Chapter 1. About the DSP Builder Advanced Blockset

Base Blocks ................................................................. 1–2
Control Block ............................................................... 1–2
Hardware Generation .................................................... 1–2
Memory-Mapped Bus Interface ........................................ 1–2
Memory and Multiplier Trade-Off Options ............................ 1–2
Signals Block ................................................................. 1–4
Device Block ................................................................. 1–5
ModelPrim Blocks .......................................................... 1–5
Specifying the Output Data Type ....................................... 1–5
Using the Specify via Dialog Option ................................. 1–6
SynthesisInfo Block ....................................................... 1–6
WYSIWYG Style ............................................................. 1–7
Scheduled Style .............................................................. 1–7
ModelIP Blocks ............................................................... 1–8
Displaying the Latency for ModelIP Blocks ......................... 1–8
ModelBus Blocks ............................................................. 1–9
FFT Blockset ................................................................. 1–9
Cycle Accuracy and Latency ............................................. 1–9
Reading the Latency Parameter ........................................ 1–9
Protocol for Connecting IP .............................................. 1–10
Time-Division Multiplexing ............................................ 1–12
Multichannel Operation .................................................. 1–13
Vectorized Inputs ......................................................... 1–13
Channelization ............................................................... 1–14
Memory-Mapped Interfaces ............................................. 1–17
Hardware Interfaces ....................................................... 1–17
Design Semantics ........................................................... 1–19
Fixed Point Representation ............................................. 1–19
Sample Rate and Clocks ................................................ 1–19
Model Simulation ............................................................ 1–20
Generated Files ............................................................. 1–20
Creating a Quartus II Project ........................................... 1–22
Adding a DSP Builder Advanced Blockset Design to an Existing Quartus II Project 1–22
Adding Advanced Blockset Components to SOPC Builder ........ 1–22
Comparison with RTL ....................................................... 1–22
Automatic Testbench Flow ............................................. 1–22
Loading in ModelSim ...................................................... 1–23
Running from a Command ............................................. 1–23
Running All the Automatic Testbenches in a Design ................ 1–24
Run ModelSim Flow ........................................................ 1–25
Design Space Exploration .............................................. 1–25
Interoperability with the Standard Blockset ......................... 1–27
Learning to Use the Advanced Blockset .............................. 1–28

Chapter 2. Example Designs

Opening a Demonstration Design .................................... 2–2
Copying a Demonstration Design .................................... 2–4
<table>
<thead>
<tr>
<th>Reference Designs</th>
<th>Demonstration Designs</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-Antenna DUC for WiMAX</td>
<td>1-Carrier, 2-Antenna W-CDMA DDC</td>
</tr>
<tr>
<td>Decimating CIC Filter</td>
<td>16-Channel DDC</td>
</tr>
<tr>
<td>Interpolating CIC Filter</td>
<td>16-Channel DUC</td>
</tr>
<tr>
<td>Single Rate FIR Filter</td>
<td>2-Channel DUC</td>
</tr>
<tr>
<td>Decimating FIR Filter</td>
<td>2-Antenna DUC for WiMAX</td>
</tr>
<tr>
<td>Interpolating FIR Filter</td>
<td></td>
</tr>
<tr>
<td>Fractional Rate FIR Filter</td>
<td></td>
</tr>
<tr>
<td>Half Band FIR Filter</td>
<td></td>
</tr>
<tr>
<td>Root Raised Cosine FIR Filter</td>
<td></td>
</tr>
<tr>
<td>Filter Chain with Forward Flow Control</td>
<td></td>
</tr>
<tr>
<td>NCO</td>
<td></td>
</tr>
<tr>
<td>Real Mixer</td>
<td></td>
</tr>
<tr>
<td>Complex Mixer</td>
<td></td>
</tr>
<tr>
<td>Four Channel, Two Banks NCO</td>
<td></td>
</tr>
<tr>
<td>Four Channel, Four Banks NCO</td>
<td></td>
</tr>
<tr>
<td>Four Channel, Eight Banks, Two Wires NCO</td>
<td></td>
</tr>
<tr>
<td>Four Channel, 16 Banks NCO</td>
<td></td>
</tr>
<tr>
<td>Four Channel, 64 Banks NCO</td>
<td></td>
</tr>
<tr>
<td>Hello World</td>
<td></td>
</tr>
<tr>
<td>Fibonacci Series</td>
<td></td>
</tr>
<tr>
<td>Automatic Gain Control</td>
<td></td>
</tr>
<tr>
<td>Multi-Channel IIR Filter</td>
<td></td>
</tr>
<tr>
<td>8×8 Inverse Discrete Cosine Transform</td>
<td></td>
</tr>
<tr>
<td>Quadrature Amplitude Modulation</td>
<td></td>
</tr>
<tr>
<td>Radix 2 Streaming FFT</td>
<td></td>
</tr>
<tr>
<td>Radix 4 Streaming FFT</td>
<td></td>
</tr>
<tr>
<td>4K FFT</td>
<td></td>
</tr>
<tr>
<td>8K FFT</td>
<td></td>
</tr>
<tr>
<td>4K IFFT</td>
<td></td>
</tr>
<tr>
<td>8K IFFT</td>
<td></td>
</tr>
<tr>
<td>Test CORDIC Functions Using Primitive Blocks</td>
<td></td>
</tr>
<tr>
<td>Test CORDIC Functions Using the CORDIC Block</td>
<td></td>
</tr>
<tr>
<td>Memory-Mapped Registers</td>
<td></td>
</tr>
<tr>
<td>Scale</td>
<td></td>
</tr>
<tr>
<td>Tutorial Designs</td>
<td></td>
</tr>
<tr>
<td>ModelIP</td>
<td></td>
</tr>
<tr>
<td>Primitive Library</td>
<td></td>
</tr>
<tr>
<td>System</td>
<td></td>
</tr>
<tr>
<td>Reference Designs</td>
<td></td>
</tr>
<tr>
<td>4-Carrier, 2-Antenna W-CDMA DDC</td>
<td></td>
</tr>
<tr>
<td>1-Carrier, 2-Antenna W-CDMA DDC</td>
<td></td>
</tr>
<tr>
<td>4-Carrier, 2-Antenna W-CDMA DUC</td>
<td></td>
</tr>
<tr>
<td>1-Carrier, 2-Antenna W-CDMA DDC</td>
<td></td>
</tr>
<tr>
<td>4-Carrier, 2-Antenna High-Speed W-CDMA DUC at 368.64 MHz with Total Rate Change 32</td>
<td></td>
</tr>
<tr>
<td>4-Carrier, 2-Antenna High-Speed W-CDMA DUC at 368.64 MHz with Total Rate Change 48</td>
<td></td>
</tr>
<tr>
<td>4-Carrier, 2-Antenna High-Speed W-CDMA DUC at 307.2 MHz with Total Rate Change 40</td>
<td></td>
</tr>
<tr>
<td>1-Antenna WiMAX DDC</td>
<td></td>
</tr>
</tbody>
</table>

© November 2009  Altera Corporation
2-Antenna WiMAX DDC .......................................................... 2–29
1-Antenna WiMAX DUC ......................................................... 2–29
2-Antenna WiMAX DUC .......................................................... 2–30

Chapter 3. ModelIP Tutorial
The Numerically Controlled Oscillator Design .................................. 3–1
Opening the NCO Design ......................................................... 3–1
Setting the Configuration Parameters ............................................ 3–3
Configuring the NCO .................................................................. 3–3
Simulating the Model in Simulink .................................................. 3–10
Exploring the Generated Files ....................................................... 3–10
Simulating the RTL ..................................................................... 3–12
Compiling with the Quartus II Software ......................................... 3–13
Instantiating the Design in SOPC Builder ....................................... 3–14
Customizing the NCO .................................................................. 3–17
Changing the Number of Channels ................................................ 3–17
Increasing the Spurious Free Dynamic Range ................................... 3–19

Chapter 4. Primitive Library Tutorial
The Fibonacci Design ................................................................. 4–1
Creating the Fibonacci Model ....................................................... 4–1
Create a New Model ..................................................................... 4–1
Add Blocks from the ModelPrim Library ........................................ 4–2
Create a Synthesizable Subsystem ................................................ 4–4
Complete the Top-Level Model .................................................... 4–5
Simulating the Design in Simulink .................................................. 4–7
Using Vector Types ..................................................................... 4–8
Using Complex Types .................................................................. 4–9
Exploring the Generated Files ....................................................... 4–11
Simulating the RTL ..................................................................... 4–13
Compiling with the Quartus II Software ......................................... 4–14

Chapter 5. System Tutorial
System Level Design ................................................................. 5–1
The Digital Down Converter Design ............................................... 5–1
Signals Block ............................................................................. 5–3
Control Block ............................................................................ 5–4
Source Blocks ............................................................................. 5–5
Sink Blocks ................................................................................ 5–5
Tool Interface Blocks ................................................................... 5–5
DDCChip Subsystem ..................................................................... 5–6
Primary Inputs ............................................................................. 5–7
Merge Multiplexer ....................................................................... 5–8
NCO ......................................................................................... 5–8
Mixer ....................................................................................... 5–10
Mixer Scale Block ...................................................................... 5–12
DecimatingCIC and Scale Blocks .................................................. 5–14
Decimating FIR Blocks ................................................................ 5–17
Simulating the Design in Simulink .................................................. 5–21
Exploring the Generated Files ....................................................... 5–22
Simulating the RTL ..................................................................... 5–24
Compiling with the Quartus II Software ......................................... 5–25
Appendix A. Latency Management
Zer0 Latency Example ................................................................. A–1
Non-Explicit Delays ........................................................................ A–2
Distributed Delays ........................................................................ A–2
Using Latency Constraints .......................................................... A–4
Latency and $f_{\text{MAX}}$ Constraint Conflicts ................................. A–5
Using Latency Parameters ........................................................... A–5

Appendix B. Folding
Using Folding ............................................................................... B–1
  Folded Subsystem without TDM Demuxing ................................. B–2
  Folded Subsystem with TDM Demuxing ......................................... B–2
Effects of Folding ......................................................................... B–3
  Effects on Manual Delays ............................................................ B–3
  Effects on Combinational Logic ..................................................... B–4
Examples ..................................................................................... B–5

Index

Additional Information
Revision History ........................................................................ Info–1
How to Contact Altera ................................................................. Info–1
Typographic Conventions ........................................................... Info–1
The **DSP Builder Advanced Blockset** adds specialized Simulink libraries to the MATLAB design environment that allow you to implement DSP designs quickly and easily. The blockset is based on a high level synthesis technology that optimizes the high level, untimed netlist into low level, pipelined hardware targeted to your chosen Altera® FPGA device and chosen clock rate. The hardware is written out as plain text VHDL, along with scripts that integrate with the Quartus® II software and the ModelSim simulator.

The combination of these features allows you to create a design without needing detailed device knowledge, and generate a high quality implementation that runs on a variety of FPGA families with different hardware architectures.

By specifying your desired clock frequency, you can solve timing closure issues by generating register transfer level (RTL) code that is pipelined to meet your goal. Filters in the blockset automatically use a high clock rate to increase folding, and reduce hardware size.

The **DSP Builder Advanced Blockset** uses Simulink fixed-point types for all operations and requires licensed versions of the Simulink **Fixed Point Blockset** and **Fixed-Point Toolbox**. The **Signal Processing Blockset** and **Communications Blockset** are also recommended and are used in the demonstration designs.

For information about Simulink fixed-point types, the **Signal Processing Blockset** and the **Communications Blockset**, refer to the MATLAB Help.

You can use the advanced blockset entirely independently of the DSP Builder standard blockset, or its blocks can be embedded in top-level DSP Builder designs that use the standard blockset.

For information about interoperability with the DSP Builder standard blockset, refer to the **DSP Design Flow User Guide**.

The blockset comprises the following six Simulink libraries:

- Base Blocks
- Filters (ModelIP)
- Waveform Synthesis (ModelIP)
- ModelBus
- ModelPrim
- FFT Blockset

For detailed information about the advanced blockset libraries, refer to the **DSP Builder Advanced Blockset Reference Manual**.
Base Blocks

The top-level of a DSP Builder Advanced Blockset design is a testbench and must include Control and Signals blocks. A design may include any number of subsystems which can combine blocks from the DSP Builder Advanced Blockset with blocks from the MATLAB Simulink libraries.

The functional subsystem containing a Device block marks the top-level of the FPGA device and specifies the target device used for the generated hardware.

Other blocks are provided to control and view the signals in your design, and to automatically load your design into the ModelSim simulator or the Quartus II software.

Control Block

The Control block traverses your design, synthesizes the individual primitive or ModelIP blocks into RTL, and maintains an internal data flow representation of your design. The internal representation of your design model is then used for Simulink simulation and to write out the RTL and scripts for other tools.

A single Control block must be present at the top-level of your model. The name of this block (Control by default) is the root name used in many of the output files. For example, the top-level ModelSim simulation (.do) file is named after this block. There is rarely any need to rename this block.

The options in the Control block are applied globally to your design.

Hardware Generation

Options in the Control block specify whether hardware is generated for your model and the location of the generated RTL. You can also choose to create automatic RTL testbenches for each subsystem in your model and specify the depth of signals that are included when your model is simulated in ModelSim.

Memory-Mapped Bus Interface

You can specify the address and data bus widths used by the memory-mapped bus interface and choose whether the high-order byte of the address is stored in memory at the lowest address and the low-order byte at the highest address (Big Endian), or the high-order byte at the highest address and the low-order byte at the lowest address (Little Endian).

Memory and Multiplier Trade-Off Options

When your design is synthesized to logic, delay blocks are often created, whether explicitly from primitive delays, or in the ModelIP blocks. DSP Builder tries to balance the implementation between logic elements (LEs) and block memories (M512, M4K, M9K, or M144K). The trade-off depends on the selected FPGA family, but as a rough guideline the trade-off is set to minimize the absolute silicon area used. For example, if a block of RAM occupies the silicon area of two logic array blocks (LABs), then a delay that requires more than 20 logic elements (two LABs) is implemented as a block of RAM. This is generally appropriate, but there may be cases when you want to influence this trade-off.
Table 1–1 describes the memory and multiplier threshold trade-offs and provides some usage examples.

### Table 1–1. Memory and Multiplier Trade-Offs

<table>
<thead>
<tr>
<th><strong>CDelay RAM Block Threshold</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Trade-off between simple delay LEs and small ROM blocks. If any delay’s size is such that the number of LEs used is greater than this parameter, then the delay is implemented as block RAM.</td>
</tr>
<tr>
<td><strong>Default (–1)</strong></td>
<td>20 bits.</td>
</tr>
<tr>
<td><strong>Usage</strong></td>
<td>To make more delays use block RAM, enter a lower number—for example a value in the range 20 to 30. To use fewer block memories, enter a larger number—for example 100. To never use block memory for simple delays, set to a very large number, such as 10000.</td>
</tr>
<tr>
<td><strong>Notes</strong></td>
<td>Delays of length less than 3 cycles must be implemented in logic elements due to the nature of the block RAM behavior. This threshold only applies to implementing simple delays in memory blocks or logic elements. Dual Memories cannot be pushed back into logic elements. Some ModelIP blocks are built using dual memories, for example dual and multirate filters; these blocks always use some memory blocks, regardless of the value of this threshold.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>CDualMem Dist RAM Threshold</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Trade-off between small and medium RAM blocks. This threshold is similar to the CDelay RAM Block Threshold except that it applies only to the dual-port memories in the ModelPrim library. Any dual-port memory is always implemented in a block memory, rather than logic elements, but for some device families there may be different sizes of block memory available. The threshold value determines which medium size RAM memory blocks are used instead of small memory RAM blocks. For example, the threshold which determines whether to use M9K blocks rather than MLAB blocks on Stratix® III and Stratix IV devices.</td>
</tr>
<tr>
<td><strong>Default (–1)</strong></td>
<td>1,280 bits.</td>
</tr>
<tr>
<td><strong>Usage</strong></td>
<td>Using Stratix III devices with the default threshold value (-1), dual memories greater than 1,280 bits are implemented as M9Ks and dual memories less than or equal to 1,280 bits are implemented as MLABs. If you change this threshold to a lower value such as 200, then dual memories greater than 200 bits are implemented as M9Ks and dual memories less than or equal to 200 bits are implemented as MLABs.</td>
</tr>
<tr>
<td><strong>Notes</strong></td>
<td>For families with only one type of memory block (for example Cyclone® II with only M4K, or Cyclone III with only M9K) this threshold has no affect.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>M-RAM Threshold</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Trade-off between medium and large RAM blocks. For larger delays, memory can be implemented in medium block RAM (M4K, M9K) or using larger M-RAM blocks (M512K, M144K).</td>
</tr>
<tr>
<td><strong>Default (–1)</strong></td>
<td>1,000,000 bits.</td>
</tr>
<tr>
<td><strong>Usage</strong></td>
<td>If the number of bits in a memory or delay is greater than this threshold, M-RAM is used for the implementation. If you set a large value such as the default of 1,000,000 bits, M-RAM blocks are never used.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Hard Multiplier Threshold</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Trade-off between hard and soft multipliers. For devices that support hard multipliers or DSP blocks, these resources can be used instead of a soft multiplier made from logic elements. For example, a 2-bit (\times) 10-bit multiplier consumes very few logic elements. The hard multiplier threshold value corresponds to the number of logic elements that are used to save a multiplier. If the hard multiplier threshold value is 100, you are allowing 100 logic elements. Therefore, an 18×18 multiplier (that requires approximately (18^2 = 350) logic elements) is not transferred to logic elements because it requires more logic elements than the threshold value. However, a 16×4 multiplier that requires approximately 64 logic elements is implemented as a soft multiplier with this setting.</td>
</tr>
</tbody>
</table>
Signals Block

Each model must have a Signals block and this should be placed at the top-level of your model. The Signals block specifies the details for the clocks and resets that are used to drive the generated logic.

The DSP Builder Advanced Blockset uses a single system clock to drive the main datapath logic, and, optionally, a second bus clock to provide an external processor interface. The rationale for this is that it is most efficient to drive the logic at as high a clock rate as possible. The standard DSP Builder blockset is therefore more suitable for managing multiple clock domains for example, when interfacing to external logic.

The Signals block provides a mechanism to name the clock, reset, and memory bus signals used in the RTL, and also provides information about the clock rate. This is important for two reasons:

- To calculate the ratio of sample rate to clock rate used to determine the amount of folding (time-division multiplexing) in the ModelIP filters.
- To determine the pipelining required at each stage of logic. For example, the amount of pipelining used in hard multipliers is modified, and also long adders are pipelined into smaller adders based on the absolute clock speed in relation to the FPGA family and speed-grade specified in the device block.

Cyclone device families do not support a separate bus clock due to the limited multiple clock support in block RAMs on those devices. For this reason, you must de-activate the separate bus clock option for the Signals block when using Cyclone device families.

### Table 1–1. Memory and Multiplier Trade-Offs

<table>
<thead>
<tr>
<th>Default (–1)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage</td>
<td>The default (–1) means always use hard multipliers. (With this value, 24×18 is implemented as two 18×18 multipliers). Set a value of approximately 300 to keep 18×18 multipliers hard, but transform smaller multipliers to logic elements. Note that a 24×18 multiplier is implemented as 6×18 + 18×18, so this setting builds the hybrid multipliers that are required. Set a value of approximately 1000 to implement the multipliers entirely as logic elements. Essentially you are allowing a high number (1000) of logic elements to save an 18×18 multiplier. Set a values of approximately 10 to implement a 24×16 multiplier as a 36×36 multiplier. With this value, you are not even allowing the adder to combine the two multipliers, therefore the system has to burn a 36×36 multiplier in a single DSP block.</td>
</tr>
<tr>
<td>Notes</td>
<td>Multipliers with a single constant input are converted into balanced adder trees. This occurs automatically where the depth of the tree is not greater than 2. If the depth is greater than 2, the hard multiplier threshold is compared with the estimated size of the adder tree. This is generally much lower than the size of a full soft multiplier. If two non-constant multipliers followed by an adder are combined into a single DSP block, the multiplier is not converted into logic elements no matter how large the threshold.</td>
</tr>
</tbody>
</table>
Device Block

The Device block marks a particular Simulink subsystem as the top-level of an FPGA device and allows you to specify the target device and speed grade for the device.

The DSP Builder Advanced Blockset supports the following target Altera device families: Stratix, Stratix GX, Stratix II, Stratix II GX, Stratix III, Stratix IV, Cyclone II, Cyclone III, and Arria® II GX.

All blocks in subsystems below this level of hierarchy, become part of the RTL design.
All blocks above this level of hierarchy become part of the testbench.

ModelPrim Blocks

The ModelPrim library allows you to create fast efficient designs captured in the behavioral domain rather than the implementation domain by combining primitive functions. For example, you can use a Delay block and let the tool decide how to implement that delay. You do not need to understand the details of the underlying FPGA architecture, as the primitive blocks are automatically mapped into efficient FPGA constructs.

Signals that pass through a single ChannelIn, ChannelOut, GPIn, or GPOut block are guaranteed to line up correctly in time at their boundaries. However, the timing relationship between different sets of inputs and outputs are not specified, and are not guaranteed to have any fixed relationship if clock frequencies or devices are changed.

You should use the protocol described in “Protocol for Connecting IP” on page 1–10 to decode outputs, rather than exact clock counting where possible.

You can design and debug your model quickly using zero-latency blocks, without having to track block latencies around your design. The additional latency required to meet the timing constraints for your model is calculated and displayed as a parameter on the ChannelOut block.

Specifying the Output Data Type

Many of the primitive blocks provide the following options in their Block Parameters dialog box to inherit or specify the output data type:

- **Inherit via internal rule**: The number of integer and fractional bits is equal to the maximum of the number of bits in the input data types. Word growth occurs if the input data types are not identical.

- **Inherit via internal rule with word growth**: The number of fractional bits is equal to the maximum of the number of fractional bits in the input data types. The number of integer bits is the maximum of the number of integer bits in the input data types plus one. This additional word growth allows for subtracting the most negative number from 0 which would exceed the maximum positive number that can be stored in the number of bits of the input.

- **Specify via dialog**: You can set the output type of the block explicitly using additional fields that are available when this option is selected.

- **Boolean**: The output type is Boolean.
In general, the output type of a primitive block can be set to **Inherit via internal rule**, or **Inherit via internal rule with word growth**. The type propagates through your design naturally, with blocks potentially growing the word length where necessary. The exception to this is if there are loops in your design, when the output type must be specified for at least one block in the loop using the dialog box.

For information about the options supported by each block, refer to the *ModelPrim Library* chapter in the *DSP Builder Advanced Blockset Reference Manual*.

**Using the Specify via Dialog Option**

Specifying the output type using the dialog box is a casting operation. This operation does not preserve the numerical value, just the underlying bits. This never adds hardware to a block—just changes the interpretation of the output bits.

For example, a *Mult* block with both input data types specified as *sfix16_En15* naturally has an output type of *sfix32_En30*. If you specify the output data type as *sfix32_En28*, the output numerical value is effectively multiplied by four, and a 1*1 input gives an output value of 4.

If you specify output data type of *sfix32_En31*, the output numerical value is effectively divided by two and a 1*1 input gives an output value of 0.5.

If you want to change the data type format in a way that preserves the numerical value, use a *Convert* block, which adds the corresponding hardware. Adding a *Convert* block directly after a primitive block allows you to specify the data type in a way that preserves the numerical value. For example, a *Mult* block followed by a *Convert* block, with input values 1*1 always gives output value 1.

**SynthesisInfo Block**

The ModelPrim library includes a *SynthesisInfo* block that sets the synthesis mode and labels a subsystem described by primitive blocks as the top-level of a synthesizable subsystem tree. The subsystem, and all those below it are flattened and synthesized as a unit. If no *SynthesisInfo* block is present, the style defaults to **WYSIWYG** and error messages may be issued if there is insufficient delay provided.

The inputs and outputs to this subsystem become the primary inputs and outputs of the RTL entity that is created. After running a Simulink simulation, the online Help page for the *SynthesisInfo* block is updated to show the latency, port interface, and estimated resource utilization for the current primitive subsystem.

The *SynthesisInfo* block can be at the same level as the *Device* block (if the synthesizable subsystem is the same as the generated hardware subsystem). However, it is often convenient to create a separate subsystem level that contains the *Device* block. Refer to the demonstration designs for some examples of design hierarchy.

There are two styles of operation during synthesis: **WYSIWYG** and **Scheduled**.
WYSIWYG Style

WYSIWYG is the default style of operation and can be useful when you want full control over the pipelining in a system. Every primitive that requires registering (such as an adder or multiplier) must be followed immediately by a Delay primitive. The delay primitive is absorbed in the preceding block to satisfy its delay requirements. Error messages are issued if there is insufficient delay provided.

The primitive logic blocks (And, Or, Nand, Not, Nor, Xnor, Xor) and the Simulink Mux and Demux blocks do not require a register.

Scheduled Style

The Scheduled style of operation uses a pipelining and delay distribution algorithm that creates fast hardware implementations from an easily described untimed block diagram. This style takes full advantage of the automatic pipelining capability. The algorithm performs the following operations:

1. Reads in and flattens your design model for any subsystem that contains a SynthesisInfo block.
2. Builds an internal graph to represent the logic.
3. Based on the absolute clock frequency requested, adds enough pipeline stages to meet that clock frequency. For example, long adders may be pipelined into several shorter adders. This additional pipelining helps reach high clock frequencies.

There are two main cases to consider:

- The simpler case is feed-forward. When there are no loops, feed-forward data paths are balanced to ensure that all the input data reaches each functional unit in the same cycle. After analysis, delays are inserted on all the non-critical paths to balance out the delays on the critical path.

- The case with loops is more complex. All loops are created with a delay to avoid combinational loops that cannot be analyzed by Simulink. Typically, there are one or more lumped delays. The delay around the loop must be preserved for correct operation, therefore delays from the lumped delay are borrowed by the functional units that need them.

Because loops can intersect or be nested with each other, a set of simultaneous constraints must be observed. The synthesis engine solves these constraints, distributes delay around the functional units as required, and leaves any residual delay as lumped delay elements. There may be cases when there is insufficient delay to distribute, in which case an error is generated. The following list give the delay requirements for some typical blocks:

- Boolean logic operations: 0 delay
- Adders: 1 delay, but potentially more at higher clock rates
- Multiplexers: 1 delay
- Multipliers: 3 cycles or 4 at higher clock rates

When you have selected the Scheduled style, you can optionally specify a latency constraint limit that can be a workspace variable or expression but must evaluate to a positive integer.
ModelIP Blocks

The ModelIP libraries include parameterizable multichannel filters and waveform synthesis blocks that allow you to quickly create designs for digital front-end applications. The ModelIP blocks are provided in the following libraries:

- The Filters library contains several decimating and interpolating cascaded integrator-comb (CIC), and finite impulse response (FIR) filters including single-rate, multi-rate, and fractional rate FIR filters.
- The Waveform Synthesis library contains a numerically controlled oscillator (NCO), complex mixer, and real mixer blocks.

After running a Simulink simulation, the online Help page for each ModelIP block is updated to show specific design documentation describing its implementation in your design. This information typically includes the latency, port interface, and resource utilization. For the blocks in the Filters library, the updated Help page also includes details of the parameterization, input and output data formats, and memory interface.

Displaying the Latency for ModelIP Blocks

You can display the latency added by a ModelIP block by adding the `<latency>` parameter as annotation on the block.

For example, Figure 1–1 shows the `<latency>` parameter added in the Block Annotation tab of the Block Properties dialog box for the NCO block. After you run a simulation, the added latency is shown as a text annotation below the block.

Figure 1–1. NCO Block Parameters Dialog Box
ModelBus Blocks

The ModelBus library provides memories and registers that can be accessed in your DSP datapath and via an external interface. You can use these blocks to configure coefficients or run-time parameters and to read calculated values. This library also includes blocks that you can use to simulate the bus interface in the Simulink environment.

