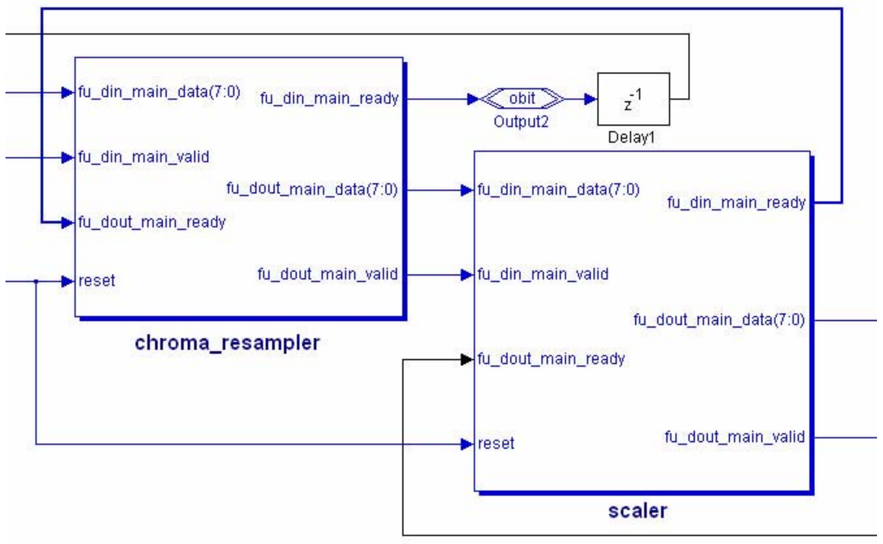
Description

Priority
1

Tag

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 ensures 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) use an improved 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
- Clipper
- Color Plane Sequencer
- Color Space Converter (CSC)
- Deinterlacer
- 2D FIR Filter
- 2D Median Filter
- Frame Buffer
- Gamma Corrector
- Line Buffer Compiler
- Scaler
- Test Pattern Generator

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 4–2 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.

☞ 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. On the File menu, point to **New** and click **Model** to create a new model window.
3. Click **Save** on the 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

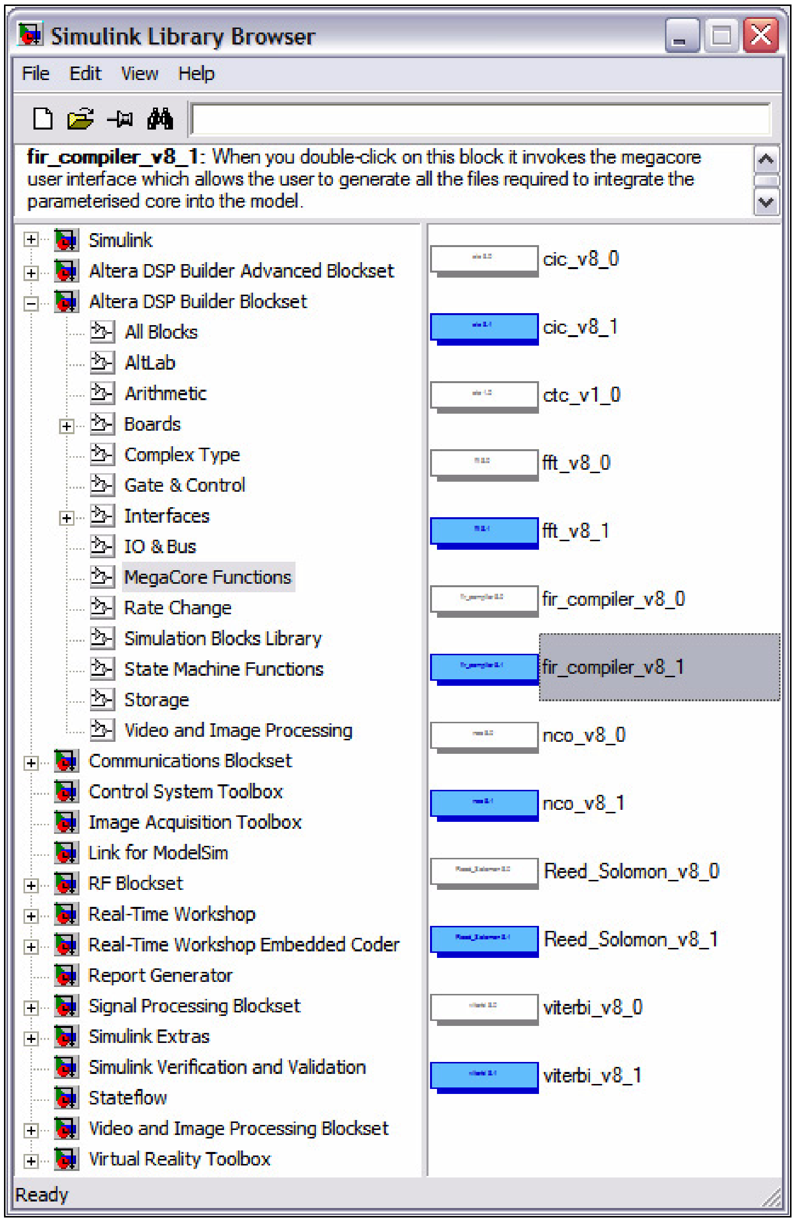
To place a `FIR Compiler` MegaCore function block in your design, perform the following steps:

1. On the View menu In your Simulink model window, click **Library Browser**. 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 4–9).

☞ If the latest versions of the MegaCore function blocks do not appear as shown in Figure 4–4, make sure that the MegaCore IP Library, including the FIR Compiler MegaCore function is installed correctly. The older versions of the MegaCore function blocks (shown dimmed in the Simulink library browser) are provided for backwards compatibility and should not be used in a new design.

For instructions on installing the MegaCore IP library for use with DSP Builder, refer to "Installing MegaCore Functions" on page 4–1.

Figure 4–4. MegaCore Functions Library



Simulink Library Browser

File Edit View Help

fir_compiler_v8_1: When you double-click on this block it invokes the megacore user interface which allows the user to generate all the files required to integrate the parameterised core into the model.

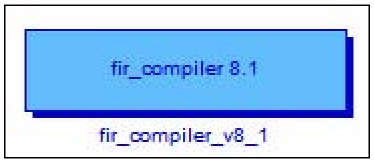
- Simulink
- Altera DSP Builder Advanced Blockset
- Altera DSP Builder Blockset
  - All Blocks
  - AltLab
  - Arithmetic
  - Boards
  - Complex Type
  - Gate & Control
  - Interfaces
  - IO & Bus
  - MegaCore Functions
  - Rate Change
  - Simulation Blocks Library
  - State Machine Functions
  - Storage
  - Video and Image Processing
- Communications Blockset
- Control System Toolbox
- Image Acquisition Toolbox
- Link for ModelSim
- RF Blockset
- Real-Time Workshop
- Real-Time Workshop Embedded Coder
- Report Generator
- Signal Processing Blockset
- Simulink Extras
- Simulink Verification and Validation
- Stateflow
- Video and Image Processing Blockset
- Virtual Reality Toolbox

cic_v8_0
cic_v8_1
ctc_v1_0
fft_v8_0
fft_v8_1
fir_compiler_v8_0
fir_compiler_v8_1
nco_v8_0
nco_v8_1
Reed_Solomon_v8_0
Reed_Solomon_v8_1
viterbi_v8_0
viterbi_v8_1

Ready

3. Drag and drop a blue versioned `fir_compiler_v8.1` block into your model as shown in Figure 4–5.

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



The block is added with a default instance name which includes the version string. This name is 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.

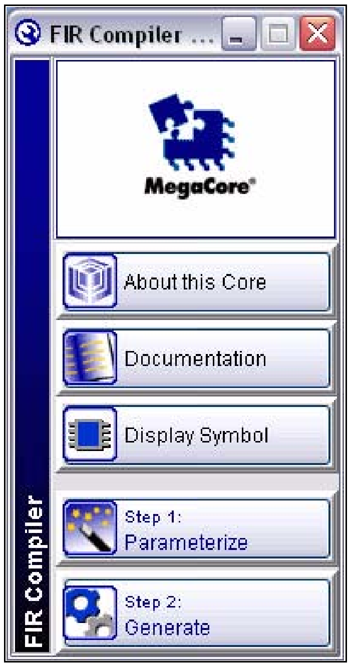
Always give blocks representing your MegaCore function variations unique names, to avoid issues caused by two or more entities in a hierarchical design.

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

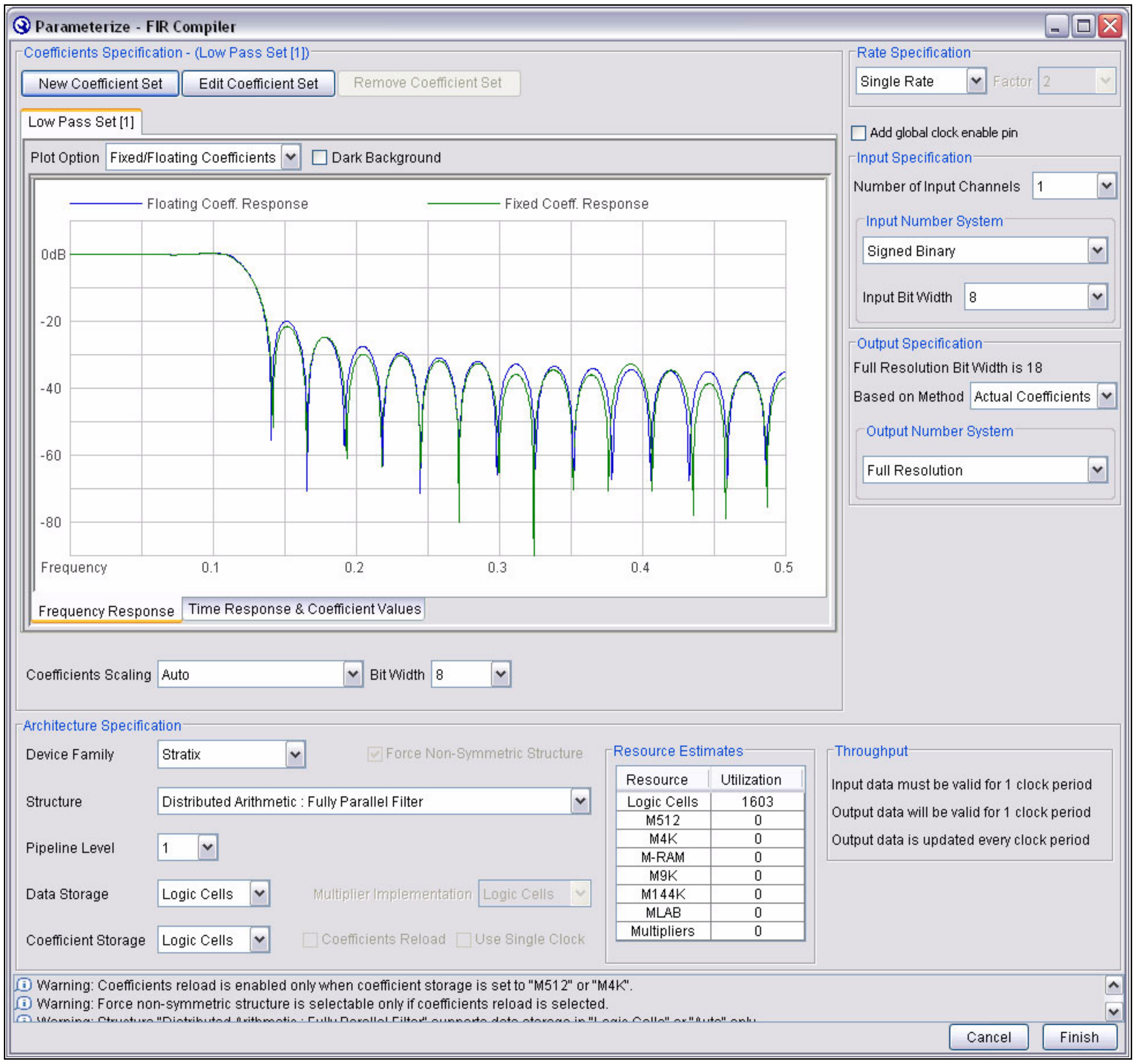


2. Click **Step 1: Parameterize** to specify how the FIR filter should operate.

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

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



Plot Option: Fixed/Floating Coefficients
Dark Background: unchecked

Coefficients Scaling: Auto
Bit Width: 8

Rate Specification: Single Rate; Factor: 2
Add global clock enable pin: unchecked

Input Specification
Number of Input Channels: 1
Input Number System: Signed Binary
Input Bit Width: 8

Output Specification
Full Resolution Bit Width is 18
Based on Method: Actual Coefficients
Output Number System: Full Resolution

Architecture Specification
Device Family: Stratix
Force Non-Symmetric Structure
Structure: Distributed Arithmetic : Fully Parallel Filter
Pipeline Level: 1
Data Storage: Logic Cells
Multiplier Implementation: Logic Cells
Coefficient Storage: Logic Cells
Coefficients Reload
Use Single Clock

Resource Estimates

Resource	Utilization
Logic Cells	1603
M512	0
M4K	0
M-RAM	0
M9K	0
M144K	0
MLAB	0
Multipliers	0

Throughput
Input data must be valid for 1 clock period
Output data will be valid for 1 clock period
Output data is updated every clock period

Warning: Coefficients reload is enabled only when coefficient storage is set to "M512" or "M4K".
Warning: Force non-symmetric structure is selectable only if coefficients reload is selected.
Warning: Structure "Distributed Arithmetic : Fully Parallel Filter" supports data storage in "Logic Cells" or "Auto" only

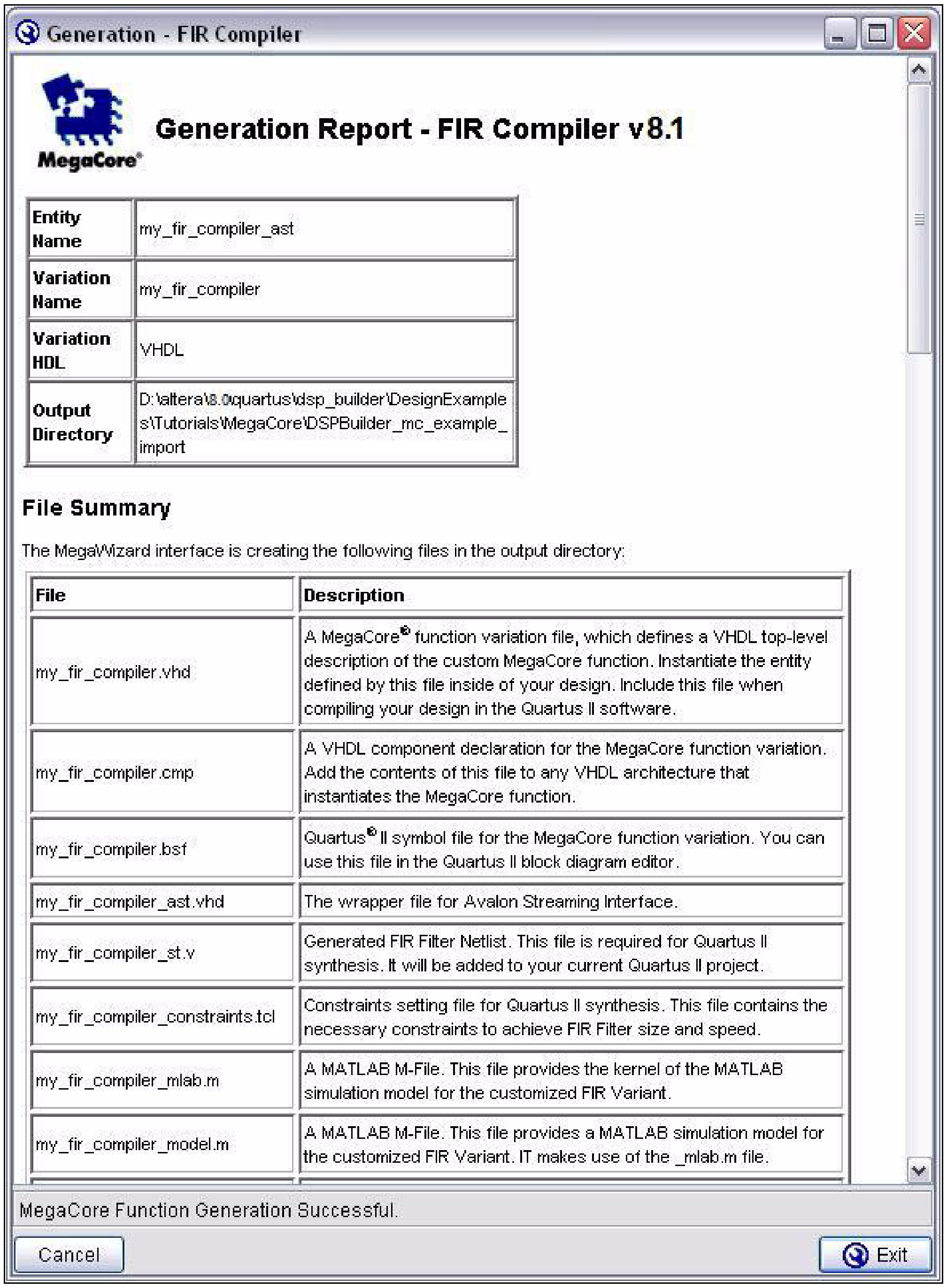
Generate the FIR Compiler Function Variation

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 4–10).

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

Figure 4–8. Generation Report


Generation - FIR Compiler

Generation Report - FIR Compiler v8.1

Entity Name	my_fir_compiler_ast
Variation Name	my_fir_compiler
Variation HDL	VHDL
Output Directory	D:\altera\8.0\quartus\dsp_builder\DesignExamples\Tutorials\MegaCore\DSPBuilder_mc_example_import

File Summary

The MegaWizard interface is creating the following files in the output directory:

File	Description
my_fir_compiler.vhd	A MegaCore® function variation file, which defines a VHDL top-level description of the custom MegaCore function. Instantiate the entity defined by this file inside of your design. Include this file when compiling your design in the Quartus II software.
my_fir_compiler.cmp	A VHDL component declaration for the MegaCore function variation. Add the contents of this file to any VHDL architecture that instantiates the MegaCore function.
my_fir_compiler.bsf	Quartus® II symbol file for the MegaCore function variation. You can use this file in the Quartus II block diagram editor.
my_fir_compiler_ast.vhd	The wrapper file for Avalon Streaming Interface.
my_fir_compiler_st.v	Generated FIR Filter Netlist. This file is required for Quartus II synthesis. It will be added to your current Quartus II project.
my_fir_compiler_constraints.tcl	Constraints setting file for Quartus II synthesis. This file contains the necessary constraints to achieve FIR Filter size and speed.
my_fir_compiler_mlab.m	A MATLAB M-File. This file provides the kernel of the MATLAB simulation model for the customized FIR Variant.
my_fir_compiler_model.m	A MATLAB M-File. This file provides a MATLAB simulation model for the customized FIR Variant. IT makes use of the _mlab.m file.

MegaCore Function Generation Successful.

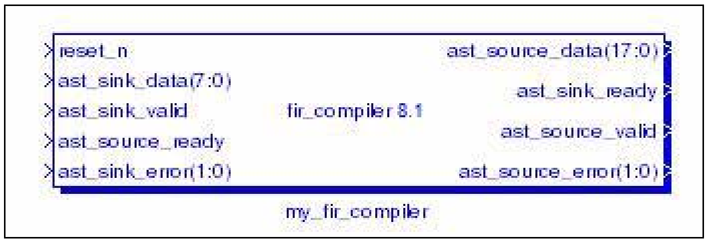
3. Click **Exit**.

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

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 4–15.

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).

   Notice that the second block is automatically given a unique name.

2. Use the **Block Parameters** dialog box to set the parameters for the `Sine Wave` block shown in Table 4–1.
3. Repeat Step 2 for the `Sine Wave1` block.

Table 4–1. Parameters for the Sine Wave Blocks

Parameter	Value	
	`Sine Wave`	`Sine Wave1`
Sine type	Sample based	Sample based
Time	Use simulation time	Use simulation time
Amplitude	64	64
Bias	0	0
Samples per period	200	7
Number of offset examples	0	0
Sample time	1	1
Interpret vector parameters as 1-D	On	On

4. Connect the outputs from the `Sine Wave` and `Sine Wave1` blocks to an `Add` block (from the Simulink **Math Operations** library).

5. 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.
6. Use the **Block Parameters** dialog box to set the parameters shown in Table 4–2.

Table 4–2. Parameters for the Input Block

Parameter	**Value**
Bus Type	Signed Integer
[number of bits].[]	8
Specify Clock	Off

7. 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.
8. 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.
9. Use the **Block Parameters** dialog box to set the parameters for the `Constant` block shown in Table 4–3.
10. Repeat Step 9 for the `Constant1` block.

Table 4–3. Parameters for the Constant Blocks

	Value	
Parameter	`Constant`	`Constant1`
Constant Value	1	0
Bus Type	Single Bit	Signed Integer
[Number of Bits].[]	–	2
Rounding Mode	Truncate	Truncate
Saturation Mode	Wrap	Wrap
Specify Clock	Off	Off

11. Add a `Single Pulse` block (from the **Gate & Control** library in the **Altera DSP Builder Blockset**) and connect it to the `reset_n` pin on the `my_fir_compiler` block.
12. Use the **Block Parameters** dialog box to set the parameters shown in Table 4–4.

Table 4–4. Parameters for the Single Pulse Block

Parameter	**Value**
Signal Generation Type	Step Up
Delay	50
Specify Clock	Off

13. 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.

14. Use the **Block Parameters** dialog box to set the parameters shown in Table 4–5.

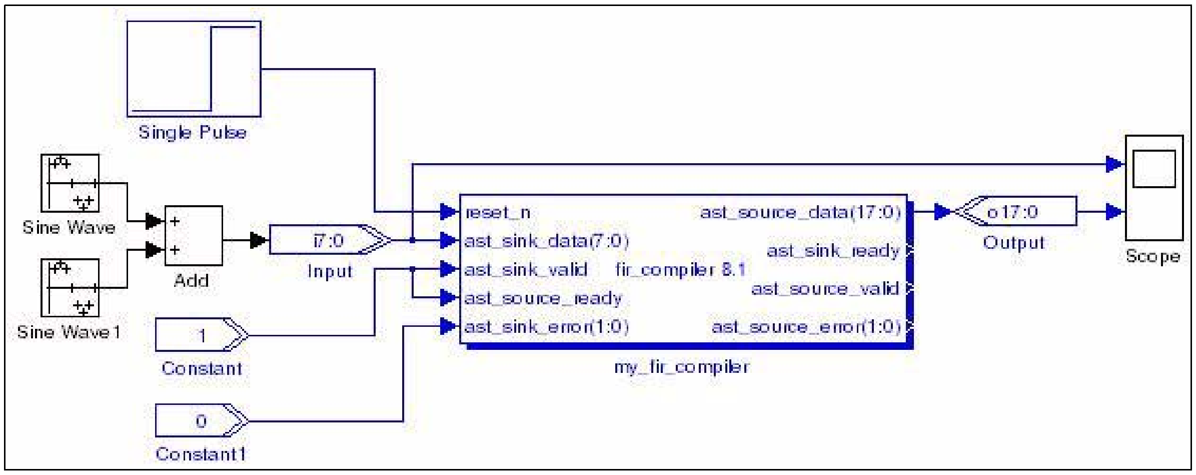
Table 4–5. Parameters for the Output Block

Parameter	**Value**
Bus Type	Signed Integer
[number of bits].[]	18
External Type	Inferred

15. 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.
16. 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. On the Simulation menu in your model, click **Configuration Parameters** to display the **Configuration Parameters** dialog box (Figure 4–11 on page 4–16).
2. Select the **Solver** page and set the parameters shown in Table 4–6.

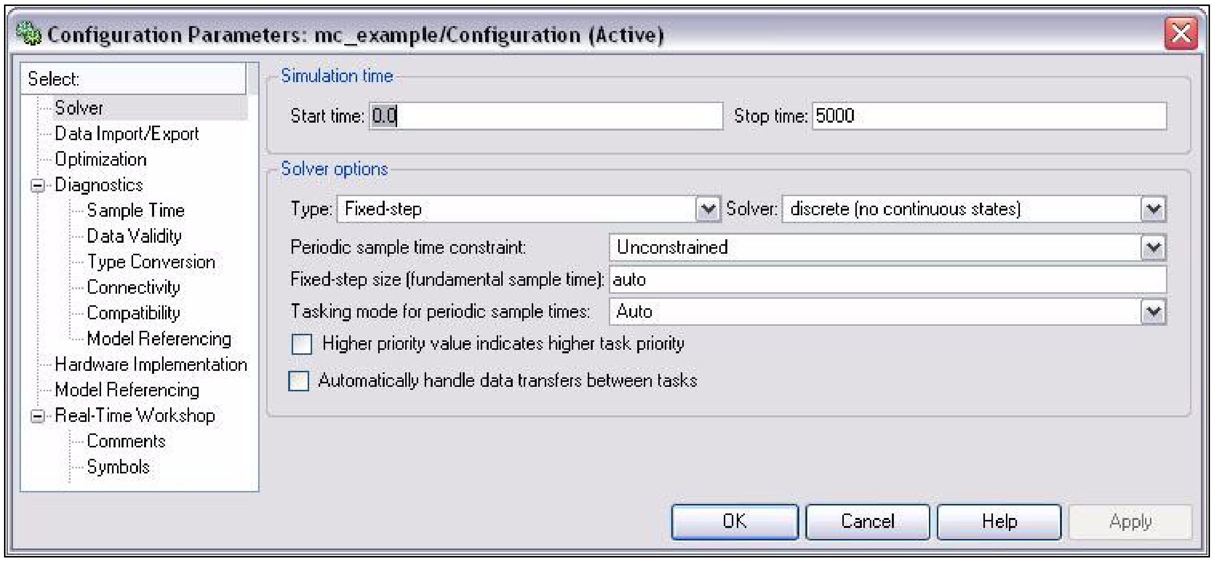
Table 4–6. Configuration Parameters for the singen Model

Parameter	**Value**
Start time	0.0
Stop time	5000
Type	Fixed-step
Solver	discrete (no continuous states)

Refer to the description of the "Solver Pane" in the Simulink Help for detailed information about solver options.

3. Click **OK**.

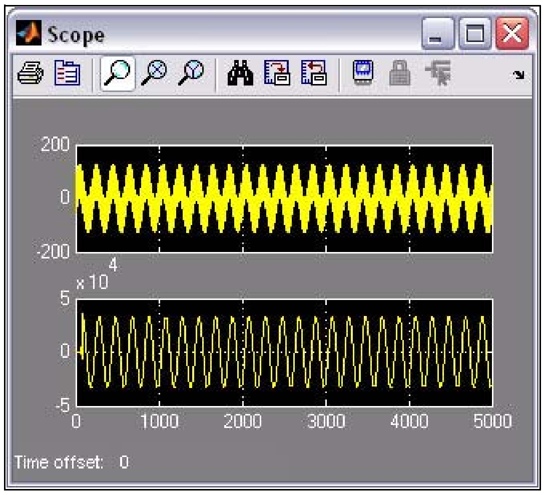
Figure 4–11. Configuration Parameters: mc_example/Configuration Dialog Box



Start time: 0.0
Stop time: 5000
Type: Fixed-step
Solver: discrete (no continuous states)
Periodic sample time constraint: Unconstrained
Fixed-step size (fundamental sample time): auto
Tasking mode for periodic sample times: Auto
☐ Higher priority value indicates higher task priority
☐ Automatically handle data transfers between tasks

4. On the Simulation menu in the simulink model, click **Start**. The scope output shows the effect of the low-pass filter in the bottom window, as shown in Figure 4–12.

Figure 4–12. Simulation Output



Check that the FIR filter block behaves as expected, filtering high-frequency data as a low-pass filter.

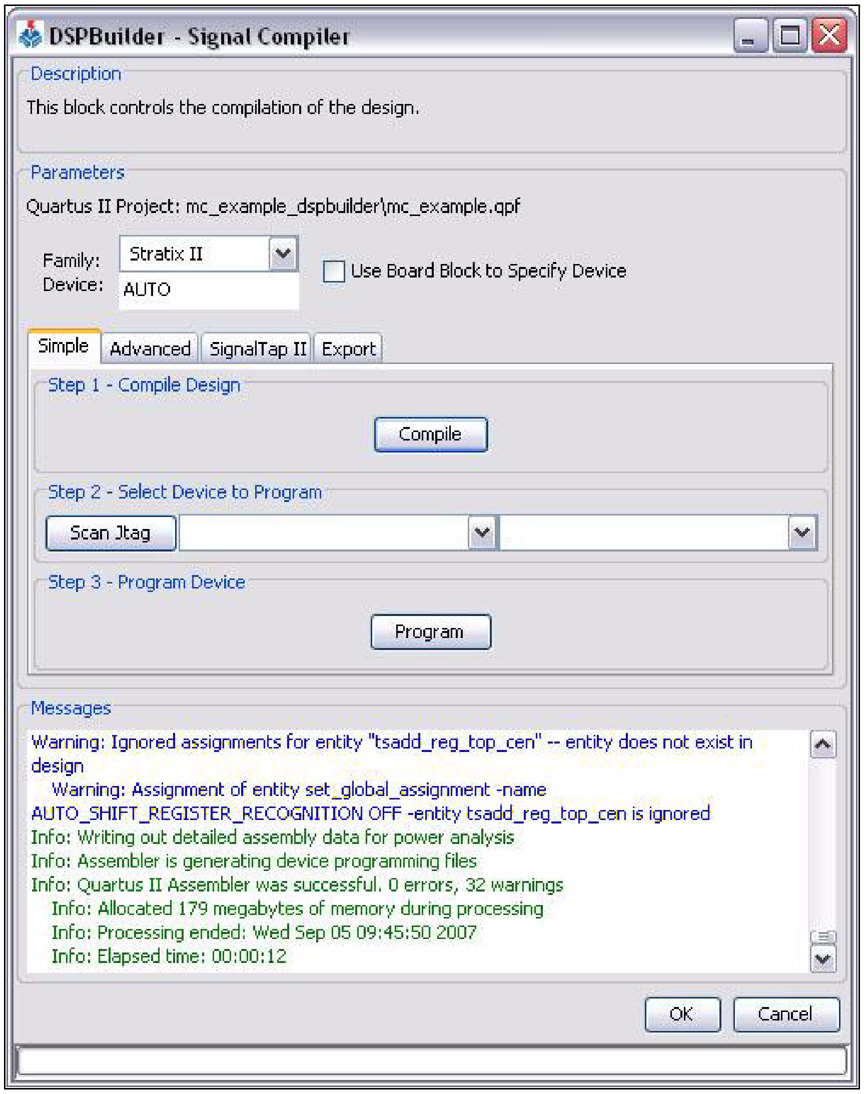
☞ You may need to use the **Autoscale** command in the Scope display to view the complete waveforms.

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).

Figure 4–13. Signal Compiler Dialog Box



This block controls the compilation of the design.

