DSP Builder

User Guide

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
http://www.altera.com

Product Version: 2.1.1
Document Version: 2.1.1 rev. 1
Document Date: February 2003

I.S. EN ISO 9001

UG-DSPBUILDER-2.2

About this User Guide

This user guide provides comprehensive information about the Altera® DSP Builder.

Table 1 shows the user guide revision history.

- See "Features" on page 13 for a complete list of the product features, including new features in this release.
- Refer to the DSP Builder readme file for late-breaking information that is not available in this user guide.

Table 1. User Guide Revision History

Date	Description
February 2003, v2.1.1	Added information on "Using DSP Builder Modules in External RTL Designs" on page 88. Added "Appendix—Creating Custom Library Blocks" on page 249. Additional minor documentation updates.
December 2002, v2.1.0	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 SignalCompiler block.
June 2002, v2.0.0	Updated information on the SignalCompiler 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, v1.0	First version of the user guide for DSP Builder version 1.0.0.

How to Find Information

- The Adobe Acrobat Find feature allows you to search the contents of a PDF file. Click the binoculars icon in the top toolbar to open the Find dialog box.
- Bookmarks serve as an additional table of contents.
- Thumbnail icons, which provide miniature previews of each page, provide a link to the pages.
- Numerous links, shown in green text, allow you to jump to related information.

How to Contact Altera

For the most up-to-date information about Altera products, go to the Altera world-wide web site at http://www.altera.com.

For additional information about Altera products, consult the sources shown in Table 2.

Table 2. How to Contact Altera

Information Type	USA & Canada	All Other Locations
Technical support	http://www.altera.com/mysupport/	http://www.altera.com/mysupport/
	(800) 800-EPLD (3753) (7:00 a.m. to 5:00 p.m. Pacific Time)	(408) 544-7000 *(1)* (7:00 a.m. to 5:00 p.m. Pacific Time)
Product literature	http://www.altera.com	http://www.altera.com
Altera literature services	lit_req@altera.com *(1)*	lit_req@altera.com *(1)*
Non-technical customer service	(800) 767-3753	(408) 544-7000 (7:30 a.m. to 5:30 p.m. Pacific Time)
FTP site	ftp.altera.com	ftp.altera.com

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

Typographic Conventions

The *DSP Builder User Guide* uses the typographic conventions shown in Table 3.

Table 3. Conventions

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold type. Examples: **f_{MAX}**, **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Design.*
Italic type	Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$. Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: <*file name*>, <*project name*>**.pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
"Subheading Title"	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: "Typographic Conventions."
`Courier type`	Signal and port names are shown in lowercase Courier type. Examples: `data1`, `tdi`, `input`. Active-low signals are denoted by suffix `n`, e.g., `resetn`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\qdesigns\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword `SUBDESIGN`), as well as logic function names (e.g., `TRI`) are shown in Courier.
1., 2., 3., and a., b., c.,...	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
☞	The hand points to information that requires special attention.
↵	The angled arrow indicates you should press the Enter key.
👣	The feet direct you to more information on a particular topic.

Abbreviations & Acronyms

CIC	Cascaded Integrator and Comb
DSP	Digital Signal Processing
EDIF	Electronic Design Interchange Format
EAB	Embedded Array Block
ESB	Embedded System Block
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
IIR	Infinite Impulse Response
IP	Intellectual Property
LSB	Least Significant Bit
MDL	Model File (**.mdl**)
MSB	Most Significant Bit
MUX	Multiplexer
NCO	Numerically Controlled Oscillator
PLD	Programmable Logic Device
POF	Programmer Object File
RTL	Register Transfer Level
SBF	Signed Binary Fractional
VQM	Verilog Quartus Mapping File

Contents

About this User Guide iii
How to Find Information iii
How to Contact Altera iv
Typographic Conventions v
Abbreviations & Acronyms vi

About DSP Builder 13
Features 13
General Description 14
MegaCore Functions 14
High-Speed DSP with Programmable Logic 15

Getting Started 17
Software Requirements 17
Design Flow 18
Install the DSP Builder 20
Obtaining the DSP Builder 20
Installing the DSP Builder 20
DSP Builder Directory Structure 21
Set Up Licensing 22
Appending the License to Your license.dat File 22
Specifying the DSP Builder License File in the Quartus II Software 23
DSP Builder Tutorial 24
Creating the Amplitude Modulation Model 24
Create a New Model 25
Add the SignalCompiler Block 26
Add the Sine Wave Block 27
Add the SinIn Block 29
Add the Delay Block 30
Add the SinDelay Block 31
Add the Mux Block 32
Add the Random Number Block 33
Add the Noise Block 34
Add the BusBuild Block 35
Add the GND Block 36
Add the Product Block 36
Add the StreamMod Block 37

Add the Scope Block 38
Simulate Your Model in Simulink 39
Performing RTL Simulation 40
Synthesizing & Compiling the Design 43
Automated Flow (within Simulink) 43
Manual Flow (Outside Simulink) 43
Automated Synthesis & Compilation 44
Manual Synthesis & Compilation 45
Manual Synthesis 45
Compile the Design Manually in the Quartus II Software 46
Create a Quartus II Symbol of Your DSP Builder Design 48
Using MegaCore Functions 48
MegaCore Functions in MATLAB/Simulink 49
MegaCore Simulation Options 52
Performing SignalTap II Logic Analysis 53
Open the Walkthrough Example Design 54
Specify the Nodes to Analyze 56
Turn On the SignalTap II Option in SignalCompiler 58
Generate VHDL, Synthesize, Compile & Download the Design to the DSP Board 59
Specify Trigger Conditions 59
Specify the Radix for the Bus Groups 61
Perform SignalTap II Analysis 61

DSP Builder Design Rules 63
About the DSP Builder Blocks 63
Bit Width Design Rule 63
Frequency Design Rule 64
Single Clock Domain 64
Multiple Clock Domains 70
Using Advanced PLL Features 74
DSP Builder Naming Conventions 75
SignalCompiler Design Rules 75
Fixed-Point Notation 75
Binary Point Location in SBF 77
Block I/O Formats 78
Goto & From Block Support 83
MegaCore Function Support 84
Hierarchical Design 85
Black Boxing 86
Using DSP Builder Modules in External RTL Designs 88

AltLab Library 91
HDL SubSystem Block 91
SignalCompiler Block 92
Data Width Propagation 95

Tapped Delay Line 97
Arithmetic Operation 97
Clock Assignment 101
DSP Builder Report File 104
SignalTap II Analysis Block 107
Design Flow 107
Nodes 111
Trigger Conditions 112
Node Block 113
SubSystem Builder Block 114

Arithmetic Library 117
Comparator Block 118
Divider Block 119
Gain Block 120
Increment Decrement Block 122
Multiply Accumulate Block 124
Multiply Add Block 126
Magnitude Block 128
Parallel Adder Subtractor Block 129
Product Block 131

Bus Manipulation Library 133
AltBus Block 133
AltBus Input Port & Output Port Modes 134
AltBus Internal Node Mode 135
AltBus Black Box Input Output Mode 136
AltBus Constant Mode 137
Input, Output & Constant Blocks 138
Binary Point Casting Block 140
BusBuild Block 141
Bus Concatenation Block 143
BusConversion Block 144
ExtractBit Block 146
GND Block 147
VCC Block 148

Complex Signals Library 149
Butterfly Operator Block 149
Complex AddSub Block 151
Complex Product Block 152
Complex Multiplexer Block 153
Complex Conjugate Block 154
Complex Delay Block 155
Complex Constant Block 156
Complex to Real-Imag Block 157

Real-Imag to Complex Block 158

DSP Board Library 159
APEX DSP Board EP20K1500EBC652-1X Library 159
APEX DSP Board EP20K200EBC652-1X Library 162
Stratix DSP Board EP1S25 Library 165

Gates Library 169
Case Statement Block 169
If Statement Block 172
Logical Bit Operator Block 174
Logical Bus Operator Block 175
LUT Block 176
n-to-1 Multiplexer Block 177

Rate Change Library 179
PLL Block 180
Multi-Rate DFF Block 182
Tsamp Block 183

SOPC Ports Library 185
Avalon Ports 186
Avalon Ports Block Description 186
Avalon Ports Walkthrough 189
Open the Walkthrough Example Design 190
Turn On the SOPC Option in SignalCompiler & Generate VHDL & PTF Files .191
Instantiating your Design as a Custom Peripheral to the Nios Embedded Processor in SOPC Builder 193
Custom Instruction 195
Custom Instruction Block Description 195
Custom Instruction Walkthrough 198
Open the Walkthrough Example Design 199
Turn On the SOPC Option in SignalCompiler & Generate VHDL & PTF Files .199
Including your Design as a Custom Instruction to the Nios Embedded Processor in SOPC Builder 201

State Machine Functions Library 205
State Machine Table Block 205

Storage Library 215
Delay Block 215
Down Sampling Block 217
Dual-Port RAM Block 218
Parallel to Serial Block 220
Pattern Block 221

ROM EAB Block 222
Serial to Parallel Block 224
Shift Taps Block 225
Up Sampling Block 227

Troubleshooting 229
Fixing "Unresolved Width" Error When Converting a Model File to VHDL 229
Bit Widths in VHDL Are Different than the Expected Bit Width Based on the Simulink Model File 231
Simulink Simulation Results for the Pattern & Increment/Decrement Blocks Does Not Match the Results from ModelSim 231
Passing Pin Constraints to the AltBus Block 232
ModelSim Error for Missing "lpm_components" when Running DSP Builder Tcl Script 232
Integrating VHDL Components into Simulink Model Files (Black Boxing) 232
Read Previous DSP Builder Designs with DSP Builder v2.1.0 233
The Altera DSP Builder Folder Does Not Appear in the Simulink Library Browser 233
Automated Flow Does Not Run Properly 234
Check the Software Paths (Registry Settings) 234
Change the System Path Settings 235
APEX DSP Development Board Troubleshooting 236
Error Message: SignalCompiler Is Unable to Check a Valid License 236
Verifying that Your DSP Builder Licensing Functions Properly 236
Verifying that the LM_LICENSE_FILE Variable Is Set Correctly 238
Windows NT 238
Windows 2000 238
Windows 98 239
I've Tried Everything but It Still Doesn't Work 239
ModelSim Fails with the SignalCompiler-Generated Tcl Scripts 239
The Simulink Library Browser Does Not Show Altera MegaCore Blocks 240
Specifying LeonardoSpectrum, Synplify & Quartus II Path Information for SignalCompiler 240
The Wizard for an IP MegaCore Block Does Not Launch 241

Appendix—Example Tcl Scripts 243
Overview 249

Appendix—Creating Custom Library Blocks 249
Create a Custom Block 250
Create a Library Model File with the HDL Subsystem Block 250
Build the HDL Subsystem Functionality 250
Create Parameters & Functionality Using the Mask Editor 252
Generate the MATLAB Initialization Script 254

Notes:

About DSP Builder

Features

- Links The Mathworks MATLAB (Signal Processing ToolBox and Filter Design Toolbox) and Simulink environment with the Altera® Quartus® II environment
- MATLAB version 6.5/Simulink 5.0 support
- Supports Altera DSP cores that are downloadable from the Altera web site (e.g., FIR Compiler, Reed-Solomon Compiler, etc.)
- Supports Altera devices:
  - Stratix™ GX devices
  - Cyclone™ devices
  - Stratix™ devices
  - APEX™ II devices
  - APEX 20KE devices
  - APEX 20KC devices
  - Mercury™ devices
  - ACEX® 1K devices
  - FLEX® 10K devices
  - FLEX 6000 devices
- Enables rapid prototyping using the 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 work space to facilitate visual analysis
- Includes blocks that you can use to build custom logic that works with the SOPC Builder and Nios® embedded processor designs
- Includes PLL block for multi-clock designs
- Includes state machine block
- Supports a unified representation of the algorithm and implementation of a DSP system
- Automatically generates a VHDL testbench or Quartus II Vector File (**.vec**) from MATLAB and Simulink test vectors
- Automatically launches VHDL synthesis and Quartus II compilation
- Enables bit- and cycle-accurate design simulation
- Provides a variety of fixed-point arithmetic and logical operators for use with the Simulink software

General Description

Digital signal processing (DSP) system design in Altera programmable logic devices 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 synthesis, simulation, and Altera development tools.

The 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. DSP Builder allows system, algorithm, and hardware designers to share a common development platform.

You can use the blocks in the DSP Builder to create a hardware implementation of a system modeled in Simulink in sampled time. The 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.

MegaCore Functions

Altera MegaCore functions are usually delivered as low-cost, encrypted functions that can be instantiated directly in your design. MegaCore functions support Altera's free IP evaluation features, which allow you to verify the functionality and timing of a function prior to purchasing a license.

- The OpenCore® evaluation feature lets you test-drive IP cores for free using the Quartus II software; however, you cannot generate device programming files to test the core in hardware.

- The OpenCore Plus evaluation feature enhances the OpenCore evaluation feature by supporting free hardware evaluation. This feature allows you to generate time-limited programming files for a design that includes Altera MegaCore functions. With these files, you can perform board-level design verification before deciding to purchase licenses for the MegaCore functions.

With both evaluation features, you only need to purchase a license when you are completely satisfied with a core's functionality and performance, and would like to take your design to production.

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

High-Speed DSP with Programmable Logic

Programmable logic offers compelling performance advantages over dedicated digital signal processors. Programmable logic can be thought of as an array of elements, each of which can be configured as a complex processor routine. These processor routines can then be linked together in serial (the same way digital signal processor would execute them), or they can be connected in parallel. In parallel, they offer many times the performance of standard digital signal processors by executing hundreds of instructions at the same time. Algorithms that benefit from this improved performance include FEC, modulation/demodulation, and encryption.

Traditionally, designers had to make a trade-off between the flexibility of off-the-shelf digital signal processors and the performance of custom-built devices. Altera Stratix devices eliminate the need for this trade-off by providing exceptional performance combined with the flexibility of programmable logic devices (PLDs). Stratix devices have dedicated DSP blocks, which have high-speed parallel processing capabilities, that are optimized for DSP applications. Additionally, the TriMatrix™ memory structures can implement a wide variety of memory functions found in complex designs.

The Stratix DSP block is composed of multipliers, adders, subtractors, accumulators, a summation unit, and pipeline registers. The DSP block is optimized for all DSP applications and can provide data throughput of up to 2.0 GMACS per DSP block (for 9×9 bit data widths). The DSP block is versatile, highly efficient, and easy to use. You can implement a 4-tap FIR filter or complex multiplication inside a single DSP block without using additional logic.

The TriMatrix memory structure is a significant breakthrough in on-chip memory technology and offers a wide range of memory features. By efficiently integrating embedded RAM, the TriMatrix memory brings unprecedented amounts of memory bits (up to 10 Mbits) and memory bandwidth (12 Tbps) to PLDs. The TriMatrix memory structure consists of three different sizes of memory blocks designed for a wide array of applications ranging from small FIFOs to system caches.

Stratix devices are built to function as the central clock manager to meet your system timing challenges and are the first FPGAs to offer on-chip PLL features previously found only in high-end discrete PLL devices. The Stratix on-chip PLL has the following features:

- Spread-spectrum clocking
- Clock switchover
- Frequency synthesis
- Programmable phase shift
- Programmable delay shift
- External feedback
- Programmable bandwidth

Stratix PLLs increase system and device performance and provide advanced clock interfacing and clock-frequency synthesis.

Getting Started

Software Requirements

The following software is required to create HDL designs that use blocks from the DSP Builder:

- MATLAB version 6.1 or 6.5
- Simulink version 4.1 or 5.0
- Quartus II version 2.0 or higher

  If you want to target Cyclone or Stratix GX devices, you must use Quartus II version 2.1 SP1 or higher.

  If you want use Quartus II integrated VHDL synthesis, you must use Quartus II version 2.1 or higher.

DSP Builder provides automated flows using Tcl scripts as well as a manual flow. In addition to Quartus II native synthesis, the automated flow also supports:

- Synplify® version 7.2 or higher *or* LeonardoSpectrum™ version 2002c or higher
- ModelSim® version 5.5 or higher

If you want to target Stratix GX or Cyclone devices, you must use LeonardoSpectrum version 2002.e or higher.

For more information on any of the software products discussed in this user guide, refer to the documentation provided with the software.

Design Flow

When using the DSP Builder to build a design, you start by creating a 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 simulation. The design flow involves the following steps:

1. Create a model using the MATLAB/Simulink software using a combination of Simulink and DSP Builder blocks.

2. Perform RTL simulation. The DSP Builder supports an automated flow for the ModelSim software with Tcl scripts. You can also use the generated VHDL for manual simulation in other simulation tools. (If you are using MegaCore functions in your model, refer to the MegaCore function's user guide for information on the types of simulation the core supports.)

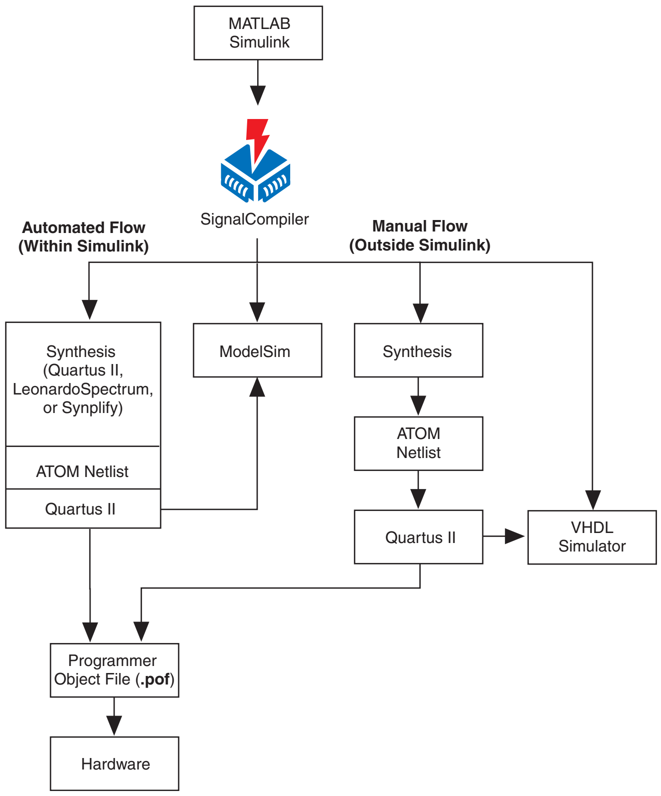
3. Use the output files generated by the DSP Builder SignalCompiler block to perform RTL synthesis. DSP Builder supports an automated synthesis flow for the Quartus II, Synplify, or LeonardoSpectrum software with Tcl scripts. Alternatively, you can synthesize the VHDL files manually using other synthesis tools.

   ☞ You can perform steps 2 and 3 in any order.

4. Compile your design in the Quartus II software.

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

Figure 1. System-Level Design Flow *Note (1)*


MATLAB Simulink
SignalCompiler
Automated Flow (Within Simulink)
Manual Flow (Outside Simulink)
Synthesis (Quartus II, LeonardoSpectrum, or Synplify)
ATOM Netlist
Quartus II
ModelSim
Synthesis
ATOM Netlist
Quartus II
VHDL Simulator
Programmer Object File (.pof)
Hardware

Note:

(1) For an automated design flow, the SignalCompiler generates VHDL files and Tcl scripts for synthesis in the Quartus II, LeonardoSpectrum, or Synplify software, compilation in the Quartus II software, and simulation in the ModelSim 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.

Install the DSP Builder

The following instructions describe how to obtain DSP Builder and install it on your PC.

Obtaining the DSP Builder

If you have Internet access, you can download DSP Builder from Altera's web site at http://www.altera.com. Follow the instructions below to obtain DSP Builder via the Internet. If you do not have Internet access, you can obtain the software from your local Altera representative.

1. Point your web browser to http://www.altera.com/products/software/system/sys-dspbuilder.html.

2. Click the link to download DSP Builder.

3. Fill out the registration form, read the license agreement, and click **I agree**.

4. Follow the on-line instructions to download the executable and save it to your hard disk.

Installing the DSP Builder

To install the DSP Builder on a PC running Windows 98/NT/2000, perform the following steps.

☞ Before you install the DSP Builder, Altera recommends that you install the MATLAB and Simulink software and the Quartus II software. You can also use the LeonardoSpectrum or Synplify software for synthesis.

1. Close the following software applications if they are open:

   – Quartus II
   – MAX+PLUS II
   – LeonardoSpectrum
   – Synplify
   – MATLAB and Simulink
   – ModelSim

2. Choose **Run** (Windows Start menu).

3. Type <*path name*>`\DSPBuilder.exe`, where <*path name*> is the location of the downloaded file.

4. Click **OK**. The **DSPBuilder v**<*version*> **- InstallShield Wizard** dialog box appears. Follow the on-line instructions to finish installation.

The DSP Builder launches the LeonardoSpectrum, Synplify, or Quartus II software by retrieving the software's path information from your PC's registry file. If you installed multiple versions of these software products, the registry may not point to the version you want to use with DSP Builder. In DSP Builder version 2.0.0 or higher you can specify a path to use other than the registry setting for each of the tools.

Refer to "Specifying LeonardoSpectrum, Synplify & Quartus II Path Information for SignalCompiler" on page 240 for information on specifying the paths.

When you have finished installing DSP Builder, view the DSP Builder libraries in the MATLAB software by performing the following steps.

1. Start the MATLAB software.

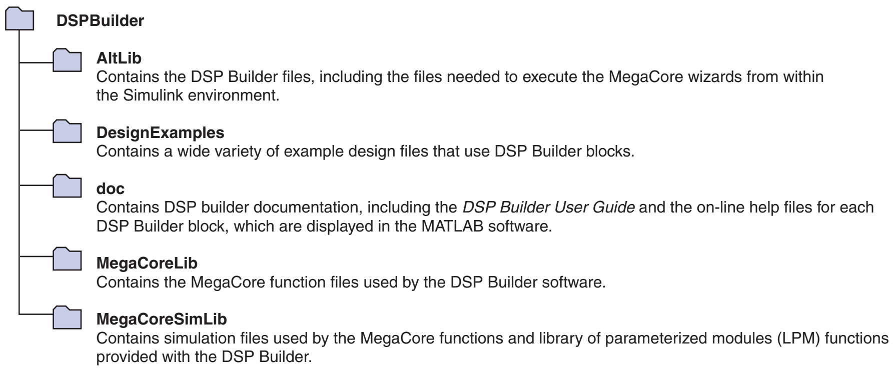
2. Expand the Simulink icon in the **Launch Pad** window.

3. Double-click the **Library Browser** icon. The **Altera DSP Builder** folder appears in the **Simulink Library Browser** window.

DSP Builder Directory Structure

The DSP Builder installation program copies files into the directories shown in Figure 2.

Figure 2. DSP Builder Directory Structure



DSPBuilder

AltLib
Contains the DSP Builder files, including the files needed to execute the MegaCore wizards from within the Simulink environment.

DesignExamples
Contains a wide variety of example design files that use DSP Builder blocks.

doc
Contains DSP builder documentation, including the *DSP Builder User Guide* and the on-line help files for each DSP Builder block, which are displayed in the MATLAB software.

MegaCoreLib
Contains the MegaCore function files used by the DSP Builder software.

MegaCoreSimLib
Contains simulation files used by the MegaCore functions and library of parameterized modules (LPM) functions provided with the DSP Builder.

Set Up Licensing

Before using the DSP Builder, you must request a license file from the Altera web site at http://www.altera.com/licensing and install it on your PC. When you request a license file, Altera e-mails you a **license.dat** file that enables HDL file and Tcl script generation. If you do not have a DSP Builder license file, you can create models with the DSP Builder blocks but you cannot generate HDL files or Tcl scripts.

☞ Before you set up licensing for the DSP Builder, you must already have the Quartus II software installed on your PC with licensing set up.

To install your license, you can either append the license to your **license.dat** file or you can specify a separate DSP Builder license file in the Quartus II software.

Appending the License to Your license.dat File

To install your license, perform the following steps.

1. Close the following software if it is running on your PC:

   - Quartus II
   - MAX+PLUS II
   - MATLAB and Simulink
   - LeonardoSpectrum
   - Synplify
   - ModelSim

2. Open the DSP Builder license file in a text editor. The file should contain one `FEATURE` line, spanning 2 lines.

3. Open your Quartus II **license.dat** file in a text editor.

4. Copy the `FEATURE` line from the DSP Builder license file and paste it into the Quartus II license file.

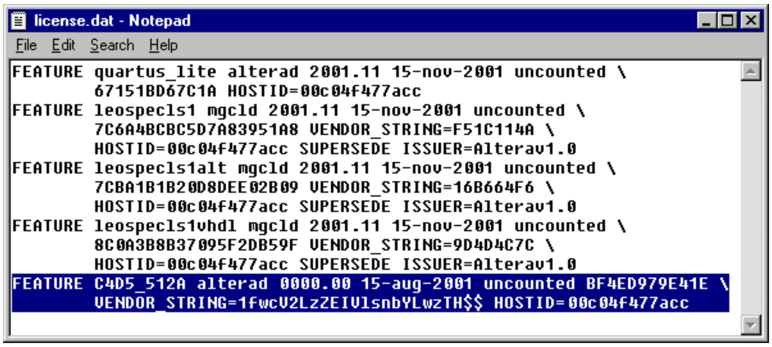
   ☞ Do not delete any `FEATURE` lines from the Quartus II license file.

5. Save the Quartus II license file.

   ☞ When using editors such as Microsoft Word or Notepad, ensure that the file does not have extra extensions appended to it after you save (e.g., **license.dat.txt** or **license.dat.doc**). Verify the filename in a DOS box or at a command prompt.

Figure 3 shows an example updated **license.dat** file that includes the DSP Builder `FEATURE` line (highlighted).

Figure 3. Example license.dat File



```
FEATURE quartus_lite alterad 2001.11 15-nov-2001 uncounted \
        67151BD67C1A HOSTID=00c04f477acc
FEATURE leospecls1 mgcld 2001.11 15-nov-2001 uncounted \
        7C6A4BCBC5D7A83951A8 VENDOR_STRING=F51C114A \
        HOSTID=00c04f477acc SUPERSEDE ISSUER=Alterav1.0
FEATURE leospecls1alt mgcld 2001.11 15-nov-2001 uncounted \
        7CBA1B1B20D8DEE02B09 VENDOR_STRING=16B664F6 \
        HOSTID=00c04f477acc SUPERSEDE ISSUER=Alterav1.0
FEATURE leospecls1vhdl mgcld 2001.11 15-nov-2001 uncounted \
        8C0A3B8B37095F2DB59F VENDOR_STRING=9D4D4C7C \
        HOSTID=00c04f477acc SUPERSEDE ISSUER=Alterav1.0
FEATURE C4D5_512A alterad 0000.00 15-aug-2001 uncounted BF4ED979E41E \
        VENDOR_STRING=1fwcV2LzZEIVlsnbYLwzTH$$ HOSTID=00c04f477acc
```

Specifying the DSP Builder License File in the Quartus II Software

To specify the license file, perform the following steps:

1. Create a text file with the `FEATURE` line and save it to your hard disk.

   ☞ Altera recommends that you give the file a unique name, e.g., **dsp_builder_license.dat**.

2. Run the Quartus II software.

3. Choose **License Setup** (Tools menu). The **Options** dialog box opens to the **License Setup** page.

4. In the **License file** box, add a semicolon to the end of the existing license path and filename.

5. Type the path and filename of the core license file after the semicolon.

   ☞ Do not include any spaces either around the semicolon or in the path/filename.

6. Click **OK** to save your changes.

DSP Builder Tutorial

This tutorial uses an example amplitude modulation design, **SinGen.mdl**, to demonstrate the DSP Builder design flow. The amplitude modulator 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—in which you can enter the parameters—opens. Click the **Help** button in the dialog boxes to view the block's on-line help.

The instructions provided in this tutorial assume that:

- You are using a PC running Windows.
- You installed the DSP Builder in the default location, **C:\DSPBuilder**.
- You are familiar with the MATLAB, Simulink, LeonardoSpectrum, Quartus II, and ModelSim software and the software is installed on your PC in the default locations.

☞ The instructions in this tutorial assume that you have basic knowledge of the Simulink software. Refer to Simulink Help for information on using the software.

This tutorial includes the following sections:

- Creating the Amplitude Modulation Model
- Performing RTL Simulation
- Synthesizing & Compiling the Design
  - Automated Synthesis & Compilation
  - Manual Synthesis & Compilation

Creating the Amplitude Modulation Model

You can create the amplitude modulation model using the instructions in this section or you can use the Altera-created model file that is provided in the DSP Builder **DesignExamples** directory.

If you want to use the Altera-provided file instead of creating the model yourself, the file **SinGen.mdl** is located in the **C:\DSPBuilder\DesignExamples\GettingStarted\SinMdl** directory. Skip to "Performing RTL Simulation" on page 40 to begin using the model.

If you did not install DSP Builder in the default location, **C:\DSPBuilder**, you must define your working directory before synthesizing and compiling the design:

a. Open the **SinGen.mdl** model.

b. Double-click SignalCompiler.

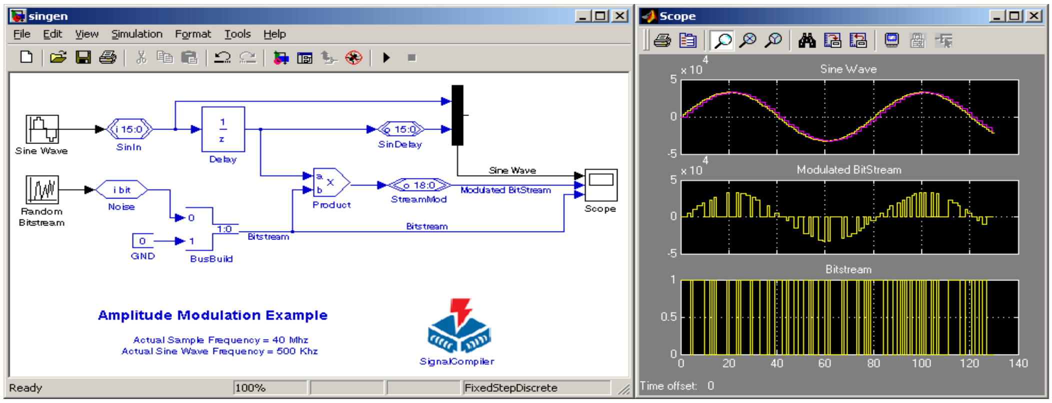
c. Click the button next to **SinGen.mdl**.

d. Browse to the directory in which the **SinGen.mdl** Model File is installed, <*path*>**\DSPBuilder\DesignExamples\GettingStarted\SinMdl**.

e. Select the Model File, **SinGen.mdl**, and click **Open**.

To create the amplitude modulation model yourself, perform the steps in the following sections. Figure 4 shows the design you will be creating.

Figure 4. Amplitude Modulation Design Example



Create a New Model

1. Start the MATLAB software.

2. Choose the **New > Model** command (File menu) to create a new model file.

3. Choose **Save** (File menu) in the new model window.

4. Browse to the directory in which you want to save the file. This directory will be your working directory. This tutorial uses the working directory **C:\DSPBuilder\DesignExamples\GettingStarted\my_SinMdl**.

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

6. Click **Save**.

7. Expand the Simulink icon in the MATLAB **Launch Pad** window by clicking the + symbol next to the icon.

8. Double-click the **Library Browser** icon.

The following sections describe how to add blocks to your model and simulate it.

Add the SignalCompiler Block

Perform the following steps to add the SignalCompiler block.

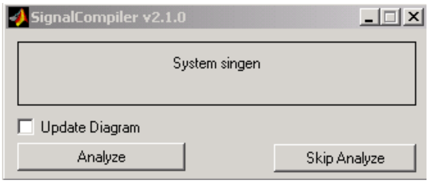
1. Select the **AltLab** library from the Altera DSP Builder folder in the **Simulink Library Browser**.

2. Drag and drop a SignalCompiler block into your model.

3. Double-click the SignalCompiler block in your model.

4. Click **Analyze** (see Figure 5).

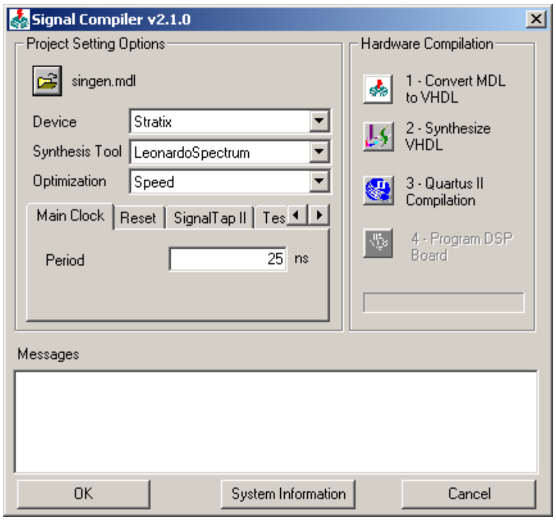
Figure 5. Click Analyze



5. Leave all other settings at the defaults (see Figure 6).

☞ You will make other SignalCompiler parameter settings in later steps to perform synthesis, compilation, and simulation.

Figure 6. Insert the SignalCompiler Block


Signal Compiler v2.1.0

Project Setting Options

singen.mdl

Device: Stratix

Synthesis Tool: LeonardoSpectrum

Optimization: Speed

Main Clock | Reset | SignalTap II | Tes

Period: 25 ns

Hardware Compilation

1 - Convert MDL to VHDL

2 - Synthesize VHDL

3 - Quartus II Compilation

4 - Program DSP Board

Messages

OK | System Information | Cancel

6. Click **OK**.

7. Choose **Save** (File menu) to save the model.

Add the Sine Wave Block

Perform the following steps to add the Sine Wave block.

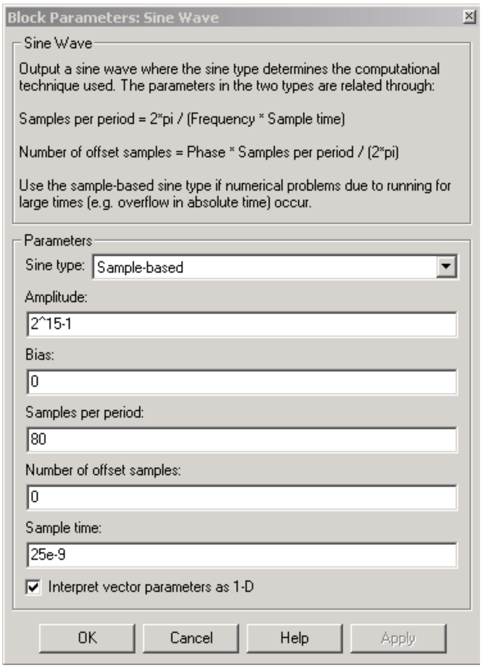
1. In the **Simulink Library Browser**, click the Simulink **Sources** library to view the blocks in the library.

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

3. Double-click the Sine Wave block in your model.

4. Set the Sine Wave block parameters as shown in Figure 7 and click **OK**. (See the equation in "Frequency Design Rule" on page 64 for information on how to calculate the frequency).

Figure 7. 500-kHz, 16-Bit Sine Wave


Block Parameters: Sine Wave

Sine Wave

Output a sine wave where the 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

Amplitude: 2^15-1

Bias: 0

Samples per period: 80

Number of offset samples: 0

Sample time: 25e-9

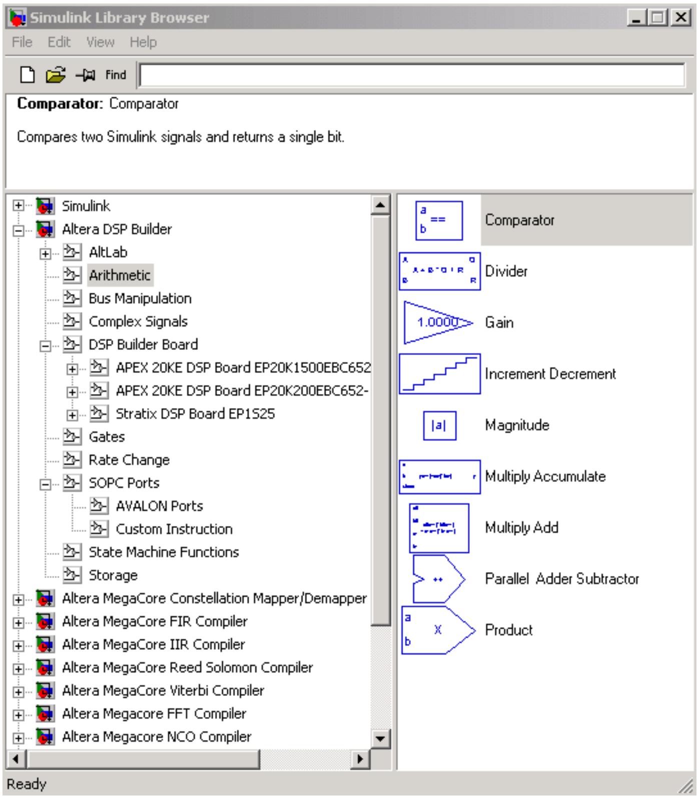
[✓] Interpret vector parameters as 1-D

Add the SinIn Block

Perform the following steps to add the SinIn block.

1. Expand the Altera DSP Builder folder by clicking the + symbol next it. The DSP Builder libraries are displayed. Leave the Altera DSP Builder folder expanded for the rest of the tutorial. See Figure 8.

Figure 8. Altera DSP Builder Folder in the Simulink Library Browser



Comparator: Comparator

Compares two Simulink signals and returns a single bit.

Simulink
Altera DSP Builder
AltLab
Arithmetic
Bus Manipulation
Complex Signals
DSP Builder Board
APEX 20KE DSP Board EP20K1500EBC652
APEX 20KE DSP Board EP20K200EBC652-
Stratix DSP Board EP1S25
Gates
Rate Change
SOPC Ports
AVALON Ports
Custom Instruction
State Machine Functions
Storage
Altera MegaCore Constellation Mapper/Demapper
Altera MegaCore FIR Compiler
Altera MegaCore IIR Compiler
Altera MegaCore Reed Solomon Compiler
Altera MegaCore Viterbi Compiler
Altera Megacore FFT Compiler
Altera Megacore NCO Compiler
Comparator
Divider
1.0000
Gain
Increment Decrement
Magnitude
Multiply Accumulate
Multiply Add
Parallel Adder Subtractor
Product
Ready

2. Select the **Bus Manipulation** library.

   ☞ If you are unsure how to position the blocks or draw connection lines, refer to the completed design shown in Figure 4 on page 25.

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

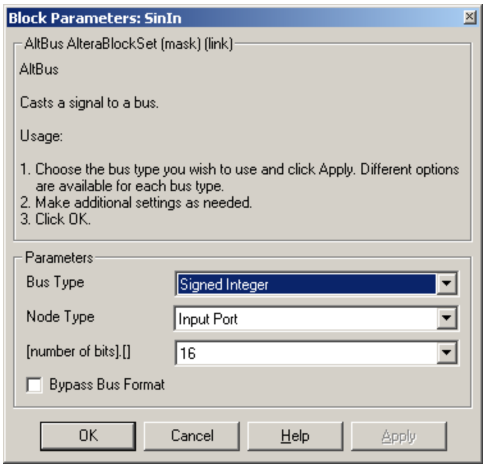
4. Click the text `AltBus` under the block icon in your model.

5. Delete the text `AltBus` and type the text `SinIn` to change the name of the block instance.

6. Double-click the SinIn block in your model. The block's parameter dialog box displays.

7. Set the parameters as shown in Figure 9 and click **OK**.

Figure 9. Setting the 16-Bit Sign Integer Input



Block Parameters: SinIn

AltBus AlteraBlockSet (mask) (link)

AltBus

Casts a signal to a bus.

Usage:

1. Choose the bus type you wish to use and click Apply. Different options are available for each bus type.
2. Make additional settings as needed.
3. Click OK.

Parameters

Bus Type	Signed Integer
Node Type	Input Port
[number of bits].[]	16

☐ Bypass Bus Format

8. Draw a connection line from the right side of the Sine Wave block to the left side of the SinIn block.

Add the Delay Block

Perform the following steps to add the Delay block.

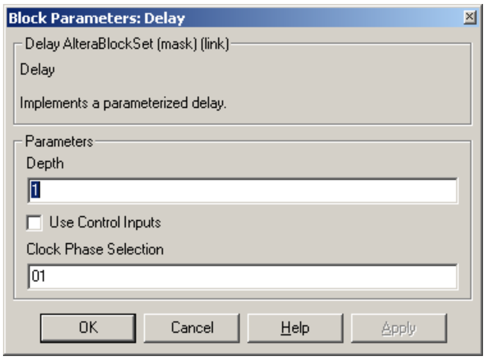
1. Select the **Storage** library from the Altera DSP Builder folder in the **Simulink Library Browser**.

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.

4. Set the parameters as shown in Figure 10 and click **OK**.

Figure 10. Downsampling Delay by 2



Block Parameters: Delay

Delay AlteraBlockSet (mask) (link)

Delay

Implements a parameterized delay.

Parameters

Depth: 1

☐ Use Control Inputs

Clock Phase Selection: 01

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

Add the SinDelay Block

Perform the following steps to add the SinDelay block.

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

2. Drag and drop an AltBus block into your model, positioning it to the right of the Delay block.

3. Click the text `AltBus` under the block icon in your model.

4. Change the block instance name by deleting the text `AltBus` and typing in the text `SinDelay`.

5. Double-click the SinDelay block in your model.

6. Choose **Output Port** from the **Node Type** list.

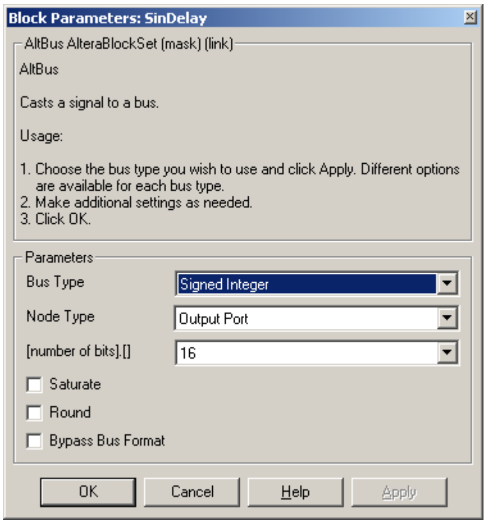
7. Click **Apply**.

   The dialog box options change when you select a new node type and click **Apply**.

8. Choose **16** from the **[number of bits].[]** list. Figure 11 shows the dialog box after you have made these settings.

9. Click **OK** to save your changes.

Figure 11. 16-Bit Signed Output Bus


Block Parameters: SinDelay

AltBus AlteraBlockSet (mask) (link)

AltBus

Casts a signal to a bus.

Usage:

1. Choose the bus type you wish to use and click Apply. Different options are available for each bus type.
2. Make additional settings as needed.
3. Click OK.

Parameters

Bus Type: Signed Integer

Node Type: Output Port

[number of bits].[]: 16

☐ Saturate

☐ Round

☐ Bypass Bus Format

10. 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.

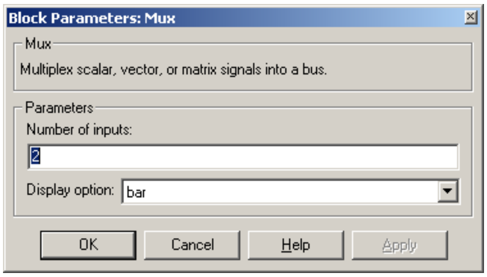
1. Select the Simulink **Signal Routing** library (**Signals & Systems** in Simulink v4.1) in the **Simulink Library Browser**.

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.

4. Set the parameters as shown in Figure 12 and click **OK**.

Figure 12. 2-to-1 Multiplexer


Block Parameters: Mux

Mux

Multiplex scalar, vector, or matrix signals into a bus.

Parameters

Number of inputs: 2

Display option: bar

5. Draw a connection line from the bottom left of the Mux block to the right side of the SinDelay block.

6. Draw a connection line from the top left of the Mux block to the line between the SinIn and Delay blocks.

Add the Random Number Block

Perform the following steps to add the Random Number block.

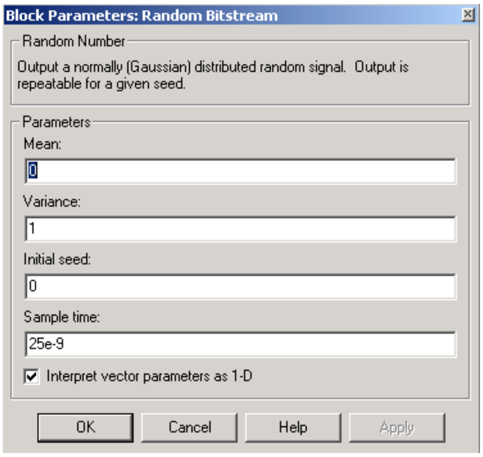
1. Select the Simulink **Sources** library in the **Simulink Library Browser**.

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.

4. Set the parameters as shown in Figure 13 and click **OK**.

Figure 13. Random Number Generator



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

Add the Noise Block

Perform the following steps to add the Noise block.

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

2. Drag and drop an AltBus block into your model, positioning it to the right of the Random Number block.

3. Click the text `AltBus` under the block icon in your model.

4. Change the name of the block instance by deleting the text `AltBus` and typing in the text `Noise`.

5. Double-click the Noise block.

6. Choose the **Single Bit** option from the **Bus Type** list.

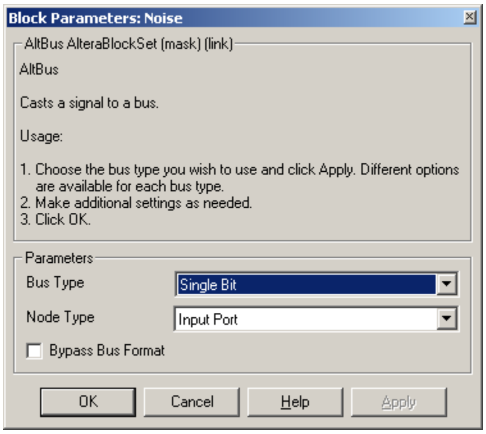
7. Turn off the **Bypass Bus Format** option.

8. Click **Apply**. Figure 14 shows the dialog box after you have made this setting.

   ☞ The dialog box options change when you select a new bus type and click **Apply**.

9. Click **OK**.

Figure 14. 1-Bit Input Port


Block Parameters: Noise

AltBus AlteraBlockSet (mask) (link)

AltBus

Casts a signal to a bus.

Usage:

1. Choose the bus type you wish to use and click Apply. Different options are available for each bus type.
2. Make additional settings as needed.
3. Click OK.

Parameters

Bus Type: Single Bit

Node Type: Input Port

☐ Bypass Bus Format

10. Draw a connection line from the right side of the Random Number block to the left side of the Noise block.

Add the BusBuild Block

Perform the following steps to add the BusBuild block. This block converts a bit to a signed bus.

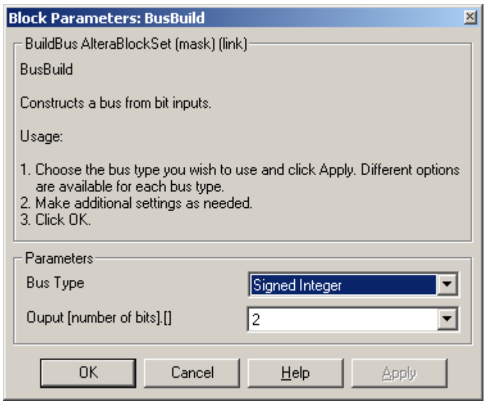
1. Select the **Bus Manipulation** library from the Altera DSP Builder folder in the **Simulink Library Browser**.

2. Drag and drop the BusBuild block into your model, positioning it to the right of the Noise block.

3. Double-click the BusBuild block in your model.

4. Set the parameters as shown in Figure 15 and click **OK**.

Figure 15. Build a 2-Bit Signed Bus



Block Parameters: BusBuild

BuildBus AlteraBlockSet (mask) (link)

BusBuild

Constructs a bus from bit inputs.

Usage:

1. Choose the bus type you wish to use and click Apply. Different options are available for each bus type.
2. Make additional settings as needed.
3. Click OK.

Parameters

Bus Type: Signed Integer

Ouput [number of bits].[]: 2

5. Draw a connection line from the right side of the Noise block to the top left side of the BusBuild block.

Add the GND Block

Perform the following steps to add the Ground block.

1. Select the **Bus Manipulation** library from the Altera DSP Builder 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 BusBuild block.

Add the Product Block

Perform the following steps to add the Product block.

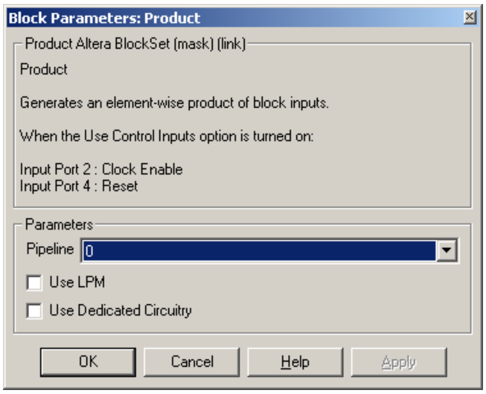
1. Select the **Arithmetic** library from the Altera DSP Builder folder in the **Simulink Library Browser**.

2. Drag and drop a Product block into your model, positioning it to the right of the BusBuild block and slightly above it. Leave enough space so that you can draw a connection line underneath the Product block.

3. Double-click the Product block.

4. Set the parameters as shown in Figure 16.

Figure 16. Multiply Two Signals



Block Parameters: Product

Product Altera BlockSet (mask) (link)

Product

Generates an element-wise product of block inputs.

When the Use Control Inputs option is turned on:

Input Port 2 : Clock Enable
Input Port 4 : Reset

Parameters

Pipeline: 0

☐ Use LPM

☐ Use Dedicated Circuitry

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 Block

Perform the following steps to add the StreamMod block.

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

2. Drag and drop an AltBus block into your model, positioning it to the right of the Product block.

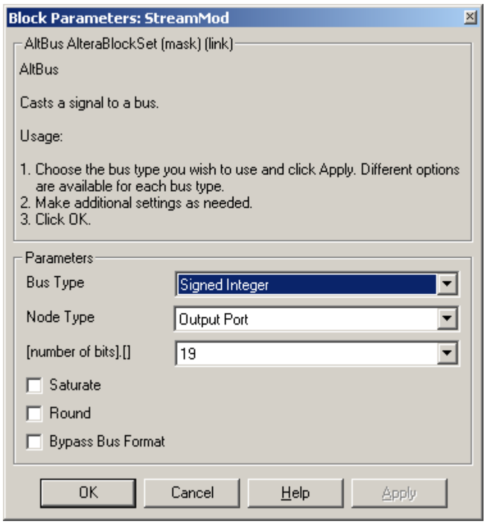
3. Click the text `AltBus` under the block icon in your model.

4. Delete the text `AltBus` and type the text `StreamMod` to change the block instance name.

5. Double-click the StreamMod block.

6. Set the parameters as shown in Figure 17.

Figure 17. 19-Bit Signed Output Bus



Block Parameters: StreamMod

AltBus AlteraBlockSet (mask) (link)

AltBus

Casts a signal to a bus.

Usage:

1. Choose the bus type you wish to use and click Apply. Different options are available for each bus type.
2. Make additional settings as needed.
3. Click OK.

Parameters

Bus Type	Signed Integer
Node Type	Output Port
[number of bits].[]	19

☐ Saturate

☐ Round

☐ Bypass Bus Format

7. Click **OK.**

8. Draw a connection line from the right side of the Product block to the left side of the StreamMod 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.

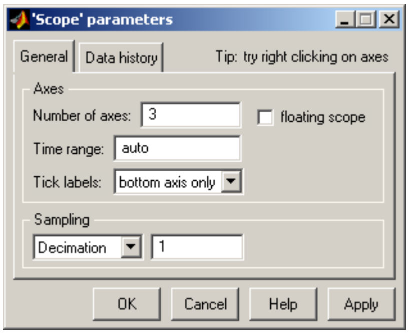
4. Click the Parameters icon.

5. Type `3` in the **Number of axes** box to display three signals in time. Figure 18 shows the **'Scope' Parameters** dialog box after you have made the setting.