FFT Blockset

The FFT Blockset library contains common blocks that support fast Fourier transform (FFT) design. It also includes several blocks that support the Radix-2^2 algorithm.

For information about the Radix-2^2 algorithm, refer to A New Approach to Pipeline FFT Processor – Shousheng He & Mats Torkleson, Department of Applied Electronics, Lund University, Sweden.

Cycle Accuracy and Latency

The DSP Builder Advanced Blockset supports the following design styles:

- Designs created using ModelIP blocks, such as the FIR and CIC filters. These blocks are 100% cycle accurate and the Simulink behavior represents exactly the RTL behavior. You can turn on display of the latency added by each block as described in “Displaying the Latency for ModelIP Blocks” on page 1–8.

- Designs using synthesized primitive blocks from the ModelPrim library. These blocks are 100% cycle accurate at their boundaries, therefore interfacing to other blocks is straightforward.

  The primitive blocks are designed internally as untimed circuits, so are not cycle accurate. In fact, there is not even a one-to-one mapping between the blocks in the Simulink model and the blocks are used to implement your design in RTL. It is this decoupling of design intent from design implementation that gives rise to the productivity benefits. The boundary between the untimed block and the outsized, cycle accurate world is the ChannelOut block. This block models the additional delay introduced by the RTL, so that data going in to the ChannelOut block is delayed internally, before being presented to the outside world. The latency of the block is displayed on the ChannelOut mask.

- You can also use primitive blocks outside of synthesizable subsystems. This can be useful to create glue logic around other subsystems. The Boolean logic and delay blocks are cycle accurate, but other primitive blocks are not.

Reading the Latency Parameter

You can read the added latency value for a ModelIP block (or for the ChannelOut ModelIP block) by selecting the block and typing the following command:

```
get_param(gcb, 'latency')
```

You can also use this command in an m-script, for example when you want to use the returned latency value to balance delays with external circuitry.
If you use an m-script to get this parameter and set latency elsewhere in your design, by the time it is updated and set on the ModelIP block, it is too late to initialize the delays elsewhere. This means that you must run your design twice after any changes to make sure that you have the correct latency. If you are scripting the whole flow, your must run once with end time 0, and then run again immediately with the desired simulation end time.

For more information about latency, refer to “Latency Management” on page A–1.

Protocol for Connecting IP

ModelIP or synthesized subsystems can be connected together using the three signals data, valid, and channel. These three wires connect most of the blocks in a DSP Builder Advanced Blockset design. Data on the data wire is only valid when the valid wire is asserted high. During this clock cycle, the channel carries an 8-bit integer channel identifier. This channel identifier is preserved through the data path so that data can be easily tracked and decoded.

This simple protocol is easy to interface with external circuitry when required and avoids the necessity of balancing delays, and counting cycles, because you can simply decode the valid and channel signals to determine when to capture the data in any downstream blocks. The control structures are distributed in each block of your design.

For example, Figure 1–2 shows the Interpolating FIR filter demonstration design.
Notice how the FilterSystem and Channel Viewer blocks are connected using just the three data, valid, and channel signals.

In the FIR filters, the valid signal is an enable, therefore for as long as the sample rate is less than or equal to that specified in the dialog box, data can be passed more slowly to the filter. This is useful when the sample rates of the filters are not all divisible by the clock rate.

Always, take care to generate the first valid signal when there is some real data.

In primitive subsystems, all signals connected to ChannelOut blocks are guaranteed to line up in the same clock cycle. That is, the delays on all paths from and to these blocks are balanced. However, you must ensure all the signals arrive at a ChannelIn block in the same clock cycle.

The ModelIP blocks follow the same rules. This makes it easy to connect ModelIP and primitive subsystems together.

The ModelIP filters all use the same protocol with an additional simplification—all the channels for a frame in a multichannel filter are produced in adjacent cycles. This is also a requirement on the filter inputs. If a FIR filter needs to use flow control, then the valid signal should be pulled down between frames of data, that is, just before channel 0 data is presented.

The same <data, valid, channel> protocol connects all CIC and FIR filter blocks and all subsystems defined using primitive blocks. The blocks in the Waveform Synthesis library support separate real and complex (or sine and cosine) signals. Some splitting or combining logic may be required when using the mixer blocks. You can implement this logic using a primitive subsystem.

Figure 1–3 shows how a splitter is required to extract the real and imaginary channels when connecting a CIC or FIR filter to the Complex Mixer. Similarly, a combiner may be required when connecting the output from a mixer to a downstream filter.

For an example that connects a CIC Filter, NCO, and Complex Mixer blocks, refer to “16-Channel DUC” on page 2–6. For an example that connects NCO, Real Mixer, and CIC Filter blocks, refer to “16-Channel DUC” on page 2–6.
Time-Division Multiplexing

Hardware utilization is optimized by using time-division multiplexing (TDM). The TDM factor (or folding factor) is the ratio of the clock rate to the sample rate.

By clocking a ModelIP block faster than the sample rate, you can re-use the same hardware. For example, by implementing a filter with a TDM factor of 2 and an internal clock multiplied by 2, you can halve the required hardware as shown in Figure 1–4.

**Figure 1–4.** Time-Division Multiplexing to Save Hardware Resource

To achieve the TDM, a serializer and deserializer are required before and after the reused hardware block to control the timing. The ratio of system clock frequency to sample rate determines the amount of resource saving except for a small amount of additional logic for the serializer and deserializer.

**Table 1–2** shows the estimated resource required for a 49-tap symmetric FIR filter.

**Table 1–2.** Example Resource Saving for a 49-Tap Single rate FIR Filter

<table>
<thead>
<tr>
<th>Clock Rate (MHz)</th>
<th>Sample Rate (MSPS)</th>
<th>Logic</th>
<th>Multipliers</th>
<th>Memory Bits</th>
<th>TDM Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td>72</td>
<td>2230</td>
<td>25</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>144</td>
<td>72</td>
<td>1701</td>
<td>13</td>
<td>468</td>
<td>2</td>
</tr>
<tr>
<td>288</td>
<td>72</td>
<td>1145</td>
<td>7</td>
<td>504</td>
<td>4</td>
</tr>
<tr>
<td>72</td>
<td>36</td>
<td>1701</td>
<td>13</td>
<td>468</td>
<td>2</td>
</tr>
</tbody>
</table>

When the sample rate equals the clock rate, because the filter is symmetric you just need 25 multipliers. When the clock rate is increased to 2 × sample rate, the number of multipliers required drops by half to 13. When the clock rate is set to 4 × sample rate, the number of multipliers required drops to 7. If the clock rate stays the same while the new data sample rate is only 36 MSPS (million samples per second), the resource consumption is the same as for the 2 × sample rate case.
Multichannel Operation

You can build multichannel systems directly using the required channel count, rather than creating a single channel system and scaling it up. The block diagrams use vectors of wires to scale without having to cut and paste multiple blocks.

Each channel is an independent data source. In an IF Modem design, two channels are required for the complex pair from each antenna.

The ModelIP blocks are vectorizable, meaning that if data going into a block is a vector requiring multiple instances of, for example a FIR filter, then multiple FIR blocks are created in parallel behind a single ModelIP block. If a decimating filter requires a smaller vector on the output, then data from individual filters are time division multiplexed onto the output vector automatically. This relieves the necessity to glue filters together with custom logic.

Consider the following two cases:

- ModelIP blocks typically take a channel count as a parameter. This is simple to conceptualize. The channels are numbered 0 to (N-1), and you can uses the channel indicator at any point to filter out some channels. To merge two streams, some logic must be created to multiplex the data together, and sequence and counter blocks are used to regenerate suitable valid and channel signals.

- Primitive Subsystems. The primitive subsystems contain Channel In and Channel Out blocks, but do not have explicit support for multiple channels. However, it is very easy to create multichannel logic: Simply draw out the logic required for your design to create a single channel version. To transform this to a multichannel system, increase all the delays by the channel count required. This can be parameterized using a mask variable to create a parameterizable component. For an example, refer to the “Multi-Channel IIR Filter” on page 2–17.

Vectorized Inputs

The data inputs and outputs for the ModelIP blocks can be vectors. This capability is used when the clock rate is insufficiently high to carry the total aggregate data. For example, 10 channels at 20 MSPS require 10×20 = 200 MSPS aggregate data rate. If the system clock rate was set to 100 MHz, then two wires are required to carry this data, and so the Simulink model uses a vector of width 2.

This approach is unlike traditional methods because you do not need to manually instantiate two ModelIP blocks and pass a single wire to each in parallel. Each ModelIP block internally vectorizes itself to, for example, build two FIR filters in parallel, and wire one element of the vector up to each FIR. The same paradigm is used on outputs, where high data rates on multiple wires are represented as vectors.

The input and output wire counts are determined by each ModelIP block, based on the clock rate, sample rate, and number of channels.
The output wire count is additionally affected by any rate changes in the ModelIP block. If there is a rate change, such as interpolating by two, then the output aggregate sample rate doubles. The output channels are then packed into the fewest number of wires (vector width) that will support that rate. For example, an interpolate by two FIR filter may have two wires at the input, but three wires at the output.

Any necessary multiplexing and packing is performed by the ModelIP block. The blocks connected to the inputs and outputs must have the same vector widths. This is enforced by Simulink. Vector width errors can usually be resolved by carefully changing the sample rates.

**Channelization**

The number of wires and the number of channels carried on each wire are determined by parameterization which you can specify using the following variables:

- **ClockRate** is the system clock frequency.
- **SampleRate** is the data sample rate per channel (MSPS).
- **ChanCount** is the number of channels.

Channels are enumerated from 0 to **ChanCount** – 1.

- The **Period** (or TDM factor) is the ratio of the clock rate to the sample rate and determines the number of available time slots:
  
  \[
  \text{Period} = \text{floor}(\text{ClockRate} / \text{SampleRate})
  \]

- The number of channel wires required to carry all the channels can be calculated by dividing the number of channels by the TDM factor:
  
  \[
  \text{ChanWireCount} = \text{ceil}(\text{ChanCount} / \text{Period})
  \]

- The number of channels carried per wire is calculated by dividing the number of channels by the number of channels per wire:
  
  \[
  \text{ChanCycleCount} = \text{ceil}(\text{ChanCount} / \text{ChanWireCount})
  \]

  The channel signal counts through 0 to **ChanCycleCount** – 1.

**Figure 1–5 on page 1–15** shows how a TDM factor of 3 combines two input channels into a single output wire (ChanCount = 2, ChanWireCount = 1, ChanCycleCount = 2).

If the number of channels is greater than the period, multiple wires are required. Each ModelIP block in your design is internally vectorized to build multiple blocks in parallel.

**Figure 1–6 on page 1–15** shows how a TDM factor of 3 combines four input channels into two wires (ChanCount = 4, ChanWireCount = 2, ChanCycleCount = 2).
Chapter 1: About the DSP Builder Advanced Blockset

Multichannel Operation

Figure 1–5. Channelization for Two Channels with a TDM Factor of 3

![Diagram for Channelization for Two Channels with a TDM Factor of 3]

Note to Figure 1–5:
(1) In this example, there are three available time slots in the output channel and every third time slot has a “don't care” value when the valid signal is low. The value of the channel signal while the valid signal is low does not matter.

Figure 1–6. Channelization for Four Channels with a TDM Factor of 3

![Diagram for Channelization for Four Channels with a TDM Factor of 3]

Note to Figure 1–6:
(1) In this example, two wires are required to carry the four channels and the cycle count is two on each wire. The channels are evenly distributed on each wire leaving the third time slot as don’t care on each wire.

The input and output data channel format used by a FIR or CIC filter are shown in the generated Help page for the block after you have run a Simulink simulation.
The channel signal is used for synchronization and scheduling of data. It specifies the channel data separation per wire. Note that the channel signal counts from 0 to \( ChanCycleCount - 1 \) in synchronization with the data. Thus, for \( ChanCycleCount = 1 \), the channel signal is the same as the channel count, enumerated 0 to \( ChanCount - 1 \).

For a case with more than a single data wire, this is not equal to the channel count on data wires, but specifies the synchronous channel data alignment across all the data wires. For example, Figure 1–7 shows the case for four channels of data on one data wire with no invalid cycles.

**Figure 1–7.** Four Channels on One Wire

<table>
<thead>
<tr>
<th>valid</th>
<th>channel</th>
<th>data0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>c0(0)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>c1(0)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>c2(0)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>c3(0)</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>c0(1)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>c1(1)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>c2(1)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>c3(1)</td>
</tr>
</tbody>
</table>

For the single wire case, the channel signal is the same as a channel count. However, for \( ChanWireCount > 1 \), the channel signal specifies the channel data separation per wire, rather than the actual channel number: it counts from 0 to \( ChanCycleCount -1 \) rather than 0 to \( ChanCount -1 \). Figure 1–8 shows the case for four channels on two wires with no invalid cycles.

**Figure 1–8.** Four Channels on Two Wires

<table>
<thead>
<tr>
<th>valid</th>
<th>channel</th>
<th>data0</th>
<th>data1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>c0(0)</td>
<td>c2(0)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>c1(0)</td>
<td>c3(0)</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>c0(1)</td>
<td>c2(1)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>c1(1)</td>
<td>c3(1)</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>c0(2)</td>
<td>c2(2)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>c1(2)</td>
<td>c3(2)</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>c0(3)</td>
<td>c2(3)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>c1(3)</td>
<td>c3(3)</td>
</tr>
</tbody>
</table>

Notice that the channel signal remains a single wire, not a wire for each data wire. It counts over 0 to \( ChanCycleCount -1 \). Figure 1–9 shows the case with four channels simultaneously on four wires.

**Figure 1–9.** Four Channels on Four Wires

<table>
<thead>
<tr>
<th>valid</th>
<th>channel</th>
<th>data0</th>
<th>data1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>c0(0)</td>
<td>c2(0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c0(1)</td>
<td>c3(0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c0(2)</td>
<td>c2(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c0(3)</td>
<td>c3(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c0(4)</td>
<td>c2(2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c0(5)</td>
<td>c3(2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c0(6)</td>
<td>c2(3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c0(7)</td>
<td>c3(3)</td>
</tr>
</tbody>
</table>
The ModelIP block is shown as a single block in Simulink but the data input and output wires from the ModelIP blocks are shown as a vector with multiple dimension. Multiples wires are required to accommodate all the channels and the Simulink model uses a vector of width 2 (Figure 1–10).

**Figure 1–10.** Simulink and Hardware Representations of a Single Rate FIR Filter

To display the ChanWireCount in Simulink, point to Port/Signal Displays in the Format menu and click Signal Dimensions.

For more information about channelization in a real design, refer to *AN 544: Digital Modem Design with the DSP Builder Advanced Blockset.*

**Memory-Mapped Interfaces**

A pipelined memory-mapped interface is built for all the programming registers in a design. For example, in the demo_ddc design, these include coefficient registers for filters, control registers for NCO frequencies, and individual control registers that you can use for any purpose such as providing programmable scaling. The register map for your design is gathered into an XML file that is processed to produce design documentation that is displayed in updated online Help for the Control block.

**Hardware Interfaces**

A typical ModelIP block has four main interfaces:

- **Control Signals:** A synchronous clock and two asynchronous reset signals. The reset is used to return internal state machines to initialized states.

- **Input Port:** This consists of three input signals, data, valid, and channel. The data is written to the core by asserting the valid signal, and providing the channel input all during the same clock cycle. The data for multiple channels must be provided on consecutive cycles, with a single gap of non-valid data until the next sample time. Figure 1–11 shows typical inputs data $a_v$, $a_c$, and $a_0$ for a 4-channel FIR filter with 10 cycles used to process the channels ($t_{12} - t_2$ is the time between successive data samples for channel 0).
**Output Port**: Output data is provided using the same data, valid and channel signals as the input. The data for each channel for each sample is written contiguously. Typical output data is shown in Figure 1–11. The interpolation rate determines the output data timing. For interpolation of two, there are twice as many valid cycles as on the input.

**Memory-Mapped Interface**: Access to the internal data coefficients is provided via a memory-mapped interface consisting of inputs address, write data, and write enable; and outputs read data and read valid. A separate bus clock may be used for this interface. A write cycle (time-steps 2 to 6) and a read cycle (time-steps 10 to 14) are shown in Figure 1–12.

**Figure 1–11.** Timing diagram for a Typical FIR Filter Interface

**Figure 1–12.** Memory-Mapped Registers Timing Diagram
The DSP Builder Advanced Blockset environment uses these interfaces to build up chains of filters without additional glue logic, even with sample rate changes. The tool builds the necessary bus decode logic to access all registers in the datapath.

**Design Semantics**

A design is represented in Simulink using the time-step representation of the system clock rate. This means that although the multi-rate system has multiple sample rates, this is not represented using the Simulink multiple sample-time features. This is a key advantage over other Simulink-based design systems that disable the hardware based on the sample time (wasting hardware that is inactive for much of the time).

**Fixed Point Representation**

All the signals in a DSP Builder Advanced Blockset design are specified using the built-in Simulink fixed point types. This makes it easy to debug your design, because the signals can be displayed as familiar floating point types.

A more subtle advantage of using fixed point types is that the extra information of binary point position can be preserved through hardware blocks, so that it is easy to perform rounding and shifting operations without having to manually track the interpretation of an integer value. This is important when you start to tune the performance of your model, because a fixed point type change propagates through your design, with all downstream calculations being automatically adjusted.

**Sample Rate and Clocks**

The sample rate is the rate at which real data is clocked through the system at any point. In a multi-rate environment, the sample rate may vary along the datapath.

The datapath components in the DSP Builder Advanced Blockset use a single clock rate. A separate bus clock is used by an external processor to read and write any primitive registers or shared memories and ModelIP block parameters through an Avalon® Memory-Mapped (Avalon-MM) interface.

**Figure 1–13. Bus and System Clocks**

![Diagram showing bus and system clocks](image-url)
The bus clock is usually not performance critical and can run at a lower clock rate. This permits easier timing closure in many cases. Running the clock as an integer fraction of the system clock rate ensures that there are no timing issues and that both clocks having co-incident edges. In many cases, this does not matter and the bus clock can have any frequency. The clock rates are set in the Signals block as described in “Signals Block” on page 1–4.

The memory-mapped input and output registers are cleared when the system is reset but the contents of the dual memory used by the FIR Filter, NCO, and ModelBus blocks are retained.

For information about the Avalon-MM Interface, refer to the Avalon Interface Specifications.

Model Simulation

A design is simulated in Simulink by starting simulation in the usual way. At the start of each simulation run, each ModelIP block is re-synthesized into an internal representation of the hardware components and written out as VHDL RTL.

The internal representation makes simulation fast, and ensures bit and cycle accuracy. This process is performed very quickly, and gives greatly increased productivity over traditional flows where you must configure your design in an external tool, import it into the simulator, and then run the simulation.

Generated Files

Table 1–3 lists the files that are generated for an Advanced Blockset design.

The files are created in a directory structure at the location specified in the Control block (which defaults to ..\rtl (relative to the working directory that contains the .mdl file). This directory structure uses the same hierarchy as the subsystems in your design with the files in lower level directories referenced from a corresponding file in its parent directory.

For example, there is a Quartus II IP (.qip) file in the <model name> subdirectory that references a .qip file in a lower level subdirectory that in turn references a .qip file in the next lower level and so on throughout your design hierarchy.

Any RAM in your design may have associated Intel format hexadecimal (.hex) files generated to initialize the RAM. Usually, the RAM are part of ModelIP blocks (such as the CIC, NCO, or FIR Compiler) which are expandable to lower level functions such as RAM or ADDSub. A .qip file for a subsystem containing such blocks has the initializing .hex files referenced from the .qip file.

Table 1–3. Generated Files for the DDC Demonstration Design

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtl directory</td>
<td></td>
</tr>
<tr>
<td>&lt;model name&gt;.xml</td>
<td>An XML file that describes the attributes of your model.</td>
</tr>
<tr>
<td>&lt;model name&gt;_entity.xml</td>
<td>An XML file that describes the boundaries of the system (used by Signal Compiler in designs that combine blocks from the standard and advanced blocksets).</td>
</tr>
</tbody>
</table>
Table 1–3. Generated Files for the DDC Demonstration Design

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtl&lt;model name&gt; subdirectory</td>
<td>Running this script compiles your design, loads it into ModelSim, and runs it for the same simulation time that your model ran for in Simulink.</td>
</tr>
<tr>
<td>Control.do</td>
<td>This script loads the VHDL files in this subdirectory and in the subsystem hierarchy below it into the ModelSim project.</td>
</tr>
<tr>
<td>Control.wav.do</td>
<td>This script loads the signals from your design into the ModelSim Wave window.</td>
</tr>
<tr>
<td>&lt;block name&gt;.xml</td>
<td>An XML file containing information about each block in the advanced blockset which is translated into HTML on demand for display in the MATLAB Help viewer.</td>
</tr>
<tr>
<td>&lt;model name&gt;.vhd</td>
<td>This is the top-level testbench file. It may contain non-synthesizable blocks, and may also contain empty black boxes for Simulink blocks that are not fully supported.</td>
</tr>
<tr>
<td>&lt;model name&gt;.add.do</td>
<td>This script compiles the VHDL files in this subdirectory and in the subsystem hierarchy below it into the ModelSim project.</td>
</tr>
<tr>
<td>&lt;model name&gt;.compile.do</td>
<td>This script compiles the VHDL files in this subdirectory and in the subsystem hierarchy below it into the Quartus II project.</td>
</tr>
<tr>
<td>&lt;model name&gt;.add.tcl</td>
<td>This file contains information about all the files required to process your model in the Quartus II software. The file includes a reference to any .qip file in the next level of the subsystem hierarchy.</td>
</tr>
<tr>
<td>&lt;model name&gt;_&lt;block name&gt;.vhd</td>
<td>A VHDL file is generated for each component in your model.</td>
</tr>
<tr>
<td>&lt;model name&gt;_&lt;subsystem&gt;_entity.xml</td>
<td>An XML file that describes the boundaries of a subsystem as a block box (used by Signal Compiler in designs that combine blocks from the standard and advanced blocksets).</td>
</tr>
<tr>
<td>&lt;subsystem&gt;.xml</td>
<td>An XML file that describes the attributes of a subsystem.</td>
</tr>
<tr>
<td>*.stm</td>
<td>Stimulus files.</td>
</tr>
<tr>
<td>safe_path.vhd</td>
<td>Helper function that is referenced in the .qip and .add.tcl files to ensure that pathnames are read correctly in the Quartus II software.</td>
</tr>
<tr>
<td>safe_path_msim.vhd</td>
<td>Helper function that ensures a pathname is read correctly in ModelSim.</td>
</tr>
<tr>
<td>rtl&lt;model name&gt;_subsystem subdirectories</td>
<td>There are separate subdirectories for each hierarchical level in your design. These include additional .xml, .vhdl, .qip, and .stm files describing the blocks in each level. There are also additional .do, .tcl, and .add.tcl files that are automatically called from the corresponding files in the top-level of your model.</td>
</tr>
<tr>
<td>&lt;subsystem&gt;_atb.do</td>
<td>Script that loads the subsystem automatic test bench into ModelSim.</td>
</tr>
<tr>
<td>&lt;subsystem&gt;_atb.wav.do</td>
<td>Script that loads signals for the subsystem automatic test bench into ModelSim.</td>
</tr>
<tr>
<td>&lt;subsystem&gt;_block/*.hex</td>
<td>These are Intel format Hex files that are used to initialize the RAM in your design for either simulation or synthesis.</td>
</tr>
<tr>
<td>&lt;subsystem&gt;.sdc</td>
<td>Design constraint file for TimeQuest support.</td>
</tr>
<tr>
<td>&lt;subsystem&gt;.tcl</td>
<td>This Tcl script exists only in the subsystem that contains a Device block. You can use this script to setup the Quartus II project.</td>
</tr>
<tr>
<td>&lt;subsystem&gt;_hw.tcl</td>
<td>A TCL script that is used to load the generated hardware into SOPC Builder.</td>
</tr>
</tbody>
</table>

For an example of the files generated for a typical design, refer to “Exploring the Generated Files” on page 5–22 in the System Tutorial.
Creating a Quartus II Project

A Quartus II project is created in the design directory that contains the .mdl file when you click on the Run Quartus II block.

The Quartus II project file (.qpf), Quartus II settings file (.qsf), and .qip files have the same name as the subsystem in your model that contains the Device block. For example, the files DDCChip.qpf, DDCChip.qsf, and DDCChip.qip are created for the demo_ddc design.

These files contain all required references to the files in the hardware destination directory specified by the Control block that are generated when you run a Simulink simulation. The project is automatically loaded into the Quartus II software.

You can then compile your design by clicking Start Compilation on the Processing menu. The project is compiled using the .tcl scripts in the hardware destination directory.

The .qip file references all the files required by the project. You can use this file to archive the project using the Archive Project command in the Quartus II software.

For information about archiving projects, refer to the Quartus II Help.

Adding a DSP Builder Advanced Blockset Design to an Existing Quartus II Project

You can add an advanced blockset design to an existing Quartus II project by sourcing the .add.tcl file in the subsystem that contains the Device block. Alternatively, you can add a reference to the .qip file in this subsystem from the .qip file for the top-level Quartus II project. For information about the .add.tcl, .qip and other generated files, refer to Table 1–3 on page 1–20.

For information about using Tcl files in the Quartus II software, refer to the Quartus II Help.

Adding Advanced Blockset Components to SOPC Builder

You can add a DSP Builder component to SOPC Builder by adding a directory that contains generated hardware to the IP Search Path in the SOPC Builder Options dialog box.

For an example of this procedure, refer to “Instantiating the Design in SOPC Builder” on page 3–14.

Comparison with RTL

You can compare the Simulink results with the generated RTL by using an automatic testbench that is created for each ModelIP and primitive block, or by simulating your model in an interactive ModelSim session.

To use either of these flows, you must turn on Generate hardware and Create automatic testbenches in the Control block parameters and the ModelSim executable (vsim.exe) must be available on your path.
Chapter 1: About the DSP Builder Advanced Blockset

Comparison with RTL

Automatic Testbench Flow

Each ModelIP block, and each synthesized primitive block writes out test vectors to a stimulus file (*.stm) during a Simulink simulation run. An RTL testbench is also created for each separate entity in your design (that is, for each ModelIP block and primitive subsystem. These testbenches replay the test vectors through the generated RTL, and compares the output from the RTL to the output from the Simulink model. If there is a mismatch at any cycle, the simulation is stopped and an error issued. By using these automatic testbenches, it is easy to verify the correct behavior of the synthesis engine.

The automatic testbench flow uses a stimulus and capture method and is therefore not restricted to a limited set of source blocks. The Simulink simulation stores data at the inputs and outputs of each entity during simulation, then the testbench for each entity uses this data as a stimulus and compares the ModelSim output to the Simulink captured output. A result is returned that indicates whether or not the outputs match when the valid signal is high.

Loading in ModelSim

You can load an automatic testbench by clicking Execute Macro on the Tools menu in ModelSim and selecting the required .do file. For example, to run the automatic testbench for the NCOSubsystem subsystem in the NCO demonstration design, select the demo_nco_NCOSubsystem_NCO_atb.do file.

Running from a Command

You can also run an automatic testbench from the MATLAB command line using the command run_modelsim_atb. This command has the syntax:

```matlab
run_modelsim_atb('model', 'entity', ['rtl_path']);
```

where

- `model` = model name (without extension, in single quotes)
- `entity` = entity to test (the name of a primitive subsystem or a ModelIP block, in single quotes)
- `rtl_path` = optional path to the generated RTL (in single quotes, if not specified the path is read from the Control block in your model)

For example:

```matlab
run_modelsim_atb('demo_fft16_radix2', 'FFTChip');
```

The return values are in the format [pass, status, result] where:

- `pass` = 1 or 0
- `status` = should be 0
- `result` = should be a string such as:

  "# ** Note: Arrived at end of stimulus data on clk <clock name>"

An output file named using the full path to the component under test is written in the the working directory. A new file is created with an automatically incremented suffix each time the testbench is run. For example:

```plaintext
demo_fft_radix2_DUT_FFTChip_atb.6.out
```
This output file includes the ModelSim transcript and can be useful for debugging if any errors are encountered.