Quartus II Project: mc_example_dspbuilder\mc_example.qpf

Family: Stratix II
Device: AUTO
☐ Use Board Block to Specify Device

```
Warning: Ignored assignments for entity "tsadd_reg_top_cen" -- entity does not exist in
design
   Warning: Assignment of entity set_global_assignment -name
AUTO_SHIFT_REGISTER_RECOGNITION OFF -entity tsadd_reg_top_cen is ignored
Info: Writing out detailed assembly data for power analysis
Info: Assembler is generating device programming files
Info: Quartus II Assembler was successful. 0 errors, 32 warnings
   Info: Allocated 179 megabytes of memory during processing
   Info: Processing ended: Wed Sep 05 09:45:50 2007
   Info: Elapsed time: 00:00:12
```

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

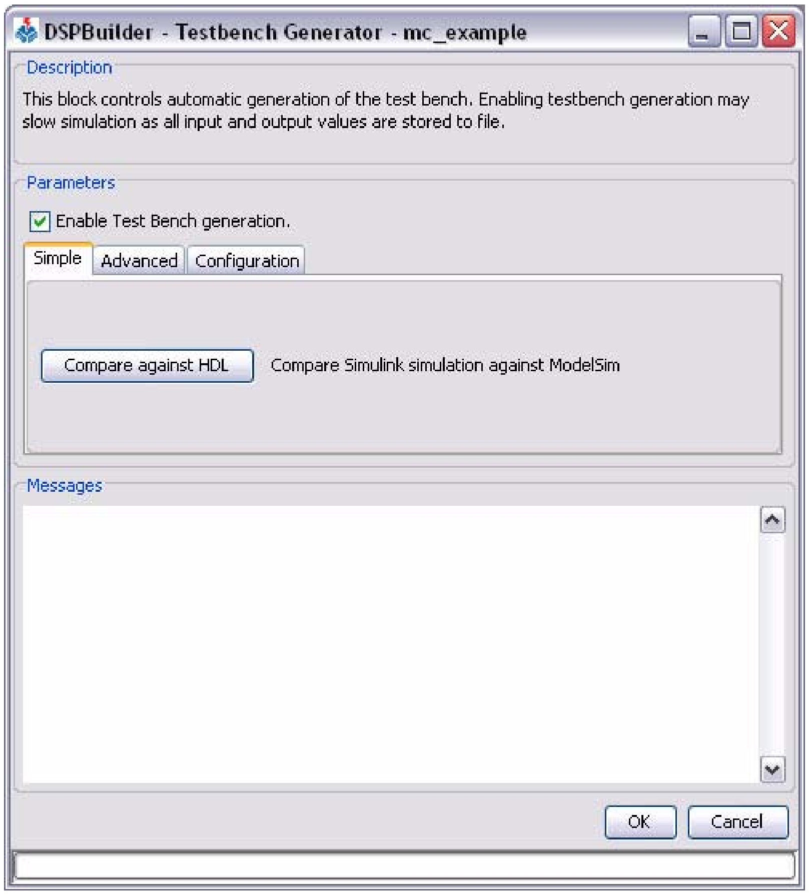
Perform RTL Simulation

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



DSPBuilder - Testbench Generator - mc_example

Description

This block controls automatic generation of the test bench. Enabling testbench generation may slow simulation as all input and output values are stored to file.

Parameters

[✓] Enable Test Bench generation.

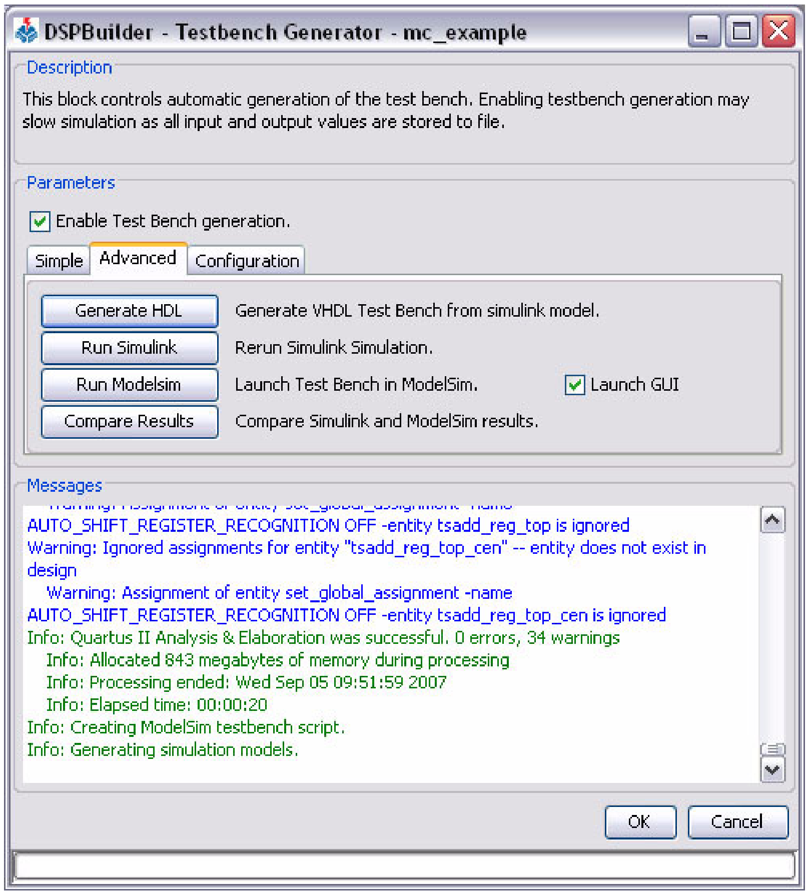
Compare Simulink simulation against ModelSim

Messages

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

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

Figure 4–15. TestBench Generator Dialog Box Advanced Tab



This block controls automatic generation of the test bench. Enabling testbench generation may slow simulation as all input and output values are stored to file.

✓ Enable Test Bench generation.

Generate VHDL Test Bench from simulink model.
Rerun Simulink Simulation.
Launch Test Bench in ModelSim. ✓ Launch GUI
Compare Simulink and ModelSim results.

```
AUTO_SHIFT_REGISTER_RECOGNITION OFF -entity tsadd_reg_top is ignored
Warning: Ignored assignments for entity "tsadd_reg_top_cen" -- entity does not exist in
design
   Warning: Assignment of entity set_global_assignment -name
AUTO_SHIFT_REGISTER_RECOGNITION OFF -entity tsadd_reg_top_cen is ignored
Info: Quartus II Analysis & Elaboration was successful. 0 errors, 34 warnings
   Info: Allocated 843 megabytes of memory during processing
   Info: Processing ended: Wed Sep 05 09:51:59 2007
   Info: Elapsed time: 00:00:20
Info: Creating ModelSim testbench script.
Info: Generating simulation models.
```

6. Turn on the **Launch GUI** option. This launches the ModelSim GUI if 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 simulate the design in ModelSim.

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

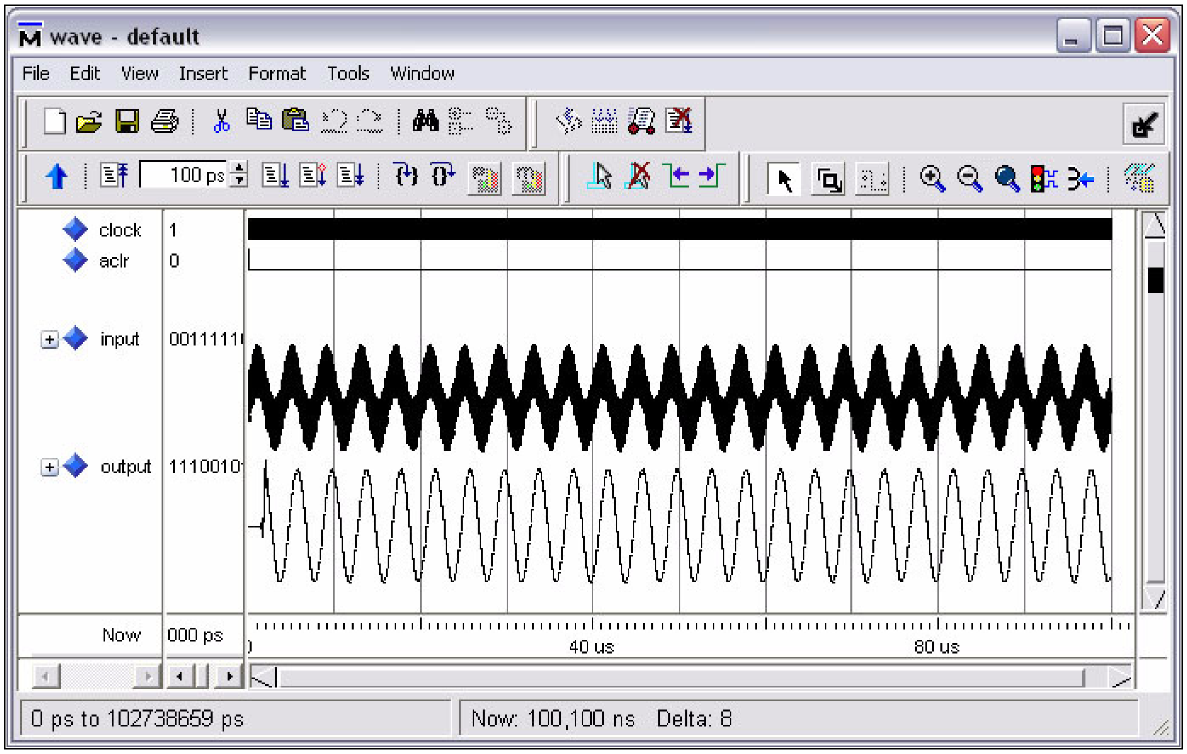
   ☞ All waveforms are initially shown using digital format in the ModelSim Wave window.

10. Right-click the `input` signal in the ModelSim Wave window and click **Properties** in the pop-up 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. Repeat Step 10 for 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. Click **Zoom Full** on the right button right button pop-up 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



0 ps to 102738659 ps
Now: 100,100 ns Delta: 8

Note to Figure 4–16:

(1) The waveform display shown here has been formatted to show the input and output signals as analog waveforms.

1. Click **Compare Results** in the **Testbench Generator** dialog box to compare the simulink results with those generated by ModelSim. The message `Exact Match` should be issued indicating that the results are identical.
2. Click **OK** to close the **Testbench Generator** 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 runs 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.

The HIL design flow comprises the following steps:

1. Create a Quartus II project that defines the functions you want to co-simulate in hardware and use `Signal Compiler` block to compile the Quartus II project through the Quartus II Fitter.
2. Add the `HIL` block to your Simulink model and import the compiled Quartus II project into the `HIL` block. You can also connect instrumentation to your `HIL` block by adding additional blocks from the Simulink Sinks and Sources libraries.

   If the original design contains a `Clock` block that defines a period and sample time that is different from the default values, you must add a `Clock` block with the same values to the HIL model.

3. Specify parameters for the `HIL` block, including the following options:
   - The Quartus II project you compiled to define its functionality
   - The clock and reset pins
   - The reset active level
   - The input and output pin characteristics
   - The use of single-step versus burst and frame mode

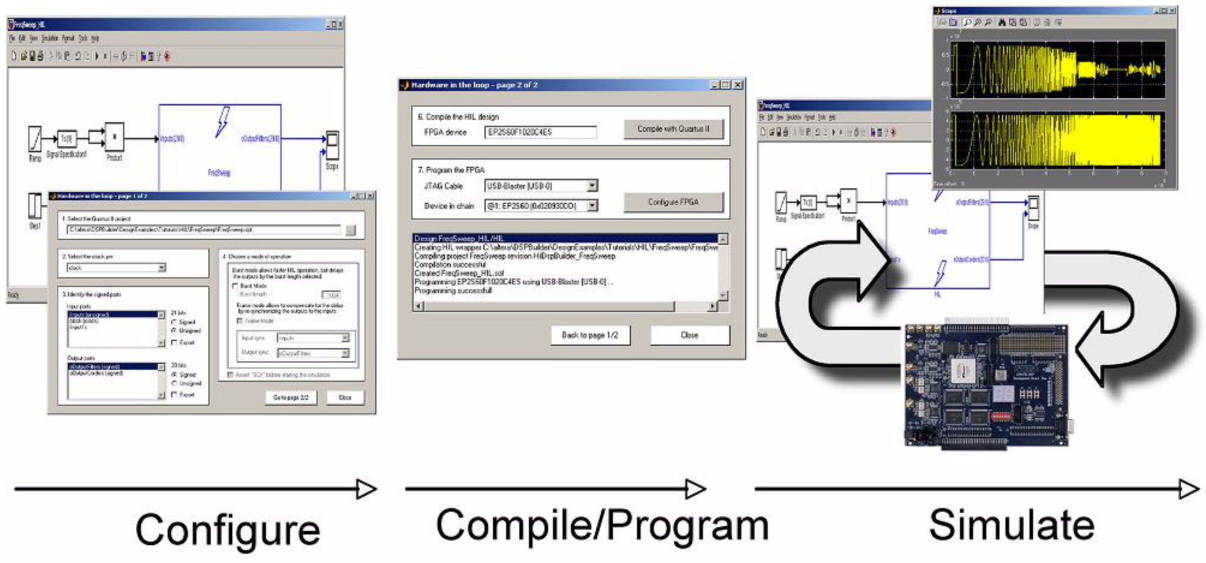
4. Compile the `HIL` block to create a programming object file that can be used for hardware co-simulation.
5. Scan for JTAG cables and hardware devices connected to the local host or any remotely enabled hosts.
6. Program the board that contains your target FPGA.
7. Simulate the combined software and hardware system in Simulink.

When using a `HIL` block in a Simulink model, set a fixed-step, single tasking solver.

Refer to "Design Flow" on page 1–3 for information about setting the Simulink solver.

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

Figure 5–1. System-Level Design Flow



6. Compile the HIL design
FPGA device: EP2S60F1020C4ES

7. Program the FPGA
JTAG Cable: USB-Blaster [USB-0]
Device in chain: @1: EP2S60 (0x020930DD)

```
Design FreqSweep_HIL/HIL
Creating HIL wrapper C:\altera\DSPBuilder\DesignExamples\Tutorials\HIL\FreqSweep\FreqSwe
Compiling project FreqSweep revision HilDspBuilder_FreqSweep
Compilation successful
Created FreqSweep_HIL.sof
Programming EP2S60F1020C4ES using USB-Blaster [USB-0] ...
Programming successfull
```

HIL Requirements

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

- An FPGA board with a JTAG interface (Stratix III, Stratix II, Stratix, Cyclone III, 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 the following 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.

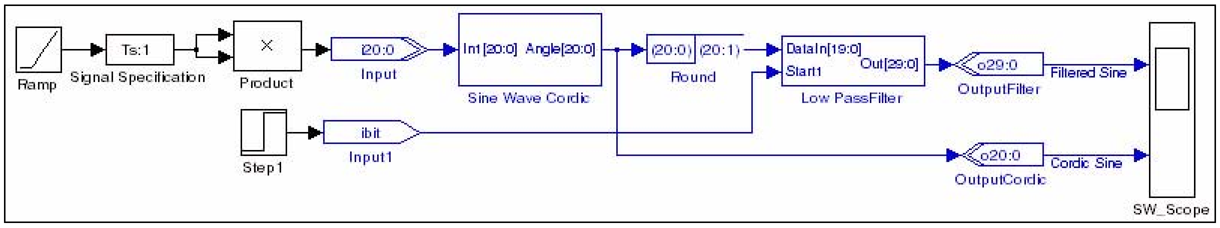
You also need an FPGA board connected to your computer with a JTAG download cable.

This walkthrough uses a Quartus II project which is created using DSP Builder and the Stratix II hardware device on an 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.

Perform the following steps:

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

Figure 5–2. Frequency Sweep Model

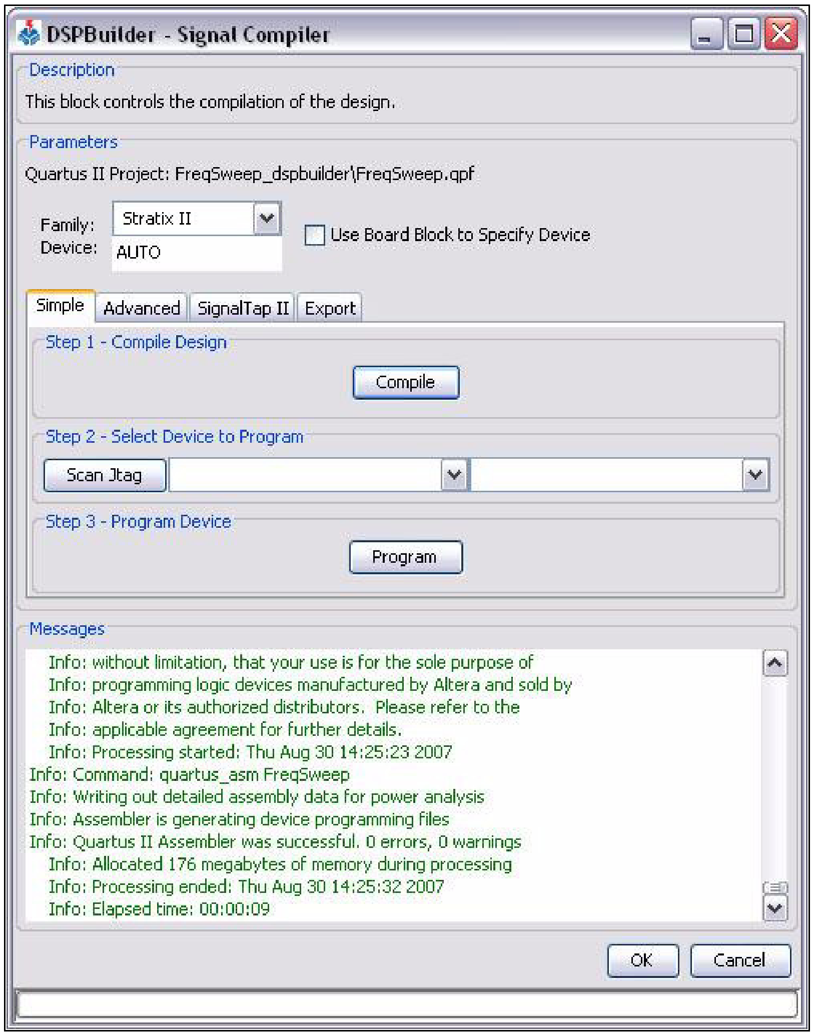


2. Double-click the `Signal Compiler` block. In the dialog box that appears (Figure 5–3 on page 5–4), click **Compile**.

   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.

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



This block controls the compilation of the design.

Quartus II Project: FreqSweep_dspbuilder\FreqSweep.qpf

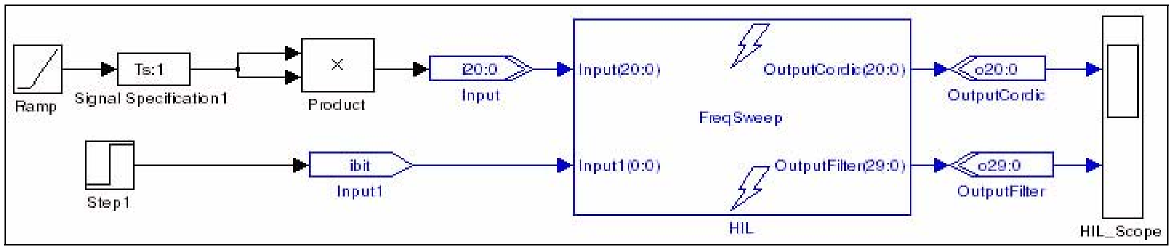
Family: Stratix II
Device: AUTO
Use Board Block to Specify Device

```
   Info: without limitation, that your use is for the sole purpose of
   Info: programming logic devices manufactured by Altera and sold by
   Info: Altera or its authorized distributors.  Please refer to the
   Info: applicable agreement for further details.
   Info: Processing started: Thu Aug 30 14:25:23 2007
Info: Command: quartus_asm FreqSweep
Info: Writing out detailed assembly data for power analysis
Info: Assembler is generating device programming files
Info: Quartus II Assembler was successful. 0 errors, 0 warnings
   Info: Allocated 176 megabytes of memory during processing
   Info: Processing ended: Thu Aug 30 14:25:32 2007
   Info: Elapsed time: 00:00:09
```

3. Review the Messages, then click **OK** to close the **Signal Compiler** dialog box.
4. Replace the internal functions of the Frequency Sweep model with an `HIL` block. For this walkthrough, you can achieve this by opening the prepared model **FreqSweep_HIL.mdl** from the **FreqSweep** directory specified in Step 1.

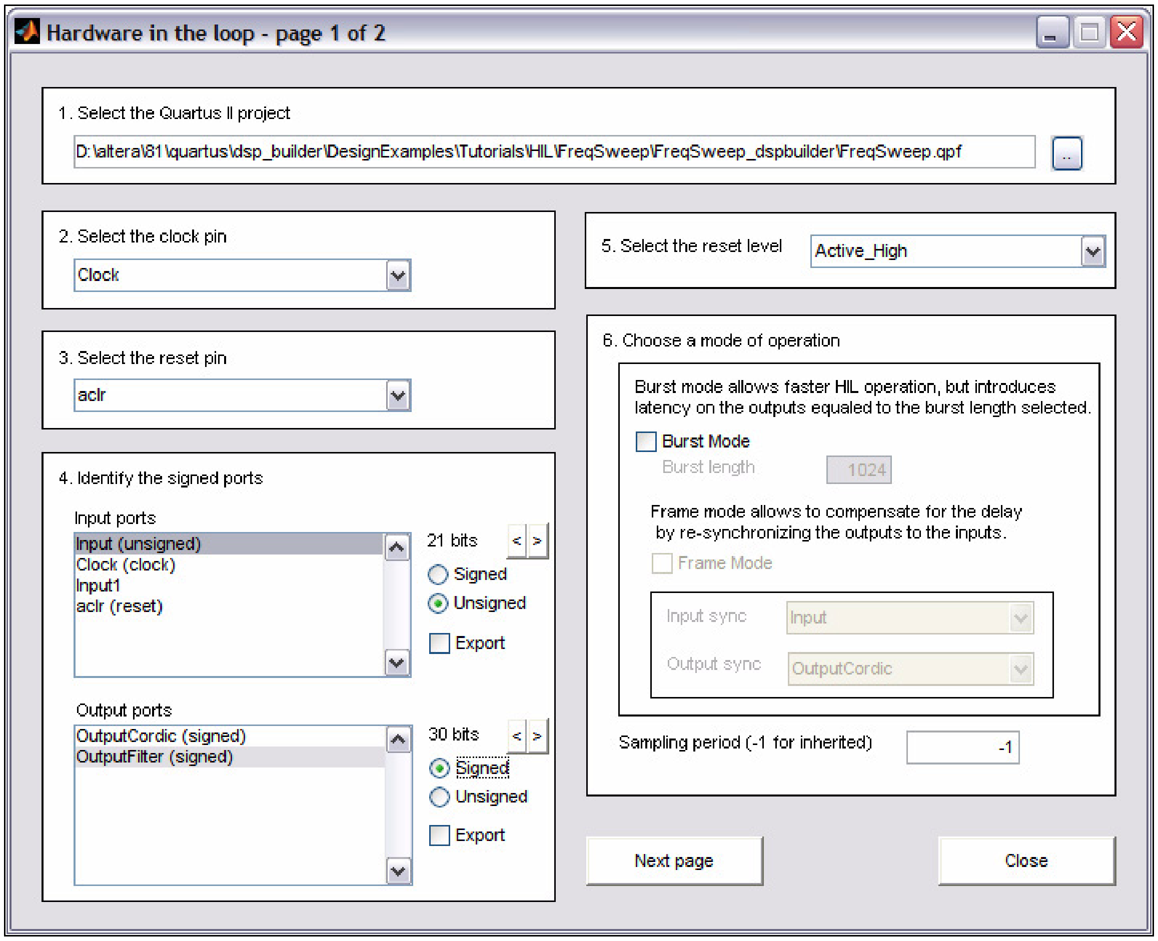
   Figure 5–4 shows this model, with the `HIL` block in place.

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



Hardware in the loop - page 1 of 2

1. Select the Quartus II project
D:\altera\81\quartus\dsp_builder\DesignExamples\Tutorials\HIL\FreqSweep\FreqSweep_dspbuilder\FreqSweep.qpf

2. Select the clock pin: Clock

3. Select the reset pin: aclr

4. Identify the signed ports

Input ports
Input (unsigned)
Clock (clock)
Input1
aclr (reset)

21 bits
Signed [✓ Unsigned]
Export

Output ports
OutputCordic (signed)
OutputFilter (signed)

30 bits
[✓ Signed] Unsigned
Export

5. Select the reset level: Active_High

6. Choose a mode of operation

Burst mode allows faster HIL operation, but introduces latency on the outputs equaled to the burst length selected.
Burst Mode
Burst length: 1024

Frame mode allows to compensate for the delay by re-synchronizing the outputs to the inputs.
Frame Mode
Input sync: Input
Output sync: OutputCordic

Sampling period (-1 for inherited): -1

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

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

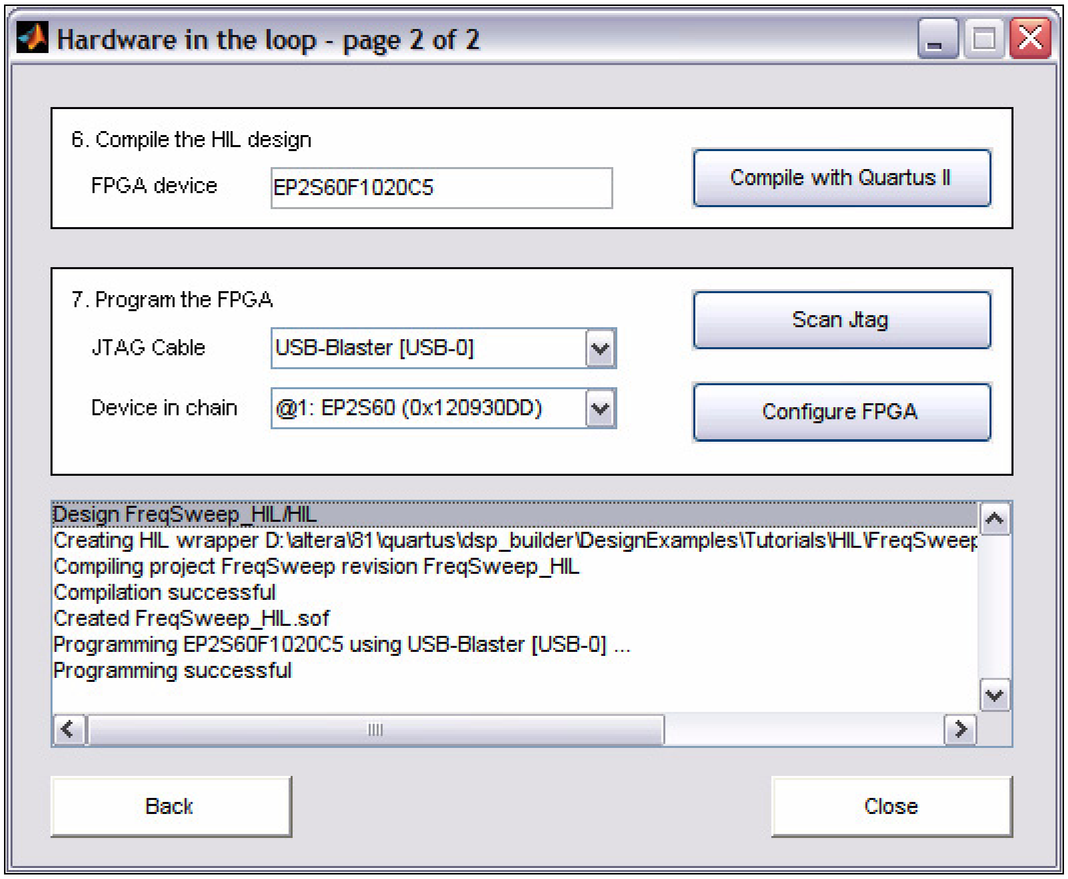
7. Select **Clock** from the list of available clock pins.

   HIL does not support multiple clock domains and only the selected signal can be used as the HIL clock signal. Any other clocks in the design are treated as input signals.

8. Select **aclr** from the list of available reset pins.
9. Identify the signed ports:
   - Select the **input** port and click **Unsigned**.
   - Select each output port (**OutputCordic** and **OutputFilter**) and click **Signed**.

10. Select the reset level to be **Active_High**.
11. Choose the mode of operation by turning off **Burst Mode**.
12. Click **Next page**. to display the second page of the **Hardware in the loop** dialog box (Figure 5–6).

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



FPGA device: EP2S60F1020C5

JTAG Cable: USB-Blaster [USB-0]

Device in chain: @1: EP2S60 (0x120930DD)

```
Design FreqSweep_HIL/HIL
Creating HIL wrapper D:\altera\81\quartus\dsp_builder\DesignExamples\Tutorials\HIL\FreqSweep
Compiling project FreqSweep revision FreqSweep_HIL
Compilation successful
Created FreqSweep_HIL.sof
Programming EP2S60F1020C5 using USB-Blaster [USB-0] ...
Programming successful
```

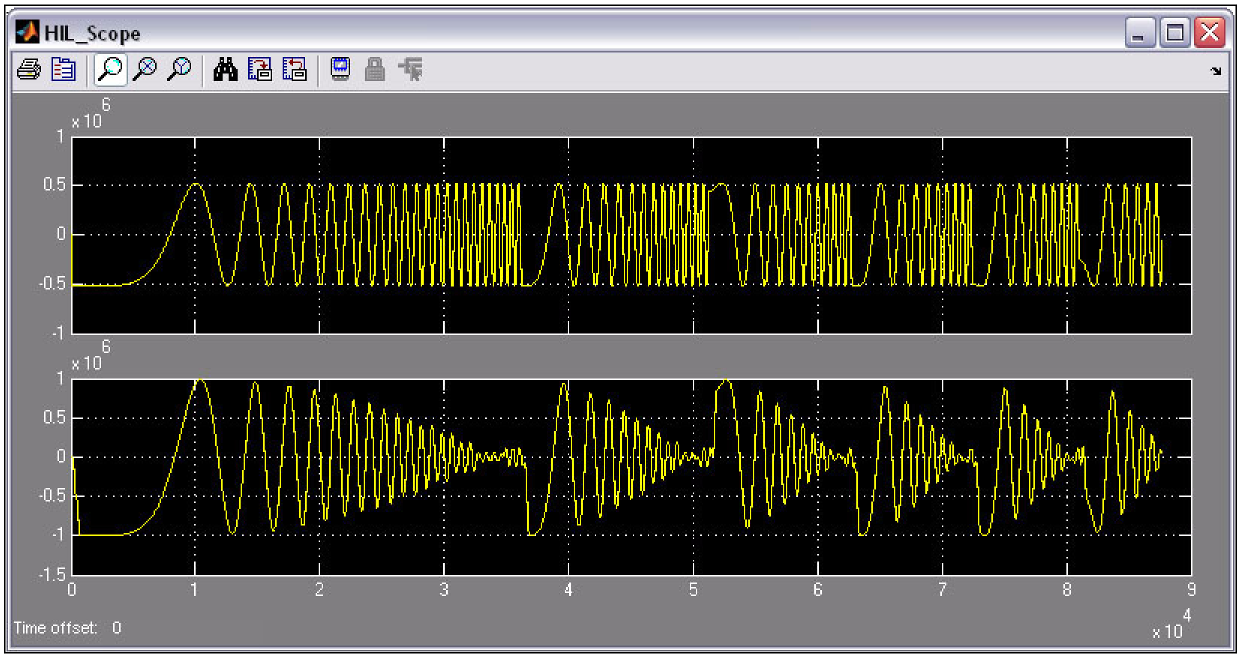
13. Specify a value for the **FPGA device** and click **Compile with Quartus II** to compile the HIL design.

    ☞ If no output is written to the MATLAB command window, check that the original Quartus II project is up-to-date and has been compiled by the same version of the Quartus II software as that used to compile your Simulink model.

14. Click **Scan Jtag** to find available cables and hardware devices in the chain.
15. Select the JTAG download cable that references the required FPGA device and click **Configure FPGA** to program the FPGA on the board.
16. Click **Close**.

17. 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 debugging (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 for certain types of designs.

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

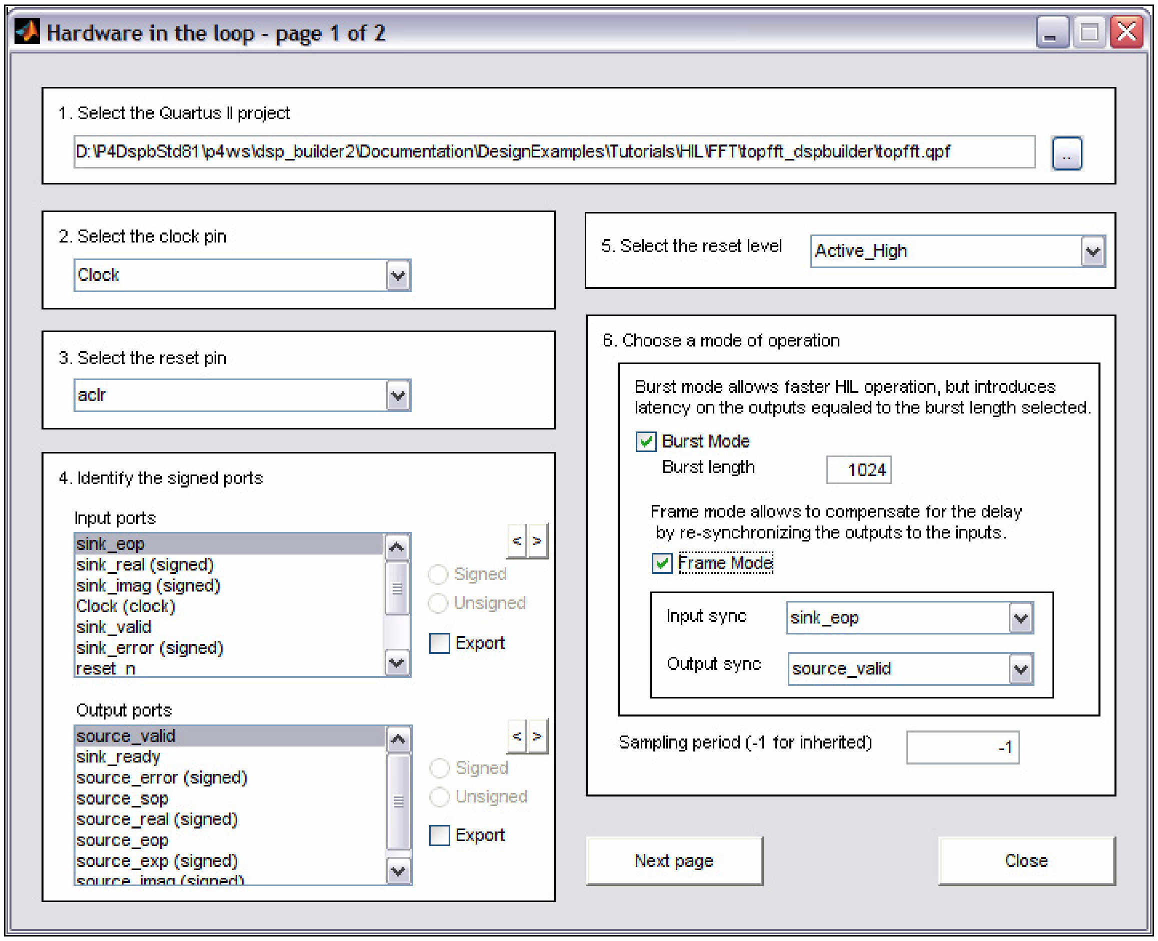
Table 5–1. Comparing Single-Step and Burst Modes

	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** option in the **Hardware in the loop** dialog box as shown in Figure 5–8.