6. Click **OK**.

7. Close the Scope.

Figure 18. Display 3 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

8. Draw a connection line from the right side of the Mux block to the top left side of the Scope block.

9. Draw a connection line from the right side of the StreamMod block to the middle left side of the Scope block.

10. Draw a connection line from the right side of the BusBuild block to the bottom left of the Scope block.

11. Draw a connection line from the bottom left of the Product block to the line between the BusBuild block and the Scope block.

Simulate Your Model in Simulink

Before simulating your design, check to make sure it is drawn as shown in Figure 4 on page 25.

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

1. Choose **Simulation parameters** (Simulation menu).

2. Type `0.000002` in the **Stop time** box to display 200 samples.

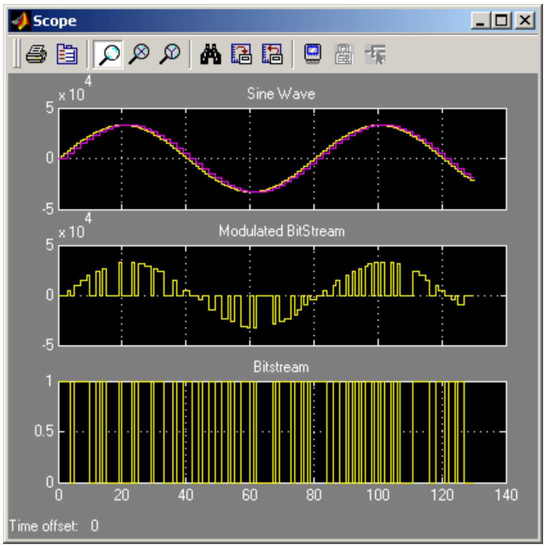
3. Click **OK**.

4. Start simulation by choosing **Start** (Simulation menu) or by pressing the Ctrl+T keys.

5. Double-click the Scope block to view the simulation results.

6. Click the binoculars icon to auto-scale the waveforms. Figure 19 shows the scaled waveforms.

Figure 19. Scope Simulation Results



You are now ready to perform RTL simulation.

Performing RTL Simulation

When you turn on the **Generate Stimuli for VHDL Testbench** option in SignalCompiler, SignalCompiler generates a VHDL testbench and Tcl script for your model. You can use both files with the ModelSim software or you can use just the testbench in another simulation tool.

Refer to “MegaCore Simulation Options” on page 52 for additional information on simulating MegaCore functions.

To generate the simulation files for the amplitude modulation design example, perform the following steps:

1. Double-click the SignalCompiler block in your model.

2. Click **Analyze**.

3. Click **1 - Convert MDL to VHDL**.

4. Click the right arrow to scroll the tabs to the left.

5. Click the **Testbench** tab.

6. Turn on the **Generate Stimuli for VHDL Testbench** option.

7. Click **OK**.

8. Run the simulation in Simulink to generate the input stimulus files by choosing **Start** (Simulation menu). SignalCompiler generates a simulation script, **Tb_SinGen.tcl**, and a VHDL testbench that imports the Simulink input stimuli, **Tb_ SinGen.vhd**.

To perform RTL simulation with the ModelSim software, perform the following steps:

1. Start the ModelSim software.

2. Choose **Change Directory** (File menu).

3. Browse to your working directory and click **Open**.

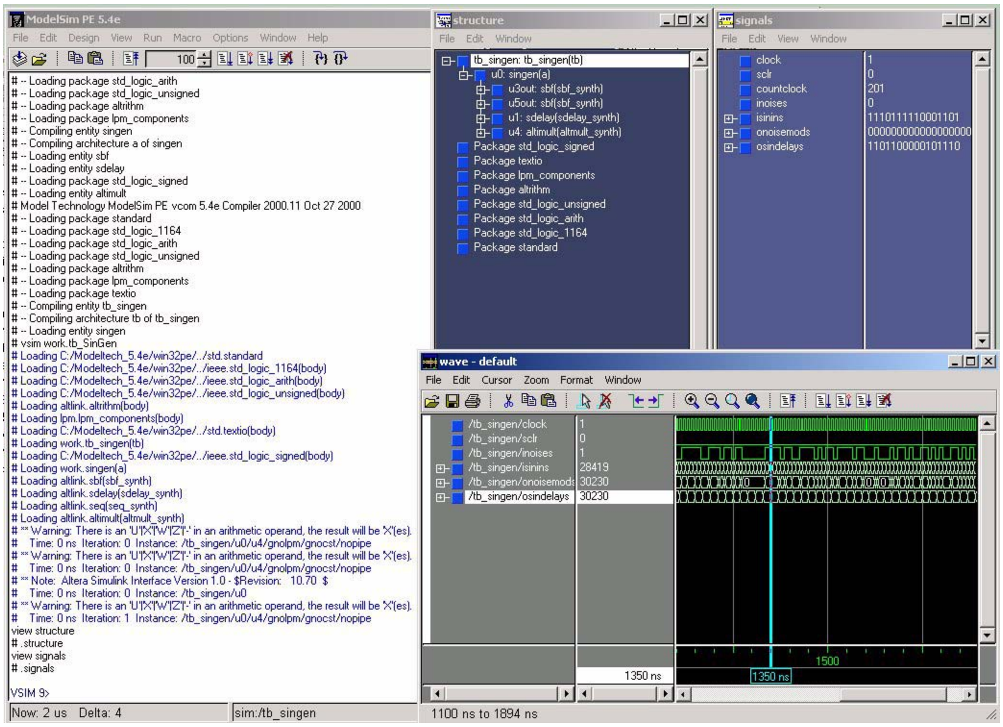
4. Choose **Execute Macro** (Macro menu).

5. Browse for the **Tb_SinGen.tcl** script and click **Open**.

The simulation results are displayed in a waveform. The testbench initializes all of the design registers with a pulse on the `SRST` input signal.

The ModelSim waveform editor displays the signals in decimal notation. You can view the signals as a digital or analog waveform. Figure 20 on page 42 shows the simulation and Figure 21 shows the analog display.

Figure 20. VHDL Simulation



```
# -- Loading package std_logic_arith
# -- Loading package std_logic_unsigned
# -- Loading package altrithm
# -- Loading package lpm_components
# -- Compiling entity singen
# -- Compiling architecture a of singen
# -- Loading entity sbf
# -- Loading entity sdelay
# -- Loading package std_logic_signed
# -- Loading entity altimult
# Model Technology ModelSim PE vcom 5.4e Compiler 2000.11 Oct 27 2000
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Loading package std_logic_arith
# -- Loading package std_logic_unsigned
# -- Loading package altrithm
# -- Loading package lpm_components
# -- Loading package textio
# -- Compiling entity tb_singen
# -- Compiling architecture tb of tb_singen
# -- Loading entity singen
# vsim work.tb_SinGen
# Loading C:/Modeltech_5.4e/win32pe/../std.standard
# Loading C:/Modeltech_5.4e/win32pe/../ieee.std_logic_1164(body)
# Loading C:/Modeltech_5.4e/win32pe/../ieee.std_logic_arith(body)
# Loading C:/Modeltech_5.4e/win32pe/../ieee.std_logic_unsigned(body)
# Loading altlink.altrithm(body)
# Loading lpm.lpm_components(body)
# Loading C:/Modeltech_5.4e/win32pe/../std.textio(body)
# Loading work.tb_singen(tb)
# Loading C:/Modeltech_5.4e/win32pe/../ieee.std_logic_signed(body)
# Loading work.singen(a)
# Loading altlink.sbf(sbf_synth)
# Loading altlink.sdelay(sdelay_synth)
# Loading altlink.seq(seq_synth)
# Loading altlink.altimult(altmult_synth)
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ns  Iteration: 0  Instance: /tb_singen/u0/u4/gnolpm/gnocst/nopipe
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ns  Iteration: 0  Instance: /tb_singen/u0/u4/gnolpm/gnocst/nopipe
# ** Note:  Altera Simulink Interface Version 1.0 - $Revision:   10.70  $
#    Time: 0 ns  Iteration: 0  Instance: /tb_singen/u0
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ns  Iteration: 1  Instance: /tb_singen/u0/u4/gnolpm/gnocst/nopipe
view structure
# .structure
view signals
# .signals

VSIM 9>
```

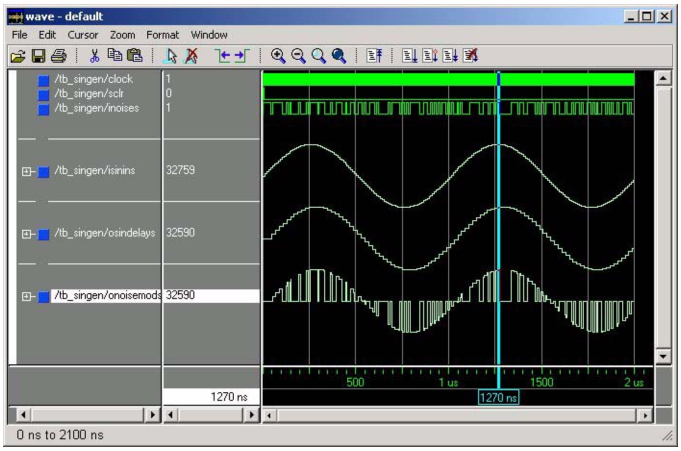
```
Now: 2 us   Delta: 4        sim:/tb_singen
```

```
tb_singen: tb_singen(tb)
  u0: singen(a)
    u3out: sbf(sbf_synth)
    u5out: sbf(sbf_synth)
    u1: sdelay(sdelay_synth)
    u4: altimult(altmult_synth)
  Package std_logic_signed
  Package textio
  Package lpm_components
  Package altrithm
  Package std_logic_unsigned
  Package std_logic_arith
  Package std_logic_1164
  Package standard
```

clock	1
sclr	0
countclock	201
inoises	0
isinins	1110111110001101
onoisemods	000000000000000000
osindelays	1101100000101110

/tb_singen/clock	1
/tb_singen/sclr	0
/tb_singen/inoises	1
/tb_singen/isinins	28419
/tb_singen/onoisemods	30230
/tb_singen/osindelays	30230

Figure 21. Analog Display



/tb_singen/clock	1
/tb_singen/sclr	0
/tb_singen/inoises	1
/tb_singen/isinins	32759
/tb_singen/osindelays	32590
/tb_singen/onoisemods	32590

You are now ready to perform synthesis and Quartus II compilation.

Synthesizing & Compiling the Design

Altera provides two synthesis and compilation flows for DSP Builder: the automated and manual flows. If the DSP Builder design is the top-level design, you can use either flow with the associated tool's Tcl script created on-the-fly by SignalCompiler. If the DSP Builder design is not the top-level design but is instead a hierarchical module of a non-DSP Builder hardware design, you cannot use the automated flow or the synthesis or compilation Tcl scripts created by SignalCompiler. You must manually create the compilation settings of a top-level design that is created outside of DSP Builder, including:

- Adding all of the DSP Builder VHDL files into the synthesis project. This information is available in the File section of the synthesis tool's (Quartus II, Synplify, or LeonardoSpectrum) Tcl file created by SignalCompiler.
- Adding all the required IP libraries to the Quartus II project. This information is available in the User Library section of the Quartus II Tcl script created by SignalCompiler.

Automated Flow (within Simulink)

The automated flow allows you to control your entire synthesis and compilation flow from within the MATLAB/Simulink environment using the SignalCompiler block. With the automated flow, the SignalCompiler block outputs VHDL files and Tcl scripts, performs synthesis in the Quartus II, LeonardoSpectrum, or Synplify software, compiles in the Quartus II software, and, optionally, downloads the design to one of the DSP development boards. You do not need to make additional settings in these software tools or run them separately. The automated flow can help you prototype designs rapidly.

Manual Flow (Outside Simulink)

With the manual flow, you use SignalCompiler to output VHDL files and Tcl scripts; however, you do not use it to perform synthesis or Quartus II compilation. After SignalCompiler generates the VHDL files, you can perform synthesis in tools other than the Quartus II, LeonardoSpectrum or Synplify software, and then perform compilation in the Quartus II software. Additionally, you should use the manual flow if you want to specify your own synthesis or compilation settings.

When generating output files, SignalCompiler maps each Altera DSP Builder block in the MATLAB/Simulink design to the DSP Builder's VHDL library. MegaCore functions are treated as black boxes.

See "Using MegaCore Functions" on page 48 for more information on using MegaCore functions in a model.

Automated Synthesis & Compilation

SignalCompiler maps each Altera DSP Builder block in the **SinGen.mdl** design to the DSP Builder VHDL library. With the automated flow, you use the buttons under **Hardware Compilation** to perform synthesis in the Quartus II, Synplify or LeonardoSpectrum software and compilation in the Quartus II software.

To synthesize and compile the design manually, go to "Manual Synthesis & Compilation" on page 45.

Perform the following steps for automatic synthesis and compilation:

1. Double-click the SignalCompiler block in your model.

2. Choose **Stratix** from the **Device Family** list.

3. Choose **Speed** from the **Optimization** list.

4. Choose **LeonardoSpectrum** from the **Synthesis Tool** list. This tutorial uses the LeonardoSpectrum synthesis software. If you want to use the Synplify software, choose **Synplify** from the **Synthesis Tool** list. If you want to use the Quartus II software, choose **Quartus II** from the **Synthesis Tool** list.

   If you select Quartus II VHDL synthesis, Quartus II VHDL synthesis and Quartus II compilation are performed consecutively when you click the **Quartus II Compilation** button.

5. Click the button next to **1 - Convert MDL to VHDL**. SignalCompiler converts the design to VHDL.

6. Click the button next to **2 - Synthesize VHDL**. Synthesis begins in the tool you selected.

7. Click the button next to **3 - Quartus II Compilation**. The Quartus II software compiles the design.

> For the Quartus II software to compile automatically, you must have Quartus II Tcl script support enabled. If you are using a version of the Quartus II software that does not support Tcl scripting (e.g., Quartus II Web Edition), you will receive a message saying the Tcl Script Support feature is not available.
>
> If your version of the Quartus II software does not support Tcl scripts, you must perform compilation manually. Complete the remaining steps below and then go to "Compile the Design Manually in the Quartus II Software" on page 46.

8. Click **OK** to close the SignalCompiler block when you are finished.

Manual Synthesis & Compilation

You should use the manual synthesis and compilation flow if you want to:

- Use a synthesis tool other than the Quartus II, LeonardoSpectrum, or Synplify software.
- Use specific synthesis settings in any synthesis tool or use Quartus II settings such as the LogicLock™ feature or timing-driven compilation.

In this tutorial, with the manual flow, you synthesize the SignalCompiler-created **SinGen.vhd** file manually in the LeonardoSpectrum software. The VHDL file contains the RTL design.

> To synthesize and compile the design automatically, go to "Automated Synthesis & Compilation" on page 44.

Manual Synthesis

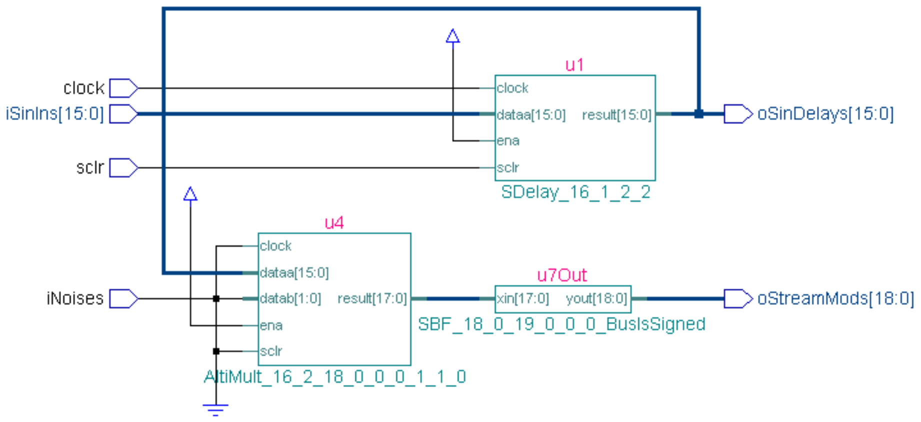
You can synthesize the design files manually by performing the following steps:

1. Double-click the SignalCompiler block in your model.
2. Click the button next to **1 - Convert MDL to VHDL**. SignalCompiler converts the design to VHDL.
3. Click **OK** to close the SignalCompiler block.
4. Open the generated **SinGen.vhd** file in your synthesis tool and perform synthesis.

Refer to the documentation for your synthesis tool for instructions on how to synthesize the design.

Figure 22 shows the amplitude modulation design in the LeonardoSpectrum software.

Figure 22. Amplitude Modulation Design in LeonardoSpectrum



Compile the Design Manually in the Quartus II Software

When you synthesize the design, the software generates atom netlist files (EDIF Netlist File (**.edf**) or Verilog Quartus Mapping File (**.vqm**)) in your working directory. Atoms are parameterized, family-dependent representations of WYSIWYG primitives that correspond to Altera device features, such as logic cells, I/O elements, product terms, and embedded system blocks (ESBs). You can compile the atom netlist files in the Quartus II software to generate the Programmer Object File (**.pof**) used to program an Altera device.

To synthesize and compile the design automatically, go to "Automated Synthesis & Compilation" on page 44.

To compile the design in the Quartus II software, perform the following steps:

1. Start the Quartus II software.
2. Choose **Auxiliary Windows > Tcl Console** (View menu).
3. Change to your DSP Builder working directory.

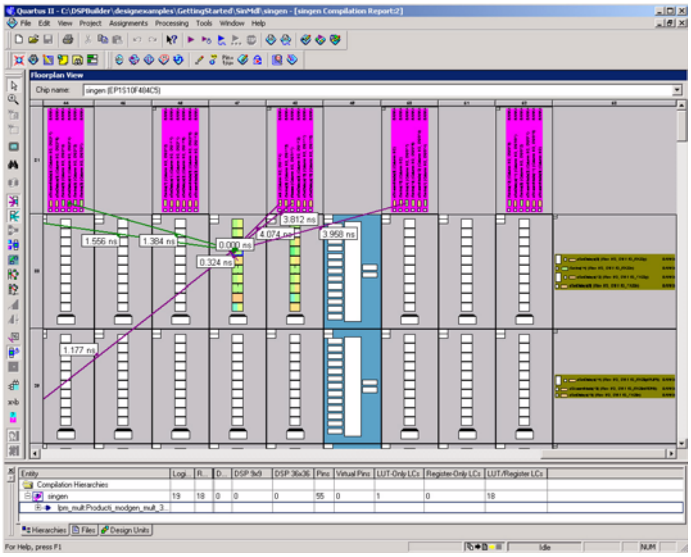
4. Type `source` <*file name*>`_quartus.tcl` ↵. The Quartus II software executes the Tcl script that sets up the project and environment for your design.

5. Choose **Start Compilation** (Processing menu) to begin compilation.

6. Double-click **Floorplan View** in the **SinGen Compilation Report** window to view the results of the compilation.

Figure 23 shows the Quartus II compilation results in the Floorplan View. The shift register of depth 2 and width 16 (Delay block in Simulink) is mapped to 32 APEX 20KE logic elements. 33 inputs/outputs are used to bring data and clock signals into the device.

Figure 23. Quartus II Floorplan View


Quartus II - C:\DSPBuilder\designexamples\GettingStarted\SinMdl\singen - [singen Compilation Report:2]
File Edit View Project Assignments Processing Tools Window Help
Floorplan View
Chip name: singen (EP1S10F484C5)
3.812 ns
4.074 ns
3.958 ns
1.556 ns
1.384 ns
0.000 ns
0.324 ns
1.177 ns
Hierarchies
Files
Design Units
For Help, press F1
Idle
NUM

Entity	Logi...	R...	D...	DSP 9x9	DSP 36x36	Pins	Virtual Pins	LUT-Only LCs	Register-Only LCs	LUT/Register LCs
Compilation Hierarchies										
singen	19	18	0	0	0	55	0	1	0	18
lpm_mult:Producti_modgen_mult_3...										

Create a Quartus II Symbol of Your DSP Builder Design

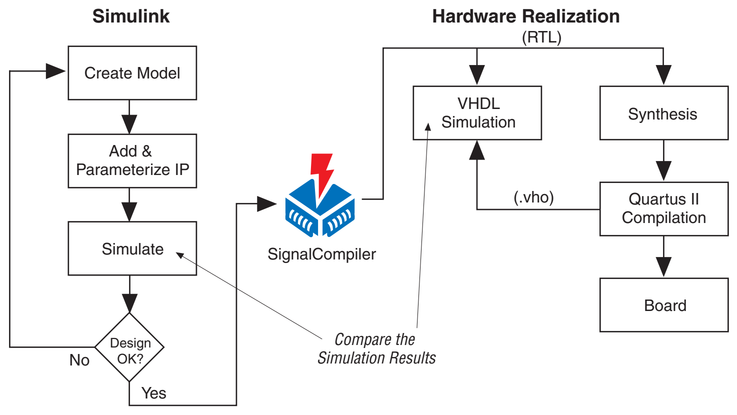
You can create a Quartus II symbol from a DSP Builder design. You can incorporate this symbol into a larger design. After Quartus II compilation, the Quartus II software creates a subdirectory called **atom_netlists** that contains the Verilog Quartus Mapping File (**.vqm**) of the design (e.g., **singen.vqm**). To create a Quartus II symbol from this file, perform the following steps:

1. Open the Quartus II project for the DSP Builder design, e.g., **singen.quartus**.
2. Choose **Open** (File menu)
3. Browse to the **atom_netlists** directory.
4. Open the file <*design name*>.vqm.
5. Choose **Create/Update > Create Symbol Files for Current File** (File menu). The symbol is created and added to your project.

Using MegaCore Functions

For complex signal processing designs, you can incorporate MegaCore blocks (that you install separately) into your model. This section describes how to use MegaCore functions in your model and the simulation options that are available. Figure 24 shows the DSP Builder design flow using IP.

Figure 24. DSP Builder Design Flow Using IP



MegaCore Functions in MATLAB/Simulink

You can download and install DSP MegaCore functions from the Altera web site and use them modularly with DSP Builder. Refer to the MegaCore function's product web page for information on whether you can use the core with DSP Builder. To use a MegaCore function in a new model, perform the following steps:

1. If you have not already done so, download and install the MegaCore function from the Altera web site at http://www.altera.com/IPmegastore.

   Ensure that the MATLAB/Simulink software is not running when you install the core(s).

2. Run MATLAB/Simulink.

3. Create a new model if you have not already done so.

4. Expand the Simulink icon in the MATLAB **Launch Pad** window by clicking the + symbol next to the icon.

5. Double-click the **Library Browser** icon.

6. The **Simulink Library Browser** displays the DSP Builder and MegaCore function(s) you installed.

7. Drag and drop the SignalCompiler block into your design if you have not already done so.

8. If you have not already done so, specify the top-level **.mdl**, which sets your working directory:

   a. Double-click SignalCompiler in your model.

   b. Click the button next to **Select the Top-Level Model File**.

   c. Browse to the directory in which you saved the Model File, e.g., **C:\DSPBuilder\DesignExamples\GettingStarted\my_SinMdl**.

   d. Select the Model File, e.g., **singen.mdl**, and click **Open**.

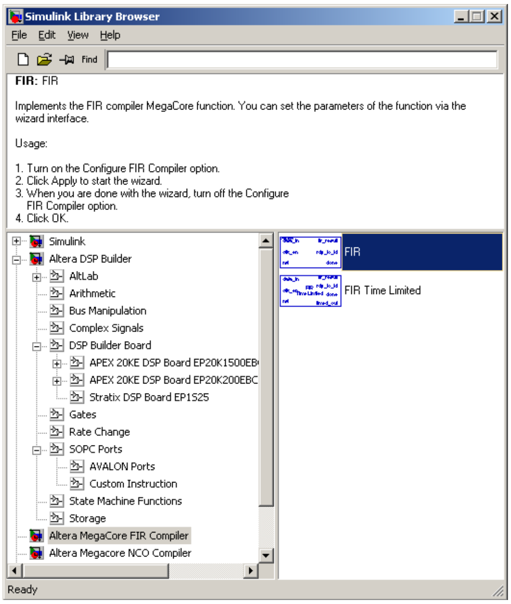
   You must include the SignalCompiler block in your Simulink model file and set the top-level Model File for the MegaCore wizards to operate properly.

9. Click **OK**.

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

11. Click the + icon next to the name of the MegaCore function you want to use so that DSP Builder recognizes the block. See Figure 25.

Figure 25. Simulink Library Browser with IP



FIR: FIR

Implements the FIR compiler MegaCore function. You can set the parameters of the function via the wizard interface.

Usage:

1. Turn on the Configure FIR Compiler option.
2. Click Apply to start the wizard.
3. When you are done with the wizard, turn off the Configure FIR Compiler option.
4. Click OK.

12. Drag and drop the MegaCore block into your model.

13. Double-click the block to launch the MegaCore wizard.

14. Go through the wizard, setting the desired parameters. Refer to the MegaCore function's user guide for more information on how to use the wizard.

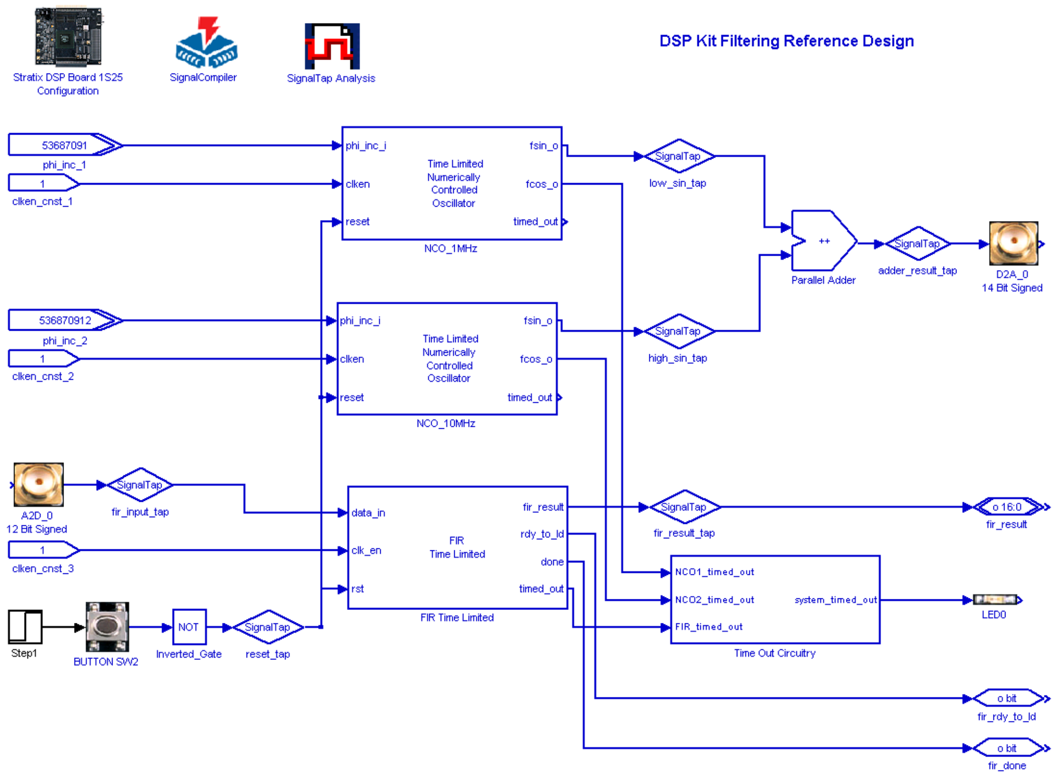
You can download the latest user guides for Altera MegaCore functions from the Literature section on the Altera web site or from the MegaCore function's product web page.

15. Click **Finish** to return to the model file.

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

Figure 26 shows an example design using the NCO Compiler MegaCore function. In this example, the design makes the LEDs on the APEX DSP development board blink. The clock frequency of the board oscillator is converted from MHz to Hz using an NCO and a counter.

Figure 26. Example MegaCore IP Design Using the NCO Compiler



MegaCore Simulation Options

There are four ways to simulate designs that use MegaCore functions:

- *Using the Quartus II software*—You can simulate your design in the Quartus II software before purchasing a license using the OpenCore feature. You can simulate any Altera MegaCore function using this method.

- *Using the ModelSim software and precompiled VHDL models before you have purchased a license for the MegaCore function*—Some MegaCore functions include precompiled VHDL models. With this simulation option, you can combine the precompiled IP models with SignalCompiler-generated VHDL file(s) and Tcl script to simulate your design in the ModelSim software. Refer to "Performing RTL Simulation" on page 40 for information on how to perform the simulation.

- *Using the ModelSim software after you have purchased a license for the MegaCore function*—After you purchase a license for MegaCore functions, you can generate VHDL Output Files (**.vho**) for timing simulation in the ModelSim software. You can simulate any Altera MegaCore function using this method.

  With this simulation option, you must modify the SignalCompiler-generated Tcl script to use the **.vho** files. Refer to Quartus II Help for instructions on how to use **.vho** files for simulation.

- *Using another VHDL simulator with Visual IP models*—Some MegaCore functions include Visual IP models, which you can use to simulate the MegaCore function with your simulator. For more information on using Visual IP models, refer to the *Simulating the Visual IP Models with the ModelSim (PC) Simulator White Paper* or the *Simulating the Visual IP Models with the NC-Verilog, Verilog-XL, VCS, or ModelSim (UNIX) Simulators White Paper*.

You can use any of these simulation models in a testbench to simulate your design.

Refer to the user guide for the MegaCore function for the simulation options that are available for use with DSP Builder.

Performing SignalTap II Logic Analysis

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

- Stratix EP1S25 DSP development board
- APEX DSP development board (starter)
- APEX DSP development board (professional)

To target a specific board, your design must include the board configuration block for the board you want to use and a board connector. You must use the board connector block that corresponds with the appropriate board (the board and connector blocks that work together are located in the same folders.) See "DSP Board Library" on page 159 for more information.

In this design, an LED on the DSP development board board turns on or off based on pressing the 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 output are feed an `OR` gate, which feeds an LED on the DSP development board (starter version). 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 the DSP board. The logic analyzer retrieves the values and displays them in the MATLAB work space.

Refer to "SignalTap II Analysis Block" on page 107 and "Node Block" on page 113 for detailed information on these blocks and their functionality.

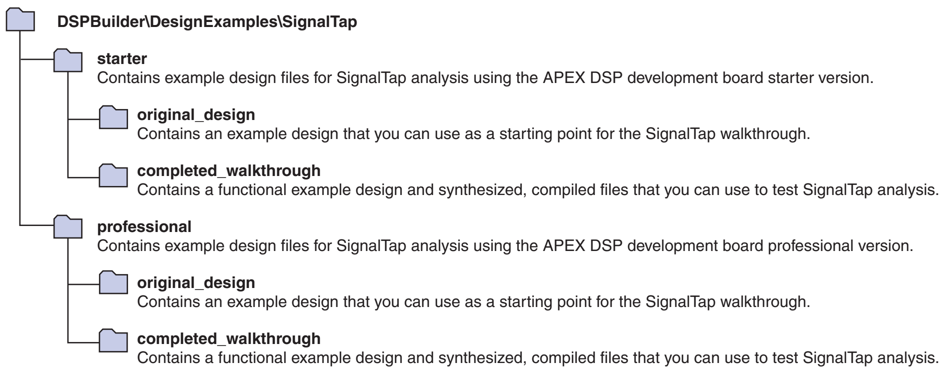
This walkthrough involves the following steps:

1. "Open the Walkthrough Example Design" on page 54
2. "Specify the Nodes to Analyze" on page 56
3. "Turn On the SignalTap II Option in SignalCompiler" on page 58
4. "Generate VHDL, Synthesize, Compile & Download the Design to the DSP Board" on page 59
5. "Specify Trigger Conditions" on page 59
6. "Specify the Radix for the Bus Groups" on page 61
7. "Perform SignalTap II Analysis" on page 61

Open the Walkthrough Example Design

Altera provides the design files for this walkthrough in the directory structure shown in Figure 27. You can start from the design in the **original_design** directory and go through the walkthrough. Alternatively, you can use the design in the **completed_walkthrough** directory and skip to "Specify Trigger Conditions" on page 59.

Figure 27. SignalTap II Design Example Directory Structure



DSPBuilder\DesignExamples\SignalTap

- **starter**
  Contains example design files for SignalTap analysis using the APEX DSP development board starter version.
  - **original_design**
    Contains an example design that you can use as a starting point for the SignalTap walkthrough.
  - **completed_walkthrough**
    Contains a functional example design and synthesized, compiled files that you can use to test SignalTap analysis.
- **professional**
  Contains example design files for SignalTap analysis using the APEX DSP development board professional version.
  - **original_design**
    Contains an example design that you can use as a starting point for the SignalTap walkthrough.
  - **completed_walkthrough**
    Contains a functional example design and synthesized, compiled files that you can use to test SignalTap analysis.

☞ The DSP Builder only supports the SignalTap II embedded logic analyzer for the DSP development boards.

To open the example design:

1. Choose **Open** (File menu) in the MATLAB software.

2. Browse to the directory that contains the file you want to use (see Figure 27).

3. Select the file and click **Open**.

4. If you did not install the DSP Builder into the default location, **c:\DSPBuilder**, perform the following steps to set your working directory.

   a. Double-click the SignalCompiler block in the example design.

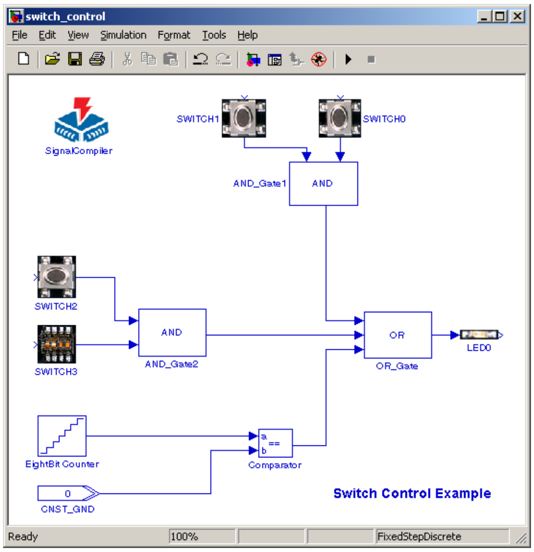
   b. Click the folder icon next to the file name **top.mdl** under **Project Setting Options**.

c. Browse to the directory that contains the example design you opened in step 3.

d. Select the file and click **Open**.

Figure 28 shows the design in the **original_design** directory, which you will use for the following sections.

Figure 28. Starting Point for the SignalTap II Walkthrough



Specify the Nodes to Analyze

In the following steps you will add Node blockNode blocks to the signals (also called nodes) that you want to analyze: the output of each `AND` gate and the output of the incrementer. To add a Node block to a node, perform the following steps:

1. Go to the **AltLab** library in the **Simulink Library Browser**. Refer to the "DSP Builder Tutorial" on page 24 for instructions on accessing the library.

2. Drag and drop a Node block into your design. Position the block so that it is on top of the conection line between the Logical Bit Operator block (AND) and the Logical Bit Operator2 block (OR). Refer to Figure 30 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 to both sides of it.

3. Click the text `SignalTap` under the block icon in your model.

4. Change the block instance name by deleting the text `SignalTap` and typing in the text `firstandout`.

5. Add a Node block between the Logical Bit Operator1 block (AND) and the Logical Bit Operator2 block (OR) and name it `secondandout`.

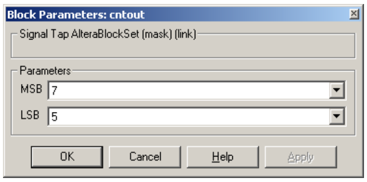
6. Add a Node block between the Increment Decrement block and the Comparator block and name it `cntout`.

7. Double-click the cntout block.

8. Specify that you want to analyze the 3 most significant bits at the output of the incrementer by setting the following parameters. See Figure 29.

   – **MSB:** 7
   – **LSB:** 5

Figure 29. Setting cntout Parameters



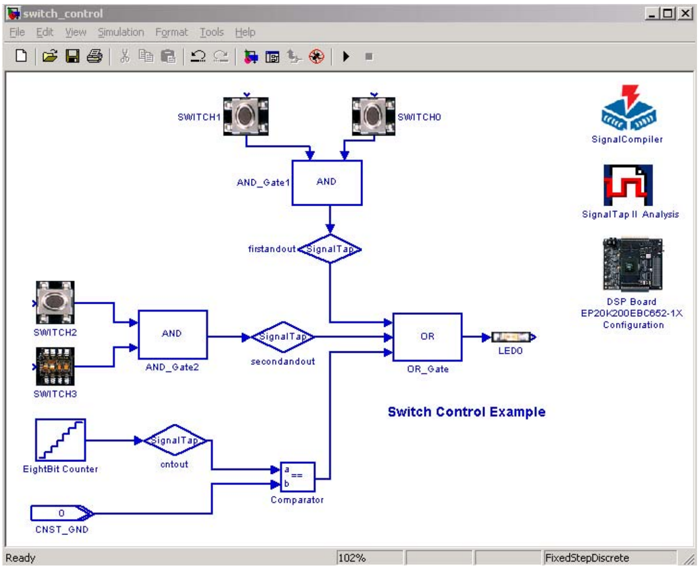
MSB 7
LSB 5

9. Click **OK** to save your settings.

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

Figure 30 shows the completed design.

Figure 30. Completed SignalTap II Design

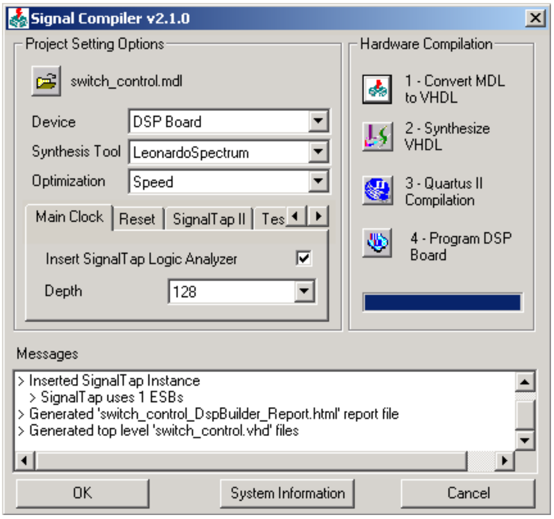


Turn On the SignalTap II Option in SignalCompiler

When you add Node blocks to signals, the block is connected to the input of the SignalTap II embedded logic analyzer, which alters your overall design. Therefore, you must compile the design before you can use the SignalTap II embedded logic analyzer. To compile the design, perform the following steps:

1. Double-click on the SignalCompiler block.
2. Click the **SignalTap II** tab (see Figure 31).

Figure 31. SignalTap II Tab in the SignalCompiler



switch_control.mdl

Device: DSP Board

Synthesis Tool: LeonardoSpectrum

Optimization: Speed

Insert SignalTap Logic Analyzer ✓

Depth: 128

```
> Inserted SignalTap Instance
  > SignalTap uses 1 ESBs
> Generated 'switch_control_DspBuilder_Report.html' report file
> Generated top level 'switch_control.vhd' files
```

3. Turn on the **Insert SignalTap II Logic Analyzer** option. When you turn on this option the SignalCompiler inserts an instance of the SignalTap II embedded logic analyzer into the design.
4. Select a depth of 128 for the SignalTap II embedded logic analyzer sample buffer (i.e., the number of samples stored for each input signal) in the **Sample Depth** list.
5. Click **OK**.

Generate VHDL, Synthesize, Compile & Download the Design to the DSP Board

Generate VHDL, synthesize, compile, and download the design to the DSP board by clicking the buttons under **Hardware Compilation**. Refer to "DSP Builder Tutorial" on page 24 for detailed instructions. The DSP Builder only supports the SignalTap II embedded logic analyzer for the DSP development boards.

Click the button next to **1 - Convert MDL to VHDL** to view the estimated ESBs used to perform SignalTap II analysis. The estimated usage is a function of the number of nodes to be analyzed and the sample depth (see Table 9 on page 112). The **Messages** box displays the ESB usage.

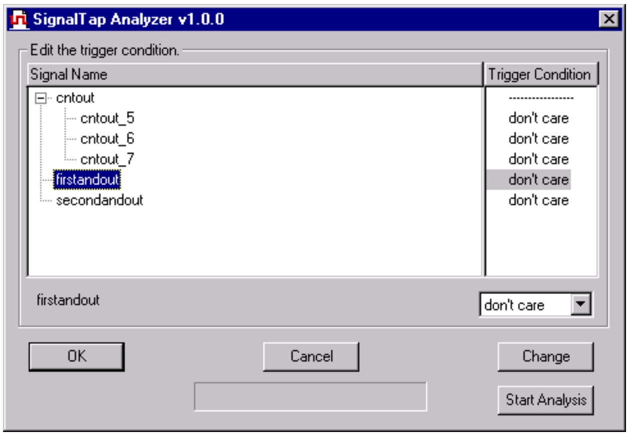
Specify Trigger Conditions

In the **switch_control.mdl** design, specify "Falling edge" as the trigger condition for firstandout and "High" as the trigger condition for `secondandout`. To specify trigger conditions perform the following steps:

If you used the Altera-provided file in the **completed_walkthrough** directory, generate VHDL before attempting to open the SignalTap II Analyzer block by clicking the button next to **1 - Convert MDL to VHDL** in the SignalCompiler. You cannot open the SignalTap II Analyzer block unless you have generated VHDL because the block relies on data files that are created when SignalCompiler generates the VHDL.

1. Double-click the SignalTap II Analysis block. The SignalTap II analyzer displays all of the nodes connected to Node blocks as signals to be analyzed. See Figure 32.

Figure 32. SignalTap II Analyzer


SignalTap Analyzer v1.0.0

Edit the trigger condition.

Signal Name	Trigger Condition
cntout	----------------
cntout_5	don't care
cntout_6	don't care
cntout_7	don't care
firstandout	don't care
secondandout	don't care

firstandout: don't care

2. Specify the trigger condition for the `firstandout` node:

    a. Click `firstandout` under **Signal Name**.

    b. Select **Falling Edge** in the **Trigger Condition** list.

    c. Click **Change**. The condition is updated.

3. Specify the trigger condition **High** for the `secondandout` node.

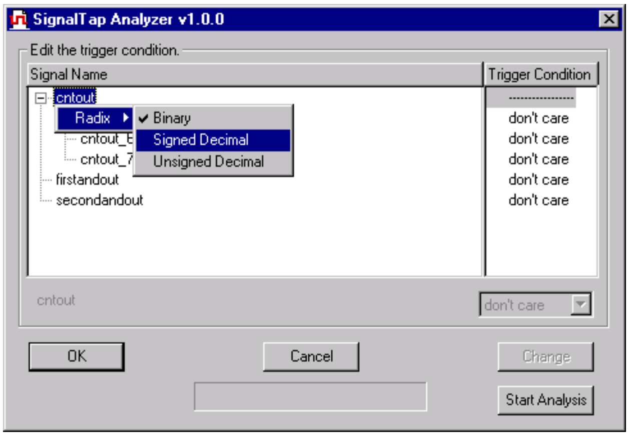
The SignalTap II embedded logic analyzer captures data for analysis when it simultaneously detects all trigger patterns on the input signals. For example, because you specified **Falling Edge** for `firstandout` and **High** for `secondandout` in the **top.mdl** design, the SignalTap II embedded logic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

Specify the Radix for the Bus Groups

In the **top.mdl** design, specify Signed Decimal for the bus group cntout by performing the following steps:

1. Right-click `cntout` under **Signal Name**. A popup menu displays. See Figure 33.

Figure 33. Specifying the Radix



2. Select **Signed Decimal**. The radix is updated.

Perform SignalTap II Analysis

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

1. Click **Start Analysis**.
2. Press switch 0 on the DSP development board to trigger the SignalTap II embedded logic analyzer.
3. Click **OK** in the SignalTap II Analysis block when you are finished. Two MATLAB plots display the captured data: in binary format, Figure 34, and in the radix you specified, .

Figure 34. MATLAB Plot of SignalTap II Analysis (Binary Format)

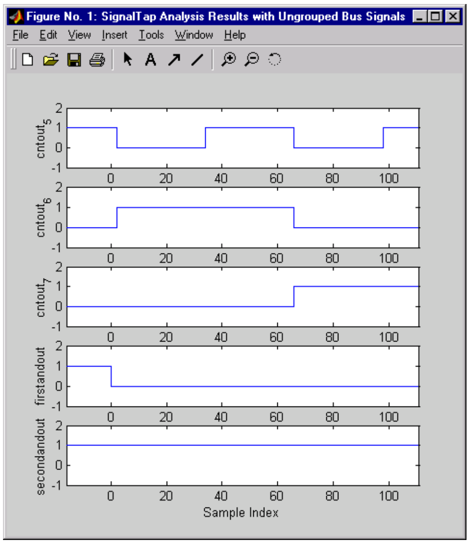


Figure No. 1: SignalTap Analysis Results with Ungrouped Bus Signals
File Edit View Insert Tools Window Help
$cntout_5$
$cntout_6$
$cntout_7$
firstandout
secondandout
Sample Index

Figure 35. MATLAB Plot of SignalTap II Analysis (User-Specified Radix)

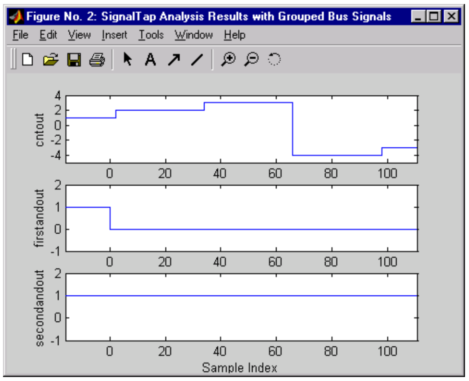


Figure No. 2: SignalTap Analysis Results with Grouped Bus Signals
File Edit View Insert Tools Window Help
cntout
firstandout
secondandout
Sample Index

DSP Builder Design Rules

About the DSP Builder Blocks

DSP Builder blocks are delivered in libraries, classified by functionality. The following libraries are included with the DSP Builder:

- AltLab
- Arithmetic
- Bus Manipulation
- Complex Signals
- DSP Boards
  - DSP Board Stratix EP1S25
  - DSP Board EP20K200EBC652-1X
  - DSP Board EP20K1500EBC652-1X
- Gates
- Rate Change
- SOPC Ports
  - Avalon™ Bus
  - Custom Instruction
- State Machine
- Storage

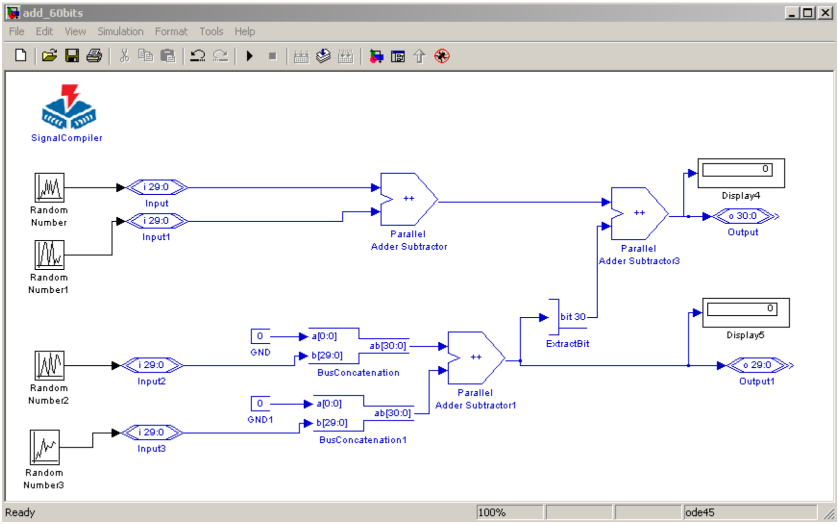
Bit Width Design Rule

You must specify the bit width at the source and destination of the data path. The SignalCompiler block propagates this bit width from the source to the destination through all intermediate blocks. You can optionally specify the bit width of intermediate blocks.

For example, in the amplitude modulation design (see the "DSP Builder Tutorial" on page 24), the SinIn and SinDelay blocks have a bit width of 16. Therefore, SignalCompiler automatically assigns a bit width of 16 to the intermediate Delay block. Each DSP Builder block has specific design rules. The bit width growth rule is described in the documentation for each block.

DSP Builder blocks are of type double, which supports data widths up to 51 bits. If you need more than 51 bits, divide the data bus into multiple slices. For example, Figure 36 shows a 60-bit adder.

Figure 36. 60-Bit Adder with Multiple Bus Slices



Frequency Design Rule

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

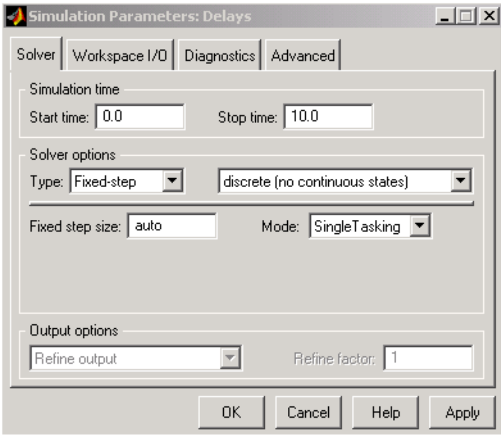
Single Clock Domain

If your design does not contain the PLL block from the Rate Change library, the 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. For these blocks, the clock pin is not graphically displayed in Simulink. However, when SignalCompiler converts the design to VHDL it automatically connects the clock pin of the registered blocks (i.e., the delay block) to the single clock domain of the system.

By default, Simulink does not graphically display the clock enable and synchronous reset input pins of the DSP Builder registered blocks. When SignalCompiler converts the design to VHDL, it connects these pins to V_{CC} and the system reset, respectively. If you turn on the **Use Control Inputs** parameter in the parameter dialog box of each of the DSP Builder registered blocks, you can access and drive the clock enable and synchronous reset input pins graphically in the Simulink software.

To maintain cycle accuracy between the Simulink and the VHDL domain, you should set the solver options to fixed-step discrete and set the mode to single-tasking. see Figure 37.

Figure 37. Simulation Delay Parameters



Simulation Parameters: Delays

Solver | Workspace I/O | Diagnostics | Advanced

Simulation time

Start time: 0.0 Stop time: 10.0

Solver options

Type: Fixed-step discrete (no continuous states)

Fixed step size: auto Mode: SingleTasking

Output options

Refine output Refine factor: 1

From a Simulink simulation standpoint, all DSP Builder blocks (including registered DSP Builder blocks) have an inherited sampling frequency. The sampling period values are propagated from the source block to the destination block via the I/O pins, specifically, the sampling period information is propagated from the output pins of the source block to the input pins of the destination block. If a DSP Builder block has no input pins, (i.e., the Increment or Pattern block), this propagation mechanism cannot take place. Therefore, you must set the sampling period manually in the block parameter. To do so, perform the following steps:

1. Right-click the block.
2. Choose **Block Parameters** from the pop-up menu.
3. In the **S-Function** box, replace the first parameter (by default -1) with the desired sampling period.
4. Click **OK** to save your changes.

When a block has multiple inputs driven by blocks, with different sampling frequencies values, the destination block operates at the fastest rate in the Simulink software.

☞ In the Simulink software, set the **Sample Time Color** (Format menu) to see the sampling frequency of each data path.

Figure 38 shows the design **SingleClockDelay.mdl** (in the **dspbuilder/designexamples/multi_clock** directory). There are many ways to specify the sampling frequency of the sources. Figure 39 shows how you can specify the sample time in a sine wave block using the **Sample Time** edit box.