Typical error messages have the form:

```bash
# ** Error (vcom-13) Recompile <path>altera_mf.altera_mf_components because <path>iee.std_logic_1164 has changed.
...
# ** Error: <path>mdl_name_system_subsystem_component.vhd(30): (vcom-1195) Cannot find expanded name: 'altera_mf.altera_mf_components'.
...
# ** Error: <path>vcom failed.
...
# At least one module failed to compile, not starting simulation.
```

These errors may occur when a ModelSim pre-compiled model is out of date, but not automatically recompiled.

A similar problem may occur after making design changes when ModelSim has cached a previously compiled model for a component and does not detect when it changes.

In either of these cases, delete the rtl directory, re-simulate your model and run the run_modelsim_atb command again.

### Running All the Automatic Testbenches in a Design

You can automatically run all the individual automatic testbenches in a design by using the command run_all_atbs. This command must be run from the same directory that contains the mdl file.

This command has the syntax:

```bash
run_all_atbs('model', [runSimulation], [runFit]);
```

where

- `model` = model name (without extension, in single quotes)
- `runSimulation` = optional flag which runs a simulation when specified (if not specified a simulation must have been run previously to generate the required files)
- `runFit` = optional flag which runs the Quartus II Fitter when specified

For example:

```bash
run_all_atbs('demo_agc');
run_all_atbs('demo_agc', true);
run_all_atbs('demo_agc', false, true);
run_all_atbs('demo_agc', true, true);
```

The return value is 1 if all tests are successful or 0 if any tests fail. The output is written to the MATLAB command window. If you choose to run the Quartus II Fitter, the command also reports whether the target fMAX was achieved. For example:

```bash
Met FMax Requirement (FMax(291.04) >= Required(200))
```
A summary is also written to a file `results.txt` in the current working directory. For example:

```
Starting demo_agc Tests at 2009-01-23 14:58:48

demo_agc: demo_agc/AGC_Chip/AGC hardware matches simulation (atb#1): PASSED

demo_agc: Quartus II compilation was successful. (Directory=../quartus_demo_agc_AGC_Chip_2): PASSED

demo_agc: Met FMax Requirement (FMax(291.04) >= Required(200)): PASSED

Finished demo_agc Tests at 2009-01-23 15:01:59 (3 Tests, 3 Passes, 0 Skipped, 0 Failed (fmax), 0 Failed (non-fmax))
```

Note that you can get a list of the blocks in a model that have automatic test benches by running the command:

```
getBlocksWithATBs('model')
```

You must have run a simulation on the model before running this command (or set the `runSimulation` argument to true). If directed to simulate, the script deletes any existing RTL directory before simulating. If it cannot delete the directory, for example if the files are in use, the script fails with an appropriate error message.

### Run ModelSim Flow

The complete Simulink model can be compared with hardware using the ModelSim simulator by double-clicking on the `Run ModelSim` block.

This comparison uses the same stimulus capture and comparison method as the automatic testbenches.

Stimulus files are captured on the device level inputs and Simulink output data recorded on the device level outputs. A ModelSim testbench is created that contains the HDL generated for the device fed by the captured inputs and. The Simulink outputs are compared to the ModelSim simulation outputs in an HDL testbench process, and any mismatches are reported and stop ModelSim simulation.

### Design Space Exploration

It is possible to write scripts that directly change parameters (such as the hardware destination directory) on the `Control` and `Signals` blocks.

For example, in a script where you have been passed the model name (without `.mdl` extension) as `model` you could use:

```
%% Load the model
load_system(model);

%% Get the Signals block
signals = find_system(model, 'type', 'block', 'MaskType', 'DSP Builder Advanced Blockset Signals Block');
if (isempty(signals))
    error('The design must contain a Signals Block. ');
end;

%% Get the Controls block
```
control = find_system(model, 'type', 'block', 'MaskType', 'DSP Builder Advanced Blockset Control Block');
if (isempty(control))
    error('The design must contain a Control Block. ');
end;

%% Example: set the RTL destination directory
dest_dir = ['./rtl' num2str(freq)];
set_param(control{1},'destination',dest_dir);

Similarly you can get and set other parameters. For example, on the Signals Block you can set the target clock frequency:

fmax_freq = 300.0;
set_param(signals{1},'freq', fmax_freq);

You can also change threshold values which are parameters on the Control block:

- distRamThresholdBits
- hardMultiplierThresholdLuts
- mlabThresholdBits
- ramThresholdBits

You could loop over changing these values, change the destination directory, run the Quartus II software each time and perform design space exploration. For example:

%% Run a simulation; which also does the RTL generation.
t = sim(model);

%% Then run the Quartus II compilation flow.
[success, details] = run_hw_compilation(<model>, './')

%% where details is a struct containing resource and timing information

details.Logic,
details.Comb_Aluts,
details.Mem_Aluts,
details.Regs,
details.ALM,
details.DSP_18bit,
details.Mem_Bits,
details.M9K,
details.M144K,
details.IO,
details.FMax,
details.Slack,
details.Required,
details.FMax_unres,
details.timingpath,
details.dir,
details.command,
details.pwd

such that >> disp(details) gives output something like:

Logic: 4915
Comb_Aluts: 3213
Mem_Aluts: 377
  Regs: 4725
  ALM: 2952
DSP_18bit: 68
Mem_Bits: 719278
  M9K: 97
  M144K: 0
  IO: 116
FMax: 220.1700
Slack: 0.4581
Required: 200
FMax_unres: 220.1700
timingpath: [1x4146 char]
  dir: '../quartus_demo_ifft_4096_natural_for_SPR_FFT_4K_n_2'
  command: [1x266 char]
  pwd: 'D:\test\script'

The Timing Report is contained in the timingpath variable and can be displayed by disp(details.timingpath). Unused resources may appear as -1, rather than 0.

Note that you must have previously executed load_system before commands such as find_system and run_hw_compilation can work.

A useful set of commands to generate RTL, compile in the Quartus II software and return the details is:

load_system(<model>);
sim(<model>);
[success, details] = run_hw_compilation(<model>, './')

Interoperability with the Standard Blockset

You can use the advanced blockset in a design flow that includes blocks from the DSP Builder standard blockset.

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

The best way to learn about the advanced blockset is to examine the demonstration designs listed in Chapter 2 and perform the tutorials described in Chapters 3, 4, and 5:

- The **ModelIP Tutorial (Chapter 3)** shows how to create a customized numerically controlled oscillator using the ModelIP NCO block. The tutorial includes procedures for performing RTL simulation in ModelSim, compiling your design in the Quartus II software and instantiating your design as a subsystem in the SOPC Builder. This basic design flow is illustrated for other ModelIP blocks by the demonstration designs in the Filters and Waveform Synthesis sections of the DSP Builder Advanced Blockset Demos. These sections also include several more complex designs built using ModelIP blocks.

- The **Primitive Library Tutorial (Chapter 4)** shows how you can build a simple design using blocks from the ModelPrim library. Refer to the Primitive Blocks section of the DSP Builder Advanced Blockset Demos for examples of more complex designs built using primitive blocks.

- The **System Tutorial (Chapter 5)** shows how you can plug blocks together to create a digital down converter (DDC). Refer to the Platforms section of the DSP Builder Advanced Blockset Demos for examples of more complex system designs.
The Altera DSP Builder Advanced Blockset provides a variety of example designs, which you can use to learn from or as a starting point for your own design. This section describes the available designs.

To view all the example designs, type `demo` at the MATLAB command prompt. The Demos tab opens in the Help window displaying a list of the available example designs.

You can click DSP Builder Advanced Blockset in the Help window to expand the list as shown in Figure 2–1 and click on an entry to display a Help page for each example design.

![Figure 2–1. DSP Builder Demonstration Designs](image)

All the demonstration designs have the same basic structure: a top-level testbench containing an instantiated functional block, which represents the hardware design. The testbench typically includes Simulink source blocks that generate the stimulus signals and sink blocks that display simulation results. You can use other Simulink blocks to define the testbench logic.
Opening a Demonstration Design

You can display the model corresponding to each demonstration design by clicking Open this model in the Help window. You can also open the models by typing a command of the following form in the MATLAB window:

```
open_system('demo_nco');
```

or simply:

```
demo_nco
```

The demonstration models are grouped under the following folders:

- **Platforms.** This folder contains several reference designs that illustrate how you can implement a digital down converter (DDC) or digital up converter (DUC) suitable for use in a radio basestation. These provide a starting point to build your own filter chain that meets your exact needs.
  - 16-Channel DDC (demo_ddc)
  - 16-Channel DUC (demo_duc)
  - 2-Channel DUC (demo_AD9856)
  - 2-Antenna DUC for WiMAX (demo_wimax_duc)

- **Filters.** This folder contains examples of cascaded integrator-comb (CIC) and finite impulse response (FIR) filters:
  - Decimating CIC Filter (demo_dcic)
  - Interpolating CIC Filter (demo-icic)
Chapter 2: Example Designs
Opening a Demonstration Design

- Single Rate FIR Filter (demo_firs)
- Decimating FIR Filter (demo_fird)
- Interpolating FIR Filter (demo_firi)
- Fractional Rate FIR Filter (demo_firf)
- Half Band FIR Filter (demo_firih)
- Root Raised Cosine FIR Filter (demo_fir_rrc)
- Fractional FIR Filter Chain (demo_fir_fractional)
- Super-Sample FIR Filter (demo_ssfiri)
- Filter Chain with Forward Flow Control (demo_filters_flow_control)

**Waveform Synthesis.** This folder contains examples which synthesize waveforms using a numerically controlled oscillator (NCO) or direct digital synthesis (DDS):

- NCO (demo_nco)
- Real Mixer (demo_mix)
- Complex Mixer (demo_complex_mixer)
- Four Channel, Two Banks NCO (demo_mc_nco_2banks_mem_interface)
- Four Channel, Four Banks NCO (demo_mc_nco_4banks_mem_interface)
- Four Channel, Eight Banks, Two Wires NCO (demo_mc_nco_8banks_2wires)
- Four Channel, 16 Banks NCO (demo_mc_nco_16banks)
- Four Channel, 64 Banks NCO (demo_mc_nco_64banks)

**Primitive Blocks (ModelPrim).** This folder contains various example designs built with primitive blocks from the ModelPrim library:

- Hello World (helloWorld)
- Fibonacci Series (demo_fibonacci)
- Automatic Gain Control (demo_agc)
- Multi-Channel IIR Filter (demo_iir)
- 8×8 Inverse Discrete Cosine Transform (demo_idct8x8)
- Quadrature Amplitude Modulation (demo_QAM256)
- Radix 2 Streaming FFT (demo_fft16_radix2)
- Radix 4 Streaming FFT (demo_fft256_radix4)
- 4K FFT (demo_fft_4096_br)
- 8K FFT (demo_fft_8192_br)
- 4K IFFT (demo_ifft_4096_natural)
- 8K IFFT (demo_ifft_8192_natural)
- Test CORDIC Functions Using Primitive Blocks (demo_cordic_primitives)
- Test CORDIC Functions Using the CORDIC Block (demo_cordic_lib_block)
■ **Host Interface (ModelBus).** This folder contains an example showing how you can control memory-mapped registers:
  - Memory-Mapped Registers (demo_regs)

■ **Base Blocks.** This folder contains an example using the Scale block:
  - Scale (demo_scale)

■ **Tutorials.** This folder accesses the designs used in the tutorial examples:
  - ModelIP (demo_nco)
  - System (demo_ddc)
  - Primitive Library (demo_fibonacci)

■ **Reference Designs.** This folder accesses two groups of reference designs that illustrate the design of digital down converter (DDC) and digital up converter (DUC) systems for digital intermediate frequency (IF) processing.

The first group of models implement IF modem designs compatible with the wideband Code Division Multiple Access (W-CDMA) standard:

  - 4-Carrier, 2-Antenna W-CDMA DDC (wcdma_multichannel_ddc_mixer)
  - 1-Carrier, 2-Antenna W-CDMA DDC (wcdma_picocell_ddc_mixer)
  - 4-Carrier, 2-Antenna W-CDMA DUC (wcdma_multichannel_duc_mixer)
  - 1-Carrier, 2-Antenna W-CDMA DDC (wcdma_picocell_duc_mixer)
  - 4-Carrier, 2-Antenna High-Speed W-CDMA DUC at 368.64 MHz with Total Rate Change 32 (wcdma_multichannel_duc_mixer_96x_32R)
  - 4-Carrier, 2-Antenna High-Speed W-CDMA DUC at 368.64 MHz with Total Rate Change 48 (wcdma_multichannel_duc_mixer_96x_48R)
  - 4-Carrier, 2-Antenna High-Speed W-CDMA DUC at 307.2 MHz with Total Rate Change 40 (wcdma_multichannel_duc_mixer_80x_40R)

The second group implements IF modem designs compatible with the Worldwide Interoperability for Microwave Access (WiMAX) standard. Separate models are provided for one and two antenna receivers and transmitters:

  - 1-Antenna WiMAX DDC (wimax_ddc_1rx)
  - 2-Antenna WiMAX DDC (wimax_ddc_2rx_iqq)
  - 1-Antenna WiMAX DUC (wimax_duc_1tx)
  - 2-Antenna WiMAX DUC (wimax_ddc_2tx_iqq)

### Copying a Demonstration Design

If you want to make a copy of a demonstration design, ensure that you also copy any corresponding `setup_<name>.m` file that has the same name as the `<name>.mdl` file.

You may also want to change the location of the generated files that is specified in the Control block for each design (which is set to the relative path `../rtl` by default).
Running a Demonstration Design

You can run a demonstration design by clicking on in the model window or by typing a command of the following form in the MATLAB window:

\[
\text{sim('demo_nco', 20000*SampleTime)};
\]

After you have run a simulation, the Help page for each DSP Builder Advanced Blockset block (displayed when you click Help on the popup menu for the blocks) is replaced by a page showing information about the actual block usage.

For example, Figure 2–2 on page 2–5 shows the updated Help page for an Interpolating CIC block.

You can return to the normal Help page by clicking on the Help on core functionality of <block name> link.

Figure 2–2. Updated Help Page for an Interpolating CIC Block
Demonstration Designs

The available demonstration designs are briefly described in the following sections of this user guide. More information is provided as annotation text in the models.

16-Channel DDC

This model shows how to build a 16-channel digital down converter as found in modern radio systems using ModelIP and ModelBus blocks.

A decimating filter chain is presented. Decimating CIC and FIR filters are used to down convert eight complex carriers (16 real channels) from 61.44 MHz. The total decimation rate is 64. A real mixer and NCO are used to isolate the eight carriers. The testbench isolates two channels of data from the time-division multiplexed (TDM) signals using a channel viewer.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus a Channel Viewer block that is used to deserialize the output bus. An Edit Params block allows easy access to the setup variables in the setup_demo_ddc.m script.

The DDCChip subsystem includes Device, Decimating FIR, Decimating CIC, Mixer, NCO, Scale, Register Bit, and Register Field blocks.

The example file is named demo_ddc.mdl.

This demonstration design uses the Simulink Signal Processing Blockset.

16-Channel DUC

This model shows how to build a 16-channel digital up converter as found in modern radio systems using ModelIP, ModelBus, and ModelPrim blocks.

An interpolating filter chain is presented. Interpolating CIC and FIR filters are used to up convert eight complex channels (16 real channels). The total interpolation rate is 50. Several primitive subsystems are integrated into the data path. This example shows how you can integrate ModelIP blocks with primitive subsystems:

- There is a programmable Gain subsystem at the beginning of the data path. This subsystem shows how you can use processor-visible register blocks to control a datapath element.
- The Sync subsystem is a primitive subsystem that shows how to manage two data streams coming together and being synchronized. The data from the NCOs are written to a memory using the channel as an address: the data stream using its channel signals to read out the NCO signals. This resynchronizes the data correctly. (Alternatively, you could simply delay the NCO value by the correct number of cycles to ensure that the NCO and channel data arrive at the Mixer on the same cycle).

Extensive use is made of Simulink multiplexer and demultiplexer blocks to manage vector signals.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus a Channel Viewer block that is used to deserialize the output bus. An Edit Params block allows easy access to the setup variables in the setup_demo_duc.m script.
The **DUCChip** subsystem includes a **Device block** and a lower level **DUC16** subsystem.

The **DUC16** subsystem includes **Interpolating FIR**, **Interpolating CIC**, **Complex Mixer**, **NCO**, and **Scale blocks**.

It also includes lower level **Gain**, **Sync**, and **CarrierSum** subsystems which make use of other ModelBus and ModelPrim blocks including **Add SLoad**, **And**, **Bit Extract**, **Channel In**, **Channel Out**, **Compare Equality**, **Constant**, **Sample Delay**, **Dual Memory**, **Multiply**, **Multiplexer**, **Not**, **Or**, **Register Bit**, **Register Field** blocks, and **Synthesis Information blocks**.

The example file is named **demo_duc.mdl**.

This demonstration design uses the Simulink Signal Processing Blockset.

### 2-Channel DUC

This model shows how to build a 2-channel digital up converter as found in an ASSP chip.

Interpolating CIC and FIR filters are used to up convert a single complex channel (2 real channels). A **NCO** and **Mixer** subsystem are used to combine the complex input channels into a single output channel.

This example shows how quick and easy it is to emulate the contents of an existing data path. A primitive block is used to implement the mixer in this design as the data rate is low enough to save resource using a time shared hardware technique.

The top-level testbench includes **Control**, **Signals**, **Run ModelSim**, and **Run Quartus II** blocks, plus a **Channel Viewer** block that is used to deserialize the output bus. An **Edit Params** block allows easy access to the setup variables in the **setup_demo_AD9856.m** script.

The **AN9856** subsystem includes a **Device block** and a lower level **DUCIQ** subsystem.

The **DUCIQ** subsystem includes **Constant**, **Interpolating FIR**, **Single Rate FIR**, **Interpolating CIC**, **NCO**, **Scale blocks**, and a lower level **Mixer** subsystem.

The **Mixer** subsystem includes **Channel In**, **Channel Out**, **Multiply**, **Constant**, **Bit Extract**, **Compare Equality**, **And**, **Delay**, **Subtract**, and **Synthesis Information blocks**.

The example file is named **demo_AD9856.mdl**.

This demonstration design uses the Simulink Signal Processing Blockset.

### 2-Antenna DUC for WiMAX

This model shows how to build a 2-antenna digital up converter to meet a WiMAX specification.

The top-level testbench includes **Control**, **Signals**, **Run ModelSim**, and **Run Quartus II** blocks, plus a **Channel Viewer** block that is used to deserialize the output bus.

The **DUCChip** subsystem includes a **Device block** and a lower level **DUC2Antenna** subsystem.
The DUC2Antenna subsystem includes Interpolating FIR, Single Rate FIR, Constant, Complex Mixer, NCO, and Scale blocks.

The example file is named `demo_wimax_duc.mdl`.

This demonstration design uses the Simulink Signal Processing Blockset.

### Decimating CIC Filter

This model implements a decimating CIC filter.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus Channel Viewer block that are used to deserialize the output buses. An Edit Params block allows easy access to the setup variables in the `setup_demo_d cic.m` script.

The CICSystem subsystem includes the Device and Decimating CIC blocks.

The example file is named `demo_d cic.mdl`.

This demonstration design uses the Simulink Signal Processing Blockset.

### Interpolating CIC Filter

This model implements an interpolating CIC filter.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus Channel Viewer block that are used to deserialize the output buses. An Edit Params block allows easy access to the setup variables in the `setup_demo_i cic.m` script.

The FilterSystem subsystem includes the Device and Interpolating CIC blocks.

The example file is named `demo_i cic.mdl`.

This demonstration design uses the Simulink Signal Processing Blockset.

### Single Rate FIR Filter

This model uses the Single Rate FIR block to build a 16-channel single rate 49-tap FIR filter with a target system clock frequency of 360 MHz.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus Channel Viewer block that are used to deserialize the output buses. An Edit Params block allows easy access to the setup variables in the `setup_demo_firs.m` script.

The FilterSystem subsystem includes the Device and Single Rate FIR blocks.

The example file is named `demo_firs.mdl`.

This demonstration design uses the Simulink Signal Processing Blockset.
Chapter 2: Example Designs

Demonstration Designs

Decimating FIR Filter
This model uses the Decimating FIR block to build a 20-channel decimate by 5, 49-tap FIR filter with a target system clock frequency of 240 MHz.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus Channel Viewer block that are used to deserialize the output buses. An Edit Params block allows easy access to the setup variables in the setup_demo_fird.m script.

The FilterSystem subsystem includes the Device and Decimating FIR blocks.

The example file is named demo_fird.mdl.

Interpolating FIR Filter
This model uses the Interpolating FIR block to build a 16-channel interpolate by 2, symmetrical, 49-tap FIR filter with a target system clock frequency of 360 MHz.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus Channel Viewer block that are used to deserialize the output buses. An Edit Params block allows easy access to the setup variables in the setup_demo_firi.m script.

The FilterSystem subsystem includes the Device and Interpolating FIR blocks.

The example file is named demo_firi.mdl.

Fractional Rate FIR Filter
This model is implements a fractional rate FIR filter.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus Channel Viewer block that are used to deserialize the output buses. An Edit Params block allows easy access to the setup variables in the setup_demo_firf.m script.

The FilterSystem subsystem includes the Device and Fractional Rate FIR blocks.

The example file is named demo_firf.mdl.

Half Band FIR Filter
This model is implements a half band interpolating FIR filter.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus Channel Viewer block that are used to deserialize the output buses. An Edit Params block allows easy access to the setup variables in the setup_demo_firih.m script.

This demonstration design uses the Simulink Signal Processing Blockset.
The FilterSystem subsystem includes the Device block and two separate Interpolating FIR blocks for the regular and interpolating filters.

The example file is named `demo_firih.mdl`.

This demonstration design uses the Simulink Signal Processing Blockset.

### Root Raised Cosine FIR Filter

This model uses the Decimating FIR block to build a 4-channel decimate by 5, 199-tap root raised cosine filter with a target system clock frequency of 304 MHz.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus Channel Viewer block that are used to deserialize the output buses. An Edit Params block allows easy access to the setup variables in the `setup_demo_fir_rrc.m` script.

The FilterSystem subsystem includes the Device and Decimating FIR blocks. The example file is named `demo_fir_rrc.mdl`.

This demonstration design uses the Simulink Signal Processing Blockset.

### Fractional FIR Filter Chain

This model uses a chain of Interpolating FIR and Decimating FIR blocks to build a 16-channel fractional rate filter with a target system clock frequency of 360 MHz.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus Channel Viewer block that are used to deserialize the output buses. An Edit Params block allows easy access to the setup variables in the `setup_demo_fir_fractional.m` script.

The FilterSystem subsystem includes Channel Viewer, Decimating FIR, Interpolating FIR, and Scale blocks. The example file is named `demo_fir_fractional.mdl`.

This demonstration design uses the Simulink Signal Processing Blockset.

### Super-Sample FIR Filter

This model uses the Interpolating FIR block to build a super-sample filter with a target system clock frequency of 200 MHz. The Interpolating FIR in this design can support an input sample rate above the clock rate for both symmetric and asymmetric cases.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks. An Edit Params block allows easy access to the setup variables in the `setup_demo_ssfiri.m` script.

The FilterSystem subsystem includes the Interpolating FIR block. The example file is named `demo_ssfiri.mdl`.

This demonstration design uses the Simulink Signal Processing Blockset.
**Filter Chain with Forward Flow Control**

This model builds a filter chain with forward flow control.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus Channel Viewer block that are used to deserialize the output buses. An Edit Params block allows easy access to the setup variables in the `setup_demo_filters_flow_control.m` script.

The FilterSystem subsystem includes Fractional Rate FIR, Interpolating FIR, Interpolating CIC, Constant and Scale blocks.

The example file is named `demo_filters_flow_control.mdl`.

This demonstration design uses the Simulink Signal Processing Blockset.

**NCO**

This model implements a numerically controlled oscillator (NCO) using the NCO block from the Waveform Synthesis library. The results are compared with a Simulink double precision sine/cosine wave.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus Channel Viewer blocks that are used to deserialize the output buses. An Edit Params block allows easy access to the setup variables in the `setup_demo_nco.m` script.

The NCOSubSystem subsystem includes the Device and NCO blocks.

The example file is named `demo_nco.mdl`.

This demonstration design uses the Simulink Signal Processing Blockset.

**Real Mixer**

This model shows how to mix non-complex signals.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus Channel Viewer block that are used to deserialize the output buses. An Edit Params block allows easy access to the setup variables in the `setup_demo_mix.m` script.

The MixerSystem subsystem includes the Device and Mixer blocks.

The example file is named `demo_mix.mdl`.

This demonstration design uses the Simulink Signal Processing Blockset.
### Complex Mixer

This model shows how to mix complex signals.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus Channel Viewer block that are used to deserialize the output buses. An Edit Params block allows easy access to the setup variables in the setup_demo_complex_mixer.m script.

The FilterSystem subsystem includes the Device and Complex Mixer blocks.

The example file is named demo_complex_mixer.mdl.

This demonstration design uses the Simulink Signal Processing Blockset.

### Four Channel, Two Banks NCO

This model implements a numerically controlled oscillator (NCO) with four channels and two banks. This design demonstrates frequency-hopping using the NCO block to generate four channels of sinusoidal waves that can be switched from one set (bank) of frequencies to another.

The phase increment values are set directly into the NCO Parameter dialog box as a 2 (rows) × 4 (columns) matrix. The input for the bank index is set up so that it alternates between the two predefined banks with each one lasting 2000 steps.

A Bus Stimulus block is used to setup an Avalon-MM interface that writes into the phase increment memory registers and shows how you can use the Avalon-MM interface to dynamically change the frequencies of the sinusoidal signals generated by the NCO at run time. A 16-bit memory interface is used in this design (as specified in the Control block) and a 24-bit the accumulator is used in the NCO block. Two registers are required for each phase increment value. With the base address of the phase increment memory map set to 1000 in this example design, the addresses [1000 1001 1002 1003 1012 1013 1014 1015] are used to write to the phase increment memory registers of channels 1 and 2 in bank 1, and to the registers of channels 3 and 4 in bank 2. The write data is also made up of two parts with each part writing to one of the registers feeding the selected phase increment accumulators.

There are two banks of frequencies set up in this demonstration design with each bank being processed for 2000 steps before switching to the other. A new value should be written into the phase increment memory register for each bank to change the NCO output frequencies after 8000 steps during simulation. To avoid writing new values to the active bank, the write enable signals are configured as:

```
[zeros(1,7000) 1 1 1 1 zeros(1,2000) 1 1 1 1 zeros(1,8000)].'
```

This configuration ensures that a new phase increment value for bank 0 is written at 7000 steps when the NCO is processing bank 1; and a new phase increment value for bank 1 is written at 9000 steps when the NCO is processing bank 0.

There are four writes for each bank to write new values for channel 1 and 2 into bank 0, and new values for channel 3 and 4 into bank 1. Each new phase value needs two registers due to the size of the memory interface.
The spectrum scope shows that there are three peaks for a selected channel with the first two peaks representing the two banks and the third peak showing the frequency specified through the memory interface. The scope of the select channel shows the sinusoidal waves of the selected channel. You can zoom in to see that smooth and continuous sinusoidal signals are generated at the switching point. You can also see the frequency changes after 8000 steps where the phase increment value is altered through the memory interface.

The top-level testbench includes Control, Signals, Bus Stimulus, Run ModelSim, and Run Quartus II blocks, plus Channel Viewer blocks that are used to deserialize the output buses. An Edit Params block allows easy access to the setup variables in the `setup_demo_mc_nco_2banks_mem_interface.m` script.

The NCOSubSystem subsystem includes the Device and NCO blocks.