Figure 5–8. Setting Parameters for the HIL Block in Figure 5–10


Hardware in the loop - page 1 of 2

1. Select the Quartus II project
D:\P4DspbStd81\p4ws\dsp_builder2\Documentation\DesignExamples\Tutorials\HIL\FFT\topfft_dspbuilder\topfft.qpf

2. Select the clock pin
Clock

3. Select the reset pin
aclr

4. Identify the signed ports

Input ports
sink_eop
sink_real (signed)
sink_imag (signed)
Clock (clock)
sink_valid
sink_error (signed)
reset_n

Signed
Unsigned
Export

Output ports
source_valid
sink_ready
source_error (signed)
source_sop
source_real (signed)
source_eop
source_exp (signed)
source_imag (signed)

Signed
Unsigned
Export

5. Select the reset level
Active_High

6. Choose a mode of operation

Burst mode allows faster HIL operation, but introduces latency on the outputs equaled to the burst length selected.
[✓] Burst Mode
Burst length 1024

Frame mode allows to compensate for the delay by re-synchronizing the outputs to the inputs.
[✓] Frame Mode

Input sync sink_eop
Output sync source_valid

Sampling period (-1 for inherited) -1

When this option is set, you can specify the required number of data packets as the **Burst length**. The HIL block sends 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** option in the **Hardware in the loop** dialog box as shown in Figure 5–8 on page 5–8. 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 is 1027 (1024 + 3) without frame mode or 2048 (aligned to the next frame) with frame mode on.

The burst packet size in frame mode must 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* × 100 clocks (*N* positive integer).

Figure 5–9 illustrates a DSP Builder design using a FFT MegaCore function which has been configured for the Stratix II target device family, with a transform length of 64 points, data precision of 16 bits, and twiddle precision of 16 bits.

Figure 5–9. DSP Builder Design Using the FFT MegaCore Function

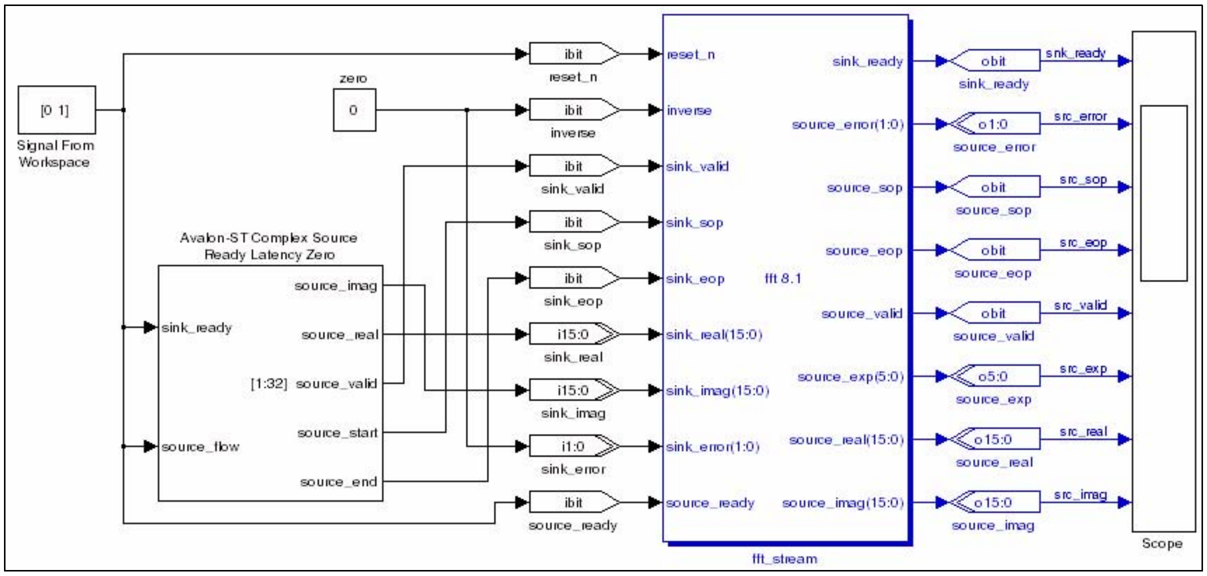
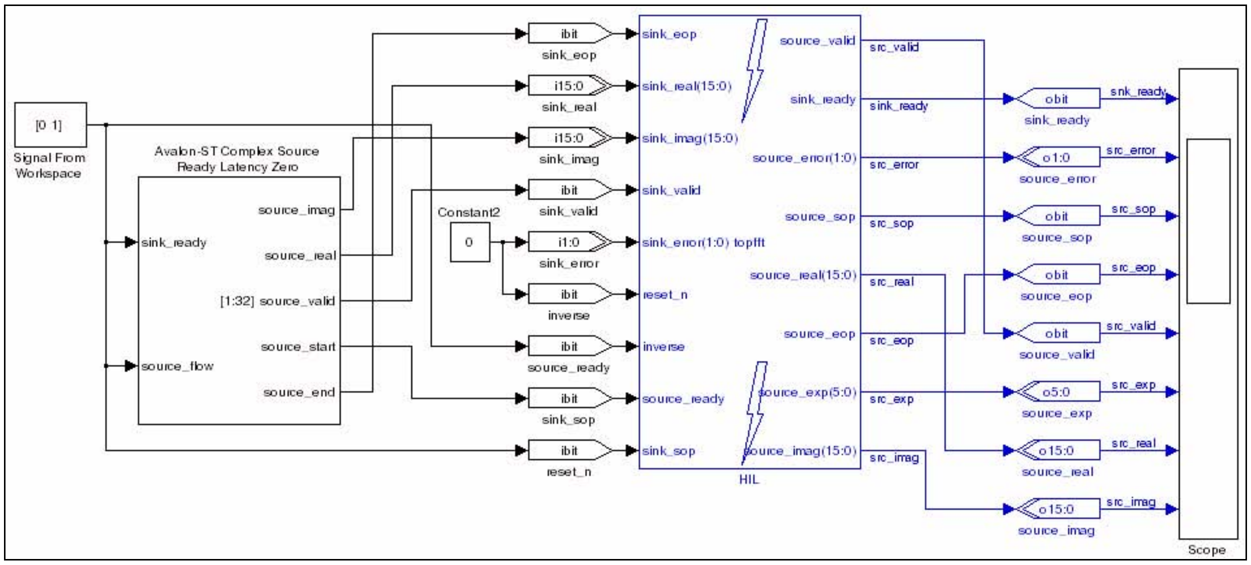


Figure 5–10 on page 5–10 shows the FFT design implemented using a `HIL` block (with the parameters shown in Figure 5–8 on page 5–8).

Figure 5–10. Using the FFT Design With an HIL Block



The Avalon-ST 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.

Troubleshooting HIL Designs

This section describes various issues that you may encounter when you are using HIL designs.

If the top-level of your design has changed, the Quartus II project must be compiled and reloaded into HIL to ensure that all information is up-to-date.

Failed to Load the Specified Quartus II Project

HIL reads 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 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 the 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 project using the registered Quartus II software before running HIL.

Quartus II Project File is Not Up-to-Date

This occurs when the specified Quartus II project file (**.qpf**) 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 Inputs 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.

No Outputs Found From the Quartus II Project

This could occur if the design does not have any outputs and makes the HIL simulation meaningless.

Action:

None required.

HIL Design Stays in Reset During Simulation

An asynchronous reset is permanently asserted for a HIL design.

Action:

Check that the reset active level matches the setting in the original design. Recompile the HIL design after you have changed the reset level.

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 or to Volume 3 of the *Quartus II handbook*.

You can instantiate a `SignalTap II Logic Analyzer` block in DSP Builder with the following characteristics:

- 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

Alternatively, you can use the Quartus II software to instantiate 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 the percentage of data 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 the **Signal Compiler** dialog box.
4. Choose one of the JTAG cable ports in the **Signal Compiler** dialog box or the **SignalTap II Logic Analyzer** dialog box.
5. Using `Signal Compiler`, synthesize your model, perform compilation in the Quartus II software, 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 details of 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) or 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. The SignalTap II Embedded Logic Analyzer uses a comparison register 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:

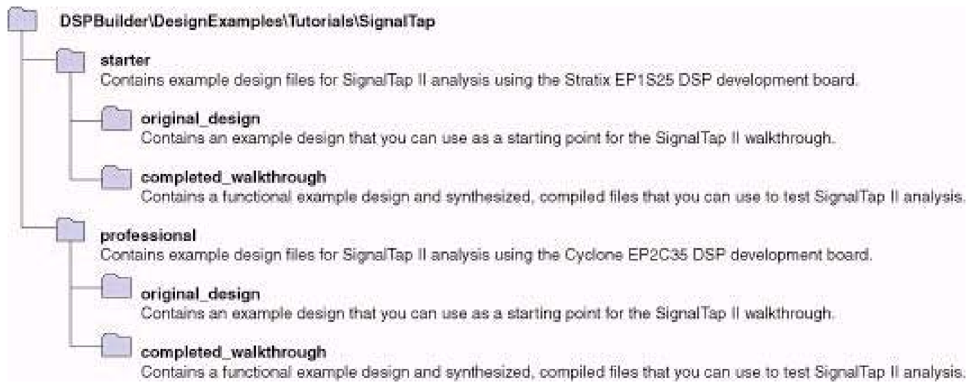
- Don’t Care
- Low
- High
- Rising Edge
- Falling Edge
- Either Edge

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



DSPBuilder\DesignExamples\Tutorials\SignalTap

starter
Contains example design files for SignalTap II analysis using the Stratix EP1S25 DSP development board.

original_design
Contains an example design that you can use as a starting point for the SignalTap II walkthrough.

completed_walkthrough
Contains a functional example design and synthesized, compiled files that you can use to test SignalTap II analysis.

professional
Contains example design files for SignalTap II analysis using the Cyclone EP2C35 DSP development board.

original_design
Contains an example design that you can use as a starting point for the SignalTap II walkthrough.

completed_walkthrough
Contains a functional example design and synthesized, compiled files that you can use to test SignalTap II analysis.

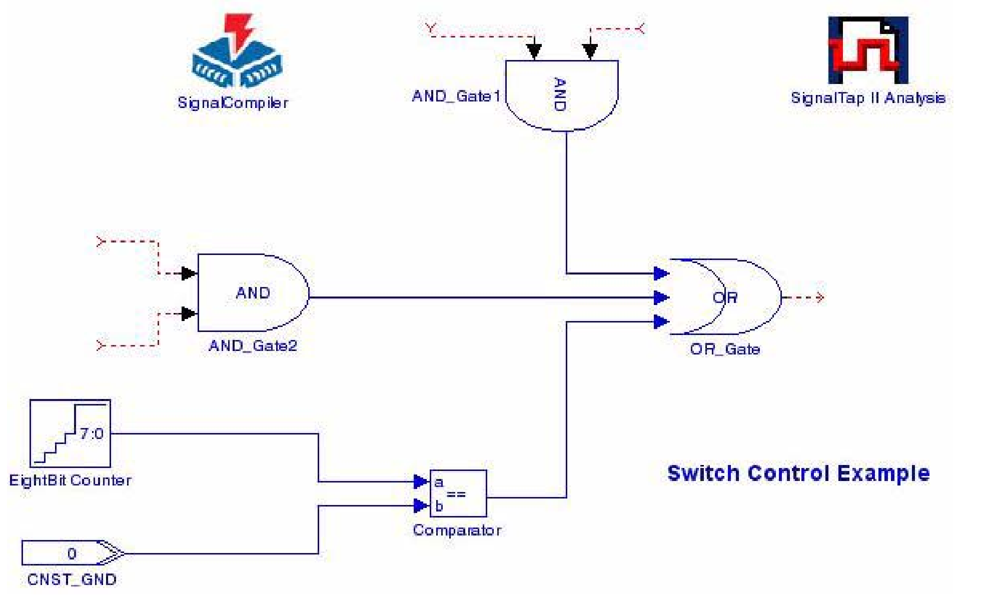
You can start from the design in the **original_design** directory and go through the complete walkthrough.

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 6–7.

Open the Walkthrough Example Design

Open the template **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 II EP2C35 development board.

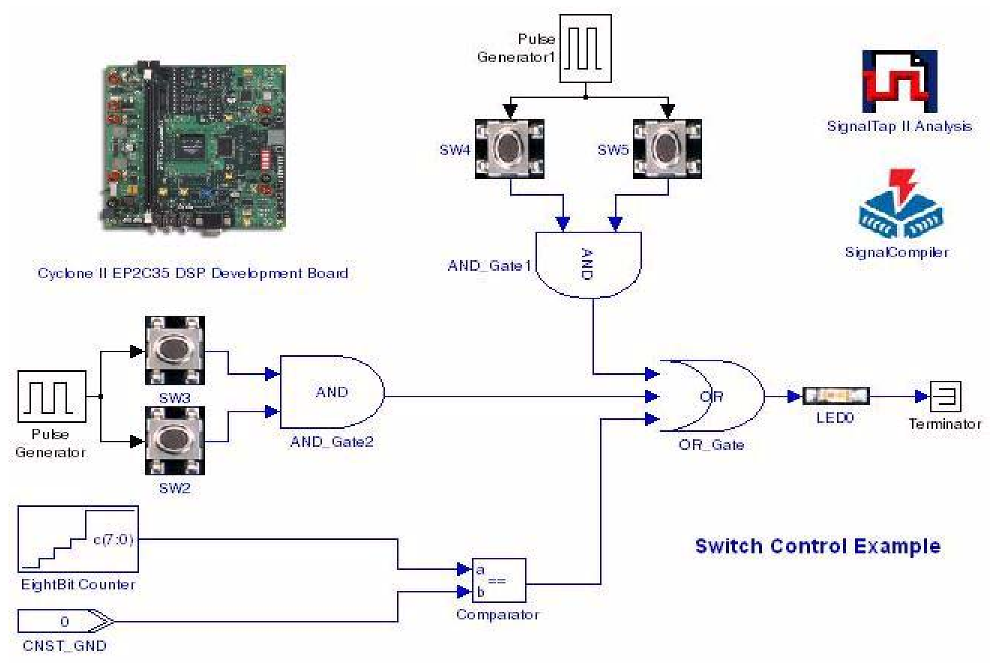
1. Select the **Boards** library from the **Altera DSP Builder Blockset** folder in the Simulink library browser. See Chapter 2, Getting Started Tutorial for instructions on accessing libraries.
2. Open the **CycloneIIEP2C35** folder. Drag and drop the `Cyclone II EP2C35 DSP Development Board` configuration 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 right-clicking the block and clicking **Rotate Block** on the Format menu.

5. Drag and drop the `LED0` block close to the `OR_Gate` block in your model. Connect this block to the `OR_Gate` output.
6. Select the Simulink **Sources** library. Drag and drop a `Pulse Generator` block near to 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

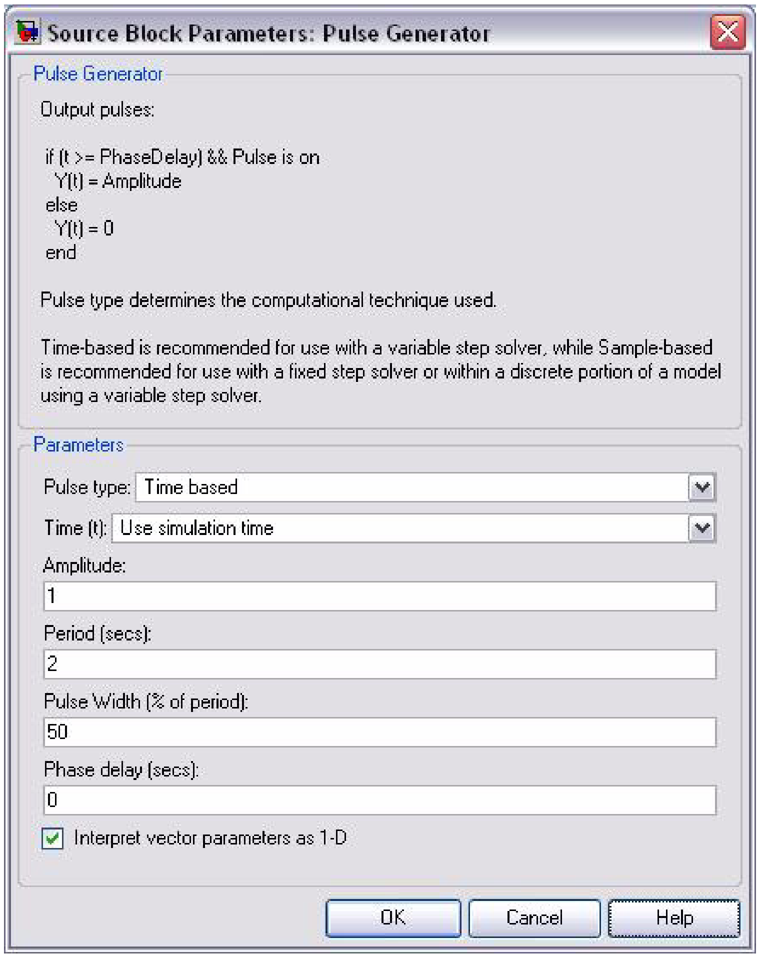


8. Use the **Block Parameters** dialog box to set the parameters shown in Table 6–1 for both pulse generator blocks (Figure 6–4).

Table 6–1. Parameters for the Pulse Generator Blocks

Parameter	**Value**
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


Source Block Parameters: Pulse Generator

Pulse Generator

Output pulses:

```
if (t >= PhaseDelay) && Pulse is on
  Y(t) = Amplitude
else
  Y(t) = 0
end
```

Pulse type determines the computational technique used.

Time-based is recommended for use with a variable step solver, while Sample-based is recommended for use with a fixed step solver or within a discrete portion of a model using a variable step solver.

Parameters

Pulse type: Time based

Time (t): Use simulation time

Amplitude: 1

Period (secs): 2

Pulse Width (% of period): 50

Phase delay (secs): 0

✓ Interpret vector parameters as 1-D

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

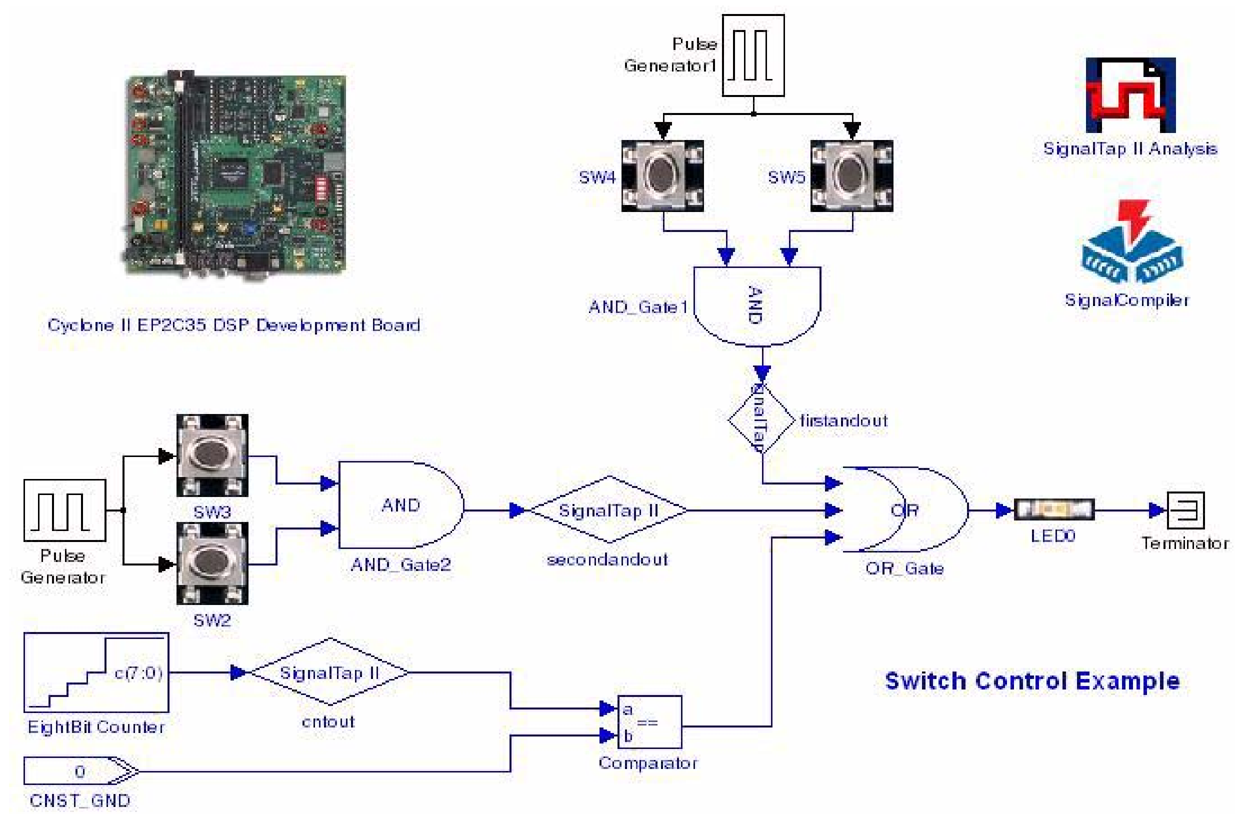
Specify the Nodes to Analyze

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 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. Click **Save** on the File menu.

Turn On the SignalTap II Option in Signal Compiler

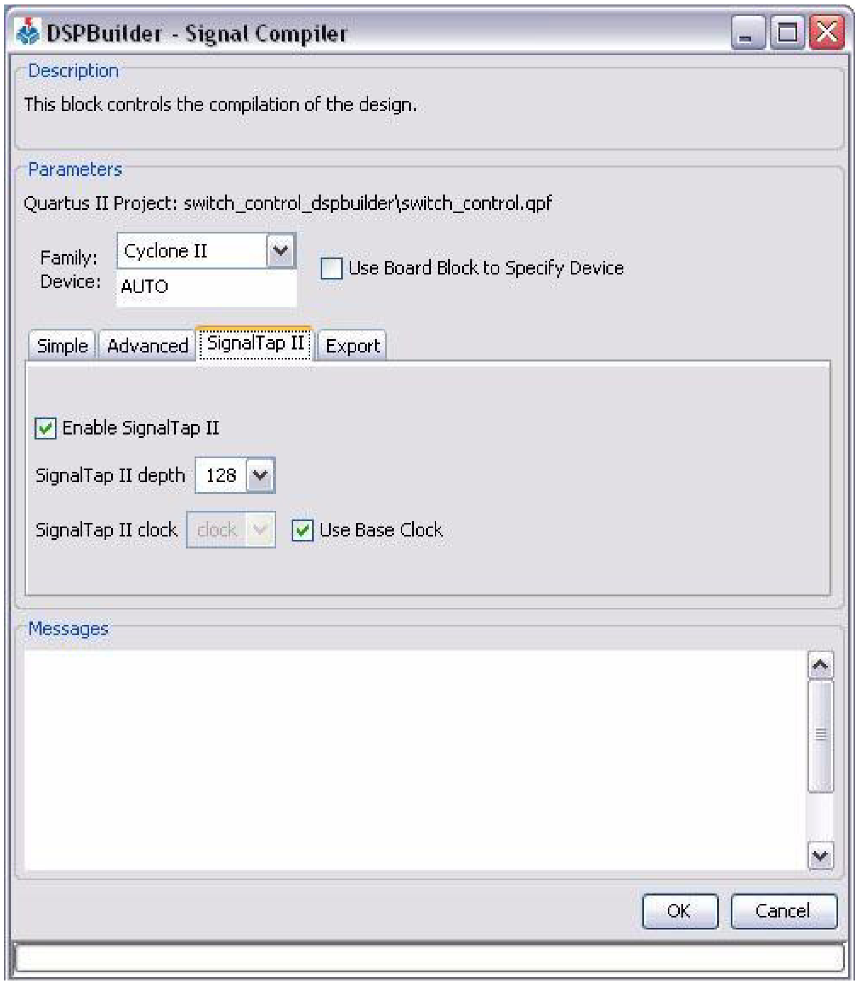
When you add node blocks to signals, each block is implicitly connected to the SignalTap II embedded logic analyzer.

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 click the **SignalTap II** tab in the **Signal Compiler** dialog box (Figure 6–6).

Figure 6–6. SignalTap II Tab in Signal Compiler


DSPBuilder - Signal Compiler

Description
This block controls the compilation of the design.

Parameters
Quartus II Project: switch_control_dspbuilder\switch_control.qpf

Family: Cyclone II
Device: AUTO
☐ Use Board Block to Specify Device

Simple | Advanced | SignalTap II | Export

✓ Enable SignalTap II
SignalTap II depth: 128
SignalTap II clock: clock ✓ Use Base Clock

Messages

2. Verify that the **Enable SignalTap II** option is on.

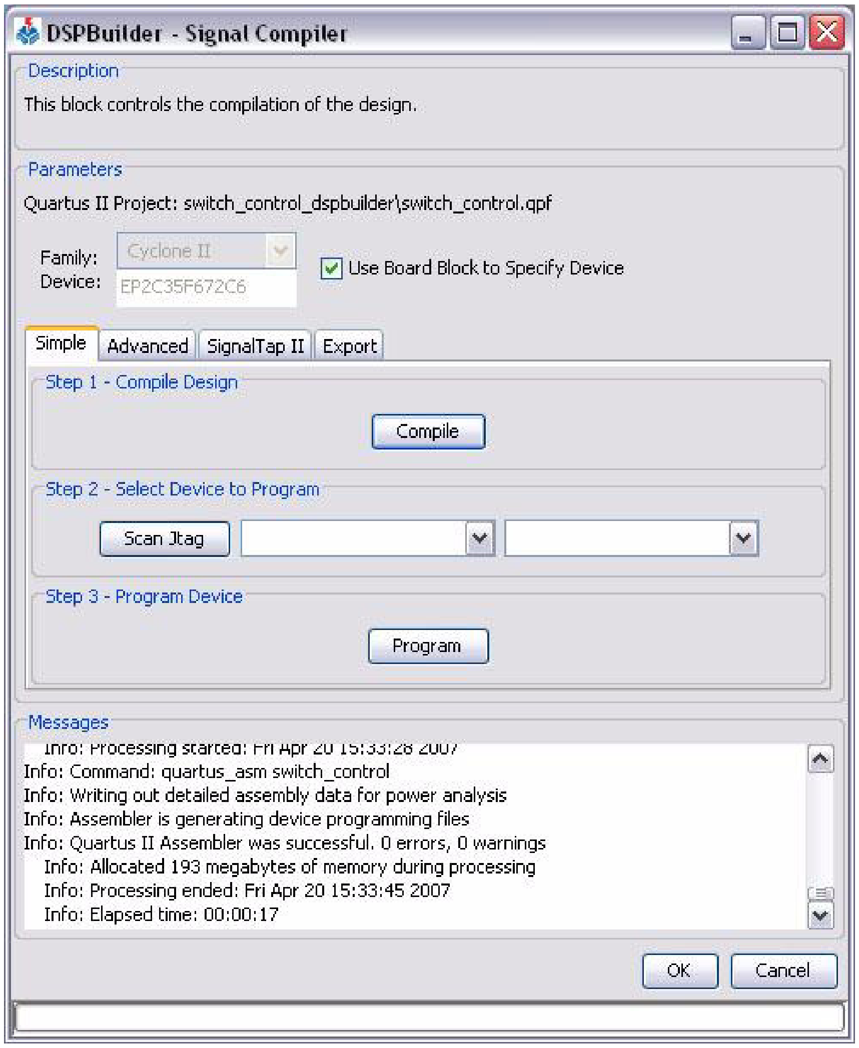
   When this option is on, `Signal Compiler` inserts an instance of the SignalTap II embedded logic analyzer into the design.

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 **SignalTap II depth** list.

4. Verify that the **Use Base Clock** option is on.

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

Figure 6–7. Simple Tab in Signal Compiler


DSPBuilder - Signal Compiler
Description
This block controls the compilation of the design.
Parameters
Quartus II Project: switch_control_dspbuilder\switch_control.qpf
Family: Cyclone II
Device: EP2C35F672C6
Use Board Block to Specify Device
Simple
Advanced
SignalTap II
Export
Step 1 - Compile Design
Compile
Step 2 - Select Device to Program
Scan Jtag
Step 3 - Program Device
Program
Messages
OK
Cancel

```
  Info: Processing started: Fri Apr 20 15:33:28 2007
Info: Command: quartus_asm switch_control
Info: Writing out detailed assembly data for power analysis
Info: Assembler is generating device programming files
Info: Quartus II Assembler was successful. 0 errors, 0 warnings
   Info: Allocated 193 megabytes of memory during processing
   Info: Processing ended: Fri Apr 20 15:33:45 2007
   Info: Elapsed time: 00:00:17
```

6. Click the **Compile** button.

   When the conversion is complete, information messages in the dialog box 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.

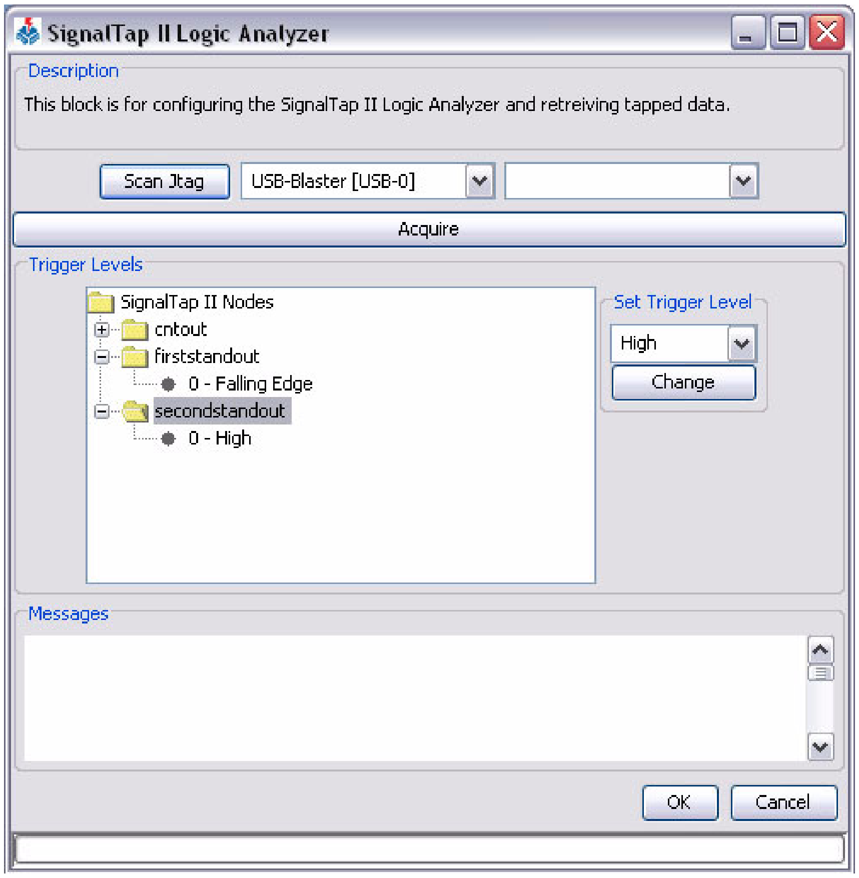
7. Click **Scan Jtag** and select the appropriate download cable and device (for example, **USB-Blaster** cable and **EP2C35** device).
8. Click **Program** to download your design to the development board.
9. Click **OK**.