Figure 38. SingleClockDelay.mdl

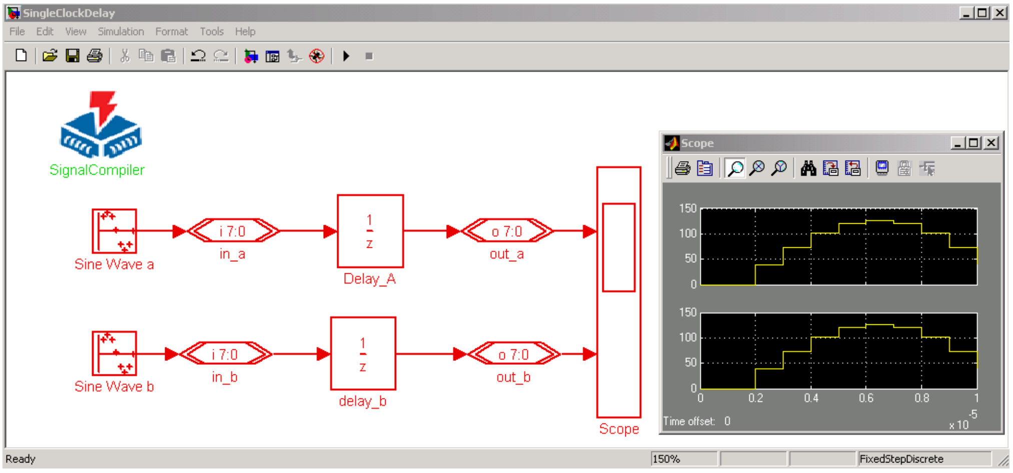
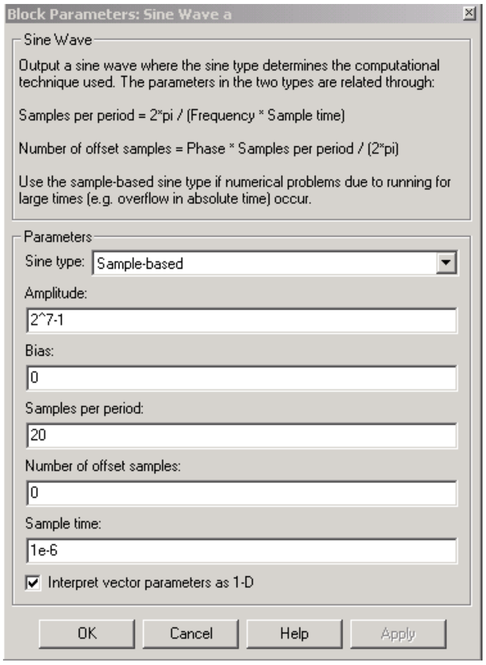


Figure 39. Specify the Sample Time



Block Parameters: Sine Wave a

Sine Wave

Output a sine wave where the 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

Amplitude: 2^7-1

Bias: 0

Samples per period: 20

Number of offset samples: 0

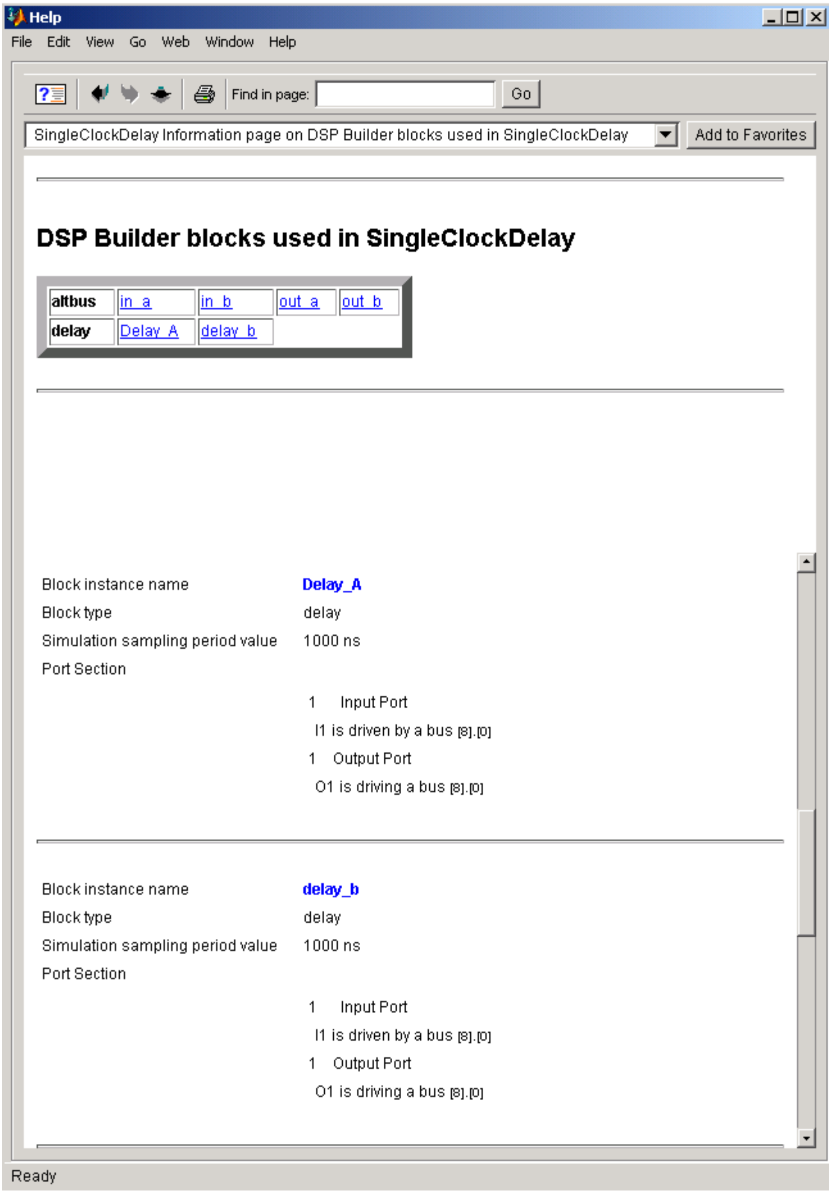
Sample time: 1e-6

☑ Interpret vector parameters as 1-D

In **SingleClockDelay.mdl** for a single clock domain, where the DSP Builder design does not use the PLL block, the sample time of all DSP Builder blocks must be identical and therefore appear with the same color when the sample time color simulation parameter is on. In the **SingleClockDelay.mdl**, the sample time value of the Simulink blocks Sine Wave a and Sine Wave b are set to 1e–6, which is interpreted as 1000 ns by the SignalCompiler.

When you convert the DSP Builder design into RTL using the SignalCompiler, it generates a report file that lists all of the design's DSP Builder blocks with bit-width and simulation-sampling period values (see Figure 40). For single-clock domain designs, the SignalCompiler issues warnings, if it detects multiple simulation-sampling period values. This design has no warnings.

Figure 40. Single-Clock Domain Report File


Help
File Edit View Go Web Window Help
Find in page: Go
SingleClockDelay Information page on DSP Builder blocks used in SingleClockDelay
Add to Favorites
Ready

DSP Builder blocks used in SingleClockDelay

altbus	in_a	in_b	out_a	out_b
delay	Delay_A	delay_b		

Block instance name	Delay_A
Block type	delay
Simulation sampling period value	1000 ns
Port Section	

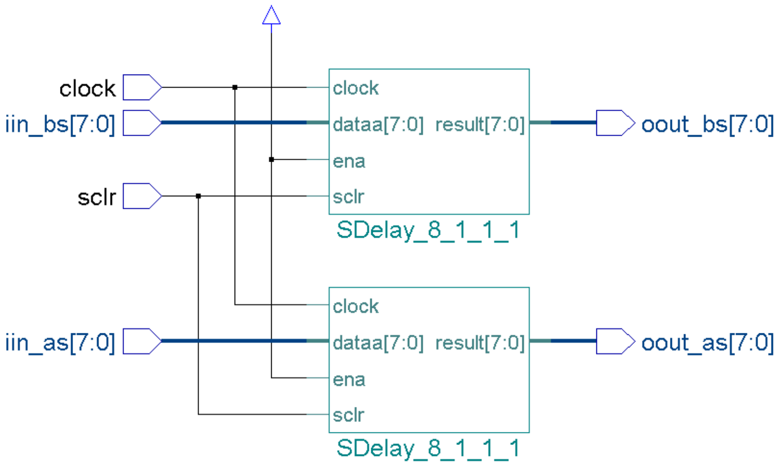
```
1    Input Port
 I1 is driven by a bus [8].[0]
1   Output Port
 O1 is driving a bus [8].[0]
```

Block instance name	delay_b
Block type	delay
Simulation sampling period value	1000 ns
Port Section	

```
1    Input Port
 I1 is driven by a bus [8].[0]
1   Output Port
 O1 is driving a bus [8].[0]
```

Figure 41 shows the LeonardoSpectrum RTL represenation of **SingleClockDelay.mdl**. A single clock input pin is feeding the two registers `Delaya` and `Delayb`.

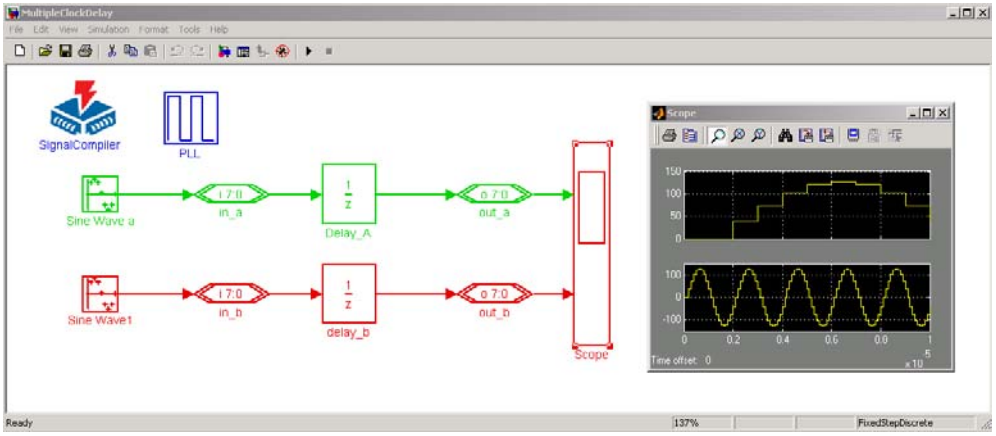
Figure 41. LeonardoSpectrum Representation of SingleClockDelay.mdl



Multiple Clock Domains

If your design contains the PLL block from the rate change library, the DSP Builder registered blocks operate on the positive edge of one of the PLL's output clocks. Figure 42 shows the design **MultilpleClockDelay.mdl** (in the **dspbuilder/designexamples/multi_clock** directory), which shows an example of the multiple-clock domain support.

Figure 42. MultipleClockDelay.mdl

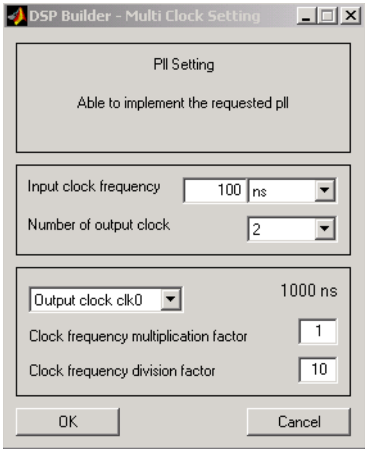


The DSP Builder maps the PLL block to the hardware device PLL; the PLL block supports the following device families:

- *Stratix*—Support for up to 6 output clocks
- *Cyclone*—Support for up to 6 output clocks

The **MultilpleClock.mdl** design shows the PLL block's configuration. The output `pllclock1` is set to `1000` ns; the output `pllclock2` is set to `100` ns. There are several ways to specify that the data path A (shown in green in Figure 42) operates on `pllclock1` and data path B (shown in red in Figure 42) operates on `pllclock2`. In this design the **Sample Time** edit box of the Sine Wave block a and Sine Wave block b are set to 1e-6 and 1e-7, respectively (see Figure 43).

Figure 43. PLL Setting


DSP Builder - Multi Clock Setting

Pll Setting

Able to implement the requested pll

Input clock frequency 100 ns

Number of output clock 2

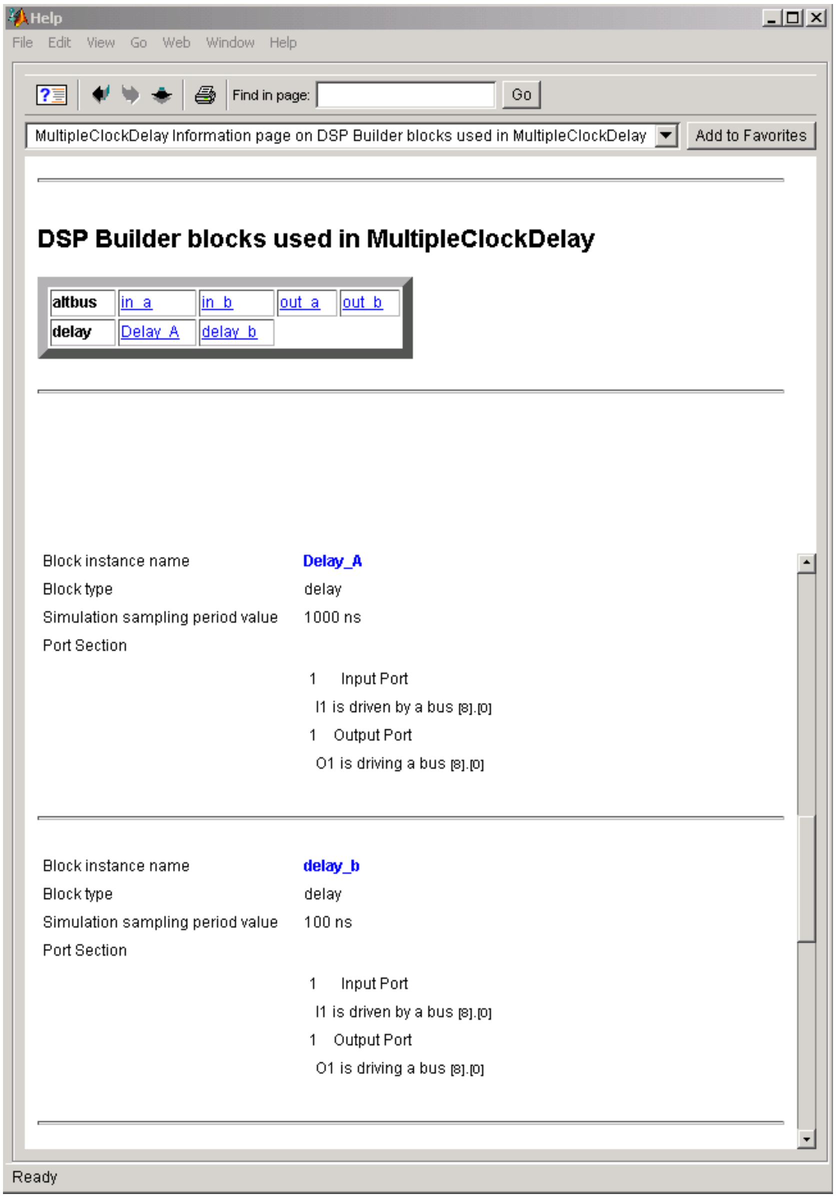
Output clock clk0 1000 ns

Clock frequency multiplication factor 1

Clock frequency division factor 10

When you convert the DSP Builder design into RTL using the SignalCompiler, it generates a report file that lists all of the design's DSP Builder blocks with bit-width and simulation-sampling period values (see Figure 44). For multiple-clock domain designs, the SignalCompiler issues warnings if the DSP Builder block simulation-sampling period values are not equal to one of the PLL's output clock periods.

Figure 44. Multiple-Clock Domain Report File


Help
File Edit View Go Web Window Help
Find in page:
Go
MultipleClockDelay Information page on DSP Builder blocks used in MultipleClockDelay
Add to Favorites
Ready

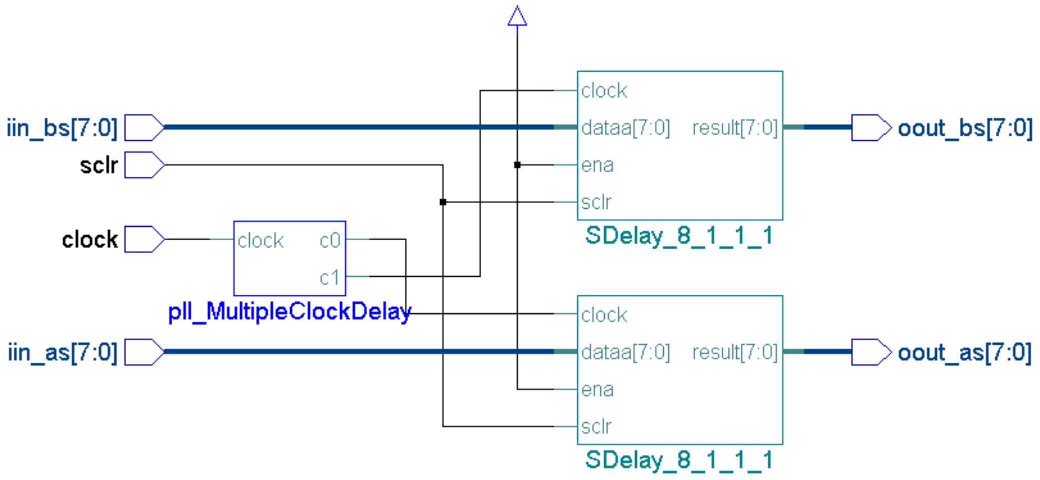
DSP Builder blocks used in MultipleClockDelay

altbus	in_a	in_b	out_a	out_b
delay	Delay_A	delay_b		

```
Block instance name                Delay_A
Block type                         delay
Simulation sampling period value   1000 ns
Port Section
                                    1     Input Port
                                     I1 is driven by a bus [8].[0]
                                    1    Output Port
                                     O1 is driving a bus [8].[0]

Block instance name                delay_b
Block type                         delay
Simulation sampling period value   100 ns
Port Section
                                    1     Input Port
                                     I1 is driven by a bus [8].[0]
                                    1    Output Port
                                     O1 is driving a bus [8].[0]
```

Figure 45 shows the LeonardoSpectrum RTL representation of **MultipleClock.mdl**. The registers `Delaya` and `Delayb` operate on the Stratix PLL's output clocks 1 and 2, respectively.

Figure 45. LeonardoSpectrum Representation of MultipleClockDelay.mdl



Using Advanced PLL Features

The DSP Builder PLL blockset supports the fundamental multiplication and division factor for the PLL. If you want to use other features (phase shift, duty cycle, etc.) of the PLL, you can edit the <*file_name*>**_pll.vhd** generated by SignalCompiler. After doing so, you can make the necessary changes to the PLL component in the top-level VHDL file for your design.

DSP Builder Naming Conventions

DSP Builder block instance names must follow VHDL naming conventions. Some guidelines to follow include:

- Be careful of case. VHDL is not case sensitive. For example the subsystems Mydesign and MYDESIGN will be the same VHDL entity.
- Avoid using VHDL keywords for DSP Builder block instance names.
- Watch for illegal characters. VHDL entity names can only contain a - z, 0 - 9, and underscore (_) characters.
- Begin all instance names with a letter (a - z). VHDL does not allow entity names to begin with non-alphabetic characters.
- Do not use two underscores in succession (__) because it is illegal in VHDL.

☞ White spaces in the names for the blocks/components are ignored when SignalCompiler converts the Model File into VHDL.

Aditionally, SignalCompiler generates a separate VHDL file for each DSP Builder HDL SubSystem block, each containing a single entity/architecture pair. DSP Builder creates VHDL files in which the entity name space is global; therefore, all subsystem names must be unique.

SignalCompiler Design Rules

Refer to "SignalCompiler Block" on page 92 for detailed information on the SignalCompiler block and its design rules.

Fixed-Point Notation

Table 4 describes the fixed-point notation used in the DSP Builder. Refer to "Block I/O Formats" on page 78 for the number formats each block supports.

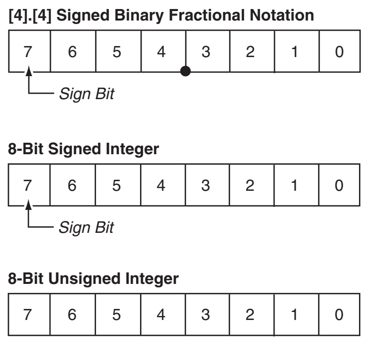
Table 4. Number Types

Number Type	Description	Notation	Simulink-to-HDL Translation (1)
SBF	Signed binary fractional representation; a fractional number.	[L].[R] where: [L] is the number of bits to the left of the binary point; the left-most bit 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 to STD_LOGIC_VECTOR({L + R - 1} DOWNTO 0) in VHDL.
Signed binary	Signed binary; integer.	[L] where: [L] is the number of bits of the signed bus; the sign bit is the left-most bit.	A Simulink signed binary signal $A_{[L]}$ maps to STD_LOGIC_VECTOR({L - 1} DOWNTO 0) in VHDL.
Unsigned binary	Unsigned binary; integer.	[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) in VHDL.
Single bit	Integer that takes the values 1 or 0.	[1]	A Simulink single bit signal maps to a STD_LOGIC signal in VHDL.

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

Figure 46 graphically compares the signed binary fractional, signed binary, and unsigned binary number formats.

Figure 46. Number Format Comparison



Binary Point Location in SBF

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 performance.

Choosing a large number of bits yields excellent performance—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/performance trade-off. The DSP Builder speeds up the design cycle by enabling simulation with fixed-point and floating-point signals in the same environment.

The AltBus 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) on page 76). 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, i.e., 2 bits on the left side of the binary point and two bits on the right side, the numbers are represented as:

Simulink	This signed integer is interpreted as –0.75
VHDL	This signed STD_LOGIC_VECTOR is interpreted as –3

From a system-level analysis standpoint, multiplying a number by –0.75 or –3 is obviously very different, especially when looking at the bit width growth. In the one case the multiplier output bus grows on the MSB, in the other case the multiplier output bus grows on the LSB.

In both cases the binary numbers are identical. However, the location of the binary point affects how a simulator formats the signal representation. For complex systems, you can adjust the binary point location to define the signal range and the area of interest.

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

Block I/O Formats

Table 5 describes the I/O formats for the DSP Builder blocks. The HDL SubSystem, SubSystemBuilder, and SignalCompiler blocks do not have I/O formats. Refer to the following notation conventions for Table 5:

- In the [L].[R] notation, [L] is the number of bits on the left side of the binary point and [R] is the number of bits on the right side of the binary point.
- $I1_{[L].[R]}$ is an input port.
- $O1_{[L].[R]}$ is an output port.
- For signed or unsigned integers R = 0, i.e., [L].[0].
- For signed integers and signed binary fractional numbers, the MSB bit is the sign bit.
- For single bits, R = 0, i.e., [1] is a single bit.
- *Explicit* means that the port bit width information is a block parameter.
- *Implicit* means that the port bit width information is set by the data path bit with propagation mechanism. If you want to specify the bus format of an implicit input port, use a BusConversion block to set the width.

Table 5. Block I/O Formats (Part 1 of 5)

Block	I/O	Simulink	VHDL	Type
Arithmetic				
Comparator	I	$I1_{[L1].[R1]}$ $I2_{[L2].[R2]}$	I1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0) I2: in STD_LOGIC_VECTOR({L2 + R2 - 1} DOWNTO 0)	Implicit Implicit
	O	$O1_{[1]}$	O1: out STD_LOGIC	Implicit
Divider	I	$I1_{[L].[R]}$ $I2_{[L].[R]}$	I1: in STD_LOGIC_VECTOR({L + R - 1} DOWNTO 0) I2: in STD_LOGIC_VECTOR({L + R - 1} DOWNTO 0)	Explicit Explicit
	O	$O1_{[L].[R]}$ $O2_{[L].[R]}$	O1: out STD_LOGIC_VECTOR({L + R - 1} DOWNTO 0) O2: out STD_LOGIC_VECTOR({L + R - 1} DOWNTO 0)	Explicit Explicit
Gain	I	$I1_{[L1].[R1]}$	I1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0)	Implicit
	O	$O1_{[L1 + LK].2*max(R1,RK)]}$ where K is the gain constant with the format $K_{[LK].[RK]}$	O1: out STD_LOGIC_VECTOR({L1 + LK + 2*max(R1,RK) - 1} DOWNTO 0)	Implicit
Increment Decrement	I	$I1_{[1]}$	I1: in STD_LOGIC	Explicit - Optional
		$I2_{[1]}$	I2: in STD_LOGIC	Explicit - Optional
	O	$O1_{[LP].[RP]}$	O1: out STD_LOGIC_VECTOR({LP + RP - 1} DOWNTO 0)	Explicit
Magnitude	I	$I1_{[L1].[R1]}$	I1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0)	Implicit
	O	$O1_{[L1 + 1].[R1]}$	O1: in STD_LOGIC_VECTOR({L1 + R1} DOWNTO 0)	Implicit
Multiply Accumulate	I	$I1_{[L1].[R1]}$ $I2_{[L2].[R2]}$	I1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0) I2: in STD_LOGIC_VECTOR({L2 + R2 - 1} DOWNTO 0)	Implicit
	O	$O1_{[LO].[RO]}$	O1: out STD_LOGIC_VECTOR({L0 + R0 - 1} DOWNTO 0)	Explicit
Multiply Add	I	$I1_{[L1].[R1]}$ .... $Ii_{[L1].[R1]}$ ... $In_{[L1].[R1]}$ where $3 < n < 9$	I1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0) … Ii: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0) …. In: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0) where $3 < n < 9$	Explicit
	O	$O1_{2 \times [L1]+ ceil(log2(n)).2 \times [R1]}$	O1: out STD_LOGIC_VECTOR({(2 x L1) + ceil(log2(N)) + (2 x R1) - 1} DOWNTO 0)	Implicit
Parallel Adder Subtractor	I	$I1_{[L1].[R1]}$ .... $Ii_{[LI].[RI]}$ ... $In_{[LN].[RN]}$	I1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0) … Ii: in STD_LOGIC_VECTOR({LI + RI - 1} DOWNTO 0) …. In: in STD_LOGIC_VECTOR({LN + RN - 1} DOWNTO 0)	Implicit
	O	$O1_{[max(LI) + ceil(log2(N))].[max(RI)]}$	O1: out STD_LOGIC_VECTOR({max(LI) + ceil(log2(N)) + max(RI) - 1} DOWNTO 0)	Implicit
Product	I	$I1_{[L1].[R1]}$ $I2_{[L2].[R2]}$	I1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0) I2: in STD_LOGIC_VECTOR({L2 + R2 - 1} DOWNTO 0)	Implicit
	O	$O1_{[L1 + L2].[2 \times max(R1,R2)]}$	O1: out STD_LOGIC_VECTOR({L1 + L2 + 2 x max(R1,R2) - 1} DOWNTO 0)	Implicit
Bus Manipulation				
AltBus	I	$I1_{[L1].[R1]}$		Implicit - Optional
	O	$O1_{[LP].[RP]}$	O1: out STD_LOGIC_VECTOR({LP + RP - 1} DOWNTO 0)	Explicit

Table 5. Block I/O Formats (Part 2 of 5)

Block	I/O	Simulink	VHDL	Type
BusBuild	I	$I1_{[1]}$ … $INPi_{[1]}$ …. $INPn_{[1]}$	I1: in STD_LOGIC … INPi: in STD_LOGIC …. INPn: in STD_LOGIC	Explicit
	O	$O1_{[LP].[RP]}$ with LP + RP = N where N is the number of inputs	O1: out STD_LOGIC_VECTOR({LP + RP - 1} DOWNTO 0)	Explicit
Binary Point Casting	I	$I1_{[Li].[Ri]}$	I1: in STD_LOGIC_VECTOR({Li + Ri - 1} DOWNTO 0) I2: in STD_LOGIC_VECTOR({L2 - 1} DOWNTO 0)	Explicit
	O	$O1_{[LO].[RO]}$ where LO + RO - Li + Ri	O1: out STD_LOGIC_VECTOR({LO + RO - 1} DOWNTO 0)	Explicit
Bus Concatenation	I	$I1_{[L1].[0]}$ $I2_{[L2].[0]}$	I1: in STD_LOGIC_VECTOR({L1 - 1} DOWNTO 0) I2: in STD_LOGIC_VECTOR({L2 - 1} DOWNTO 0)	Explicit
	O	$O1_{[L1 + L2].[0]}$	O1: out STD_LOGIC_VECTOR({L1 + L2 - 1} DOWNTO 0)	Explicit
Bus Conversion	I	$I1_{[LPi].[RPi]}$	I1: in STD_LOGIC_VECTOR({LPi + RPi - 1} DOWNTO 0)	Explicit
	O	$O1_{[LPO].[RPO]}$	O1: out STD_LOGIC_VECTOR({LPO + LPO - 1} DOWNTO 0)	Explicit
Extract Bit	I	$I1_{[L1].[R1]}$	I1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0)	Explicit
	O	$O1_{[1]}$	O1: out STD_LOGIC	Explicit
GND	O	$O1_{[1]}$	O1: out STD_LOGIC	Explicit
VCC	O	$O1_{[1]}$	O1: out STD_LOGIC	Explicit
Complex Signals *(1)*				
Butterfly Operator	I	$I1_{Real([Li].[0])Imag([Li].[0])}$ $I2_{Real([Li].[0])Imag([Li].[0])}$ $I3_{Real([Li].[0])Imag([Li].[0])}$	I1Real: in STD_LOGIC_VECTOR({Li - 1} DOWNTO 0) I1Imag: in STD_LOGIC_VECTOR({Li - 1} DOWNTO 0) I2Real: in STD_LOGIC_VECTOR({Li - 1} DOWNTO 0) I2Imag: in STD_LOGIC_VECTOR({Li - 1} DOWNTO 0) I3Real: in STD_LOGIC_VECTOR({Li - 1} DOWNTO 0) I3Imag: in STD_LOGIC_VECTOR({Li - 1} DOWNTO 0)	Explicit
	O	$O1_{Real([Lo].[0])Imag([Li].[0])}$ $O2_{Real([Lo].[0])Imag([Li].[0])}$	O1Real: in STD_LOGIC_VECTOR({Lo - 1} DOWNTO 0) O1Imag: in STD_LOGIC_VECTOR({Lo - 1} DOWNTO 0) O2Real: in STD_LOGIC_VECTOR({Lo - 1} DOWNTO 0) O2Imag: in STD_LOGIC_VECTOR({Lo - 1} DOWNTO 0)	Explicit
Complex AddSub	I	$I1_{Real([L1].[R1])Imag([L1].[R1])}$ $I2_{Real([L2].[R2])Imag([L2].[R2])}$	I1Real: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0) I1Imag: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0) I2Real: in STD_LOGIC_VECTOR({LP2 + RP2 - 1} DOWNTO 0) I2Imag: in STD_LOGIC_VECTOR({LP2 + RP2 - 1} DOWNTO 0)	Implicit
	O	$O1_{Real(max(L1,L2) + 1),(max(RI,R2) + 1)Imag(max(L1,L2) + 1),(max(RI,R2) + 1)}$	O1Real: in STD_LOGIC_VECTOR({max(LI,L2) + max(RI,R2)} DOWNTO 0) O1Imag: in STD_LOGIC_VECTOR({max(LI,L2) + max(RI,R2)} DOWNTO 0)	Implicit

Table 5. Block I/O Formats (Part 3 of 5)

Block	I/O	Simulink	VHDL	Type
Complex Product	I	$I1_{Real([L1].[R1])Imag([L1].[R1])}$ $I2_{Real([L2].[R2])Imag([L2].[R2])}$	I1Real: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0) I1Imag: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0) I2Real: in STD_LOGIC_VECTOR({LP2 + RP2 - 1} DOWNTO 0) I2Imag: in STD_LOGIC_VECTOR({LP2 + RP2 - 1} DOWNTO 0)	Implicit
	O	$O1_{Real(2 x max(LI,L2)),(2 x max(RI, R2))Imag(2 x max(LI,L2)),(2 x max(RI, R2))}$	O1Real: in STD_LOGIC_VECTOR({(2 x max(LI,L2)) + (2 x max(RI,R2)) -1} DOWNTO 0) O1Imag: in STD_LOGIC_VECTOR({(2 x max(LI,L2)) + (2 x max(RI,R2)) -1} DOWNTO 0)	Implicit
Complex Multiplier	I	$I1_{Real([L1].[R1])Imag([L1].[R1])}$ $I2_{Real([L2].[R2])Imag([L2].[R2])}$ $I3_{[1]}$	I1Real: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0) I1Imag: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0) I2Real: in STD_LOGIC_VECTOR({LP2 + RP2 - 1} DOWNTO 0) I2Imag: in STD_LOGIC_VECTOR({LP2 + RP2 - 1} DOWNTO 0) I3: in STD_LOGIC	Implicit
	O	$O1_{Real(max(L1,L2)),(max(RI,R2))Imag(max(L1,L2)),(max(RI,R2))}$	O1Real: in STD_LOGIC_VECTOR({max(LI,L2) + max(RI,R2) - 1} DOWNTO 0) O1Imag: in STD_LOGIC_VECTOR({max(LI,L2) + max(RI,R2) - 1} DOWNTO 0)	Implicit
Complex Conjugate	I	$I1_{Real([L1].[R1])Imag([L1].[R1])}$	I1Real: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0) I1Imag: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0)	Implicit
	O	$O1_{Real([L1] + 1.[R1])Imag([L1] + 1.[R1])}$	O1Real: in STD_LOGIC_VECTOR({LP1 + RP1} DOWNTO 0) O1Imag: in STD_LOGIC_VECTOR({LP1 + RP1} DOWNTO 0)	Implicit
Complex Delay	I	$I1_{Real([L1].[R1])Imag([L1].[R1])}$	I1Real: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0) I1Imag: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0)	Implicit
	O	$O1_{Real([L1].[R1])Imag([L1].[R1])}$	O1Real: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0) O1Imag: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0)	Implicit
Complex to Real-Imag	I	$I1_{Real([L1].[R1])Imag([L1].[R1])}$	I1Real: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0) I1Imag: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0)	Implicit
	O	$O1_{Real([L1].[R1])}$ $O2_{Imag([L1].[R1])}$	O1Real: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0) O2Imag: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0)	Explicit
Real-Imag to Complex	I	$I1_{Real([L1].[R1])}$ $I2_{Imag([L1].[R1])}$	I1Real: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0) I1Imag: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0)	Implicit
	O	$O1_{Real([L1].[R1])Imag([L1].[R1])}$	O1Real: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0) O1Imag: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0)	Explicit
Gates				
Case Statement	I	$I1_{[L1].[R1]}$	I1: in STD_LOGIC_VECTOR({LP1 + RP1 - 1} DOWNTO 0)	Implicit
	O	$O1_{[1]}$ … $OUTi_{[1]}$ …. $OUTn_{[1]}$	O1: out STD_LOGIC … OUTi: out STD_LOGIC …. OUTn: out STD_LOGIC	Explicit

Table 5. Block I/O Formats (Part 4 of 5)

Block	I/O	Simulink	VHDL	Type
If Statement	I	$I1_{[L1].[R1]}$ .... $INPi_{[LI].[RI]}$ ... $INPn_{[LN].[RN]}$	I1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0) … INPi: in STD_LOGIC_VECTOR({LI + RI - 1} DOWNTO 0) …. INPn: in STD_LOGIC_VECTOR({LN + RN - 1} DOWNTO 0)	Implicit
	O	$O1_{[1]}$ $O2_{[1]}$	O1: out STD_LOGIC O2: out STD_LOGIC	Explicit
Logical Bit Operator	I	$I1_{[1]}$ … $INPi_{[1]}$ …. $INPn_{[1]}$	I1: in STD_LOGIC … INPi: in STD_LOGIC …. INPn: in STD_LOGIC	Explicit
	O	$O1_{[1]}$	O1: out STD_LOGIC	Explicit
Logical Bus Operator	I	$I1_{[L1].[R1]}$	I1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0)	Explicit
	O	$O1_{[L1].[R1]}$	O1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0)	Explicit
LUT	I	$I1_{[L1].[0]}$	I1: in STD_LOGIC_VECTOR({L1 - 1} DOWNTO 0)	Explicit
	O	$O1_{[LPO].[RPO]}$	O1: out STD_LOGIC_VECTOR({LPO + LPO - 1} DOWNTO 0)	
n-to-1 Multiplexer	I	$IS_{[LS].[0]}$ (select input) $I1_{[L1].[R1]}$ .... $Ii_{[LI].[RI]}$ ... $In_{[LN].[RN]}$ (n = number of inputs)	IS: in STD_LOGIC_VECTOR({LS - 1} DOWNTO 0) I1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0) … Ii: in STD_LOGIC_VECTOR({LI + RI - 1} DOWNTO 0) …. In: in STD_LOGIC_VECTOR({LN + RN - 1} DOWNTO 0)	Implicit
	O	$O1_{[max(LI)].[max(RI)]}$ with $(0 < I < N + 1)$	O1: out STD_LOGIC_VECTOR({max(LI)) + max(RI) - 1} DOWNTO 0)	Implicit
Rate Change				
Multi-Rate DFF	I	$I1_{[L].[R]}$	I1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0)	Implicit
	O	$O1_{[L].[R]}$	O1: out STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0)	Implicit
Tsamp	I	$I1_{[L].[R]}$	I1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0)	Implicit
	O	$O1_{[L].[R]}$	O1: out STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0)	Implicit
Storage				
Delay	I	$I1_{[L1].[R1]}$	I1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0)	Implicit
	O	$O1_{[L1].[R1]}$	O1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0)	Implicit
Down Sampling	I	$I1_{[L1].[R1]}$	I1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0)	Implicit
	O	$O1_{[L1].[R1]}$	O1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0)	Implicit
Dual Port RAM	I	$I1_{[L1].[R1]}$ $I2_{[L2].[0]}$ $I3_{[L2].[0]}$ $I4_{[1]}$	I1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0) I2: in STD_LOGIC_VECTOR({L2 - 1} DOWNTO 0) I3: in STD_LOGIC_VECTOR({L3 - 1} DOWNTO 0) I4: in STD_LOGIC	Explicit
	O	$O1_{[L1].[R1]}$	O1: out STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0)	Explicit

Table 5. Block I/O Formats (Part 5 of 5)

Block	I/O	Simulink	VHDL	Type
Parallel to Serial	I	I1$_{[L1].[R1]}$ I2$_{[1]}$ I3$_{[1]}$	I1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0) I2: in STD_LOGIC I3: in STD_LOGIC	Explicit
	O	O1$_{[1]}$	O1: out STD_LOGIC	Explicit
Pattern	I	I1$_{[1]}$	I1: in STD_LOGIC	Explicit - Optional
		I2$_{[1]}$	I2: in STD_LOGIC	Explicit - Optional
	O	O1$_{[1]}$	O1: out STD_LOGIC	Explicit
ROM EAB	I	I1$_{[L1].[0]}$	I1: in STD_LOGIC_VECTOR({L1 - 1} DOWNTO 0)	Explicit
	O	O1$_{[LPO].[RPO]}$	O1: out STD_LOGIC_VECTOR({LPO + LPO - 1} DOWNTO 0)	Explicit
Serial to Parallel	I	I1$_{[1]}$ I2$_{[1]}$ I3$_{[1]}$	I1: in STD_LOGIC I2: in STD_LOGIC I3: in STD_LOGIC	Explicit
	O	O1$_{[L1].[R1]}$	O1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0)	Explicit
Shift Taps	I	I1$_{[L1].[R1]}$	I1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0)	Implicit
	O	O1$_{[L1].[R1]}$ .... Oi$_{[L1].[R1]}$ ... On$_{[L1].[R1]}$ (n = number of taps)	O1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0) ... Oi: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0) .... On: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0) (n = number of taps)	Implicit
Up Sampling	I	I1$_{[L1].[R1]}$	I1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0)	Implicit
	O	O1$_{[L1].[R1]}$	O1: in STD_LOGIC_VECTOR({L1 + R1 - 1} DOWNTO 0)	Implicit

Note:
(1) The bit widths you set for complex numbers apply to both the real and imaginary parts.

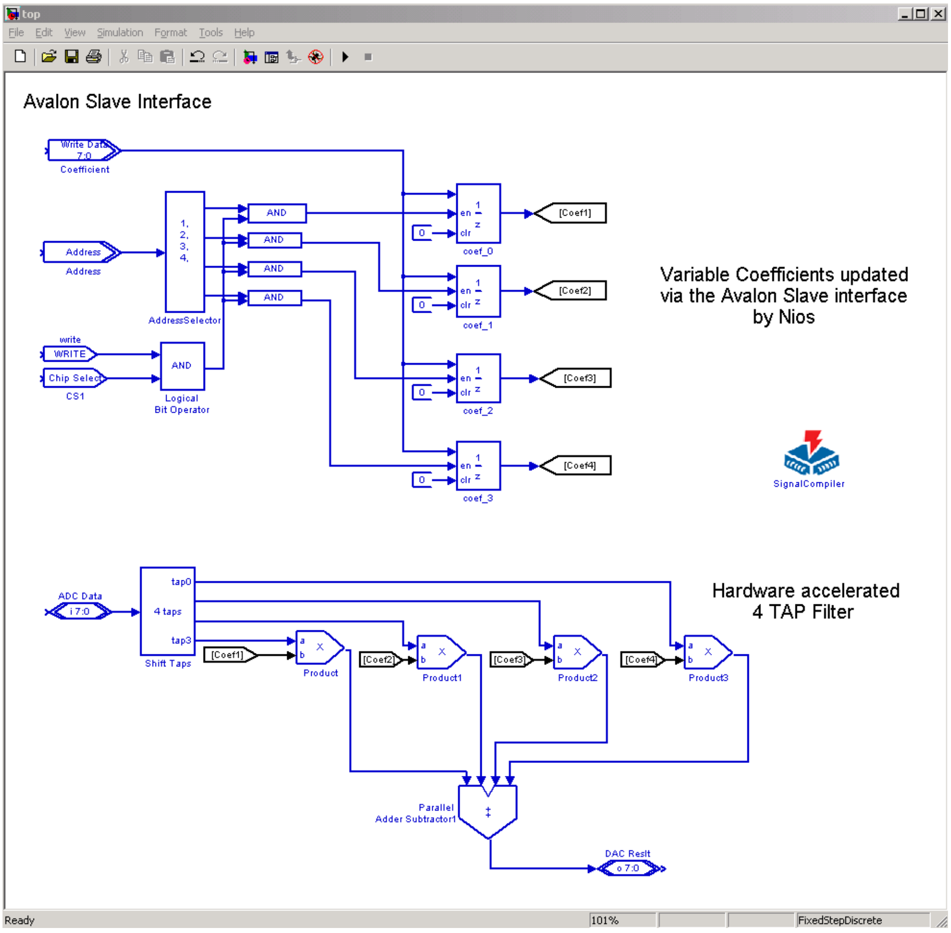
Goto & From Block Support

DSP Builder supports the Goto and From blocks from the generic Simulink library (Signal and System Folder). These blocks are used for large fan-out signals and enhance the diagram clarity. The DSP Builder supports the Goto and From mode **Tag Visibilty = local**.

Goto and From blocks must be in the same hierarchical level.

Figure 47 shows an example of the Goto and From blocks. The Goto blocks ([coef1], [coef2], [coef3], [coef4]) are used respectively with the From blocks ([coef1], [coef2], [coef3], [coef4]), which are connected to the Product blocks.

Figure 47. Goto & From Block Example



MegaCore Function Support

Altera MegaCore functions are rigorously tested and optimized for the highest performance and lowest cost in Altera programmable logic devices (PLDs). All MegaCore functions are fully parameterizable through Altera's unique MegaWizard® Plug-In Manager.

MegaCore functions support Altera's free IP evaluation features, which allow you to verify the functionality and timing of a function prior to purchasing a license.

- The OpenCore® evaluation feature lets you test-drive IP cores for free using the Quartus® II software; however, you cannot generate device programming files to test the core in hardware.

- The OpenCore Plus evaluation feature enhances the OpenCore evaluation feature by supporting free hardware evaluation. This feature allows you to generate time-limited programming files for a design that includes Altera MegaCore functions. With these files, you can perform board-level design verification before deciding to purchase licenses for the MegaCore functions.

With both evaluation features, you only need to purchase a license when you are completely satisfied with a core's functionality and performance, and would like to take your design to production. DSP Builder supports a variety of Altera DSP cores such as FIR Compiler, Reed-Solomon Compiler, and IIR Compiler.

Refer to the IP MegaStore on the Altera web site for a listing of which cores DSP Builder supports.

To use a core with the DSP Builder, download a DSP core from the Altera web site at http://www.altera.com/IPmegastore and install it on your PC. When you restart the MATLAB software, the DSP Builder automatically detects the new core and adds it to the **Simulink Library Browser**. Refer to the core's product web page for information on whether it works modularly with DSP Builder.

Some IP core features are not available for use with the DSP Builder. Refer to the core user guide for a description of which features are unavailable.

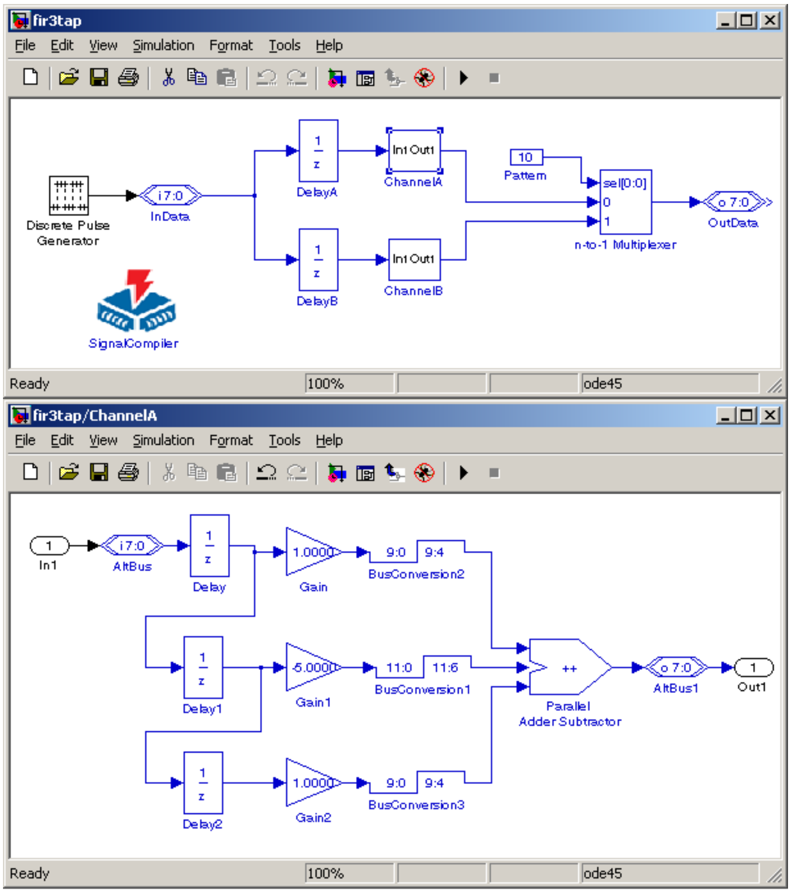
Refer to "Using MegaCore Functions" on page 48 for more information on using IP in your model.

Hierarchical Design

The DSP Builder supports hierarchical design via the subsystem mechanism available in the Simulink software. You define the boundaries of each hierarchical level by connecting the Altera AltBus block to the Simulink Input/Output block. The SignalCompiler block preserves the hierarchy structure in the VHDL design and each Simulink Model File (**.mdl**) hierarchical level translates into one VHDL file. Refer to "DSP Builder Naming Conventions" on page 75 for information on naming HDL subsystem instances.

Figure 48 illustrates a hierarchy for the sample design **fir3tap.mdl**, which implements two FIR filters as described in "Data Width Propagation" on page 95.

Figure 48. Hierarchical Design Example

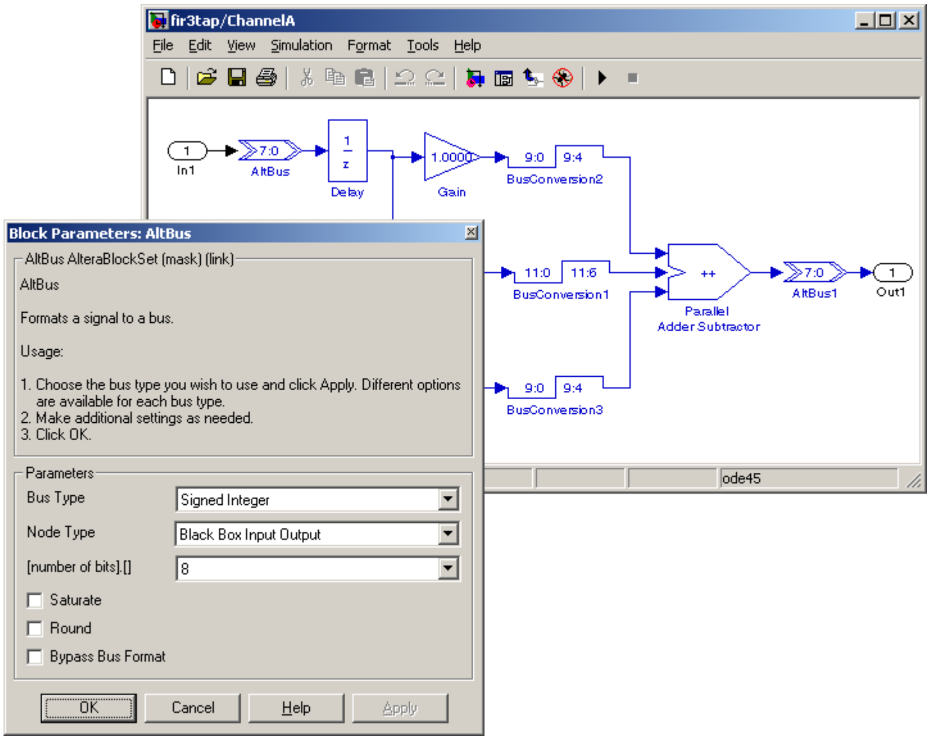


Black Boxing

You can add your own VHDL code to the design and specify which subsystem block(s) should be translated into VHDL by the SignalCompiler. This process, called black boxing, uses the AltBus block in Black Box Input Output mode by setting the AltBus **Node Type** parameter to **Black Box Input Output**. When processing SubSystem 1 in Figure 49, SignalCompiler replaces SubSystem 1 with a black box in the VHDL design. Figure 49 shows the parameter settings for creating a black box.

Refer to “AltBus Black Box Input Output Mode” on page 136 for more information on using the AltBus block to create a black box.

Figure 49. Using a Black Box



Block Parameters: AltBus

AltBus AlteraBlockSet (mask) (link)

AltBus

Formats a signal to a bus.

Usage:

1. Choose the bus type you wish to use and click Apply. Different options are available for each bus type.
2. Make additional settings as needed.
3. Click OK.

Parameters

Bus Type	Signed Integer
Node Type	Black Box Input Output
[number of bits].[]	8

☐ Saturate

☐ Round

☐ Bypass Bus Format

If you use a MegaCore IP block in your model, the IP block is treated as a black box when SignalCompiler generates HDL output files for synthesis. Refer to "Using MegaCore Functions" on page 48 for more information on using IP.

To add your own customized entity to a DSP Builder design, perform the following steps:

1. Define a customized entity in an HDL file.

2. Declare synchronous clear (`sclr`) and clock as inputs in the entity, even if your design uses a synchronous reset and a clock. These inputs are routed to a global clock pin and a global synchronous clear pin. If the entity does not need a clock and/or a global synchronous clear, you should still declare these signals as inputs and leave them unconnected.

☞ To map the clock port of the back boxed VHDL code to the DSP Builder global system clock, you must name the Altbus input block (configured as **Black Box Input Output**) `simulink_clock`.

3. Create an **.mdl** and save it into the same directory as the HDL file you created in step 1.

4. Add a SubSystem block to the **.mdl** and give it the same name as the customized entity.

5. Assign the inputs and outputs of the SubSystem block so that they have the same name as the inputs and outputs of the customized entity. You do not have to create inputs in the SubSystem block for `sclr` and the clock.

6. In the SubSystem block, connect all of the inputs and outputs to Altbus blocks of type **Black Box Input Output.**

7. In the SubSystem block, describe the functionality of the customized entity using blocks from any available libraries (e.g., the Simulink library, DSP Builder, the Communications Blockset library, etc.).