The example file is named `demo_mc_nco_2banks_mem_interface.mdl`.

This demonstration design uses the Simulink Signal Processing Blockset.

### Four Channel, Four Banks NCO

This model implements a numerically controlled oscillator (NCO) with four channels and four banks. This design is similar to the Four Channel, Two Banks NCO design apart from having four banks of frequencies defined for the phase increment values. You can see that there are five peaks on each spectrum plot, where the fifth peak shows the changes written through the memory interface.

A 32-bit memory interface is used with a 24-bit accumulator. Hence only one phase increment memory register is required for each phase increment value. This is reflected in the address and data setup on the Bus Stimulus block inside this design.

There are four banks of frequencies set up in this demonstration design with each bank being processed for 2000 steps before switching to the other. A new value should be written into the phase increment memory register for each bank to change the NCO output frequencies after 16000 steps during simulation. To avoid writing new values to the active bank, the write enable signals are configured as:

```
[zeros(1,15000) 1 zeros(1,2000) 1 zeros(1,2000) 1 zeros(1,2000) 1 zeros(1,8000)].'
```

This configuration ensures that a new phase increment value for bank 0 is written at 15000 steps when the NCO is processing bank 3; a new phase increment value for bank 1 is written at 17000 steps when the NCO is processing bank 0; a new phase increment value for bank 2 is written at 19000 steps when the NCO is processing bank 1; and a new phase increment value for bank 3 is written at 21000 steps when the NCO is processing bank 2.

There is one write for each bank to write a new value for channel 1 into bank 0; a new value for channel 2 into bank 1; a new value for channel 3 into bank 2; and a new value for channel 4 into bank 3. Each new phase value needs only one register due to the size of the memory interface.

The top-level testbench includes Control, Signals, Bus Stimulus, Run ModelSim, and Run Quartus II blocks, plus Channel Viewer blocks that are used to deserialize the output buses. An Edit Params block allows easy access to the setup variables in the `setup_demo_mc_nco_4banks_mem_interface.m` script.
The NCOSubSystem subsystem includes the Device and NCO blocks.

The example file is named `demo_mc_nco_4banks_mem_interface.mdl`.

This demonstration design uses the Simulink Signal Processing Blockset.

### Four Channel, Eight Banks, Two Wires NCO

This model implements a numerically controlled oscillator (NCO) with four channels and eight banks. This design is similar to the Four Channel, 16 Banks NCO design apart from having only eight banks of phase increment values (specified in the setup script for the workspace variable) feeding into the NCO. Furthermore, the sample time is changed so that the NCO requires two wires to output the four channels of the sinusoidal signals. Because there are two wires for the NCO output, each wire only contains two channels, hence the channel indicator is changed from 0 .. 3 to 0 .. 1.

You can inspect the eight peaks on the spectrum graph for each channel and see the smooth continuous sinusoidal waves on the scope display.

An additional subsystem (`Select_bank_out`) is used in this design to extract the NCO generated sinusoidal signal of a selected bank on a channel.

The extract data is output to the workspace and plotted through using the separate `demo_mc_nco_extracted_waves.mdl`, which demonstrates that the output of the select bank does represent a genuine sinusoidal wave. However, from the scope display, you can see that the sinusoidal wave is no longer smooth at the switching point. This is because of the different values of phase increment values used in between the selected banks. The `demo_mc_nco_extracted_waves.mdl` model can only be run after `demo_mc_nco_8banks_2wires.mdl` has been run.

The top-level testbench includes Control, Signals, Bus Stimulus, Run ModelSim, and Run Quartus II blocks, plus Channel Viewer blocks that are used to deserialize the output buses. An Edit Params block allows easy access to the setup variables in the `setup_demo_mc_nco_8banks_2wires.m` script.

The NCOSubSystem subsystem includes the Device and NCO blocks.

The `Select_bank_out` subsystem contains Constant, Compare Equality, and AND Gate blocks.

The example file is named `demo_mc_nco_8banks_2wires.mdl`.

This demonstration design uses the Simulink Signal Processing Blockset.

### Four Channel, 16 Banks NCO

This model implements a numerically controlled oscillator (NCO) with four channels and 16 banks. This design demonstrates frequency-hopping using the NCO block to generate 4 channels of sinusoidal waves which can be switched from one set (bank) of frequencies to another in the 16 predefined frequency sets.

A workspace variable `phaseIncr` is used to define the 16 (rows) × 4 (columns) matrix for the phase increment input with the phase increment values calculated inside the setup script.

The input for the bank index is set up so that it cycles from 0 to 15 with each bank lasting 1200 steps.
The spectrum display shows clearly that there are 16 peaks for the selected channel indicating indeed that there are 16 different frequencies generated for that channel. The scope of the selected channel shows the sinusoidal waves of the selected channel. You can zoom in to see that smooth and continuous sinusoidal signals are generated at the switching point.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus Channel Viewer blocks that are used to deserialize the output buses. An Edit Params block allows easy access to the setup variables in the setup_demo_mc_nco_16banks.m script.

The NCOSubSystem subsystem includes the Device and NCO blocks.

The example file is named demo_mc_nco_16banks.mdl.

This demonstration design uses the Simulink Signal Processing Blockset.

**Four Channel, 64 Banks NCO**

This model implements a numerically controlled oscillator (NCO) with four channels and 64 banks. This design demonstrates frequency-hopping using the NCO block to generate 4 channels of sinusoidal waves which can be switched from one set (bank) of frequencies to another in the 64 predefined frequency sets.

A workspace variable `phaseIncr` is used to define the 64 (rows) x 4 (columns) matrix for the phase increment input with the phase increment values calculated inside the setup script.

The input for the bank index is set up so that it cycles through banks 0, 5, 9, 15, 21, 28, 35, 43, 52, 61, and 63 with each bank lasting 1200 steps.

The spectrum display shows clearly that there are 11 peaks for the selected channel indicating indeed that there are 11 different frequencies generated for that channel. The scope of the selected channel shows the sinusoidal waves of the selected channel. You can zoom in to see that smooth and continuous sinusoidal signals are generated at the switching point.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus Channel Viewer blocks that are used to deserialize the output buses. An Edit Params block allows easy access to the setup variables in the setup_demo_mc_nco_64banks.m script.

The NCOSubSystem subsystem includes the Device and NCO blocks.

The example file is named demo_mc_nco_64banks.mdl.

This demonstration design uses the Simulink Signal Processing Blockset.

**Hello World**

This model is a very simple primitive design that outputs a simple text message stored in a look-up table.

An external input is used to enable a counter that addresses a lookup-table (LUT) that contains some text. The result is written to a MATLAB array and the contents can be examined using a `char(message)` command in the MATLAB command window.
No Channel In, Channel Out, General Purpose Input, or General Purpose Output blocks are used in this design. Simulink ports are used instead for simplicity although they prevent the automatic testbench flow from working.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks.

The Chip subsystem includes Device, Counter, Look-Up Table, and Synthesis Information blocks.

In this design, the top-level of the FPGA device (marked by the Device block) and the synthesizable primitive subsystem (marked by the Synthesis Information block) are at the same level.

The example file is named helloWorld.mdl.

**Fibonacci Series**

This model generates a Fibonacci sequence.

This design shows that even for circuitry with tight feedback loops and 120-bit adders, high data rates can be achieved by the pipelining algorithms.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks.

The Chip subsystem includes the Device block and a lower level FibSystem subsystem.

The FibSystem subsystem includes Channel In, Channel Out, Sample Delay, Add, Multiplexer, and Synthesis Information blocks.

In this design, the top-level of the FPGA device (marked by the Device block) and the synthesizable primitive subsystem (marked by the Synthesis Information block) are at different hierarchy levels.

The example file is named demo_fibonacci.mdl.

**Automatic Gain Control**

This model implements an automatic gain control.

This design shows a complex loop, with several sub-loops all being scheduled and pipelined without register insertion by the user. A lumped delay is spread around the circuit to satisfy timing whilst maintaining correctness. Processor visible registers are used to control the thresholds and gains.

In a complex algorithmic circuit such as this, the zero-latency blocks make it easy to follow a data value through the circuit and investigate the algorithm without having to mentally offset all the results by the pipelining delays.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks.
The AGC_Chip subsystem includes the Device block, a Register Field block and a lower level AGC subsystem.

The AGC subsystem includes Register Field, Channel In, Channel Out, Multiply, Sample Delay, Add, Subtract, Convert, Absolute Value, Compare Greater Than, Look-Up Table, Constant, Shared Memory, Shift, Bit Extract, Select, and Synthesis Information blocks.

The example file is named demo_agc.mdl.

**Multi-Channel IIR Filter**

This model implements a masked multi-channel infinite impulse response (IIR) filter using a masked subsystem built from primitive blocks.

This is an example of a primitive design with many feedback loops. The design is implemented in textbook fashion and all the pipelined delays in the circuit are implemented automatically. The multiple channels are used to provide more latency around the circuit to ensure a high clock frequency result. Because lumped delays are used, the design is easy to parameterize when changing the channel counts. This is shown by masking the subsystem, providing the benefits of a black-box IP block but with visibility when needed.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus Channel Viewer block that are used to deserialize the output buses.

The IIRCHip subsystem includes the Device block and a masked IIRSubsystem subsystem. The coefficients for the filter are set from \([b, a] = \text{ellip}(2, 1, 10, 0.3)\); in the callbacks for the masked subsystem. You can look under the mask to see the implementation details of the IIRSubsystem subsystem which includes Channel In, Channel Out, Sample Delay, Constant, Multiply, Add, Subtract, Convert, and Synthesis Information blocks.

The example file is named demo_iir.mdl.

**8×8 Inverse Discrete Cosine Transform**

This model uses the Chen-Wang algorithm to implement a fully pipelined 8×8 inverse discrete cosine transform (IDCT).

Separate subsystems are used to perform the row transformation (Row), corner turner (CornerTurn), and column transformation (Col) functions. Each separate subsystem is synthesized separately. In the Row and Col subsystems, there are additional levels of hierarchy for the different stages but because the SynthesisInfo block is at the row/column level, these subsystems are flattened prior to synthesis.

The CornerTurn turn block makes extensive use of Simulink Goto/From blocks to reduce the wiring complexity.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks.
The IDCTChip subsystem includes the Device block and a lower level IDCT subsystem.

The IDCT subsystem includes lower level subsystems, which are described using the Channel In, Channel Out, Constant, Bit Combine, Shift, Multiply, Add, Subtract, Bit Extract, Sample Delay, OR Gate, Not, Sequence, and Synthesis Information primitive blocks.

The example file is named demo_idct8x8.mdl.

**Quadrature Amplitude Modulation**

This model implements a simple quadrature amplitude modulation (QAM256) design with noise addition. Various Simulink blocks are used in the testbench.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks.

The QAM256Chip subsystem includes Add, General Purpose Input, General Purpose Output, Bit Extract, Look-Up Table, Bit Combine, and Synthesis Information blocks.

The example file is named demo_QAM256.mdl.

This demonstration design uses the Simulink Communications Blockset.

**Radix 2 Streaming FFT**

This model implements a complex, Radix-2, streaming, fast Fourier transform.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks. An Edit Params block allows easy access to the setup variables in the setup_demo_fft16_radix2.m script.

The FFT designs do not inherit bit width and scaling information. These values are specified by the Wordlength and FractionLength variables in the setup script which are set to 16 and 15 for this design.

The DUT subsystem includes the Device block and a lower level FFTChip subsystem.

The FFTChip subsystem uses the Complex Multiplier, Complex Sample Delay, and Butterfly I blocks from the FFT library. It also includes Channel In, Channel Out, Counter, Bit Extract, Look-Up Table, Convert, Sample Delay, Dual Memory, and Synthesis Information blocks.

There is also a BitReverse subsystem that is very similar to the more general purpose Bit Reverse Core masked subsystem in the FFT library but without the channel signal.

The example file is named demo_fft16_radix2.mdl.
Radix 4 Streaming FFT

This model implements a 256 point, single channel, radix 4, complex fast Fourier transform.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks. An Edit Params block allows easy access to the setup variables in the setup_demo_fft256_radix4.m script.

The FFT designs do not inherit bit width and scaling information. These values are specified by the Wordlength and FractionLength variables in the setup script which are set to 16 and 15 for this design.

The DUT subsystem includes the Device block and a lower level FFTChip subsystem. The FFTChip subsystem uses the Complex Multiplier, Complex Sample Delay, Butterfly I, and Butterfly II blocks from the FFT library. It also includes Channel In, Channel Out, Counter, Bit Extract, Look-Up Table, Sample Delay, Convert, Dual Memory, and Synthesis Information blocks. There is also a Bit Reverse subsystem that is very similar to the more general purpose Bit Reverse Core masked subsystem in the FFT library but without the channel signal.

The example file is named demo_fft256_radix4.mdl.

4K FFT

This model implements a 4,096 point, Radix \(2^2\) fast Fourier transform. This design accepts data in bit reversed order, but the data is in natural order at the output of the FFT. The design includes a bit-reversal block before the FFT algorithm, that makes the testbench easier to understand.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks. An Edit Params block allows easy access to the setup variables in the setup_demo_fft_4096_br.m script.

The FFT designs do not inherit bit width and scaling information. These values are specified by the Wordlength and FractionLength variables in the setup script which are set to 16 and 19 for this design. You can also set the maximum bit width by setting the MaxOut variable. For most applications, the maximum bit width is not needed and you can save resources by setting a threshold value for this variable. The default value of \(\text{inf}\) allows worst case bit growth.

The FFT_4K_br subsystem includes the Device block, a Bit Reverse Core block from the Filter library and a lower level FFT_4K_BR_Natural subsystem.

The FFT_4K_BR_Natural subsystem uses the Complex Multiplier, Complex Sample Delay, Butterfly I, Butterfly II, and Twiddle Generator blocks from the FFT library. It also includes Channel In, Channel Out, Counter, Bit Extract, Sequence, Sample Delay, Convert, Dual Memory, Constant, Compare Equality, OR Gate, Not Gate, Multiplexer, and Synthesis Information blocks.

The example file is named demo_fft_4096_br.mdl.
8K FFT

This model implements a 8,192 point, Radix $2^2$ fast Fourier transform. This design accepts data in bit reversed order, but the data is in natural order at the output of the FFT. The design includes a bit-reversal block before the FFT algorithm, that makes the testbench easier to understand.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks. An Edit Params block allows easy access to the setup variables in the setup_demo_fft_8192_br.m script.

The FFT designs do not inherit bit width and scaling information. These values are specified by the `Wordlength` and `FractionLength` variables in the setup script which are set to 16 and 19 for this design. You can also set the maximum bit width by setting the `MaxOut` variable. For most applications, the maximum bit width is not needed and you can save resources by setting a threshold value for this variable. The default value of `inf` allows worst case bit growth.

The FFT_8K_br subsystem includes the Device block, a Bit Reverse Core block from the Filter library and a lower level FFT_8K_BR_Natural subsystem.

The FFT_8K_BR_Natural subsystem uses the Complex Multiplier, Complex Sample Delay, Butterfly I, Butterfly II, and Twiddle Generator blocks from the FFT library. It also includes Channel In, Channel Out, Counter, Bit Extract, Sequence, Sample Delay, Convert, Dual Memory, Constant, Compare Equality, OR Gate, Not Gate, Multiplexer, and Synthesis Information blocks.

The example file is named demo_fft_8192_br.mdl.

4K IFFT

This model implements a 4,096 point, Radix $2^2$ inverse fast Fourier transform. This design accepts naturally ordered inputs, and the output of the FFT algorithm is in bit reversed order. The design includes a bit-reversal block after the FFT algorithm so that the final result is easier to visualize.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks. An Edit Params block allows easy access to the setup variables in the setup_demo_ifft_4096_natural.m script.

The FFT designs do not inherit bit width and scaling information. These values are specified by the `Wordlength` and `FractionLength` variables in the setup script which are set to 16 and 19 for this design. You can also set the maximum bit width by setting the `MaxOut` variable. For most applications, the maximum bit width is not needed and you can save resources by setting a threshold value for this variable. The default value of `inf` allows worst case bit growth.

The FFT_4K_n subsystem includes the Device block, a Bit Reverse Core block from the Filter library and a lower level FFT_4K_Natural_BR subsystem.
The FFT_4K_Natural_BR subsystem uses the Complex Multiplier, Complex Sample Delay, Butterfly I, Butterfly II, and Twiddle Generator blocks from the FFT library. It also includes Channel In, Channel Out, Counter, Bit Extract, Sequence, Sample Delay, Convert, Dual Memory, Constant, Compare Equality, OR Gate, Not Gate, Multiplexer, and Synthesis Information blocks.

The example file is named demo_ifft_4096_natural.mdl.

8K IFFT

This model implements a 8,192 point, Radix 2^2 inverse fast Fourier transform. This design accepts naturally ordered inputs, and the output of the FFT algorithm is in bit reversed order. The design includes a bit-reversal block after the FFT algorithm so that the final result is easier to visualize.

The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks. An Edit Params block allows easy access to the setup variables in the setup_demo_ifft_8192_natural.m script.

The FFT designs do not inherit bit width and scaling information. These values are specified by the Wordlength and FractionLength variables in the setup script which are set to 16 and 19 for this design. You can also set the maximum bit width by setting the MaxOut variable. For most applications, the maximum bit width is not needed and you can save resources by setting a threshold value for this variable. The default value of inf allows worst case bit growth.

The FFT_8K_n subsystem includes the Device block, a Bit Reverse Core block from the Filter library and a lower level FFT_8K_Natural_BR subsystem.

The FFT_8K_Natural_BR subsystem uses the Complex Multiplier, Complex Sample Delay, Butterfly I, Butterfly II, and Twiddle Generator blocks from the FFT library. It also includes Channel In, Channel Out, Counter, Bit Extract, Sequence, Sample Delay, Convert, Dual Memory, Constant, Compare Equality, OR Gate, Not Gate, Multiplexer, and Synthesis Information blocks.

The example file is named demo_ifft_8192_natural.mdl.

Test CORDIC Functions Using Primitive Blocks

This model demonstrates how to use primitives blocks to build a subsystem that implements the coordinate rotation digital algorithm (CORDIC) algorithm. Supported functions include calculating the trigonometric functions of sine, cosine, magnitude and phase (arctangent) to any desired precision using shift-add algorithms instead of iterative multiplications.

This design uses a 16-bit signed fixed-point and illustrates rotating the input vector by a specified angle; and rotating the input vector to the x-axis while recording the angle required to make that rotation.
The top-level testbench includes Control, Signals, and Run Quartus II blocks.
The CordicChip subsystem includes Channel In, Channel Out, and Synthesis Information blocks, plus sixteen Vector and sixteen Rotate masked subsystems that perform the CORDIC calculations.
The Vector and Rotate masked subsystems are built using Add, Bit Extract, Constant, Multiplexer, Not, and Subtract blocks.
The example file is named test_cordic_prim.mdl.

**Test CORDIC Functions Using the CORDIC Block**

This model demonstrates how to use the primitive CORDIC block to implement the coordinate rotation digital algorithm (CORDIC) algorithm. A Mode input can be used to either rotate the input vector by a specified angle; or rotate the input vector to the x-axis while recording the angle required to make that rotation. You can experiment with different size of inputs to control the precision of the CORDIC output.
The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks.
The SinCos/AGC subsystem includes Channel In, Channel Out, CORDIC, and Synthesis Information blocks.
The example file is named demo_cordic_lib_block.mdl.

For more information, refer to the Help page for the CORDIC block.

**Memory-Mapped Registers**

This model is an extreme example of using the processor registers to implement a simple calculator. Registers and shared memories are used to write arguments and read results.
The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks.
The RegChip subsystem includes Register Field, Register Bit, Register Out, Shared Memory, Constant, Add, Subtract, Multiply, Convert, Select, Bit Extract, Shift, and Synthesis Information blocks.
The example file is named demo_regs.mdl.

**Scale**

This model demonstrates the use of the Scale block. The testbench allows you to see a vectorized block in action. Displays in the testbench track the smallest and largest values to be scaled and verify the correct behavior of the saturation modes.
The top-level testbench includes Control, Signals, Run ModelSim, and Run Quartus II blocks, plus a Channel Viewer block that is used to deserialize the output bus.
The ScaleSystem subsystem includes the Device and Scale blocks.
The example file is named demo_scale.mdl.
Tutorial Designs

Several of the demonstration design examples are used for tutorials that are described in separate chapters of this user guide.

ModellIP

The ModellIP tutorial describes how you can build a numerically controlled oscillator design using the NCO block from the Waveform Synthesis library.

This tutorial is described in “ModellIP Tutorial” on page 3–1.

This demonstration design uses the Simulink Signal Processing Blockset.

Primitive Library

The Primitive Library tutorial shows how you can build a simple design that generates a Fibonacci sequence using blocks from the ModelPrim library.

This tutorial is described in “Primitive Library Tutorial” on page 4–1.

System

The System tutorial describes the 16-Channel DDC demonstration design in detail and shows how you can combine blocks to build a system level design.

This tutorial is described in “System Tutorial” on page 5–1.

This demonstration design uses the Simulink Signal Processing Blockset.

Reference Designs

The installation also includes reference designs that demonstrate the design of digital down converter (DDC) and digital up converter (DUC) systems suitable for digital intermediate frequency (IF) processing.

For more information about these designs, refer to AN 544: Digital IF Modem Design with the DSP Builder Advanced Blockset.

4-Carrier, 2-Antenna W-CDMA DDC

This model uses ModelIP and ModelBus blocks to build a 16-channel, 2-antenna, multiple-frequency modulation digital down converter (DDC) suitable for use in an IF modem design compatible with the W-CDMA standard.

The top-level testbench includes Control, Signals, and Run Quartus II blocks, plus a Channel Viewer block that is used to isolate two channels of data from the time-division multiplexed (TDM) signals.
The DDCChip subsystem includes Device, Decimating CIC, Decimating FIR, Real Mixer, NCO, and Scale blocks. It also contains a Sync subsystem which provides the synchronization of the channel data to the NCO carrier waves.

The CIC and FIR filters are used to implement a decimating filter chain that down converts the eight complex carriers (16 real channels from two antennas with four pairs of I and Q inputs from each antenna) from a frequency of 122.88 MSPS to a frequency of 7.68 MSPS (a total decimation rate of 16). The real mixer and NCO are used to isolate the four channels. The NCO is configured with four channels to provide four pairs of sine and cosine waves at frequencies of 12.5 MHz, 17.5 MHz, 22.5 MHz, and 27.5 MHz, respectively. The NCO has the same sample rate (122.88 MSPS) as the input data sample rate.

The Sync subsystem shows how to manage two data streams coming together and being synchronized. The data from the NCOs are written to a memory using the channel as an address: the data stream using its channel signals to read out the NCO signals. This resynchronizes the data correctly.

A system clock rate of 245.76 MHz drives the design on the FPGA defined by the Device block inside the DDCChip subsystem.

The example file is named wcdma_multichannel_ddc_mixer.mdl.

This demonstration design uses the Simulink Signal Processing Blockset.

1-Carrier, 2-Antenna W-CDMA DDC

This model uses ModelIP and ModelBus blocks to build a 4-channel, 2-antenna, single-frequency modulation digital down converter (DDC) suitable for use in an IF modem design compatible with the W-CDMA standard.

The top-level testbench includes Control, Signals, and Run Quartus II blocks, plus a Channel Viewer block that is used to isolate two channels of data from the time-division multiplexed (TDM) signals.

The DDCChip subsystem includes Device, Decimating CIC, Decimating FIR, Real Mixer, NCO, and Scale blocks.

The CIC and FIR filters are used to implement a decimating filter chain that down converts the two complex carriers (4 real channels from two antennas with one pair of I and Q inputs from each antenna) from a frequency of 122.88 MSPS to a frequency of 7.68 MSPS (a total decimation rate of 16). The real mixer and NCO are used to isolate the four channels. The NCO is configured with a single channel to provide one sine and one cosine wave at a frequency of 17.5 MHz. The NCO has the same sample rate (122.88 MSPS) as the input data sample rate.

A system clock rate of 122.88 MHz drives the design on the FPGA defined by the Device block inside the DDCChip subsystem.

The example file is named wcdma_picocell_ddc_mixer.mdl.

This demonstration design uses the Simulink Signal Processing Blockset.
4-Carrier, 2-Antenna W-CDMA DUC

This model uses ModelIP and ModelBus blocks to build a 16-channel, 2-antenna, multiple-frequency modulation digital up converter (DUC) suitable for use in an IF modem design compatible with the W-CDMA standard.

The top-level testbench includes Control, Signals, and Run Quartus II blocks. A Spectrum Scope block computes and displays the periodogram of the outputs from the two antennas.

The DUCChip subsystem includes a Device block to specify the target FPGA device, and a DUC subsystem that contains Interpolating FIR, Interpolating CIC, NCO, Complex Mixer, and Scale blocks.

The FIR and CIC filters are used to implement an interpolating filter chain that up converts the 16-channel input data from a frequency of 3.84 MSPS to a frequency of 122.88 MSPS (a total interpolation factor of 32). The complex mixer and NCO are used to modulate the four channel baseband input signal onto the IF region. The NCO has the same sample rate (122.88 MSPS) as the final interpolated output sample rate from the last CIC filter in the interpolating filter chain.

The Sync subsystem shows how to manage two data streams coming together and being synchronized. The data from the NCOs are written to a memory using the channel as an address: the data stream using its channel signals to read out the NCO signals. This resynchronizes the data correctly.

The CarrierSum and SignalSelector subsystems are used to sum up the right modulated signals to the designated antenna.

A system clock rate of 245.76 MHz drives the design on the FPGA defined by the Device block inside the DUC subsystem.

The example file is named wcdma_multichannel_duc_mixer.mdl.

This demonstration design uses the Simulink Signal Processing Blockset.

1-Carrier, 2-Antenna W-CDMA DDC

This model uses ModelIP and ModelBus blocks to build a 4-channel, 2-antenna, single-frequency modulation digital up converter (DUC) suitable for use in an IF modem design compatible with the W-CDMA standard.

The top-level testbench includes Control, Signals, and Run Quartus II blocks. A Spectrum Scope block computes and displays the periodogram of the outputs from the two antennas.

The DUCChip subsystem includes a Device block to specify the target FPGA device, and a DUC subsystem that contains Interpolating FIR, Interpolating CIC, NCO, Complex Mixer, and Scale blocks.

The FIR and CIC filters are used to implement an interpolating filter chain that up converts the four channel input data from a frequency of 3.84 MSPS to a frequency of 122.88 MSPS (a total interpolation factor of 32). The complex mixer and NCO are used to modulate the four channel baseband input signal onto the IF region.
The NCO is configured with a single channel to provide one sine and one cosine wave at a frequency of 17.5 MHz. The NCO has the same sample rate (122.88 MSPS) as the final interpolated output sample rate from the last CIC filter in the interpolating filter chain.

A system clock rate of 122.88 MHz drives the design on the FPGA defined by the Device block inside the DDC subsystem.

The example file is named \texttt{wcdma\_picocell\_duc\_mixer.mdl}.

This demonstration design uses the Simulink Signal Processing Blockset.

\textbf{4-Carrier, 2-Antenna High-Speed W-CDMA DUC at 368.64 MHz with Total Rate Change 32}

This model uses ModelIP and ModelBus blocks to build a high-speed 16-channel, 2-antenna, multiple-frequency modulation digital up converter (DUC) suitable for use in an IF modem design compatible with the W-CDMA standard.

The top-level testbench includes \texttt{Control}, \texttt{Signals}, and \texttt{Run Quartus II} blocks. A \texttt{Spectrum Scope} block computes and displays the periodogram of the outputs from the two antennas.

The \texttt{DUCChip} subsystem includes a \texttt{Device} block to specify the target FPGA device, and a \texttt{DUC} subsystem that contains \texttt{Interpolating FIR}, \texttt{Interpolating CIC}, \texttt{NCO}, \texttt{Complex Mixer}, and \texttt{Scale} blocks.