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 following trigger condition settings for the `firstandout` block:
   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` block.

The SignalTap II embedded logic analyzer captures data for analysis when it detects all trigger patterns simultaneously 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 now 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 Logic 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 Logic Analyzer** 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

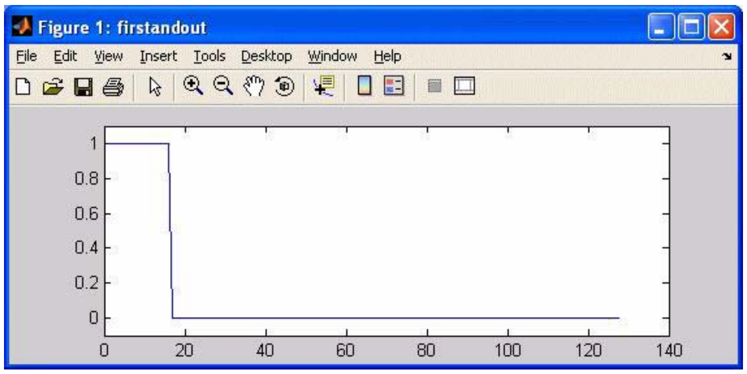


Figure 6–10 shows the MATLAB plot for the SignalTap II node `secondandout`.

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

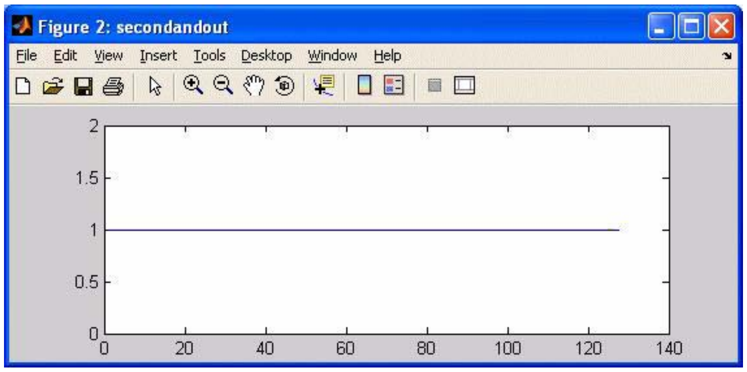
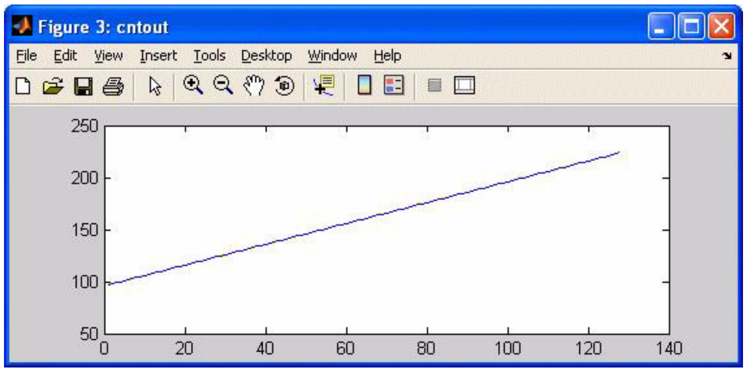


Figure 6–10 shows the MATLAB plot for the SignalTap II node `cntout`.

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 AltLab Library chapter in the *DSP Builder Reference Manual*.

7. Using the Interfaces Library

Introduction

This chapter describes how to use the 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 that can contain 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-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 Interface Specifications* provide 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 Interface Specifications*.

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

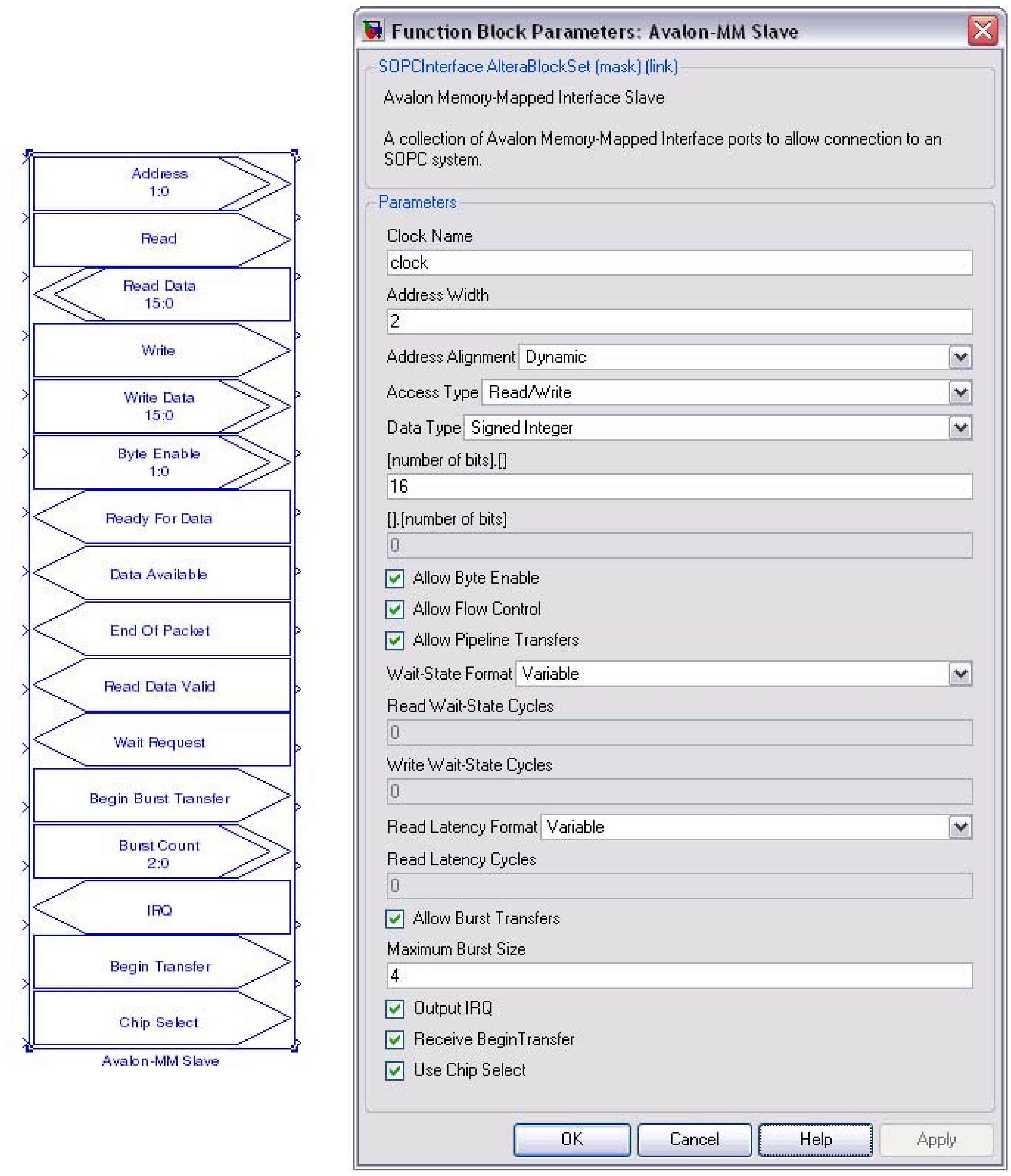
The `Avalon-MM Slave` block supports the following signals:

- `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



Function Block Parameters: Avalon-MM Slave

SOPCInterface AlteraBlockSet (mask) (link)

Avalon Memory-Mapped Interface Slave

A collection of Avalon Memory-Mapped Interface ports to allow connection to an SOPC system.

Parameters

Clock Name: clock

Address Width: 2

Address Alignment: Dynamic

Access Type: Read/Write

Data Type: Signed Integer

[number of bits].[]: 16

[].[number of bits]: 0

✓ Allow Byte Enable

✓ Allow Flow Control

✓ Allow Pipeline Transfers

Wait-State Format: Variable

Read Wait-State Cycles: 0

Write Wait-State Cycles: 0

Read Latency Format: Variable

Read Latency Cycles: 0

✓ Allow Burst Transfers

Maximum Burst Size: 4

✓ Output IRQ

✓ Receive BeginTransfer

✓ Use Chip Select

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.

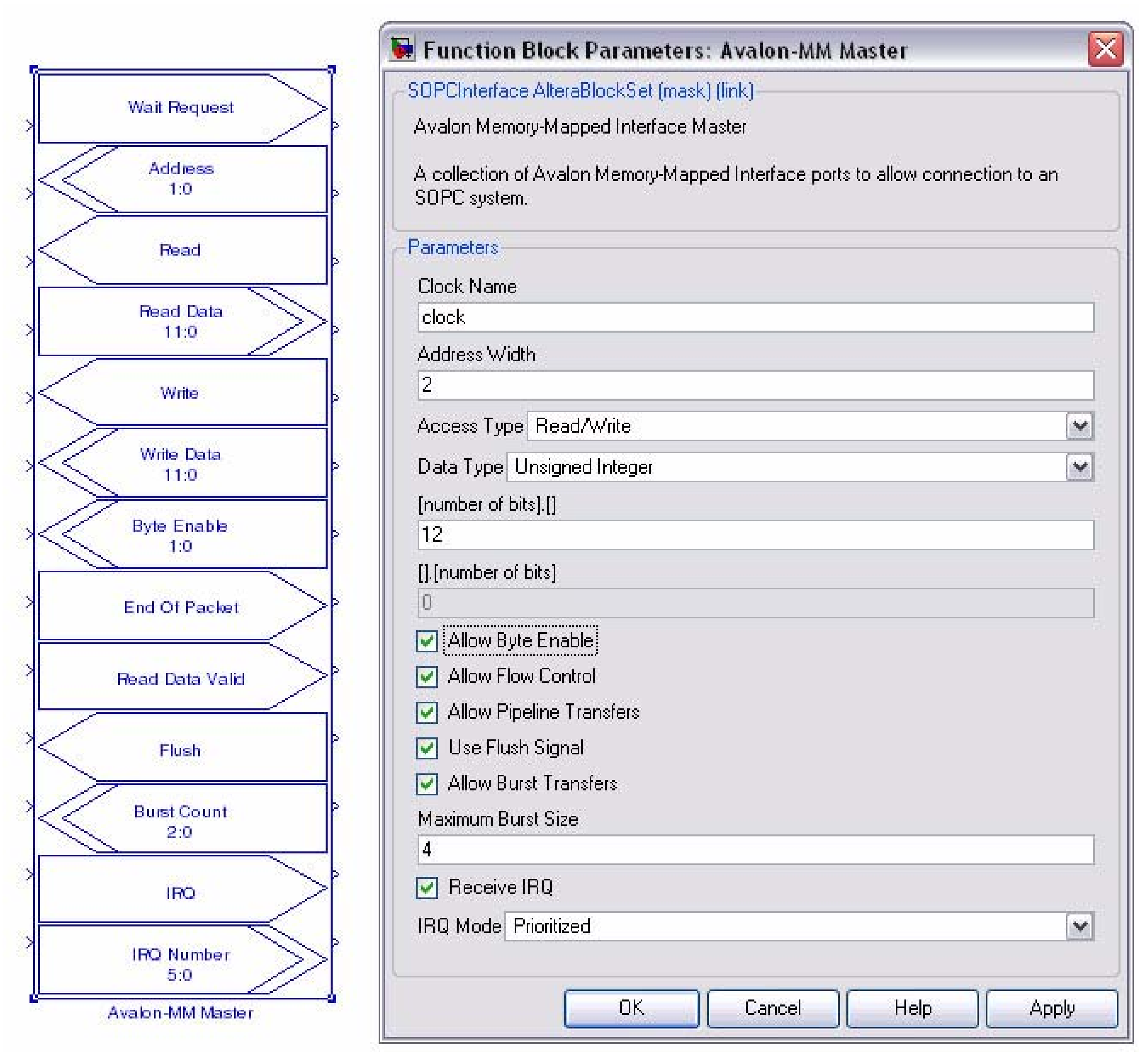
The `Avalon-MM Master` block is similar to the `Avalon-MM Slave` block and supports the following signals:

- `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



Function Block Parameters: Avalon-MM Master

SOPCInterface AlteraBlockSet (mask) (link)

Avalon Memory-Mapped Interface Master

A collection of Avalon Memory-Mapped Interface ports to allow connection to an SOPC system.

Parameters

Clock Name: clock

Address Width: 2

Access Type: Read/Write

Data Type: Unsigned Integer

[number of bits].[]: 12

[].[number of bits]: 0

[✓] Allow Byte Enable

[✓] Allow Flow Control

[✓] Allow Pipeline Transfers

[✓] Use Flush Signal

[✓] Allow Burst Transfers

Maximum Burst Size: 4

[✓] Receive IRQ

IRQ Mode: Prioritized

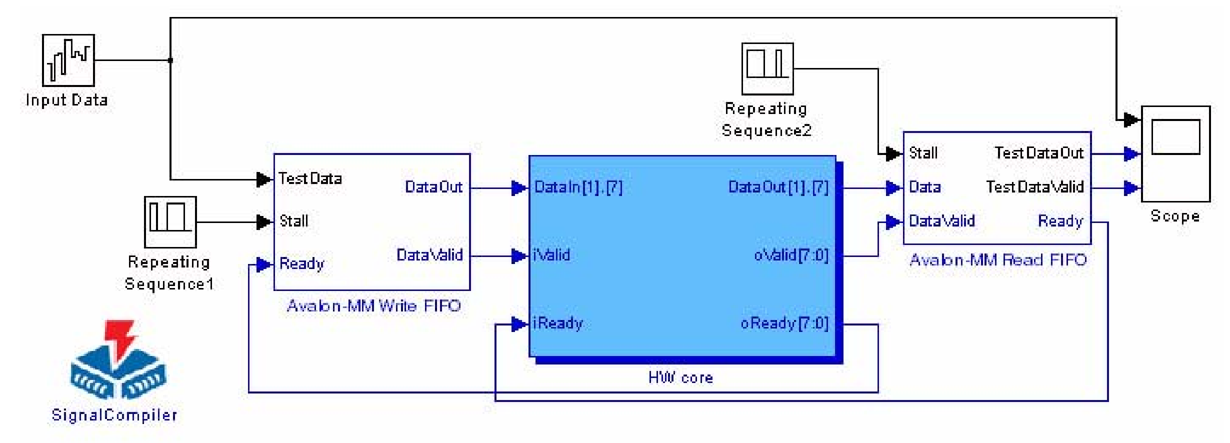
Wrapped Blocks

While the `Avalon-MM Master` and `Avalon-MM Slave` interface blocks allow you to generate a SOPC Builder 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 that handles 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

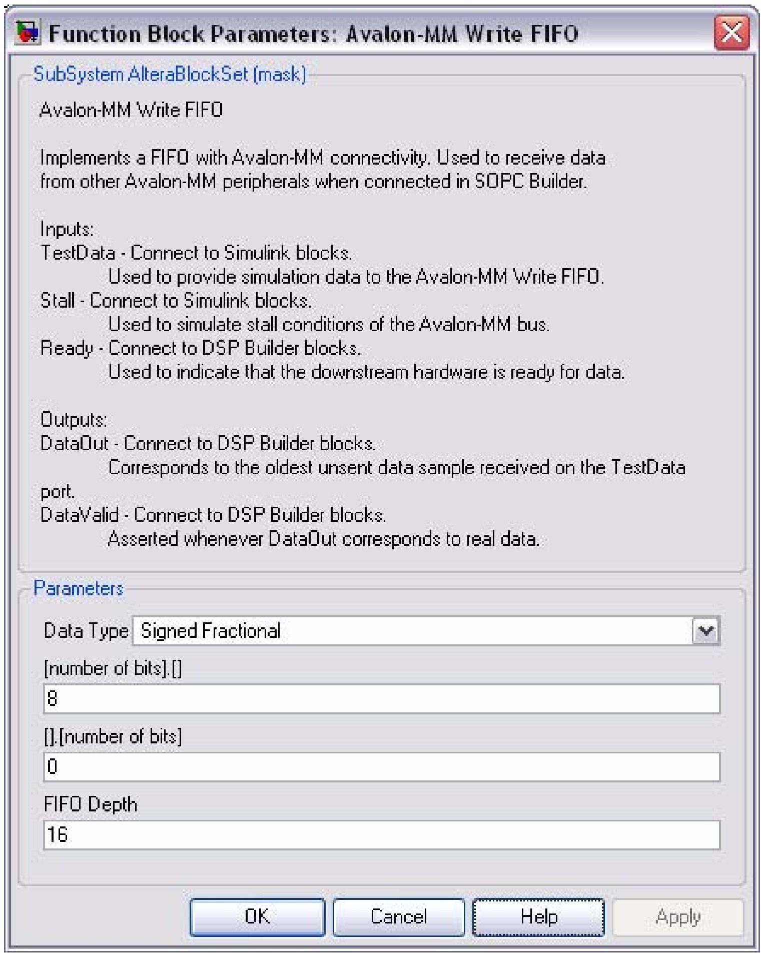
An `Avalon-MM Write FIFO` has the following ports:

- **TestData** (input): This port should be connected 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 a Simulink block and is used to simulate stall conditions of the Avalon-MM bus and hence underflow to the SOPC Builder 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 a DSP Builder block and is used to indicate that the downstream hardware is ready for data.
- **DataOut** (output): This port should be connected to a DSP Builder block and corresponds to the oldest unsent data sample received on the `TestData` port.
- **DataValid** (output): This port should be connected to a DSP Builder block and is asserted whenever `DataOut` corresponds to real data.

Double-clicking on an `Avalon-MM Write FIFO` block opens the **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



Function Block Parameters: Avalon-MM Write FIFO

SubSystem AlteraBlockSet (mask)

Avalon-MM Write FIFO

Implements a FIFO with Avalon-MM connectivity. Used to receive data
from other Avalon-MM peripherals when connected in SOPC Builder.

Inputs:
TestData - Connect to Simulink blocks.
    Used to provide simulation data to the Avalon-MM Write FIFO.
Stall - Connect to Simulink blocks.
    Used to simulate stall conditions of the Avalon-MM bus.
Ready - Connect to DSP Builder blocks.
    Used to indicate that the downstream hardware is ready for data.

Outputs:
DataOut - Connect to DSP Builder blocks.
    Corresponds to the oldest unsent data sample received on the TestData
port.
DataValid - Connect to DSP Builder blocks.
    Asserted whenever DataOut corresponds to real data.

Parameters

Data Type: Signed Fractional

[number of bits].[]: 8

[].[number of bits]: 0

FIFO Depth: 16

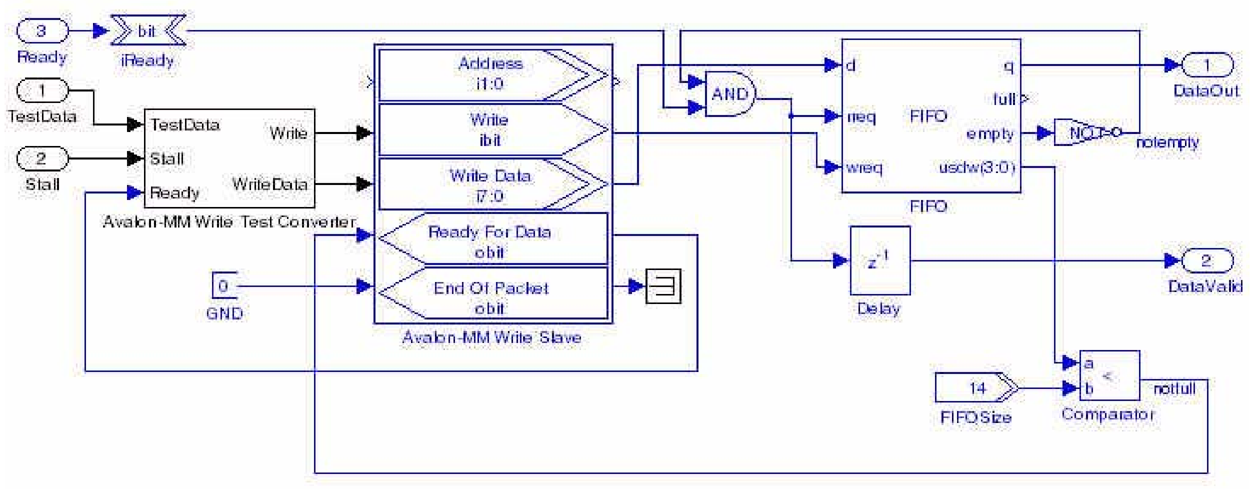
Refer to the *DSP Builder Reference Manual* for information about these parameters.

You can open the hierarchy below the `Avalon-MM Write FIFO` block by right-clicking the block and clicking **Look Under Mask** on the pop-up 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` block handles caching and conversion of Simulink/MATLAB data into accesses over the Avalon-MM interface and can be used to test the functionality of your design. The `Avalon-MM Write Test Converter` 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 Builder 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 a DSP Builder block and should be connected to outgoing data from the user design.
- **DataValid** (input): This port should be connected to a DSP Builder block and should be asserted whenever the signal on the `Data` port corresponds to real data.
- **TestDataOut** (output): This port should be connected to a Simulink block and corresponds to data received over the Avalon-MM bus.
- **TestDataValid** (output): This port should be connected to a Simulink block and is asserted whenever `TestDataOut` corresponds to real data.
- **Ready** (output): When asserted, indicates that the block is ready to receive data.

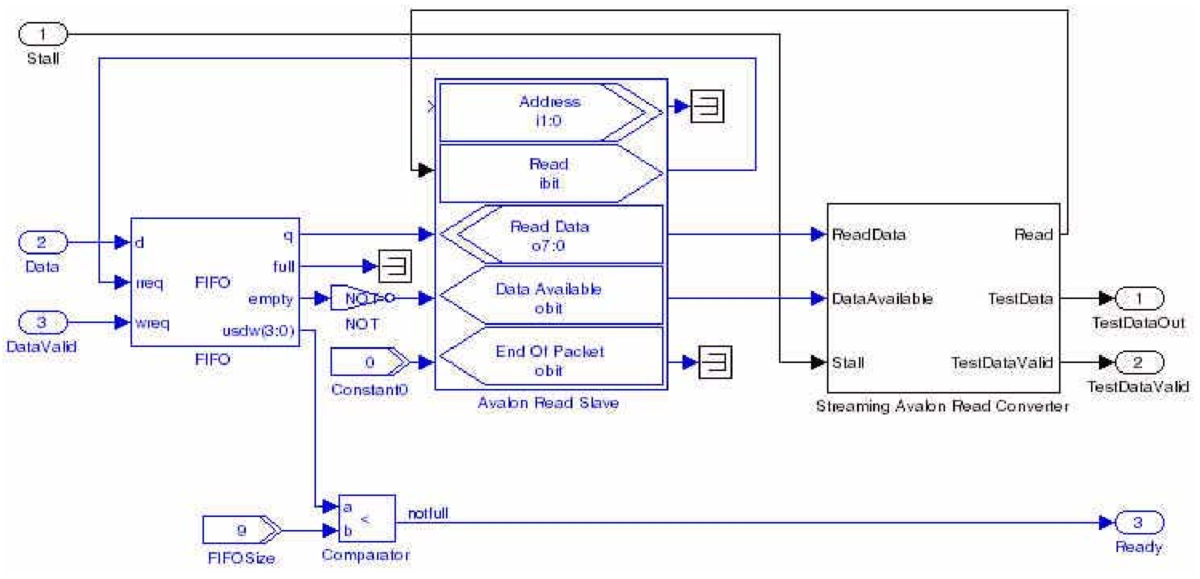
Double-clicking on an `Avalon-MM Write FIFO` block opens the **Block Parameters** dialog box which can be used to set parameters for the data type, data width and FIFO depth.

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 right-clicking on the block and choosing **Look Under Mask** from the pop-up menu.

The internal content of an Avalon-MM Read FIFO is shown in Figure 7–6.

Figure 7–6. Avalon-MM Read FIFO Content



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

Avalon-MM Interface Blocks Walkthrough

This walkthrough describes how to interface a design 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 II 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

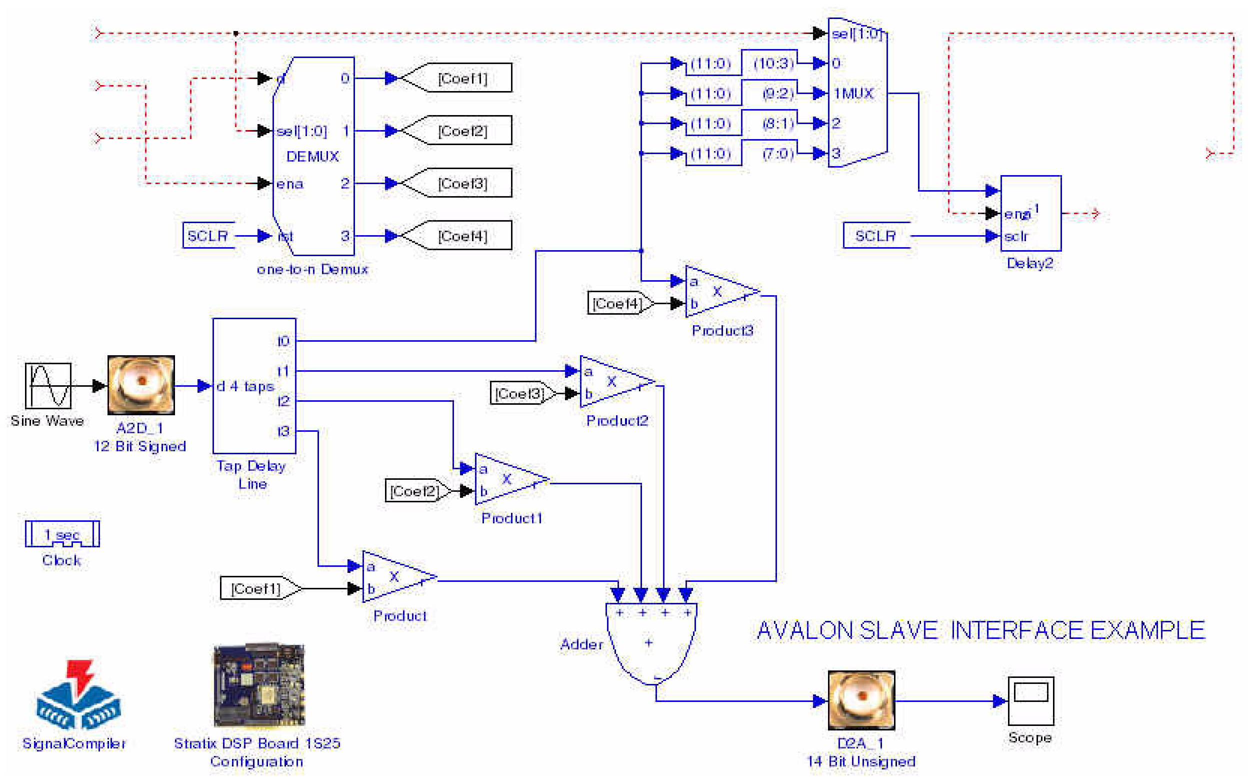
To complete the example design, perform the following steps:

1. Click **Open** on the 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 clicking **Save As** on the 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.

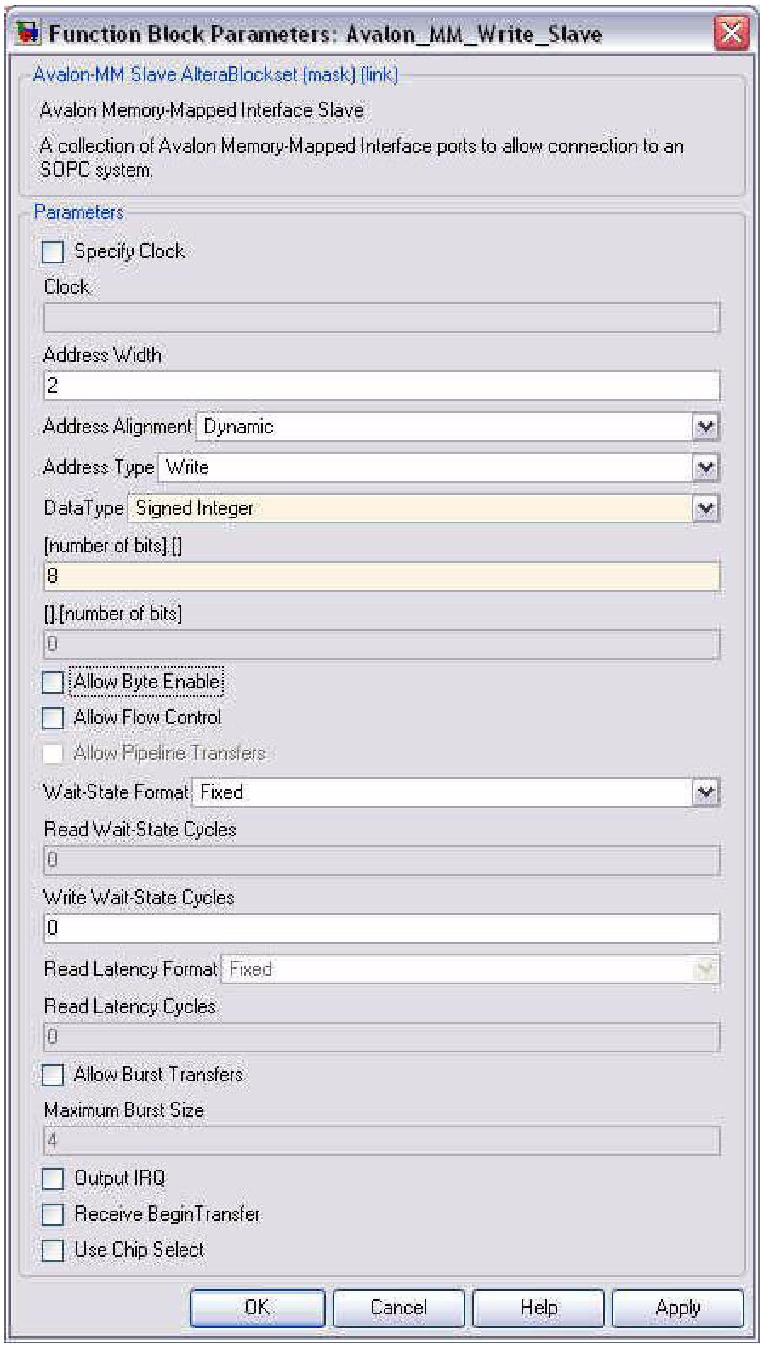
6. Expand the **Altera DSP Builder Blockset** in the Simulink Library Browser and select **Avalon Memory-Mapped** in the Interfaces library.

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

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

9. Select **Write** for the address type, **Signed Integer** for the data type, and specify **8** bits for the data width. Turn off the **Allow Byte Enable** option (Figure 7–8 on page 7–11).

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



Function Block Parameters: Avalon_MM_Write_Slave

Avalon-MM Slave AlteraBlockset (mask) (link)

Avalon Memory-Mapped Interface Slave

A collection of Avalon Memory-Mapped Interface ports to allow connection to an SOPC system.