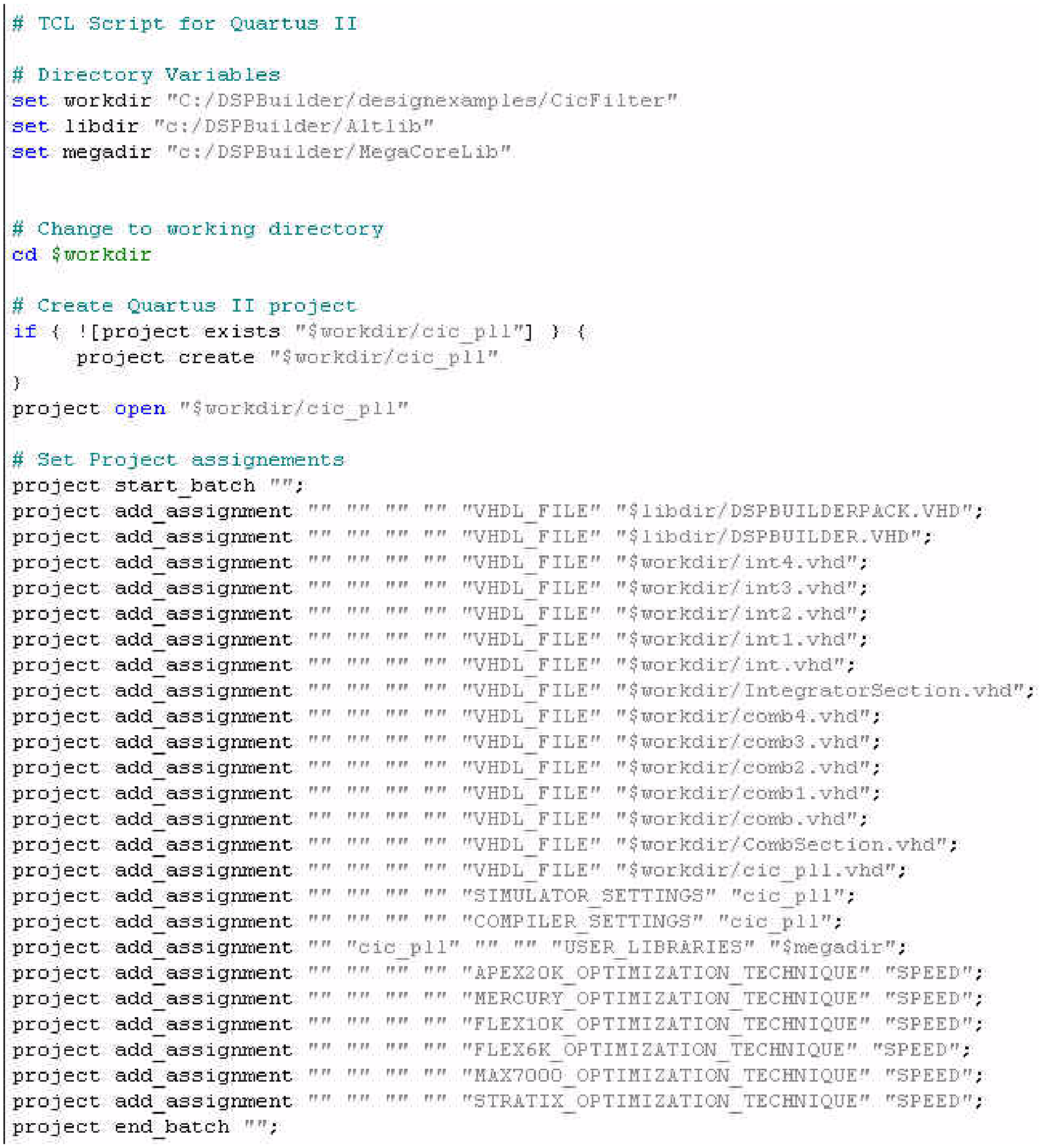
Using DSP Builder Modules in External RTL Designs

You can use DSP Builder designs as modules in RTL designs created in other applications, e.g., the Quartus II software. When incorporating your DSP Builder design, you must adjust your top-level design settings to support the DSP Builder design, including:

- Quartus II compilation settings
- Project settings (include all DSP Builder VHDL files)
- Library path settings

For a top-level design that has a DSP Builder submodule, the Quartus II project must include all of the Quartus II compilation settings of the standalone DSP Builder design. The Tcl script <*DSP Builder design*>**_quartus.tcl** contains the DSP Builder design's Quartus II compilation settings. Figure 50 shows the Tcl script for the CIC filter example (<*path*>**\designexamples\cicfilter\cic_pll_quartus.tcl**).

Figure 50. CIC Filter Example Quartus II Compilation Settings Tcl Script



```
# TCL Script for Quartus II

# Directory Variables
set workdir "C:/DSPBuilder/designexamples/CicFilter"
set libdir "c:/DSPBuilder/Altlib"
set megadir "c:/DSPBuilder/MegaCoreLib"


# Change to working directory
cd $workdir

# Create Quartus II project
if { ![project exists "$workdir/cic_pll"] } {
     project create "$workdir/cic_pll"
}
project open "$workdir/cic_pll"

# Set Project assignements
project start_batch "";
project add_assignment "" "" "" "" "VHDL_FILE" "$libdir/DSPBUILDERPACK.VHD";
project add_assignment "" "" "" "" "VHDL_FILE" "$libdir/DSPBUILDER.VHD";
project add_assignment "" "" "" "" "VHDL_FILE" "$workdir/int4.vhd";
project add_assignment "" "" "" "" "VHDL_FILE" "$workdir/int3.vhd";
project add_assignment "" "" "" "" "VHDL_FILE" "$workdir/int2.vhd";
project add_assignment "" "" "" "" "VHDL_FILE" "$workdir/int1.vhd";
project add_assignment "" "" "" "" "VHDL_FILE" "$workdir/int.vhd";
project add_assignment "" "" "" "" "VHDL_FILE" "$workdir/IntegratorSection.vhd";
project add_assignment "" "" "" "" "VHDL_FILE" "$workdir/comb4.vhd";
project add_assignment "" "" "" "" "VHDL_FILE" "$workdir/comb3.vhd";
project add_assignment "" "" "" "" "VHDL_FILE" "$workdir/comb2.vhd";
project add_assignment "" "" "" "" "VHDL_FILE" "$workdir/comb1.vhd";
project add_assignment "" "" "" "" "VHDL_FILE" "$workdir/comb.vhd";
project add_assignment "" "" "" "" "VHDL_FILE" "$workdir/CombSection.vhd";
project add_assignment "" "" "" "" "VHDL_FILE" "$workdir/cic_pll.vhd";
project add_assignment "" "" "" "" "SIMULATOR_SETTINGS" "cic_pll";
project add_assignment "" "" "" "" "COMPILER_SETTINGS" "cic_pll";
project add_assignment "" "cic_pll" "" "" "USER_LIBRARIES" "$megadir";
project add_assignment "" "" "" "" "APEX20K_OPTIMIZATION_TECHNIQUE" "SPEED";
project add_assignment "" "" "" "" "MERCURY_OPTIMIZATION_TECHNIQUE" "SPEED";
project add_assignment "" "" "" "" "FLEX10K_OPTIMIZATION_TECHNIQUE" "SPEED";
project add_assignment "" "" "" "" "FLEX6K_OPTIMIZATION_TECHNIQUE" "SPEED";
project add_assignment "" "" "" "" "MAX7000_OPTIMIZATION_TECHNIQUE" "SPEED";
project add_assignment "" "" "" "" "STRATIX_OPTIMIZATION_TECHNIQUE" "SPEED";
project end_batch "";
```

The Quartus II top-level design project settings must include all of the DSP Builder VHDL files needed to compile the DSP Builder module. For the CIC filter example, these files are:

- **DSPBUILDERPACK.VHD**
- **DSPBUILDER.VHD**
- **int4.vhd**
- **int3.vhd**
- **int2.vhd**
- **int1.vhd**
- **int.vhd**
- **IntegratorSection.vhd**
- **comb4.vhd**
- **comb3.vhd**
- **comb2.vhd**
- **comb1.vhd**
- **comb.vhd**
- **CombSection.vhd**
- **cic_pll.vhd**

The top-level design's Quartus II project settings must include all of the user library paths needed to compile the DSP Builder module. For the CIC filter example, this path is:

```
c:/DSPBuilder/MegaCoreLib
```

This path is shown in the Tcl script in the lines:

```
set megadir "c:/DSPBuilder/MegaCoreLib"
project add_assignment "" "cic_pll" "" "" "USER_LIBRARIES" "$megadir"
```

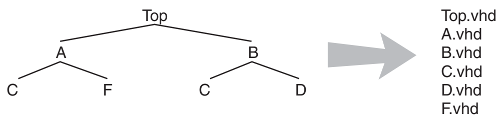
The blocks in the AltLab library are used to manage design hierarchy and generate RTL VHDL for synthesis and simulation.

HDL SubSystem Block

You can use the HDL SubSystem block to add a level of hierarchy to your design. SignalCompiler generates a separate VHDL file for each DSP Builder HDL SubSystem block, each containing a single entity/architecture pair. For example, if you have a single model file with two subsystems, SignalCompiler creates 3 output files, one for the model file and one for each of the subsystems.

DSP Builder creates VHDL files in which the entity name space is global; therefore, all subsystem names must be unique. If your model has the structure shown in Figure 51, both instances of subsystem C must be identical. If they are different, only one will be used.

Figure 51. Example Design Hierarchy



Refer to the DSP Builder Report File, which is output when SignalCompiler finishes generating HDL, for a listing of the HDL SubSystem blocks used in the design.

SignalCompiler Block

The SignalCompiler block is the heart of the DSP Builder, and performs the following functions:

- Converts your Simulink design into synthesizable RTL VHDL.
- Generates VHDL testbenches.
- Exports Simulink stimulus into a VHDL testbench and produces the expected response in an ASCII (**.txt**) file.
- Generates Tcl scripts for Quartus II compilation.
- Generates Tcl scripts for the LeonardoSpectrum, Synplify, and ModelSim software.
- Generates a vector file (**.vec**) for Quartus II simulation
- Generates a PTF configuration file, which you can use to import the design automatically into the SOPC Builder
- Enables generation of a SignalTap II **.stp** file

For more information on the SOPC Builder, refer to http://www.altera.com/products/software/system/products/sopc/sop-index.html

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

- Synthesis and compilation
  - *Automated Flow*—The automated flow allows you to control the entire design process from within the MATLAB/Simulink environment using the SignalCompiler block. With this flow, the SignalCompiler creates RTL HDL design files and Tcl scripts in your working directory. Then, SignalCompiler executes the Tcl scripts to synthesize the design in the Quartus II, LeonardoSpectrum, or Synplify software and compile it in the Quartus II software. The results of the synthesis and compilation are displayed in the SignalCompiler **Messages** box. You can also use the automated flow to download your design into the DSP development boards (starter or professional).
  - *Manual Flow*—With the manual flow, you use SignalCompiler to output VHDL files and Tcl scripts; however, you do not use the scripts to run synthesis or compilation tools. After you choose to generate VHDL, SignalCompiler generates the RTL HDL design files and Tcl scripts in your working directory. You can then use the VHDL files to synthesize the design in your tool of choice. You can compile the synthesized results in the Quartus II software. You should use the manual flow if you want to make your own synthesis or compilation settings.

- Simulation—The **Generate Stimuli for VHDL Testbench** option (**Testbench** tab) in the SignalCompiler block (**Testbench** tab), causes the SignalCompiler to generate a VHDL testbench and ModelSim Tcl script for your model as well as a Vector File for Quartus II simulation. You can use the testbench and Tcl script with the ModelSim software, you can use the testbench in another simulation tool, or you can use the Vector File with the Quartus II software.

Table 6 shows the parameters for the SignalCompiler analyzer.

Table 6. SignalCompiler Analyzer Parameters

Name	Value	Description
Update Simulation	On or Off	When this option is turned on, SignalCompiler performs a simulation update command on the design and then extracts DSP Builder block information.
Analyze	–	When you click this button, SignalCompiler reads the current MDL file and detects hierarchy level and sample period of all DSP Builder blocks. You must analyze the design every time you modify it.
Skip Analysis	–	Click this button to bypass the analysis, i.e., to change the design.

Table 7 shows the parameters for this block.

Table 7. SignalCompiler Parameters (Part 1 of 3)

Name	Value	Description
Device	APEX 20KE, APEX 20KC, APEX II, Stratix, Mercury, ACEX 1K, FLEX 10K, FLEX 6000, Cyclone, DSP Board	Indicate which Altera device family you want to target. You can also target the Stratix or APEX DSP development boards. If you are using the automated design flow, the Quartus II software chooses the smallest device in which your design fits.
Synthesis Tool	Quartus II, LeonardoSpectrum, Synplicity	Indicate which synthesis tool you want to use. If you choose Quartus II, LeonardoSpectrum, or Synplify, SignalCompiler generates Tcl scripts for use with the automated design flow.
Optimization	Area or Speed	Indicate whether you want to optimize the design for area or speed.
Select the Top-Level Model File	User Defined	Click the icon to browse to the location of your top-level Model File (**.mdl**). Specifying the file sets your working directory.

Table 7. SignalCompiler Parameters (Part 2 of 3)

Name	Value	Description
Generate Stimuli for VHDL Testbench	On or Off	To perform simulation in a third-party simulator, turn on this option and then run your simulation in Simulink. SignalCompiler outputs files for use with ModelSim or other simulators. This option is located on the **Testbench** tab. If this option is turned on, the Simulink simulation runs more slowly than if it is turned off. Therefore, you should only turn on this option when you wish to generate files for use with ModelSim or other simulators.
Period (ns)	User Defined	Specify the clock period. This box is located on the **Clock** tab.
Location	User Defined	This option is available the **Reset** tab. For the DSP development boards, you can assign the reset to switch 1, 2, or 3 on the board.
Connect to Ground	On or Off	Indicate whether you want to connect the reset signal to ground. This option is located on the **Reset** tab.
Insert SignalTap II Logic Analyzer	On or Off	Indicate whether you want to insert an instance of the SignalTap II logic analyzer into the design. You can use the logic analyzer to capture and probe signals on the DSP development boards. This option is located on the **SignalTap** tab and only applies if your are targeting the DSP development boards.
Generate PTF SOPC File	On or Off	When this option is turned on, SignalCompiler generates a **class.ptf** configuration file. With this file, the DSP Builder design is a plug-and-play peripheral for use with SOPC Builder.
1 - Convert MDL to VHDL	–	Click this button to generate VHDL design files and Tcl scripts for your design.
2 - Synthesize VHDL	–	Click this button to synthesize the generated VHDL in the synthesis tool that you selected. When the synthesis tool is Quartus II, this button is grayed out and the VHDL synthesis step is merged with Quartus II compilation into a single step, which is invoked by clicking the Quartus II Compilation button.

Table 7. SignalCompiler Parameters (Part 3 of 3)

Name	Value	Description
3 - Quartus II Compilation	–	Click this button to compile the design in the Quartus II software.
4 - Program DSP Board	–	If you selected one of the DSP development boards from the **Device** list, this button is enabled. Click this button to download your design into the targeted board. Refer to the *APEX DSP Development Kit Getting Started User Guide* or the *DSP Development Kit, Stratix Edition Getting Started User Guide* for information on setting up the board and connecting it to your PC.
System Information	–	Click this button to view the paths to the Synplify, LeonardoSpectrum, or Quartus II software, or your working directory.

Data Width Propagation

During the Simulink-to-VHDL conversion, SignalCompiler assigns a bit width to all of the Altera blocks in the design. You can specify the bit width of an Altera block in the Simulink design. However, you do not need to specify the bit width for all blocks. SignalCompiler propagates the bit width from the source of a data path to its destination.

The design **fir3tapsub.mdl**, which is provided in the <*path*>**\DSPBuilder\DesignExamples\GettingStarted\Fir3Tap** directory, illustrates bit width propagation. The **fir3tapsub.mdl** design is a 3-tap finite impulse response (FIR) filter. See Figure 52.

☞ Before simulating the model or generating HDL files, ensure that the path to your working directory is set up properly. The **Working Directory** box is case-sensitive. The default working directory used in the example is:

C:\DSPBuilder\DesignExamples\GettingStarted\Fir3Tap

The design 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 52. 3-Tap FIR Filter

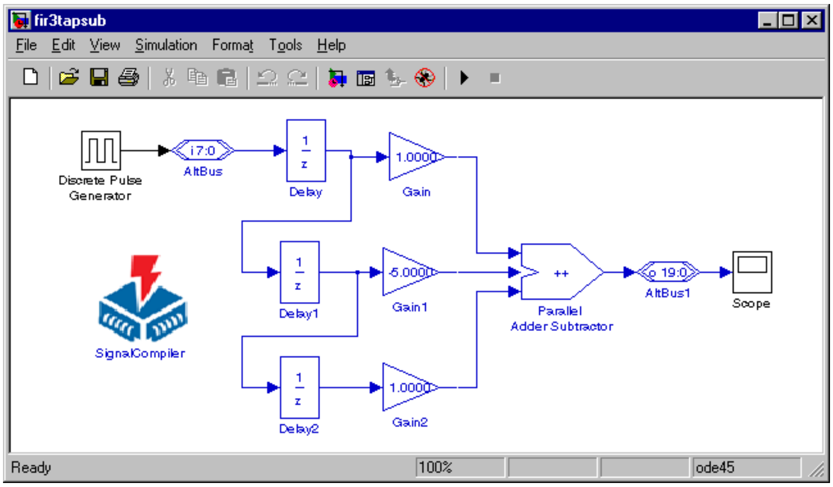
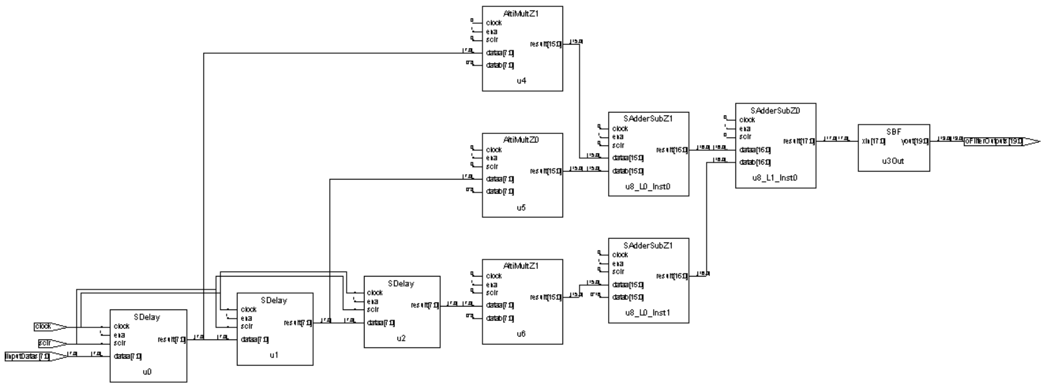


Figure 53 shows the RTL representation (in the Synplify software) of **fir3tapsub.mdl** created by SignalCompiler.

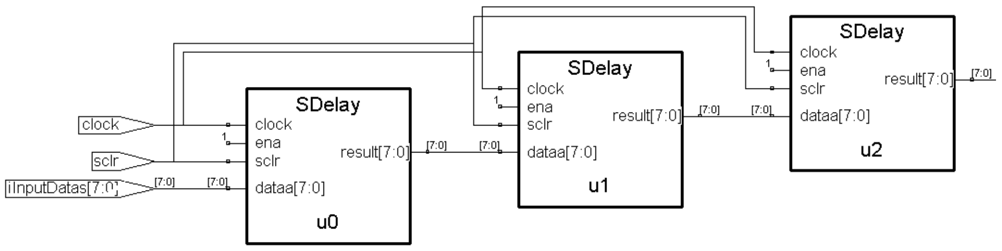
Figure 53. 3-Tap FIR Filter in RTL View (Synplify Software)



Tapped Delay Line

The bit width propagation mechanism starts at the source of the data path, in this case at the AltBus block iInputDatas, which is an 8-bit input bus. This bus feeds the register `U0`, which feeds `U1`, which feeds `U2`. SignalCompiler propagates the 8-bit bus in this register chain where each register is 8 bits wide. See Figure 54.

Figure 54. Tap Delay Line in RTL View (Synplify Software)



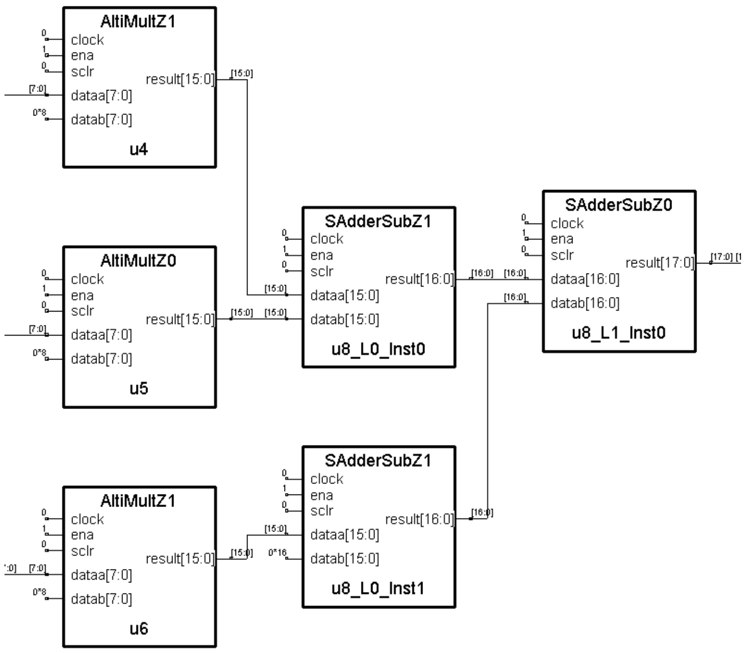
Arithmetic Operation

Figure 55 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 55. 3-Tap FIR Filter Arithmetic Operation in RTL View (Synplify Software)



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:

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

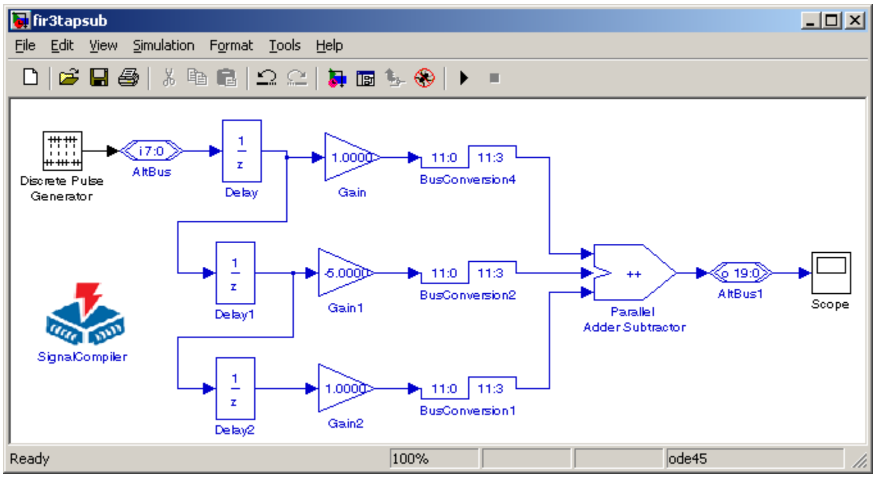
The parallel adder has three input buses of 14, 16, and 14 bits. To perform this addition in binary, SignalCompiler automatically sign extends the 14 bit buses to 16 bits. The output bit width of the parallel adder is 18 bits, which covers the full resolution.

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

- Change the bit width of the input data.
- Change the bit width of the output data. The VHDL synthesis tool will remove the unused logic.
- Insert BusConversion blocks to change the internal signal bit width.

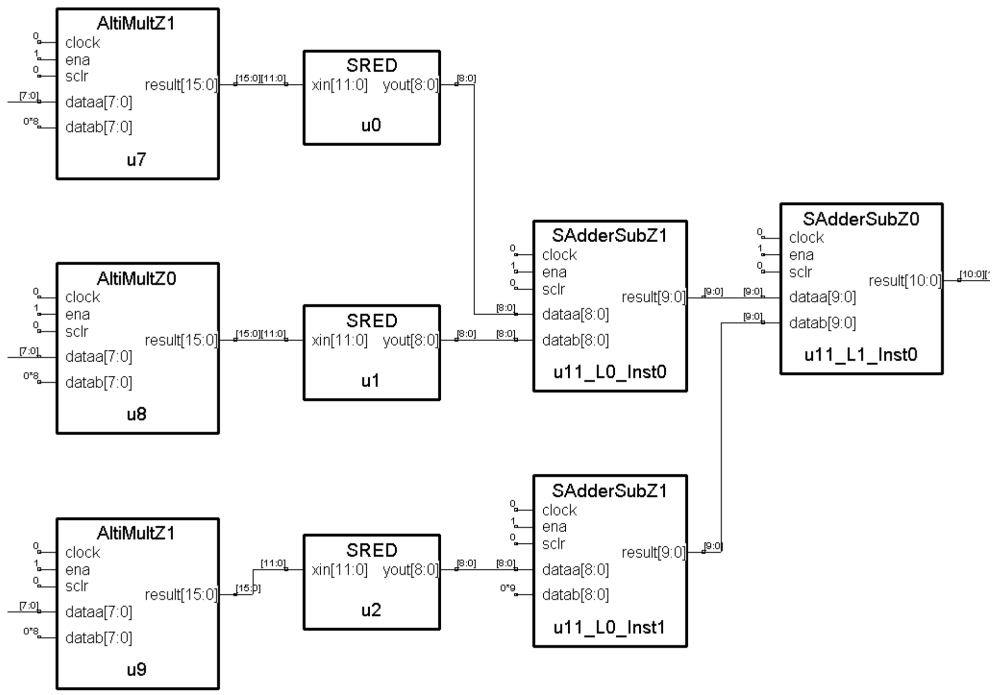
Figure 56 shows how AltBus blocks are used to control internal bit widths. In this example, the output of the Gain block has 4 bits removed (BusConversion format converts [11:0] to [11:3]). The scope displays the functional effect of this truncation on the coefficient values.

Figure 56. 3-Tap Filter with BusConversion to Control Bit Widths



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 57.

Figure 57. 3-Tap Filter with BusConversion to Control Bit Widths in RTL View (Synplify Software)



Clock Assignment

As described in "Frequency Design Rule" on page 64, SignalCompiler 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 the PLL block, the SignalCompiler implicitly connects all of the registered DSP Builder blocks' clock pins to a single clock domain (signal 'clock' in VHDL). When the design contains the PLL block and the detected Simulink sampling period of the registered DSP Builder blocks is equal to one of the output clock periods defined in the PLL block, the SignalCompiler implicitly connects all of the registered blocks' clock pins to one of the PLL's output clocks, otherwise the SignalCompiler returns an error.

From a clocking standpoint, DSP Builder blocks fall into two categories:

- *Combinatorial 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 58 illustrates DSP Builder block combinatorial behavior. The Magnitude block translates as a combinatorial signal in VHDL. SignalCompiler does not add clock pins to this function.

Figure 58. Magnitude Block

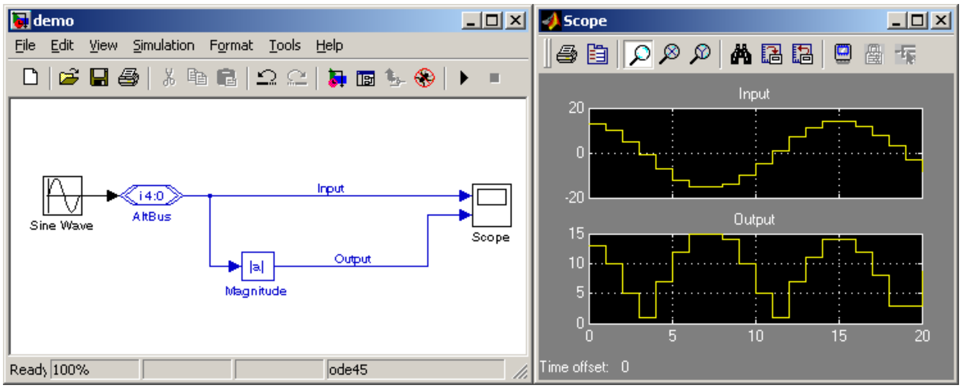
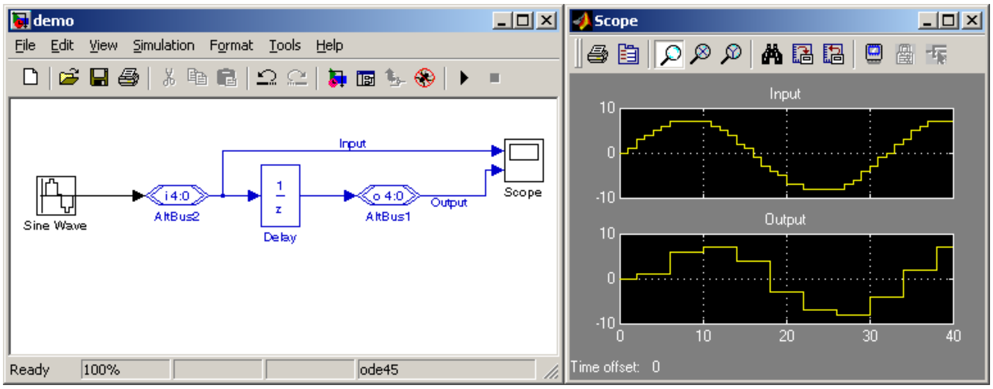


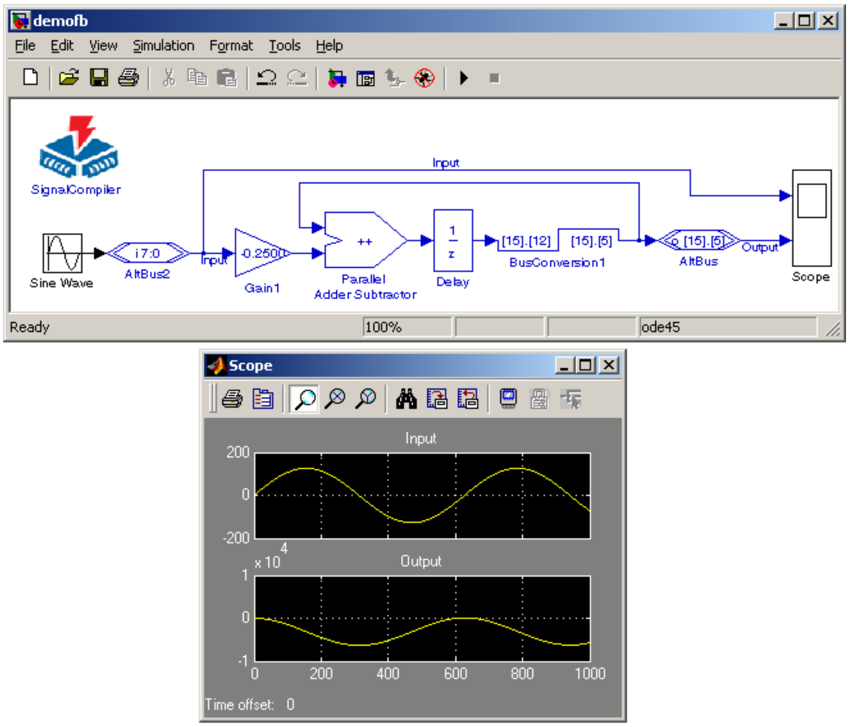
Figure 59 illustrates the behavior of a registered DSP block. In the VHDL netlist, SignalCompiler adds clock pin inputs to this function. The Delay block, with the `clock phase selection` parameter equal to 100, is converted into a VHDL shift register with a decimation of 3 and an initial value of zero.

Figure 59. Delay Block



For feedback circuitry (i.e., 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 combinatorial loop is created. See Figure 60.

Figure 60. 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 and the following design rules. For a multi-rate design using a single clock domain with clock enable:

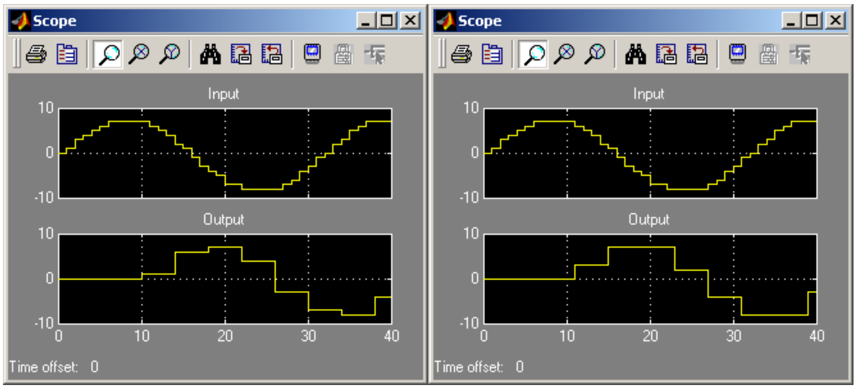
- The fastest sample rate is an integer multiple of the slower sample rates. The values are specified in the **Clock Phase Selection** box in the **Delay** dialog box.
- The **Clock Phase Selection** box accepts a binary pattern string to describe the clock phase selection. Each digit or bit of this 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 8.

Table 8. 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 second phase out of 4 and only the second 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 61 compares the scopes for the Delay block operating at a one quarter rate on the `1000` and `0100` phases, respectively.

Figure 61. 1000 vs. 0100 Phase Delay



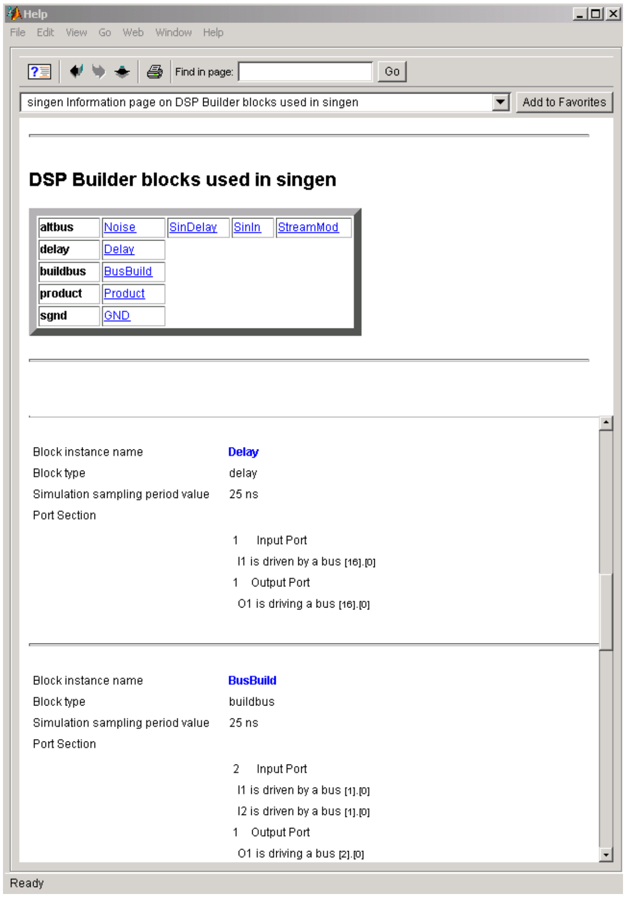
DSP Builder Report File

After generating VHDL, SignalCompiler outputs a report file that lists the SignalCompiler block parameters, the files generated by SignalCompiler, links to the synthesis log file and the Quartus II Report File, and the following information for each VHDL entity:

- SignalCompiler settings
- Signal width mismatches
- Out-of-range signals
- Detected Simulink sampling period
- Block port bit width information for the entity
- Number of HDL SubSystem instances used (heirarchy information)

The report file shows how SignalCompiler propagates the bit widths to all of the design blocks at each level of hierarchy. Figures 62 and 63 show an example DSP Builder report file.

Figure 62. DSP Builder Block Report File


Help
File Edit View Go Web Window Help
Find in page: Go
singen Information page on DSP Builder blocks used in singen
Add to Favorites
Ready

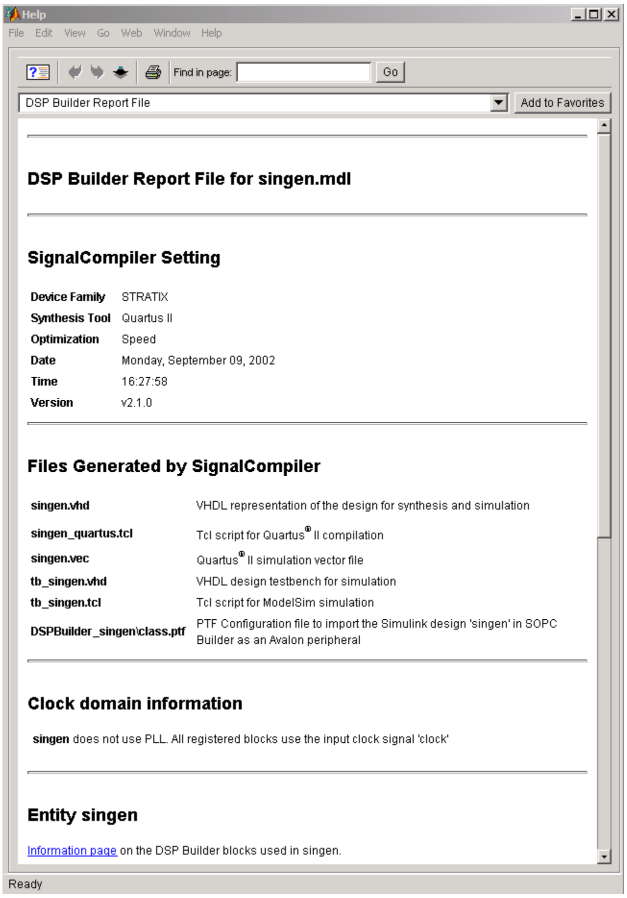
DSP Builder blocks used in singen

altbus	Noise	SinDelay	SinIn	StreamMod
delay	Delay			
buildbus	BusBuild			
product	Product			
sgnd	GND			

```
Block instance name                 Delay
Block type                          delay
Simulation sampling period value    25 ns
Port Section
                                    1   Input Port
                                     I1 is driven by a bus [16].[0]
                                    1   Output Port
                                     O1 is driving a bus [16].[0]
```

```
Block instance name                 BusBuild
Block type                          buildbus
Simulation sampling period value    25 ns
Port Section
                                    2   Input Port
                                     I1 is driven by a bus [1].[0]
                                     I2 is driven by a bus [1].[0]
                                    1   Output Port
                                     O1 is driving a bus [2].[0]
```

Figure 63. Example DSP Builder Report File



DSP Builder Report File for singen.mdl

SignalCompiler Setting

Device Family	STRATIX
Synthesis Tool	Quartus II
Optimization	Speed
Date	Monday, September 09, 2002
Time	16:27:58
Version	v2.1.0

Files Generated by SignalCompiler

singen.vhd	VHDL representation of the design for synthesis and simulation
singen_quartus.tcl	Tcl script for Quartus® II compilation
singen.vec	Quartus® II simulation vector file
tb_singen.vhd	VHDL design testbench for simulation
tb_singen.tcl	Tcl script for ModelSim simulation
DSPBuilder_singen\class.ptf	PTF Configuration file to import the Simulink design 'singen' in SOPC Builder as an Avalon peripheral

Clock domain information

singen does not use PLL. All registered blocks use the input clock signal 'clock'

Entity singen

Information page on the DSP Builder blocks used in singen.

SignalTap II Analysis Block

As programmable logic design complexity increases, system verification in software becomes time consuming and replicating real-world stimulus is increasingly difficult. To alleviate these problems, you can supplement traditional methods of system verification with efficient board-level verification. DSP Builder version 2.0 and higher supports the SignalTap® II embedded logic analyzer, which lets you capture signal activity from internal Altera device nodes while the system under test runs at speed.

The DSP Builder includes the SignalTap II Analysis block with which you can set up event triggers, configure memory, and display captured waveforms (you use the Node block to select signals to monitor). Samples are saved to internal embedded system blocks (ESBs) when the logic analyzer is triggered, and are subsequently streamed off chip via the JTAG port using the ByteBlasterMV™ download cable. The captured data is then stored in a text file, displayed as a waveform in a MATLAB plot, and transferred to the MATLAB work space as a global variable.

The DSP Builder and the SignalTap II Analysis block create a SignalTap II embedded logic analyzer that:

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

☞ You can also use the Quartus II software to insert an instance of the SignalTap II embedded logic analyzer in your design. The Quartus II software supports additional features such as analyzing nodes in all levels of the design hierarchy, using multiple clock domains, and adjusting what percentage of data is captured around the trigger point. Refer to Quartus II Help for more information on using the SignalTap II embedded logic analyzer with the Quartus II software.

Design Flow

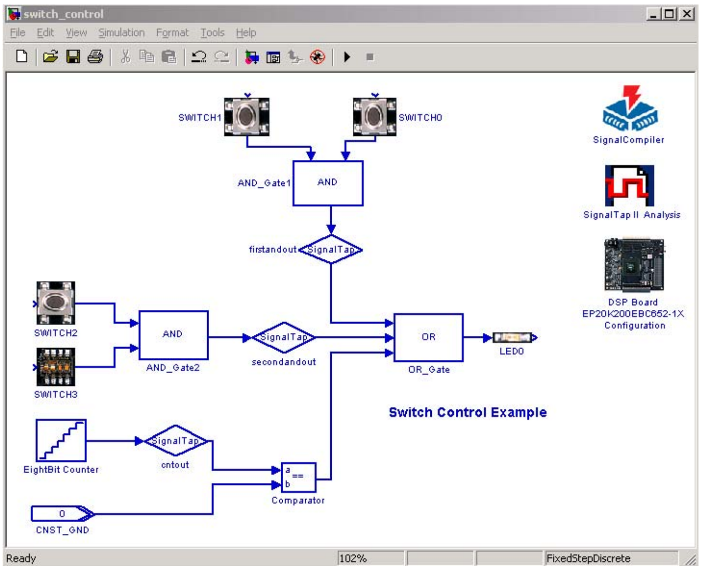
Working with the SignalTap II embedded logic analyzer and the DSP Builder involves the following flow.

1. Add a SignalTap II Analysis block to your Simulink design.

   ☞ You cannot open the SignalTap Analyzer block unless you have generated VHDL by clicking the button next to **1 - Convert MDL to VHDL** in the SignalCompiler.

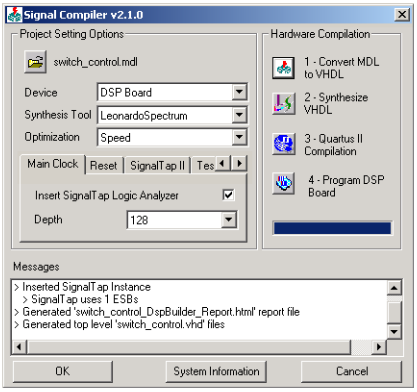
2. Specify the signals (i.e., nodes) that you want to analyze by inserting Node blocks. Figure 64 shows an example design.

Figure 64. Example SignalTap II Analysis Model



3. Turn on the **Insert SignalTap Logic Analyzer** option in the SignalTap tab of SignalCompiler. See Figure 65.

Figure 65. SignalTap II Tab in SignalCompiler



switch_control.mdl

Device: DSP Board
Synthesis Tool: LeonardoSpectrum
Optimization: Speed

Insert SignalTap Logic Analyzer ✓
Depth: 128

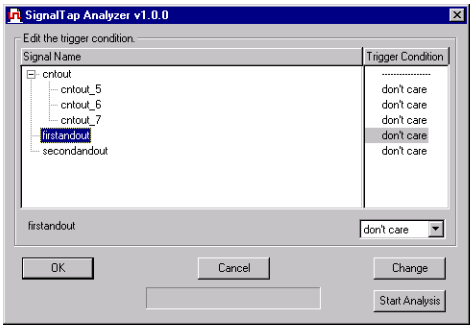
```
> Inserted SignalTap Instance
  > SignalTap uses 1 ESBs
> Generated 'switch_control_DspBuilder_Report.html' report file
> Generated top level 'switch_control.vhd' files
```

4. Target one of the DSP development board devices in the SignalCompiler.

5. Using SignalCompiler, generate VHDL, synthesize it, perform Quartus II compilation, and download the design into the DSP development board (starter or professional).

6. Specify trigger conditions in the SignalTap II Analysis block. See Figure 66.

Figure 66. Specifying Trigger Conditions



Signal Name	Trigger Condition
cntout	----------------
cntout_5	don't care
cntout_6	don't care
cntout_7	don't care
firstandout	don't care
secondandout	don't care

firstandout don't care

7. Specify the radix for the signal groups and run the analyzsis. You can view the captured data as a waveform in two MATLAB plots. The first plot displays the signals in binary format. The second plot displays the signals in the radix you specified. See Figures 67 and 68.

☞ DSP Builder only supports the SignalTap II embedded logic analyzer with the ByteBlasterMV download cable.

Figure 67. Example SignalTap II Analysis MATLAB Plot

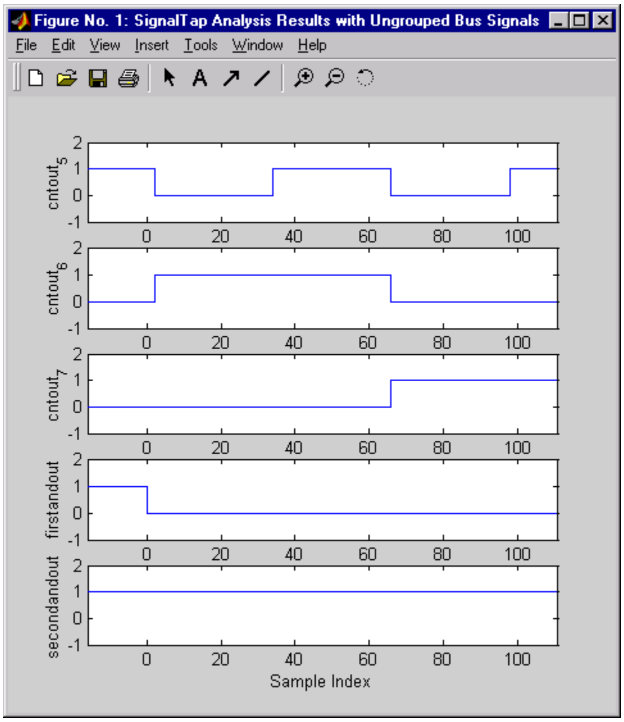
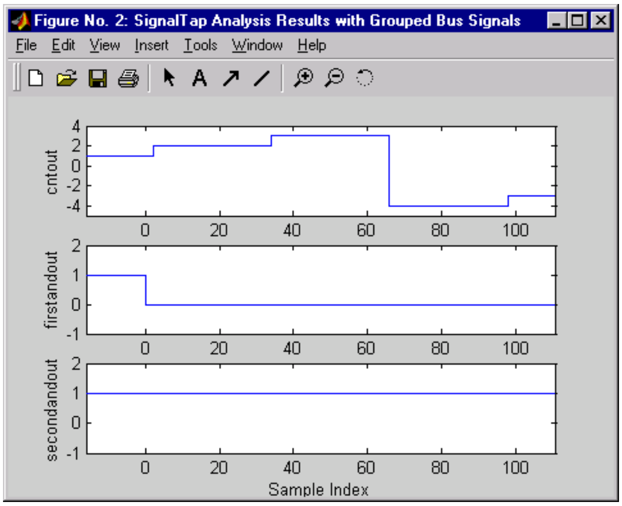


Figure No. 1: SignalTap Analysis Results with Ungrouped Bus Signals
File Edit View Insert Tools Window Help
cntout_5
cntout_6
cntout_7
firstandout
secondandout
2
1
0
-1
0
20
40
60
80
100
Sample Index

Figure 68. MATLAB Plot with User-Specified Radix



For detailed instructions on using the SignalTap II Analysis and SignalTap II blocks, refer to "Performing SignalTap II Logic Analysis" on page 53.

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 top-level design file, including I/O pins.

When you implement the SignalTap II embedded logic analyzer using DSP Builder, you can only analyze signals in a top-level design file. You cannot probe signals within a subsystem.

The SignalTap II embedded logic analyzer can analyze up to 128 internal nodes or I/O elements. As more signals are captured, more logic elements (LEs) and embedded system blocks (ESBs) are used. Table 9 lists the number of Altera device ESBs consumed for a given sample depth and number of signals.

Table 9. ESB Usage Note (1)

Signals (Width)	Samples (Depth)					
	0	128	256	512	1,024	2,048
1	0					1
2	0				1	2
4	0			1	2	4
8	0		1	2	4	8
16	0	1	2	4	8	16
32	0	2	4	8	16	32
64	0	4	8	16	32	64
128	0	8	16	32	64	128

Note:
(1) The signals and samples you can implement are dependant on the Altera device you use. Rows shaded in blue indicate signals and samples that you can implement in the APEX DSP development board (starter version). You can implement all signal widths and samples in the professional version.

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 Node block.

Trigger Conditions

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

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

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

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

Node Block

You use the Node block with the SignalTap II Analysis block to capture signal activity from internal Altera device nodes while the system under test runs at speed. The Node block indicates the signals (also called nodes) for which you want to capture activity. When you add a Node block, you specify a range of bits to analyze by choosing the bit position of the most significant bit (MSB) and least significant bit (LSB). For example, if you want to analyze the 3 most significant bits of an 8-bit bus, you would specify 7 for the **MSB** parameter and 5 for the **LSB** parameter.

Table 10 shows the block parameters.

Table 10. Node Parameters

Name	Value	Description
MSB	0 to 51	Indicate the bit position for the MSB of the range of bits you want to analyze.
LSB	0 to 51	Indicate the bit position for the LSB of the range of bits you want to analyze.

Figure 64 on page 108 shows an example using the Node block.

Refer to the following sources for more information:

- "Block I/O Formats" on page 78
- "SignalTap II Analysis Block" on page 107
- "Performing SignalTap II Logic Analysis" on page 53

SubSystem Builder Block

The SubSystemBuilder block makes it easy for you to import a VHDL design's input and output signals into a Simulink subsystem. You can then add the rest of the VHDL design functionality to the subsystem (using DSP Builder blocks or MATLAB functions) and simulate it in Simulink. You can optionally treat the VHDL design as a black box.

The SubSystemBuilder block automatically maps any input ports named `simulink_clock` in the VHDL entity section to the main Simulink system clock.

The VHDL entity should be formatted according to the following guidelines:

- The VHDL file should contain a single entity
- Port direction: `in` or `out`
- Port type: `STD_LOGIC` or `STD_LOGIC_VECTOR`
- Bus size:
  - `a(7 DOWNTO 0)` is supported
  - `a(8 DOWNTO 1)` is not supported
  - `a(0 TO 7)` is not supported
- Single port declaration per line:
  - `a :STD_LOGIC;` is supported
  - `a,b,c:STD_LOGIC` is not supported

To use the block, drag and drop it into your model, click **Select VHDL File**, specify the file to import, and click the **Add** <*entity name*> **in** <*model name*> button.

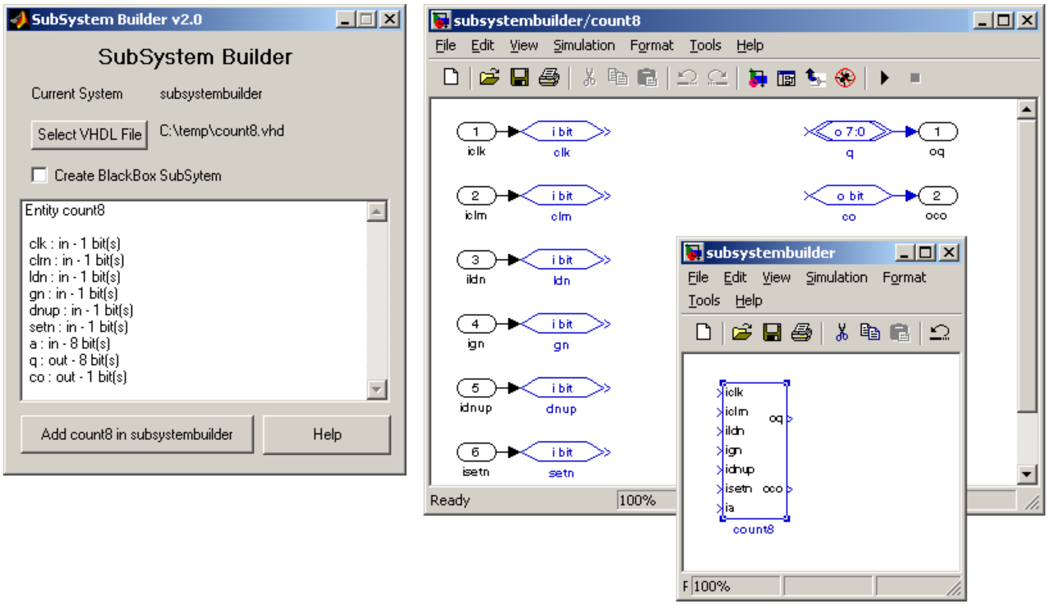
Table 11 shows the block parameters.

Table 11. SubSystem Builder Parameters

Name	Value	Description
Select VHDL File	User defined	Use this button to specify the VHDL file to import.
Create Black Box SubSystem	On or Off	Turn on this option if you want to treat the imported VHDL design as a black box.

Figure 69 shows an example using the SubSystemBuilder block.