The \texttt{FIR} and \texttt{CIC} filters are used to implement an interpolating filter chain that up converts the 16-channel input data from a frequency of 3.84 MSPS to a frequency of 122.88 MSPS (a total interpolation factor of 32). Because of the unusual FPGA clock frequency and total rate change combination, this example uses dummy signals and carriers to achieve the desired rate up conversion. The complex mixer and NCO are used to modulate the four channel baseband input signal onto the IF region. The NCO is configured with four channels to provide four pairs of sine and cosine waves at frequencies of 12.5 MHz, 17.5 MHz, 22.5 MHz and 27.5 MHz, respectively. The NCO has the same sample rate (122.88 MSPS) as the final interpolated output sample rate from the last CIC filter in the interpolating filter chain.

The \texttt{Sync} subsystem shows how to manage two data streams coming together and being synchronized. The data from the NCOs are written to a memory using the channel as an address: the data stream using its channel signals to read out the NCO signals. This resynchronizes the data correctly.

The \texttt{GenCarrier} subsystem manipulates the NCO outputs to generate carrier signals that could align with the data path signals.

The \texttt{CarrierSum} and \texttt{SignalSelector} subsystems are used to sum up the right modulated signals to the designated antenna.

A system clock rate of 368.64 MHz, which is 96 times the input sample rate, drives the design on the FPGA defined by the \texttt{Device} block inside the \texttt{DUC} subsystem. The higher clock rate could potentially allow resource re-use in other modules of a digital system implemented on an FPGA.

The example file is named \texttt{wcdma\_multichannel\_duc\_mixer\_96x\_32R.mdl}.

This demonstration design uses the Simulink Signal Processing Blockset.
4-Carrier, 2-Antenna High-Speed W-CDMA DUC at 368.64 MHz with Total Rate Change 48

This model uses ModelIP and ModelBus blocks to build a high-speed 16-channel, 2-antenna, multiple-frequency modulation digital up converter (DUC) suitable for use in an IF modem design compatible with the W-CDMA standard.

The top-level testbench includes Control, Signals, and Run Quartus II blocks. A Spectrum Scope block computes and displays the periodogram of the outputs from the two antennas.

The DUCChip subsystem includes a Device block to specify the target FPGA device, and a DUC subsystem that contains Interpolating FIR, Interpolating CIC, NCO, Complex Mixer, and Scale blocks.

The FIR and CIC filters are used to implement an interpolating filter chain that up converts the 16-channel input data from a frequency of 3.84 MSPS to a frequency of 184.32 MSPS (a total interpolation factor of 48).

The complex mixer and NCO are used to modulate the four channel baseband input signal onto the IF region. The NCO is configured with four channels to provide four pairs of sine and cosine waves at frequencies of 12.5 MHz, 17.5 MHz, 22.5 MHz, and 27.5 MHz, respectively. The NCO has the same sample rate (184.32 MSPS) as the final interpolated output sample rate from the last CIC filter in the interpolating filter chain.

The Sync subsystem shows how to manage two data streams coming together and being synchronized. The data from the NCOs are written to a memory using the channel as an address: the data stream using its channel signals to read out the NCO signals. This resynchronizes the data correctly.

The CarrierSum and SignalSelector subsystems are used to sum up the right modulated signals to the designated antenna.

A system clock rate of 368.64 MHz, which is 96 times the input sample rate, drives the design on the FPGA defined by the Device block inside the DUC subsystem. The higher clock rate could potentially allow resource re-use in other modules of a digital system implemented on an FPGA.

The example file is named wcdma_multichannel_duc_mixer_96x_48R.mdl.

This demonstration design uses the Simulink Signal Processing Blockset.

4-Carrier, 2-Antenna High-Speed W-CDMA DUC at 307.2 MHz with Total Rate Change 40

This model uses ModelIP and ModelBus blocks to build a high-speed 16-channel, 2-antenna, multiple-frequency modulation digital up converter (DUC) suitable for use in an IF modem design compatible with the W-CDMA standard.

The top-level testbench includes Control, Signals, and Run Quartus II blocks. A Spectrum Scope block computes and displays the periodogram of the outputs from the two antennas.

The DUCChip subsystem includes a Device block to specify the target FPGA device, and a DUC subsystem that contains Interpolating FIR, Interpolating CIC, NCO, Complex Mixer, and Scale blocks.
The FIR and CIC filters are used to implement an interpolating filter chain that up converts the 16-channel input data from a frequency of 3.84 MSPS to a frequency of 153.6 MSPS (a total interpolation factor of 40).

The complex mixer and NCO are used to modulate the four channel baseband input signal onto the IF region. The NCO is configured with four channels to provide four pairs of sine and cosine waves at frequencies of 12.5 MHz, 17.5 MHz, 22.5 MHz, and 27.5 MHz, respectively. The NCO has the same sample rate (153.6 MSPS) as the final interpolated output sample rate from the last CIC filter in the interpolating filter chain.

The Sync subsystem shows how to manage two data streams coming together and being synchronized. The data from the NCOs are written to a memory using the channel as an address: the data stream using its channel signals to read out the NCO signals. This resynchronizes the data correctly.

The CarrierSum and SignalSelector subsystems are used to sum up the right modulated signals to the designated antenna.

A system clock rate of 307.2 MHz, which is 80 times the input sample rate, drives the design on the FPGA defined by the Device block inside the DUC subsystem. The higher clock rate could potentially allow resource re-use in other modules of a digital system implemented on an FPGA.

The example file is named wcdma_multichannel_duc_mixer_80x_40R.mdl.

This demonstration design uses the Simulink Signal Processing Blockset.

1-Antenna WiMAX DDC

This model uses ModelIP and ModelBus blocks to build a 2-channel, 1-antenna, single-frequency modulation digital down converter (DDC) suitable for use in an IF modem design compatible with the WiMAX standard.

The top-level testbench includes Control, Signals, and Run Quartus II blocks. An Edit Params block is included to allow easy access to the setup variables in the setup_wimax_ddc_1rx.m script.

The DDCChip subsystem includes Device, Decimating FIR, Real Mixer, NCO, Single Rate FIR, and Scale blocks. There is also an Interleaver subsystem to extract the correct I and Q channel data from the demodulated data stream.

The FIR filters are used to implement a decimating filter chain that down convert the two channels from a frequency of 89.6 MSPS to a frequency of 11.2 MSPS (a total decimation rate of eight). The real mixer, NCO, and Interleaver subsystem are used to isolate the two channels. The NCO is configured with a single-channel to provide one sine and one cosine wave at a frequency of 22.4 MHz. The NCO has the same sample rate (89.6 MSPS) as the input data sample rate.

A system clock rate of 179.2 MHz drives the design on the FPGA defined by the Device block inside the DDCChip subsystem.

The example file is named wimax_ddc_1rx.mdl.

This demonstration design uses the Simulink Signal Processing Blockset.
Chapter 2: Example Designs 2–29

Demonstration Designs

2-Antenna WiMAX DDC

This model uses ModelIP and ModelBus blocks to build a 4-channel, 2-antenna, 2-frequency modulation digital down converter (DDC) suitable for use in an IF modem design compatible with the WiMAX standard.

The top-level testbench includes Control, Signals, and Run Quartus II blocks. An Edit Params block is included to allow easy access to the setup variables in the setup_wimax_ddc_2rx_iqq.m script.

The DDCChip subsystem includes Device, Decimating FIR, Real Mixer, NCO, Single Rate FIR, and Scale blocks.

The FIR filters are used to implement a decimating filter chain that down convert the two channels from a frequency of 89.6 MSPS to a frequency of 11.2 MSPS (a total decimation rate of 8). The real mixer and NCO are used to isolate the two channels. The NCO is configured with two channels to provide two sets of sine and cosine waves at the same frequency of 22.4 MHz. The NCO has the same sample rate of (89.6 MSPS) as the input data sample rate.

A system clock rate of 179.2 MHz drives the design on the FPGA defined by the Device block inside the DDCChip subsystem.

The example file is named wimax_ddc_2rx_iqq.mdl.

1-Antenna WiMAX DUC

This model uses ModelIP, ModelBus, and ModelPrim blocks to build a 2-channel, 1-antenna, single-frequency modulation digital up converter (DUC) suitable for use in an IF modem design compatible with the WiMAX standard.

The top-level testbench includes Control, Signals, and Run Quartus II blocks. An Edit Params block is included to allow easy access to the setup variables in the setup_wimax_duc_1tx.m script.

The DUCChip subsystem includes a Device block to specify the target FPGA device, and a DUC2Channel subsystem which contains Single Rate FIR, Scale, Interpolating FIR, NCO, and Complex Mixer blocks. There is also an de-interleaver subsystem containing a series of ModelPrim blocks including delays and multiplexers that de-interleave the two I and Q channels.

The FIR filters are used to implement an interpolating filter chain that up converts the two channels from a frequency of 11.2 MSPS to a frequency of 89.6 MSPS (a total interpolating rate of 8). The complex mixer and NCO are used to modulate the two input channel baseband signals to the IF domain. The NCO is configured with a single channel to provide one sine and one cosine wave at a frequency of 22.4 MHz. The NCO has the same sample rate (89.6 MSPS) as the input data sample rate.

A system clock rate of 179.2 MHz drives the design on the FPGA defined by the Device block inside the DUCChip subsystem.

The example file is named wimax_duc_1tx.mdl.

This demonstration design uses the Simulink Signal Processing Blockset.
2-Antenna WiMAX DUC

This model uses ModelIP, ModelBus, and ModelPrim blocks to build a 4-channel, 2-antenna, single-frequency modulation digital up converter (DUC) suitable for use in an IF modem design compatible with the WiMAX standard.

The top-level testbench includes Control, Signals, and Run Quartus II blocks. An Edit Params block is included to allow easy access to the setup variables in the setup_wimax_duc_2tx_iqq.m script.

The DUCChip subsystem includes a Device block to specify the target FPGA device, and a DUC2Channel subsystem which contains Single Rate FIR, Scale, Interpolating FIR, NCO, Complex Mixer, and Constant blocks. It also contains a Sync subsystem which shows how to manage two data streams coming together and being synchronized. The data from the NCO are written to a memory using the channel index as an address: the data stream using its channel signals to read out the NCO signals. This resynchronizes the data correctly. (Alternatively, you could simply delay the NCO value by the correct number of cycles to ensure that the NCO and channel data arrive at the Mixer on the same cycle). There is also an de-interleaver subsystem containing a series of ModelPrim blocks including delays and multiplexers that de-interleave the four I and Q channels.

The FIR filters are used to implement an interpolating filter chain that up converts the two channels from a frequency of 11.2 MSPS to a frequency of 89.6 MSPS (a total interpolating rate of 8).

A complex mixer and NCO are used to modulate the two input channel baseband signals to the IF domain. The NCO is configured to provide two sets of sine and cosine waves at a frequency of 22.4 MHz. The NCO has the same sample rate (89.6 MSPS) as the input data sample rate.

The Sync subsystem shows how to manage two data streams coming together and being synchronized. The data from the NCOs are written to a memory using the channel as an address: the data stream using its channel signals to read out the NCO signals. This resynchronizes the data correctly.

A system clock rate of 179.2 MHz drives the design on the FPGA defined by the Device block inside the DUCChip subsystem.

The example file is named wimax_duc_2tx_iqq.mdl.

This demonstration design uses the Simulink Signal Processing Blockset.
The Numerically Controlled Oscillator Design

The ModelIP tutorial is based on a numerically controlled oscillator (NCO) design. Numerically controlled oscillators are efficient means of generating sinusoidal signals. They are useful when a continuous phase sinusoidal signal with variable frequency is required. The Altera DSP Builder Advanced NCO block uses an octant based algorithm with trigonometric interpolation.

The NCO block builds a 4-channel programmable NCO suitable for use in a digital down-converter (DDC) or digital up-converter (DUC).

The design is built in the following stages:
1. Opening the design model and setting the configuration parameters
2. Simulating the design model in Simulink and exploring the generated files
3. Simulating the RTL in ModelSim
4. Compiling with the Quartus II software
5. Instantiating the design in SOPC Builder

The design is then customized in two different directions:
1. A 24-channel NCO targeting a system clock frequency of 360 MHz
2. An NCO with increased spurious-free dynamic range (SFDR) of 130 dB

Opening the NCO Design

The demonstration model uses a preconfigured NCO in a testbench that allows you to examine the spectral properties of the NCO.

To open the model, type the following command in the MATLAB window:

demo_nco

You can also open the model by clicking Open this model in the header for the ModelIP Tutorial in the Demos tab of the MATLAB Help viewer.

Most of the top-level model blocks are concerned with the testbench which contains the sources required to drive the NCO and a collection of blocks to post process the data, displaying the results in the time and frequency domains.

\[ \text{Control Block}\]

The Control Block controls whether to generate hardware, the destination directory, the parameters for testbench creation, selects a big endian or little endian system bus, and specifies the memory-mapped interface details. A Control block should be placed at the top-level of each model.
The Signals block sets up the clock and reset names and parameters for the system. One Signals block should be placed at the top-level of each model. At this point, it is important to note that the system clock has been set to 240MHz.

The Edit Params block provides access to the setup_demo_nco.m script that specifies MATLAB workspace variables used to configure your design.

The source blocks (Channel Counter and Valid Counter) provide channel and valid stimulus for the NCO block. These provide the essential first stage of the DSP Builder Advanced blockset interface protocol.

The NCOSubSystem contains the NCO block itself and is described later.

The remainder of your model compares the NCO output to that of the Simulink Sine Wave and Cosine Wave blocks for reference. A key part of this are the two Channel Viewer blocks which are used to deserialize the time-division multiplexed (TDM) output from the NCO.

The top-level NCO demonstration model is shown in Figure 3–1.

**Figure 3–1. Top-Level NCO Demonstration Model**
Setting the Configuration Parameters

The configuration parameters are specified by MATLAB workspace variables that are specified in the setup_demo_nco.m script. This script can be accessed using the Edit Params block in the top-level model.

1. Double-click on the Edit Params block to open the script in the MATLAB text editor.

The following variables are specified:

- `ChanCount = 4;`
- `ClockRate = 240.00;`
- `ClockMargin = 0.0;`
- `SampleRate = 60;`
- `Period = ClockRate / SampleRate;`
- `SampleTime = 1;`

2. Check that the `SampleTime` variable is set to 1 (and the alternative line for real world time is commented out):

   ```matlab
   SampleTime = 1;
   % SampleTime = 1/(ClockRate * 1e6); % uncomment this line to simulate the model with real world time
   ```

   Setting the sample time to 1 means that each step in your model corresponds to a clock cycle in the hardware. This facilitates easy debugging in the hardware, and communication with any hardware interfaces required.

3. Save the file and close the editor window.

   ![Warning]
   If you change the sample time to be the clock period (1/240e6), and run simulation for 20000/240e6 seconds, this gives a more architectural view of your model making it easier to relate to the intended real world behavior.

Configuring the NCO

1. Double-click on the NCOSubsystem block to open the NCO subsystem (Figure 3–2 on page 3–4).

   The Device block labels this level of hierarchy as the top-level of the RTL that is implemented in the chosen device. Everything above this level is a testbench. Everything below this level is synthesized into hardware and implemented on the FPGA. In this example, the device family is set to Stratix II and the device set to AUTO. This allows the Quartus II software to pick an appropriately sized member of the Stratix II family.

   The NCO block represents the actual NCO itself.

   Two Constant blocks are used to blank off the phase update logic. You can use these inputs to update the phase values in applications that need known phases.
2. Double click on the NCO block to open its configuration parameter dialog box (Figure 3–3).

This example produces four channels of complex output (one sine and one cosine) at a data rate of 60MHz where the frequency is specified within 5Hz.
This is achieved by the following parameters:

- The **Output rate per channel** determines the amount of folding used. This parameter is set to the value of the *SampleRate* variable (which can be edited using the *Edit Params* block on the top-level model). The value is initially set to a value of 60 MSPS which creates outputs with single width wires.

- The **Output data type** is set to `sfix(18)`. This setting produces sines and cosines with 22 bits of precision.

- The **Output scaling value** is set to $2^{-17}$. This setting ensures that the value is interpreted as a `sfix18_En17 (Q1.17)` signed number and that Simulink reports the values inside the range (-1.0 to 1.0). Note that to generate symmetrical data the extreme values should not be used.

- The **Accumulator bit width** is set to 24 and specifies the width of the memory-mapped accumulator bit width. This governs the precision with which the NCO frequency can be controlled.

- The **Phase increment and inversion** is a vector that represents the step in phase used between each sample. This setting controls the frequencies that are generated during simulation. The length of the vector determines how many channels of data are generated. The following vector value is specified:

  $$[1:0.01:(1.0+((\text{ChanCount})-1)/100.0)] \cdot (2^{24} \times 0.051) + (0 \times 2^{24}) + (0 \times 2^{25})$$

  The first part of this vector $[1:0.01:(1.0+((\text{ChanCount})-1)/100.0)] \cdot (2^{24} \times 0.051)$ specifies the values $1.0e+005 \times [8.5564 \ 8.6419 \ 8.7275 \ 8.8131]$ of the phase increments for four channels. The frequency is defined by the equation:

  $$\text{Frequency} = (\text{Output Rate}) \times (\text{Phase Increment Value}) / (2^{(\text{Accumulator Bit Width})})$$

  Therefore, the phase increment values produce a set of frequencies in MHz: $[3.06 \ 3.0906 \ 3.1212 \ 3.1518]$. The value of the **Accumulator bit width** parameter in this example is set to 24. Hence the binary format of the phase increment values are expected in 26 binary digits as follows:

  00 0000 1101 0000 1110 0101 0110 00 0000 1101 0010 1111 1100 0010 00 0000 1101 0101 0001 0010 1111 00 0000 1101 0111 0010 1001 1011

  The lower 24 bits are used to specify the frequencies for the four channels. The top two bits are used to control the inversion of sine (26th bit) and cosine (25th bit) respectively.

  The second part of the phase increment and inversion vector, $(0 \times 2^{24})$, manipulates the 25th bit of the values. When $(0 \times 2^{24})$ is specified, a 0 is set to the 25th bit, as shown by the bit pattern above. However, if it is changed to $(1 \times 2^{24})$, a 1 is produced at the 25th bit, as shown by the following bit pattern, indicating that the cosine wave is inverted:

  01 0000 1101 0000 1110 0101 0110 01 0000 1101 0010 1111 1100 0010 01 0000 1101 0101 0001 0010 1111 01 0000 1101 0111 0010 1001 1011
The third part of the phase increment and inversion vector, \((0 \times 2^{25})\), manipulates the 26th bit of the phase increment and inversion values. When \((0 \times 2^{25})\) is specified, a 0 is set to the 26th bit. However, if it is changed to \((1 \times 2^{25})\), a 1 is produced at the 26th bit, indicating that the sine wave is inverted.

The following example demonstrates the case when both the sine and cosine waves are inverted:

\[
11\ 0000\ 1101\ 0000\ 1110\ 0101\ 0110\\
11\ 0000\ 1101\ 0010\ 1111\ 1100\ 0010\\
11\ 0000\ 1101\ 0101\ 0001\ 0010\ 1111\\
11\ 0000\ 1101\ 0111\ 0010\ 1001\ 1011
\]

The following example demonstrates the case when only the sine wave is inverted:

\[
10\ 0000\ 1101\ 0000\ 1110\ 0101\ 0110\\
10\ 0000\ 1101\ 0010\ 1111\ 1100\ 0010\\
10\ 0000\ 1101\ 0101\ 0001\ 0010\ 1111\\
10\ 0000\ 1101\ 0111\ 0010\ 1001\ 1011
\]

The top two bits (the 25th bit and 26th bit) change the binary bit patterns of the phase increment and inversion parameter, but they do not alter the frequencies of the generated sinusoidal signals. Figure 3–4 shows the frequency of the sinusoidal waves for the 1st channel. Any values in bits higher than the 26th bit are ignored.

Figure 3–4. Frequency Spectrum in the SinSpectrum Block

Figure 3–5, Figure 3–6, Figure 3–7, and Figure 3–8 show how the sine and cosine waves appear in the Simulink scopes after they have been inverted.
Figure 3–5. Neither Inverted: \[1.0.01:(1.0+(\text{ChanCount}-1)/100.0)] \cdot (2^{24} \times 0.051) + (0 \times 2^{24}) + (0 \times 2^{25})

Figure 3–6. Cosine Inverted: \[1.0.01:(1.0+(\text{ChanCount}-1)/100.0)] \cdot (2^{24} \times 0.051) + (1 \times 2^{24}) + (0 \times 2^{25})
Figure 3–7. Sine Inverted: \[(1:0.01:(1.0+(\text{ChanCount}-1)/100.0)) \cdot (2^{24} \times 0.051) + (0 \times 2^{24}) + (1 \times 2^{25})\]

Figure 3–8. Both Inverted: \[(1:0.01:(1.0+(\text{ChanCount}-1)/100.0)) \cdot (2^{24} \times 0.051) + (1 \times 2^{24}) + (1 \times 2^{25})\]
Chapter 3: ModelIP Tutorial

Configuring the NCO

- The **Phase increment and inversion memory map** value controls where in the memory-mapped space the NCO registers are mapped. This value is set to 0.

- The **Read/Write mode** controls access to the memory-mapped registers. This option is set to **Read/Write**.

For more information about the NCO parameters, refer to the description of the NCO block in the *DSP Builder Advanced Blockset Reference Manual*.

3. Click on the **Results** tab to display the results for the specified parameters (Figure 3–9).

**Figure 3–9.** NCO Function Block Parameters Dialog Box

This shows the worst case SFDR based on the bit width. Measured values may be significantly better than this, and you should perform further analysis.

The accumulator precision is shown for the specified accumulator bit width.

The actual frequencies implied by the initial phase increment values are shown as being around 3MHz for each channel.

The number of output wires (width of output vector or outputs per cycle) is also shown.
Simulating the Model in Simulink

Simulating your model in Simulink generates enough data for frequency analysis.

1. Simulate your design by clicking on .

2. To verify the results, examine the sine data displayed in the time domain by the sinScope block (Figure 3–10).

Figure 3–10. Sine Data Displayed by the SinScope Block

Because each sample time represents 1/240MHz, you can see a sine wave with period of roughly 80 cycles. This corresponds to 240/80 = 3MHz which is what was specified. A more detailed analysis can be performed in the frequency domain. Looking at the frequency domain spectrum in the SinSpectrum block, you can see the expected SFDR which you could analyze more closely from the data saved to the MATLAB workspace (Figure 3–4).

Exploring the Generated Files

If the Generate Hardware option is on in the parameters for the Control block, then every time the simulation runs, the underlying hardware is synthesized, and VHDL is written out into the specified directory. The directory structure mirrors the structure of your model. For example, if the current directory is C:\my_projects\designs, the structure shown in Figure 3–11 is created.

Figure 3–11. Generated Directory Structure for the NCO Demonstration Design
You can specify the root to this directory as an absolute path name or as a relative path name. If you specify a relative path name (such as `./rtl`), the directory structure is created relative to the MATLAB current directory.

Table 3–1 lists the generated files for the NCO Demonstration Design.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rtl directory</strong></td>
<td></td>
</tr>
<tr>
<td>demo_nco.xml</td>
<td>An XML file that describes the attributes of your model.</td>
</tr>
<tr>
<td>demo_nco_entity.xml</td>
<td>An XML file that describes the boundaries of the system (used by Signal Compiler in designs that combine blocks from the standard and advanced blocksets).</td>
</tr>
<tr>
<td><strong>rtl\demo_nco subdirectory</strong></td>
<td></td>
</tr>
<tr>
<td>Control.do</td>
<td>Running this script compiles your design, loads it into ModelSim, and runs it for the same simulation time that your model ran for in Simulink.</td>
</tr>
<tr>
<td>Control.wav.do</td>
<td>This script loads the signals from your design into the ModelSim Wave window.</td>
</tr>
<tr>
<td>&lt;block name&gt;.xml</td>
<td>An XML file containing information about each block in the advanced blockset which is translated into HTML on demand for display in the MATLAB Help viewer.</td>
</tr>
<tr>
<td>demo_nco.vhd</td>
<td>This is the top-level testbench file. It may contain non-synthesizable blocks, and may also contain empty black boxes for Simulink blocks that are not fully supported.</td>
</tr>
<tr>
<td>demo_nco.add.do</td>
<td>This script loads the VHDL files in this subdirectory and in the subsystem hierarchy below it into the ModelSim project.</td>
</tr>
<tr>
<td>demo_nco.compile.do</td>
<td>This script compiles the VHDL files in this subdirectory and in the subsystem hierarchy below it into the ModelSim project.</td>
</tr>
<tr>
<td>demo_nco.add.tcl</td>
<td>This script loads the VHDL files in this subdirectory and in the subsystem hierarchy below it into the Quartus II project.</td>
</tr>
<tr>
<td>demo_nco.qip</td>
<td>This file contains all the assignments and other information required to process the demo_nco model in the Quartus II software. The file includes a reference to the .qip file in the NCOSubsystem.</td>
</tr>
<tr>
<td>demo_nco_&lt;block name&gt;.vhd</td>
<td>A VHDL file is generated for each component in your model.</td>
</tr>
<tr>
<td>demo_nco_NCOSubsystem_entity.xml</td>
<td>An XML file that describes the boundaries of the subsystem as a block box (used by Signal Compiler in designs that combine blocks from the standard and advanced blocksets).</td>
</tr>
<tr>
<td>NCOSubsystem.xml</td>
<td>An XML file that describes the attributes of the NCOSubsystem.</td>
</tr>
<tr>
<td>*.stm</td>
<td>Stimulus files for ModelSim.</td>
</tr>
<tr>
<td>safe_path.vhd</td>
<td>Helper function that ensures a path name is read correctly in the Quartus II software.</td>
</tr>
<tr>
<td>safe_path_msim.vhd</td>
<td>Helper function that ensures a path name is read correctly in ModelSim.</td>
</tr>
<tr>
<td><strong>rtl\demo_nco\NCOSubsystem subdirectory</strong></td>
<td></td>
</tr>
<tr>
<td>Device.xml</td>
<td>An XML files containing information about the Device block which is translated into HTML on demand for display in the MATLAB Help viewer.</td>
</tr>
<tr>
<td>demo_nco_NCOSubsystem.vhd</td>
<td>This is the VHDL file created for the subsystem.</td>
</tr>
<tr>
<td>demo_nco_NCOSubsystem_NCO.vhd</td>
<td>A VHDL file that describes the NCO component.</td>
</tr>
</tbody>
</table>
Simulating the RTL

Whenever hardware is generated, several support scripts are generated to make it easy to verify the hardware in RTL. The top-level command script generated for ModelSim is named `Control.do` after the Simulink name of the Control block in your model.

This script is called when you double-click on the Run ModelSim block in your model. This starts ModelSim and runs the script automatically for the same time period that was specified in the Simulink run.

For script based flows, the script can be called from the appropriate directory in the ModelSim shell.

```bash
    cd ../rtl
    do ../rtl/demo_nco/Control.do
```

The script compiles any FPGA vendor libraries, compiles all the RTL generated for the project, opens a ModelSim Wave window, and adds signals to that Wave window. The signals added are determined by the Signal View Depth parameter in the Control block.
Where appropriate, the signals are automatically displayed in analog format as shown in Figure 3–12.

Figure 3–12. NCOSubsystem Simulation Waveform in ModelSim for the NCO Demonstration Design

Compiling with the Quartus II Software

The most convenient way to start the Quartus II software is to double-click on the Run Quartus II block in the top-level model. This is equivalent to starting the Quartus II software manually and running the Tcl file:

```
   cd ..rtl
   source demo_nco/NCOSubsystem/NCOSubsystem.tcl
```

After your design has been loaded, you can compile your design by clicking Start Compilation on the Processing menu.
For these parameters, you can see from the Fitter Summary report that the Quartus II software chooses a small device as shown in Figure 3–13.