Parameters

- [ ] Specify Clock

Clock:

Address Width: 2

Address Alignment: Dynamic

Address Type: Write

DataType: Signed Integer

[number of bits].[]: 8

[].[number of bits]: 0

- [ ] Allow Byte Enable
- [ ] Allow Flow Control
- [ ] Allow Pipeline Transfers

Wait-State Format: Fixed

Read Wait-State Cycles: 0

Write Wait-State Cycles: 0

Read Latency Format: Fixed

Read Latency Cycles: 0

- [ ] Allow Burst Transfers

Maximum Burst Size: 4

- [ ] Output IRQ
- [ ] Receive BeginTransfer
- [ ] Use Chip Select

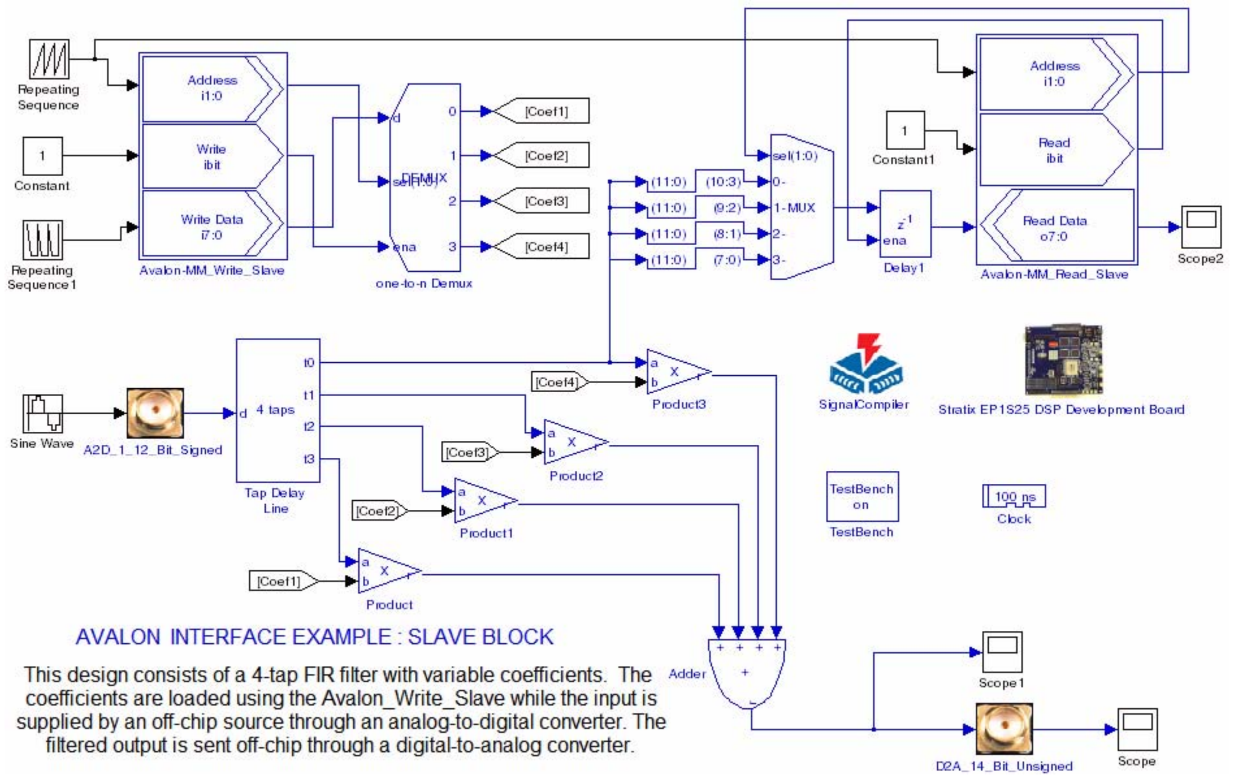
10. Click **OK**.

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

11. 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.

Figure 7–9. topavalon.mdl Example Design



AVALON INTERFACE EXAMPLE : SLAVE BLOCK

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

12. 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.`
13. Double-click on the `Avalon_MM_Read_Slave` block to bring up the **Block Parameters** dialog box.
14. Select **Read** for the address type, `Signed Integer` for the data type, and specify **8** bits for the data width.
15. 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`.
16. Complete the design by connecting the `Avalon_MM_Read_Slave` ports as shown in Figure 7–9.
17. Click **Save** on the File menu in the model window to save your model.

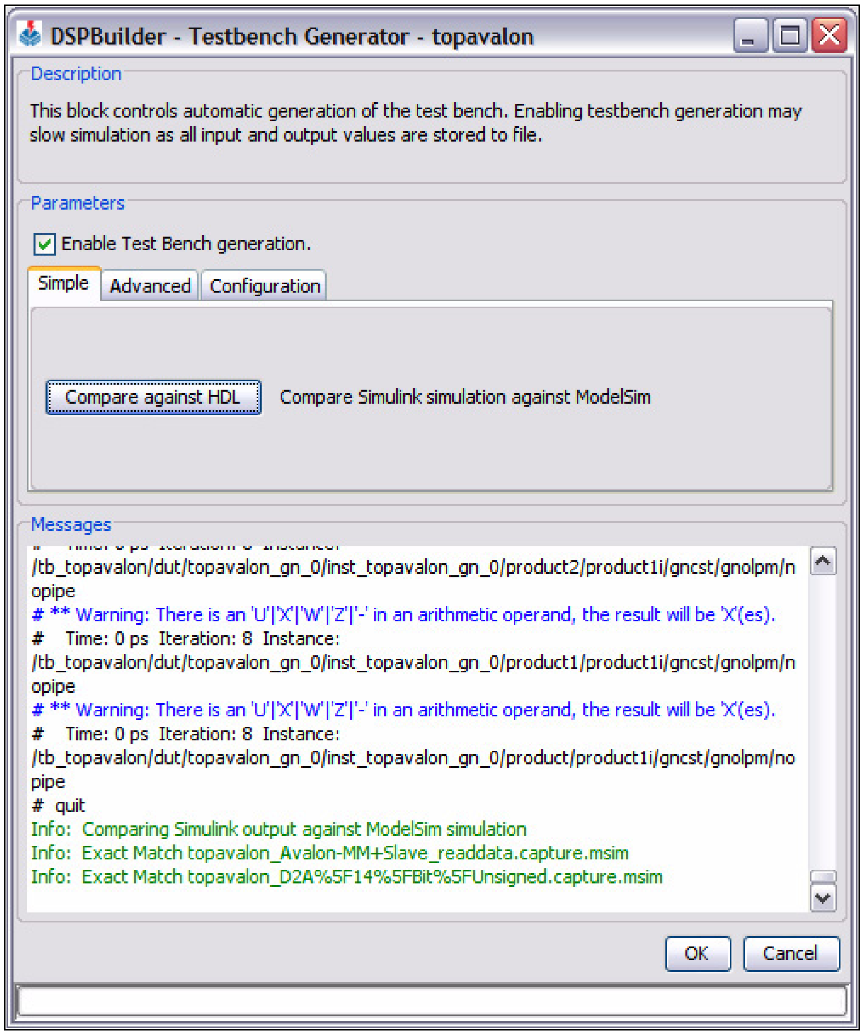
Verify Your Design

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

Figure 7–10. TestBench Dialog Box



This block controls automatic generation of the test bench. Enabling testbench generation may slow simulation as all input and output values are stored to file.

[✓] Enable Test Bench generation.

Compare Simulink simulation against ModelSim

```
/tb_topavalon/dut/topavalon_gn_0/inst_topavalon_gn_0/product2/product1i/gncst/gnolpm/n
opipe
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ps  Iteration: 8  Instance:
/tb_topavalon/dut/topavalon_gn_0/inst_topavalon_gn_0/product1/product1i/gncst/gnolpm/n
opipe
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ps  Iteration: 8  Instance:
/tb_topavalon/dut/topavalon_gn_0/inst_topavalon_gn_0/product/product1i/gncst/gnolpm/no
pipe
#  quit
Info:  Comparing Simulink output against ModelSim simulation
Info:  Exact Match topavalon_Avalon-MM+Slave_readdata.capture.msim
Info:  Exact Match topavalon_D2A%5F14%5FBit%5FUnsigned.capture.msim
```

This process generates HDL, runs Simulink and ModelSim, and then compares the simulation results. Progress messages are issued in the dialog box which should complete with a message "`Exact Match`".

3. Click **OK**.

Instantiate Your Design in SOPC Builder

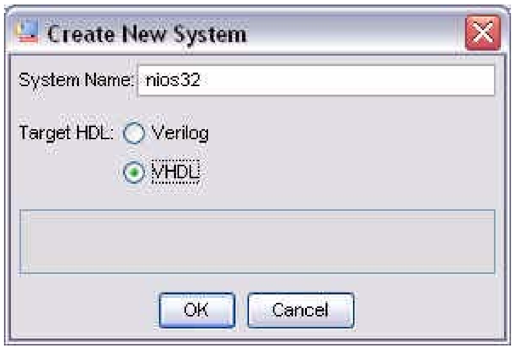
To instantiate your design as a custom peripheral to the Nios II embedded processor in SOPC Builder, perform the following steps:

1. Start the Quartus® II software.
2. On the File menu in the Quartus II software, click **New Project Wizard**.
   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. On the Tools menu, click **Tcl Scripts** and perform the following steps:
   a. Select **topavalon_add.tcl** in the Project folder.
   b. Click **Run** to load your **.mdl** file and other required files into the Quartus II project.
4. On the Tools menu, click **SOPC Builder** and set the following parameters in the **Create New System** dialog box (Figure 7–11):
   a. Specify `nios32` as the system name.
   b. Select **VHDL** for the target HDL.
   c. Click **OK**.

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

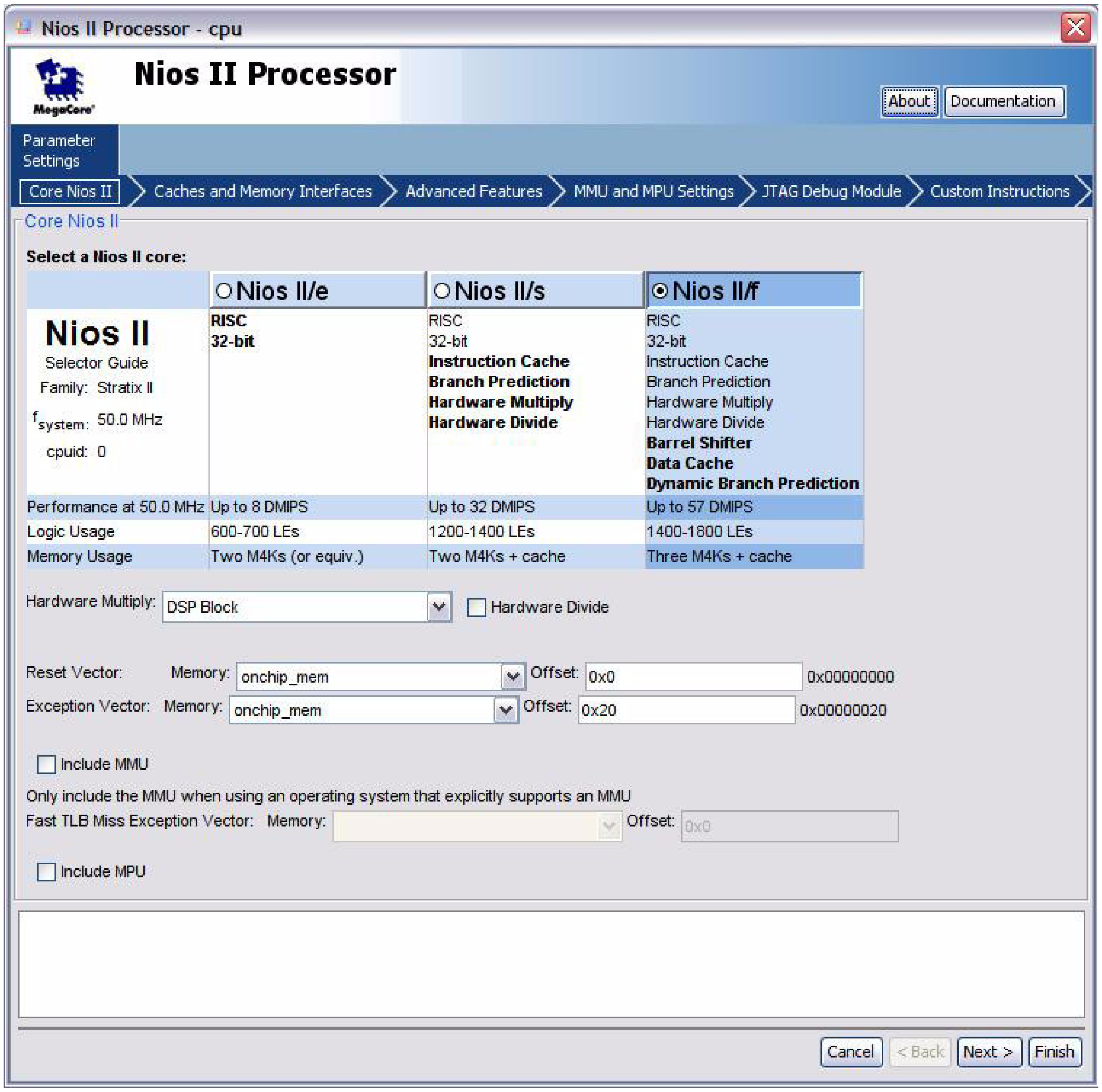


System Name: nios32
Target HDL: ( ) Verilog [✓ VHDL]

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

6. Double-click **Nios II Processor** in the **System Contents** tab to display the MegaWizard interface (Figure 7–12).

Figure 7–12. Nios II Processor Configuration



Nios II Processor - cpu

Nios II Processor

Core Nios II

Select a Nios II core:

Nios II Selector Guide Family: Stratix II f_{system}: 50.0 MHz cpuid: 0	○ Nios II/e	○ Nios II/s	◉ Nios II/f
	RISC **32-bit**	RISC 32-bit **Instruction Cache** **Branch Prediction** **Hardware Multiply** **Hardware Divide**	RISC 32-bit Instruction Cache Branch Prediction Hardware Multiply Hardware Divide **Barrel Shifter** **Data Cache** **Dynamic Branch Prediction**
Performance at 50.0 MHz	Up to 8 DMIPS	Up to 32 DMIPS	Up to 57 DMIPS
Logic Usage	600-700 LEs	1200-1400 LEs	1400-1800 LEs
Memory Usage	Two M4Ks (or equiv.)	Two M4Ks + cache	Three M4Ks + cache

Hardware Multiply: DSP Block ☐ Hardware Divide

Reset Vector: Memory: onchip_mem Offset: 0x0 0x00000000
Exception Vector: Memory: onchip_mem Offset: 0x20 0x00000020

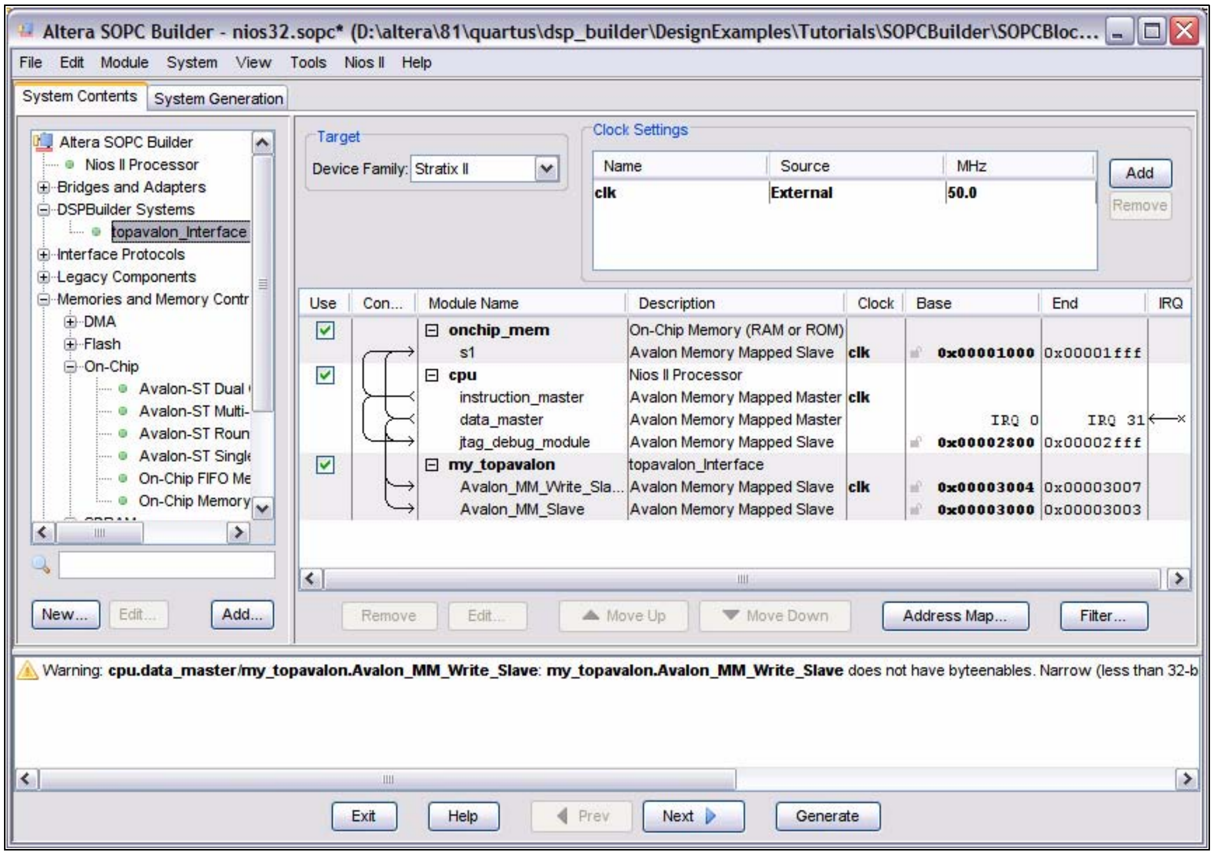
☐ Include MMU
Only include the MMU when using an operating system that explicitly supports an MMU
Fast TLB Miss Exception Vector: Memory: Offset: 0x0

☐ Include MPU

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 **topavalon_interface** module to include it in your Nios II system (see Figure 7–13).

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



Altera SOPC Builder - nios32.sopc* (D:\altera\81\quartus\dsp_builder\DesignExamples\Tutorials\SOPCBuilder\SOPCBloc...

Target
Device Family: Stratix II

Clock Settings

Name	Source	MHz
clk	External	50.0

Use	Con...	Module Name	Description	Clock	Base	End	IRQ
✓		onchip_mem	On-Chip Memory (RAM or ROM)				
		s1	Avalon Memory Mapped Slave	clk	0x00001000	0x00001fff	
✓		cpu	Nios II Processor				
		instruction_master	Avalon Memory Mapped Master	clk			
		data_master	Avalon Memory Mapped Master		IRQ 0	IRQ 31	
		jtag_debug_module	Avalon Memory Mapped Slave		0x00002800	0x00002fff	
✓		my_topavalon	topavalon_Interface				
		Avalon_MM_Write_Sla...	Avalon Memory Mapped Slave	clk	0x00003004	0x00003007	
		Avalon_MM_Slave	Avalon Memory Mapped Slave		0x00003000	0x00003003	

Warning: cpu.data_master/my_topavalon.Avalon_MM_Write_Slave: my_topavalon.Avalon_MM_Write_Slave does not have byteenables. Narrow (less than 32-b

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 click **Auto-Assign Base Addresses** on 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 a Nios II embedded processor. The edge detected image is output through a VGA controller. The DSP Builder model uses Simulink to read in the original image and to capture the edge detected result.

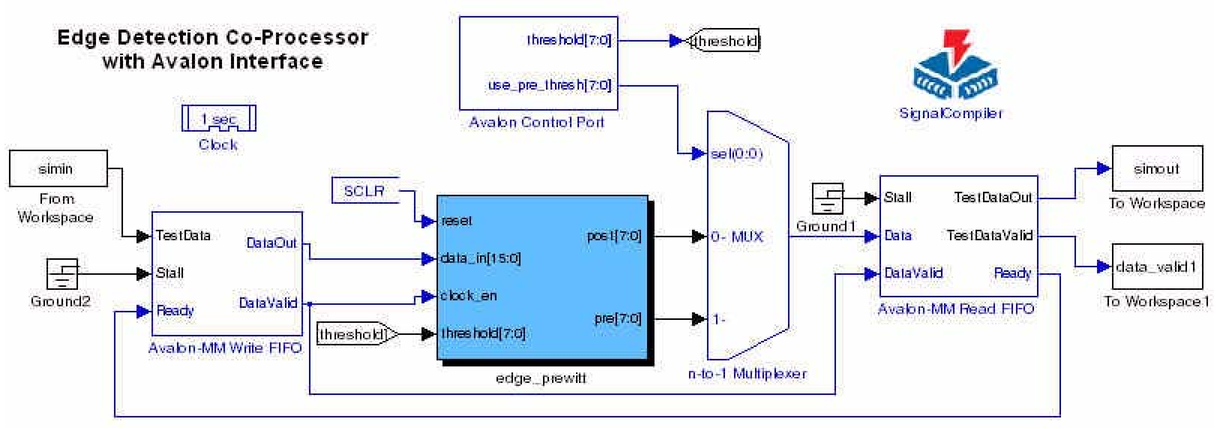
Open the Walkthrough Example Design

To open the example design, perform the following steps:

1. Click **Open** on the 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

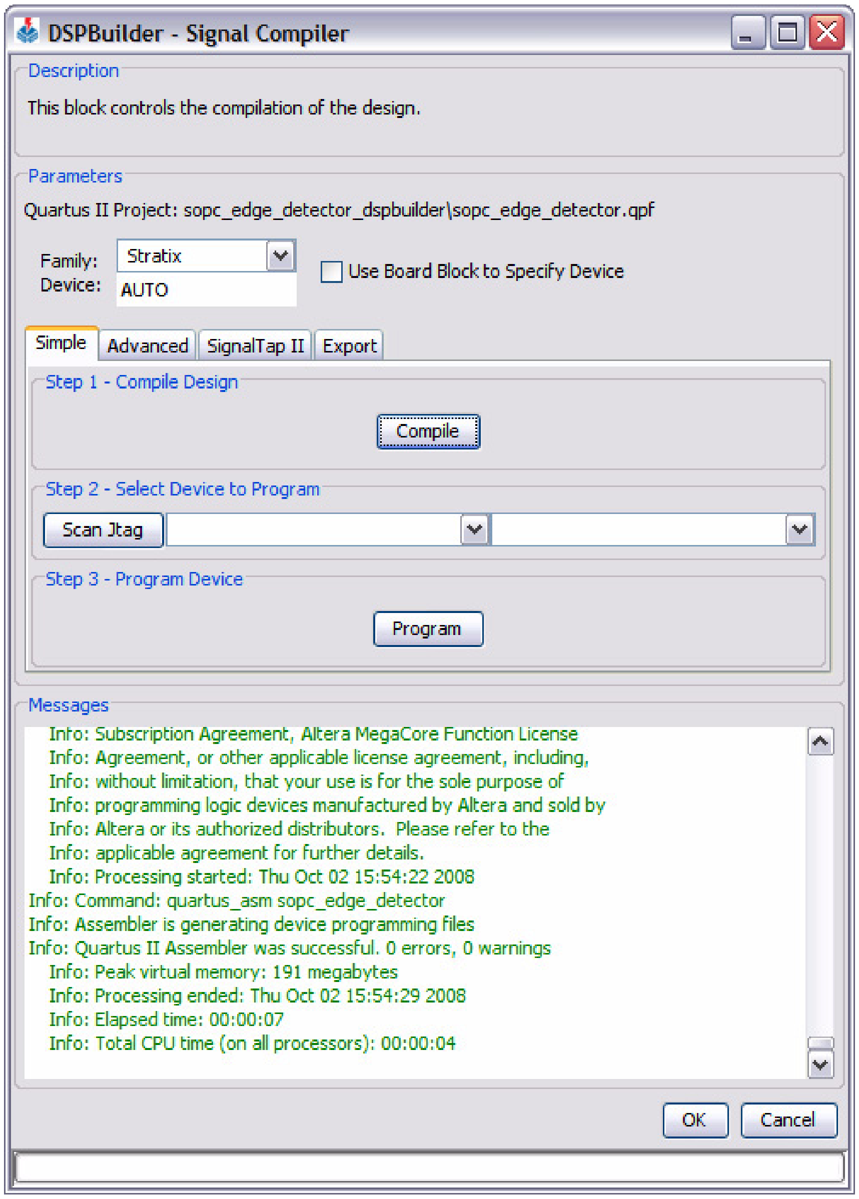
In this example, you use the `Signal Compiler` block to verify that the design generates valid HDL.

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 to verify the design:

1. Double-click the `Signal Compiler` block.
2. Select the family and device for the DSP Development board you are using. The walkthrough design is configured for a Stratix 1S25 board (Figure 7–15).
3. Click **Compile**.

Figure 7–15. Signal Compiler Dialog Box



This block controls the compilation of the design.

Quartus II Project: sopc_edge_detector_dspbuilder\sopc_edge_detector.qpf

Family: Stratix
Device: AUTO

Use Board Block to Specify Device

```
   Info: Subscription Agreement, Altera MegaCore Function License
   Info: Agreement, or other applicable license agreement, including,
   Info: without limitation, that your use is for the sole purpose of
   Info: programming logic devices manufactured by Altera and sold by
   Info: Altera or its authorized distributors.  Please refer to the
   Info: applicable agreement for further details.
   Info: Processing started: Thu Oct 02 15:54:22 2008
Info: Command: quartus_asm sopc_edge_detector
Info: Assembler is generating device programming files
Info: Quartus II Assembler was successful. 0 errors, 0 warnings
   Info: Peak virtual memory: 191 megabytes
   Info: Processing ended: Thu Oct 02 15:54:29 2008
   Info: Elapsed time: 00:00:07
   Info: Total CPU time (on all processors): 00:00:04
```

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:

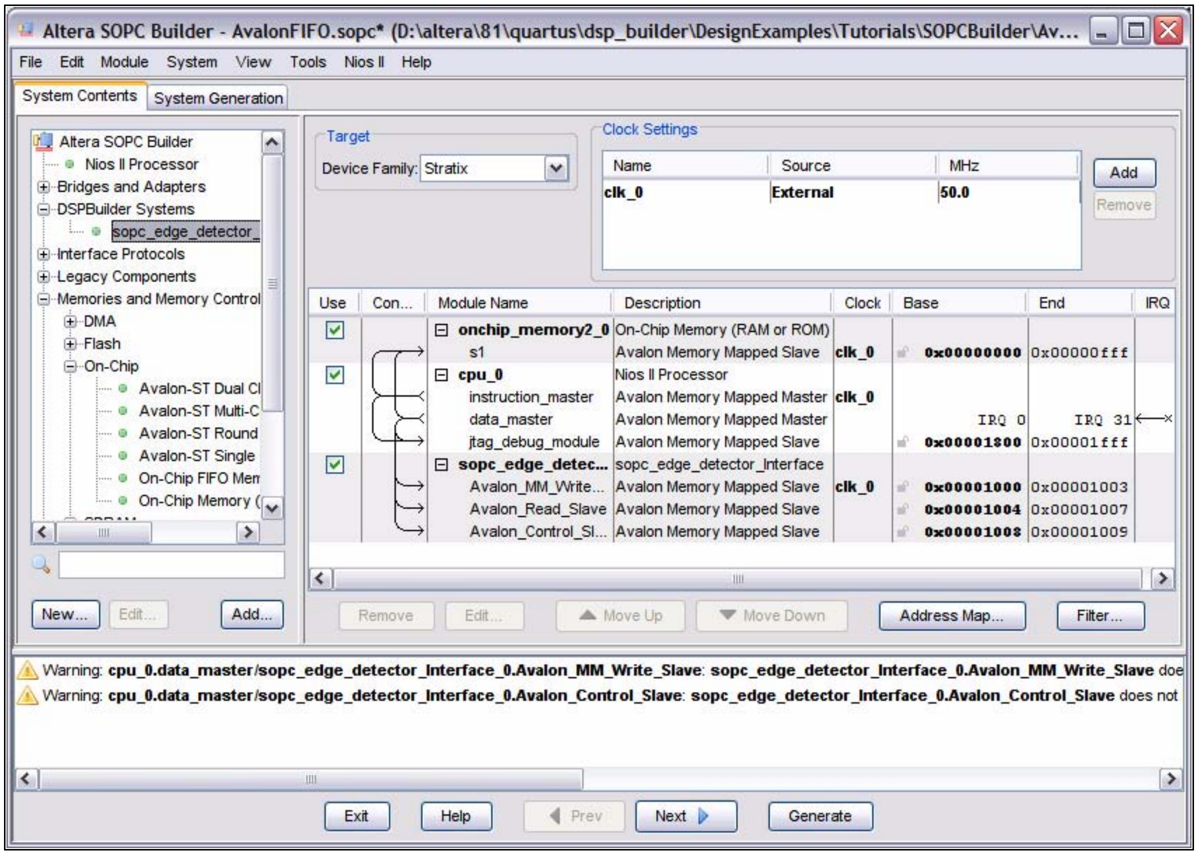
1. Start the Quartus II software.
2. On the File menu in the Quartus II software, click **New Project Wizard** and set the following options:
   a. Specify the working directory for your project by browsing to the <*DSP Builder install path*>**\DesignExamples\Tutorials\SOPCBuilder\AvalonFIFO** directory.
   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. On the Tools menu, click **Tcl Scripts** and set the following options:
   a. Load your design by selecting **sopc_edge_detector_add.tcl** in the Project folder.
   b. Click **Run**.
4. On the Tools menu, click **SOPC Builder** to display the **Create New System** dialog box.
   a. Specify `AvalonFIFO` as the system name.
   b. Select **VHDL** for the target HDL.
   c. Click **OK**.
5. Click the **System Contents** tab in SOPC Builder and set the following options:
   a. Expand **Memories and Memory Controllers**.
   b. Expand **On-Chip** and double-click **On Chip Memory (RAM or ROM)**.
   c. 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_memory2_0** 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



Device Family: Stratix

Name	Source	MHz
clk_0	External	50.0

Use	Con...	Module Name	Description	Clock	Base	End	IRQ
✓		⊟ onchip_memory2_0	On-Chip Memory (RAM or ROM)				
		s1	Avalon Memory Mapped Slave	clk_0	0x00000000	0x00000fff	
✓		⊟ cpu_0	Nios II Processor				
		instruction_master	Avalon Memory Mapped Master	clk_0			
		data_master	Avalon Memory Mapped Master		IRQ 0	IRQ 31	
		jtag_debug_module	Avalon Memory Mapped Slave		0x00001800	0x00001fff	
✓		⊟ sopc_edge_detec...	sopc_edge_detector_Interface				
		Avalon_MM_Write...	Avalon Memory Mapped Slave	clk_0	0x00001000	0x00001003	
		Avalon_Read_Slave	Avalon Memory Mapped Slave		0x00001004	0x00001007	
		Avalon_Control_Sl...	Avalon Memory Mapped Slave		0x00001008	0x00001009	

Warning: **cpu_0.data_master/sopc_edge_detector_Interface_0.Avalon_MM_Write_Slave**: **sopc_edge_detector_Interface_0.Avalon_MM_Write_Slave** doe

Warning: **cpu_0.data_master/sopc_edge_detector_Interface_0.Avalon_Control_Slave**: **sopc_edge_detector_Interface_0.Avalon_Control_Slave** does not

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 7–14 in the "Avalon-MM Interface Blocks Walkthrough" for more detailed instructions.

Avalon-ST Interface

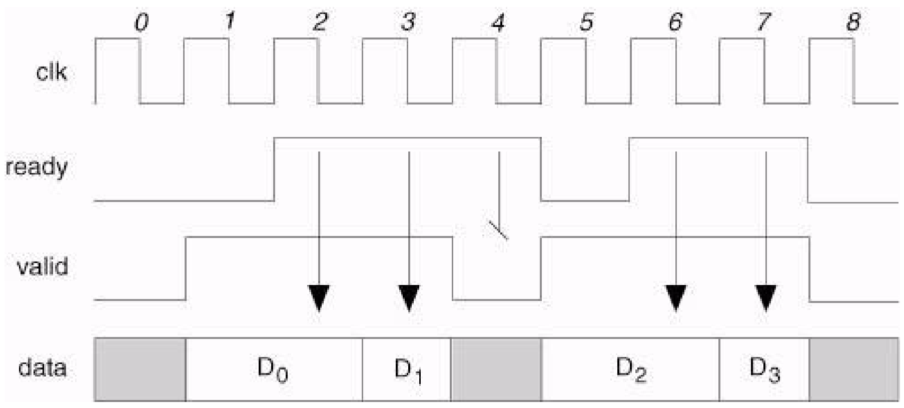
All DSP MegaCore functions in the DSP Builder MegaCore Functions library have interfaces that are compliant with the *Avalon Interface Specifications*. You can combine multiple MegaCore functions and easily because they use a common interface. This section summarizes the key features of the Avalon-ST interface.

The *Avalon Interface Specifications* define how to convey data between a source interface and a sink interface. The integrity of the data is indicated by a feed forward signal, `valid`. The specification also defines how the MegaCore functions may stall other blocks (backpressure) or regulate the rate at which data is provided by using a feedback sideband signal, `ready`.

You can configure the DSP Builder `Avalon-ST Source` and `Avalon-ST Sink` blocks with a ready latency of 0 or 1. The ready latency is the number of cycles that a source must wait after a sink has asserted ready so that a data transfer is possible. The source interface provides valid data at the earliest time possible, and it holds that data until ready is asserted by the sink. The `ready` signal notifies the source interface that the data has been sampled on that clock cycle.