Figure 69. SubSystemBuilder Block & Example



SubSystem Builder

Current System subsystembuilder

Select VHDL File C:\temp\count8.vhd

Create BlackBox SubSytem

```
Entity count8

clk : in - 1 bit(s)
clrn : in - 1 bit(s)
ldn : in - 1 bit(s)
gn : in - 1 bit(s)
dnup : in - 1 bit(s)
setn : in - 1 bit(s)
a : in - 8 bit(s)
q : out - 8 bit(s)
co : out - 1 bit(s)
```

Notes:

Arithmetic Library

This library contains two’s complement signed arithmetic blocks such as multipliers and adders. Some blocks have the **Use Dedicated Circuitry** option, which implements functionality into dedicated hardware in Altera Stratix (e.g., in the DSP block) or Mercury devices.

For more information on these device families, refer to:

- *Stratix Programmable Logic Device Family Data Sheet*
- *AN 214: Using the DSP Blocks in Stratix Devices*
- *Mercury Programmable Logic Device Family Data Sheet*

Comparator Block

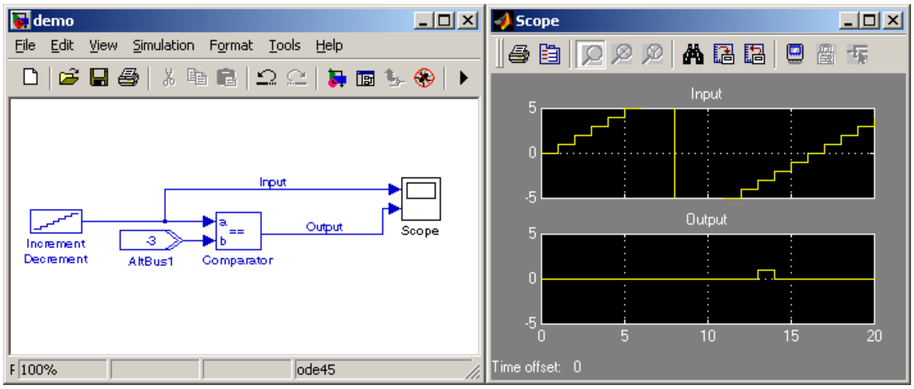
The Comparator block compares two Simulink signals and returns a single bit. The block implicitly understands the input data type (e.g., signed binary or unsigned integer) and produces a single-bit output. Table 12 shows the block parameters.

Table 12. Comparator Parameters

Name	Value	Description
Operator	a == b, a ~= b, a < b, a <= b, a >= b, a > b	Indicate which operation you wish to perform on the two buses.

Figure 70 shows an example using the Comparator block.

Figure 70. Comparator Example



See "Block I/O Formats" on page 78 for information on the block I/O formats.

Divider Block

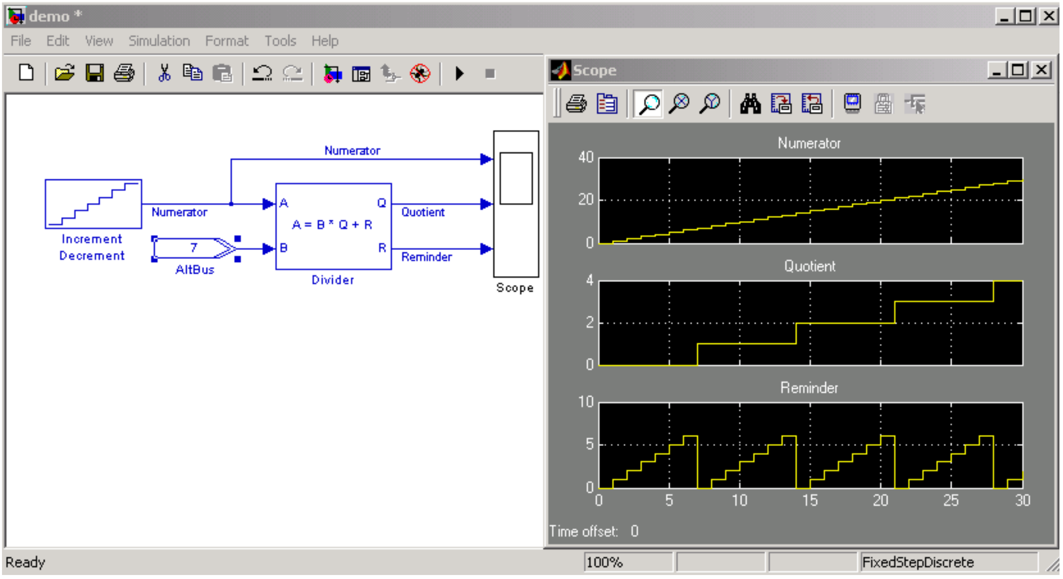
The Divider block takes a numerator and a denominator and computes a quotient and a remainder. The bit-width format of the numerator, denominator, quotient, and reminder are identical. Table 13 shows the block parameters.

Table 13. Divider Block Parameters

Name	Value	Description
Bus Type	Signed Integer, Signed Fractional, Unsigned Integer	Indicate the bus number format that you want to use for the divider.
[number of bits].[]	1 to 51	Select the number of bits to the left of the binary point.
[].[number of bits]	0 to 51	Select the number of bits to the right of the binary point for the gain. This option is only available when Signed Fractional is selected.
Pipeline	On or off	When on, adds one pipeline level to increase the data throughput.

Figure page page104_dividerexample.bmp shows an example using the Divider block.

Figure 71. Divider Example



Gain Block

The Gain block generates its output by multiplying the signal input by a specified gain factor. You must enter the gain as a numeric value in the Gain parameter field. The signal input and gain must be scalars.

☞ The Simulink software also provides a Gain block. If you use the Simulink Gain block in your model, you can only use it for simulation; SignalCompiler cannot convert it to HDL.

Table 14 shows the block parameters.

Table 14. Gain Parameters (Part 1 of 2)

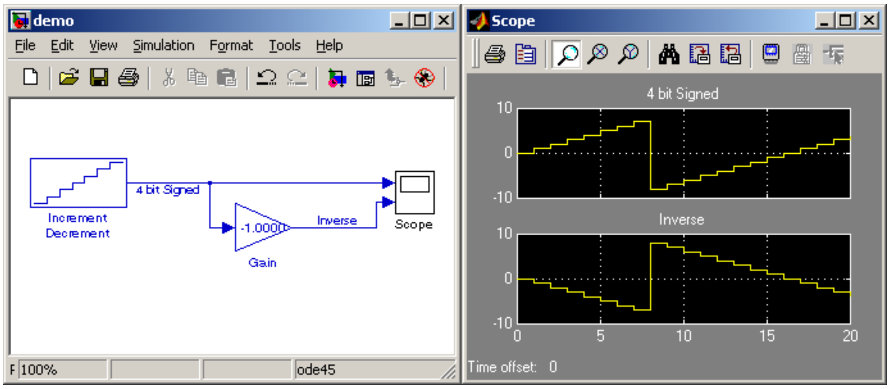
Name	Value	Description
Gain Value	User Defined	Indicate the gain value you want to use as a decimal number. The gain is masked to the number format (bus type) you select.
Map Gain Value to Bus Type	Signed Integer, Signed Fractional, Unsigned Integer	Indicate the bus number format you want to use for the gain value.
[Gain value number of bits].[]	1 - 51	Select the number of bits to the left of the binary point, including the sign bit for the gain.
[].[Gain value number of bits]	0 - 51	Select the number of bits to the right of the binary point for the gain. This option is only available when **Signed Fractional** is selected.
Number of Pipeline Levels	0 - 4	Specify the pipeline delay.
Use LPM	On or Off	This parameter is used for synthesis. When the **Use LPM** option is turned on, the Gain block is mapped to the `LPM_MULT` library of parameterized modules (LPM) function and the VHDL synthesis tool uses the Altera `LPM_MULT` implementation.
Use Control Inputs	On or Off	Indicate whether you would like to use additional control inputs (clock enable and reset). This option is only available if the **Number of Pipeline Levels** setting is greater than 1.

Table 14. Gain Parameters (Part 2 of 2)

Name	Value	Description
Clock Phase Selection	User Defined	Phase selection. This option is only available if the **Number of Pipeline Levels** setting is greater than 1. Indicate the phase selection with a binary string, where a 1 indicates the phase in which the block is enabled. For example: `1`—The block is always enabled and captures all data passing through the block (sampled at the rate 1). `10`—The block is enabled every other phase and every other data (sampled at the rate 1) passes through. `0100`—The block is enabled on the second phase out of 4 and only the second 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 72 shows an example using the Gain block.

Figure 72. Gain Example



See "Block I/O Formats" on page 78 for information on the block I/O formats.

Increment Decrement Block

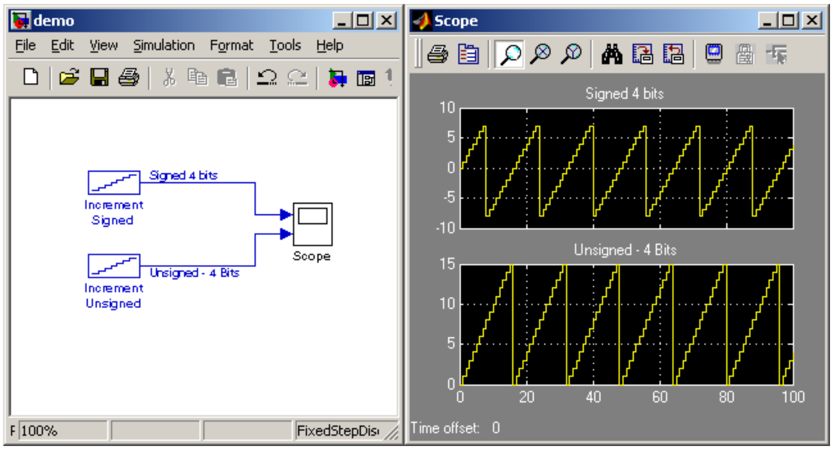
The Increment Decrement block generates a counting sequence in time. The output can be a signed integer, unsigned integer, or signed binary fractional number. For all number formats, the counting sequence increments/decrements the LSB bit by one. Table 15 shows the block parameters. The block has a clock phase selection control that operates as described in Table 15.

Table 15. Increment Decrement Parameters

Name	Value	Description
Bus Type	Signed Integer, Signed Fractional, Unsigned Integer	Choose the number format you wish to use for the bus.
<number of bits>.[]	1 - 51	Select the number of bits to the left of the binary point, including the sign bit.
[].<number of bits>	0 - 51	Select the number of bits to the right of the binary point. This option is only available when **Signed Fractional** is selected.
Direction	Increment or Decrement	Indicate whether you wish to count up or down.
Starting Value	User Defined	Enter the value with which to begin counting.
Use Control Inputs	On or Off	Indicate whether you would like to use additional control inputs (clock enable and reset).
Clock Phase Selection	User Defined	Phase selection. Indicate the phase selection with a binary string, where a 1 indicates the phase in which the block is enabled. For example: `1`—The block is always enabled and captures all data passing through the block (sampled at the rate 1). `10`—The block is enabled every other phase and every other data (sampled at the rate 1) passes through. `0100`—The block is enabled on the second phase out of 4 and only the second 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 73 shows an example using the Increment Decrement block.

Figure 73. Increment Decrement Example



See "Block I/O Formats" on page 78 for information on the block I/O formats.

Multiply Accumulate Block

The Multiply Accumulate block consists of a single multiplier feeding an accumulator. The input can be in signed integer, unsigned integer, or signed binary fractional formats.

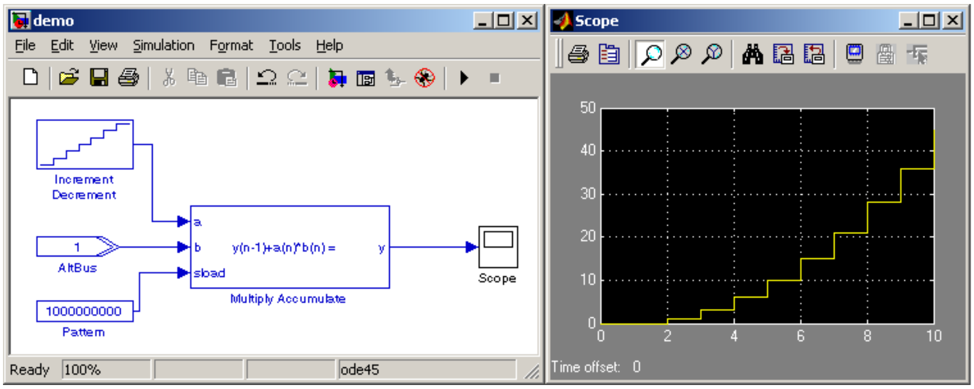
Table 16 shows the block parameters.

Table 16. Multiply Accumulate Parameters

Name	Value	Description
Bus Type	Signed Integer, Signed Fractional, Unsigned Integer	Choose the number format you wish to use for the bus.
Input A [number of bits].[]	1 - 51	Select the number of data input bits to the left of the binary point, including the sign bit.
Input A [].[number of bits]	0 - 51	Select the number of data input bits to the right of the binary point. This option is only available when **Signed Fractional** is selected.
Input B [number of bits].[]	1 - 51	Select the number of data input bits to the left of the binary point, including the sign bit.
Input B [].[number of bits]	0 - 51	Select the number of data input bits to the right of the binary point. This option is only available when **Signed Fractional** is selected.
Output Result Bits	1 - 51	Indicate the number of output bits.
Pipeline Register	None, Data Inputs, Multiplier Output, Data Inputs and Multiplier	Indicate whether you want to add pipelining to the data inputs, multiplier output, both, or neither.
Accumulator Direction	Add, Subtract	Indicate whether to add or subtract the result of the multiplier.
Use Control Inputs	On or Off	Indicate whether you would like to use additional control inputs (clock enable and reset).
Create Overflow Output Port	On or Off	Indicate whether you want to use an overflow port for the accumulator adder.
Use Dedicated Circuitry	On or Off	If you are targeting Stratix devices, turn on this option to implement the functionality in Stratix DSP blocks. If you are not targeting Stratix devices, the functionality is implemented in logic elements.

Figure 74 shows an example using the Multiply Accumulate block.

Figure 74. Multiply Accumulate Example



See "Block I/O Formats" on page 78 for information on the block I/O formats.

Multiply Add Block

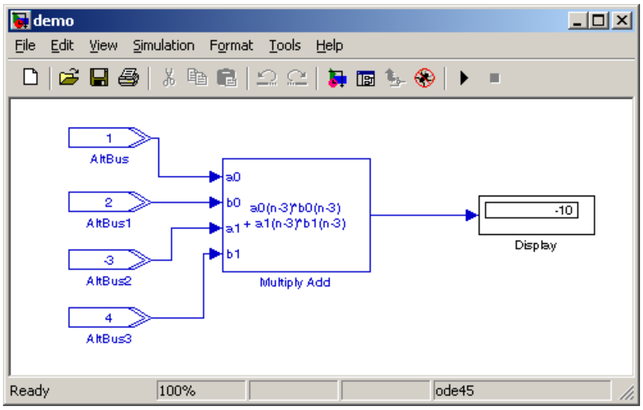
The Multiply Add block consists of one or more multipliers feeding a parallel adder. The input can be in signed integer, unsigned integer, or signed binary fractional formats. Table 17 shows the block parameters.

Table 17. Multiply Add Parameters

Name	Value	Description
Number of Multipliers	2, 3, 4	Choose how many multipliers you want to feed the adder.
Bus Type	Signed Integer, Signed Fractional, Unsigned Integer	Choose the number format you wish to use for the bus.
Inputs [number of bits].[]	1 - 51	Select the number of data input bits to the left of the binary point, including the sign bit.
Inputs [].[number of bits]	0 - 51	Select the number of data input bits to the right of the binary point. This option is only available when **Signed Fractional** is selected.
Adder Mode	Add Add, Add Sub, Sub Add, Sub Sub	Choose the operation of the adder.
Pipeline Register	No Register, Inputs Only, Multiplier Only, Adder Only, Inputs and Multiplier, Inputs and Adder, Multiplier and Adder, Inputs Multiplier and Adder	Choose the elements to which you want to add pipelining.
Use Clock Enable	On or Off	Indicate whether you would like to use an additional control input (clock enable).
Use Dedicated Circuitry	On or Off	If you are targeting Stratix devices, turn on this option to implement the functionality in Stratix DSP blocks. If you are not targeting Stratix devices, the functionality is implemented in logic elements.
One Input is Constant	On or Off	Turn on this option if one of the inputs is a constant. This option is used together with the **Constant Values** parameter.
Constant Values	User Defined	Type the constant value in this box as a MATLAB array. This option is only available if you have turned on the **One Input is Constant** option.

Figure 75 shows an example using the Multiply Add block.

Figure 75. Multiply Add Example

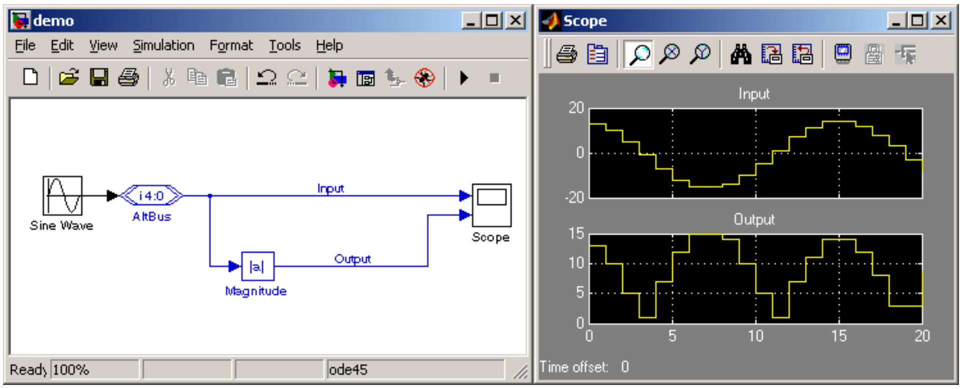


See “Block I/O Formats” on page 78 for information on the block I/O formats.

Magnitude Block

The scalar Magnitude block generates the output as the absolute value of the input. Figure 76 shows an example using the Magnitude block.

Figure 76. Magnitude Example



See "Block I/O Formats" on page 78 for information on the block I/O formats.

Parallel Adder Subtractor Block

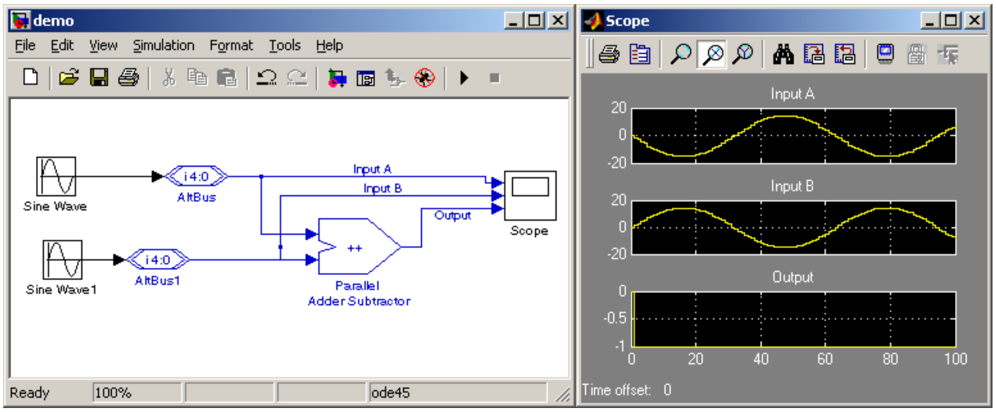
The Parallel Adder Subtractor block takes any input data type. If the input widths are not the same, the SignalCompiler sign extends the buses so that they match the largest input width. The VHDL generated has an optimized, balanced adder tree. Table 18 shows the block parameters.

Table 18. Parallel Adder Subtractor Parameters

Name	Value	Description
Number of Inputs	2 - 16	Indicate the number of inputs you wish to use.
Add (+) Sub (-)	User Defined	Specify addition or subtraction operation of each port with the characters (+) / (-). i.e., for 3 ports +-+ yields a - b + c. SignalCompiler will not accept two consecutive subtractions (i.e., --); however, -+- is acceptable.
Pipeline	On or Off	When this option is turned on, the pipeline delay is equal to ceil(log$_2$(number of inputs)).
Use Control Inputs	On or Off	Indicate whether you would like to use additional control inputs (clock enable and reset).
Clock Phase Selection	User Defined	Phase selection. Indicate the phase selection with a binary string, where a 1 indicates the phase in which the block is enabled. For example: `1`—The block is always enabled and captures all data passing through the block (sampled at the rate 1). `10`—The block is enabled every other phase and every other data (sampled at the rate 1) passes through. `0100`—The block is enabled on the second phase out of 4 and only the second 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 77 shows an example using the Parallel Adder Subtractor block.

Figure 77. Parallel Adder Subtractor Example



See “Block I/O Formats” on page 78 for information on the block I/O formats.

Product Block

The Product block supports two scalar inputs (no multi-dimensional Simulink signals).

The Simulink software also provides a Product block. If you use the Simulink Product block in your model, you can only use it for simulation; SignalCompiler cannot convert it to HDL. If you try to use the Simulink Product block with SignalCompiler, SignalCompiler will either treat it as a block box or will generate an error.

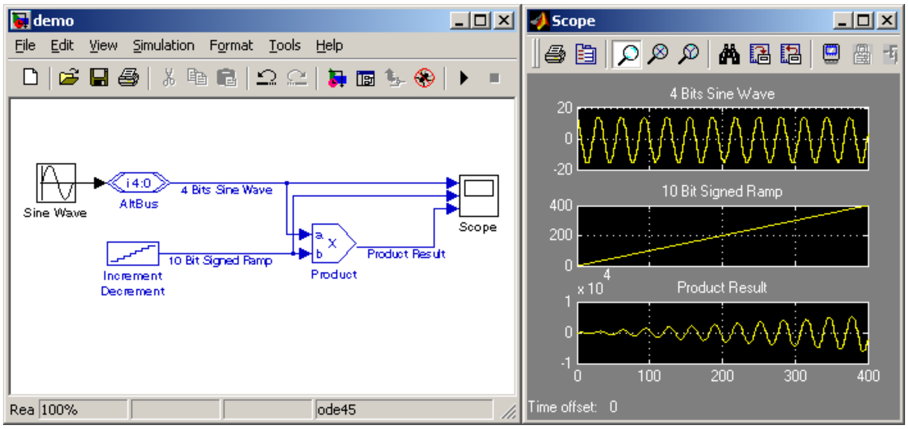
Table 19 shows the block parameters.

Table 19. Product Parameters

Name	Value	Description
Pipeline	0 - 4	The Pipeline value represents the delay.
Use LPM	On or Off	This parameter is used for synthesis. When the **Use LPM** option is turned on, the Product block is mapped to the `LPM_MULT` library of parameterized modules (LPM) function and the VHDL synthesis tool uses the Altera `LPM_MULT` implementation. If the option is turned off, the VHDL synthesis tool uses the `*` native operator to synthesize the product. If your design does not need arithmetic boundary optimization—such as connecting a multiplier to constant combinatorial logic or register balancing optimization—the `LPM_MULT` implementation generally yields a better result for both speed and area.
Use Dedicated Multiplier Circuitry	On or Off	Turn on this option if you want to use the dedicated multiplier circuitry in Mercury or Stratix devices. This option is ignored if you do not target one of these devices.
Use Control Inputs	On or Off	Indicate whether you would like to use additional control inputs (clock enable and reset).
Clock Phase Selection	User Defined	Phase selection. Indicate the phase selection with a binary string, where a 1 indicates the phase in which the block is enabled. For example: `1`—The block is always enabled and captures all data passing through the block (sampled at the rate 1). `10`—The block is enabled every other phase and every other data (sampled at the rate 1) passes through. `0100`—The block is enabled on the second phase out of 4 and only the second 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 78 shows an example using the Product block.

Figure 78. Product Example



See “Block I/O Formats” on page 78 for information on the block I/O formats.

Bus Manipulation Library

The blocks in the Bus Manipulation library are used to manipulate signals and buses to perform operations such as truncation, saturation, bit extraction, or bus format conversion.

AltBus Block

The AltBus block casts a floating-point Simulink bus to a fixed-point bus. You can insert AltBus into a data or I/O primary path for inputs and outputs.

When casting a signal to fixed point, you must specify the bit width. You have the option of truncating, saturating, or rounding the result. If you choose rounding or saturation, the appropriate logic is inserted.

Table 20 shows the AltBus parameters.

Table 20. AltBus Parameters (Part 1 of 2)

Name	Value	Description
Node Type	Internal Node, Input Port, Output Port, Constant, Black Box Input, Black Box Output	Indicate the type of node you wish to create.
Bus Type	Signed Integer, Signed Fractional, Unsigned Integer, or Single Bit	Choose the number format of the bus.
[number of bits].[]	1 - 51	Indicate the number of bits to the left of the binary point, including the sign bit. This parameter does not apply to single-bit buses.
[].[number of bits]	0 - 51	Indicate the number of bits to the right of the binary point. This parameter only applies to signed fractional buses.
Saturate	On or Off	When this option is turned on, if the output is greater than the maximum positive or negative value to be represented, the output is forced (or saturated) to the maximum positive or negative value, respectively. When this option is turned off, the MSB is truncated. This option is not valid for the input port or constant node types.

Table 20. AltBus Parameters (Part 2 of 2)

Name	Value	Description
Round	On or Off	When this option is turned on, the output is rounded away from zero. When this option is turned off, the LSB is truncated: <int>(input +0.5). This option is not valid for the input port or constant node types.
Bypass Bus Format	On or Off	Turn on this option if you wish to perform simulation in Simulink using floating-point numbers.
Constant Value	Double	Indicate the constant value that will be formatted with the bus parameter specified.

See "Block I/O Formats" on page 78 for information on the block I/O formats.

You can use the AltBus block in a Simulink design in any of the following modes:

- AltBus Input Port & Output Port Modes
- AltBus Internal Node Mode
- AltBus Black Box Input Output Mode
- AltBus Constant Mode

AltBus Input Port & Output Port Modes

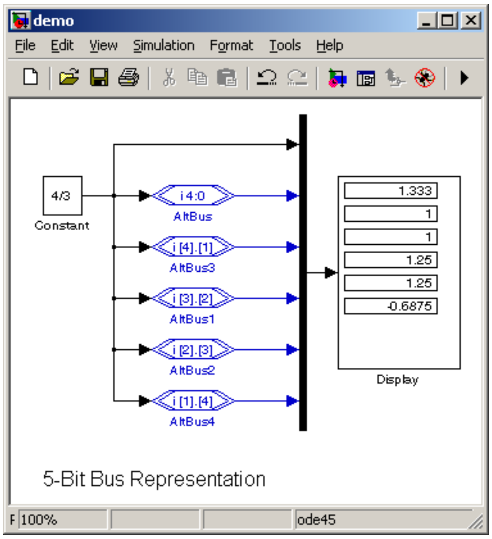
The Input Port and Output Port modes are used to define the boundaries of the hardware implementation as well as to cast floating-point Simulink signals (coming from generic Simulink blocks) to signed binary fractional format (feeding DSP Builder blocks). Table 21 and Figure 79 illustrate how a floating-point number (4/3 = 1.3333) is cast into SBF format with 3 different binary point locations.

Table 21. Floating-Point Numbers Cast to SBF

Bus Notation	Input	Simulink	VHDL
[4].[1]	4/3	1.00	2.00
[2].[3]	4/3	1.25	10.00
[1].[4]	4/3	-0.6875 *(1)*	-11.00

Note:
(1) In this case, more bits are needed to represent the integer part of the number.

Figure 79. Floating-Point Conversion

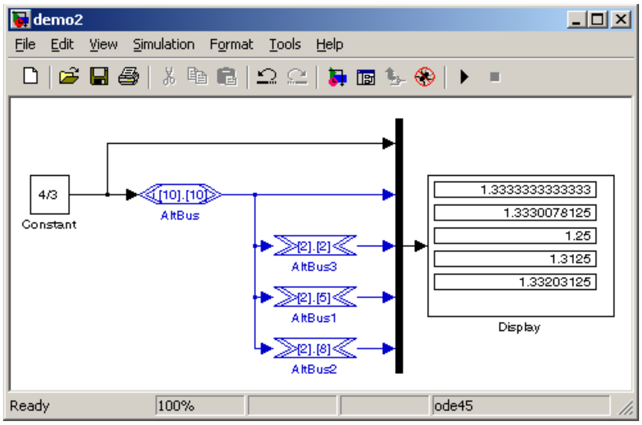


```
1.333
1
1
1.25
1.25
-0.6875
```

AltBus Internal Node Mode

This mode is used to convert a Simulink signal from one SBF format to another. This mode is used to assign the bus width of an internal node that will be implemented in hardware. Figure 80 illustrates the usage of AltBus in Internal Node mode (blocks Altbus1, AltBus2, and AltBus3) and Input Port mode (block AltBus). In this example a 20-bit bus with a ([10].[10]) SBF format is converted to a 4-bit bus with a [2].[2] SBF format.

Figure 80. Internal Node



```
1.3333333333333
1.3330078125
1.25
1.3125
1.33203125
```

In VHDL, this operation results in extracting a 4-bit bus (AltBus(3 DOWNTO 0)) from a 20-bit bus (AltBus(19 DOWNTO 0)) with the assignment:

```
AltBus3(3 DOWNTO 0)) <= Altbus(11 DOWNTO 8))
```

You can also perform additional internal bus manipulation with the Altera BusConversion, ExtractBit, or BuildBus blocks.

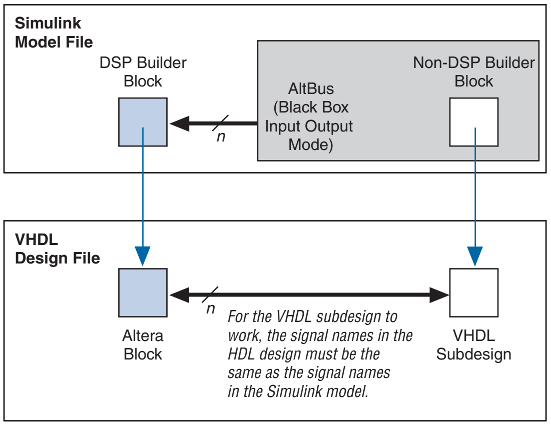
AltBus Black Box Input Output Mode

This AltBus mode is used for hierarchical designs. You should use this node type if you do not want SignalCompiler to translate the sub-level design to HDL (i.e., only the top-level symbol appears in the HDL). This mode is useful when your model has a Simulink block that is associated with a separate HDL block.

☞ The pin names of the HDL block must match the pin names of the Simulink block.

Figure 81 illustrates the Black Box Input Output mode. Refer to "Black Boxing" on page 86"Black Boxing" on page 89 for an example.

Figure 81. Black Box Input Output Mode


Simulink Model File
DSP Builder Block
Non-DSP Builder Block
AltBus (Black Box Input Output Mode)
n
VHDL Design File
n
Altera Block
For the VHDL subdesign to work, the signal names in the HDL design must be the same as the signal names in the Simulink model.
VHDL Subdesign

AltBus Constant Mode

Use this mode when a bus or bit must be set to a static value. SignalCompiler translates the static value to a constant STD_LOGIC or STD_LOGIC_VECTOR in VHDL. During synthesis, the synthesis tool typically reduces the gate count of any logic fed by this constant signal.

☞ If you use the Simulink Constant block in your Altera design, you can only use it for simulation. If you try to use the Simulink Constant block with SignalCompiler, SignalCompiler will either treat it as a block box or will generate an error.

Input, Output & Constant Blocks

The Input, Output, and Constant blocks are derived from the AtlBus block. These blocks perform a subset of the functionality of the AltBus block, i.e., input, output, or constant.

Table 22 shows the Input parameters.

Table 22. Input Parameters

Name	Value	Description
Bus Type	Signed Integer, Signed Fractional, Unsigned Integer, or Single Bit	Choose the number format of the bus.
[number of bits].[]	1 - 51	Indicate the number of bits to the left of the binary point, including the sign bit. This parameter does not apply to single-bit buses.
[].[number of bits]	0 - 51	Indicate the number of bits to the right of the binary point. This parameter only applies to signed fractional buses.

Table 23 shows the Output parameters.

Table 23. Output Parameters

Name	Value	Description
Bus Type	Signed Integer, Signed Fractional, Unsigned Integer, or Single Bit	Choose the number format of the bus.
[number of bits].[]	1 - 51	Indicate the number of bits to the left of the binary point, including the sign bit. This parameter does not apply to single-bit buses.
[].[number of bits]	0 - 51	Indicate the number of bits to the right of the binary point. This parameter only applies to signed fractional buses.
Saturate	On or Off	When this option is turned on, if the output is greater than the maximum positive or negative value to be represented, the output is forced (or saturated) to the maximum positive or negative value, respectively. When this option is turned off, the MSB is truncated. This option is not valid for the input port or constant node types.
Round	On or Off	When this option is turned on, the output is rounded away from zero. When this option is turned off, the LSB is truncated: <int>(input +0.5). This option is not valid for the input port or constant node types.

Table 24 shows the Constant parameters.

Table 24. Constant Parameters

Name	Value	Description
Bus Type	Signed Integer, Signed Fractional, Unsigned Integer, or Single Bit	Choose the number format of the bus.
[number of bits].[]	1 - 51	Indicate the number of bits to the left of the binary point, including the sign bit. This parameter does not apply to single-bit buses.
[].[number of bits]	0 - 51	Indicate the number of bits to the right of the binary point. This parameter only applies to signed fractional buses.
Constant Value	Double	Indicate the constant value that will be formatted with the bus parameter specified.

Binary Point Casting Block

The Binary Point Casting block moves the input bus binary point position. The output bit width remains equal to the input bit width.

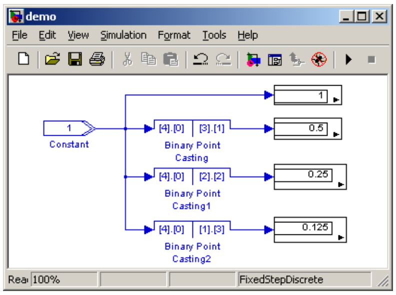
Table 25 shows the Binary Point Casting parameters.

Table 25. Binary Point Casting Parameters

Name	Value	Description
Bus Type	Signed Integer, Signed Fractional, Unsigned Integer, or Single Bit	Choose the number format of the bus.
[number of bits].[]	1 - 51	Indicate the number of bits to the left of the binary point, including the sign bit. This parameter does not apply to single-bit buses.
[].[number of bits]	0 - 51	Indicate the number of bits to the right of the binary point. This parameter only applies to signed fractional buses.
Output Binary Point Position	0 - 51	Specify the binary point location of the output.

Figure 82 shows a design example using the Binary Point Casting block.

Figure 82. Binary Point Casting Example



See "Block I/O Formats" on page 78 for information on the block I/O formats.

BusBuild Block

The BusBuild block is used to construct buses from single-bit inputs. The output bus is defined in signed binary fractional representation. You can choose the bus type that you wish to use, and specify the number of bits on either side of the binary point. The BusBuild and ExtractBit blocks are typically used when mixing bit-level Boolean operation with arithmetic operations. The HDL mapping of BusBuild is a simple wire.

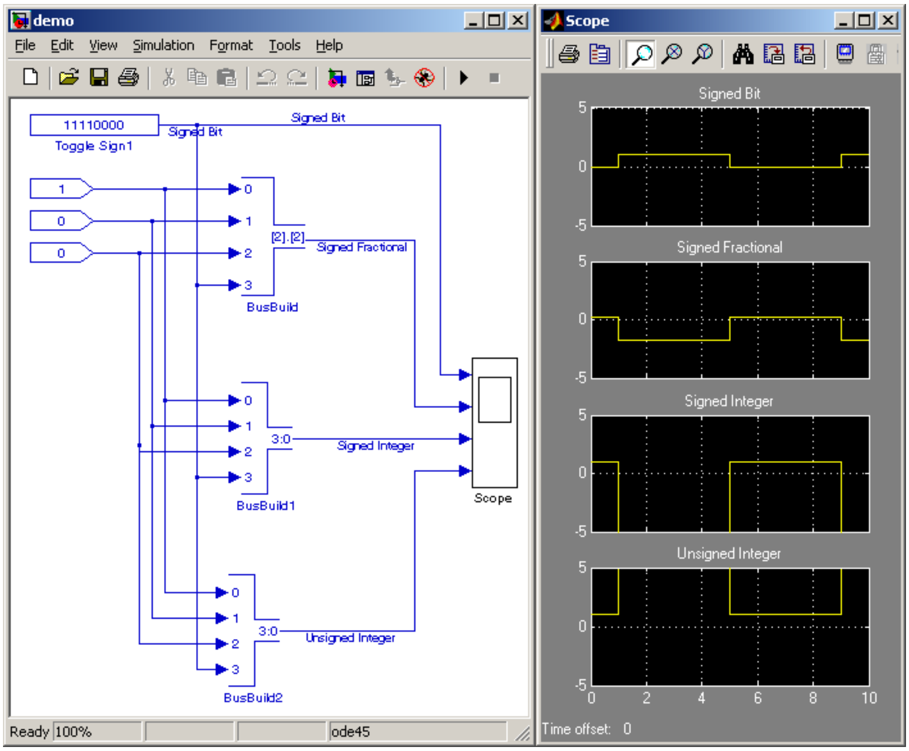
The input MSB (which is the sign bit of the signed binary fractional bus) is shown at the bottom left of the symbol and the input LSB is displayed at the top left of the symbol. Table 26 shows the block parameters.

Table 26. BusBuild Parameters

Name	Value	Description
Bus Type	Signed Integer, Signed Fractional, or Unsigned Integer	Choose the number format of the bus.
Output [number of bits].[]	1 - 51	Indicate the number of bits to the left of the binary point, including the sign bit.
Output [].[number of bits]	0 - 51	Indicate the number of bits to the right of the binary point. This parameter only applies to signed binary fractional buses.

Figure 83 shows a design example using the BusBuild block.

Figure 83. BusBuild Example



See “Block I/O Formats” on page 78 for information on the block I/O formats.

Bus Concatenation Block

This block concatenates two buses. The result is A + B bits wide, where A is the most significant bit (MSB) slice of the output bus and B is the least significant bit (LSB) slice of the output bus. Table 27 shows the block parameters.

Table 27. Bus Concatenation Parameters

Name	Value	Description
Bus A Width	1 - 51	Enter the width of the first bus to concatenate.
Bus B Width	1 - 51	Enter the width of the first bus to concatenate.
Output Is Signed	On or Off	Indicate whether the output bus is signed.

See "Block I/O Formats" on page 78 for information on the block I/O formats.

BusConversion Block

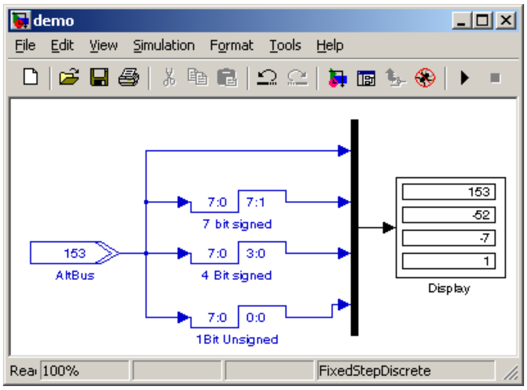
This block converts a bus from one node type to another. Table 28 shows the block parameters.

Table 28. BusConversion Parameters

Name	Value	Description
Input Bus Type	Signed Integer, Signed Fractional, Unsigned Integer	Choose the input bus type for the simulator, VHDL or both.
Input [number of bits].[]	1 - 51	Indicate the number of bits to the left of the binary point including the sign bit.
Input [].[number of bits]	1 - 51	Indicate the number of bits to the right of the binary point. This parameter only applies to signed binary fractional buses.
Output Bus Type	Signed Integer, Signed Fractional, Unsigned Integer	Choose the output bus type for the simulator, VHDL or both.
Output [number of bits].[]	1 - 51	Indicate the number of bits to the left of the binary point
Output [].[number of bits]	1 - 51	Indicate the number of bit on the right side of the binary point. This parameter only applies to signed binary fractional buses.
Input Bit Connected to Output MSB	0 - 51	Indicate which slice of the input bus to use. This parameter designates the starting point (LSB) of the slice.
Input Bit Connected to Output LSB	0 - 51	Indicate which slice of the input bus to use. This parameter designates the ending point (MSB) of the slice.
Round	On or Off	Indicate whether rounding should be turned on or off.
Saturate	On or Off	Indicate whether saturation should be turned on or off.

Figure 84 shows a design example using the BusConversion block.

Figure 84. BusConversion Example



See "Block I/O Formats" on page 78 for information on the block I/O formats.

ExtractBit Block

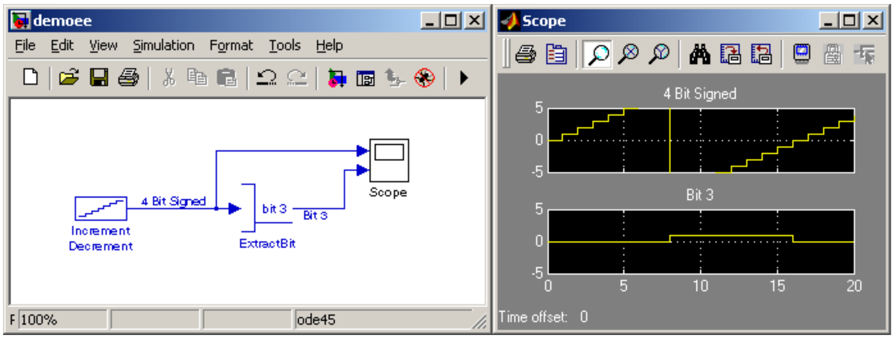
The ExtractBit block reads a Simulink bus in signed binary fractional format and outputs the bit specified with the **Extracted Bit** parameter. Table 29 shows the block parameters.

Table 29. ExtractBit Parameters

Name	Value	Description
Input [number of bits].[]	1 - 51	Indicate the number of bits to the left of the binary point, including the sign bit.
Input [].[number of bits]	0 - 51	Indicate the number of bits to the right of the binary point.
Extracted Bit	0 - 32	Indicate which bit to extract.

Figure 85 shows a design example using the ExtractBit block.

Figure 85. ExtractBit Example



See "Block I/O Formats" on page 78 for information on the block I/O formats.

GND Block

The GND block is a single bit that outputs a constant 0.

See "Block I/O Formats" on page 78 for information on the block I/O formats.

VCC Block

The VCC block is a single bit that outputs a constant 1.

See “Block I/O Formats” on page 78 for information on the block I/O formats.

Complex Signals Library

Like Simulink, DSP Builder supports native complex signal types. Using complex number notation simplifies the design of applications such as FFT, I-Q modulation, and complex filters.

☞ When connecting DSP Builder blocks to blocks from the Complex Signals library (e.g., connecting AltBus to Complex AddSub), you must use Real-Imag to Complex or Complex to Real-Imag blocks between the blocks. See Figure 87 for an example.

Butterfly Operator Block

The Butterfly Operator block performs the following arithmetic operation on complex signed integer numbers:

$A = a + bW$
$B = a - bW$

where a, b, W, A, and B are complex numbers (type signed integer) such as:

$a = x + jX$
$b = y+jY$
$W = v + jV$
$A = (x + yv) - YV + j(X + Yv + yV)$
$B = (x - yv) + YV + j(X - Yv - yV)$

This function operates with full bit width precision. The full bit width precision of A andB is 2 x [input bit width] + 2. The output bit width and output LSB bit parameters are used to specfy the bit slice used for the output ports A and B. For example, if the input bit width is 16, the output bit width is 16, and the output LSB is 4, the full precision is 34 bits and the output ports are A[19:4] and B[19:4]

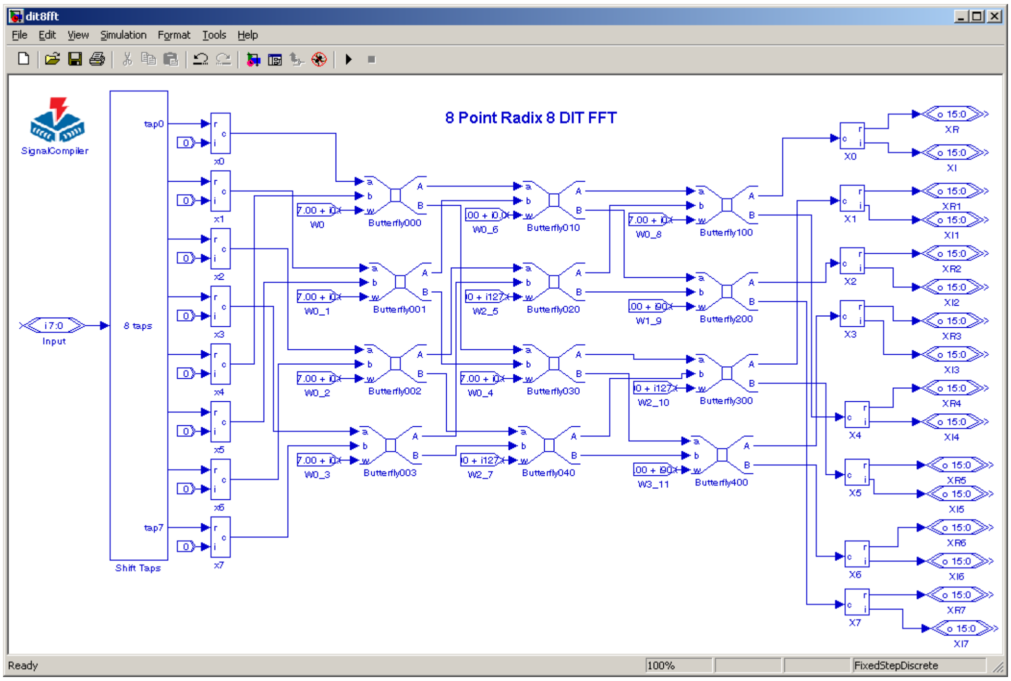
Table 30 shows the Butterfly Operator parameters.

Table 30. Butterfly Operator Parameters

Name	Value	Description
Input Bit Width (a, b, W)	4 - 51	Specify the bit width of the complex signed integer inputs a, b, and W.
Output Bit Width (A and B)	4 - 51	Specify the bit width of the complex signed integer outputs A and B.
Output LSB Bit	4 - 51	Specify the LSB bit of the output bus slice of the full resolution computation.

Figure 86 shows a design example using the Butterfly Operator block.

Figure 86. Butterfly Operator Example


dit8fft
File Edit View Simulation Format Tools Help
8 Point Radix 8 DIT FFT
SignalCompiler
Input
8 taps
tap0
tap7
Shift Taps
x0 x1 x2 x3 x4 x5 x6 x7
W0 W0_1 W0_2 W0_3 W0_6 W2_5 W0_4 W2_7 W0_8 W1_9 W2_10 W3_11
Butterfly000 Butterfly001 Butterfly002 Butterfly003 Butterfly010 Butterfly020 Butterfly030 Butterfly040 Butterfly100 Butterfly200 Butterfly300 Butterfly400
X0 X1 X2 X3 X4 X5 X6 X7
XR XI XR1 XI1 XR2 XI2 XR3 XI3 XR4 XI4 XR5 XI5 XR6 XI6 XR7 XI7
Ready
100%
FixedStepDiscrete

See "Block I/O Formats" on page 78 for information on the block I/O formats.

Complex AddSub Block

The Complex AddSub block performs output addition or subtraction of two scalar complex inputs.

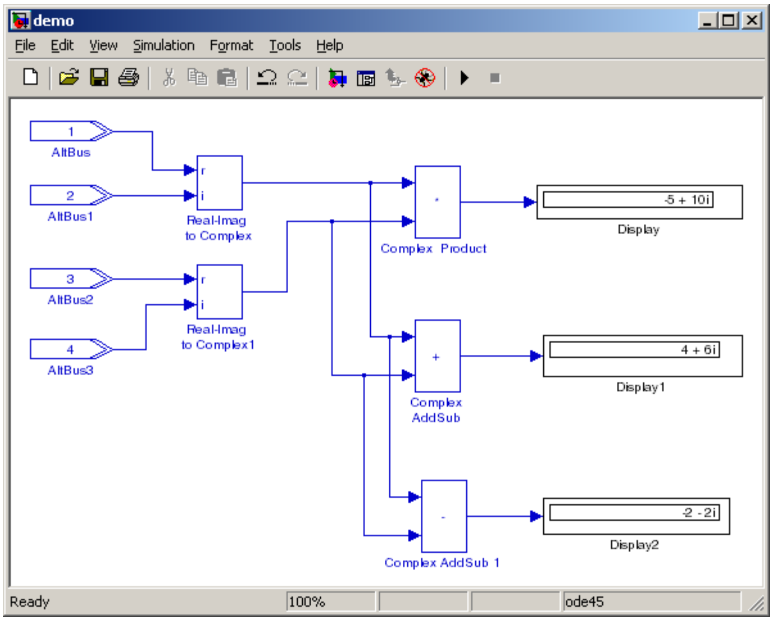
Table 31 shows the Complex AddSub parameters.

Table 31. Complex AddSub Parameters

Name	Value	Description
Arithmetic Operation	Add or Subtract	Choose which operation to perform.

Figure 87 shows a design example using the Complex Add Sub block.

Figure 87. Complex Add Sub Example



See "Block I/O Formats" on page 78 for information on the block I/O formats.

Complex Product Block

The Complex Product block performs output multiplication of two scalar complex inputs.

Table 32 shows the Complex Product parameters.

Table 32. Complex Product Parameters

Name	Value	Description
Arithmetic Operation	Multiply	Choose which operation to perform.

Figure 87 on page 151 shows a design example using the Complex Product block.

See "Block I/O Formats" on page 78 for information on the block I/O formats.

Complex Multiplexer Block

The Complex Multiplexer block is a 2 to 1 multiplexer for complex numbers. The select line (port number 3) is a non-complex scalar.

See "Block I/O Formats" on page 78 for information on the block I/O formats.

Complex Conjugate Block

The Complex Conjugate block outputs the conjugate or the inverse of the scalar complex inputs.

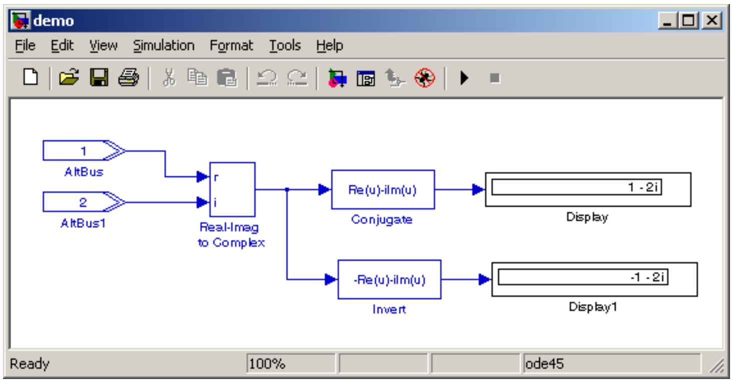
Table 33 shows the Complex Conjugate parameters.

Table 33. Complex Conjugate Parameters

Name	Value	Description
Operation	Conjugate, Inverse	Choose which operation to perform.

Figure 88 shows a design example using the Complex Conjugate block.

Figure 88. Complex Conjugate Example



See "Block I/O Formats" on page 78 for information on the block I/O formats.

Complex Delay Block

The Complex Delay block delays the incoming data by the amount specified by the `Depth` parameter. The input must be a complex number.

Table 34 shows the Complex Delay parameters.

Table 34. Complex Delay Parameters

Name	Value	Description
Depth	User Defined	Indicate the delay length of the block.

See "Block I/O Formats" on page 78 for information on the block I/O formats.

Complex Constant Block

The Complex Constant block outputs a fixed-point complex constant value.

Table 35 shows the Complex Constant parameters.

Table 35. Complex Constant Parameters

Name	Value	Description
Bus Type	Signed Integer, Signed Fractional, or Unsigned Integer	Choose the number format of the bus.
[number of bits].[]	1 - 51	Indicate the number of bits to the left of the binary point, including the sign bit. This parameter does not apply to single-bit buses.
[].[number of bits]	0 - 51	Indicate the number of bits to the right of the binary point. This parameter only applies to signed fractional buses.
Real Value	User Defined	Indicate the value of the real part of the constant.
Imaginary Value	User Defined	Indicate the value of the real part of the constant.

See "Block I/O Formats" on page 78 for information on the block I/O formats.