**Figure 3–13.** Quartus II Compilation Fitter Summary for the NCO Demonstration Design

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Quartus II Version</td>
<td>8.1 Internal Build 157 10/10/2008 SJ Full Version</td>
</tr>
<tr>
<td>Revision Name</td>
<td>NCOSubsystem</td>
</tr>
<tr>
<td>Top-level Entity Name</td>
<td>demo_nco_NCOSubsystem</td>
</tr>
<tr>
<td>Family</td>
<td>Stratix II</td>
</tr>
<tr>
<td>Device</td>
<td>EP2S15F484C3</td>
</tr>
<tr>
<td>Timing Models</td>
<td>Final</td>
</tr>
<tr>
<td>Logic utilization</td>
<td>3 %</td>
</tr>
<tr>
<td>Combinational ALUTs</td>
<td>264 / 12,480 ( 2 % )</td>
</tr>
<tr>
<td>Dedicated logic registers</td>
<td>418 / 12,480 ( 3 % )</td>
</tr>
<tr>
<td>Total registers</td>
<td>418</td>
</tr>
<tr>
<td>Total pins</td>
<td>103 / 343 ( 30 % )</td>
</tr>
<tr>
<td>Total virtual pins</td>
<td>0</td>
</tr>
<tr>
<td>Total block memory bits</td>
<td>9,587 / 419,328 ( 2 % )</td>
</tr>
<tr>
<td>DSP block Soft elements</td>
<td>4 / 96 ( 4 % )</td>
</tr>
<tr>
<td>Total PLLs</td>
<td>0 / 6 ( 0 % )</td>
</tr>
<tr>
<td>Total DLLs</td>
<td>0 / 2 ( 0 % )</td>
</tr>
</tbody>
</table>

The NCO circuit naturally compiles to a high clock frequency as shown in the TimeQuest Timing Analyzer, Slow Model, Fmax Summary report (Figure 3–14).

**Figure 3–14.** Quartus II Timing for the NCO Demonstration Design

For more full timing information, you can click **TimeQuest Timing Analyzer** on the Tools menu to access a more detailed timing report.

**Instantiating the Design in SOPC Builder**

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

1. On the Tools menu in the Quartus II software, click **SOPC Builder** to start SOPC Builder.
2. In the Create New System dialog box, enter nios32 as your System Name, select VHDL for the target HDL and click OK (Figure 3–15).

**Figure 3–15. SOPC Builder Create New System Dialog Box**

3. In SOPC Builder, click Options on the Tools menu and select IP Search Path. Click Add and browse to the NCOSubsystem directory. For example, in Figure 3–16 the hardware destination directory is D:\DSPBuilderExamples\rtl.

**Figure 3–16. SOPC Builder Options Dialog Box**

4. Click Finish.

5. On the File menu, click Refresh Component List.

The NCOSubsystem should now appear in the SOPC Builder System Contents tab under Altera DSP Builder Advanced.

6. Click the System Contents tab in SOPC Builder and expand Memories and Memory Controllers. Expand On-Chip and double-click On Chip Memory (RAM or ROM). Click Finish in the MegaWizard™ interface to add an on-chip RAM component with default parameters.
7. Double-click the **Nios II Processor** module in the **System Contents** tab. Set the reset and exception vectors to use *onchip_mem* in the MegaWizard interface and click **Finish** to add the processor to your system with all other parameters set to their default values.

8. Double-click **NCOSubsystem** to include it in your Nios II system. *(Figure 3–17).*

---

**Figure 3–17.** Nios II System including NCOSubsystem in SOPC Builder

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

9. Click **Generate** to complete your SOPC Builder system. Progress messages are issued in the **System Generation** tab ending with the message:

   `System generation was successful`

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

For information about SOPC Builder and the Nios II processor, refer to the Quartus II Help.
Customizing the NCO

The following sections show how to customize your NCO subsystem to change the number of channels or increase the spurious free dynamic range.

Changing the Number of Channels

Suppose you want to create 24 NCO frequencies. This section shows how to make the necessary changes.

One way to achieve this, is to modify the existing block to request 24 frequencies. This modification results in the use of roughly six times the amount of hardware resources.

Alternatively, you can run at a higher system clock frequency of 360 MHz. This change reduces the hardware resources by approximately 50%, resulting in an increase of approximately four times the hardware size compared with the original system. The system clock is 360 MHz and the NCO output sample rate is 60 MSPS, so the TDM factor is 6. That is, each output wire contains values for 6 channels repeated in a round-robin fashion.

The demonstration design is parameterized with values specified in a setup script. You can modify these values by performing the following steps:

1. Double-click on the Edit Params block to display the setup_demo_nco.m script in the MATLAB text editor.
2. Change the value of the ChanCount variable to 24.
3. Change the value of the ClockRate variable to 360.0.
4. Save the file and close the editor window.

Now you can re-simulate your model and look at the differences. Note that the sin and cos outputs from the NCO Symbol now have the type sfix18_En17(4). The (4) indicates that these wires actually represent a bundle of four parallel buses. This is modeled in Simulink as the vector width of the wire. Similarly, the constant input wires also have width (4).

The NCO Symbol also shows that it is a 24-channel NCO as shown in Figure 3–18.

**Figure 3–18. Updated NCO Symbol**
Running your model again shows similar behavior as before, except with different frequencies. At this point, it is worth looking at the raw signals emerging from the NCO block (Figure 3–19).

**Figure 3–19.** Signals Output from the NCO Block

There are four traces in Figure 3–19:

- **sin and cos.** These show the sines and cosines for the four wires emerging from the NCO. These are displayed in four different colors and all differ as expected because each channel has a different frequency.

- **The valid signal** is constantly high indicating that each value produced is valid.

- **The channel output** is a count running from 0 to 5 and repeating. This indicates the TDM slot that the data refers to.

Taken together, you can see how 24 channels of sine are represented on the sin output. Wire 0 of the 4 element vector has channels 0 to 5, wire 1 has channels 6 to 11, and so on. On each of those wires, the channels are in the order 0, 1, 2, 3, 4, 5 and you can use the channel value to determine which cycle to examine.

For example, if you want to locate channel 14, it is on the third element of the vector and corresponds to channel indicator 2.
To find the data for a particular channel, you must sample the correct element on the correct phase of the clock as indicated by the channel. This is precisely what the Channel Viewer block does to display the channel. Should the need arise, it is easy to code up an equivalent function in RTL for interfacing, or ModelPrim blocks, or in MATLAB m-code for analyzing captured data. This mapping is preserved into the hardware when it can be seen in simulation and in the device.

**Increasing the Spurious Free Dynamic Range**

There are several interrelated factors including the output precision of the NCO, the exact frequencies used, and the NCO hardware, that determine the spurious free dynamic range (SFDR). These factors can be analyzed in several ways, both theoretically and empirically.

The DSP Builder Advanced NCO Block is configured to automatically increase its precision based on the width of the output type specified. A worst case estimate gives approximately 6dB per bit, so 18 bits give around 108dB of SFDR. In practice, the SFDR can exceed this value, and you can perform your own analysis. Suppose you want to increase the SFDR of the current example by 20dB. You must add another 4 bits of precision to the output (4 × 6dB = 24dB) as described in the following steps:

1. Double-click the NCO block to open the **Function Block Parameters** dialog box and change the **Output data type** to `sfix(22)`.
2. If you want the result to stay in the range (1.0 to 1.0), change the **Output scaling value** to $2^{-21}$.
3. Close the **Function Block Parameters** dialog box and save your design.

Re-run the simulation to see the improved noise margins (Figure 3–20).

**Figure 3–20.** Improved SFDR Noise Margins
The Fibonacci Design

This tutorial shows how to build a simple model that generates a Fibonacci sequence using blocks from the ModelPrim library.

The Fibonacci sequence is the sequence of numbers that is created when you add 1 to 0 then successively add the last two numbers to get the next number:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...

Each primitive block in the model is parameterizable. When you double-click a block in the model, a dialog box appears where you can enter the parameters for the block. Click the Help button in these dialog boxes to view Help for a specific block.

The instructions in this tutorial assume that the following conditions are met:

- You are using a PC running Windows XP.
- You are familiar with the MATLAB, Simulink, Quartus II, and ModelSim software and the software is installed on your PC in the default locations.
- You have basic knowledge of the Simulink software. For information about using the Simulink software, refer to the Simulink Help.

You can perform a walkthrough by using the demo_fibonacci.mdl model file that is provided in the <DSP Builder Advanced install path>/Examples/ModelPrim directory or you can create your own Fibonacci model.

Creating the Fibonacci Model

To create the fibonacci model, follow the instructions in the following sections.

Any block parameters that are not explicitly mentioned in the procedures can be left with their default values.

Create a New Model

To create a new model, perform the following steps:

1. Start the MATLAB software.
2. On the File menu, click New to create a new model file.
3. Click Save on the File menu in the new model window.
4. Browse to the directory in which you want to save the file. This directory becomes your working directory. This tutorial uses the working directory <DSP Builder Advanced install path>/Examples/my_fibonacci.
5. Type the file name for the new model into the File name box. This tutorial uses the name fibonacci.mdl.
6. Click Save.
Add Blocks from the ModelPrim Library

1. Click the MATLAB Start button .
2. Click Simulink, then Library Browser.
   
   You can also open Simulink by using the toolbar icon.
3. In the Simulink Library Browser, click DSP Builder Advanced Blockset and ModelPrim to view the blocks in the ModelPrim library.
4. Drag and drop a ChannelIn and a ChannelOut block into your model (the fibonacci window) using Figure 4–1 as a guide.

Figure 4–1. Fibonacci Subsystem

5. Drag and drop two Sample Delay blocks into your model.
6. Select both of the Sample Delay blocks and point to Format on the popup menu and click Flip Block to reverse the direction of the blocks.
7. Drag and drop Add and Mux blocks from the ModelPrim library into your model.
8. Connect the blocks as shown in Figure 4–1.
9. Double-click on the second Sample Delay block (SampleDelay1) to display the Function Block Parameters dialog box and change the Number of delays parameter to 2 (Figure 4–2 on page 4–3).
10. Double-click on the Add block to display the **Function Block Parameters** dialog box (Figure 4–3) and set the parameters shown in Table 4–1.

### Table 4–1. Parameters for the Add Block

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output data type mode</td>
<td>Specify via dialog</td>
</tr>
<tr>
<td>Output data type</td>
<td>ufix(120)</td>
</tr>
<tr>
<td>Output scaling value</td>
<td>$2^{-0}$</td>
</tr>
<tr>
<td>Number of inputs</td>
<td>2</td>
</tr>
</tbody>
</table>

The data type must be specified because this design contains loops and the type cannot be determined if one of the inherit data options is set.
11. Complete your design by dragging and dropping a SynthesisInfo block from the ModelPrim library.

**Create a Synthesizable Subsystem**

You have now created the functional part of the Fibonacci design but this needs to be moved into a subsystem that defines the synthesizable subsystem.

1. Select all the ModelPrim blocks in your Fibonacci model and click **Create Subsystem** on the Edit menu.

2. Double-click on the new subsystem block to open the new subsystem.

3. Rename the input and output ports as shown in Table 4–2.

<table>
<thead>
<tr>
<th>Default Port Name</th>
<th>New Port Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>In1</td>
<td>d_v</td>
</tr>
<tr>
<td>In2</td>
<td>d_c</td>
</tr>
<tr>
<td>In3</td>
<td>d_0</td>
</tr>
<tr>
<td>Out1</td>
<td>q_v</td>
</tr>
<tr>
<td>Out2</td>
<td>q_c</td>
</tr>
<tr>
<td>Out3</td>
<td>fib</td>
</tr>
</tbody>
</table>

4. Drag and drop a Device block from the DSP Builder Advanced Blockset Base Blocks library into your model (Figure 4–4).
Complete the Top-Level Model

Complete the top-level Fibonacci model by adding blocks from the Simulink Sources and Sinks libraries.

1. Drag and drop a Repeating Sequence Stair block and two Constant blocks from the Simulink Sources library into your model.

2. Double-click on the Repeating Sequence Stair block to display the Source Block Parameters dialog box.

3. Set the Vector of output values to [0 1 1 zeros(1,171)].' and the Sample time to 1 in the Main tab (Figure 4–5).

Figure 4–5. Repeating Sequence Stair Block Parameters

4. Click the Signal Attributes tab (Figure 4–6) and set an unsigned fixed point Output data type: ufix(1).

Figure 4–6. Repeating Sequence Stair Block Signal Attributes Parameters
5. Double-click on each Constant block to display the Source Block Parameters dialog box, click the Signal Attributes tab (Figure 4–7), and select uint8 (unsigned 8-bit integer) as the Output data type for both blocks.

Both Constant blocks should be set to the default constant value of 1.

**Figure 4–7.** Constant Block Signal Attributes Parameters

6. Drag and drop a Scope block from the Simulink Sinks library into your model.
7. Double-click on the Scope block, open the Scope Parameters dialog box and set the Number of axes to 3.
8. Connect up the blocks as shown in Figure 4–8.

**Figure 4–8.** Completed Fibonacci Model
9. Complete the top-level diagram by dragging and dropping the Control, Signals, Run ModelSim and, Run Quartus II, and Signals blocks from the DSP Builder Advanced Blockset Base Blocks library.

10. Double-click on the Signals block and change the Clock Margin to 260 (Figure 4–9).

Figure 4–9. Parameters for the Signals Block

For this tutorial, you can use the Control block with its default settings.

For more information about the Control and Signals block, refer to the “System Tutorial” on page 5–1, or to their block descriptions in the DSP Builder Advanced Blockset Reference Manual.

11. Save the Fibonacci model.

Simulating the Design in Simulink

Check your design by simulating your model in Simulink:

1. Set the simulation stop time to 30.0 and simulate your design by clicking on .

2. Double-click on the Scope block and click Autoscale in the scope to display the simulation results (Figure 4–10 on page 4–8).
Notice that the sequence on the fib output starts at 0, and increments to 1 when q_v and q_c are both high at time 21.0. It then follows the expected Fibonacci sequence incrementing through 0, 1, 1, 2, 3, 5, 8, 13 and 21 to 34 at time 30.0.

You can verify that the fib output continues to increment according to the Fibonacci sequence by simulating for longer time periods.

**Using Vector Types**

Modify the top-level Fibonacci model to generate vector signals. In this example a vector signal automatically generates the Fibonacci sequence, and the Fibonacci sequence multiplied by three.

1. Double-click on the Constant block (Constant1) connected to the d_0 Subsystem Port. Click on the Main tab and specify a vector of width 2 by setting the constant value to be [1,3] (Figure 4–11).
2. Simulate the design in Simulink. The Scope block output (when Autoscale is used) reveals two sets of sequence values, one incrementing at a rate 3 times faster than the other (Figure 4–12).

Using Complex Types

To use complex types:

1. Modify the top-level model to use complex types. Vector and complex types can be used independently; in this tutorial, they are used together). In this case, imaginary data containing twice and four times the Fibonacci sequence are automatically generated, along with the two sequences of real data previously generated.

2. Double-click on the same Constant block (Constant1) modified in the previous section. Click on the Main tab and modify the constant value to be [complex(1,2), complex(3,4)] (Figure 4–13).
Complex values may either be specified using the notation: (a+bi), or using the complex function: complex(a,b). The complex notation has the advantage that it treats values with zero imaginary component b, as complex.

Because the Simulink Scope block does not support complex values, the model needs some modification to split the complex signals into real and imaginary components before simulation is possible,

3. Drag and drop a Complex to Real-Imag block from the Simulink Math Operations library into the top-level diagram.

4. Double-click on the Scope block, open the Scope Parameters dialog box and set the Number of axes to 4.

Figure 4–14 shows Fibonacci Model using complex data.

Figure 4–13. Connect Blocks
5. Simulate the design in Simulink. The Scope block output (when Autoscale is used) reveals that the real data and imaginary components now contain the Fibonacci sequence multiplied by the expected constant factors.

Figure 4–15 shows scope using complex data.

![Figure 4–15. Scope Using Real Data](image)

Exploring the Generated Files

If the Generate Hardware option is turned on in the parameters for the Control block, then every time the simulation runs, the underlying hardware is synthesized, and VHDL is written out into the specified directory.

A directory structure is created that mirrors the structure of your model. The root to this directory can be provided as an absolute path name or as a relative path name. If a relative path name (such as ../rtl) is given, the directory structure is created relative to the MATLAB current directory.

For example, if the current directory is C:\my_projects\designs, the structure shown in Figure 4–16 on page 4–12 is created.
Figure 4–16. Generated Directory Structure for the NCO Demonstration Design

Table 4–3 lists the generated files for the Fibonacci Design.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtl directory</td>
<td></td>
</tr>
<tr>
<td>fibonacci.xml</td>
<td>An XML file that describes the attributes of your model.</td>
</tr>
<tr>
<td>fibonacci_entity.xml</td>
<td>An XML file that describes the boundaries of the system (used by Signal Compiler in designs that combine blocks from the standard and advanced blocksets).</td>
</tr>
<tr>
<td>rtl/fibonacci subdirectory</td>
<td></td>
</tr>
<tr>
<td>Control.do</td>
<td>Running this script compiles your design, loads it into ModelSim, and runs it for the same simulation time that your model ran for in Simulink.</td>
</tr>
<tr>
<td>Control.wav.do</td>
<td>This script loads the signals from your design into the ModelSim Wave window.</td>
</tr>
<tr>
<td>&lt;block name&gt;.xml</td>
<td>An XML file containing information about each block in the advanced blockset which is translated into HTML on demand for display in the MATLAB Help viewer.</td>
</tr>
<tr>
<td>fibonacci.vhd</td>
<td>This is the top-level testbench file. It may contain non-synthesizable blocks, and empty black boxes for Simulink blocks that are not fully supported.</td>
</tr>
<tr>
<td>fibonacci.add.do</td>
<td>This script loads the VHDL files in this subdirectory and in the subsystem hierarchy below it into the ModelSim project.</td>
</tr>
<tr>
<td>fibonacci.compile.do</td>
<td>This script compiles the VHDL files in this subdirectory and in the subsystem hierarchy below it into the ModelSim project.</td>
</tr>
<tr>
<td>fibonacci.add.tcl</td>
<td>This script loads the VHDL files in this subdirectory and in the subsystem hierarchy below it into the Quartus II project.</td>
</tr>
<tr>
<td>fibonacci.qip</td>
<td>This file contains all the assignments and other information required to process the fibonacci model in the Quartus II software. The file includes a reference to the .qip file in the subsystem hierarchy.</td>
</tr>
<tr>
<td>&lt;block name&gt;.vhd</td>
<td>A VHDL file is generated for each component in your model.</td>
</tr>
<tr>
<td>fibonacci_Subsystem_entity.xml</td>
<td>An XML file that describes the boundaries of the subsystem as a block box (used by Signal Compiler in designs that combine blocks from the standard and advanced blocksets).</td>
</tr>
<tr>
<td>Subsystem.xml</td>
<td>An XML file that describes the attributes of the subsystem.</td>
</tr>
<tr>
<td>*.stm</td>
<td>Stimulus files.</td>
</tr>
<tr>
<td>safe_path.vhd</td>
<td>Helper function that ensures a pathname is read correctly in the Quartus II software.</td>
</tr>
<tr>
<td>safe_path_msim.vhd</td>
<td>Helper function that ensures a pathname is read correctly in ModelSim.</td>
</tr>
</tbody>
</table>
Table 4–3. Generated Files for the Fibonacci Design

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtl\fibonacci\Subsystem subdirectory</td>
<td></td>
</tr>
<tr>
<td>*.stm</td>
<td>Generated stimulus files for ModelSim.</td>
</tr>
<tr>
<td>Device.xml</td>
<td>An XML file containing information about the Device block which is translated into HTML for display in the MATLAB Help viewer.</td>
</tr>
<tr>
<td>*.vhd</td>
<td>This is the VHDL file created for the subsystem. It is generated as plain VHDL text that you can use in any design flow.</td>
</tr>
<tr>
<td>fibonacci_Subsystem_atb.do</td>
<td>Script that loads the subsystem automatic test bench into ModelSim.</td>
</tr>
<tr>
<td>fibonacci_Subsystem_atb.wav.do</td>
<td>Script that loads signals for the subsystem automatic test bench into ModelSim.</td>
</tr>
<tr>
<td>Subsystem.add.do</td>
<td>This script loads the VHDL files in this subdirectory into the ModelSim project.</td>
</tr>
<tr>
<td>Subsystem.add.tcl</td>
<td>This script loads the VHDL files in this subdirectory into the Quartus II project.</td>
</tr>
<tr>
<td>Subsystem.compile.do</td>
<td>This script compiles the VHDL files in this subdirectory into the ModelSim project.</td>
</tr>
<tr>
<td>Subsystem.qip</td>
<td>This file contains all the assignments and other information required to process the subsystem in the Quartus II software.</td>
</tr>
<tr>
<td>Subsystem.sdc</td>
<td>Design constraint file for TimeQuest timing analyzer support.</td>
</tr>
<tr>
<td>Subsystem.tcl</td>
<td>You can use this Tcl script to setup a Quartus II project.</td>
</tr>
<tr>
<td>Subsystem_hw.tcl</td>
<td>A Tcl script that is used to load the generated hardware into SOPC Builder.</td>
</tr>
</tbody>
</table>

Simulating the RTL

Verify that the same results are given when you simulate the generated RTL by clicking on the Run ModelSim block (Figure 4–17).

Figure 4–17. Fibonacci Sequence in the ModelSim Wave Window
Compiling with the Quartus II Software

You can load your design in the Quartus II software by double-clicking on the Run Quartus II block and compile your design by clicking Start Compilation on the Processing menu. Figure 4–18 shows the Quartus II Fitter Summary report.

Figure 4–18. Quartus II Compilation Fitter Summary for the Fibonacci Design

You can instantiate a DSP Builder Advanced primitive subsystem into an SOPC Builder design by using similar procedures to those described for the ModelIP Tutorial in “Instantiating the Design in SOPC Builder” on page 3–14.
System Level Design

FPGAs are powerful devices for performing DSP functions with speeds of up to 100 GMACs per second available. Achieving this level of design performance using RTL is a time consuming process, often negating the overall cost benefits available.

The Altera DSP Builder Advanced Blockset provides high level synthesis technology that bridges the gap between the application domains and programmable hardware. The blockset is parameterized in ways that are accessible to DSP experts, and lets you focus on your design goals, rather than the implementation details.

The DSP flow integrated with Simulink and MATLAB allows easy parameterization of designs and automatic generation of technology independent RTL which targets the required system clock speed.

Intermediate frequency (IF) modem designs are characterized by multi-channel, multi-rate filter lineups. The filter lineup is often programmable from a host processor, and has stringent demands on DSP performance and accuracy.

This tutorial shows how to build a digital down converter (DDC). The tutorial can be completed in approximately 30 minutes, and yields a high performance design running at over 300MHz in a Stratix II device. The design is very efficient and at under 4000 logic registers gives a low cost per channel.

The Digital Down Converter Design

This example design uses NCO/DDS, mixer, CIC, and FIR filter ModellIP components to build a 16-channel programmable DDC suitable for use in a wide range of radio applications.

To open the model, type the following command in the MATLAB window:
```
demo_ddc
```

The top-level model is shown in Figure 5–1 on page 5–2, which shows the testbench components while the design itself is contained in the `DDCChip` subsystem.

The design is built in the following stages:
1. The design is modeled using fixed-point types in Simulink
2. The design model is simulated in Simulink
3. The generated RTL is simulated in ModelSim
4. The RTL is synthesized and fitted in the Quartus II software
The entire design for the DDC is shown in Figure 5–1 (the testbench) and Figure 5–2 (the DDCChip subsystem), which show just how few components are required to build a complex, production ready system.

**Figure 5–1.** Testbench for the DDC Design

The top-level testbench includes integration blocks for ModelSim and the Quartus II software, Control and Signals blocks, and some Simulink blocks to generate source signals and visualize the output. Note that the full power of the Simulink blocksets is available for use.

**Figure 5–2.** DDCChip Subsystem

The DDCChip subsystem block contains the NCO and mixer, decimate by 16 CIC filter, and two decimate by 4 FIR odd-symmetric filters, one with length 21, the other length with 63. These blocks form the lowest level of the design hierarchy. The other blocks in this subsystem perform a range of rounding and saturation functions. They also allow dynamic scaling to be performed.

There is also a device block that specifies the target FPGA.
Signals Block

The starting point for planning a design is the system clock frequency. The first requirement is to provide a relation between the sample rates and the system clock. This step tells the synthesis engines how much folding or time sharing to perform. Increasing the system clock permits more folding, and therefore typically results in more resource sharing, and a smaller design.

The second reason you need a system clock rate is so that the synthesis engines know how much to pipeline the logic. For example, by considering the device and speed grade, the synthesis tool can calculate the maximum length that an adder can have. If this length is exceeded, the adder is pipelined and the whole pipeline is adjusted to compensate. This typically results in a small increase in logic size, that is usually more than compensated for by the decrease in logic size through increased folding.

The clock and reset names, along with the system clock frequency are specified in the Signals block as shown in Figure 5–3. The bus clock or FPGA internal clock for the memory-mapped interfaces can be run at a lower clock frequency. This lets the low speed operations such as coefficient update be moved completely off the critical path.

Figure 5–3. Configuration of the System Clock and Reset Signals in the Signals Block

![Block Parameters: Signals](image)

The clock frequency, clock margin, and bus clock frequency values in this design are specified using the MATLAB workspace variables `ClockRate` and `ClockMargin` which can be edited by double-clicking on the Edit Params block.
Control Block

The Control block (Figure 5–4) controls the whole advanced blockset environment. It examines every block in the system, controls the synthesis flow, and writes out all RTL and scripts. A single control block must be present at the top-level of the model.

Figure 5–4. Configuration of the Environment in the Control Block

For full details of each parameter, refer to their block descriptions in the DSP Builder Advanced Blockset Reference Manual.

In this design, hardware generation is enabled to create RTL. The RTL and associated scripts are all placed in the directory ../rtl. This is a relative path based on the current MATLAB directory. Automatic self-checking testbenches are created. This saves the data captured from a Simulink simulation to build testbench stimulus for each block in your design. Scripts are generated to run these simulations.

There are many memory-mapped registers in the design such as filter coefficients and control registers for gains. These are all accessed through a memory port that is automatically created at the top-level of your design. All address decode and data multiplexing logic is created automatically. A memory map is generated in XML and HTML that you can use to understand the design. You can access this memory map by clicking Help in the Control Block Parameters dialog box after your design has been simulated. The address and data widths are set to 8 and 32 in the design.
The threshold values are used to control the hardware generation, and are described in the reference manual. They control the trade-offs between hardware resources, such as hard DSP blocks or soft logic element implementations of multipliers. This means that you can perform resource balancing for your particular design needs with a few top-level controls.

**Source Blocks**

The Simulink/MATLAB environment enables you to create any required input data for your design. In the DDC design, sine wave or random noise generators can be selected using the manual switches. A simple six-cycle sine wave is encoded as a table in a Repeating Sequence Stair block from the Simulink Sources library. This sine wave has been set to a frequency that is close to the carrier frequencies specified in the NCOs, allowing you to see some signals being decoded by the filter lineup. VHDL for each block is created as part of the testbench RTL.

**Sink Blocks**

Simulink Sink library blocks are used to display the results of the DDC simulation. The Scope block displays the raw output from the DDC design. The design has time division multiplexed (TDM) outputs and all the data is presented as data, valid and channel signals.

At each clock cycle, the value on the data wire either carries a genuine data output, or data that can be safely discarded. The valid signal differentiates between these two cases. If the data is valid, then the channel wire identifies the channel to which the data belongs. Thus you can use the valid and channel wires to filter the data as required. The Channel Viewer block automates this task and is configured to decode 16 channels of data and to output channels 0 and 15. These channels are then decimated by the same rate as the whole filter line up and passed to a spectrum scope block (OutSpectrum) that examines the behavior in the frequency domain.