For the *ready_latency* = 0 mode, Figure 7–17 shows the interaction that occurs between the source interface `valid` signal and the sink interface `ready` signal.

Figure 7–17. Avalon-ST Interface Timing for ready-latency=0

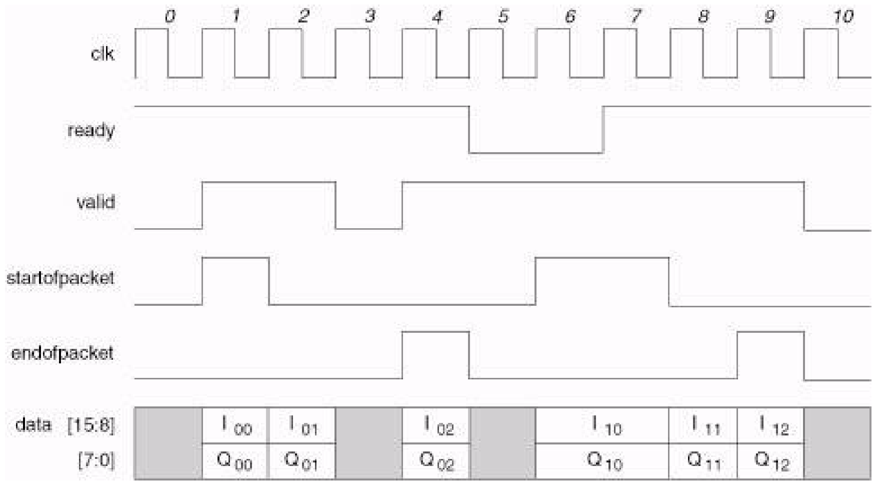


On cycle one, the source provides data and asserts `valid` even though the sink is not `ready`. The source waits until cycle two and the sink acknowledges that it has sampled the data by asserting `ready`. On cycle three, the source happens to provide data on the same cycle that the sink is ready to receive it and so the transfer occurs immediately. On the fourth cycle, the sink is ready but because the source has not provided any valid data, the data bus is not sampled.

A beat is defined as the transfer of one unit of data between a source and sink interface. This unit of data may consist of one or more symbols, so it can support modules that convey more than one piece of information on each valid cycle. This concept is useful because some modules have parallel input interfaces and other instances require serial input interfaces. For example, when conveying an in-phase and quadrature component on the same clock cycle. The choice depends on the algorithm, optimization technique, and throughput requirements.

Figure 7–18 gives an example of a data transfer where two symbols are conveyed on each beat - an in phase symbol I and a quadrature symbol Q. In this example, each symbol is eight bits wide.

Figure 7–18. Packetized Data Transfer



The *Avalon Interface Specifications* also describe several mechanisms to support the transfer of data associated with multiple channels. Altera recommends that this is achieved using packet based transfers where each packet has a deterministic format and each channel is allocated a specific field (time slot within a packet).

Packet transfers require two additional signals that mark the start and the end of the packet. The MegaCore functions are designed with internal counters that count the samples in a packet so they know which channel a particular sample is associated with and synchronize appropriately with the start and end of packet signals. In Figure 7–18, the in phase and quadrature components associated with three different channels are conveyed between two MegaCore functions.

Avalon-ST Packet Formats

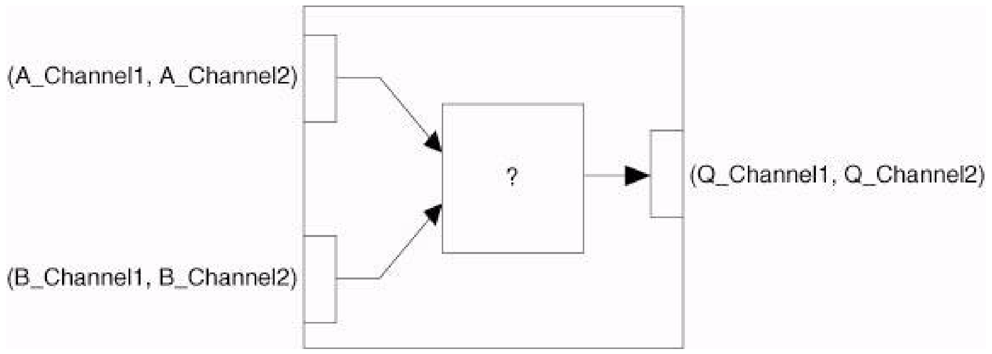
The data associated with each channel can be allocated a field within a packet. To describe the relationship between the input and the output interfaces of a MegaCore function, you must define the packets associated with each interface.

The basic format of a packet is described using two parameters: `SymbolsPerBeat`, and `PacketDescription`. The `SymbolsPerBeat` parameter defines the number of symbols that are presented in parallel on every valid cycle. The `PacketDescription` is a string description of the fields in the packet.

A basic `PacketDescription` is a comma-separated list of field names, where a field name starts with a letter and may include the characters `a-zA-Z0-9_`. Typical field names include `Channel1`, `Channel2`, and `Q`. Field names are case sensitive and white space is not permitted.

Figure 7–19 shows an example of a generic function that has two input interfaces and performs a transformation on the two input streams.

Figure 7–19. Generic Function



Avalon-ST Packet Format Converter

The packet format converter (PFC) is a flexible, multipurpose component that transforms packets that are received from one function into a packet format that is supported by another function.

The PFC takes packet data from one or more input interfaces, and provides field reassignment in time and space to one or more output packet interfaces. You can specify the input packet format and the desired output packet format. The appropriate control logic is automatically generated.

Each input interface has Avalon-ST `ready`, `valid`, `startofpacket`, `endofpacket`, `empty`, and `data` signals. Each output interface has an additional `error` bit, which is asserted to indicate a frame delineation error.

The PFC performs data mapping on a packet by packet basis, so that there is exactly one input packet on each input interface for each output packet on each output interface. The packet rate of the converter is limited by the interface with the longest packet. When the PFC has multiple output interfaces, the packets on each output interface are aligned so that the `startofpacket` signal is presented on the same clock cycle.

If each interface supports fixed-length packets, the multi-packet mapping option can be selected, and the PFC can map fields from multiple input packets to multiple output packets.

For a complete description of the Avalon-ST interface, refer to the *Avalon Interface Specifications*. For an example of a design that uses Avalon-ST interfaces and the Packet Format Converter blocks, refer to *AN442: Tool Flow for Design of Digital IF for Wireless Systems*.

8. Using Black Boxes for HDL 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*.

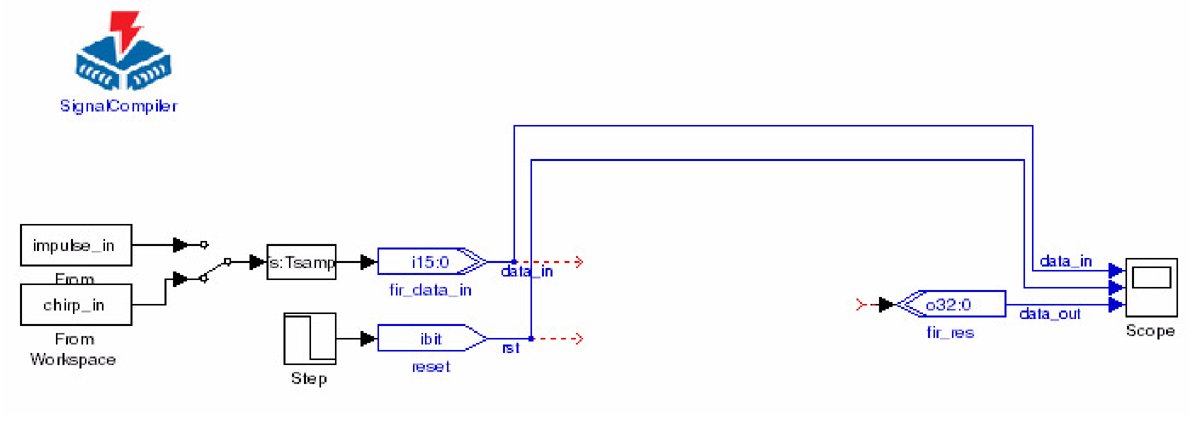
The following sections show an example of importing an existing HDL design written in VHDL into the DSP Builder environment using the `HDL Import` block.

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. On the File menu, click **Open** and select **empty_MyFilter.mdl**.

Figure 8–1.

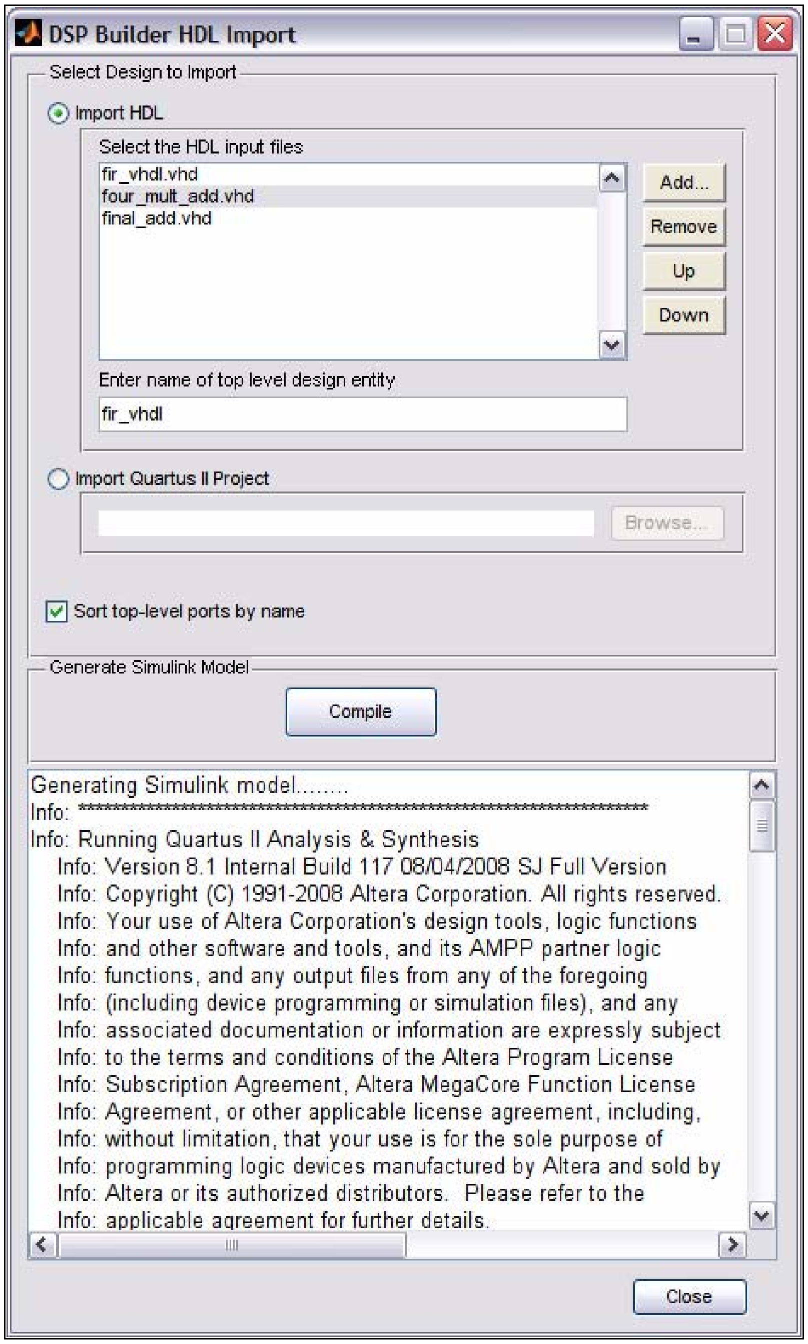


This design file has some of the peripheral blocks instantiated including the input/output ports and source blocks that provide appropriate stimulus for simulation. It is missing the main filter function which you can import as HDL.

3. Rename the file by clicking **Save As** on the 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 8–6 on page 8–7).
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 the files **fir_vhdl.vhd**, **four_mult_add.vhd**, and **final_add.vhd,** then click on **Open**.
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. Under **Generate Simulink Model**, click **Compile** to generate a Simulink simulation model for the imported HDL design.

Figure 8–2. HDL Import Dialog Box



Import HDL [✓]

Select the HDL input files:
fir_vhdl.vhd
four_mult_add.vhd
final_add.vhd

Enter name of top level design entity: fir_vhdl

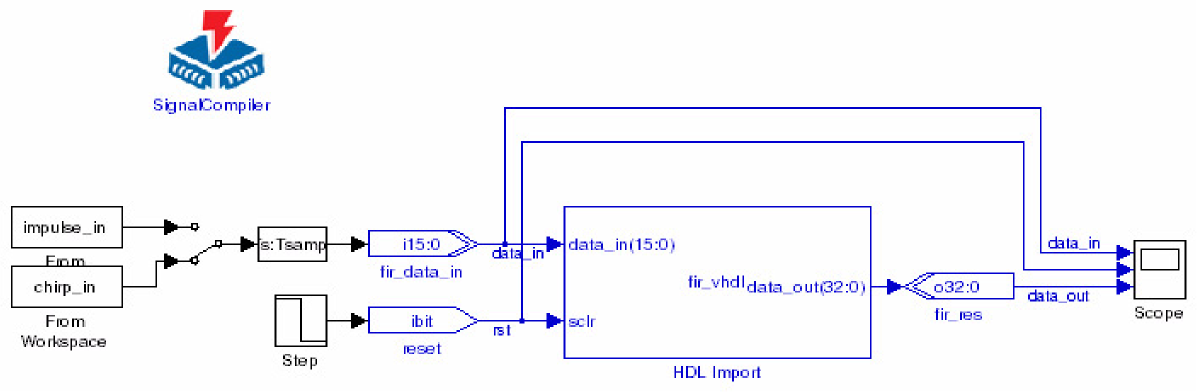
Sort top-level ports by name [✓]

```
Generating Simulink model........
Info: ************************************************************************
Info: Running Quartus II Analysis & Synthesis
    Info: Version 8.1 Internal Build 117 08/04/2008 SJ Full Version
    Info: Copyright (C) 1991-2008 Altera Corporation. All rights reserved.
    Info: Your use of Altera Corporation's design tools, logic functions
    Info: and other software and tools, and its AMPP partner logic
    Info: functions, and any output files from any of the foregoing
    Info: (including device programming or simulation files), and any
    Info: associated documentation or information are expressly subject
    Info: to the terms and conditions of the Altera Program License
    Info: Subscription Agreement, Altera MegaCore Function License
    Info: Agreement, or other applicable license agreement, including,
    Info: without limitation, that your use is for the sole purpose of
    Info: programming logic devices manufactured by Altera and sold by
    Info: Altera or its authorized distributors.  Please refer to the
    Info: applicable agreement for further details.
```

13. Progress messages are issued in the **HDL Import** dialog box ending with the message:

    `Quartus II Analysis & Synthesis was successful.`
14. The `HDL Import` block in the **MyFilter.mdl** model is updated to show the ports defined in the imported HDL.
15. Click **OK** to close the **HDL Import** dialog box.
16. Connect the input and output ports to the symbol, as shown in Figure 8–3. The code generated for the `HDL Import` block is automatically black boxed.

Figure 8–3. Completed Design



17. Click **Save** on the File menu to save the **MyFilter.mdl** file.

Simulate the HDL Import Model using Simulink

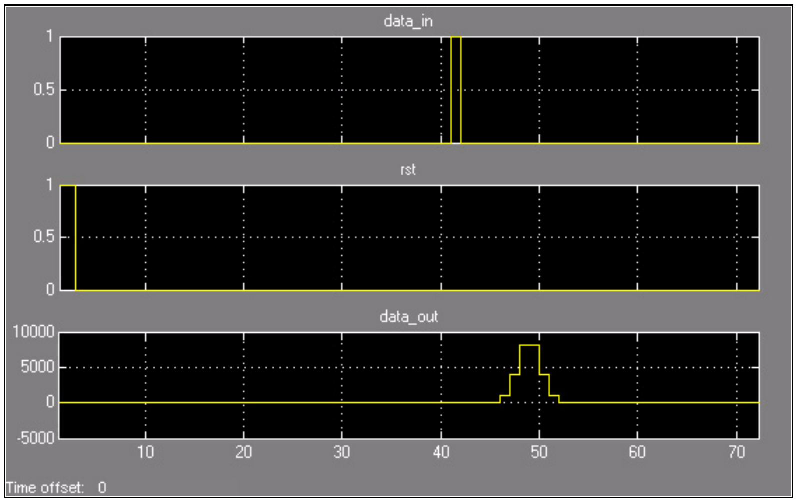
Perform the following steps to run simulation in Simulink:

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 8–4 on page 8–5.

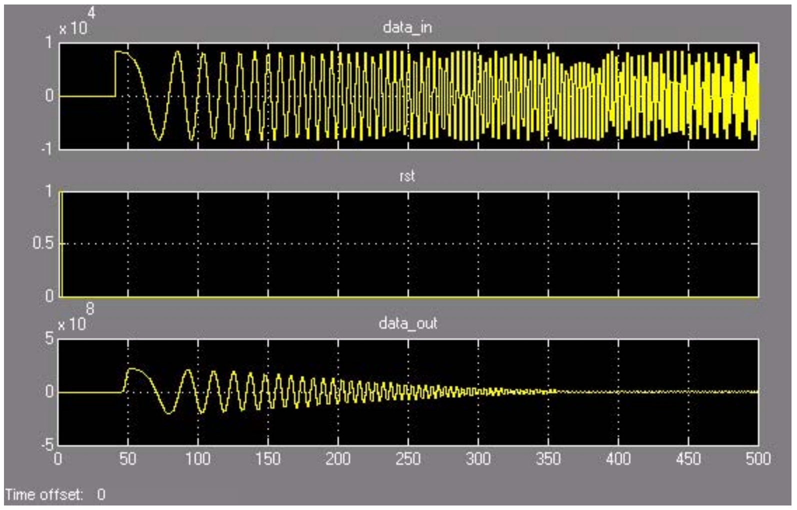
Figure 8–4. Simulink Simulation Results for the Impulse Stimulus



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 8–5.

Figure 8–5. 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 must 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

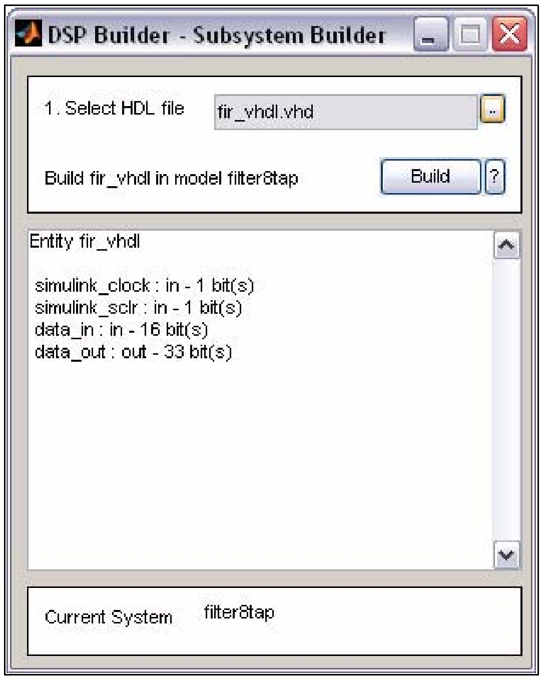
To create a black box system, perform the following steps:

1. In MATLAB, change the current directory to: <*DSP Builder install path*>**\DesignExamples\Tutorials\BlackBox\SubSystemBuilder**
2. Click **Open** on the 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 8–6).

Figure 8–6. Subsystem Builder Dialog Box


DSP Builder - Subsystem Builder

1. Select HDL file fir_vhdl.vhd

Build fir_vhdl in model filter8tap Build ?

```
Entity fir_vhdl

simulink_clock : in - 1 bit(s)
simulink_sclr : in - 1 bit(s)
data_in : in - 16 bit(s)
data_out : out - 33 bit(s)
```

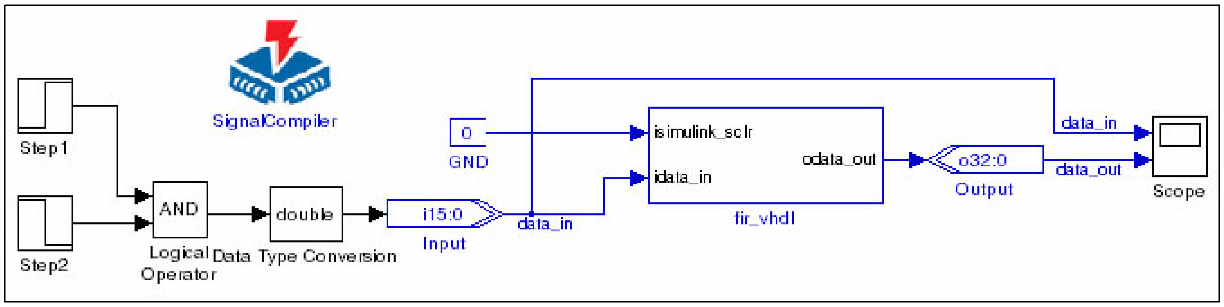
Current System filter8tap

7. In the dialog box, browse for the **fir_vhdl.vhd** file and click **Build**.

   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.

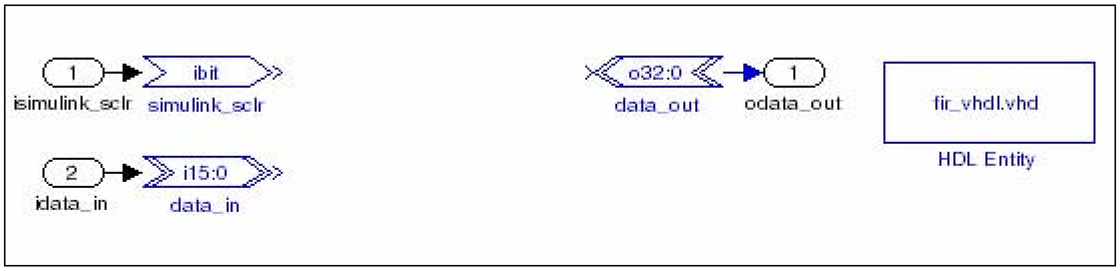
8. Connect the ports as shown in Figure 8–7.

Figure 8–7. filter8tap Design



9. Double-click on the `fir_vhdl` symbol. The **filter8tap/fir_vhdl** subsystem is opened (Figure 8–8).

Figure 8–8. Library: filter8tap/fir_vhdl Window



The 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 model 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:

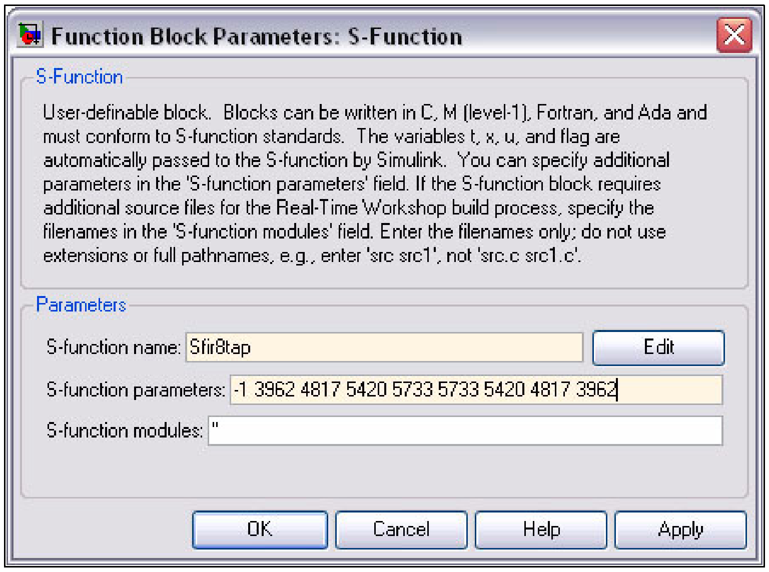
1. In the Simulink Library Browser, expand the **Simulink** folder.
2. From the **User-Defined Functions** library, drag and drop a `S-Function` block into the model window.
3. Double-click the `S-Function` block to display the **Function Block Parameters: S-Function** dialog box (Figure 8–9 on page 8–9).
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.

Figure 8–9. Block Parameters: S-Function Dialog Box



User-definable block. Blocks can be written in C, M (level-1), Fortran, and Ada and must conform to S-function standards. The variables t, x, u, and flag are automatically passed to the S-function by Simulink. You can specify additional parameters in the 'S-function parameters' field. If the S-function block requires additional source files for the Real-Time Workshop build process, specify the filenames in the 'S-function modules' field. Enter the filenames only; do not use extensions or full pathnames, e.g., enter 'src src1', not 'src.c src1.c'.

S-function name: Sfir8tap

S-function parameters: -1 3962 4817 5420 5733 5733 5420 4817 3962

S-function modules: ''

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 code 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
    ssSetNumModes(S, 0);       // pointers vector 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 8–1.

Table 8–1. 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.

1. At the MATLAB command prompt, type:

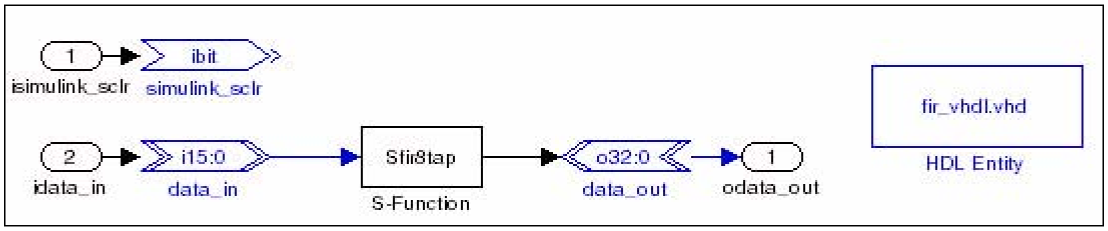
   ```
   mex Sfir8tap.CPP ↵
   ```

   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).

2. Close the editor window and click on **OK** to close the **Function Block Parameters** dialog box.

3. 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 8–10.

Figure 8–10. 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` to the global synchronous clear signal.

4. Click **Save** on the File menu to save the **filter8tap.mdl** file.

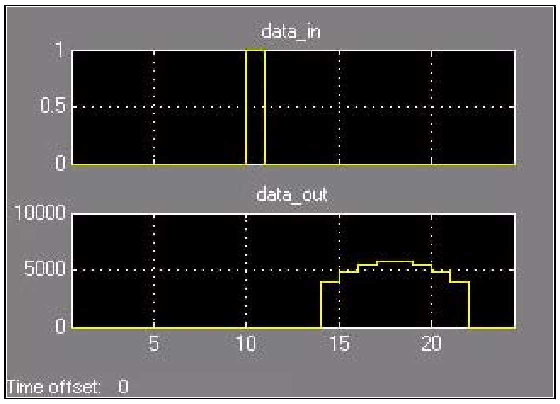
Simulate the Subsystem Builder Model

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 appear similar to Figure 8–11.

Figure 8–11. 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 is dependent on two other VHDL files. As currently specified, these two files are not examined by the Quartus II software or ModelSim, and compilation either fails or gives unexpected results. To resolve this, perform the following steps:

1. 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.

   ☞ Alternatively, you can create the directory **DSPBuilder_filter8tap_import** directly.

2. Copy the **extra_add.tcl** and **extra_add_msim.tcl** files from the original design directory to the **DSPBuilder_filter8tap_import** directory.

The **extra_add.tcl** file adds **final_add.vhd** and **four_mult_add.vhd** to the Quartus II project, while **extra_add_msim.tcl** compiles them in ModelSim 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:

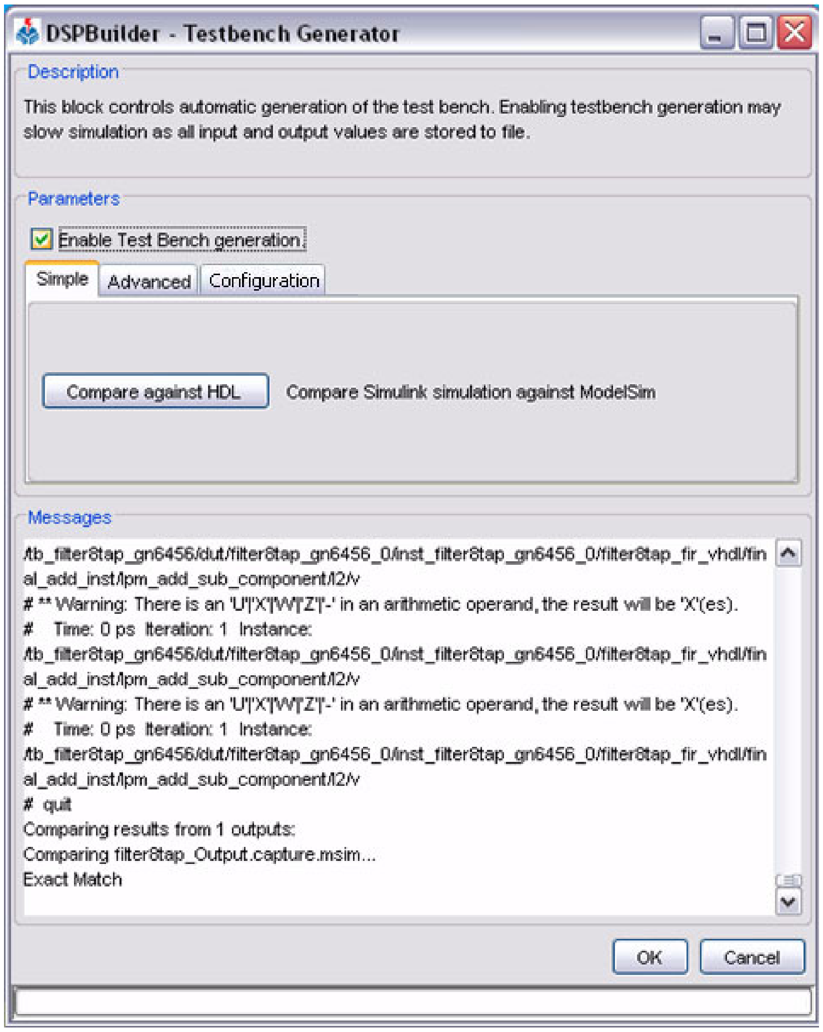
1. In the Simulink Library Browser, expand **AltLab** library under **Altera DSP Builder Blockset**.
2. Drag a `TestBench` block into your model.
3. Double-click on the `TestBench` block and click **Compare against HDL** (Figure 8–12 on page 8–13).

   When the comparison has completed successfully an `Exact Match` message is issued in the **TestBench Generator** dialog box.

☞ 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".

Figure 8–12. Testbench Generator Dialog Box for the filter8tap Design



This block controls automatic generation of the test bench. Enabling testbench generation may slow simulation as all input and output values are stored to file.