Complex to Real-Imag Block

The Complex to Real-Imag block constructs a fixed-point real and fixed-point imaginary output from a complex input.

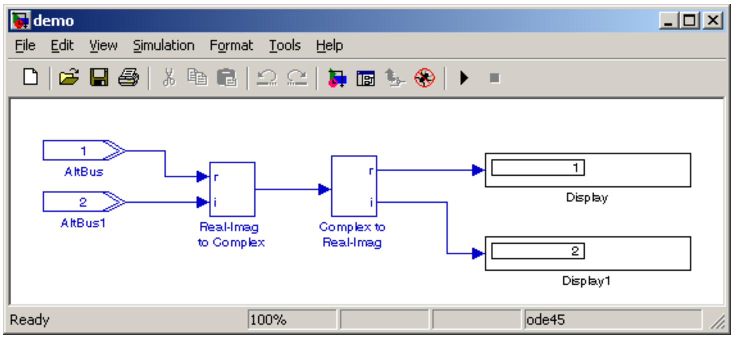
Table 36 shows the Complex to Real-Imag parameters.

Table 36. Complex to Real-Imag Parameters

Name	Value	Description
Bus Type	Signed Integer, Signed Fractional, Unsigned Integer	Choose the number format you wish to use for the bus.
Inputs [number of bits].[]	1 - 51	Select the number of data input bits to the left of the binary point, including the sign bit.
Inputs [].[number of bits]	0 - 51	Select the number of data input bits to the right of the binary point. This option is only available when **Signed Fractional** is selected.

Figure 89 shows a design example using the Complex to Real-Imag block.

Figure 89. Complex to Real-Imag Example



See "Block I/O Formats" on page 78 for information on the block I/O formats.

Real-Imag to Complex Block

The Real-Imag to Complex block constructs a fixed-point complex output from real and imaginary inputs.

Table 37 shows the Real-Imag to Complex parameters.

Table 37. Real-Imag to Complex Parameters

Name	Value	Description
Bus Type	Signed Integer, Signed Fractional, Unsigned Integer	Choose the number format you wish to use for the bus.
Inputs [number of bits].[]	1 - 51	Select the number of data input bits to the left of the binary point, including the sign bit.
Inputs [].[number of bits]	0 - 51	Select the number of data input bits to the right of the binary point. This option is only available when **Signed Fractional** is selected.

See "Block I/O Formats" on page 78 for information on the block I/O formats.

DSP Board Library

APEX DSP Board EP20K1500EBC 652-1X Library

The APEX DSP development board (professional version) is a prototyping platform that provides system designers with an economical solution for hardware and software verification. This rapid prototyping board enables the user to debug and verify both functionality and design timing. With two analog input and output channels per board and the ability to combine boards easily with right angle connectors, the board can be used to construct an extremely powerful processing system. Combined with DSP IP from Altera and the Altera Megafunction Partners Program (AMPP™) partners, the user can solve design problems that formerly required custom hardware and software solutions.

Refer to the *APEX DSP Development Kit Getting Started User Guide* for information on setting up the board and connecting it to your PC.

The blocks in this library provide a connection to analog-to-digital (A/D) converters, digital-to-analog (D/A) converters, LEDs, and switches on the APEX DSP development board (professional version). With these blocks, you do not have to make pin assignments to connect to these board components. When targeting the DSP development board EP20K1500EBC652-1X, the design must contain the DSP Board EP20K1500EBC652-1X configuration block at the top hierarchical level, to specify global clock and reset board connections. This library contains the following blocks (see Figure 90):

- DSP Board EP20K1500EBC652-1X Configuration—Board configuration
- A2D_0—Controls A/D converter 0 (J2)
- A2D_1—Controls A/D converter 1 (J3)
- D2A_0—Controls D/A converter 0 (J5)
- D2A_1—Controls D/A converter 1 (J6)
- LED0—Controls user LED 0 (D6)
- LED1—Controls user LED 1 (D7)
- LED2—Controls user LED 1 (D8)
- SWITCH0—Controls push-button switch 0 (SW0)
- SWITCH1—Controls push-button switch 1 (SW1)
- SWITCH2—Controls push-button switch 2 (SW2)
- SWITCH3—Controls slider switch 3 (SW3)
- SWITCH4—Controls slider switch 4 (SW3)
- SWITCH5—Controls slider switch 5 (SW3)

- SWITCH6—Controls slider switch 6 (SW3)
- Digital I/O Connectors—Choose to which bit in the bank of jumpers you want to connect

Figure 90. APEX DSP EP20K1500EBC652-1X Blocks

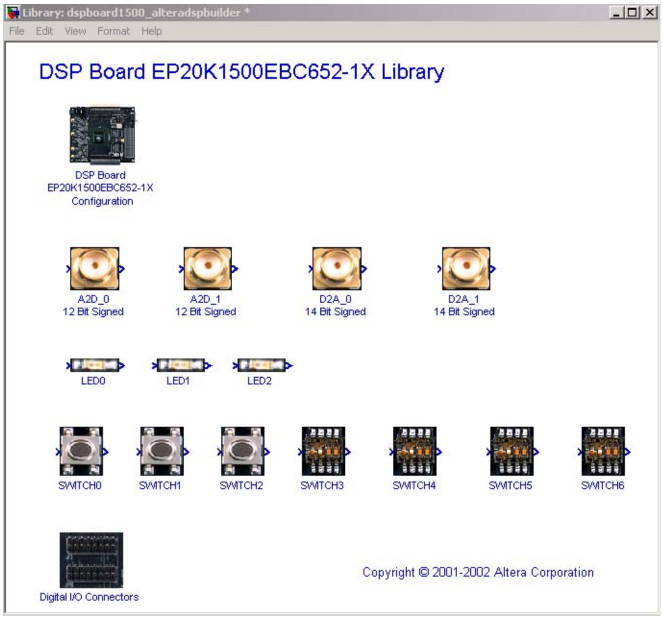
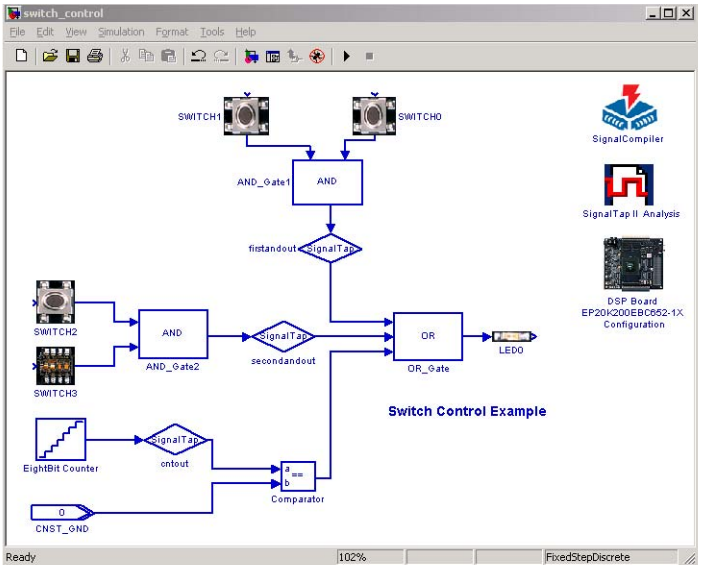

Library: dspboard1500_alteradspbuilder *
File Edit View Format Help
DSP Board EP20K1500EBC652-1X Library
DSP Board EP20K1500EBC652-1X Configuration
A2D_0 12 Bit Signed
A2D_1 12 Bit Signed
D2A_0 14 Bit Signed
D2A_1 14 Bit Signed
LED0
LED1
LED2
SWITCH0
SWITCH1
SWITCH2
SWITCH3
SWITCH4
SWITCH5
SWITCH6
Digital I/O Connectors
Copyright © 2001-2002 Altera Corporation

Figure 91 shows an example switch controller design using the SignalTap II and board blocks. Based on the user-controlled switches and the value of the incrementer, an LED on the APEX DSP development board turns on or off.

Figure 91. Switch Controller Design Using DSP Board Blocks



APEX DSP Board EP20K200EBC652-1X Library

The APEX DSP development board (starter version) is a prototyping platform that provides system designers with an economical solution for hardware and software verification. This rapid prototyping board enables the user to debug and verify both functionality and design timing. With two analog input and output channels per board and the ability to combine boards easily with right angle connectors, the board can be used to construct an extremely powerful processing system. Combined with DSP IP from Altera and the Altera Megafunction Partners Program (AMPP™) partners, the user can solve design problems that formerly required custom hardware and software solutions.

Refer to the *APEX DSP Development Kit Getting Started User Guide* for information on setting up the board and connecting it to your PC.

The blocks in this library provide a connection to analog-to-digital (A/D) converters, digital-to-analog (D/A) converters, LEDs, and switches on the APEX DSP development board (starter version). With these blocks, you do not have to make pin assignments to connect to these board components. When targeting the DSP development board EP20K2000EBC652-1X, the design must contain the DSP Board EP20K200EBC652-1X configuration block at the top hierarchical level, to specify global clock and reset board connections. This library contains the following blocks (see Figure 92):

- DSP Board EP20K200EBC652-1X Configuration—Board configuration
- A2D_0—Controls A/D converter 0 (J2)
- A2D_1—Controls A/D converter 1 (J3)
- D2A_0—Controls D/A converter 0 (J5)
- D2A_1—Controls D/A converter 1 (J6)
- LED0—Controls user LED 0 (D6)
- LED1—Controls user LED 1 (D7)
- LED2—Controls user LED 1 (D8)
- SWITCH0—Controls push-button switch 0 (SW0)
- SWITCH1—Controls push-button switch 1 (SW1)
- SWITCH2—Controls push-button switch 2 (SW2)
- SWITCH3—Controls slider switch 3 (SW3)
- SWITCH4—Controls slider switch 4 (SW3)
- SWITCH5—Controls slider switch 5 (SW3)
- SWITCH6—Controls slider switch 6 (SW3)
- Digital I/O Connectors—Choose to which bit in the bank of jumpers you want to connect

Figure 92. APEX DSP EP20K200EBC652-1X Blocks

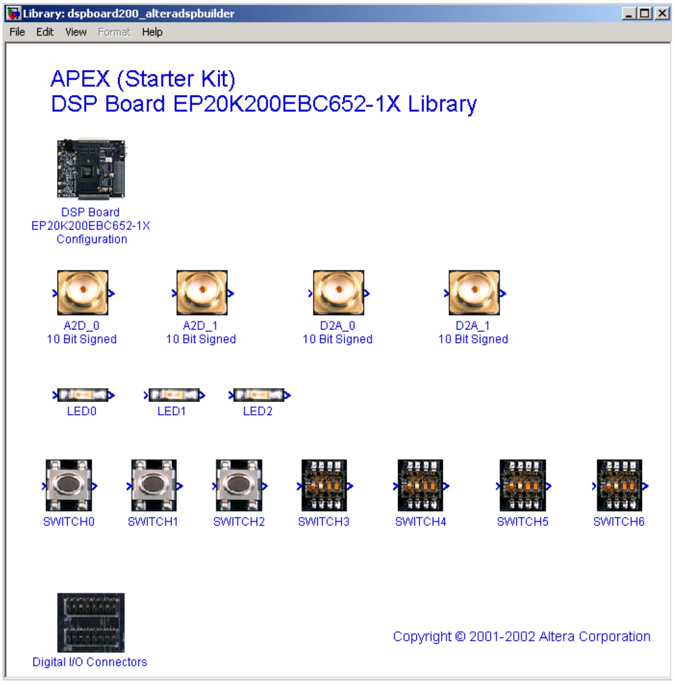
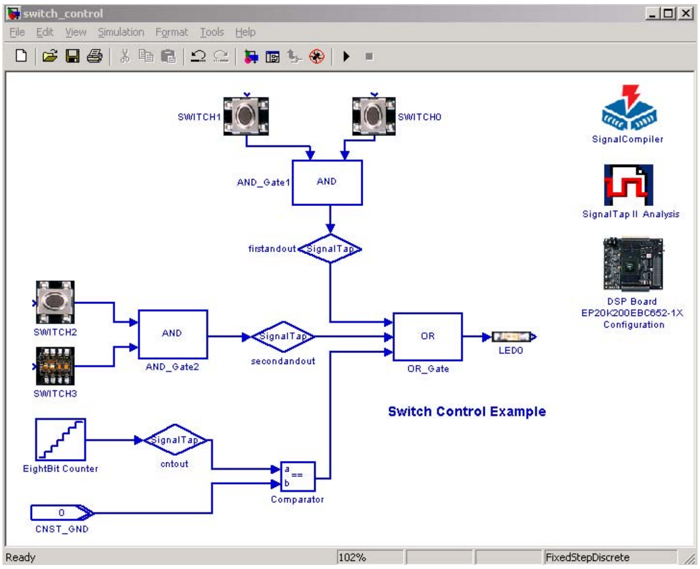

Library: dspboard200_alteradspbuilder
File Edit View Format Help
APEX (Starter Kit)
DSP Board EP20K200EBC652-1X Library
DSP Board EP20K200EBC652-1X Configuration
A2D_0 10 Bit Signed
A2D_1 10 Bit Signed
D2A_0 10 Bit Signed
D2A_1 10 Bit Signed
LED0
LED1
LED2
SWITCH0
SWITCH1
SWITCH2
SWITCH3
SWITCH4
SWITCH5
SWITCH6
Digital I/O Connectors
Copyright © 2001-2002 Altera Corporation

Figure 93 shows an example switch controller design using the SignalTap II and board blocks. Based on the user-controlled switches and the value of the incrementer, an LED on the APEX DSP development board turns on or off.

Figure 93. Switch Controller Using DSP Board Blocks



Stratix DSP Board EP1S25 Library

The Stratix EP1S25 DSP development board is a prototyping platform that provides system designers with an economical solution for hardware and software verification. This rapid prototyping board enables the user to debug and verify both functionality and design timing. With two analog input and output channels per board and the ability to combine boards easily with right angle connectors, the board can be used to construct an extremely powerful processing system. Combined with DSP IP from Altera and the Altera Megafunction Partners Program (AMPP™) partners, the user can solve design problems that formerly required custom hardware and software solutions.

Refer to the *DSP Development Kit, Stratix Edition Getting Started User Guide* for information on setting up the board and connecting it to your PC.

The blocks in this library provide a connection to analog-to-digital (A/D) converters, digital-to-analog (D/A) converters, LEDs, and switches on the Stratix EP1S25 DSP development board. With these blocks, you do not have to make pin assignments to connect to these board components. When targeting the DSP development board EP1S25F780C5, the design must contain the Stratix DSP Board 1S25 Configuration block at the top hierarchical level, to specify global clock and reset board connections. This library contains the following blocks (see Figure 94):

- DSP Board EP1S25F780C5 Configuration—Board configuration
- A2D_0—Controls A/D converter 0 (J2)
- A2D_1—Controls A/D converter 1 (J3)
- D2A_0—Controls D/A converter 0 (J5)
- D2A_1—Controls D/A converter 1 (J6)
- LED0—Controls user LED 0 (D6)
- LED1—Controls user LED 1 (D7)
- LED2—Controls user LED 1 (D8)
- SWITCH0—Controls push-button switch 0 (SW0)
- SWITCH1—Controls push-button switch 1 (SW1)
- SWITCH2—Controls push-button switch 2 (SW2)
- SWITCH3—8 dipswitches
- DEBUG_A—Controls debugging port A
- DEBUG_B—Controls debugging port B
- PROTO—Controls the prototyping area I/O
- EVALIO_IN
- EVALIO_OUT
- RS2323 TIN—RS-232 serial port transmit input
- RS2323 ROUT—RS-232 serial port receive output

Figure 94. Stratix DSP Board EP1S25 Blocks

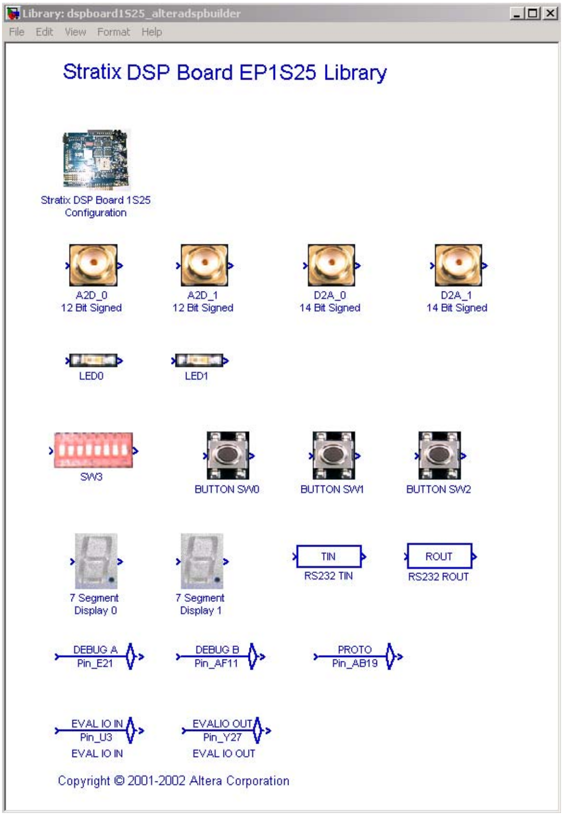
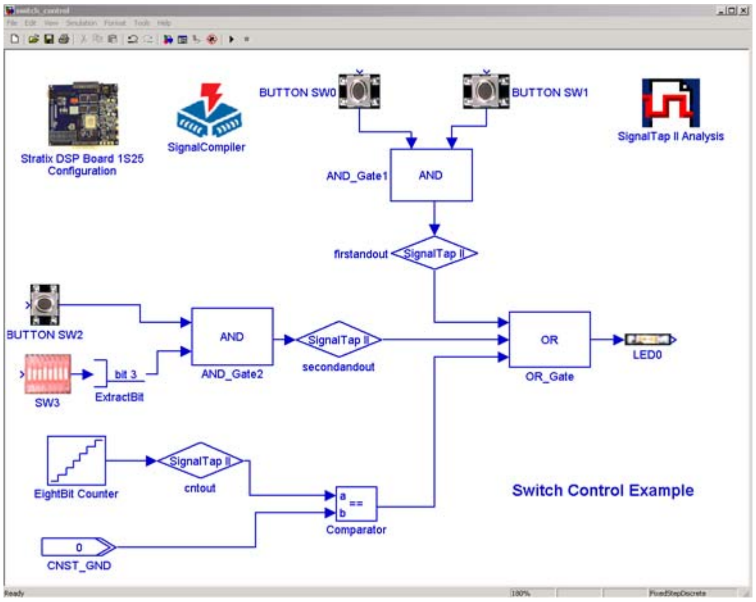

Library: dspboard1S25_alteradspbuilder
File Edit View Format Help
Stratix DSP Board EP1S25 Library
Stratix DSP Board 1S25 Configuration
A2D_0 12 Bit Signed
A2D_1 12 Bit Signed
D2A_0 14 Bit Signed
D2A_1 14 Bit Signed
LED0
LED1
SW3
BUTTON SW0
BUTTON SW1
BUTTON SW2
7 Segment Display 0
7 Segment Display 1
TIN RS232 TIN
ROUT RS232 ROUT
DEBUG A Pin_E21
DEBUG B Pin_AF11
PROTO Pin_AB19
EVAL IO IN Pin_U3 EVAL IO IN
EVALIO OUT Pin_Y27 EVAL IO OUT
Copyright © 2001-2002 Altera Corporation

Figure 95 shows an example switch controller design using the SignalTap II and board blocks. Based on the user-controlled switches and the value of the incrementer, an LED on the Stratix EP1S25 DSP development board turns on or off.

Figure 95. Switch Controller Using DSP Board Blocks



Notes:

This library contains boolean operators, which you can use for combinatorial functions.

Case Statement Block

The Case Statement block compares the input signal (which must be a signed or unsigned integer) with a set of case values. For each case, a single-bit output is generated. You can implement as many cases as you wish; append a comma (,) after each case. Additionally, you can have multiple conditons for each case; use a pipe (|) to separate the conditions. For example, for four cases with the first of which having 2 conditions, you would enter `1|2,3,4,5,` in the **Case Values** box. Table 38 shows the Case Statement block parameters.

Table 38. Case Statement Parameters

Name	Value	Description
Default Case	On or Off	Turn on this option if you want the `default` output signal to go high when the other outputs are false.
Case Values	User Specified	Indicate the values with which you want to compare the input. Include a comma after each value.

Figure 96 shows an example model using the Case Statement block.

Figure 96. Case Statement Example

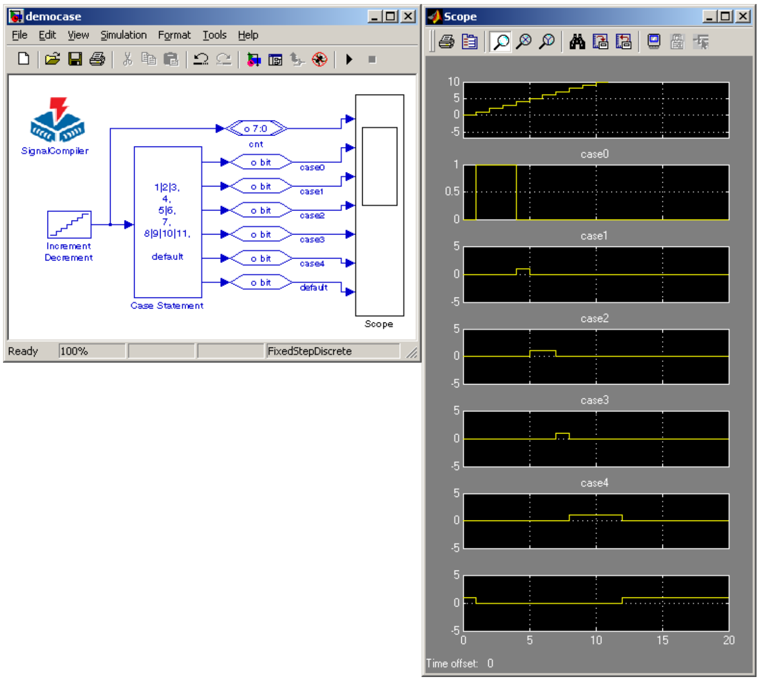


Figure 97 shows an excerpt of the VHDL that SignalCompiler generates from the model.

Figure 97. Combinatorial Case Statement VHDL Output

```
p0:process(A1W)
    begin
        case A1W is
            when "000001"|"000010"|"000011" =>
                    A0W <= '1';
                    A2W <= '0';
                    A6W <= '0';
                    A5W <= '0';
                    A4W <= '0';
                    A3W <= '0';
            when "000100" =>
                    A0W <= '0';
                    A2W <= '1';
                    A6W <= '0';
                    A5W <= '0';
                    A4W <= '0';
                    A3W <= '0';
.
.
.
            when "001000"|"001001"|"001010"|"001011" =>
                    A0W <= '0';
                    A2W <= '0';
                    A6W <= '0';
                    A5W <= '0';
                    A4W <= '1';
                    A3W <= '0';
            when others=>
                    A0W <= '0';
                    A2W <= '0';
                    A6W <= '0';
                    A5W <= '0';
                    A4W <= '0';
                    A3W <= '1';
        end case;
end process;
```

In Simulink, each wire is named `A`<*number*>`W` where <*number*> is auto-generated.

See "Block I/O Formats" on page 78 for information on the block I/O formats.

If Statement Block

The If Statement block returns a boolean result based on the IF condition equation. The input arguments—*a, b, c, d, e, f, g, h, i*, or *j*—must be signed or unsigned integers. You can use any number of parentheses. The operators are given in Table 39.

Table 39. Supported If Statement Operators

Operator	Operation
+	OR
&	AND
$	XOR
=	Equal To
~	Not Equal To
>	Greater Than
<	Less Than

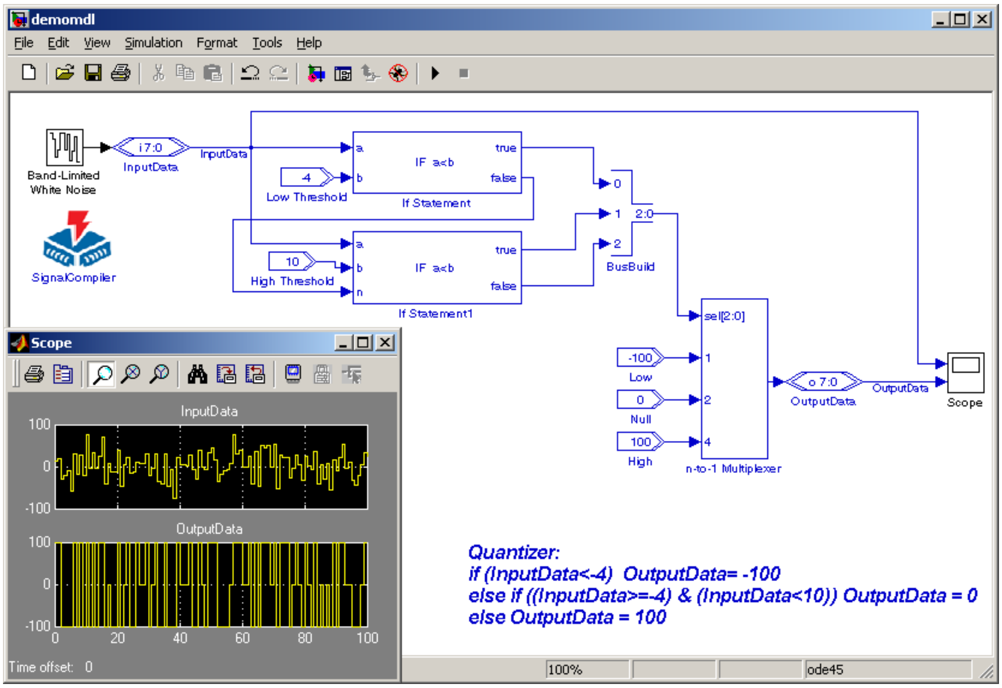
Table 40 shows the If Statement block parameters.

Table 40. If Statement Parameters

Name	Value	Description
Number of Inputs	2 - 10	Indicate the number of inputs to the If Statement.
IF	User Defined	Indicate the if condition using any of the following operators: +, &, $, =, ~, >, or < and the variables *a, b, c, d, e, f, g, h, i,* or *j.*
ELSE Output	On or Off	This option turns on the `false` output signal, which goes high if the condition evaluated by the If Statement is false.
ELSE IF Input	On or Off	This option enables the `n` input signal (where `n` is the ELSE IF input), which you can use to cascade multiple IF Statement blocks together.

Figure 98 on page 173 shows an example using the If Statement block.

Figure 98. If Statement Example



```
Quantizer:
if (InputData<-4)  OutputData= -100
else if ((InputData>=-4) & (InputData<10)) OutputData = 0
else OutputData = 100
```

See "Block I/O Formats" on page 78 for information on the block I/O formats.

Logical Bit Operator Block

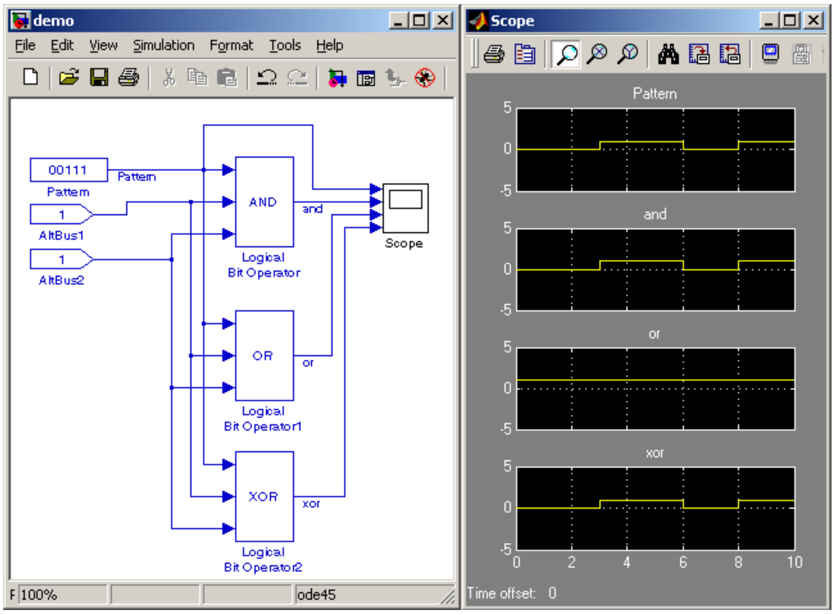
This block performs logical operation on single-bit inputs. You can specify a variable number of inputs. If the integer is positive, it is interpreted as a boolean 1, otherwise it is interpreted as 0. The number of inputs is variable.Table 41 shows the block parameters.

Table 41. Logical Bit Operator Parameters

Name	Value	Description
Logical Operator	AND, OR, XOR, NAND, NOR, or NOT	Indicate which operator you wish to use.
Number of Inputs	1 - 16	Indicate the number of inputs. This parameter defaults to 1 if the NOT logical operator is selected.

Figure 99 shows an example using the Logical Bit Operator block.

Figure 99. Logical Bit Operator Example



See "Block I/O Formats" on page 78 for information on the block I/O formats.

Logical Bus Operator Block

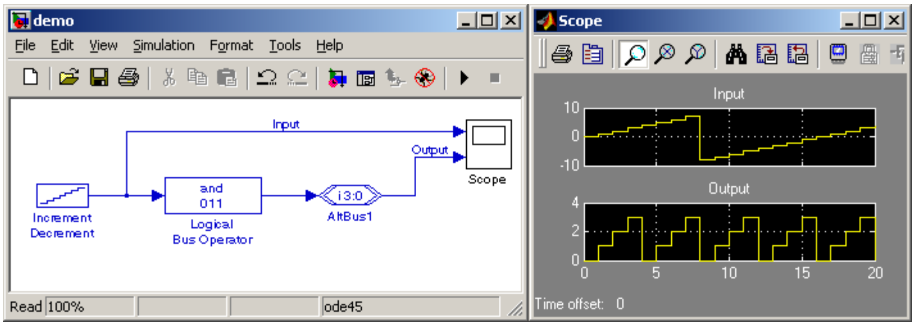
This block performs logical operations on a bus such as AND, OR, XOR, or invert. You can perform masking by entering in decimal notation or shifting. Table 44 shows the block parameters.

Table 42. Logical Bus Operator Parameters

Name	Value	Description
[number of bits].[]	1 - 51	Indicate the number of bits to the left of the binary point, including the sign bit.
[].[number of bits]	0 - 51	Indicate the number of bits to the right of the binary point.
Logic Operation	AND, OR, XOR, Invert, Shift Left, Shift Right, Rotate Left, Rotate Right	Indicate the logical operation to perform.
Number of Bits to Shift	User Defined	Indicate how many bits you wish to shift.

Figure 100 shows an example using the Logical Bus Operator block.

Figure 100. Logical Bus Operator Example



See "Block I/O Formats" on page 78 for information on the block I/O formats.

LUT Block

The LUT block stores data as $2^{(\text{address width})}$ words of data in a look-up table. The values of the words are specified in the data vector field as a MATLAB array.

Depending on the look-up table size, the synthesis tool may use logic cells or embedded array blocks (EABs)/embedded system blocks (ESBs)/TriMatrix memory. You should use a manual synthesis and compilation flow (see "Manual Synthesis & Compilation" on page 45) if you want to control the memory implementation.

If you want to use a Hexadecimal File (**.hex**) to store data, use the ROM EAB block not the LUT block.

Table 43 shows the block parameters.

Table 43. LUT Parameters

Name	Value	Description
Output Data [number of bits].[]	1 - 51	Indicate the number of data bits stored on the left side of the binary point including the sign bit.
Output Data [].[number of bits]	0 - 51	Indicate the number of data bits stored on the right side of the binary point.
Address Width	2 - 16	Indicate the address width.
MATLAB Array	User Defined	This field must be a one-dimensional MATLAB array with a length smaller than 2 to the power of the address width. You can specify the array in the MATLAB **Command Window** using a variable, or you can specify it directly in the **MATLAB Array** box.
Use LPM	On or Off	If you turn on the **Use LPM** option, the SignalCompiler will implement the look-up table using the `lpm_rom` library of parameterized modules (LPM) function. If you turn off this option, SignalCompiler implements the look-up table using Case conditions. Altera recommends that you turn on the **Use LPM** option for large look-up tables, e.g., greater than 8 bits.
Register Address	On of Off	When on, the input address bus is generated. If you are targeting either Stratix or Cylone devices and using the Use LPM option, you must turn on the register address option.

See "Block I/O Formats" on page 78 for information on the block I/O formats.

n-to-1 Multiplexer Block

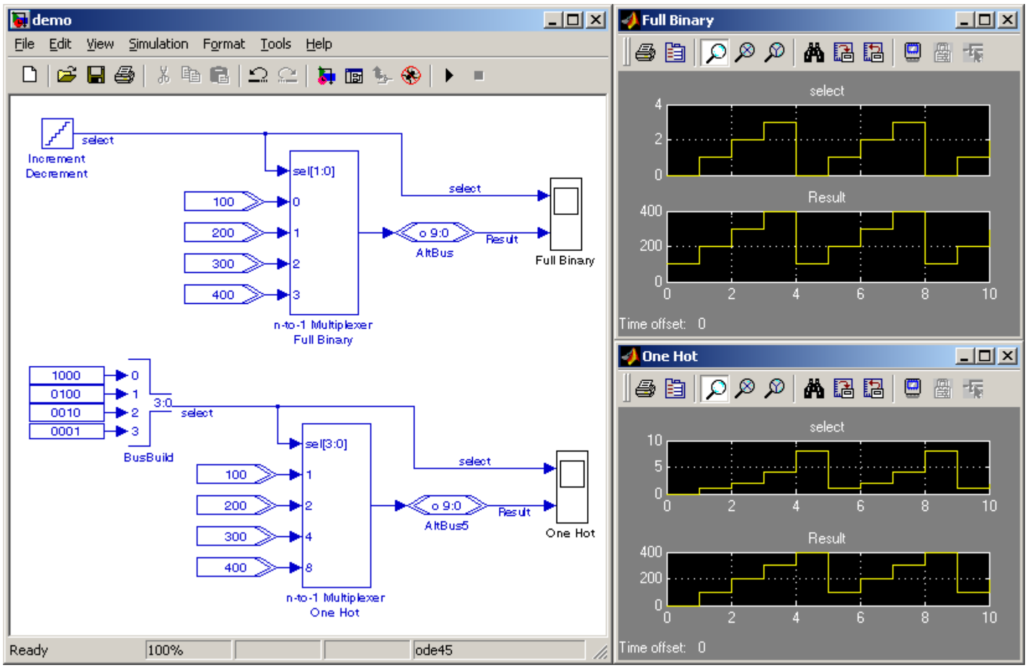
The n-to-1 Multiplexer block is a *n*-to-1 full binary bus multiplexer with one select control. The output width of the multiplexer is equal to the maximum width of the input data lines. The block works on any data type and sign extends the inputs if there is a bit width mismatch. Table 44 shows the block parameters.

Table 44. n-to-1 Multiplexer Parameters

Name	Value	Description
Number of Input Data Lines	2 to 10	Choose how many inputs you want the multiplexer to have.
One Hot Select Bus	On or Off	Indicate whether you want to use one-hot selection for the bus select signal instead of full binary.

Figure 101 shows an example using the n-to-1 Multiplexer block.

Figure 101. n-to-1 Multiplexer Example



See "Block I/O Formats" on page 78 for information on the block I/O formats.

Notes:

Rate Change Library

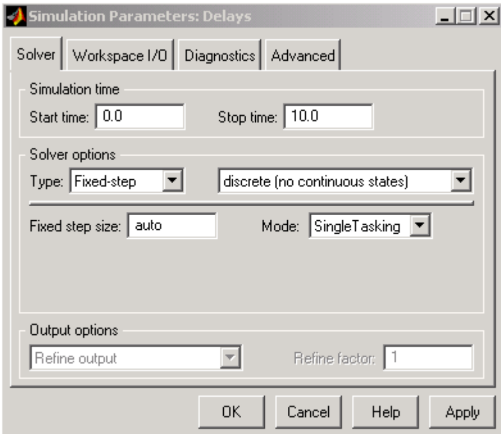
This library contains the blocks that allow you to control the clock assignment to registered DSP Builder blocks, such as Delay or Increment Decrement blocks.

To maintain cycle accuracy between Simulink and the VHDL domain, make the following settings in the **Solver** tab of the Simulink block:

- Choose **Fixed-step** from the **Type** list box under **Solver options**.
- Choose **discrete (no continuous state)** under **Solver options**.
- Choose **Single Tasking** from the **Mode** list box.

See Figure 102.

Figure 102. Set the Simulation Parameters



Start time: 0.0
Stop time: 10.0
Type: Fixed-step
discrete (no continuous states)
Fixed step size: auto
Mode: SingleTasking
Refine output
Refine factor: 1

PLL Block

The DSP Builder uses the PLL block to synthesize a clock signal that is based on a reference clock. PLLs have become an important building block of most high-speed digital systems today. Their use ranges from improving timing as zero delay lines to full-system clock synthesis. Stratix devices offer the most advanced on-chip PLL features, which were previously offered only by the most complex discrete devices.

Each PLL has multiple outputs that can source any of the 40 system clocks in the Stratix devices, which gives you complete control over your clocking needs. The PLLs offer full frequency synthesis capability (the ability to multiply up or divide down the clock frequency) and phase shifting for optimizing I/O timing. Additionally, the PLLs have high-end features such as programmable bandwidth, spread spectrum, and clock switchover.

The PLL block generates internal clocks that operate at frequencies that are multiples of the frequency of the system clock. The Cyclone and Stratix PLLs can simultaneously multiply and divide the reference clock. The PLL block checks the parameters' validity on the fly.

The PLL block supports the following device families:

- Stratix
- Cyclone

For more information on the built-in PLLs, see *AN 200: Using PLLs in Stratix Devices PLL & Timing Glossary*.

Table 45 shows the number of clock outputs that are available for each family.

Table 45. Device Clock Outputs

Device	Number of Clock Outputs
Stratix	6
Cyclone	6

If you use the PLL block, the following restrictions apply:

- The design must contain a single PLL instance at the top level
- Each output clock of the PLL has a zero degree phase shift and 50% duty cycle
- All DSP Builder block Simulink sample times must equal one of the PLL's output clock periods

Table 46 shows the PLL parameters.

Table 46. PLL Parameters

Name	Value	Description
Input Clock Frequency	See AN 200	Input reference clock.
Number of Output Clocks	1 to 6	Number of PLL clock outputs.
Clock Frequency multiplication factor	See AN 200	Multiply the reference clock by this value.
Clock Frequency division factor	See AN 200	Divide the reference clock by this value.

See “Block I/O Formats” on page 78 for information on the block I/O formats.

Multi-Rate DFF Block

The DSP Builder uses the multi-rate DFF block to synchronize data path intersections involving multiple rates.

Table 47 shows the multi-rate DFF parameters.

Table 47. Multi-Rate DFF Parameters

Name	Value	Description
Use Control Inputs	On or off	Indicate whether you want to use additional control inputs (clock enable and reset).
Clock Source	1 to 6	Specify PLL output clock source to the clock pin of the multi-rate DFF block. When the design does not contain PLLs, the DSP Builder connects the multi-rate DFF block clock pin to the main single clock.

See "Block I/O Formats" on page 78 for information on the block I/O formats.

Tsamp Block

The DSP Builder uses the Tsamp block to specify the sample time period of the internal data path source.

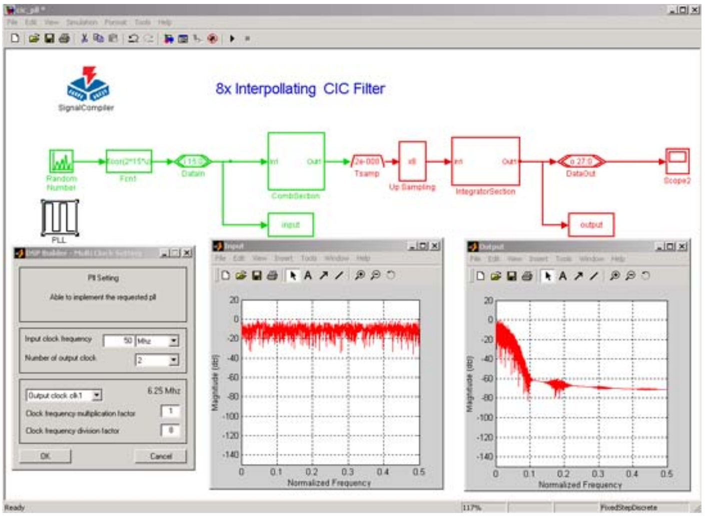
Table 48 shows the Tsamp parameters.

Table 48. Tsamp Parameters

Name	Value	Description
Sample Time Period	Any	Specified the sample time period in seconds.

Figure 103 shows an example design with the Tsamp block. The design is available in the **/designexamples/cicfiletr/** directory.

Figure 103. Tsamp Example



Pll Setting

Able to implement the requested pll

Input clock frequency 50 Mhz

Number of output clock 2

Output clock clk1 6.25 Mhz

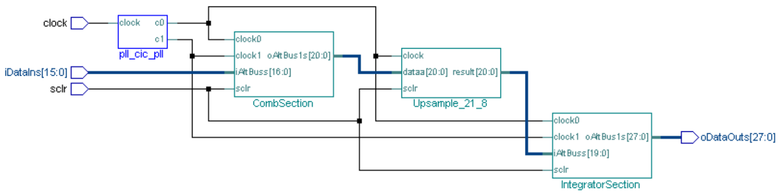
Clock frequency multiplication factor 1

Clock frequency division factor 8

OK Cancel

Figure 104 shows the Tsamp example design in the LeonardoSpectrum software.

Figure 104. LeonardoSpectrum Representation of Tsamp Example



See "Block I/O Formats" on page 78 for information on the block I/O formats.

SOPC Ports Library

The Nios® embedded processor is an industry-leading soft-core embedded processor optimized for the high-performance, high-bandwidth needs of network, telecommunications, and mass storage applications. Part of the Excalibur™ embedded processor solutions, the Nios embedded processor offers a variety of features to expand the performance and flexibility of system-on-a-chip (SOPC) designs. The Nios embedded processor includes a RISC CPU and the industry-standard GNUPro software from Cygnus, a Red Hat company.

You can use the blocks in the SOPC Ports library to build a custom logic block that works with the SOPC Builder and Nios embedded processor designs. The SOPC Builder supports two general types of custom logic blocks:

- Peripherals that use the Avalon bus
- Custom instructions that extend the function of the Nios arithmetic logic unit (ALU) and instruction set

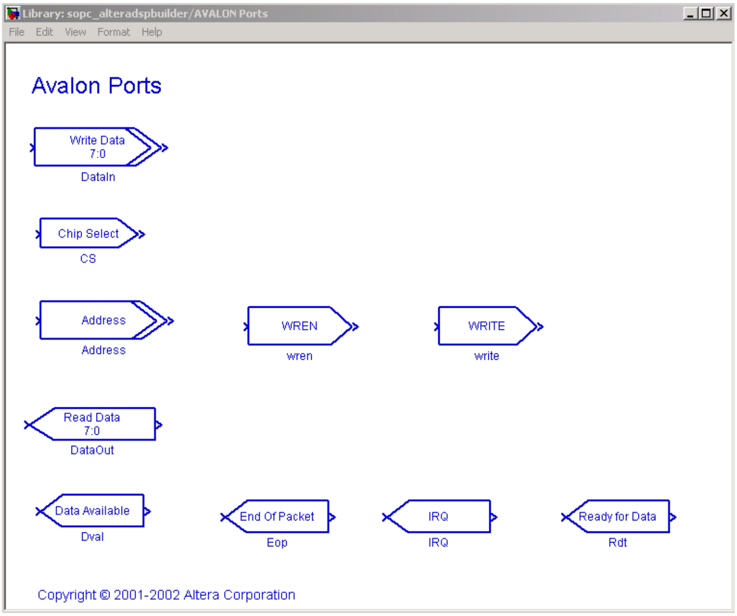
Avalon Ports

This section contains a description of the Avalon Ports library and a walkthrough.

Avalon Ports Block Description

You use the blocks in the Avalon Ports library to build Nios peripherals that connect to the Avalon bus. The Avalon Ports blocks automate the process of specifying slave ports that are compatible with the Avalon bus. After you build a model of your DSP Builder peripheral, add blocks from the Avalon Ports library to control the peripheral's inputs and outputs. Figure 105 shows the blocks in the Avalon Ports library.

Figure 105. Avalon Ports Library



When you convert your model to VHDL, the SignalCompiler creates a file, **class.ptf**, in your working directory. The **class.ptf** file is a text file that decribes:

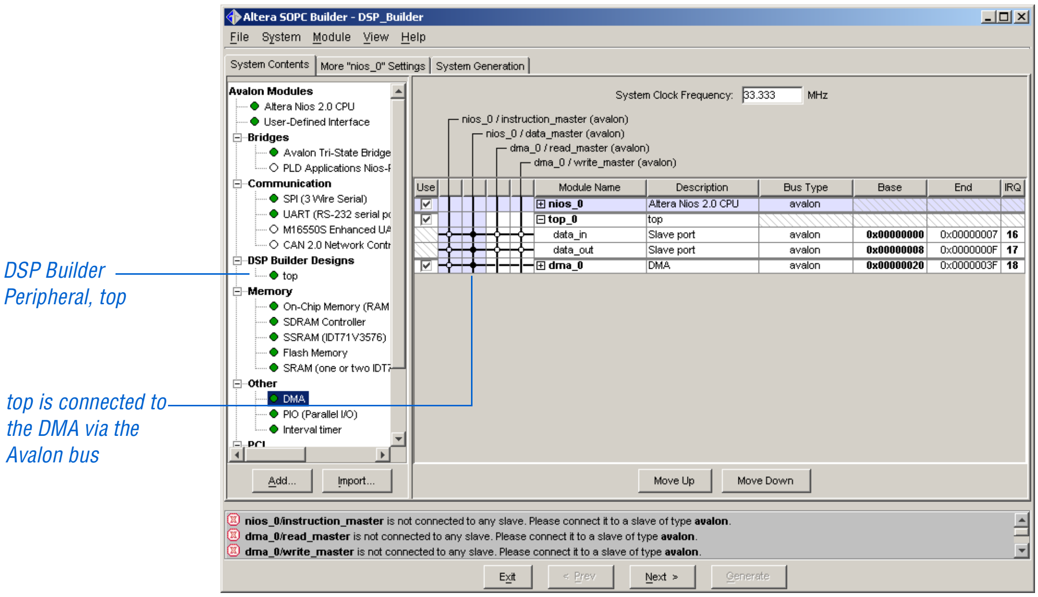
- Parameters that define the peripheral's structure and/or functionality
- The peripheral's slave role
- The peripheral's ports (such as read enable, read data, write enable, write data)

- The arbitration mechanism for each slave port that can be accessed by multiple master ports

For more information on the Avalon bus and PTF files, refer to the *Avalon Bus Specification Reference Manual*.

After you synthesize your model and compile it in the Quartus II software, you can add it to your Nios system using the SOPC Builder. Your peripheral appears in the SOPC Builder peripherals listing. Figure 106 shows the SOPC Builder with a Nios CPU, a DSP Builder-created peripheral named **top**, and a DMA peripheral that connects to **top**.

Figure 106. SOPC Builder with DSP Builder Peripheral



System Clock Frequency: 33.333 MHz

Use	Module Name	Description	Bus Type	Base	End	IRQ
✓	**nios_0**	Altera Nios 2.0 CPU	avalon			
✓	**top_0**	top				
	data_in	Slave port	avalon	**0x00000000**	0x00000007	**16**
	data_out	Slave port	avalon	**0x00000008**	0x0000000F	**17**
✓	**dma_0**	DMA	avalon	**0x00000020**	0x0000003F	**18**

nios_0/instruction_master is not connected to any slave. Please connect it to a slave of type **avalon**.
dma_0/read_master is not connected to any slave. Please connect it to a slave of type **avalon**.
dma_0/write_master is not connected to any slave. Please connect it to a slave of type **avalon**.

For the peripheral to appear in the SOPC Builder, the working directory for your SOPC Builder project must be the same as your DSP Builder working directory.

Refer to the *Nios Tutorial* for information on using the SOPC Builder to create Nios designs.

Figure 107 shows the design flow using DSP Builder and SOPC Builder.

Figure 107. DSP Builder & SOPC Builder Design Flow

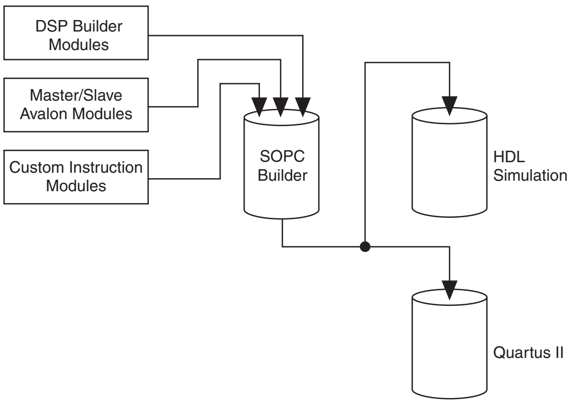
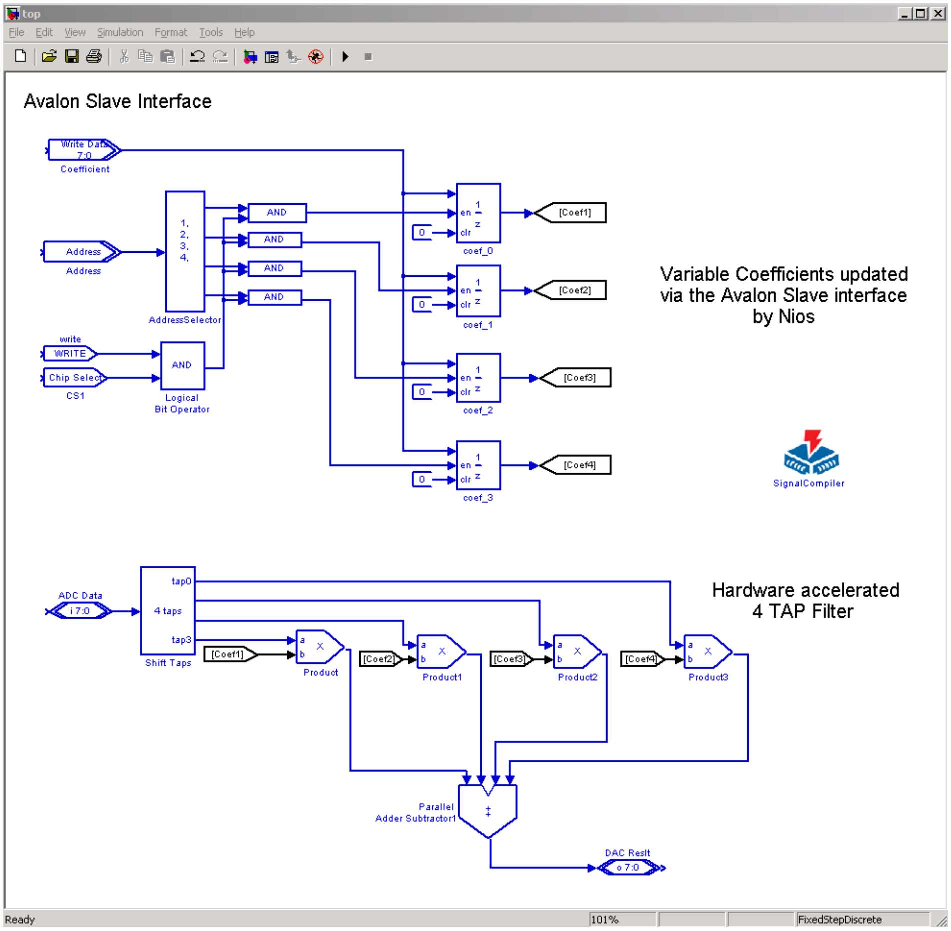


Figure 108 shows an example model using blocks from the Avalon Ports library.

Figure 108. Avalon Ports Example



Avalon Ports Walkthrough

This walkthrough describes how to interface your design built using the Avalon Ports library as a custom peripheral to the Nios embedded processor in SOPC Builder. In this walkthrough, the **toparc.mdl** design is included as a peripheral to the Avalon bus.

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

This walkthrough involves the following steps:

1. "Open the Walkthrough Example Design" on page 190.
2. "Turn On the SOPC Option in SignalCompiler & Generate VHDL & PTF Files" on page 191.
3. "Instantiating your Design as a Custom Peripheral to the Nios Embedded Processor in SOPC Builder" on page 193.