**Tool Interface Blocks**

There are three additional blocks that provide a convenient way to interact with tools:

- The Edit Params blocks allows you to edit the script setup_demo_ddc.m, which sets up the MATLAB variables that are used to configure your model. This script is called using the MATLAB model properties callback mechanism.

  - The PreloadFcn callback uses this script to setup the parameters when your model is opened and the InitFcn callback re-initializes your model to take account of any changes when the simulation is started.

- The Run ModelSim block starts ModelSim using a script generated during the hardware generation process, and simulates the testbench and design contents. This process compiles all the VHDL files, adds signals to the ModelSim Wave window and simulates for the same time as the Simulink model was run.

- The Run Quartus II block opens the Quartus II software and adds all your design files to a project. You can then compile your design for your chosen device and clock rate using the Quartus II software.
You can edit the parameters in the `setup_demo_ddc.m` script by double-clicking on the `Edit Params` block to open the script in the MATLAB text editor (Figure 5–5).

**Figure 5–5. MATLAB Workspace Variables for the DDC Design**

```matlab
% File: setup_demo_ddc.m
% Description: Script that sets variables in the MATLAB workspace to configure demo_ddc model
% Revision: $Id: setup_demo_ddc.m,v 1.2 2006/11/01 15:48:11 jonah Exp$

FreqCount=4;
ChanCount=FreqCount*4, % Freqcount * MainDiv * I/Q = 16
SampleRate=61.44;
ClockRate=SampleRate*4;
ClockMargin=0.0;
Perio=ClockRate / SampleRate;
SampleTime = 1;
%SampleTime = 1 / (ClockRate * 1e6); % Uncomment this line to simulate with real world time

% Memory map, the following are the base addresses for various programmable % components in the ddc demo
DDC_NCO_PHASE_INCR = hex2dec(000); % NCO phase increment register
DDC_COMP_FIR_COEFS = hex2dec(000); % CIC compensating FIR coefficients
DDC_DECIM_FIR_COEFS = hex2dec(030); % Decimating FIR coefficients

% This is the address of the gains register, it has a number of parts for % the different gains in the system
DDC_GAINS = hex2dec(CfE);
DDC_BYPASS_CIC = hex2dec(0ff); % CIC bypass register
disp(['Parameters: ' ...
       'ChanCount = ' num2str(ChanCount) ', ' ...
       'SampleRate = ' num2str(SampleRate) ', ' ...
       'ClockRate = ' num2str(ClockRate) ', ' ...
       'ClockMargin = ' num2str(ClockMargin) ', ' ...
       'SampleTime = ' num2str(SampleTime) ']);

% Desired Parameters
% WARNING - DO NOT MODIFY!!!
FreqWireCount=cell(FreqCount/Perio);
FreqCycleCount=cell(FreqCount/FreqWireCount);
```

The script sets up MATLAB workspace variables. One Key variable is the `SampleRate`, which is set to 61.44MHz. This is typical of a CDMA system, and represents a quarter of the system clock rate that the FPGA runs at. This lets you time division multiplex four signals onto any given wire.

**DDCChip Subsystem**

The DDCChip subsystem contains a Device block. This block labels this level of design hierarchy that is compiled onto the FPGA. VHDL is generated for all levels of the hierarchy, but this level has additional script files generated that build a project in the Quartus II software. The Device block sets the FPGA family, device and speed grade. The family and speed grade are used to optimize the hardware. In combination with the requested clock frequency, the device determines the degree of pipelining.
The following sections describe the blocks contained in the DDCChip subsystem. Note that there are, in general, the following three types of block:

- The grey blocks are ModelIP blocks. These represent functional IP such as black box filters, NCOs, and mixers.
- The blue blocks are processor visible registers.
- The black and white blocks are Simulink blocks.

You can mix and match blocks in a natural way. Each stage is described below in some detail starting from the beginning of the pipeline. This level of detail helps you to understand some of the subtleties involved in building your own designs.

Figure 5–6 shows the first part of the data path.

**Figure 5–6. DDCChip Data Path (NCO and Mixer)**

The inputs, NCO, and mixer stages are shown with Simulink signal formats turned on.

**Primary Inputs**

The primary inputs to the hardware are two parallel data signals (DataInMain and DataInDiversity), a channel signal (DataChan), and a valid signal (DataValid). The parallel data signals represent inputs from two antennas. Note that they are of type `sfix14_13` which is a Simulink fixed point type of total width 14 bits. The type is signed, and there are 13 bits of fraction. This is a typical number format generated from an analog-to-digital converter.
The data channel DataChan is always an 8-bit unsigned integer (uint8) and the top bits are synthesized away if not used. The valid signal DataValid indicates when real data is being transmitted. The first rising edge of the valid signal starts operation of the first blocks in the chain. As the first blocks start producing outputs, their valid outputs are used to start the next blocks in the chain. This mechanism ensures that filter chain starts up in a coordinated way without having a global controller that has to know about the latencies of each block. This is important because the actual latencies of the blocks may change based on the clock frequency and FPGA selection.

**Merge Multiplexer**

The ModelIP blockset supports vectors on its input and output data wires. This ensures that a block diagram is scalable when, for example, changing channel counts and operating frequencies. The merge multiplexer (DDCMerge1) takes two individual wires and combines them into a vector wire of width 2. Notice that although this is a Simulink Mux block, it does not actually perform any multiplexing in the hardware sense—it should be thought of as a vectorizing block. You can verify this by examining the RTL that is generated, and noticing that it contains just wires.

**NCO**

The NCO block generates sine and cosine waveforms to a given precision. These waveforms can be thought of as representing a point in the complex plane rotating around the origin at a given frequency. This waveform is then multiplied by the incoming data stream to obtain the data from the transmitted signal.

**Figure 5–7. Parameters for the NCO Block**
Four frequencies are generated, because the vector in the **Phase Increment and Inversion** field is of length 4.

The NCO block is configured to produce a signed 18-bit value with 17 bits of fraction. The internal accumulator width is set to 24 bits. This internal precision affects the spurious free dynamic noise (SFDN). The initial frequencies for the simulation are specified as phase increments. Because, the phase accumulator bit width is $2^{24}$, one complete revolution of the unit circle corresponds to a value of $2^{24}$. Dividing this number by 5.95, means that 5.95 cycles are required to perform one complete rotation. That is, the wavelength of the sine and cosine produced are 5.95 cycles. Because the sample rate is 61.44MHz, this corresponds to a frequency of 61.44/5.95 which equals 10.32MHz.

The input frequency used in the testbench rotates every 6 cycles for a frequency of 61.44/6=10.24MHz. Therefore, because these signals are mixed together you can expect to recover the difference of these frequencies which is 0.08MHz, or 80KHz, which falls in the low pass filters pass bands.

There is more information about how to configure the NCO block in the “ModellIP Tutorial” on page 3–1.

Note that the phase values are exposed through a memory-mapped interface at the address specified by the variable **DDC_NCO_PHASE_INCR**, which is set to address 0x0000 in the setup script. After the simulation has run, a report file is generated that confirms and documents that address map. This can be accessed by right-clicking on the Control block in the top-level model and clicking Help (Figure 5–8).

**Figure 5–8.** Memory Interface Help for the Control Block

**Memory Interface**

**demo_ddc**

NCO Phase Increment Registers (sin inversion@MSB cos inversion@MSB-1) for demo_ddcDDCChipNCO

<table>
<thead>
<tr>
<th>NCO Phase Increment Register 0 (sin@MSB cos@MSB-1)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Bits (Width)</td>
<td>25, 0 (26)</td>
</tr>
<tr>
<td>Reset</td>
<td>269700</td>
</tr>
<tr>
<td>Word Address</td>
<td>0x0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NCO Phase Increment Register 1 (sin@MSB cos@MSB-1)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Bits (Width)</td>
<td>25, 0 (26)</td>
</tr>
<tr>
<td>Reset</td>
<td>269700</td>
</tr>
<tr>
<td>Word Address</td>
<td>0x1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NCO Phase Increment Register 2 (sin@MSB cos@MSB-1)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Bits (Width)</td>
<td>25, 0 (26)</td>
</tr>
<tr>
<td>Reset</td>
<td>269700</td>
</tr>
<tr>
<td>Word Address</td>
<td>0x2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NCO Phase Increment Register 3 (sin@MSB cos@MSB-1)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Bits (Width)</td>
<td>25, 0 (26)</td>
</tr>
<tr>
<td>Reset</td>
<td>269700</td>
</tr>
<tr>
<td>Word Address</td>
<td>0x3</td>
</tr>
</tbody>
</table>

For each register, the name, width, reset value, and address are reported. This report collates all the registers from your design into a single location.
You can view the estimated results for this NCO configuration in the **Results** tab of the dialog box, as shown in **Figure 5–9**.

**Figure 5–9. Estimated Results for the NCO Block**

Based on the selected accumulator width and output width, an estimated SFDR, and accumulator precision is calculated. This can be verified in a separate testbench, perhaps using `demo_nco.mdl` as a starting point.

**Mixer**

The **Mixer** block performs the superheterodyne operation by multiplying each of the two received signals (DataInMain and DataInDiversity) by each of the four frequencies. This produces eight complex signals or 16 scalar signals (the 16 channels in the DDC design).

The mixer requires sufficient multipliers to perform this calculation. The total number of real\*complex multiplications required for each sample is 2 signals × 4 frequencies = 8. Eight real\*complex multiplies require 8 × 2 = 16 scalar multiplications. Because this processing is spread over four cycles (the TDM factor given by the ratio of clock rate to sample rate), four physical multipliers are required.

This result is confirmed in the report file shown in **Figure 5–10**, which can be viewed by right-clicking the **Mixer** block (after running a simulation) and clicking **Help**. The resource utilization section reports that four 18×18 multipliers are used.
The Help report also lists the input and output ports that are created for this block, along with the data width and brief description. Notice that the vector inputs are suffixed with 0 and 1 to implement the vector. This list of signals corresponds to the signals in the VHDL entity.

Notice that the results for the mixer are provided as separate in phase and quadrature outputs—each is a vector of width 2. The remaining operations are performed on both the I and Q signals, so that they can be combined with another Simulink multiplexer to provide a vector of width 4. This is carrying the 16 signals, with a TDM factor of 4. At this point the channel counts count 0, 1, 2, 3, 0, 1, ....
Mixer Scale Block

At this point in the data path, the data width is 32 bits representing the full precision output of multiplying a 14-bit data signal with an 18-bit sine/cosine signal.

The data width needs to be reduced to a lower precision to pass on to the remaining filters. This reduces the resource count considerably, and does not cause significant information loss. The Scale3 block performs a shift-round-saturate operation to achieve this. The shift is usually a 1 or 2 bit shift that can be set to adjust the gain in your design at run time.

The setup is usually determined by a microprocessor, which writes to a register to set the shift amount. In this design, this is accomplished by using a RegField block (Mixer_Scaling_Register). This block behaves like a constant in the Simulink simulation, but in hardware the block performs as a processor-writable register that is initialized to the value used in your model (Figure 5–11).

Figure 5–11. Parameters for the Mixer_Scaling_Register Block

This parameterization results in a register mapped to address DDC_GAINS, which is a MATLAB variable specified in the setup_demo_ddc.m script.

The register is writable from the processor, but not readable.
The register produces a 2-bit output of type \texttt{ufix(2)}—an unsigned fixed point number. The scaling is $2^{-0}$ so is, in effect, a 2-bit unsigned integer. These 2 bits are mapped into bits 0 and 1 of the word (another register may use other bits of this same address). The initial value for the register is set to 0. A description of the memory map is provided in the updated Help for the block. Sometimes, an explicit sample time may be needed by Simulink, but you can use the default -1 for this tutorial.

The 2-bit unsigned integer is fed to the \texttt{Scale3} block (Figure 5–12). This block has a vector of width 4 as its data input. The \texttt{Scale3} block builds a vector of 4 internal scale units. These are not visible through the user interface, but can be seen in the resource report by clicking \texttt{Help} on the \texttt{Scale3} block.

The block produces four outputs which are presented at the output as a vector of width 4. The order in the vector is preserved. Notice that it is possible to create quite a large block of hardware by passing many channels through a ModelIP block. Note also that the exception output of the scale block provides signals to say when saturation occurs. In this design they are not required, so are simply terminated.

**Figure 5–12.** Parameters for the Scale3 Block

The output format is set to be 16-bit signed with 15 bits of fraction. The \texttt{Unbiased} rounding method is selected. This method is sometimes known as convergent rounding or round-to-even and is typically used to avoid introducing a DC bias.

The saturation method uses \texttt{Symmetric} rounding which clips values to within +0.9999 to -0.9999 (for example) rather than clipping to -1. Again this avoids introducing a DC bias.

The number of bits to shift is a vector of values that is indexed by the scaling register block (\texttt{Mixer_Scaling_Register}). Because this is a vector of 4 values, a 2-bit input is required.
An input of 0 uses the 0th value in the vector (address 1 in Simulink), and so on. Therefore, in this example `inout0` shifts by 0 and the result at the input has the same numerical range as the input. An input of 1 shifts left by 1, and so multiply the input value by 2, thus increasing the gain.

**DecimatingCIC and Scale Blocks**

This part of the data path comprises a decimating cascaded integrator comb (CIC) filter and scale block and is shown in Figure 5–13.

**Figure 5–13.** DDCChip Data Path (DecimatingCIC and Scale Blocks)

There are two main blocks here—the DecimatingCIC and the Scale block. The CIC Filter is configured by double clicking on the DecimatingCIC block. The options are shown in Figure 5–14 on page 5–15.

The input sample rate is still the same as the data from the antenna, and is specified by the `SampleRate` variable. The number of channels, `ChanCount`, is a variable set to 16. The CIC filter is configured to have 5 stages, and perform decimation by a factor of 16. 1/16 is entered in the dialog box to indicate that the output rate is 1/16th of the input sample rate. The CIC parameter differential delay controls how many delays are used in each CIC section. This is nearly always set to 1.

The CIC has no registers to configure, so there are no memory map elements.

The input data is a vector of four elements, so the Decimating CIC is internally built from four separate CICs, each operating on four channels. Because the data rate at the output is reduced by the decimation behavior, all 16 data samples (now at 61.44/16 MSPS each channel) can fit onto 1 wire.
The DecimatingCIC block multiplexes the results from each of the internal CIC filters onto a single wire. That is, four channels from vector element 1, followed by the four channels from vector element 2. The data is packed together onto a single TDM wire. Data is active for 25% of the cycles because the aggregate sample rate is now 61.44 MSPS * 16 channels / 16 decimation = 61.44MSPS and the clock rate for the system is 245.76MHz.

This behavior is shown in Figure 5–15, which is the CIC_All_Scope found in the CIC_Scopes subsystem. Bursts of data occur, with 16 contiguous samples followed by a gap. Each burst is tagged with the valid signal. Also the channel indicator shows that the channel order is 0..15.
This is confirmed in the updated Help for the DecimatingCIC block (Figure 5–16).

**Figure 5–16.** Help Report for the DecimatingCIC Block

demo_ddc/DDCChip/DecimatingCIC

- Created using Altera DSP Builder Advanced software
- Written on Tue Jul 01 15:26:19 2008
- Decimating CIC Filter Version: $Revision: 1.20 $
- Number of physical input buses / integrators: 4
- Number of physical output buses / combs: 1
- Calculated output bitwidth: 36
- Calculated stage bitwidths: 36 36 36 36 36 36 36 36
- Gain: 1048576
- Integrator section utilization: 4 cycles used of 4 available (100.00%)  
  Comb section utilization: 16 cycles used of 64 available (25.00%)
- Latency is 13

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimation Rate</td>
<td>10</td>
<td>The decimation rate of the filter</td>
</tr>
<tr>
<td>Number of stages</td>
<td>5</td>
<td>Number of integrator and comb sections</td>
</tr>
<tr>
<td>Differential Delay</td>
<td>1</td>
<td>Differential Delay</td>
</tr>
<tr>
<td>Number of channels</td>
<td>10</td>
<td>Number of channels</td>
</tr>
<tr>
<td>Clock Rate</td>
<td>245.76MHz</td>
<td>The frequency of the clock when running in hardware</td>
</tr>
<tr>
<td>Input Rate</td>
<td>61.44MHz</td>
<td>The sample rate of each channel</td>
</tr>
</tbody>
</table>

**Port Interface**

<table>
<thead>
<tr>
<th>Port</th>
<th>Direction</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xIn_v</td>
<td>in</td>
<td>1</td>
<td>Qualifying signal to indicate validity of data signals</td>
</tr>
<tr>
<td>xIn_o</td>
<td>in</td>
<td>8</td>
<td>Qualifying signal to indicate channel of data signals</td>
</tr>
<tr>
<td>xIn_0</td>
<td>in</td>
<td>10</td>
<td>Data signal</td>
</tr>
<tr>
<td>xIn_1</td>
<td>in</td>
<td>16</td>
<td>Data signal</td>
</tr>
<tr>
<td>xIn_2</td>
<td>in</td>
<td>16</td>
<td>Data signal</td>
</tr>
<tr>
<td>xIn_3</td>
<td>in</td>
<td>16</td>
<td>Data signal</td>
</tr>
<tr>
<td>bypass_i</td>
<td>in</td>
<td>1</td>
<td>General Purpose Input</td>
</tr>
<tr>
<td>xOut_v</td>
<td>out</td>
<td>1</td>
<td>Qualifying signal to indicate validity of data signals</td>
</tr>
<tr>
<td>xOut_lo</td>
<td>out</td>
<td>8</td>
<td>Qualifying signal to indicate channel of data signals</td>
</tr>
<tr>
<td>xOut_o</td>
<td>out</td>
<td>36</td>
<td>Data signal</td>
</tr>
</tbody>
</table>

**Input Data Format (Repeats every 4 clock cycles)**

```
<co0><co1><co2><co3>
<co4><co5><co6><co7>
<co8><co9><co10><co11>
<co12><co13><co14><co15>
```

**Output Data Format (Repeats every 64 clock cycles)**

```
<co0><co1><co2><co3><co4><co5><co6><co7><co8><co9><co10><co11><co12><co13><co14><co15><---<```

**Resource Utilization**

<table>
<thead>
<tr>
<th>Component</th>
<th>Type</th>
<th>LUT4s</th>
<th>Muls</th>
<th>Memory bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td></td>
<td>1226</td>
<td>0</td>
<td>3628</td>
</tr>
</tbody>
</table>
The number of input integrator sections is 4, and the number of output comb sections is 1. The lower data rate reduces the size of the overall group of 4 CICs. The updated Help page also reports the gain for the DCIC to be 1,048,576 or approximately $2^{20}$. The comb section utilization confirms the 25% calculation for the TDM factor. The updated Help page also shows how the four channels of input data are combined on a single output data channel.

The Scale block reduces the output bit width of the CIC results, as before with parameters set as shown in Figure 5–17.

**Figure 5–17. Parameters for the Decimating CIC Scale Block**

![Scale Block Parameters](image)

Note that in this case, no variable shifting operation is required, so the shift input can simply be tied to 0 using a Simulink constant. However, because the gain through the DecimatingCIC block is approximately $2^{20}$ division of the output data is performed by entering a scalar value -20 for the **Number of bits to shift left** in the dialog box.

A scalar rather than a vector value is entered to indicate that the scaling is static.

**Decimating FIR Blocks**

The last part of the data path (Figure 5–18 on page 5–18) comprises two decimating finite impulse response (FIR) blocks (DecimatingFIR1 and DecimatingFIR2) and their corresponding scale blocks (Scale1 and Scale2).

These two stages are very similar, with the first filter typically used to compensate for the undesirable pass band response of the CIC filter, and the second FIR used to fine tune the response required by the waveform specification.
The first decimating FIR decimates by a factor of 4. (Figure 5–19).

Figure 5–19. Parameters for the DecimatingFIR1 Block
The input rate per channel is the output sample rate of the decimating CIC. This is 16 times lower than the raw sample rate from the antenna.

Note that any MATLAB expression can be entered here, so the 16 can be extracted out as a variable to provide additional parameterization of the whole design.

This filter performs decimation by a factor of 4 and the calculations reduce the size of the FIR filter. There are 16 channels to process and the coefficients are symmetrical.

The Coefficients field contains information that is passed as a MATLAB fixed point object (fi) which contains the data itself, and also the size and precision of each coefficient. This is achieved by specifying an array of floating point objects in the square brackets to the constructor. The length of this array is the number of taps in the filter. At the end of this expression, the numbers 1, 16, 15 indicate that the fixed point object is signed, and has 16-bit wide elements of which 15 are fractional bits.

For more information about fi objects, refer to the MATLAB Help.

This simple design uses a low pass filter with the response shown in Figure 5–20. In a real design, more careful generation of coefficients may be necessary.

Figure 5–20. DecimatingFIR1 Magnitude and Phase Responses
The output of the FIR filter fits onto a single wire, but because the data is reduced further, there is a longer gap between frames of data.

A report on the generated FIR filter can be accessed from the Help page. The part of this report that covers the basic parameters and some derived parameters (including output bit width, folding factor, latency, sample rates and symmetry) is shown in Figure 5–21.

Figure 5–21. Help Report for the DecimatingFIR1 Block (Beginning)

demo_ddc/DDCChip/DecimatingFIR1

This help page relates to the specific instance 'DecimatingFIR1' of the FIRD block

Help on core functionality of FIRD

- Created using Altera DSP Builder Advanced software
- Written on Tue Feb 19 11:35:41 2008
- Decimating Filter Version: $Revision: 1.90$
- Name: demo_ddc/DDCChip/DecimatingFIR1
- Number of physical input buses: 1
- Number of physical output buses: 1
- Calculated Bitwidth of output stage: 36
- Number of different phases: 1
- Implementation Folding (per phase): 11
- Filter Utilization (per phase): -16/64 (-25.00%)
- Tap Utilization (per phase): 11/11 (100.00%)
- Latency is 9

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Clock</td>
<td>245.76MHz</td>
<td>The frequency of the clock when running in hardware</td>
</tr>
<tr>
<td>Clock Margin</td>
<td>0.00MHz</td>
<td>Additional Clock Margin</td>
</tr>
<tr>
<td>Input Rate</td>
<td>3.84MHz</td>
<td>The sample rate of each channel</td>
</tr>
<tr>
<td>Filter Length</td>
<td>21</td>
<td>Number of coefficients</td>
</tr>
<tr>
<td>Decimation</td>
<td>4</td>
<td>Decimation Rate</td>
</tr>
<tr>
<td>Channel Count</td>
<td>16</td>
<td>Number of channels</td>
</tr>
<tr>
<td>Symmetry</td>
<td>YES</td>
<td>Symmetry is used to reduce the hardware resources</td>
</tr>
</tbody>
</table>

You can scroll down in the Help page to view the port interface details. These match the hardware block, although the RTL has additional ports for clock, reset, and the bus interface.

The report shows that the input data format uses a single channel repeating every 64 clock cycles and the output data is on a single channel repeating every 256 clock cycles.

Details of the memory map are provided including the addresses required to set up the filter parameters using by an external microprocessor.

The Help page also shows the total estimated resources. This filter is estimated to use 338 LUT4s, 1 18×18 multiplier and 7844 bits of RAM.
The `Scale1` block that follows the `DecimatingFIR1` block performs a similar function to that described earlier for the `DecimatingCIC` block.

The `DecimatingFIR2` block performs a second level of decimation, in a very similar way to `DecimatingFIR1`. The main difference is that the coefficients are generated using a MATLAB function. This function (`fir1`) returns an array of 63 doubles representing a low pass filter with cut off at 0.22. This result can be wrapped in a `fi` object:

```matlab
design.computeFilterCoefficients();
fi(fir1(62, 0.22),1,16,15)
```

### Simulating the Design in Simulink

You can run the simulation in Simulink by typing the following command in the MATLAB window:

```matlab
sim('demo_ddc', 550000.0*SampleTime);
```

The `IScope` block (Figure 5–22) shows the first two channels (1 real and 1 complex for the first carrier) of data (magenta and yellow) as the input signals. The first trace shows the rapidly changing input signal generated in the testbench. The second signal shows the result of the mixer. This is essentially a slowly changing signal that contains the information to be extracted, plus a lot of high frequency residue. Applying the series of low pass filters and decimating results in the required data.

**Figure 5–22.** Simulation Results Shown in the IScope Block
Exploring the Generated Files

If the Generate Hardware option is on in the parameters for the Control block, then every time the simulation runs, the underlying hardware is synthesized, and VHDL is written out into the specified directory.

A directory structure is created that mirrors the structure of your model. The root to this directory can be provided as an absolute path name or as a relative path name. If a relative path name (such as ../rtl) is given, the directory structure is created relative to the MATLAB current directory. For example, if the current working directory is C:\my_projects\designs, the structure shown in Figure 5–23 is created.

Figure 5–23. Generated Directory Structure for the DDC Demonstration Design

There are separate subdirectories corresponding to each hierarchical level in your design.
Table 5–1 lists the generated files for the DDC design.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtl directory</td>
<td></td>
</tr>
<tr>
<td>demo_ddc.xml</td>
<td>An XML file that describes the attributes of your model.</td>
</tr>
<tr>
<td>demo_ddc_entity.xml</td>
<td>An XML file that describes the boundaries of the system (used by Signal Compiler in designs that combine blocks from the standard and advanced blocksets).</td>
</tr>
<tr>
<td>rtl\demo_ddc subdirectory</td>
<td></td>
</tr>
<tr>
<td>Control.do</td>
<td>Running this script compiles your design, loads it into ModelSim, and runs it for the same simulation time that your model ran for in Simulink.</td>
</tr>
<tr>
<td>Control.wav.do</td>
<td>This script loads the signals from your design into the ModelSim Wave window.</td>
</tr>
<tr>
<td>&lt;block name&gt;.xml</td>
<td>An XML file containing information about each block in the advanced blockset which is translated into HTML on demand for display in the MATLAB Help viewer.</td>
</tr>
<tr>
<td>demo_ddc.vhd</td>
<td>This is the top-level testbench file. It may contain non-synthesizable blocks, and may also contain empty black boxes for Simulink blocks that are not fully supported.</td>
</tr>
<tr>
<td>demo_ddc.add.do</td>
<td>This script loads the VHDL files in this subdirectory and in the subsystem hierarchy below it into the ModelSim project.</td>
</tr>
<tr>
<td>demo_ddc.compile.do</td>
<td>This script compiles the VHDL files in this subdirectory and in the subsystem hierarchy below it in the ModelSim project.</td>
</tr>
<tr>
<td>demo_ddc.add.tcl</td>
<td>This script loads the VHDL files in this subdirectory and in the subsystem hierarchy below it into the Quartus II project.</td>
</tr>
<tr>
<td>demo_ddc.qip</td>
<td>This file contains all the assignments and other information required to process the demo_ddc model in the Quartus II software. The file includes a reference to the .qip file in the DDCChip subsystem hierarchy.</td>
</tr>
<tr>
<td>&lt;block name&gt;.vhd</td>
<td>A VHDL file is generated for each component in your model.</td>
</tr>
<tr>
<td>demo_ddc_DDCChip_entity.xml</td>
<td>An XML file that describes the boundaries of the DDCChip subsystem as a block box (used by Signal Compiler in designs that combine blocks from the standard and advanced blocksets).</td>
</tr>
<tr>
<td>DDCChip.xml</td>
<td>An XML file that describes the attributes of the DDCChip subsystem.</td>
</tr>
<tr>
<td>*.stm</td>
<td>Stimulus files.</td>
</tr>
<tr>
<td>safe_path.vhd</td>
<td>Helper function that ensures a pathname is read correctly in the Quartus II software.</td>
</tr>
<tr>
<td>safe_path_msim.vhd</td>
<td>Helper function that ensures a pathname is read correctly in ModelSim.</td>
</tr>
<tr>
<td>rtl\demo_ddc&lt;subsystem&gt; subdirectories</td>
<td></td>
</tr>
<tr>
<td>&lt;subsystem&gt;_atb.do</td>
<td>Script that loads the subsystem automatic test bench into ModelSim.</td>
</tr>
<tr>
<td>&lt;subsystem&gt;_atb.wav.do</td>
<td>Script that loads signals for the subsystem automatic test bench into ModelSim.</td>
</tr>
<tr>
<td>&lt;subsystem&gt;&lt;block&gt;/*.hex</td>
<td>These are Intel format Hex files that are used to initialize the RAM blocks in your design for either Simulation or Synthesis.</td>
</tr>
<tr>
<td>&lt;subsystem&gt;.sdc</td>
<td>Design constraint file for Timequest support.</td>
</tr>
<tr>
<td>&lt;subsystem&gt;.tcl</td>
<td>You can use this Tcl file to setup a Quartus II project.</td>
</tr>
<tr>
<td>&lt;subsystem&gt;_hw.tcl</td>
<td>A TCI script that is used to load the generated hardware into SOPC Builder.</td>
</tr>
</tbody>
</table>
Simulating the RTL

Whenever hardware is generated, several support scripts are generated to make it easy to verify the hardware in RTL. The top-level .do file generated for ModelSim is named Control.do after the Simulink name of the Control block in your model. The easiest way to call this file is to double-click on the Run ModelSim block in your model. This opens a new ModelSim session and runs the script automatically for the same time period as specified in the Simulink run.

The script compiles any FPGA vendor libraries, compiles all the RTL generated for the project, opens a ModelSim Wave window, and adds signals to the Wave window.

The signals that are added to the Wave window are determined by the Signal View Depth parameter in the Control block. Where appropriate, the signals are automatically displayed in analog format as shown in Figure 5–24. Simulink scopes are used to identify the signals to display, therefore, to display a particular signal, just attach a scope, regenerate, and re-simulate.

Figure 5–24. Simulation Results in the ModelSim Wave Window for the DDC Demonstration Design
Compiling with the Quartus II Software

The most convenient way to start the Quartus II software is to double-click on the Run Quartus II block in the top-level model. This is equivalent to starting the Quartus II software manually and running the Tcl file.