[✓] Enable Test Bench generation

Compare Simulink simulation against ModelSim

```
/tb_filter8tap_gn6456/dut/filter8tap_gn6456_0/inst_filter8tap_gn6456_0/filter8tap_fir_vhdl/fin
al_add_inst/lpm_add_sub_component/l2/v
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ps  Iteration: 1  Instance:
/tb_filter8tap_gn6456/dut/filter8tap_gn6456_0/inst_filter8tap_gn6456_0/filter8tap_fir_vhdl/fin
al_add_inst/lpm_add_sub_component/l2/v
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ps  Iteration: 1  Instance:
/tb_filter8tap_gn6456/dut/filter8tap_gn6456_0/inst_filter8tap_gn6456_0/filter8tap_fir_vhdl/fin
al_add_inst/lpm_add_sub_component/l2/v
# quit
Comparing results from 1 outputs:
Comparing filter8tap_Output.capture.msim...
Exact Match
```

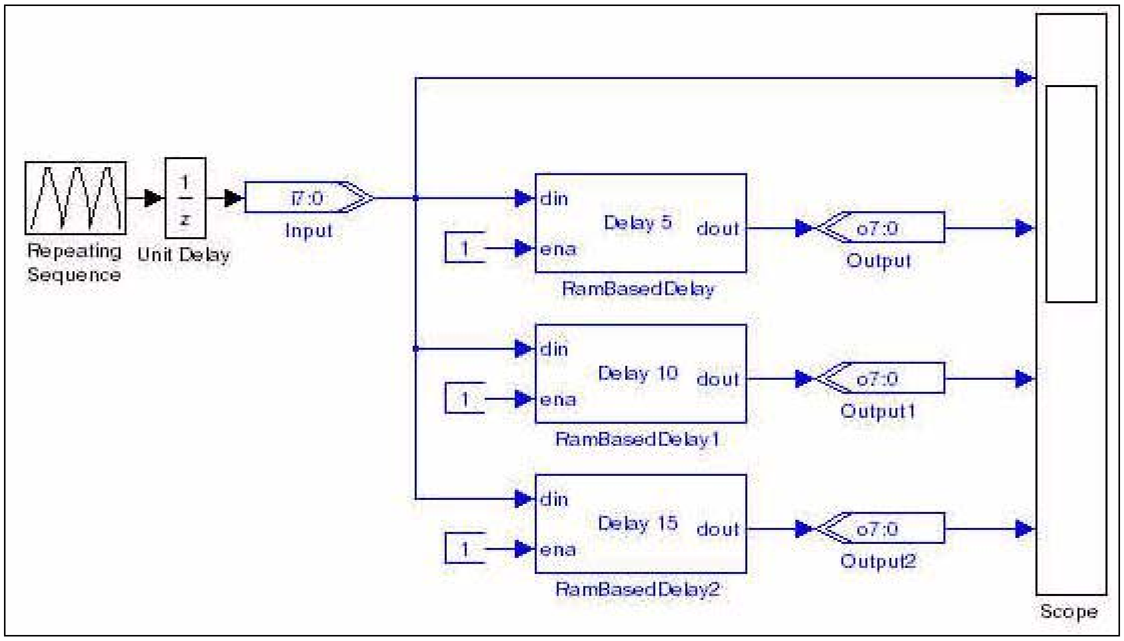
9. Using Custom Library Blocks

Introduction

A parameterizable custom library block is a Simulink subsystem in which the block functionality is described using DSP Builder primitives. This design flow also supports parameterizable hierarchical subsystem structures.

An example of a custom library block is provided at <*DSP Builder install path*>\ **DesignExamples\Tutorials\BuildingCustomLibrary\top.mdl**. (Figure 9–1).

Figure 9–1. 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, you perform the following steps:

1. Create a Library Model File
2. Build the HDL Subsystem Functionality
3. Define Parameters Using the Mask Editor
4. Link the Mask Parameters to the Block Parameters
5. Make the Library Block Read Only
6. Add the Library to the Simulink Library Browser

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. On the File menu in the Simulink Library Browser, point to **New** and click **Library** 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.

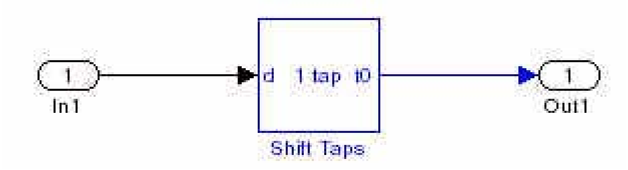
6. Click **Save** on the File menu and save the library file as **NewLib.mdl**.

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 model window. Insert the `Shift Taps` block between the input and output blocks (Figure 9–2).

Figure 9–2. Shift Taps Block



3. Double-click the `Shift Taps` block to open the **Block Parameters** dialog box (see Figure 9–3 and Figure 9–4 on page 9–3). Set the parameters shown in Table 9–1.

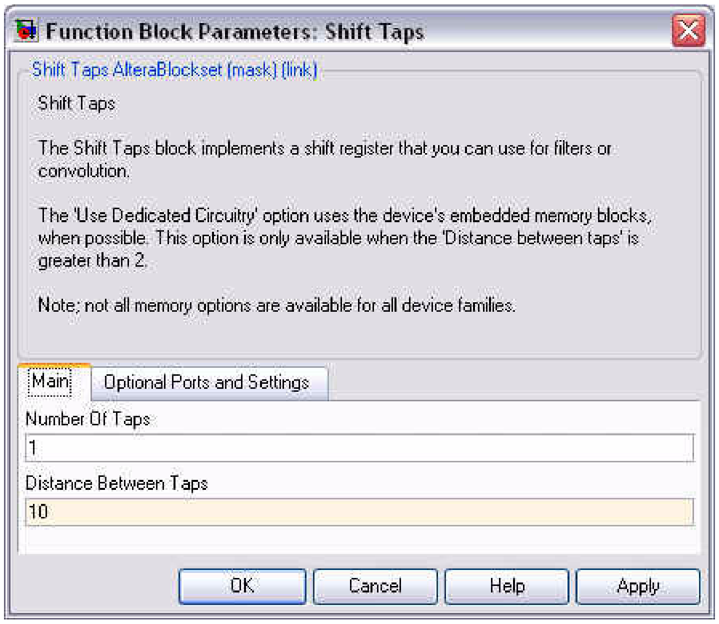
Table 9–1. Parameters for the Shift Taps Block

Parameter	**Value**
Main Tab	
Number Of Taps	1
Distance Between Taps	10
Optional Ports and Settings Tab	
Use Shift Out Port	Off

Table 9–1. Parameters for the Shift Taps Block

Parameter	Value
Use Enable port:	On
Use Dedicated Circuitry	On
Memory Block Type	Auto

Figure 9–3. Shift Taps Block Parameters


Function Block Parameters: Shift Taps
Shift Taps AlteraBlockset (mask) (link)
Main
Optional Ports and Settings
OK
Cancel
Help
Apply

Shift Taps

The Shift Taps block implements a shift register that you can use for filters or convolution.

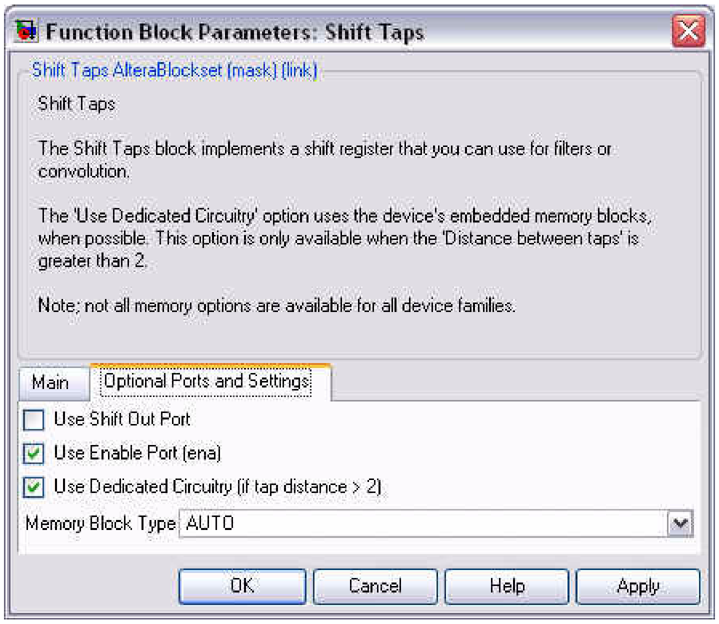
The 'Use Dedicated Circuitry' option uses the device's embedded memory blocks, when possible. This option is only available when the 'Distance between taps' is greater than 2.

Note; not all memory options are available for all device families.

Number Of Taps: 1

Distance Between Taps: 10

Figure 9–4. Shift Taps Block Optional Parameters


Function Block Parameters: Shift Taps
Shift Taps AlteraBlockset (mask) (link)
Main
Optional Ports and Settings
OK
Cancel
Help
Apply

Shift Taps

The Shift Taps block implements a shift register that you can use for filters or convolution.

The 'Use Dedicated Circuitry' option uses the device's embedded memory blocks, when possible. This option is only available when the 'Distance between taps' is greater than 2.

Note; not all memory options are available for all device families.

- [ ] Use Shift Out Port
- [✓] Use Enable Port (ena)
- [✓] Use Dedicated Circuitry (if tap distance > 2)

Memory Block Type: AUTO

4. Click **OK** to close the **Block Parameters** dialog box.
5. Add an `Input` block (`In2`) from the Simulink **Ports & Subsystems** library and connect it to the `ena` port on the `Shift Taps` block.
6. Rename the blocks as shown in Table 9–2.

Table 9–2. Renaming the Blocks

Old Name	New Name
In1	InDin
In2	InEna
Shift Taps	DRB
Out1	OutDout

7. Click **Save** on the File menu.

Figure 9–5 shows the completed **DelayFIFO** subsystem.

Figure 9–5. DelayFIFO Subsystem

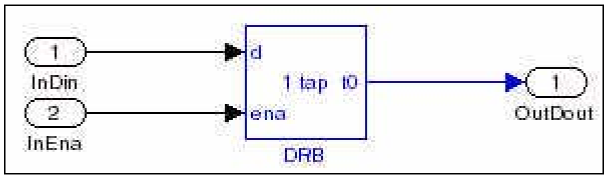
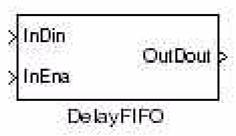


Figure 9–6 shows the **NewLib** library model which now shows the input and output ports defined in the **DelayFIFO** subsystem.

Figure 9–6. 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 click **Mask Subsystem** on the pop-up menu.
2. In the **Mask Editor** dialog box (Figure 9–7 on page 9–5, Figure 9–8 on page 9–6, and Figure 9–9 on page 9–6), set the options shown in Table 9–3 on page 9–5.

Table 9–3. Parameters for the Mask Editor

Parameter	**Value**
Icon Tab	
Frame	Visible
Transparency	Opaque
Rotation	Fixed
Units	Autoscale

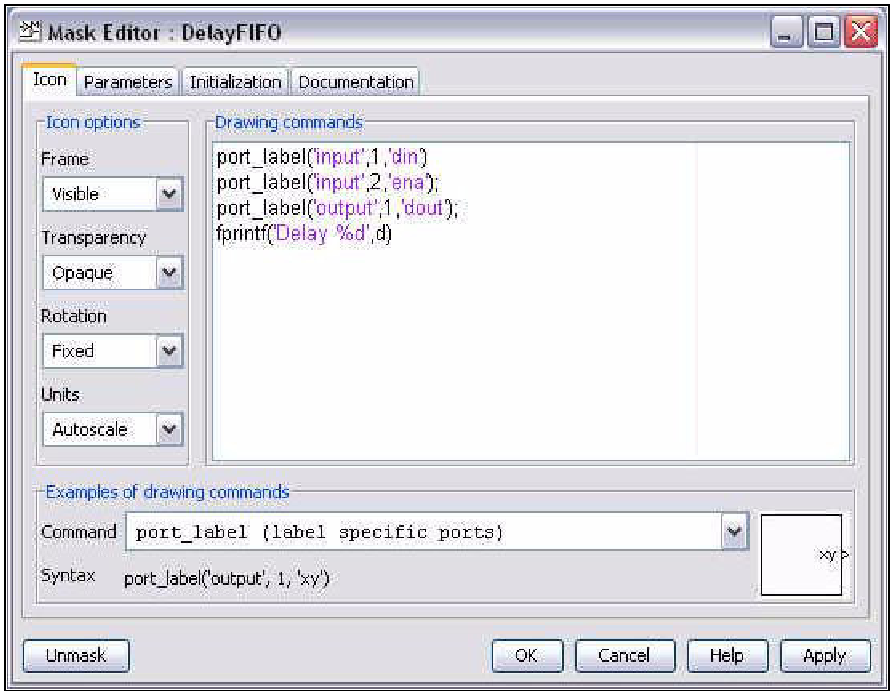
Drawing Commands	port_label('input',1,'din');
	port_label('input',2,'ena');
	port_label('output',1,'dout');
	fprintf('Delay %d',d)

Parameters Tab ***(Note 1)***	
Prompt	Delay
Variable	d
Documentation Tab	
Mask type	SubSystem AlteraBlockSet
Mask description	RAM-Based Delay Element Altera Corporation

Note to Table 9–3:

(1) To add a parameter in the **Parameters** tab, click the ⊞ button.

Figure 9–7. Mask Editor Icon Tab

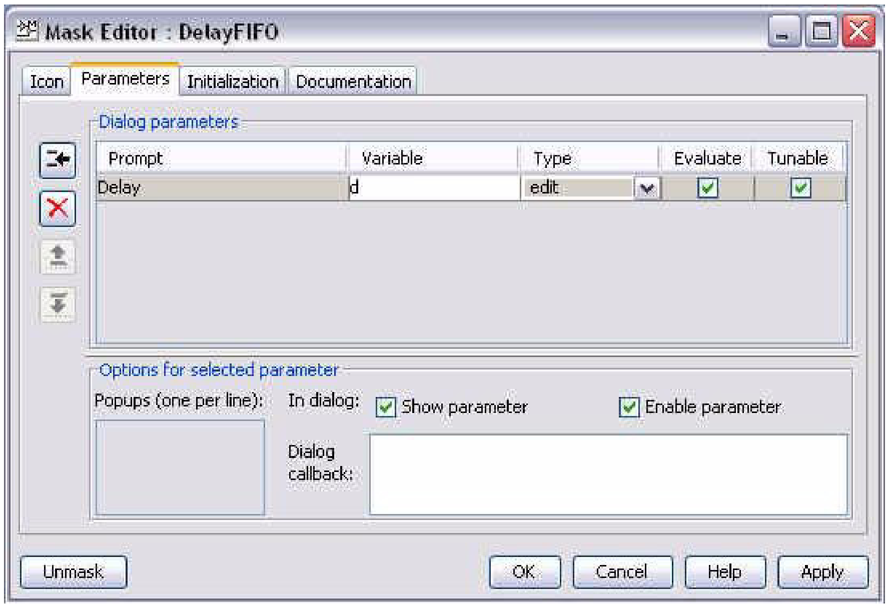


Frame: Visible
Transparency: Opaque
Rotation: Fixed
Units: Autoscale

```
port_label('input',1,'din')
port_label('input',2,'ena');
port_label('output',1,'dout');
fprintf('Delay %d',d)
```

Command: port_label (label specific ports)
Syntax: port_label('output', 1, 'xy')

Figure 9–8. Mask Editor Parameters Tab



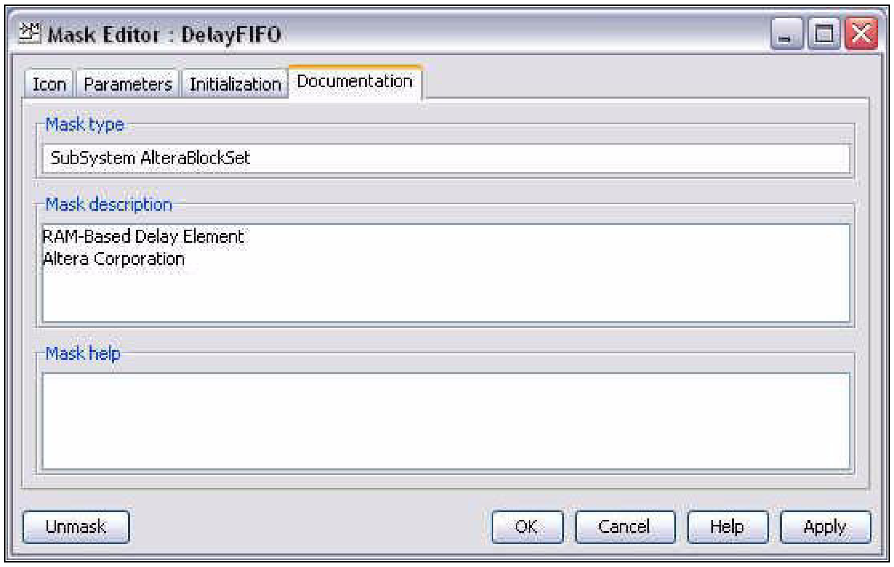
Prompt	Variable	Type	Evaluate	Tunable
Delay	d	edit	✓	✓

Popups (one per line):

In dialog: [✓] Show parameter [✓] Enable parameter

Dialog callback:

Figure 9–9. Documentation Tab



Mask type

SubSystem AlteraBlockSet

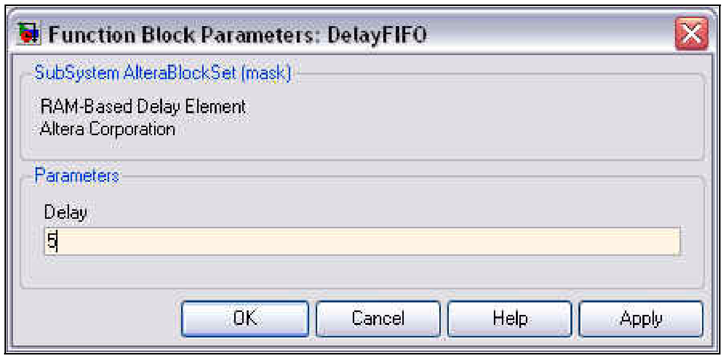
Mask description

RAM-Based Delay Element
Altera Corporation

Mask help

3. Click **OK** in the **Mask Editor** dialog box.
4. Double-click on the `DelayFIFO` block in your **NewLib** model to display the **Block Parameters** dialog box.
5. Specify a **Delay** of `5` (Figure 9–10 on page 9–7).

Figure 9–10. Delay FIFO Block Parameters



SubSystem AlteraBlockSet (mask)

RAM-Based Delay Element
Altera Corporation

Parameters

Delay
5

6. Click **OK** in the **Block Parameters** dialog box.
7. Click **Save** on the File menu to save your library model.

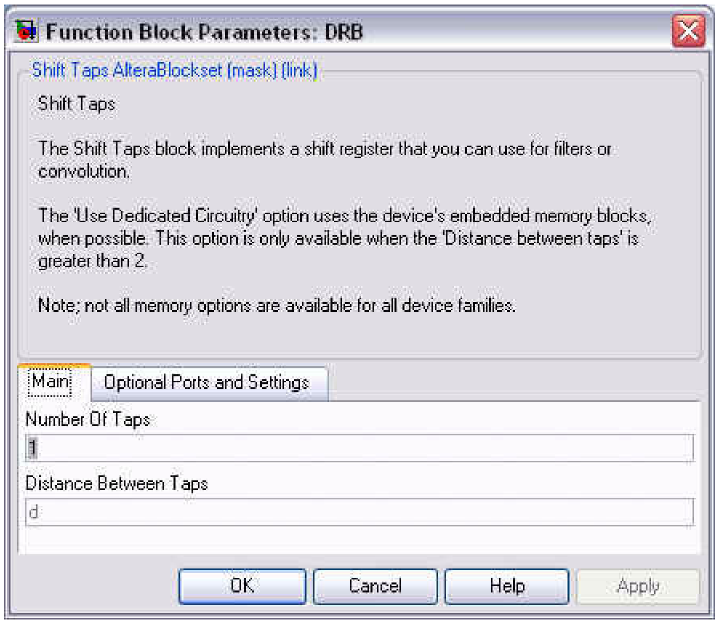
For more information on the Mask Editor, refer to the MATLAB Help.

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 DRB 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 **Block Parameters** dialog box. (Figure 9–11)

Figure 9–11. Shift Taps Block Parameters



Shift Taps AlteraBlockset (mask) (link)

Shift Taps

The Shift Taps block implements a shift register that you can use for filters or convolution.

The 'Use Dedicated Circuitry' option uses the device's embedded memory blocks, when possible. This option is only available when the 'Distance between taps' is greater than 2.

Note; not all memory options are available for all device families.

Number Of Taps
1

Distance Between Taps
d

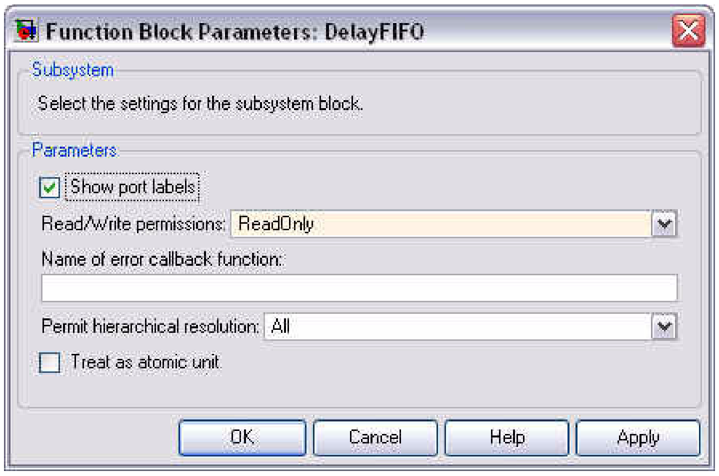
3. Click **OK** to close the Shift Taps Block Parameters dialog box.
4. Close the model window.

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 click **SubSystem Parameters** on the pop-up menu to display the **Block Parameters** dialog box.
2. In the **Read/Write permissions** list, select **ReadOnly** as shown in Figure 9–12.

Figure 9–12. Delay FIFO Block Parameters


Function Block Parameters: DelayFIFO

Subsystem

Select the settings for the subsystem block.

Parameters

☑ Show port labels

Read/Write permissions: ReadOnly

Name of error callback function:

Permit hierarchical resolution: All

☐ Treat as atomic unit

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, open the model, click **Unlock Library** on the File menu and 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. Click **Save** on the 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. On the File menu in MATLAB, point to **New** and click **M-File** to open a new editor window.

2. Enter the following text in the editor window:

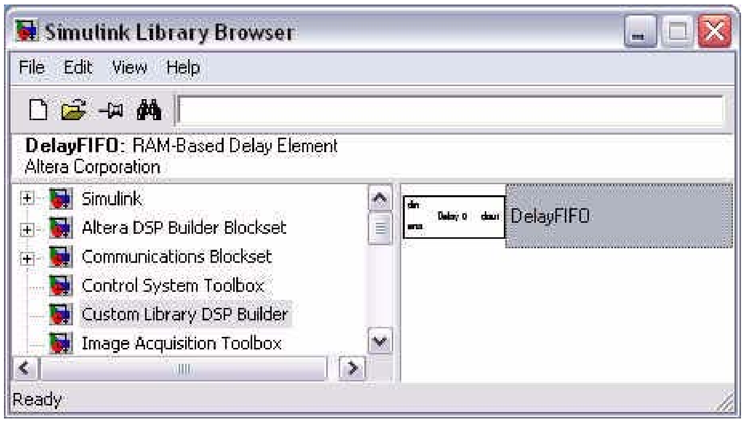
```
function blkStruct = slblocks

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 should be available as shown Figure 9–13.

Figure 9–13. Custom Library in the Simulink Library Browser



DelayFIFO: RAM-Based Delay Element
Altera Corporation

- Simulink
- Altera DSP Builder Blockset
- Communications Blockset
- Control System Toolbox
- Custom Library DSP Builder
- Image Acquisition Toolbox

DelayFIFO

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**.

Synchronizing a Custom Library

A custom library can contain MegaCore functions, HDL import, or state machine editor blocks. If your design is copied or moved, it may be necessary to synchronize the model containing these blocks by using the following command:

```
alt_dspbuilder_refresh_user_library_blocks
```

☞ This command is automatically called when you use either of the commands:

```
alt_dspbuilder_refresh_hdlimport
```

or

```
alt_dspbuilder_refresh_megacore
```

10. 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
- Cyclone III EP3C25 Starter board
- Cyclone III EP3C120 board
- Stratix EP1S25 board
- Stratix EP1S80 board
- Stratix II EP2S60 board
- Stratix II EP2S180 board
- Stratix II EP2SGX90 PCI Express board
- Stratix III EP3SL150 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**

There is a single XML file that describes each separate board component named *<component_name>*.**component**. This file 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, Fixed Size Bus, and Selectable Single Bit.

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`)

Fixed-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 corresponding 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`. You can omit this attribute for the Selectable Single Bit Type, because it is 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**

The board description file is divided into Header and Board Description 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>
    </configuration>
</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
```

11. Using the State Machine Library

Introduction

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

Separate procedures are provided for "Using the State Machine Table Block" on page 11–2 and "Using the State Machine Editor Block" on page 11–8.

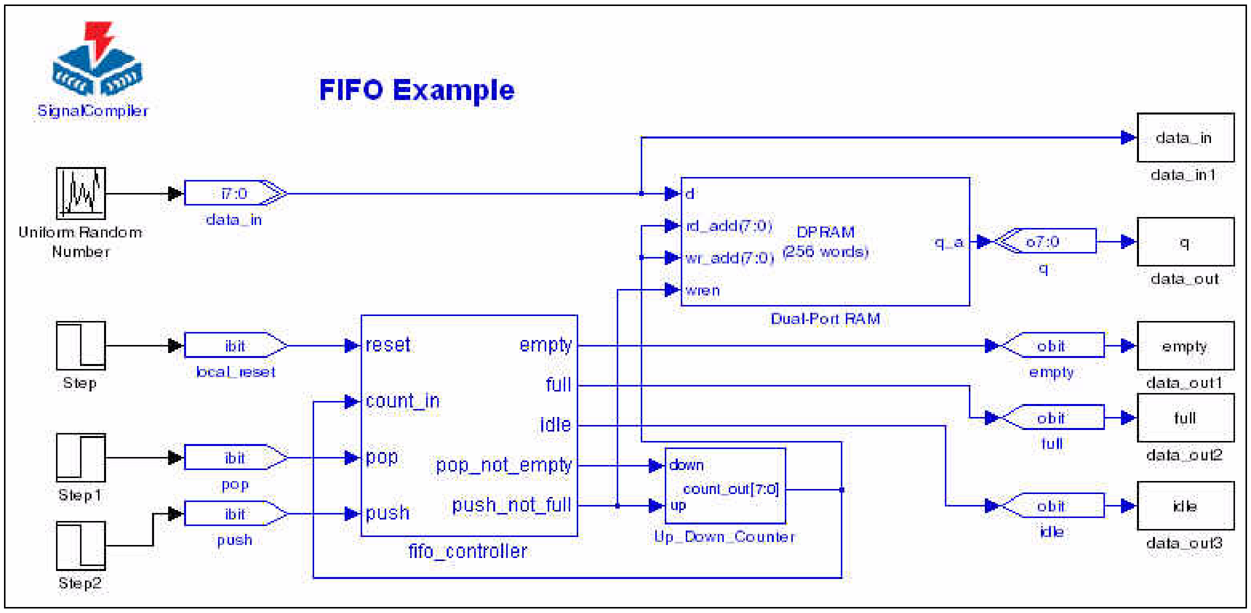
☞ The `State Machine Table` block is deprecated and should not be used in new designs.

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 the *<DSP Builder install path>*\ **DesignExamples\Tutorials\StateMachine\StateMachineTable** directory.

Figure 11–1 shows the top-level schematic for the FIFO design example.

Figure 11–1. 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:

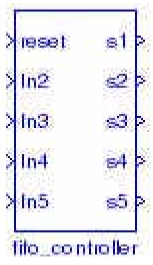
- When the `push` input is asserted and the address counter is less than 250, the address counter is incremented and a byte of data is written to memory.
- When the `pop` input is asserted and the address counter is greater than 0, the address counter is decremented and a byte of data is read from memory.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

Using the State Machine Table Block

The following steps show how you can use the `State Machine Table` block to create the FIFO controller in the example design:

1. Add a `State Machine Table` block to your Simulink design and assign it a new name. Figure 11–2 shows the default `State Machine Table` block. In this example, the block is named `fifo_controller`.

Figure 11–2. fifo_controller State Machine Table 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.

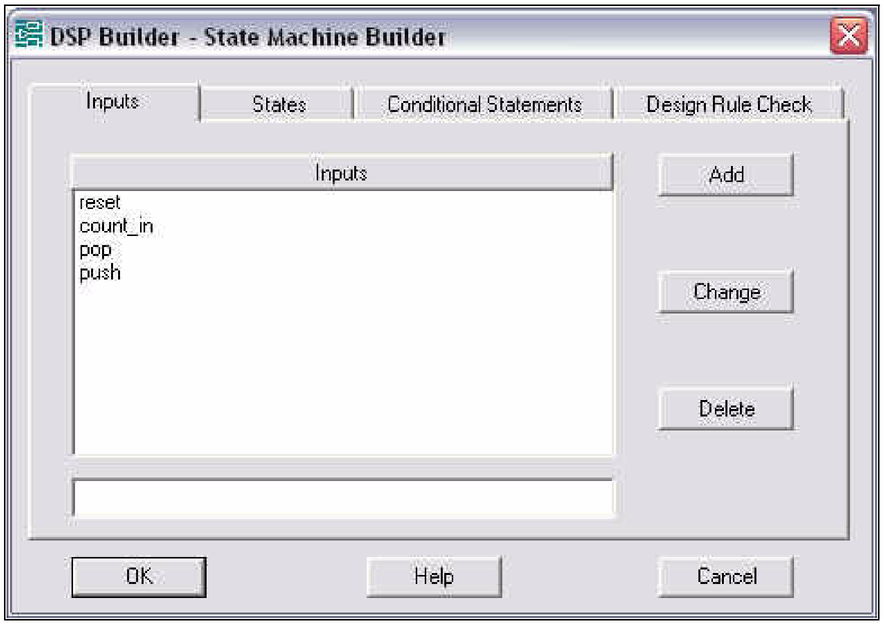
   The **State Machine Builder** dialog box appears with the **Inputs** tab selected. The **Inputs** tab displays the input names defined for your state machine and provides an interface to allow you to add, and delete input names.

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 11–3 on page 11–3 shows the **Inputs** tab after the inputs have been defined for the FIFO design example.

Figure 11–3. State Machine Builder Inputs Tab



reset
count_in
pop
push

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.

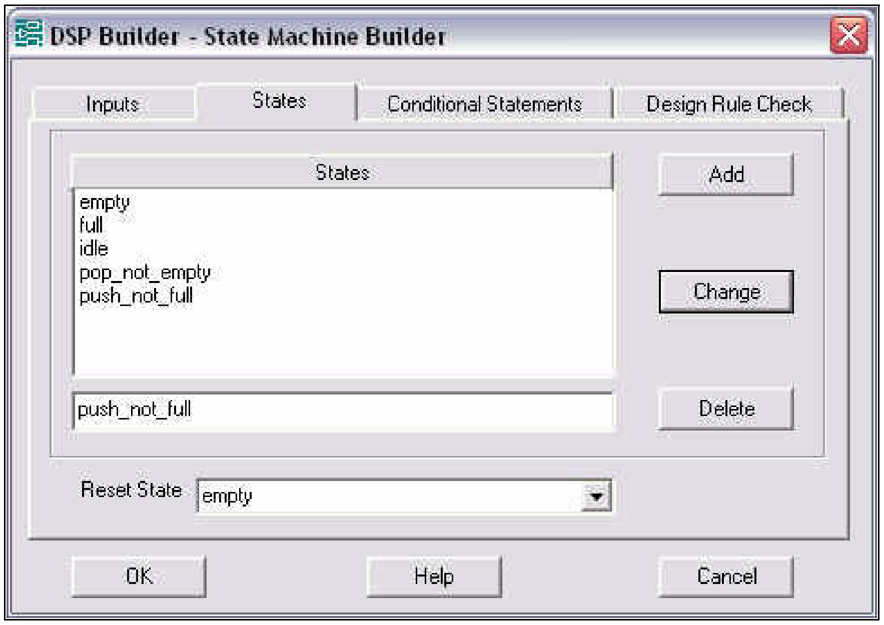
   You must define at least two states for the state machine. You cannot delete or change the name of a state while it is selected as the reset state.

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 11–4 shows the State Machine Builder **States** tab after the states have been edited for the FIFO design example.

Figure 11–4. State Machine Builder States Tab



States:
empty
full
idle
pop_not_empty
push_not_full

push_not_full

Reset State: empty

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 11–1.

Table 11–1. FIFO Controller Conditional Statements

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

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

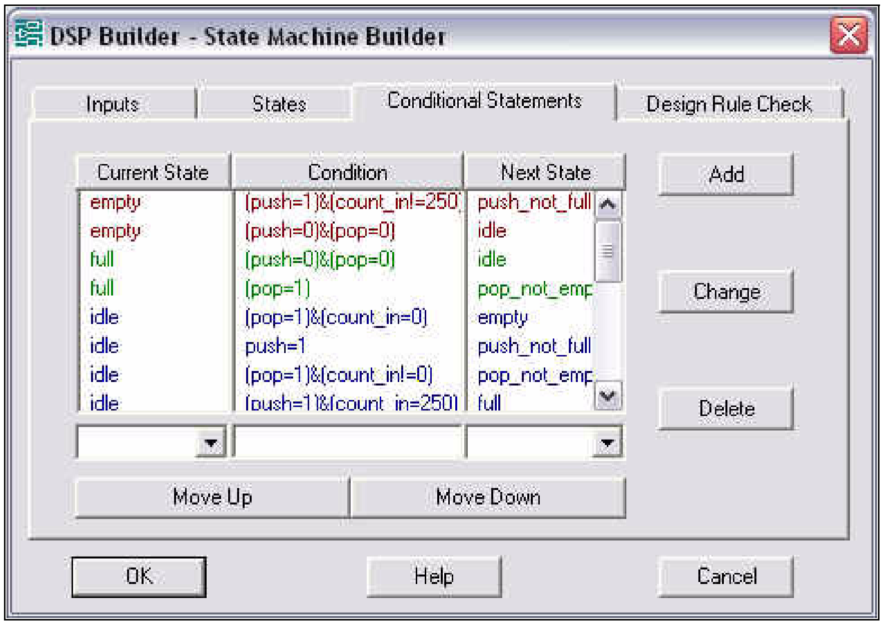
☞ There must be at least one conditional statement defined in the **Conditional Statements** tab.