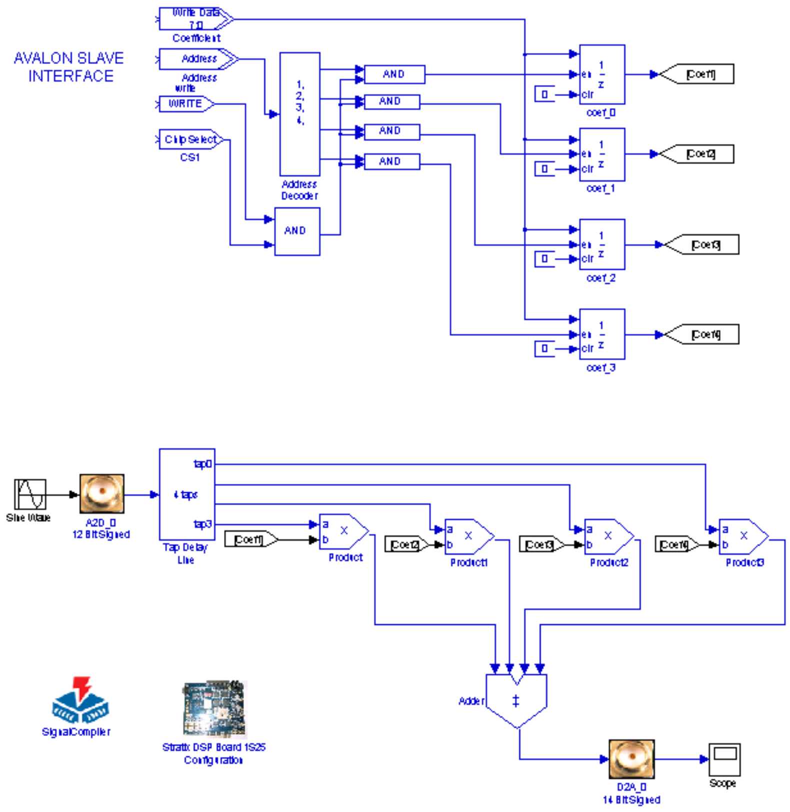
Open the Walkthrough Example Design

To open the example design, perform the following steps:

1. Choose **Open** (File menu) in the MATLAB software.
2. Browse to the **C:\DSPBuilder\DesignExamples\SOPCPort** directory.
3. Select the **toparc.mdl** file and click **Open**.

Figure 109 shows **toparc.mdl**.

Figure 109. toparc.mdl Example Design

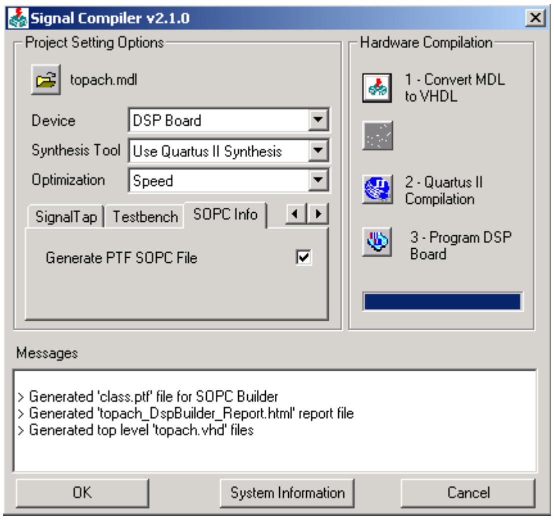


Turn On the SOPC Option in SignalCompiler & Generate VHDL & PTF Files

When you use the Avalon Ports blocks in your design, the SignalCompiler can generate a **class.ptf** file for use in SOPC Builder. To generate a **class.ptf** file within the SignalCompiler, perform the following steps:

1. Double-click the SignalCompiler block.
2. Click **Analyze**.
3. Click the right arrow to scroll the tabs to the right.
4. Click the **SOPC Info** tab (see Figure 110).

Figure 110. SOPC Info Tab in the SignalCompiler



```
> Generated 'class.ptf' file for SOPC Builder
> Generated 'topach_DspBuilder_Report.html' report file
> Generated top level 'topach.vhd' files
```

5. Turn on the **Generate PTF SOPC File** option. When you turn on this option, the SignalCompiler generates a **class.ptf** file for your design that SOPC Builder detects automatically.

6. Choose **Use Quartus II Synthesis** from the **Synthesis Tool** list box.

7. Click **1 – Convert MDL to VHDL**. SignalCompiler generates a VHDL and a **class.ptf** file for your design.

8. Click **2 – Quartus II Compilation**.

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

The VHDL file for this design is in the directory:

C:\DSPBuilder\DesignExamples\SOPCPort

The **class.ptf** file is located in the directory:

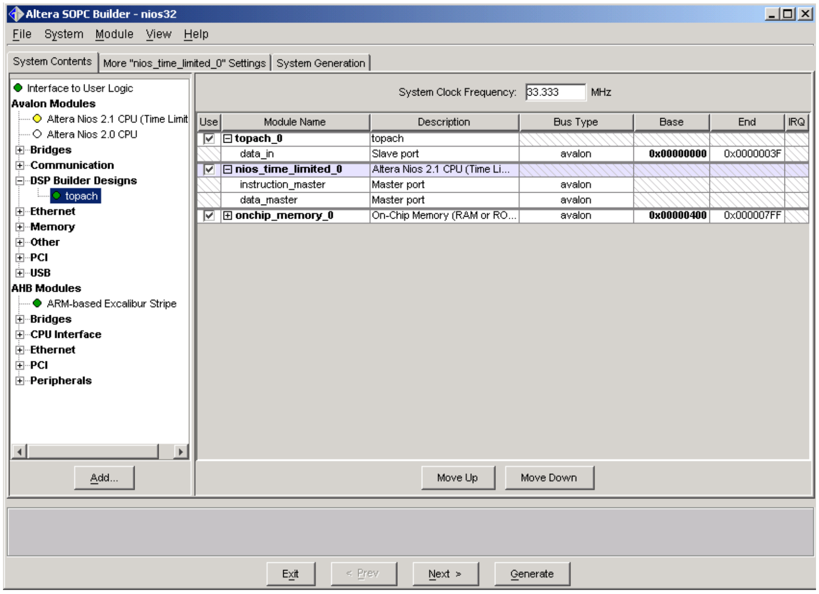
C:\DSPBuilder\DesignExamples\SOPCPort\DSPBuilder_topach

Instantiating your Design as a Custom Peripheral to the Nios Embedded Processor in SOPC Builder

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

1. Choose **Open Project** (File menu).
2. Browse to the **C:\DSPBuilder\DesignExamples\SOPCPort** directory.
3. Select the **topach.quartus** project.
4. Choose **SOPC Builder** (Tools menu).
5. Enter **nios32** as your **System Name**.
6. Select **Verilog** or **VHDL** for your HDL Language.
7. Click **OK**.
8. Click the **System Contents** tab.
9. The **topach** module is listed under **DSP Builder Designs**. Double-click the **topach** module to include it in your Nios system (see Figure 111). The **class.ptf** file, which was generated by SignalCompiler, enabled SOPC Builder to detect the **topach** design and include it as a module in SOPC Builder.
10. Select an appropriate base address for the **topach** module.

Figure 111. Including your DSP Builder Design Module in SOPC Builder



System Clock Frequency: 33.333 MHz

Use	Module Name	Description	Bus Type	Base	End	IRQ
✓	⊟ **topach_0**	topach				
	data_in	Slave port	avalon	**0x00000000**	0x0000003F	
✓	⊟ **nios_time_limited_0**	Altera Nios 2.1 CPU (Time Li...				
	instruction_master	Master port	avalon			
	data_master	Master port	avalon			
✓	⊞ **onchip_memory_0**	On-Chip Memory (RAM or RO...	avalon	**0x00000400**	0x000007FF	

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

For more information, refer to the following sources:

- For information on using SOPC Builder to create a custom Nios embedded processor, refer to the *SOPC Builder Data Sheet*.
- For more information on using the Nios embedded processor, refer to the Nios Processor literature page.

Custom Instruction

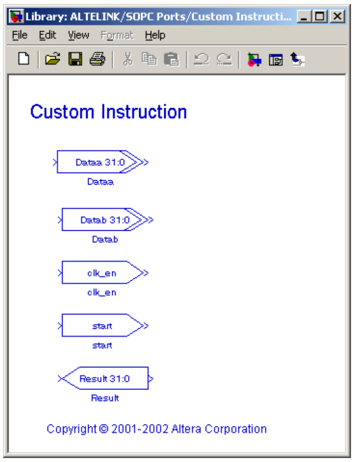
This section contains a description of the Custom Instruction library and a walkthrough.

Custom Instruction Block Description

You use the blocks in the Custom Instruction library to build custom instructions for Nios systems. The Custom Instruction blocks automate the process of specifying ports that are compatible with the SOPC Builder: the blocks correspond to the custom instruction ports used by the SOPC Builder. After you build a model of your custom instruction, add blocks from the Custom Instruction library to control the instruction's inputs and outputs. Figure 112 shows the blocks in the Custom Instruction library.

☞ The DSP Builder does not support the Nios custom instruction `prefix` port. All other ports are supported.

Figure 112. Custom Instruction Library


Library: ALTELINK/SOPC Ports/Custom Instructi...
File Edit View Format Help
Custom Instruction
Dataa 31:0
Dataa
Datab 31:0
Datab
clk_en
clk_en
start
start
Result 31:0
Result
Copyright © 2001-2002 Altera Corporation

☞ For non-combinatorial custom instructions, you must include the start and clk_en blocks as inputs at the top level

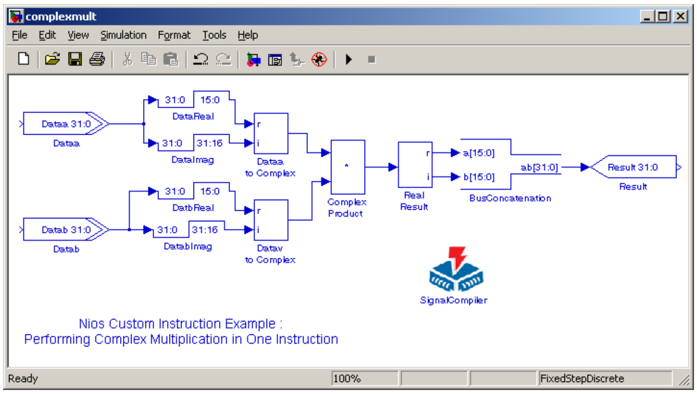
With custom instructions, system designers can add custom-defined functionality to the Nios embedded processor's arithmetic logic unit (ALU) and instruction set. Custom instructions consist of two essential elements:

- *Custom logic block*—Hardware that performs the operation. The Nios embedded processor can include up to five user-defined custom logic blocks. The blocks become part of the Nios microprocessor's ALU.
- *Software macro*—Allows the system designer to access the custom logic through software code.

For more information on Nios custom instructions, refer to *AN 188: Custom Instructions for the Nios Embedded Processor*.

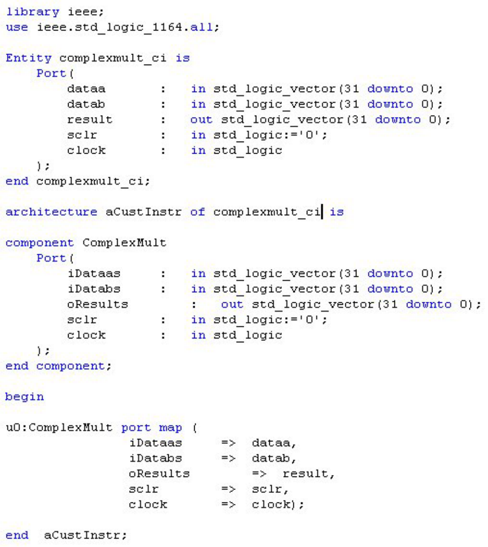
Figure 113 shows an example model using blocks from the Avalon Ports library.

Figure 113. Custom Instruction Example



When you generate VHDL of your model that uses Custom Instruction blocks, the SignalCompiler creates a wrapper file for use with the SOPC Builder. See Figure 114 for an example wrapper file.

Figure 114. Example Wrapper File for Use with SOPC Builder

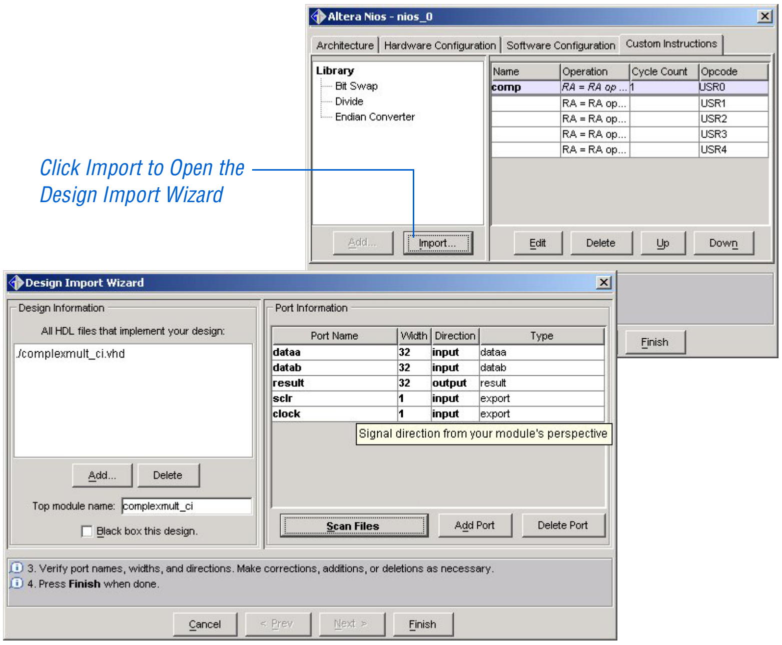


```
library ieee;
use ieee.std_logic_1164.all;

Entity complexmult_ci is
    Port(
        dataa       :   in std_logic_vector(31 downto 0);
        datab       :   in std_logic_vector(31 downto 0);
        result      :   out std_logic_vector(31 downto 0);
        sclr        :   in std_logic:='0';
        clock       :   in std_logic
    );
end complexmult_ci;

architecture aCustInstr of complexmult_ci is

component ComplexMult
    Port(
        iDataas     :   in std_logic_vector(31 downto 0);
        iDatabs     :   in std_logic_vector(31 downto 0);
        oResults        :   out std_logic_vector(31 downto 0);
        sclr        :   in std_logic:='0';
        clock       :   in std_logic
    );
end component;

begin

u0:ComplexMult port map (
                iDataas     =>  dataa,
                iDatabs     =>  datab,
                oResults        =>  result,
                sclr        =>  sclr,
                clock       =>  clock);

end  aCustInstr;
```

You can specify that the Nios embedded processor use this wrapper file as a custom instruction. Refer to *AN 188: Custom Instructions for the Nios Embedded Processor* for instructions on how to specify the file as a custom instruction. Figure 115 shows the Nios CPU **Custom Instruction** tab and the **Design Import Wizard**.

☞ When you import the custom instruction in the SOPC Builder, you must specify the number of clock cycles that your design uses.

Figure 115. Importing a Nios Custom Instruction



Name	Operation	Cycle Count	Opcode
comp	RA = RA op ...	1	USR0
	RA = RA op...		USR1
	RA = RA op...		USR2
	RA = RA op...		USR3
	RA = RA op...		USR4

./complexmult_ci.vhd

Top module name: complexmult_ci

Port Name	Width	Direction	Type
dataa	32	input	dataa
datab	32	input	datab
result	32	output	result
sclr	1	input	export
clock	1	input	export

3. Verify port names, widths, and directions. Make corrections, additions, or deletions as necessary.
4. Press **Finish** when done.

Custom Instruction Walkthrough

This walkthrough describes how to interface your design built using the Custom Instruction library as a custom instruction to the Nios embedded processor in SOPC Builder. In this walkthrough, the **complexmult.mdl** design is included as a custom instruction.

The hardware design consists of a complex multiplier. When this hardware is interfaced to a Nios embedded processor as a custom instruction, the Nios embedded processor can complete a complex multiplication in one instruction. Without the dedicated hardware to perform the computation, a processor could take several instructions to complete a similar computation entirely in software.

This walkthrough involves the following steps:

1. "Open the Walkthrough Example Design" on page 199.

2. "Turn On the SOPC Option in SignalCompiler & Generate VHDL & PTF Files" on page 199.

3. "Including your Design as a Custom Instruction to the Nios Embedded Processor in SOPC Builder" on page 201.

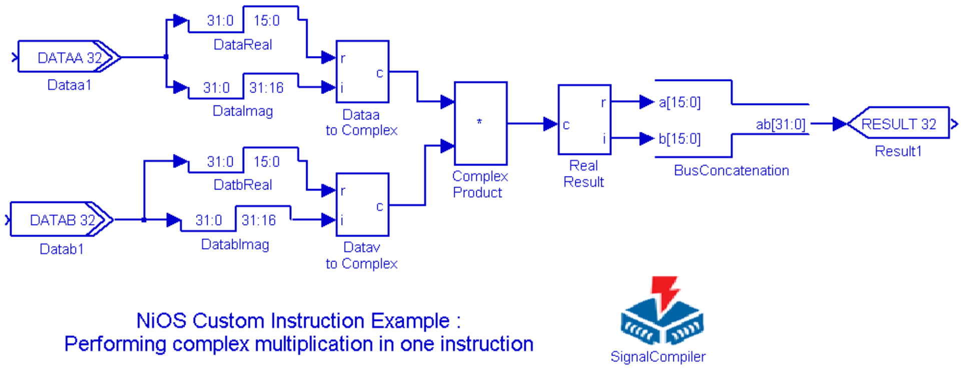
Open the Walkthrough Example Design

To open the example design, perform the following steps:

1. Choose **Open** (File menu) in the MATLAB software.
2. Browse to the **C:\DSPBuilder\DesignExamples\SopcCustomInstruction** directory.
3. Select the **complexmult.mdl** file and click **Open**.

Figure 116 shows the design that you will be using in this walkthrough.

Figure 116. complexmult.mdl Example Design



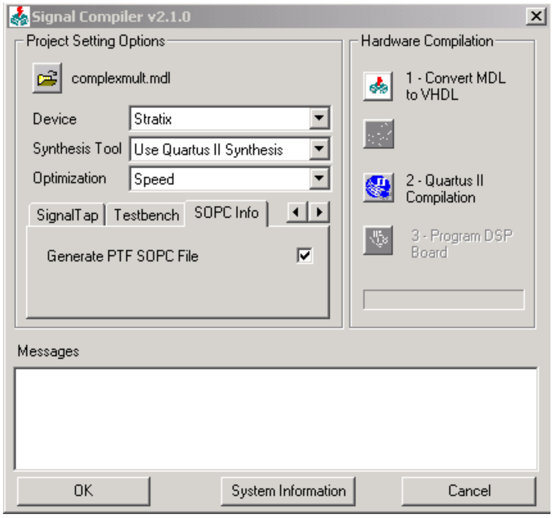
Turn On the SOPC Option in SignalCompiler & Generate VHDL & PTF Files

When you use the Custom Instruction blocks in your design the SignalCompiler can generate a VHDL wrapper file of your Simulink design for use in SOPC Builder. To generate the **class.ptf** file within the SignalCompiler, perform the following steps:

1. Double-click the SignalCompiler block.
2. Click **Analyze**.
3. Click the right arrow button to scroll the tabs to the right.

4. Click the **SOPC Info** tab (see Figure 117).

Figure 117. SOPC Info Tab in the SignalCompiler



Signal Compiler v2.1.0

Project Setting Options

complexmult.mdl

Device: Stratix

Synthesis Tool: Use Quartus II Synthesis

Optimization: Speed

SignalTap Testbench SOPC Info

Generate PTF SOPC File ☑

Hardware Compilation

1 - Convert MDL to VHDL

2 - Quartus II Compilation

3 - Program DSP Board

Messages

OK System Information Cancel

5. Turn on the **Generate PTF SOPC File** option. The SignalCompiler generates a VHDL wrapper file for your design so that you can instantiate it as a custom instruction in SOPC Builder. The wrapper file is named <*design name*>**_ci.vhd**.

6. Choose **Use Quartus II Synthesis** from the **Synthesis Tool** list box.

7. Click **1 – Convert MDL to VHDL**. The SignalCompiler generates a VHDL and a **class.ptf** file for your design.

8. Click **2 – Quartus II Compilation**.

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

The VHDL design file and the **complexmult_ci.vhd** wrapper file for this design are located in the directory:

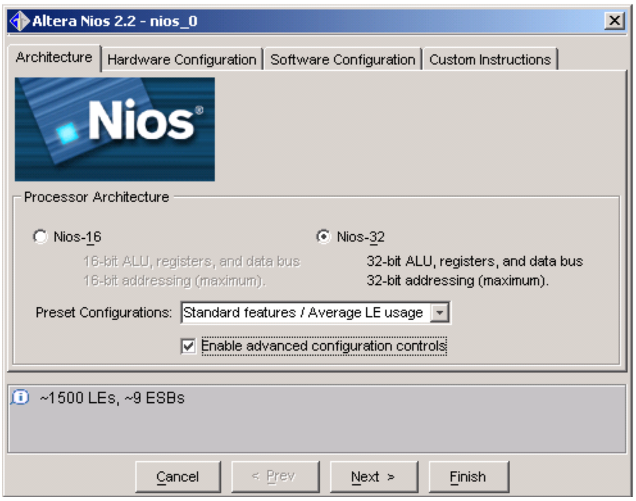
C:\DSPBuilder\DesignExamples\SopcCustomInstruction

Including your Design as a Custom Instruction to the Nios Embedded Processor in SOPC Builder

To include your design as a custom instruction to the Nios embedded processor in SOPC Builder, perform the following steps:

1. Choose **Open Project** (File menu).
2. Browse to the **C:\DSPBuilder\DesignExamples\SopcCustomInstruction** directory.
3. Select the **complexmult.quartus** project.
4. Choose **SOPC Builder** (Tools menu).
5. Enter **nios32** as your **System Name**.
6. Select **Verilog** or **VHDL** for your HDL Language.
7. Click **OK**.
8. Click the **System Contents** tab.
9. Double-click **Altera Nios 2.2 CPU**. The **Altera Nios 2.2 Wizard** dialog box is displayed (see Figure 118).

Figure 118. Altera Nios 2.2 Wizard Dialog Box



Processor Architecture

( ) Nios-16
16-bit ALU, registers, and data bus
16-bit addressing (maximum).

(●) Nios-32
32-bit ALU, registers, and data bus
32-bit addressing (maximum).

Preset Configurations: Standard features / Average LE usage

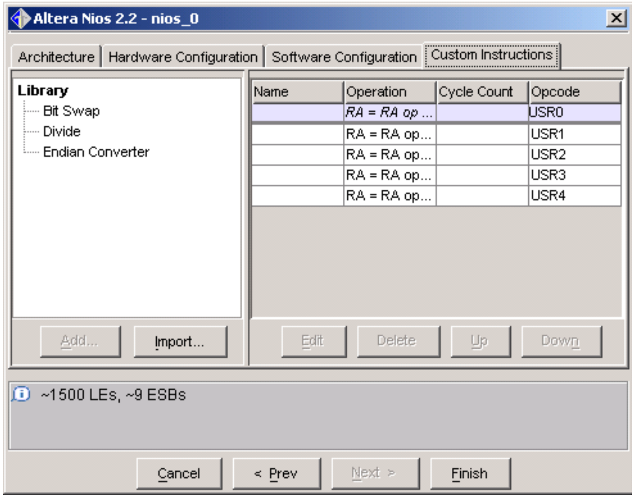
[✓] Enable advanced configuration controls

~1500 LEs, ~9 ESBs

10. Turn on the **Enable advanced configuration controls** option.

11. Click the **Custom Instructions** tab (see Figure 119).

Figure 119. Altera Nios 2.2 Wizard – Custom Instructions Tab



Name	Operation	Cycle Count	Opcode
	RA = RA op ...		USR0
	RA = RA op...		USR1
	RA = RA op...		USR2
	RA = RA op...		USR3
	RA = RA op...		USR4

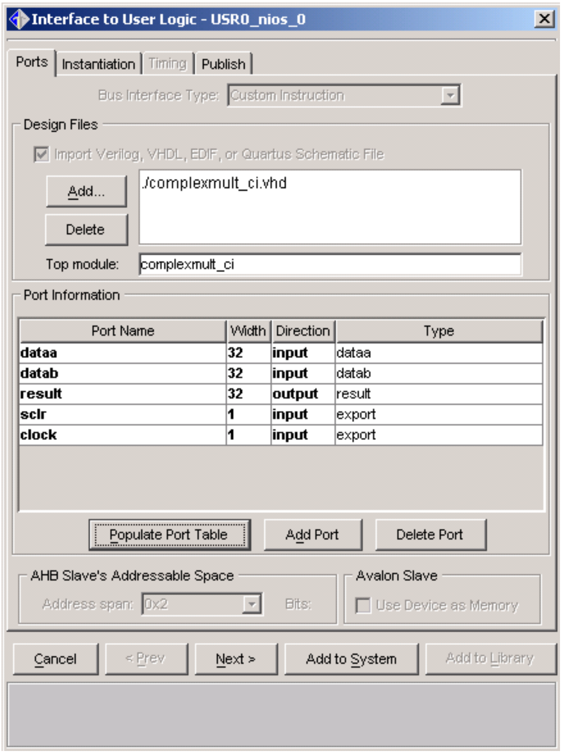
~1500 LEs, ~9 ESBs

12. Click the **USR0** under **Opcode** to select the row.

13. Click **Import**.

14. The **Interface to User Logic Dialog Box** is displayed (see Figure 120).

Figure 120. Interface to User Logic Window



Bus Interface Type: Custom Instruction

./complexmult_ci.vhd

Top module: complexmult_ci

Port Name	Width	Direction	Type
dataa	32	input	dataa
datab	32	input	datab
result	32	output	result
sclr	1	input	export
clock	1	input	export

Address span: 0x2

15. Click **Add**.

16. Browse to the **C:\DSPBuilder\DesignExamples\SopcCustomInstruction** directory and select the **complexmult_ci.vhd** file.

17. Click **Populate Port Table**.

18. Click **Add to System**. Your DSP Builder design is added as a custom instruction named **comp**.

19. Type a 1 in the **Cycle Count** column for **comp**. The SOPC Builder **cycle count** column sets the design latency, which is the number of clock cycles required to compute a valid output.

20. Click **Finish**.

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

For more information, refer to the following documentation:

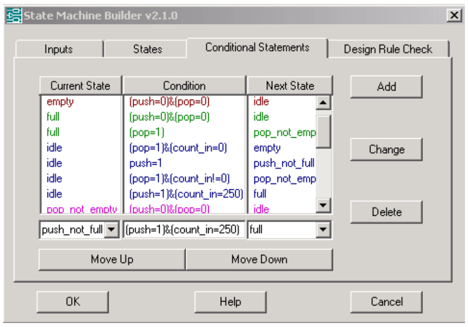
- For implementing Nios Custom Instructions in SOPC Builder, refer to *AN188: Custom Instructions for the Nios Embedded Processor*.
- For using SOPC Builder to create a custom Nios embedded processor, refer to the SOPC Builder Data Sheet.
- For using the Nios embedded processor, refer to the Nios Processor literature page.

State Machine Functions Library

State Machine Table Block

The State Machine Table block allows you use a state transition table to build a one-hot Moore state machine where the output is equal to the current state (see Figure 121).

Figure 121. State Transition Table for a FIFO Controller

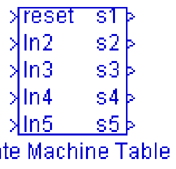


State Machine Builder v2.1.0

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

The default State Machine Table symbol is shown in Figure 122. The default state machine has five inputs and five states. Each state is represented by an output. While the state machine is operating, an output is assigned a logic level 1 if its respective state is equal to the current state. All other outputs get assigned a logic level 0. In Simulink, the input and output are represented as signed integers. In VHDL, the input and output are represented as standard logic vectors.

Figure 122. Default State Machine Table Block



The following sections describe the design flow used to implement a state machine in DSP Builder. The sections use an example design, **fifo_control_logic.mdl**, which contains a simple state machine used to implement the control logic for a FIFO. The state machine feeds the control inputs of a dual port RAM block and the inputs of an address counter. The operation of the state machine is as follows:

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

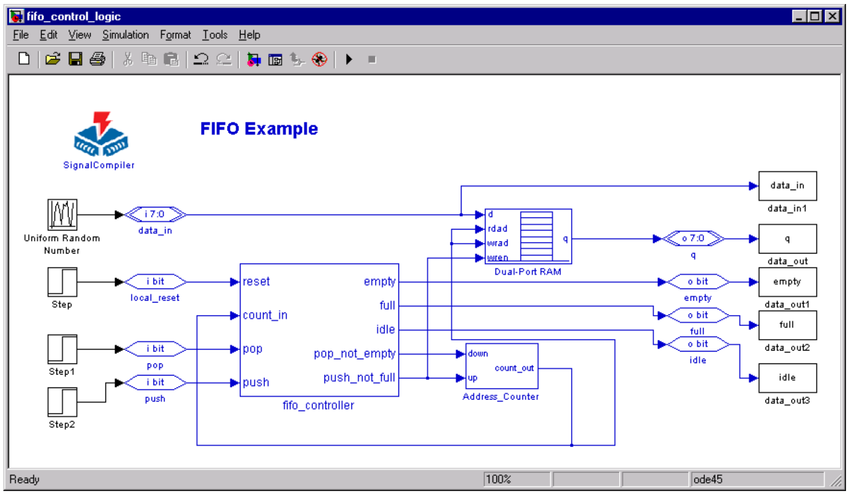
Table 49 shows the state transition table for the FIFO controller.

Table 49. Idle State Condition Priority

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)	push_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

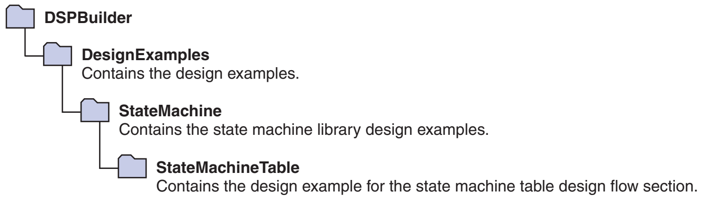
Figure 123 shows the top-level schematic for the FIFO design example.

Figure 123. FIFO Design Example Top-Level Schematic



When you install the DSP Builder software, the design files are installed in the directory structure shown in Figure 124.

Figure 124. Directory Structure for the FIFO Design Example



DSPBuilder

DesignExamples
Contains the design examples.

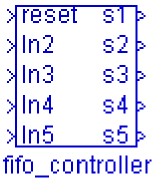
StateMachine
Contains the state machine library design examples.

StateMachineTable
Contains the design example for the state machine table design flow section.

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

1. Add a State Machine Table block to your Simulink design and assign it a name. Figure 125 shows the State Machine Table block. In this example, the block is **fifo_controller**.

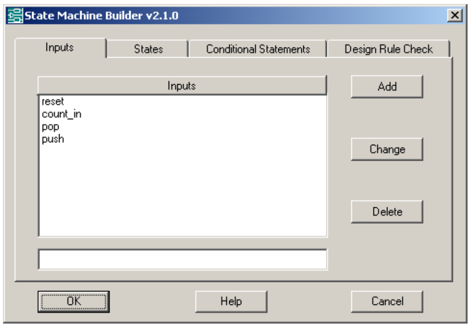
Figure 125. fifo_controller State Machine Block



You must change the default name of the State Machine Table block, before defining the state machine properties.

2. Double-click the State Machine Table block to define the state machine properties. The State Machine Builder GUI 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, change, and delete input names. Figure 126 shows the **Inputs** tab after the inputs have been defined for the FIFO design example.

Figure 126. State Machine Builder Inputs Tab



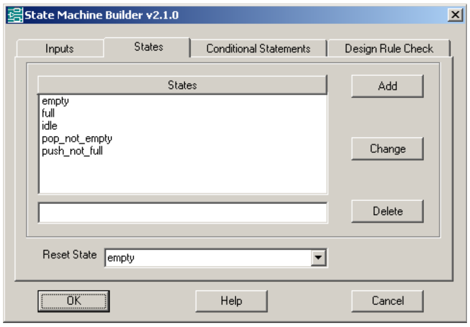
reset
count_in
pop
push

3. 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.

   Figure 127 shows the State Machine Builder **States** tab. The states have been edited for the FIFO design example.

Figure 127. State Machine Builder States Tab



States:
empty
full
idle
pop_not_empty
push_not_full

Reset State: empty

4. After specifying the input and state names, click the **Conditional Statements** tab and describe the behavior of your state machine (see Figure 128 on page 210).

   The **Conditional Statements** tab displays the state transition table, which contains the conditional statements that define your state machine. 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. Table 50 shows the priority of the conditional operators used to define a conditional expression.

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

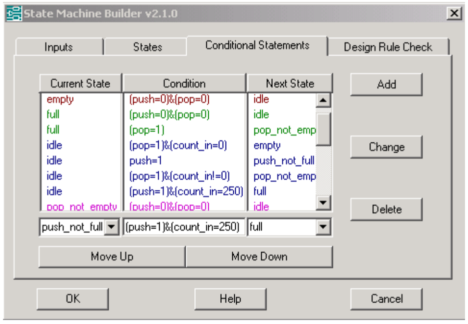
   ☞ To indicate in a conditional statement that a state machine always transitions from the current state to the next state, specify the conditional expression to be 1.

Table 50. Supported Operators in Conditional Expressions

Comparison 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)

Figure 128 shows the **Conditional Statements** tab, after defining the conditional statements for the FIFO controller.

Figure 128. State Machine Builder Conditional Statements Tab



Current State	Condition	Next State
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
pop_not_empty	(push=0)&(pop=0)	idle
push_not_full	(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. The conditions are evaluated in the order that you list them in the conditional statements table. For example, Table 51 shows the conditional statements when the current state is idle. Because the condition (pop=1) & (count_in=0) is higher in the table than the condition (push=1) & (count_in=250), it has higher priority. The condition (pop=1) & (count_in!=0) has the next highest priority and (push=1) & (count_in=250) has the lowest priority.

Table 51. 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)	push_not_empty
idle	(push=1)&(count_in=250)	full

Figure 129 shows the VHDL code that is generated from the states in Table 51. The extension `_sig` is added to the input names in the VHDL file.

Figure 129. Idle State VHDL Code

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

You can use the **Move Up** and **Move Down** buttons to change the order of the conditional statements. For example, if you move the condition push=1 down, as shown in Table 52, the State Machine Builder generates the VHDL code shown in Figure 130.

Table 52. Idle State Condition Priority (Reordered)

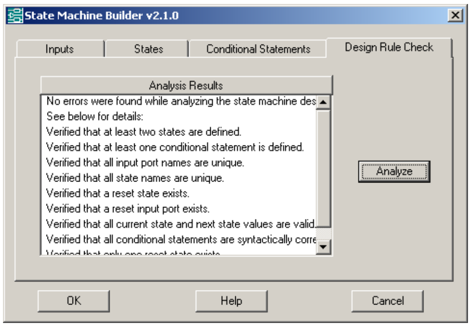
Current State	Condition	Next State
idle	(pop=1)&(count_in=0)	empty
idle	(push=1)&(count_in=250)	full
idle	(pop=1)&(count_in!=0)	push_not_empty
idle	push=1	push_not_full

Figure 130. Idle State VHDL Code (Reordered)

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

5. Click the **Design Rule Check** tab and 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. Figure 131 shows the **Design Rule Check** tab after clicking **Analyze**. 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 131. 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
Verified that only one reset state exists
```

6. To save the changes made to your state machine, click **OK**. State Machine Builder 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. Figure 132 shows the updated State Machine Table block for the FIFO design example.

Figure 132. State Machine Table Block after Specifying the State Machine

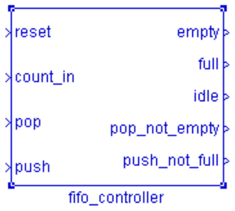
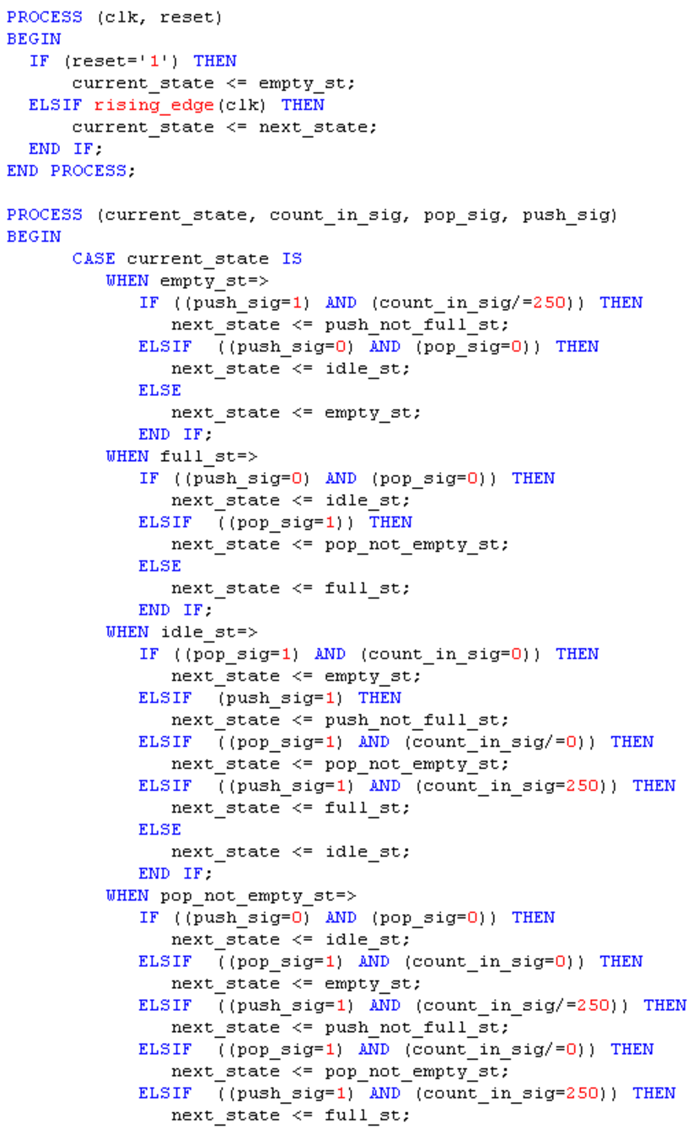


Figure 133 shows the VHDL code generated by the State Machine builder.

Figure 133. VHDL Code Generated by the State Machine Builder



```
PROCESS (clk, reset)
BEGIN
  IF (reset='1') THEN
      current_state <= empty_st;
  ELSIF rising_edge(clk) THEN
      current_state <= next_state;
  END IF;
END PROCESS;

PROCESS (current_state, count_in_sig, pop_sig, push_sig)
BEGIN
      CASE current_state IS
         WHEN empty_st=>
            IF ((push_sig=1) AND (count_in_sig/=250)) THEN
               next_state <= push_not_full_st;
            ELSIF  ((push_sig=0) AND (pop_sig=0)) THEN
               next_state <= idle_st;
            ELSE
               next_state <= empty_st;
            END IF;
         WHEN full_st=>
            IF ((push_sig=0) AND (pop_sig=0)) THEN
               next_state <= idle_st;
            ELSIF  ((pop_sig=1)) THEN
               next_state <= pop_not_empty_st;
            ELSE
               next_state <= full_st;
            END IF;
         WHEN idle_st=>
            IF ((pop_sig=1) AND (count_in_sig=0)) THEN
               next_state <= empty_st;
            ELSIF  (push_sig=1) THEN
               next_state <= push_not_full_st;
            ELSIF  ((pop_sig=1) AND (count_in_sig/=0)) THEN
               next_state <= pop_not_empty_st;
            ELSIF  ((push_sig=1) AND (count_in_sig=250)) THEN
               next_state <= full_st;
            ELSE
               next_state <= idle_st;
            END IF;
         WHEN pop_not_empty_st=>
            IF ((push_sig=0) AND (pop_sig=0)) THEN
               next_state <= idle_st;
            ELSIF  ((pop_sig=1) AND (count_in_sig=0)) THEN
               next_state <= empty_st;
            ELSIF  ((push_sig=1) AND (count_in_sig/=250)) THEN
               next_state <= push_not_full_st;
            ELSIF  ((pop_sig=1) AND (count_in_sig/=0)) THEN
               next_state <= pop_not_empty_st;
            ELSIF  ((push_sig=1) AND (count_in_sig=250)) THEN
               next_state <= full_st;
```

7. Connect the State Machine Table block to the rest of your design.

See "Block I/O Formats" on page 78 for information on the block I/O formats.

Delay Block

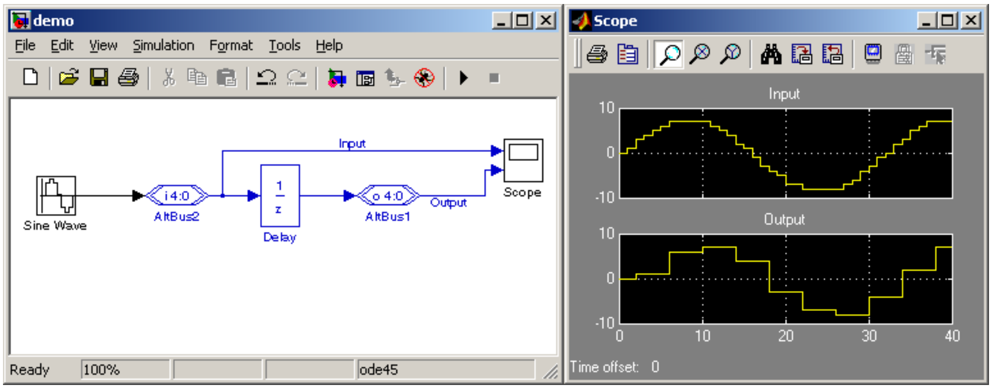
The Delay block delays the incoming data by the amount specified by the `Depth` parameter. The block accepts any data type as inputs. Table 53 shows the block parameters.

Table 53. Delay Parameters

Name	Value	Description
Depth	User Defined	Indicate the delay length of the block.
Use Control Inputs	On or Off	Indicate that you wish to use the additional control inputs, clock enable, and resets.
Clock Phase Selection	User Defined	Phase selection. Indicate the phase selection with a binary string, where a 1 indicates the phase in which the block is enabled. For example: `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 second phase out of 4 and only the second 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 134 shows an example using the Delay block.

Figure 134. Delay Example



See “Block I/O Formats” on page 78 for information on the block I/O formats.

Down Sampling Block

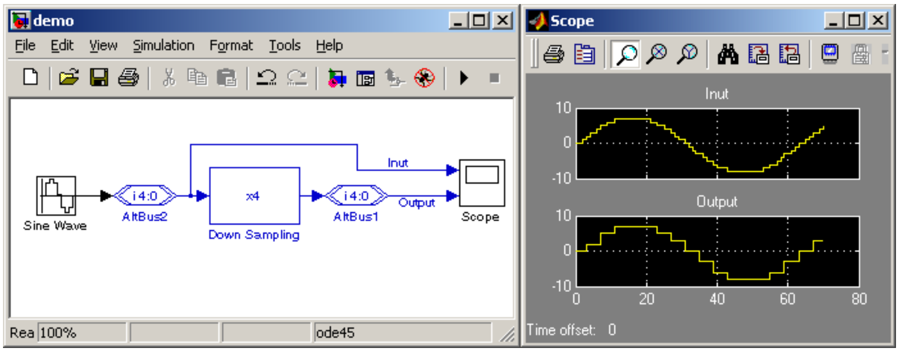
The Down Sampling block decreases the output sample rate from the input sample rate. The output data is sampled at every *m*th cycle where *m* is equal to the down sampling rate. The input sample rate is normalized to 1 in Simulink (see "Frequency Design Rule" on page 64 for a description of how to normalize the frequency). Table 54 shows the block parameters.

Table 54. Down Sampling Parameters

Name	Value	Description
Down Sampling Rate	1 - 20	Indicate the down sampling rate.

Figure 135 shows an example using the Down Sampling block.

Figure 135. Down Sampling Example



See "Block I/O Formats" on page 78 for information on the block I/O formats.

Dual-Port RAM Block

When you use the Dual-Port RAM block, SignalCompiler maps data to the embedded RAM (EABs or ESBs) in Altera devices; the contents of the RAM are pre-initialized to zero. The Dual-Port RAM block accepts any data type as input. All input ports are registered; all output ports are unregistered. Table 55 shows the block parameters.

Table 55. Dual-Port RAM Parameters

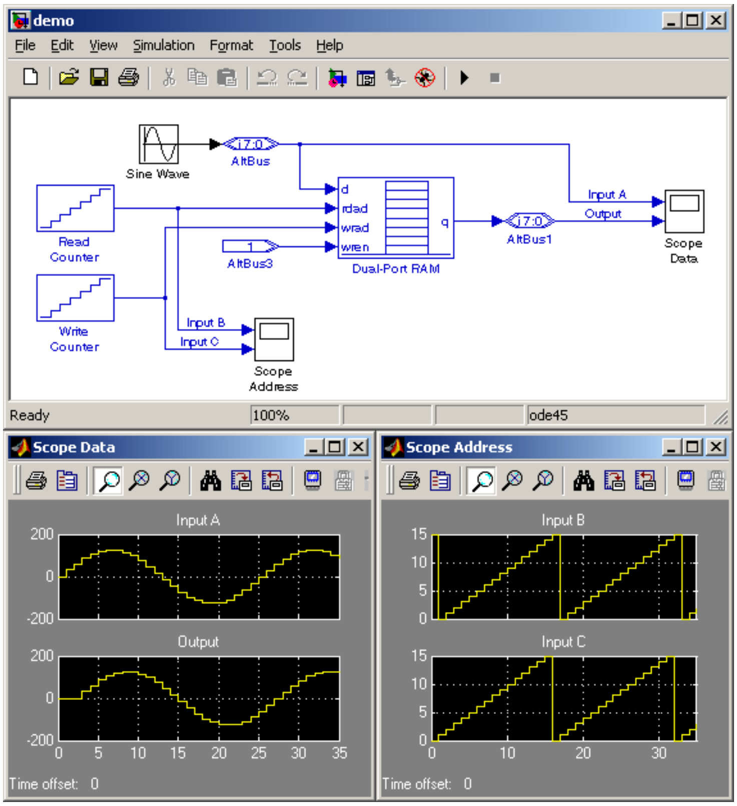
Name	Value	Description
Address Width	2 - 20	Indicate the address width.
Clock Phase Selection	User Defined	Phase selection. Indicate the phase selection with a binary string, where a 1 indicates the phase in which the block is enabled. For example: `1`—The block is always enabled and captures all data passing through the block (sampled at the rate 1). `10`—The block is enabled every other phase and every other data (sampled at the rate 1) passes through. `0100`—The block is enabled on the second phase out of 4 and only the second 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.

The Dual-Port RAM block has the following ports:

- `d`—Input data
- `q`—Output data
- `rdad`—Read address bus
- `wrad`—Write address bus
- `wren`—Write enable

Figure 136 shows an example using the Dual-Port RAM block.

Figure 136. Dual-Port RAM Example



See “Block I/O Formats” on page 78 for information on the block I/O formats.

Parallel to Serial Block

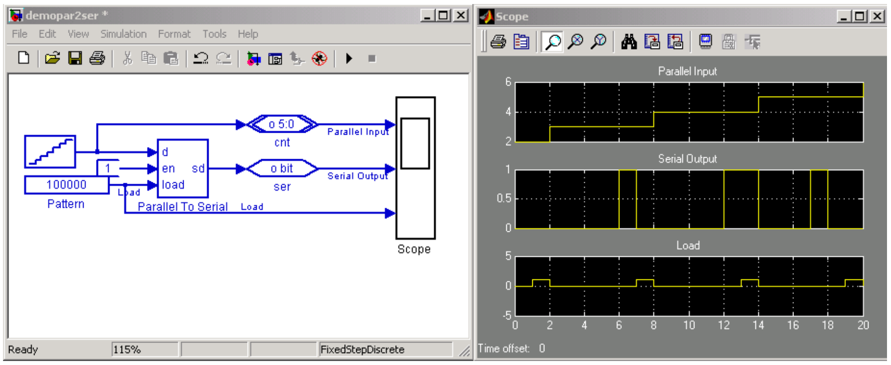
The Parallel to Serial block implements a parallel (input `d`) to serial bus conversion (output `sd`). Table 56 shows the block parameters.

Table 56. Parallel to Serial Parameters

Name	Value	Description
Data Bus Type	Signed Integer, Signed Fractional, Unsigned Integer	Choose the bus type format.
[number of bits].[]	1 - 51	Indicate the number of bits stored on the left side of the binary point including the sign bit.
[].[number of bits]	0-51	Indicate the number of bits stored on the right side of the binary point including the sign bit. This option only applies to signed fractional formats.
Serial Bit Order	MSB First, LSB First	Indicate whether the most significant bit (MSB) or least significant bit (LSB) should be transmitted first.

Figure 137 shows an example using the Parallel to Serial block.

Figure 137. Parallel to Serial Example



See "Block I/O Formats" on page 78 for information on the block I/O formats.

Pattern Block

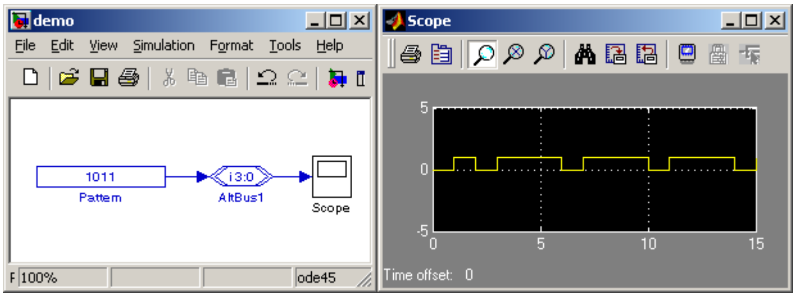
The Pattern block generates a repeating periodic bit sequence in time. For example, `01100` yields `01100011000110001100011000110001100011000` in time. You can change the output data rate for a registered block by feeding the clock enable input with the output of the Pattern block. Table 57 shows the block parameters.

Table 57. Pattern Parameters

Name	Value	Description
Binary Sequence	User Defined	Indicate the sequence that you wish to use.
Use Control Inputs	On or Off	Indicate that you wish to use additional control inputs (clock enable and reset).

Figure 138 shows an example using the Pattern block.

Figure 138. Pattern Example



See "Block I/O Formats" on page 78 for information on the block I/O formats.

ROM EAB Block

The ROM EAB block is used to read out pre-loaded data. The data input must be specified as a hexadecimal file. To use the embedded memory in an Altera device as ROM, use the ROM EAB block to read in an Intel-format Hexadecimal File (**.hex**) containing the ROM data.

You can use the Quartus II software to generate a Hex File. Search for "Creating a Memory Initialization File or Hexadecimal (Intel-Format) File" in Quartus II Help for instructions on creating this file.

☞ If you use the Quartus II software to generate a Hex File, you must save the Hex File in your DSP Builder working directory.

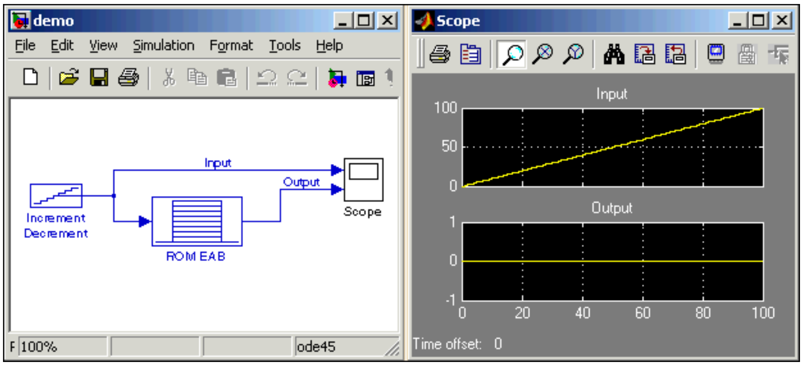
Table 58 shows the block parameters.

Table 58. ROM EAB Parameters

Name	Value	Description
Data Bus Type	Signed Integer, Signed Fractional, Unsigned Integer	Choose the bus type format.
[number of bits].[]	1 - 51	Indicate the number of bits stored on the left side of the binary point including the sign bit.
[].[number of bits]	1-51	Indicate the number of bits stored on the right side of the binary point including the sign bit. This option only applies to signed fractional formats.
Address Width	2 - 20	Indicate the address width.
Clock Phase Selection	User Defined	Phase selection. Indicate the phase selection with a binary string, where a 1 indicates the phase in which the block is enabled. For example: `1`—The block is always enabled and captures all data passing through the block (sampled at the rate 1). `10`—The block is enabled every other phase and every other data (sampled at the rate 1) passes through. `0100`—The block is enabled on the second phase out of 4 and only the second 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.
Input Hex File	User Defined; <*filename*>**.hex**	Indicate the name of the HEX File to use.

Figure 139 shows an example using the ROM EAB block.

Figure 139. ROM EAB Example



See “Block I/O Formats” on page 78 for information on the block I/O formats.

Serial to Parallel Block

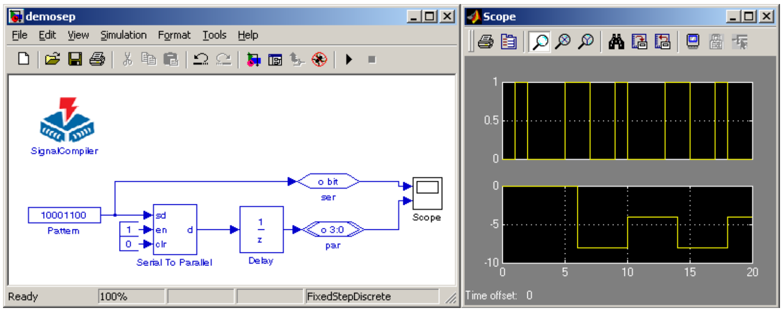
The Serial to Parallel block implements a serial (input `sd`) to parallel bus conversion (output `d`). Table 59 shows the block parameters.

Table 59. Serial to Parallel Parameters

Name	Value	Description
Data Bus Type	Signed Integer, Signed Fractional, Unsigned Integer	Choose the bus type format.
[number of bits].[]	1 - 51	Indicate the number of bits stored on the left side of the binary point including the sign bit.
[].[number of bits]	0-51	Indicate the number of bits stored on the right side of the binary point including the sign bit. This option only applies to signed fractional formats.
Serial Bit Order	MSB First, LSB First	Indicate whether the most significant bit (MSB) or least significant bit (LSB) should be transmitted first.

Figure 140 shows an example using the Serial to Parallel block.

Figure 140. Serial to Parallel Example



See "Block I/O Formats" on page 78 for information on the block I/O formats.

Shift Taps Block

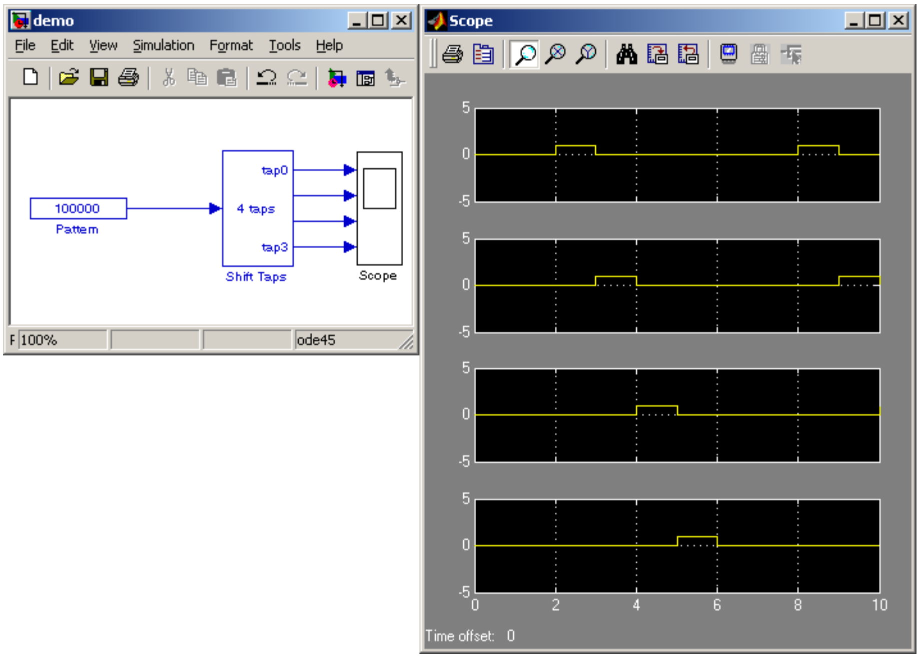
The Shift Taps block implements a shift register that you can use for filters or convolution. In the Altera device, the block implements a RAM-based shift register that is useful for creating very large shift registers efficiently. The block outputs occur at regularly spaced points along the shift register (i.e., taps). In Stratix devices, this block is implemented in the small memory. Table 60 shows the block parameters.

Table 60. Shift Taps Parameters

Name	Value	Description
Number of Taps	User Defined	Specifies the number of regularly spaced taps along the shift register
Distance Between Taps	User Defined	Specifies the distance between the regularly spaced taps in clock cycles. This number translates to the number of RAM words that will be used.
Use Shift Out	On or Off	Indicate whether you want to create an output from the end of the shift register for cascading.
Use Clock Enable	On or Off	Indicate whether you would like to use an additional control input (clock enable).
Use Dedicated Hardware	On or Off	If you are targeting Stratix devices, turn on this option to implement the functionality in Stratix small memory.

Figure 141 shows an example using the Shift Taps block.

Figure 141. Shift Taps Example



See “Block I/O Formats” on page 78 for information on the block I/O formats.

Up Sampling Block

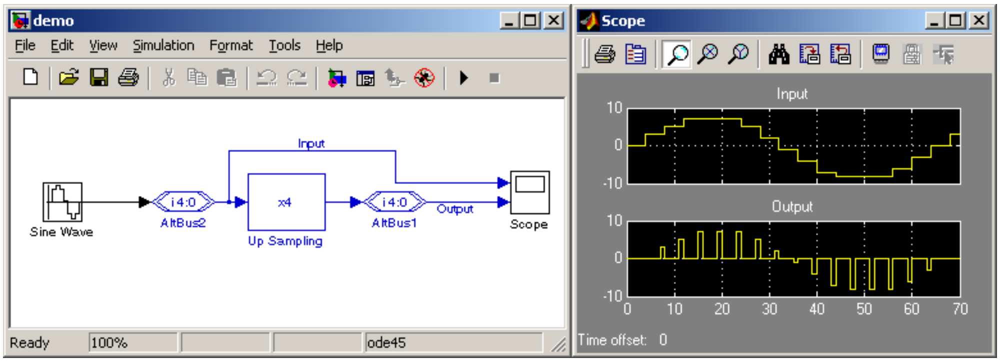
The Up Sampling block increases the output sample rate from the input sample rate. The output data is sampled every *l* cycles where *l* is equal to the up sampling rate. Table 61 shows the block parameters.

Table 61. Up Sampling Parameters

Name	Value	Description
Up Sampling Rate	1 - 20	Indicate the up sampling rate.

Figure 142 shows an example using the Up Sampling block.

Figure 142. Up Sampling Example



See "Block I/O Formats" on page 78 for information on the block I/O formats.

Notes:

Troubleshooting

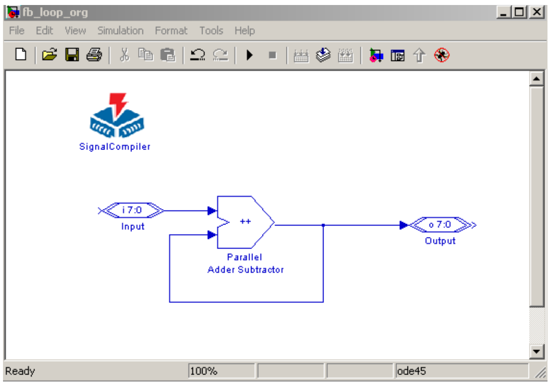
This section contains the following troubleshooting information:

- "Fixing "Unresolved Width" Error When Converting a Model File to VHDL" on page 229
- "Bit Widths in VHDL Are Different than the Expected Bit Width Based on the Simulink Model File" on page 231
- "Simulink Simulation Results for the Pattern & Increment/Decrement Blocks Does Not Match the Results from ModelSim" on page 231
- "Passing Pin Constraints to the AltBus Block" on page 232
- "ModelSim Error for Missing "lpm_components" when Running DSP Builder Tcl Script" on page 232
- "Integrating VHDL Components into Simulink Model Files (Black Boxing)" on page 232
- "Read Previous DSP Builder Designs with DSP Builder v2.1.0" on page 233
- "The Altera DSP Builder Folder Does Not Appear in the Simulink Library Browser" on page 233
- "Automated Flow Does Not Run Properly" on page 234
- "APEX DSP Development Board Troubleshooting" on page 236
- "Error Message: SignalCompiler Is Unable to Check a Valid License" on page 236
- "ModelSim Fails with the SignalCompiler-Generated Tcl Scripts" on page 239
- "The Simulink Library Browser Does Not Show Altera MegaCore Blocks" on page 240
- "Specifying LeonardoSpectrum, Synplify & Quartus II Path Information for SignalCompiler" on page 240
- "The Wizard for an IP MegaCore Block Does Not Launch" on page 241

Fixing "Unresolved Width" Error When Converting a Model File to VHDL

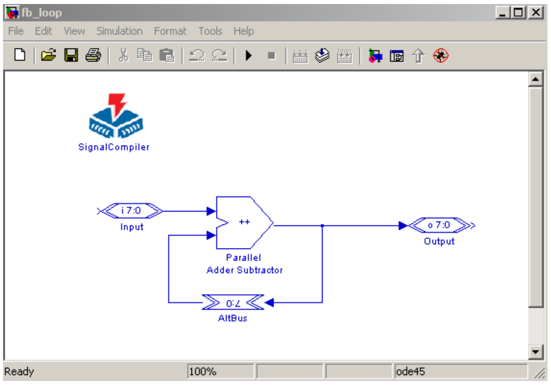
You may get an unresolved width error if you have a feedback loop in your design and the feedback loop's bit width is not defined explicitly. For example, Figure 143 exhibits this error.

Figure 143. 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 144.

Figure 144. Feedback Loop with AltBus Block as an Internal Node



Bit Widths in VHDL Are Different than the Expected Bit Width Based on the Simulink Model File

DSP Builder propagates bit widths by assuming that bit width information from a source block passes to the destination block. Sometimes, this propagation involves automatic sign extension, which applies to arithmetic operations (adders, multipliers, etc.) and logical operators (multiplexers, etc.). Propagation works across data types, including signed integer, signed binary fractional (SBF), and unsigned integer.

DSP Builder sign extends unsigned numbers with an additional MSB stuck to zero. For SBF, the MSB is signed extended, while the LSB is zero-extended. For example, if you multiply the SBF operands:

A[4].[4] x B[2].[2]

DSP Builder automatically sign extends B to B[4].[4] such that

B(0) = B(1) = 0
B(7) = B(6) = B(5)

The Signal Compiler Report File provides complete bit width information.

Simulink Simulation Results for the Pattern & Increment/Decrement Blocks Does Not Match the Results from ModelSim

You may experience this problem if you are using blocks that do not have the sample time explicitly specified. DSP Builder inherits the sample time from source blocks. However, some blocks, such as the Pattern and Increment/Decrement blocks, may not have source blocks. You must explicitly specify the sample time for these blocks using one of the following options:

- Turn on the **Use Control Inputs** option to add the Tsamp block to the control inputs.
- Specify **Fixed Step Size** as `1` instead of `auto` in the **Simulation Parameters** window (choose **Simulation Parameters** from the Simulation menu).

Passing Pin Constraints to the AltBus Block

To pass pin constraints to the ports in your Simulink Model File, associate the design with a specific board library. SignalCompiler must detect a board configuration block in the Model File before it can pass pin constraints to the Tcl script for the Quartus II project.

ModelSim Error for Missing "lpm_components" when Running DSP Builder Tcl Script

You may receive the error:

```
(vcom-11) Could not find lpm.lpm_components.
```

because of the ModelSim settings on your machine. There are two ways to work around this problem:

- Delete the **lpm** and **work** libraries in your project directory, and rerun the Tcl script, **tb_**<*filename*>**.tcl**.
- Manually compile the **220pack.vhd** and **220model.vhd** libraries found in the <*path to DSP Builder*>**\Altlib** directory into the **lpm** library.

Integrating VHDL Components into Simulink Model Files (Black Boxing)

You can use the SubSystem Builder block with the **Create Black Box SubSystem** option. This setting uses the AltBus block in **Black Box Input Output** mode to specify the port description.

If you have an existing VHDL design with a clock signal, you have to create a wrapper file with an input port named `simulink_clock`. This port automatically maps to the Simulink system clock. The symbol created by SubSystem Builder still has a `isimulink_clock` port, which must be connected. However, the port is disregarded in the actual design during the VHDL conversion.

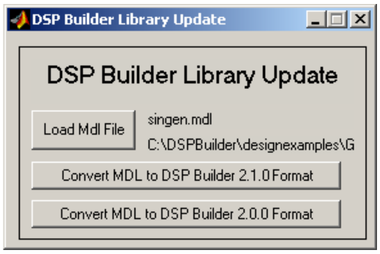
For details, refer to "Black Boxing" on page 86.

Read Previous DSP Builder Designs with DSP Builder v2.1.0

DSP Builder version 2.1.0 has a new library format. You must use the **libupd** utility to convert the library format of a Model File made using blocks from DSP Builder version 2.0.0 or earlier before opening the file in version 2.1.0. To use the **libupd** utility, perform the following steps:

1. At the MATLAB prompt type `libupd` ↵. The **DSP Builder Library Update** dialog box appears (see Figure 145).

Figure 145. Library Update Window



DSP Builder Library Update

DSP Builder Library Update

Load Mdl File
singen.mdl
C:\DSPBuilder\designexamples\G
Convert MDL to DSP Builder 2.1.0 Format
Convert MDL to DSP Builder 2.0.0 Format

2. Click **Load Mdl File**

3. Select the Model File to convert.

4. Click **Convert MDL to DSP Builder to 2.1.0 Format**.

The Altera DSP Builder Folder Does Not Appear in the Simulink Library Browser

If, after installation, the Altera DSP Builder folder does not appear in the Simulink **Library Browser**, perform the following steps in MATLAB:

1. Close Simulink.

2. At the MATLAB prompt in the MATLAB **Command Window**, change directory to <*installation path*>**\DSPBuilder\Altlib**. The default is **c:\DSPBuilder\Altlib**.

3. Type the command `Setup_DSPBuilder` ↵.

4. Open the Simulink **Library Browser**; the **Altera DSP Builder** folder should appear.

Automated Flow Does Not Run Properly

The DSP Builder automated flow allows you to control your entire synthesis and compilation flow from within the MATLAB/Simulink environment using the SignalCompiler block. With the automated flow, the SignalCompiler block outputs VHDL files and Tcl scripts and then automatically begins synthesis in the LeonardoSpectrum, Synplify, or Quartus II software and compilation in the Quartus II software.

If the LeonardoSpectrum, Synplify, or Quartus II software does not run automatically, perform the following steps:

1. Check software paths
2. Change the system path settings

Check the Software Paths (Registry Settings)

If you have multiple versions of the same software product on your PC (e.g., Quartus II Limited Edition and a full version of the Quartus II software), your registry settings may point to the wrong version.

DSP Builder obtains the paths to the software from the registry settings. To check which paths DSP Builder is using, type the following commands in the MATLAB **Command Window**:

```
cd <DSP Builder install path>\DSPBuilder\Altlib ↵
dos('vhdlconv'); ↵
```

This command outputs the status of SignalCompiler. See Figure 146.

Figure 146. SignalCompiler Status Output

```
DSP Builder
Quartus II development tool and MATLAB/Simulink Interface
Version 1.0.0$Revision:  14.24 $
Copyright © 2001 Altera Corporation. All rights reserved.
EDA Information
    Quartus II       : C:\quartus\bin\quartus_cmd -f
    Synplicity    : C:\synplicity\synplify\bin\synplify_pro -batch
    LeonardoSpectrum : C:\Exemplar\LeoSpec\v2001_1a_28_OEM_Altera\bin\win32\spectrum
       -product ls1 -nosession_file -file
License Information
  DSP Builder license is valid 16dok(1)
```

Change the System Path Settings

If the paths to the software are incorrect, fix them by performing the following steps:

1. Choose **Run** (Windows Start Menu).
2. Type `regedit` in the **Open** box and click **OK**.
3. Expand **HKEY_LOCAL_MACHINE** by clicking the plus symbol next to it.
4. Expand **SOFTWARE**.
5. Expand **Altera Corporation**.
6. Click **Quartus**.
7. Verify that the **Quartus Install Directory** is set to the location in which you installed the version of Quartus II software you want to use (e.g., **c:\quartus**). If it is not set to the correct installation directory:
   a. Double-click the text **Quartus Install Directory**.
   b. Type the correct path into the **Value data** box.
   c. Click **OK**.
8. Check the registry settings for the LeonardoSpectrum or Synplify software.
   a. The registry key that contains path information for LeonardoSpectrum is Exemplar. For the Leonardo-Spectrum-Altera software it is located in the **Exemplar Logic > LeonardoSpectrum >** *<version>***_OEM_Altera** folder.
   b. The registry key that contains path information for Synplify is Synplify and Synplify Pro.

      The keys are located in the **Synplicity > currentversion** folder.
9. Close the **Registry Editor** when you are done making changes.

APEX DSP Development Board Troubleshooting

If the SignalCompiler does not appear to have configured the device on the DSP development board, refer to the following troubleshooting information:

- Ensure that the board is set up and connected to your PC and you have installed any necessary drivers. Refer to 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 download cable, and the Quartus II Programmer in JTAG mode. SignalCompiler generates the **.sof** in your working directory. Refer to any of the lab white papers included with the Stratix or APEX DSP development kit for instructions on using a **.sof** to configure the board.

Error Message: SignalCompiler Is Unable to Check a Valid License

You will 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. Refer to "Set Up Licensing" on page 22 for information on how to obtain a license. Additionally, refer to "Verifying that Your DSP Builder Licensing Functions Properly" on page 236.

Verifying that Your DSP Builder Licensing Functions Properly

Type the following command in the MATLAB **Command Window**:

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

This command outputs the status of the DSP Builder license. See Figure 147.

Figure 147. Example DSP Builder License Status Output

```
lmutil - Copyright (C) 1989-1999 Globetrotter Software, Inc.
FLEXlm diagnostics on Wed 10/24/2001 14:36
----------------------------------------------------
License file: c:\qdesigns\license.dat
----------------------------------------------------
"C4D5_512A" v0000.00, vendor: alterad
uncounted nodelocked license, locked to Vendor-defined "GUARD_ID=T000001297" no expiration date
```

☞ You will receive the message about the `hostid` if you are using an Altera software guard for licensing.

If the command does not work as described above, your license file may not be set up correctly. Refer to "Automated Flow Does Not Run Properly" on page 234 for information on how to check your system path and registry settings. Additionally, refer to "Verifying that the LM_LICENSE_FILE Variable Is Set Correctly" on page 238.

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. See Figure 148.

Figure 148. Example DSP Builder License Status Output (Server)

```
lmutil - Copyright (C) 1989-1999 Globetrotter Software, Inc.
Flexible License Manager status on Fri 8/3/2001 15:36
License server status: 1501@shama,1501@mogul,1501@newton
License file(s) on shama: /usr/licenses/quartus/license.dat:
    shama: license server UP (MASTER) v7.0
    mogul: license server UP v7.0
    newton: license server UP v7.0
        Vendor daemon status (on shama):
        alterad: UP v7.0
        Feature usage info:
        Users of C4D5_512A: (Total of 100 licenses available)
```

If the command does not work as described above, your license file may not be set up correctly. Refer to "Automated Flow Does Not Run Properly" on page 234 for information on how to check your system path and registry settings. Additionally, refer to "Verifying that the LM_LICENSE_FILE Variable Is Set Correctly" on page 238.

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. Perform the steps below for your platform to set the variable.

☞ If you have multiple versions of software that uses a license.dat file (e.g., Quartus II Limited Edition and a full version of the Quartus II software), make sure that the `LM_LICENSE_FILE` variable points to the version of software you want to use with DSP Builder.

Other software products, such as LeonardoSpectrum, also use the `LM_LICENSE_FILE` variable to point to a license file. You can combine several **license.dat** files into one or you can specify multiple **license.dat** files in the steps below.

Windows NT

Perform the following steps:

1. Choose **Settings > Control Panel** (Windows Start menu).
2. Double-click the **System** icon in the Control Panel window.
3. In the **System Properties** dialog box, click the **Environment** tab.
4. Click the **System Variable** list to highlight it, and then in the **Variable** box, type `LM_LICENSE_FILE`.
5. In the **Value** box, type <*path to license file*>`\license.dat`.
6. Click **OK**.

Windows 2000

Perform the following steps:

1. Choose **Settings > Control Panel** (Windows Start menu).
2. Double-click the **System** icon in the Control Panel window.
3. In the **System Properties** dialog box, click the **Advanced** tab.
4. Click the **Environment Variables** button.

5. Click the **System Variable** list to highlight it, and then click **New**.

6. In the **Variable Name** box, type `LM_LICENSE_FILE`.

7. In the **Variable Value** box, type <*path to license file*>`\license.dat`.

8. Click **OK**.

Windows 98

Perform the following steps:

1. With a text editor, open your PC's autoexec.bat file.

2. Type the following environment variable on its own line in the **autoexec.bat** file:

   `set LM_LICENSE_FILE=`<*path to license file*>`\license.dat`

3. Save the autoexec.bat file.

4. Restart the PC.

I've Tried Everything but It Still Doesn't Work

- Try running the setup_DSPBuilder command (see "The Altera DSP Builder Folder Does Not Appear in the Simulink Library Browser" on page 233.
- Try adding the following paths to your system path:
  - **quartus/bin**
  - **matlab/bin**
- Uninstall and reinstall DSP Builder. After uninstalling DSP Builder, use the Windows Explorer to delete any DSP Builder files or directories that remain in the file system manually.

ModelSim Fails with the SignalCompiler-Generated Tcl Scripts

The SignalCompiler testbench reads (**vsim**) data files for each of the inputs of the design (extension **.salt**) during simulation. If these files are not present in the working directory, the simulation fails. The data files are created in Simulink during Simlink simulation when the SignalCompiler **Generate Stimuli for VHDL Testbench** option is turned on (**Testbench** tab).

The Simulink Library Browser Does Not Show Altera MegaCore Blocks

The Simulink Library Browser may not display Altera MegaCore functions if you installed DSP Builder before you installed the MATLAB software.

There are two ways to fix this problem.

- At the MATLAB prompt, change directory to <*DSP Builder installation directory*>**/Altlib**.

  Type `setup_dspbuilder` ↵.

- After installing MATLAB, you need to modify the files (or create them if they do not exist) the files **startup.m** and **StartupAlteraDspBuilder.m**, which should be located in the directory <*MATLAB root*>**\toolbox\local**.

  startup.m should contain the following lines:

  ```
  run('StartupAlteraDspBuilder');
  ```

  StartupAlteraDspBuilder.m should contain the following lines:

  ```
  path(path, 'c:\DSPBuilder\Altlib');
  dos('"c:\DSPBuilder\Altlib\GetAlteraPath"
      "c:\DSPBuilder\Altlib"');
  SetAlteraMegaCorePath;
  ```

Specifying LeonardoSpectrum, Synplify & Quartus II Path Information for SignalCompiler

To launch the LeonardoSpectrum, Synplify, or Quartus II software, SignalCompiler retrieves the software's path information from your PC's registry file. In DSP Builder version 2.0.0 or higher you can specify a different path for SignalCompiler to use for each of the tools. The <*installation path*>**\DSPBuilder\Altlib** directory contains an XML configuration file (**edaconfig.xml**) containing the paths. You can modify the file with a text editor or any XML editor such as XML Notepad.

The XML configuration file has three tags for each tool:

- `<GetPathFromRegistry>`<*on or off*>`</GetPathFromRegistry>`
- `<ForcedPath>`<*path*>`</ForcedPath>`
- `<ToolVersion>`<*version*>`</ToolVersion>`

The <GetPathFromRegistry> tag takes the values on and off. When it is on (default), SignalCompiler reads the path from the registry file. When it is off, SignalCompiler reads the path from the <ForcedPath> tag. Figure 149 shows an example.

Figure 149. Example XML Code Specifying EDA Tool Paths

```
<EdaConfig>
  <synplicity>
   <GetPathFromRegistry>on</GetPathFromRegistry>
   <ForcedPath>C:\synplicity\Synplify_71\bin</ForcedPath>
   <ToolVersion>7.1</ToolVersion>
  </synplicity>
  <quartus>
   <GetPathFromRegistry>on</GetPathFromRegistry>
   <ForcedPath>c:\quartus\bin</ForcedPath>
   <ToolVersion>2.0</ToolVersion>
  </quartus>
  <leonardospectrum>
   <GetPathFromRegistry>on</GetPathFromRegistry>
   <ForcedPath>C:\Exemplar\LeoSpec\OEM2002a_Altera_NIGHTLY_14\bin\win32</ForcedPath>
   <ToolVersion>2002a</ToolVersion>
  </leonardospectrum>
</EdaConfig>
```

For example, SignalCompiler runs LeonardoSpectrum from the **C:\Exemplar\LeoSpec\OEM2002a_Altera_NIGHTLY_14\bin\win32** directory.

If you modify the XML settings, ensure that the value for <ToolVersion> is the correct version for the software located at <ForcedPath>.

The Wizard for an IP MegaCore Block Does Not Launch

For the IP MegaCore wizards to operate properly, you must include the SignalCompiler block in the model. Set the top-level model file (i.e., working directory) so that it includes the path to your model.

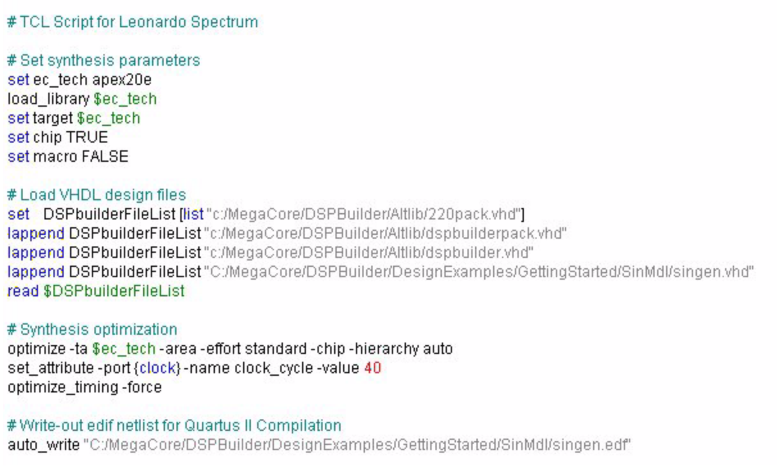
Paths in the MATLAB and Simulink software are case sensitive. Therefore, the path must be exactly as it appears in your system, including the case.

Notes:

Appendix—Example Tcl Scripts

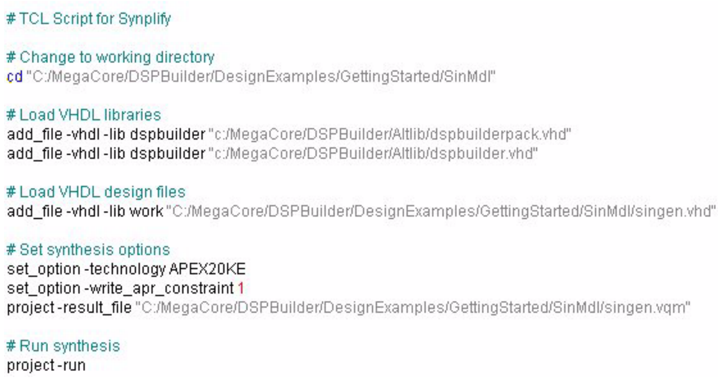
Figures 150 through 153 show sample Tcl scripts created by the SignalCompiler block for the amplitude modulation design example. See "DSP Builder Tutorial" on page 24 for information on how these files were generated.

Figure 150. Example LeonardoSpectrum Tcl Script



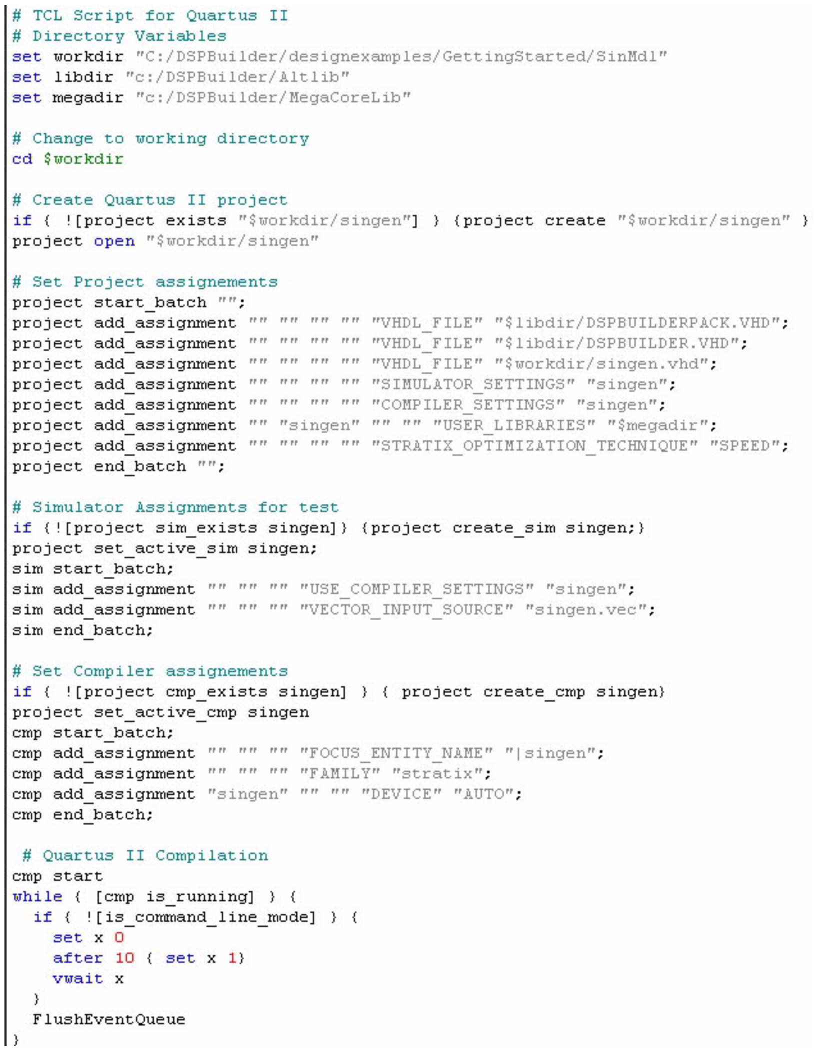
```
# TCL Script for Leonardo Spectrum

# Set synthesis parameters
set ec_tech apex20e
load_library $ec_tech
set target $ec_tech
set chip TRUE
set macro FALSE

# Load VHDL design files
set   DSPbuilderFileList [list "c:/MegaCore/DSPBuilder/Altlib/220pack.vhd"]
lappend DSPbuilderFileList "c:/MegaCore/DSPBuilder/Altlib/dspbuilderpack.vhd"
lappend DSPbuilderFileList "c:/MegaCore/DSPBuilder/Altlib/dspbuilder.vhd"
lappend DSPbuilderFileList "C:/MegaCore/DSPBuilder/DesignExamples/GettingStarted/SinMdl/singen.vhd"
read $DSPbuilderFileList

# Synthesis optimization
optimize -ta $ec_tech -area -effort standard -chip -hierarchy auto
set_attribute -port {clock} -name clock_cycle -value 40
optimize_timing -force

# Write-out edif netlist for Quartus II Compilation
auto_write "C:/MegaCore/DSPBuilder/DesignExamples/GettingStarted/SinMdl/singen.edf"
```

Figure 151. Example Synplify Tcl Script



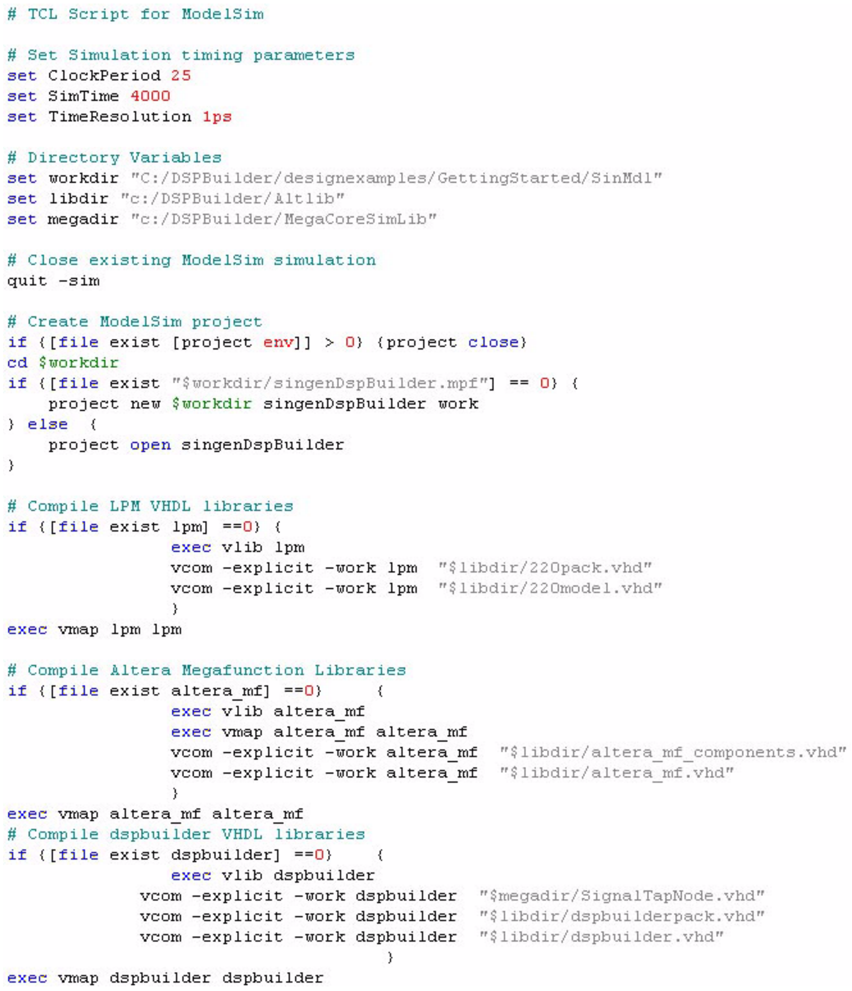
```
# TCL Script for Synplify

# Change to working directory
cd "C:/MegaCore/DSPBuilder/DesignExamples/GettingStarted/SinMdl"

# Load VHDL libraries
add_file -vhdl -lib dspbuilder "c:/MegaCore/DSPBuilder/Altlib/dspbuilderpack.vhd"
add_file -vhdl -lib dspbuilder "c:/MegaCore/DSPBuilder/Altlib/dspbuilder.vhd"

# Load VHDL design files
add_file -vhdl -lib work "C:/MegaCore/DSPBuilder/DesignExamples/GettingStarted/SinMdl/singen.vhd"

# Set synthesis options
set_option -technology APEX20KE
set_option -write_apr_constraint 1
project -result_file "C:/MegaCore/DSPBuilder/DesignExamples/GettingStarted/SinMdl/singen.vqm"

# Run synthesis
project -run
```

Figure 152. Example Quartus II Tcl Script



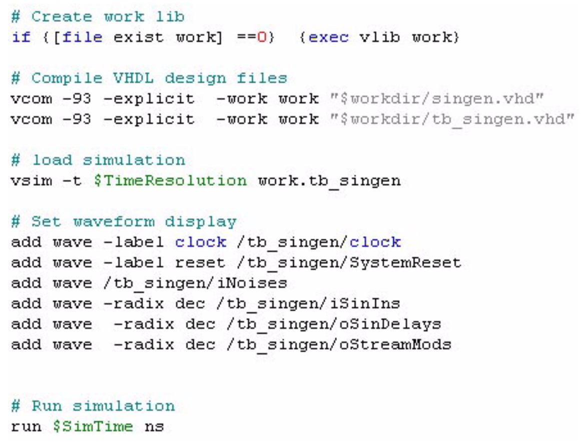
```
# TCL Script for Quartus II
# Directory Variables
set workdir "C:/DSPBuilder/designexamples/GettingStarted/SinMdl"
set libdir "c:/DSPBuilder/Altlib"
set megadir "c:/DSPBuilder/MegaCoreLib"

# Change to working directory
cd $workdir

# Create Quartus II project
if { ![project exists "$workdir/singen"] } {project create "$workdir/singen" }
project open "$workdir/singen"

# Set Project assignements
project start_batch "";
project add_assignment "" "" "" "" "VHDL_FILE" "$libdir/DSPBUILDERPACK.VHD";
project add_assignment "" "" "" "" "VHDL_FILE" "$libdir/DSPBUILDER.VHD";
project add_assignment "" "" "" "" "VHDL_FILE" "$workdir/singen.vhd";
project add_assignment "" "" "" "" "SIMULATOR_SETTINGS" "singen";
project add_assignment "" "" "" "" "COMPILER_SETTINGS" "singen";
project add_assignment "" "singen" "" "" "USER_LIBRARIES" "$megadir";
project add_assignment "" "" "" "" "STRATIX_OPTIMIZATION_TECHNIQUE" "SPEED";
project end_batch "";

# Simulator Assignments for test
if {![project sim_exists singen]} {project create_sim singen;}
project set_active_sim singen;
sim start_batch;
sim add_assignment "" "" "" "USE_COMPILER_SETTINGS" "singen";
sim add_assignment "" "" "" "VECTOR_INPUT_SOURCE" "singen.vec";
sim end_batch;

# Set Compiler assignements
if { ![project cmp_exists singen] } { project create_cmp singen}
project set_active_cmp singen
cmp start_batch;
cmp add_assignment "" "" "" "FOCUS_ENTITY_NAME" "|singen";
cmp add_assignment "" "" "" "FAMILY" "stratix";
cmp add_assignment "singen" "" "" "DEVICE" "AUTO";
cmp end_batch;

 # Quartus II Compilation
cmp start
while { [cmp is_running] } {
  if { ![is_command_line_mode] } {
    set x 0
    after 10 { set x 1}
    vwait x
  }
  FlushEventQueue
}
```

Figure 153. Example ModelSim Tcl Script



```
# TCL Script for ModelSim

# Set Simulation timing parameters
set ClockPeriod 25
set SimTime 4000
set TimeResolution 1ps

# Directory Variables
set workdir "C:/DSPBuilder/designexamples/GettingStarted/SinMdl"
set libdir "c:/DSPBuilder/Altlib"
set megadir "c:/DSPBuilder/MegaCoreSimLib"

# Close existing ModelSim simulation
quit -sim

# Create ModelSim project
if {[file exist [project env]] > 0} {project close}
cd $workdir
if {[file exist "$workdir/singenDspBuilder.mpf"] == 0} {
    project new $workdir singenDspBuilder work
} else  {
    project open singenDspBuilder
}

# Compile LPM VHDL libraries
if {[file exist lpm] ==0} {
                exec vlib lpm
                vcom -explicit -work lpm  "$libdir/220pack.vhd"
                vcom -explicit -work lpm  "$libdir/220model.vhd"
                }
exec vmap lpm lpm

# Compile Altera Megafunction Libraries
if {[file exist altera_mf] ==0}     {
                exec vlib altera_mf
                exec vmap altera_mf altera_mf
                vcom -explicit -work altera_mf  "$libdir/altera_mf_components.vhd"
                vcom -explicit -work altera_mf  "$libdir/altera_mf.vhd"
                }
exec vmap altera_mf altera_mf
# Compile dspbuilder VHDL libraries
if {[file exist dspbuilder] ==0}     {
                exec vlib dspbuilder
             vcom -explicit -work dspbuilder  "$megadir/SignalTapNode.vhd"
             vcom -explicit -work dspbuilder  "$libdir/dspbuilderpack.vhd"
             vcom -explicit -work dspbuilder  "$libdir/dspbuilder.vhd"
                                        }
exec vmap dspbuilder dspbuilder
```

Figure 154. Example ModelSim Script



```
# Create work lib
if {[file exist work] ==0}  {exec vlib work}

# Compile VHDL design files
vcom -93 -explicit  -work work "$workdir/singen.vhd"
vcom -93 -explicit  -work work "$workdir/tb_singen.vhd"

# load simulation
vsim -t $TimeResolution work.tb_singen

# Set waveform display
add wave -label clock /tb_singen/clock
add wave -label reset /tb_singen/SystemReset
add wave /tb_singen/iNoises
add wave -radix dec /tb_singen/iSinIns
add wave  -radix dec /tb_singen/oSinDelays
add wave  -radix dec /tb_singen/oStreamMods


# Run simulation
run $SimTime ns
```

Notes:

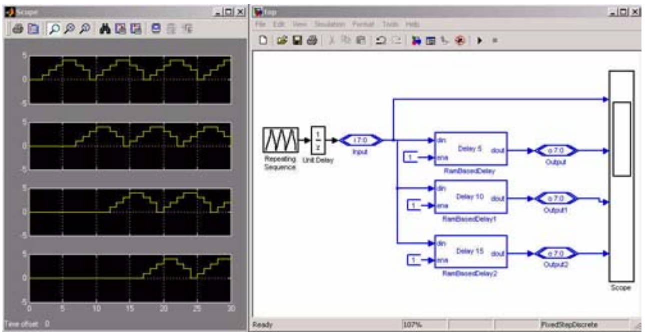
Appendix—Creating Custom Library Blocks

Overview

This appendix describes how to build a custom library block for use with Simulink and DSP Builder. Custom blocks are HDL subsystems in which the block functionality is described using DSP Builder primitives or IP. With this method, you can create a single design representation that can be used in Simulink and as VHDL RTL.

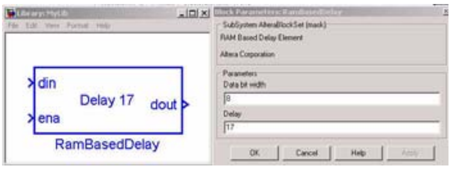
This appendix uses the design example **top.mdl** (see Figure 155), which is provided in <*DSP Builder path*>**\designexample\customlibrary**.

Figure 155. top.mdl Example



RamBasedDelay, which is used in **top.mdl**, is a paramerizable Simulink block that is compatible with DSP Builder flows (see Figure 156). RamBasedDelay has two parameters, **Data bit width** and **Delay**.

Figure 156. RamBasedDelay Block



SubSystem AlteraBlockSet (mask)

RAM Based Delay Element

Altera Corporation

Parameters

Data bit width

8

Delay

17

Create a Custom Block

To create a custom block, perform the following steps. The following sections describe these steps in greater detail.

1. Create a Library Model File with the HDL Subsystem Block
2. Build the HDL Subsystem Functionality
3. Create Parameters & Functionality Using the Mask Editor
4. Generate the MATLAB Initialization Script

Create a Library Model File with the HDL Subsystem Block

Create a Model File for your custom block by performing the following steps in the Simulink software:

1. Choose **New > Library** (File menu).
2. Choose **Save** (File menu).
3. Save the file, e.g, as **MyLib.mdl**.
4. Open the Simulink Library Browser.
5. Expand the DSP Builder library.
6. Expand the Altlab library.
7. Drag and drop the HDL Subsystem block into your model.
8. Click the text `HDL Subsystem`.
9. Rename the block as DelayFIFO.
10. Choose **Save** (File menu).

Build the HDL Subsystem Functionality

To add functionality to the DelayFIFO block, perform the following steps:

1. Double-click the DelayFIFO block. The **MyLib/DelayFIFO** window opens.
2. Remove the connection line between the AltBus and AltBus1 blocks.

3. Drag and drop the DSP Builder Shift Taps block from the Storage library into the **MyLib/DelayFIFO** window.

4. Position the block between the AltBus and AltBus1 blocks.

5. Double-click the Shift Taps block to open the block parameters dialog box.

6. Set the following parameters:

   – **Number of TAPs:** 1
   – **Distance Between Taps:** 10
   – **Use Clock Enable:** turn on
   – **Use Deicated Circuitry:** turn on

7. Click **OK**.

8. Select the In1 and AltBus blocks.

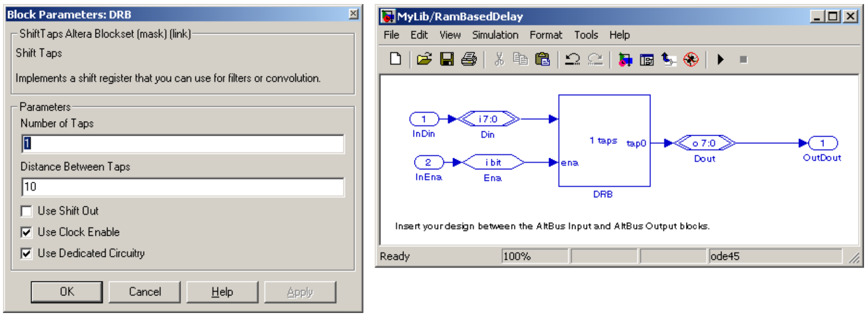
9. Copy and paste the blocks.

10. Rename the blocks as described below:

    – **Shift Taps:** DRB
    – **In1:** InDin
    – **Altbus:** Din
    – **In2:** InEna
    – **AltBus2:** Ena
    – **AltBus1:** Dout
    – **Out1:** OutDout

11. Draw connection lines from Din to the top of DRB, from Ena to the ena port on DRB, and from DRB to Dout. Figure 157 shows the completed design.

Figure 157. DelayFIFO Design



Block Parameters: DRB

ShiftTaps Altera Blockset (mask) (link)

Shift Taps

Implements a shift register that you can use for filters or convolution.

Parameters

Number of Taps: 1

Distance Between Taps: 10

☐ Use Shift Out

☑ Use Clock Enable

☑ Use Dedicated Circuitry

Insert your design between the AltBus Input and AltBus Output blocks.

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

13. Close the **MyLib/DelayFIFO** window.

Create Parameters & Functionality Using the Mask Editor

Create parameters for the DelayFIFO block using the Mask Editor by performing the following steps:

1. Right-click DelayFIFO in the MyLib model.

2. Choose **Edit Mask** from the pop-up menu.

3. Click the **Initialization** tab.

4. Make the following settings:

   – **Prompt:** Data Bit Width
   – **Variable:** bwl

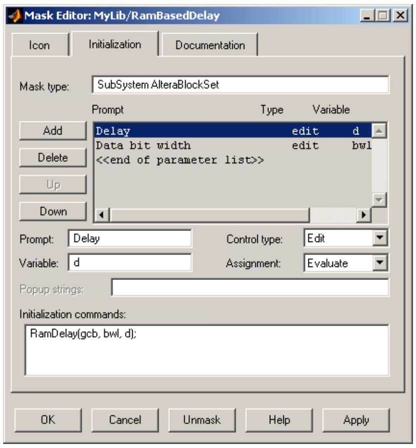
5. Click **Add**.

6. Make the following settings:

   – **Prompt:** Delay
   – **Variable:** d

7. Type `RamDelay(gcb, bwl, d);` in the **Initialization commands** box. Figure 158 shows the tab after you have made these settings.

Figure 158. Initialization Tab



Mask type: SubSystem AlteraBlockSet

Prompt	Type	Variable
Delay	edit	d
Data bit width	edit	bwl
<<end of parameter list>>		

Prompt: Delay
Control type: Edit
Variable: d
Assignment: Evaluate
Popup strings:

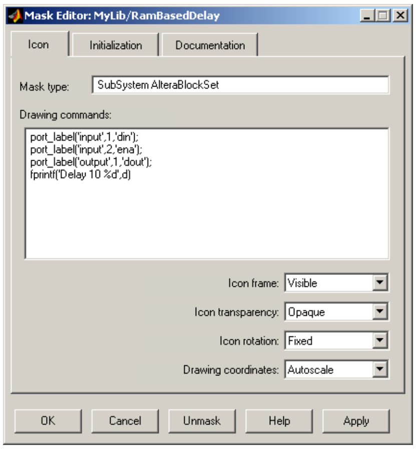
Initialization commands:

```
RamDelay(gcb, bwl, d);
```

8. Click the **Icon** tab.

9. Add the symbol information shown in Figure 159. (Optional)

Figure 159. Icon Tab



Mask type: SubSystem AlteraBlockSet

Drawing commands:

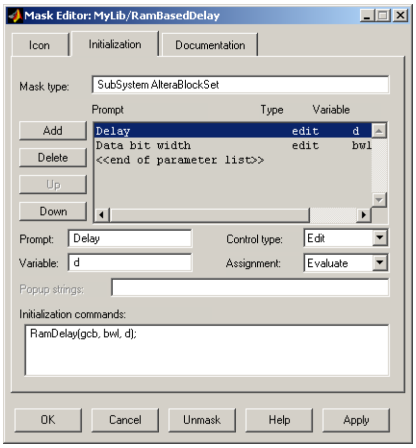
```
port_label('input',1,'din');
port_label('input',2,'ena');
port_label('output',1,'dout');
fprintf('Delay 10 %d',d)
```

Icon frame: Visible
Icon transparency: Opaque
Icon rotation: Fixed
Drawing coordinates: Autoscale

10. Click the **Documentation** tab.

11. Add symbol documentation information as shown in Figure 160. (Optional)

Figure 160. Documentation Tab



Mask type: SubSystem AlteraBlockSet

```
Prompt                      Type    Variable
Delay                       edit    d
Data bit width              edit    bwl
<<end of parameter list>>
```

Prompt: Delay
Control type: Edit
Variable: d
Assignment: Evaluate
Popup strings:
Initialization commands:

```
RamDelay(gcb, bwl, d);
```

12. Click **OK**.

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

Refer to The MathWorks documentation on using Simulink for additional information on the Mask Editor.

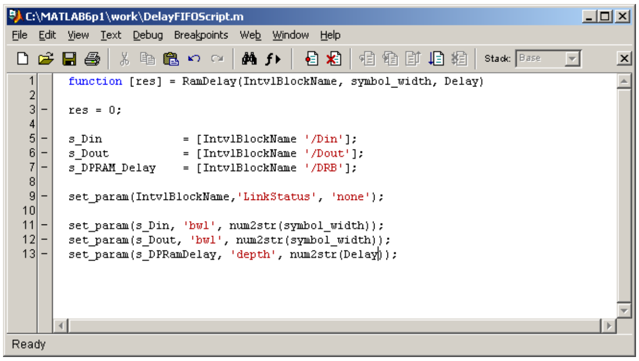
Generate the MATLAB Initialization Script

To pass parameters from the symbol's mask into the block, you use a MATLAB initialization script, **DelayFIFOScript.m**. To create this file, perform the following steps:

1. In the **MATLAB** window, choose **New > M-file** (File menu).

2. Type in the code shown in Figure 161.

Figure 161. DelayFFOScript.m File


C:\MATLAB6p1\work\DelayFIFOScript.m
File Edit View Text Debug Breakpoints Web Window Help
Stack: Base
Ready

```
function [res] = RamDelay(IntvlBlockName, symbol_width, Delay)

res = 0;

s_Din           = [IntvlBlockName '/Din'];
s_Dout          = [IntvlBlockName '/Dout'];
s_DPRAM_Delay   = [IntvlBlockName '/DRB'];

set_param(IntvlBlockName,'LinkStatus', 'none');

set_param(s_Din, 'bwl', num2str(symbol_width));
set_param(s_Dout, 'bwl', num2str(symbol_width));
set_param(s_DPRamDelay, 'depth', num2str(Delay));
```

The `set_param` command passes a parameter to a block, for example to specify that the input bit width is 8. For example:

```
set_param(s_Din, ‘bwl’, num2str(symbol_width));
```

Where:

– `symbol_width` is the input bit width
– `s_Din` is a string (the name of the block Din)
– `bwl` is the parameter name of the Din block (which is the DSP Builder block)

3. For your custom block to work properly, make sure that the files (**MyLib.Mdl**, and **DelayFIFOScript.m**) are either located in the working directory or in the MATLAB system path (refer to MATLAB help for information on setting the path).

For more information on M-files, search for the Simulink model construction command name in MATLAB online help (e.g., `get_param`, `set_param`, `bdroot`, `gcb`, `gcs`, etc.).