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 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 11–5 on page 11–5 shows the **Conditional Statements** tab, after defining the conditional statements for the FIFO controller.

Figure 11–5. State Machine Builder Conditional Statements Tab



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_emp
idle	(pop=1)&(count_in=0)	empty
idle	push=1	push_not_full
idle	(pop=1)&(count_in!=0)	pop_not_emp
idle	(push=1)&(count_in=250)	full

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 11–2 shows the conditional operators that can be used to define a conditional expression.

Table 11–2. 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
&	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 11–3 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 11–3. 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

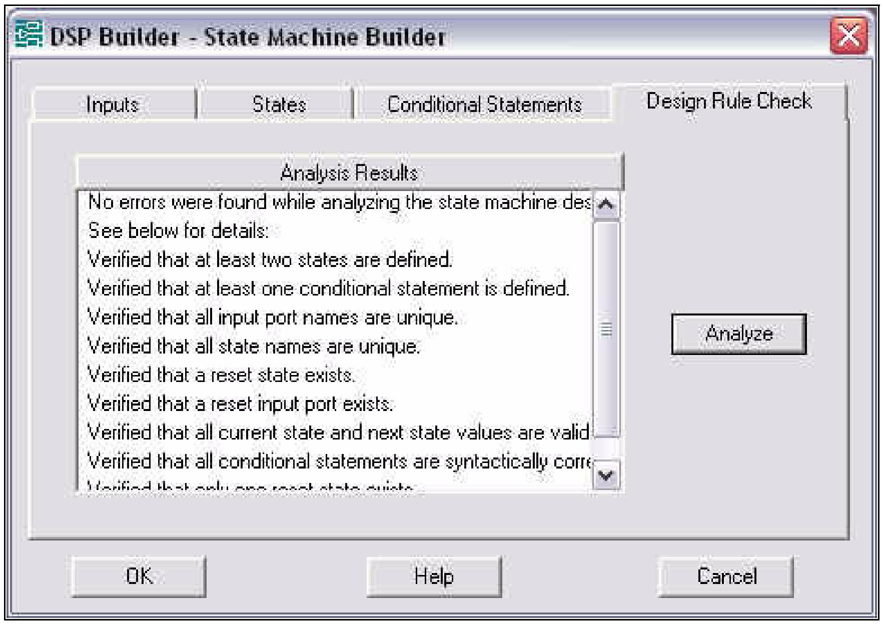
7. Use the **Move Up** and **Move Down** buttons to change the order of the conditional statements, as shown in Table 11–4.

Table 11–4. 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

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 11–6 shows the **Design Rule Check** tab after clicking Analyze.

Figure 11–6. State Machine Builder Design Rule Check Tab



```
No errors were found while analyzing the state machine des
See below for details:
Verified that at least two states are defined.
Verified that at least one conditional statement is defined.
Verified that all input port names are unique.
Verified that all state names are unique.
Verified that a reset state exists.
Verified that a reset input port exists.
Verified that all current state and next state values are valid
Verified that all conditional statements are syntactically corre
```

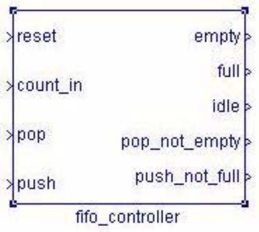
9. To save the changes made to your state machine, click **OK**.

   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 11–7 shows the updated `fifo_controller` block.

Figure 11–7. fifo_controller Block After Closing the State Machine Table



Using the State Machine Editor Block

The following steps show how you can use the `State Machine Editor` block to create the FIFO controller in the example design:

1. Add a `State Machine Editor` block to your Simulink design and assign it a new name. Figure 11–2 shows the default `State Machine Editor` block. In this example, the block is named `fifo_controller`.

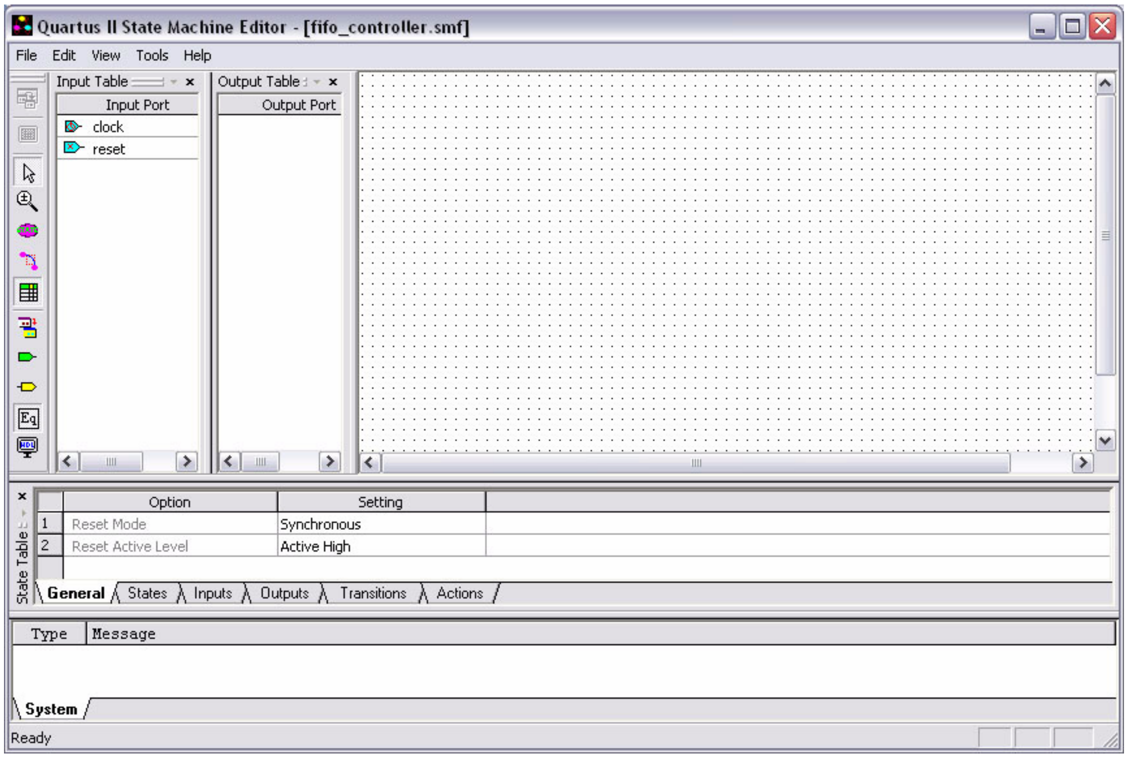
Figure 11–8. fifo_controller State Machine Editor Block

Graphical State Machine Editor
fifo_controller

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

2. Double-click the `fifo_controller` block to open the State Machine Editor in the Quartus II software (Figure 11–9).

Figure 11–9. Quartus II State Machine Editor Window


Quartus II State Machine Editor - [fifo_controller.smf]
File Edit View Tools Help
Input Table
Input Port
clock
reset
Output Table
Output Port
State Table
General States Inputs Outputs Transitions Actions
Type Message
System
Ready

	Option	Setting
1	Reset Mode	Synchronous
2	Reset Active Level	Active High

3. On the Tools menu in the Quartus II State Machine Editor, point to **State Machine Wizard** and click **Create a new state machine design**.

4. The first page of the wizard allows you to choose the reset mode, whether the reset is active-high or active-low, and whether the outputs are registered. Accept the default values (synchronous, active-high, registered outputs) and click **Next** to display the **Transitions** page of the wizard.
5. Delete the default state names (`state1`, `state2`, `state3`) and type the following new state names:
   - `empty`
   - `full`
   - `idle`
   - `pop_not_empty`
   - `push_not_full`
6. Delete the default input port names (`input1`, `input2`) and type the following new input port names:
   - `count_in[7:0]`
   - `pop`
   - `push`

   ☞ Do not change the `clock` and `reset` port names. The `count_in` port must be defined as an 8-bit vector to allow count values up to 250.

7. Edit the state transitions by entering the statements shown in Table 11–1.

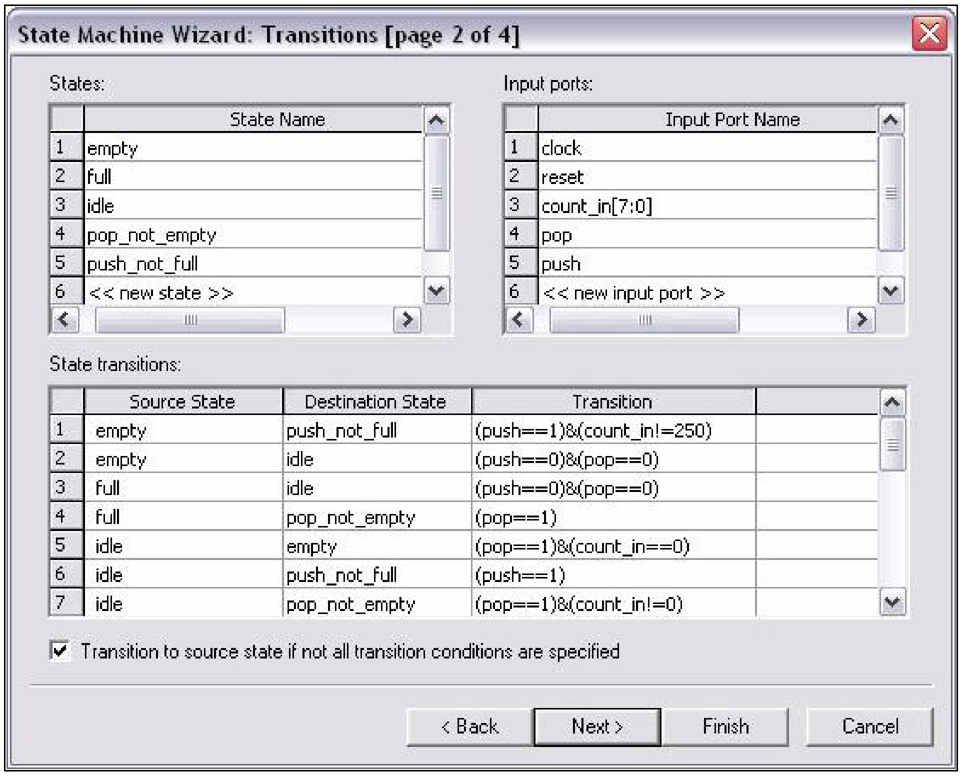
Table 11–5. FIFO Controller Transitions

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

The transitions are validated on entry and must conform with Verilog HDL syntax.

Figure 11–10 shows the **Transitions** page after the states, inputs, and transitions have been defined.

Figure 11–10. State Machine Editor Wizard Transitions Page



States:

	State Name
1	empty
2	full
3	idle
4	pop_not_empty
5	push_not_full
6	<< new state >>

Input ports:

	Input Port Name
1	clock
2	reset
3	count_in[7:0]
4	pop
5	push
6	<< new input port >>

State transitions:

	Source State	Destination State	Transition	
1	empty	push_not_full	(push==1)&(count_in!=250)	
2	empty	idle	(push==0)&(pop==0)	
3	full	idle	(push==0)&(pop==0)	
4	full	pop_not_empty	(pop==1)	
5	idle	empty	(pop==1)&(count_in==0)	
6	idle	push_not_full	(push==1)	
7	idle	pop_not_empty	(pop==1)&(count_in!=0)	

[✓] Transition to source state if not all transition conditions are specified

8. Click **Next** to display the **Actions** page. Delete the default output port name (`output1`) and enter the following new output port names:
   - `out_empty`
   - `out_full`
   - `out_idle`
   - `out_pop_not_empty`
   - `out_push_not_full`

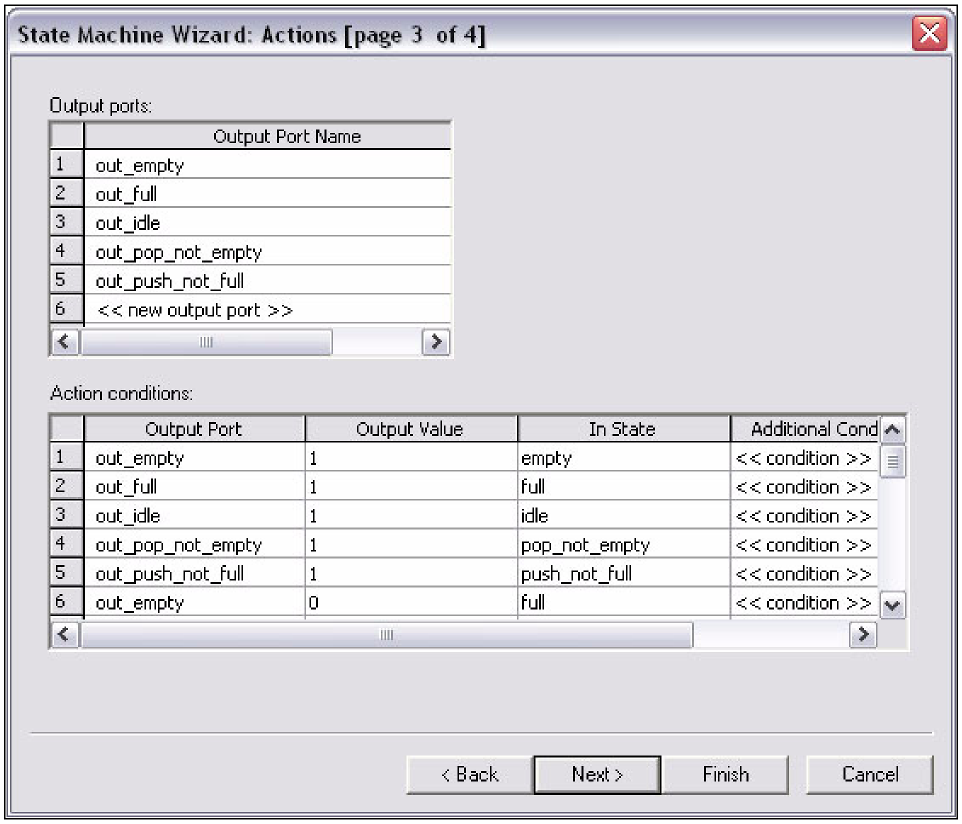
9. Specify the output logic for each output port by specifying the action conditions to set each output port to 1 when the state is true and 0 for all other states as shown in Table 11–6.

Table 11–6. FIFO Controller Output Actions

Output Port	Output Value	In State
out_empty	1	empty
out_full	1	full
out_idle	1	idle
out_pop_not_empty	1	pop_not_empty
out_push_not_full	1	push_not_full
out_empty	0	full
out_empty	0	idle
out_empty	0	pop_not_empty
out_empty	0	push_not_full
out_full	0	empty
out_full	0	idle
out_full	0	pop_not_empty
out_full	0	push_not_full
out_idle	0	empty
out_idle	0	full
out_idle	0	pop_not_empty
out_idle	0	push_not_full
out_pop_not_empty	0	empty
out_pop_not_empty	0	full
out_pop_not_empty	0	idle
out_pop_not_empty	0	push_not_full
out_push_not_full	0	empty
out_push_not_full	0	full
out_push_not_full	0	idle
out_push_not_full	0	pop_not_empty

Figure 11–11 shows the **Actions** page after the output ports, and action conditions have been defined.

Figure 11–11. State Machine Editor Wizard Actions Page



Output ports:

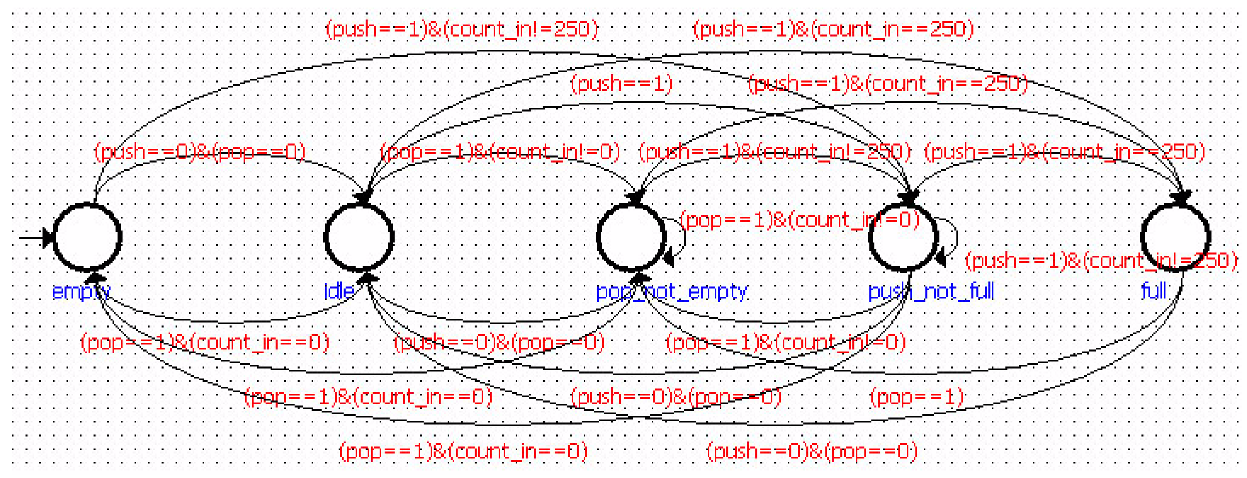
	Output Port Name
1	out_empty
2	out_full
3	out_idle
4	out_pop_not_empty
5	out_push_not_full
6	<< new output port >>

Action conditions:

	Output Port	Output Value	In State	Additional Cond
1	out_empty	1	empty	<< condition >>
2	out_full	1	full	<< condition >>
3	out_idle	1	idle	<< condition >>
4	out_pop_not_empty	1	pop_not_empty	<< condition >>
5	out_push_not_full	1	push_not_full	<< condition >>
6	out_empty	0	full	<< condition >>

10. Click **Next** to display the **Summary** page. Check that the summary lists the five states (`empty`, `full`, `idle`, `pop_not_empty`, and `push_not_full`), the five input ports (`clock`, `count_in[7:0]`, `pop`, `push`, and `reset`), and the five output ports (`out_empty`, `out_full`, `out_idle`, `out_pop_not_full`, and `out_push_not_full`).
11. Click **Finish** to complete the state machine definition. The state machine is displayed graphically in the State Editor window and should look similar to Figure 11–12 on page 11–13.

Figure 11–12. Graphical fifo_controller State Machine Diagram



The first state that you entered in the wizard is marked as the default state. This should be the empty state and is the state to which the state machine transitions when the reset input is asserted.

12. On the Tools menu in the Quartus II State Machine Editor, click **Generate HDL File** to display the **Generate HDL File** dialog box. Select **VHDL** and click **OK** to confirm your choice. Click **Yes** to save the **fifo_controller.smf** file and check that there are no FSM verification errors.

    There should be five warning messages stating that FSM verification is skipped in each state. These messages can be ignored.

    If there are any errors, you can edit the state machine using the **Properties** dialog boxes that can be displayed from the right button pop-up menu when a state or transition is selected. You can also edit the state machine in table format by clicking the tabs at the bottom of the State Machine Editor window.

For information about editing state machine properties and drawing a graphical state machine, refer to the *About the State Machine Editor* topic in the Quartus II online help that is available from the **Help** menu in the editor window.

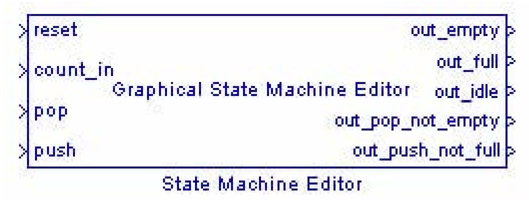
13. On the File menu in the Quartus II State Machine Editor, click **Exit**.

    The `fifo_controller` block on your model is updated with the input and output ports defined in the state machine.

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

Figure 11–7 shows the updated `fifo_controller` block for the FIFO design example.

Figure 11–13. fifo_controller Block After Closing the State Machine Editor



12. Troubleshooting

Troubleshooting Issues

This chapter contains information about resolving the issues and error conditions listed in Table 12–1.

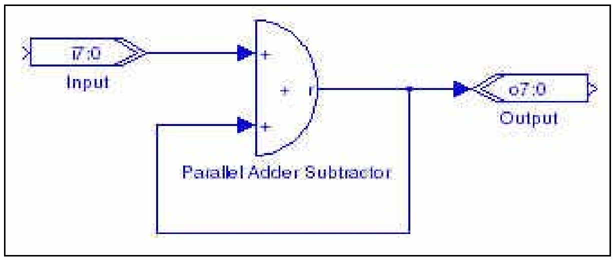
Table 12–1. Troubleshooting Issues

Loop Detected While Propagating Bit Widths
The MegaCore Blocks Folder Does Not Appear in Simulink
The Synthesis Flow Does Not Run Properly
DSP Development Board Troubleshooting
Signal Compiler is Unable to Checkout a Valid License
SignalTap II Analysis Appears to be Hung
Error if Output Block Connected to an Altera Synthesis Block
DSP Builder Start Up Dependencies
Warning if Input/Output Blocks Conflict with clock or aclr Ports
Wiring the Asynchronous Clear Signal
Simulation Mismatch After Changing Signals or Parameters
Error Issued when a Design Includes Pre-v7.1 Blocks
Creating an Input Terminator for Debugging a Design
A Specified Path Cannot be Found or a File Name is Too Long
Incorrect Interpretation of Signed Bit in Output from MegaCores
Simulation Mismatch For FIR Compiler MegaCore Function
Unexpected Exception Error when Generating Blocks
VHDL Entity Names Change if a Model is Modified
Algebraic Loop Causes Simulation to Fail

Loop Detected While Propagating Bit Widths

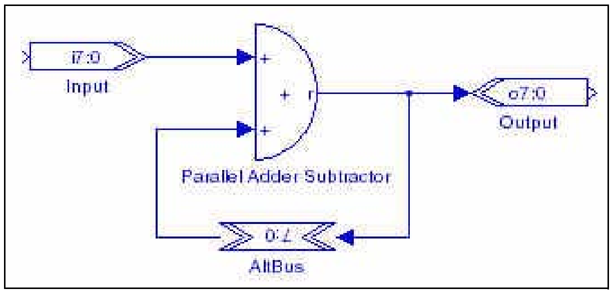
You may get an error if you have a feedback loop in your design and the feedback loop's bit width is not defined explicitly. Figure 12–1 shows this error.

Figure 12–1. 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 12–2.

Figure 12–2. 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 ↵
```

The 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 Web 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\<*version*>\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 the **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 SRAM object file (.**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 Checkout 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, refer to the *DSP Builder Installation and Licensing* manual.

Verifying That Your DSP Builder Licensing Functions Properly

Type the following command in the MATLAB Command Window:

```
dos('lmutil lmdiag C4D5_512A') ↵
```

where C4D5_512 is the DSP Builder feature ID.

This command outputs the status of the DSP Builder license.

For example, if you are using a node locked license:

```
lmutil - Copyright (C) 1989-2006 Macrovision Europe Ltd. and/or
Macrovision Corporation. All Rights Reserved.

FLEXnet diagnostics on Mon 8/11/2008 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.

Alternatively, if you are using a floating license:

```
>> dos('lmutil lmdiag  C4D5_512A')

lmutil - Copyright (c) 1989-2006 Macrovision Europe Ltd. and/or
Macrovision Corporation. All Rights Reserved.

FLEXnet diagnostics on Mon 8/11/2008 10:49

-----------------------------------------------------

License file: node@lic_server

-----------------------------------------------------

"C4D5_512A" v2030.12, vendor: alterad

License server: lic_server

floating license  expires: 31-dec-2030

This license can be checked out

-----------------------------------------------------
```

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 "The Synthesis Flow Does Not Run Properly" on page 12–2.

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 in the following format:

```
lmutil - Copyright (c) 1989-2006 Macrovision Europe Ltd. and/or
Macrovision Corporation. All Rights Reserved.

Flexible License Manager status on Mon 8/11/2008 15:36

License server status:

[Detecting lmgrd processes...]
License server status: node@lic_server
    License file(s) on shama: /usr/licenses/quartus/license.dat:

lic_server: license server UP (MASTER) v10.8

Vendor daemon status (on lic_server):

    alterad: UP v9.2

Feature usage info:

Users of C4D5_512A: (Total of 100 licenses issued; Total of 0 licenses
in use)
```

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 Mentor Graphics 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. On the Windows Start menu point to **Settings** and click **Control Panel**.
2. Double-click the **System** icon in the Control Panel window.
3. In the **System Properties** dialog box, click the **Advanced** tab.
4. Click on **Environment Variables**.
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**.

Verifying the Quartus II Path

Verify that the QUARTUS_ROOTDIR environment variable points at the correct version of the Quartus II software by typing the following command in the MATLAB Command Window:

```
!echo %QUARTUS_ROOTDIR% ↵
```

This command should return the path specified by the QUARTUS_ROOTDIR environment variable. For example:

```
C:\altera\81\quartus
```

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, delete any DSP Builder files or directories that remain in the file system to ensure that you re-install a clean file set.

SignalTap II Analysis Appears to be Hung

The SignalTap II Embedded Logic 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 does not terminate and the JTAG node remains locked.

You can either disconnect and reconnect the USB cable, or switch off the board and switch it on again. You need to program the board again if it is powered off.

Error if 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 II 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 Fitter 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` starts 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 if 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:

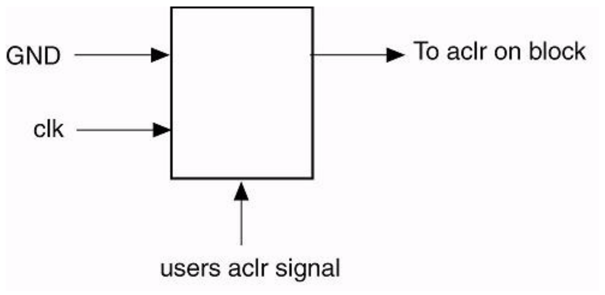
```
Warning: aclrInputPortTest/aclr has been renamed to avoid conflict:
                                   aclr has been renamed to aclr_1:
```

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 12–3.

Figure 12–3. 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 Installation and Licensing* manual 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` block followed immediately by a `Non-synthesizable Input` block. 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` block or a `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.

Unexpected Exception Error when Generating Blocks

Errors of the following form may be issued when you generate a DSP Builder system:

```
Info: IP Generator Info: stderr: No clock info for
                                     my_alt_dspbuilder_clock

Info: IP Generator Info: stderr: Failed to find clock
                                     my_alt_dspbuilder_clock

Info: IP Generator Info: stderr: Failed to find clock
                                     my_alt_dspbuilder_clock

Error: IP Generator Error: Unexpected exception thrown by MDLFactory:
                                java.lang.NullPointerException
```

```
Error: Node instance "dut" instantiates undefined entity
          "TestBarrelShifter" File: <path>/mytoplevel.vhd Line: 30
```

This problem is caused by corrupted Librarian IP cache and can be resolved by deleting the IP cache directory which is normally located at:

C:\Documents and Settings\<*user*>\.altera.quartus\ip_cache

VHDL Entity Names Change if a Model is Modified

The VHDL files generated by Signal Compiler have a random number suffix appended to the file if the model is modified.

For example, if you change the pipeline delay on a Delay block, the corresponding VHDL file: **alt_dspbuilder_delay_<*randomnumber*>** changes, while the VHDL file name for the rest of the blocks in the model remain the same.

This may be a problem if you have project assignments to a specific entity. This problem can be solved by using a regular expression in the project assignments as described in "Making Quartus II Assignments to Block Entity Names" on page 3–26.

Algebraic Loop Causes Simulation to Fail

HDL Import and IP Toolbench-based MegaCore function blocks provide an interface for changing the direct feedthrough settings of their inputs.

Algebraic loops are loops entirely comprised of blocks having some inputs which are direct feedthrough, that is, inputs that have a purely combinational path to at least one output of the block.

For more information about algebraic loops, refer to the MATLAB help.

The feature to automatically infer the correct direct feedthrough values is disabled by default for HDL Import (and all inputs are by treated as direct feedthrough). It can be enabled by typing the following command in the MATLAB command window:

```
set_param(<HDL Import block name>, 'use_dynamic_feedthrough_data', 'on')
```

The direct feedthrough settings for the HDL Import block are updated after a successful compile of the HDL when this parameter is on.

This feature is not guaranteed to generate correct settings when importing low-level LPM-based HDL.

A more direct method of changing the direct feedthrough settings is to modify the *InDelayed* parameter on HDL Import or MegaCore function blocks, using the following command:

```
set_param(<block name>, 'inDelayed', <feedthrough setting>)
```

For example, if the block is named `My_HDL`:

```
set_param(<My_HDL>, 'inDelayed', '1 0 0 1')
```

A valid value of this parameter is a series of digits, one for each of the inputs on the block (from top to bottom), with a `0` indicating direct feedthrough, and a `1` indicating that all paths to outputs from this input are registered.

☞ Specifying a value of `1` for an input, when it is in fact direct feedthrough, causes Simulink to treat combinational paths as registered, and results in incorrect simulation results.

It is possible to adjust the order in which Simulink exercises all the blocks in a feedback loop, by giving blocks a priority value. This is useful if you know which block is providing the correct initial values.

The priority of a block can be set using the **General** tab in the block properties for a block. A lower value of priority causes a block to be executed before a block with a higher value.

Part Number ######

Revision History

The table below displays the revision history for the chapters in this user guide.

Date	Version	Changes Made
November 2008	8.1	Applied new technical publications style. Removed obsolete Simulation Accelerator chapter. Updated HIL block user interface. Miscellaneous GUI updates.
May 2008	8.0	Revised introduction and design flow section, added new procedure sections for “Displaying Port data Types”, “Displaying the Pipeline Depth”, “Adding Quartus II Constraints”. and “Analyzing the Hardware Resource Usage”, Managing Projects and Files”, and “Exporting HDL”, Added procedures for using the Quartus II state machine editor.
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.
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.

Date	Version	Changes Made
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, see the following table.

Contact *(Note 1)*	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Altera literature services	Email	literature@altera.com
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note:
(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table shows the typographic conventions that this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicates command names, dialog box titles, dialog box options, and other GUI labels. For example, **Save As** dialog box.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, and software utility names. For example, **\qdesigns** directory, **d:** drive, and **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Indicates document titles. For example, *AN 519: Stratix IV Design Guidelines.*
Italic type	Indicates variables. For example, *n* + 1. Variable names are enclosed in angle brackets (< >). For example, <*file name*> and <*project name*>**.pof** file.
Initial Capital Letters	Indicates keyboard keys and menu names. For example, Delete key and the Options menu.
“Subheading Title”	Quotation marks indicate references to sections within a document and titles of Quartus II Help topics. For example, “Typographic Conventions.”

Visual Cue	Meaning
`Courier type`	Indicates signal, port, register, bit, block, and primitive names. For example, `data1`, `tdi`, and `input`. Active-low signals are denoted by suffix `n`. Example: `resetn`. Indicates command line commands and anything that must be typed exactly as it appears. For example, `c:\qdesigns\tutorial\chiptrip.gdf`. Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword `SUBDESIGN`), and logic function names (for example, `TRI`).
1., 2., 3., and a., b., c., and so on.	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■	Bullets indicate a list of items when the sequence of the items is not important.
☞	The hand points to information that requires special attention.
⚠ CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
⚠ WARNING	A warning calls attention to a condition or possible situation that can cause you injury.
↵	The angled arrow instructs you to press the Enter key.
👣	The feet direct you to more information about a particular topic.

Other Documentation

Refer to the *DSP Builder Reference Manual* for a description of the parameters supported by each of the blocks inthe DSP Builder standard blockset.

Refer to the *DSP Builder Advanced Blockset Reference Manual* for a description of the parameters supported by each of the blocks inthe DSP Builder advanced blockset.

☞ 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.

Refer to the *DSP Builder Advanced Blockset User Guide* for user information and descriptions of the example designs provided with the advanced blockset.

Refer to the *DSP Builder Installation and Licensing* manual for information about system requirements, obtaining and installing the software, setting up licensing, and upgrading from a pre-v7.1 release.

Refer to the *DSP Builder Release Notes and Errata* for information about new features, known errata issues and workarounds.

Index