```bash
cd ../rtl
source demo_ddc/DDCChip/DDCChip.tcl
```

After your design has been loaded, you can compile your design by clicking Start Compilation on the Processing menu.

For these parameters, you can see that the Quartus II software chooses a small device (Figure 5–25).

In many real production environments the generated RTL is wrapped in some hand coded RTL to perform detailed memory interfacing, high speed I/O, and interfacing to a processor. The generated RTL is structured to make this easy.

**Figure 5–25.** Quartus II Compilation Fitter Summary for the DDC Demonstration Design

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Quartus II Version</td>
<td>7.2 Build 147 09/05/2007 SJ Full Version</td>
</tr>
<tr>
<td>Revision Name</td>
<td>DDCChip</td>
</tr>
<tr>
<td>Top-level Entity Name</td>
<td>demo_ddc_DDCChip</td>
</tr>
<tr>
<td>Family</td>
<td>Stratix II</td>
</tr>
<tr>
<td>Device</td>
<td>EP2S15F484C3</td>
</tr>
<tr>
<td>Timing Models</td>
<td>Final</td>
</tr>
<tr>
<td>Logic utilization</td>
<td>29 %</td>
</tr>
<tr>
<td>Combinational ALUTs</td>
<td>2,536 / 12,480 (20 %)</td>
</tr>
<tr>
<td>Dedicated logic registers</td>
<td>3,247 / 12,480 (26 %)</td>
</tr>
<tr>
<td>Total registers</td>
<td>3247</td>
</tr>
<tr>
<td>Total pins</td>
<td>140 / 343 (41 %)</td>
</tr>
<tr>
<td>Total virtual pins</td>
<td>0</td>
</tr>
<tr>
<td>Total block memory bits</td>
<td>44,793 / 419,328 (11 %)</td>
</tr>
<tr>
<td>DSP block 9-bit elements</td>
<td>16 / 96 (17 %)</td>
</tr>
<tr>
<td>Total PLLs</td>
<td>0 / 6 (0 %)</td>
</tr>
<tr>
<td>Total DLLs</td>
<td>0 / 2 (0 %)</td>
</tr>
</tbody>
</table>

You can instantiate your DSP Builder Advanced subsystem into an SOPC Builder design by using similar procedures to those described for the ModelIP Tutorial in “Instantiating the Design in SOPC Builder” on page 3–14.
A. Latency Management

You may want to fix or constrain the latency after you have completed part of a design, for example on a ModelIP block or for a primitive subsystem. In other cases, you may want to limit the latency in advance. This allows future changes to other subsystems without causing undesirable effects upon the overall design.

Extra latency is accommodated by inserting registers. This feature applies only to primitive subsystems and is accessed using the ModelPrim Synthesis Info block.

Latency is defined to be the number of delays in the valid signal across the subsystem. The advanced blockset balances delays in the valid and channel path with delays inserted for auto-pipelining in the data path.

User-inserted sample delays in the data path are assumed to be part of the algorithm, rather than pipelining, and are not balanced. However, any uniform delays inserted across the entire data path would be optimized out. If you want to constrain the latency across the entire data path, you can specify this latency constraint in the SynthesisInfo block.

Zero Latency Example

In the example shown in Figure A–1, there are sufficient delays in the design that no extra automatic pipelining is required to reach the $f_{\text{MAX}}$ target (although DSP Builder distributes this user-added delay through the data path). Thus, the reported latency is zero. No extra pipelining registers have been inserted in the data path to meet $f_{\text{MAX}}$ and thus no balancing registers have been inserted on the channel and valid paths. The delay of the valid signal across the subsystem is zero clock cycles, as is shown by the Lat: 0 latency value on the ChannelOut block.

Figure A–1. Latency Example with a User-Specified Delay
Non-Explicit Delays

Delays to the valid path in primitive subsystems that are caused by anything other than explicit delays (for example Counter or Sequence blocks) are not analyzed and counted in the valid latency.

For example, the 4K FFT demonstration design uses a Sequence block to drive the valid signal explicitly as shown in Figure A–2.

Figure A–2. Sequence Block in the 4K FFT Example Design

The latency reported on the ChannelOut block is therefore not 4096 + the automatic pipelining value, but just the pipelining value.

Distributed Delays

Another important point is that the model is not cycle-accurate inside a primitive subsystem. This is because DSP Builder distributes and optimizes the user-specified delay.

For example, in Figure A–1 on page A–1 the Mult block has a direct feed-through simulation model, and the following SampleDelay block has a delayed simulation model with a delay of 10. Thus, there is zero delay on the Mult block in simulation, followed by a delay of 10. In the generated hardware, part of this 10-stage pipelining is distributed throughout the multiplier optimally, such that the Mult block has a delay (in this case, four pipelining stages) and the SampleDelay block a delay (in this case, six pipelining stages). The overall result is the same—10 pipelining stages, but if you try to match signals in the primitive subsystem against hardware, you may find they are shifted by several cycles.

Similarly, if you have insufficient user-inserted delay to meet the required \( f_{\text{MAX}} \) DSP Builder automatically pipelines and balances the delays, and then corrects the cycle-accuracy of the primitive subsystem as a whole, by delaying the output signals in simulation by the appropriate number of cycles at the ChannelOut block.

If there is no user-specified pipelining, the simulation model for the multiplier is direct-feed-through, and the result appears on the output immediately (Figure A–3).
To reach the desired f\text{MAX}, DSP Builder then inserts four pipelining stages in the multiplier, and balances these with four registers on the channel and valid paths. To correct the simulation model to match hardware, the ChannelOut block delays the outputs by four cycles in simulation and displays Lat: 4 on the block. Thus, if you compare the output of the multiplier simulation with the hardware it is now four cycles early in simulation; but if you compare the primitive subsystem outputs with hardware they match, because the ChannelOut block provides the simulation correction for the automatically inserted pipelining.

Suppose you want a consistent 10 cycles of delay across the valid, channel and data paths. Depending on how this is done, you may need latency constraints. Consider the example in Figure A–4.

**Figure A–3.** Latency Example without a User-Specified Delay

**Figure A–4.** Latency Example with Consistent Delays
In this example, a consistent line of SampleDelays has been inserted across the design. However, these delays are not used in the algorithm. DSP Builder recognizes that they are not required and optimizes them away, leaving only the delay that is required. In this case, a delay of four is required on each block, to balance the four delay stages required to pipeline the multiplier sufficiently to reach the target $f_{\text{MAX}}$. This may not be what you expect, or what the simulation model gives, because the delay of 10 in simulation remains from the non-direct-feed-through SampleDelay blocks. In such cases, a warning is given on the MATLAB command line:

Warning:

Some user inserted SampleDelays have been optimized away. The latency on the valid path across primitive subsystem '<design name>' in hardware will be 4, which may differ from the simulation model. If you need to preserve extra SampleDelays in this case, use the 'Constraint Latency' option on the SynthesisInfo block.

In summary, if you want to consistently apply extra latency to a primitive subsystem, use latency constraints.

**Using Latency Constraints**

You can set a latency constrain for a primitive subsystem using the SynthesisInfo block. If you have selected Scheduled synthesis style in the Block Parameters dialog box (Figure A–5), you can optionally turn on Constrain Latency and constrain the latency to $>$, $\geq$, $=$, $\leq$, or $<$ a specified limit.

**Figure A–5.** SynthesisInfo Block Parameters Dialog Box

You can specify the limit using a workspace variable or expression but it must evaluate to a positive integer.

You can use the SynthesisInfo block to specify a latency constraint for any subsystem containing a ModelIP block or primitive subsystem. However, any constraints set by $f_{\text{MAX}}$ targets have priority over latency constraints. This means that an error is issued if you are setting a latency that is smaller than that required to reach the target $f_{\text{MAX}}$. 
If you set a latency constraint of 10 for the example shown in Figure A–3 on page A–3, the hardware uses four pipelining stages across the multiplier to reach the target $f_{\text{MAX}}$, and adds an extra six pipelining stages to meet the latency constraint. This is balanced on the channel and valid paths so that all signals remain synchronized. The result can be seen in simulation, and is visible on the ChannelOut block which displays Lat: 10.

Latency constraints cannot be used in subsystems where folding is enabled.

**Latency and $f_{\text{MAX}}$ Constraint Conflicts**

You cannot set a latency constraint that conflicts with the constraint implied by the $f_{\text{MAX}}$ target. For example, if you set a latency constraint of < 2 for the example shown in Figure A–3, this conflicts with the $f_{\text{MAX}}$ implied pipelining constraint. (The multiplier needs four pipelining stages to reach the target $f_{\text{MAX}}$). The simulation fails and issues an error, highlighting the primitive subsystem (Figure A–6).

This error is issued because the constraint limit must be increased by at least 3 (that is, to < 5) to meet the target $f_{\text{MAX}}$.

**Using Latency Parameters**

The latency of a subsystem is automatically shown on the ChannelOut block after running a simulation in Simulink.

You can also optionally display the latency introduced by a ModelIP block, as described in “Displaying the Latency for ModelIP Blocks” on page 1–8.

You can get the latency value programmatically by using the following command:

```matlab
eval(get_param('<full path to block>', 'latency'))
```

For example:

```matlab
eval(get_param('design/latencydemo/ChannelOut','latency'))
```

The latency is calculated in the initialization of the model, at the start of simulation.
Because all latencies are calculated at the same stage, you cannot use the latency of one block to set the constraint on another and expect to see the changes propagated in the same simulation. For example, suppose you want to always have a latency of 40 for the CIC block in the demo_dcic design. Add a primitive subsystem after the CIC block (Figure A–7). This subsystem consists only of ChannelIn, ChannelOut, and SynthesisInfo blocks. (Figure A–8).

**Figure A–7.** Decimating CIC Example Design Illustrating Latency Propagation

![Diagram showing latency propagation](image)

**Figure A–8.** Primitive Subsystem for the CIC Design Example

![Diagram showing primitive subsystem](image)

On the SynthesisInfo block, set the following latency constraint derived from the CIC block:

```plaintext
40 - eval(get_param('demo_dcic/CICSystem/CIC', 'latency'))
```

Any changes to the CIC block that change the latency are not displayed as immediate latency changes in the primitive subsystem; but only on the subsequent simulation and hardware generation cycle. This is because the latency for the CIC block is only available after the initialization step in which it is set in the primitive subsystem.

Thus, you must simulate until after cycle 0, stop and re-start simulation (without making any changes) to guarantee that the correct latency is being applied in the primitive subsystem. The SynthesisInfo block is effectively using the evaluated CIC block latency from the previous simulation.
By default, the hardware generated for a primitive subsystem by DSP Builder Advanced Blockset is capable of receiving and processing new data every clock cycle. However, this feature may not be required for some designs. For those designs with a sample rate lower than their clock rate, DSP Builder Advanced Blockset’s folding functionality can take advantage of the disparity between both rates to optimize the generated hardware.

Setting latency constraints is incompatible with folding in primitive subsystems. Any latency constraints set in folded subsystems are ignored.

**Using Folding**

Folding is enabled and configured in the primitive subsystem’s ChannelIn and ChannelOut blocks. Figure B–1 shows the ChannelIn and ChannelOut blocks.

**Figure B–1. ChannelIn and ChannelOut Block Parameters**

Turn on Folding enabled, to edit the Number of used TDM slots, and Sample rate parameters. The correct values for these parameters depend on the desired type of folded subsystem, and are explained in the following sections.

The Folding enabled, Number of used TDM slots and Sample rate parameters must be set to identical values for all ChannelIn and ChannelOut blocks in the same primitive subsystem.
Folded Subsystem without TDM Demuxing

The simplest type of folded subsystem is the folded subsystem without TDM demuxing. One piece of new data (per input port) is received, processed, and output every folding factor clock cycles, where folding factor is a constant integer number. A typical example of this type of design is one that processes a single channel of data.

For example, for a folding factor of 4, the data format expected at ChannelIn inputs and produced at ChannelOut outputs is:

\[ a_0, x, x, x, a_1, x, x, x, a_2, x, x, x, a_3, x, x, x, \ldots \]

Where \( a_0, a_1, a_2, a_3, \ldots \) are taken as the input data to the design, and data received during the cycles marked as \( x \) is ignored.

In this type of folded subsystem, Number of used TDM slots must be set to 1, and Sample rate must be set to the clock rate divided by the folding factor. No other changes are needed with respect to the unfolded version of the same subsystem.

The primitive_fir design is an example of a folded subsystem without TDM de-muxing.

Folded Subsystem with TDM Demuxing

Folded subsystems can also accept and produce a TDM version of the data format where the TDM period is the folding factor. You enable this feature by configuring the Number of used TDM slots parameter to a number greater than 1 but smaller or equal to the folding factor.

For example, for a folding factor of 4, and the number of used TDM slots set to 3, the data format expected at ChannelIn inputs and produced at ChannelOut outputs is:

\[ a_0, b_0, c_0, x, a_1, b_1, c_1, x, a_2, b_2, c_2, x, a_3, b_3, c_3, x, \ldots \]

The received data is demuxed at ChannelIn blocks. The de-muxed data is presented at ChannelIn outputs as a Simulink vector that has as many elements as TDM slots are used. In the example, the Simulink vector has three elements: the first corresponding to the "a" channel or TDM slot, the second one to "b" and the third one to "c".

The reverse process occurs at the system’s ChannelOut blocks. A Simulink vector with the same number of elements is expected at the ChannelOut inputs; these elements are then muxed by the ChannelOut block and presented in the same TDM format described above.

Figure B–2 shows a ChannelIn block configured with "Number of used TDM slots" set to 3. A Simulink Demux block has been used to provide access to the elements of the vector, here called "r", "g" and "b". The converse structure, with a Simulink Mux block preceding the ChannelOut block, is used at the system’s output.

Figure B–2 is a screenshot from the demo_fold_csc design, which is an example of a folded subsystem with TDM de-muxing.
When enabling TDM demuxing, Simulink vectors appear as described. Ensure that the design continues to work as expected in the presence of these vectors.

The number of used TDM slots must be smaller or equal to the folding factor. If it is equal, new data is received and processed every cycle, so saving hardware (with respect to the unfolded case) is not possible.

**Effects of Folding**

When folding is enabled and the number of used TDM slots is smaller than the folding factor no data is received or produced in some clock cycles. DSP Builder Advanced Blockset can use these spare cycles to perform computation that normally requires extra hardware blocks to be created. This action can result in hardware savings, which is usually the intended result of enabling folding for some particular design.

However, enabling folding also induces other changes in the resulting hardware, as explained in the following sections. In certain cases these changes can result in hardware that is less desirable in some way than the one obtained with folding disabled. A general understanding of these changes can help to predict cases where usage of folding may not be advisable.

**Effects on Manual Delays**

When folding is enabled, the reception times for two consecutive pieces of data (for the same channel or slot, if TDM de-muxing is used) are separated by a number of clock cycles equal to the folding factor. So to produce their expected behaviour, z^n manual delay blocks in the design delay by a number of clock cycles equal to times the folding factor. Compared to the unfolded case, the length of these delays in clock cycles gets multiplied by the folding factor.

For example, if the subsystem in Figure B–3 is configured with a folding factor of 4, data at ChannelIn q0 port follows this pattern:

a0, x, x, x, a1, x, x, a2, x, x, x, a3, x, x, x, ...

If SampleDelay only delayed by one clock cycle, the adder does not add a sample to the previous one as expected. For example, when q0 is equal to a1, the output of SampleDelay corresponds to the "x" that was received in the previous clock cycle, and the output of the adder is not meaningful.

![Figure B-2. ChannelIn Block with TDM Demuxing](image-url)
To avoid this problem, DSP Builder Advanced Blockset implements `SampleDelay` as a 4-cycle delay.

**Figure B–3. Example of Folded Manual Delay**

If the subsystem’s highest latency path is one that contains manual delays, the subsystem’s global latency is affected by enabling folding.

Although the length in cycles of manual delays is multiplied by the folding factor, the cost in hardware resources grows by a smaller factor that is typically only slightly above the number of used TDM slots.

The effect of folding on manual delays can solve the situation where DSP Builder Advanced Blockset cannot implement a certain feedback loop in hardware because of insufficient delays specified. A sufficiently large folding factor increases the length in clock cycles of existing delays so that they can compensate for the required pipelining of other blocks.

**Effects on Combinational Logic**

One of the methods that folding can use to reduce hardware is the combination of blocks that are not used every cycle. For example, consider the blocks in **Figure B–4**.

**Figure B–4. Example of Folded Multipliers**
With folding disabled, the resulting hardware can closely resemble the original Simulink blocks (Figure B–5).

**Figure B–5.** Example of Unfolded Hardware

Assume that folding is enabled with a folding factor of 2. Each of the A, B, C, D inputs in Figure B–5 only receive data every second clock cycle. As the hardware multipliers can perform one multiplication per clock cycle, in this example each one is only used half of the time.

In this example, DSP Builder Advanced Blockset can use a single hardware multiplier to implement the two Simulink multiplier blocks. To timeshare the hardware multiplier between performing \((A \times B)\) and \((C \times D)\), DSP Builder Advanced Blockset must insert some extra delays and muxes. The resulting hardware may look like Figure B–6.

**Figure B–6.** Example of Folded Hardware

The folded hardware in Figure B–6 may appear more expensive than the unfolded hardware in Figure B–5. However this observation typically is not true. The saved multiplier normally requires more hardware resources than the delays and muxes that replace it. DSP Builder Advanced Blockset implements heuristics that limit block combination to those cases where it is expected to be beneficial.

**Examples**

Altera offers the following folding demonstrations:

- demo_fold_csc
- demo_iir_fold2
- demo_iir_x3a
- primitive_fir
Symbols
.hex file 1–20
.qip file 1–20
.xml file 1–20

A
Automatic testbench
  Enabling 1–2
  Loading 1–23
  Running from a command 1–23
Avalon-MM interface
  Bus clock 1–19

B
Base Blocks
  Example designs 2–4
  Library 1–2
Bus clock 1–4
  Avalon-MM interface 1–19

C
CDelay RAM Block threshold 1–3
CDualMem Dist RAM threshold 1–3
Clock rate 1–19

D
Demonstration design
  Opening 2–2
  Running 2–5
Device family support
  Advanced blockset 1–5

E
Example designs
  16-Channel DDC 2–6
  16-Channel DUC 2–6
  2-Antenna DUC for WiMAX 2–7
  2-Channel DUC 2–7
  4,096 point FFT 2–19
  4,096 point IFFT 2–20
  8,192 point FFT 2–20
  8,192 point IFFT 2–21
  Automatic Gain Control 2–16
  Complex Mixer 2–12
  Decimating CIC Filter 2–8
  Decimating FIR Filter 2–9
  Fibonacci Series 2–16
  Filter Chain with Forward Flow Control 2–11
  Fractional FIR Filter Chain 2–10
  Fractional Rate FIR Filter 2–9
  Half Band FIR Filter 2–9
  Hello World 2–15
  Interpolating CIC Filter 2–8
  Interpolating FIR Filter 2–9
  Inverse Discrete Cosine Transform 2–17
  Memory-Mapped Registers 2–22
  Multi-Channel IIR Filter 2–17
  NCO 2–11
  NCO, 4 channel, 16 banks 2–14
  NCO, 4 channel, 2 banks 2–12
  NCO, 4 channel, 4 banks 2–13
  NCO, 4 channel, 64 banks 2–15
  NCO, 4 channel, 8 banks, 2 wires 2–14
  Quadrature Amplitude Modulation 2–18
  Radix 2 Streaming FFT 2–18
  Radix 4 Streaming FFT 2–19
  Real Mixer 2–11
  Root Raised Cosine FIR Filter 2–10
  Scale 2–22
  Single Rate FIR Filter 2–8
  Super-Sample FIR Filter 2–10
  Test CORDIC Using Primitive Blocks 2–21
  Test CORDIC Using the CORDIC Block 2–22

F
FFT Blockset
  Library 1–9
Filters
  Example designs 2–2
  Library 1–8
Fixed point types 1–19

G
Generated files
  Advanced Blockset model 1–20
  DDC demonstration design 5–23
  Fibonacci demonstration design 4–12
  NCO demonstration design 3–11

H
Hard multiplier threshold 1–3
Hardware generation
  Enabling 1–2
Host Interface
  Example designs 2–4
Index–2

L
Latency parameter
  Displaying for a ModelIP block 1–8
  Displaying for a primitive subsystem 1–5
  Reading 1–9
Library
  Base Blocks 1–2
  FFT Blockset 1–9
  ModelBus 1–9
  ModelIP 1–8
    Filters 1–8
    Waveform Synthesis 1–8
  ModelPrim 1–5

M
Memory and multiplier thresholds 1–2
Memory mapped interface 1–17
Memory-Mapped bus interface
  Address and bus width 1–2
ModelBus
  Example designs 2–4
  Library 1–9, 1–9
ModelIP
  Library 1–8
  Protocol 1–10
ModelPrim
  Example designs 2–3
  Library 1–5
M-RAM threshold 1–3
Multichannel systems 1–13

P
Platforms
  Example designs 2–2
Primitive blocks
  Example designs 2–3
  Output data type 1–5
  Synthesis styles 1–6

R
Reference designs 2–4
  1-Antenna WiMAX DDC 2–28
  1-Antenna WiMAX DUC 2–29
  1-Carrier, 2-Antenna W-CDMA DDC 2–24
  1-Carrier, 2-Antenna W-CDMA DUC 2–25
  2-Antenna WiMAX DDC 2–29
  2-Antenna WiMAX DUC 2–30
  4-Carrier, 2-Antenna W-CDMA DDC 2–23
  4-Carrier, 2-Antenna W-CDMA DUC 2–25
  4-Carrier, 2-Antenna W-CDMA DUC at 307.2
     MHz with Total Rate Change 40 2–27
  4-Carrier, 2-Antenna WDCMA DUC at 368.64
     MHz with Total Rate Change 32 2–26
  4-Carrier, 2-Antenna WDCMA DUC at 368.64
     MHz with Total Rate Change 48 2–27

S
Sample rate 1–19
Standard blockset interoperability 1–1
System clock 1–4

T
Time-division multiplexing 1–12
Tutorials
  Example designs 2–4
  ModelIP 3–1
  Primitive Library 4–1
  System 5–1

V
Vectorized inputs 1–13

W
Walkthrough
  ModelIP tutorial 3–1
  Primitive Library tutorial 4–1
  System tutorial 5–1
Waveform Synthesis
  Example designs 2–3
  Library 1–8
Additional Information

Revision History

The following table displays the revision history for this user guide.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes Made</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2009</td>
<td>9.1</td>
<td></td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0</td>
<td>Added CORDIC and multiple bank NCO demonstration designs.</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1</td>
<td>Added appendix describing latency management and details of latency display for ModelIP blocks, time-division multiplexing, channelization, generated files, and Quartus II projects. Also added additional descriptions for new W-CDMA example designs.</td>
</tr>
<tr>
<td>May 2008</td>
<td>8.0</td>
<td>First release of this user guide.</td>
</tr>
</tbody>
</table>

How to Contact Altera

For the most up-to-date information about Altera products, refer to the following table.

<table>
<thead>
<tr>
<th>Contact (Note 1)</th>
<th>Contact Method</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technical support</td>
<td>Website</td>
<td><a href="http://www.altera.com/support">www.altera.com/support</a></td>
</tr>
<tr>
<td>Technical training</td>
<td>Website</td>
<td><a href="http://www.altera.com/training">www.altera.com/training</a></td>
</tr>
<tr>
<td></td>
<td>Email</td>
<td><a href="mailto:custrain@altera.com">custrain@altera.com</a></td>
</tr>
<tr>
<td>Product literature</td>
<td>Website</td>
<td><a href="http://www.altera.com/literature">www.altera.com/literature</a></td>
</tr>
<tr>
<td>Non-technical support (General) (Software Licensing)</td>
<td>Email</td>
<td><a href="mailto:nacomp@altera.com">nacomp@altera.com</a></td>
</tr>
<tr>
<td></td>
<td>Email</td>
<td><a href="mailto:authorization@altera.com">authorization@altera.com</a></td>
</tr>
</tbody>
</table>

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

Typographic Conventions

This document uses the typographic conventions shown in the following table.

<table>
<thead>
<tr>
<th>Visual Cue</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bold Type with Initial Capital Letters</strong></td>
<td>Indicates command names, dialog box titles, dialog box options, and other GUI labels. For example, <strong>Save As</strong> dialog box.</td>
</tr>
<tr>
<td><strong>bold type</strong></td>
<td>Indicates directory names, project names, disk drive names, file names, file name extensions, and software utility names. For example, \designs directory, <strong>d</strong>: drive, and <strong>chiptrip.pdf</strong> file.</td>
</tr>
<tr>
<td><em>Italic Type with Initial Capital Letters</em></td>
<td>Indicates document titles. For example, <em>AN 519: Stratix IV Design Guidelines</em>.</td>
</tr>
</tbody>
</table>
### Typographic Conventions

<table>
<thead>
<tr>
<th><strong>Visual Cue</strong></th>
<th><strong>Meaning</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Italic type</strong></td>
<td>Indicates variables. For example, ( n + 1 ). Variable names are enclosed in angle brackets ((&lt;&gt;)). For example, (&lt;\text{file name}&gt;) and (&lt;\text{project name}&gt;.\text{pof}) file.</td>
</tr>
<tr>
<td>Initial Capital Letters</td>
<td>Indicates keyboard keys and menu names. For example, Delete key and the Options menu.</td>
</tr>
<tr>
<td>“Subheading Title”</td>
<td>Quotation marks indicate references to sections within a document and titles of Quartus II Help topics. For example, “Typographic Conventions.”</td>
</tr>
<tr>
<td><strong>Courier type</strong></td>
<td>Indicates signal, port, register, bit, block, and primitive names. For example, (\text{data1}), (\text{tdi}), and (\text{input}). Active-low signals are denoted by suffix (\text{n}). Example: (\text{resetn}). Indicates command line commands and anything that must be typed exactly as it appears. For example, (\text{c:}\backslash\text{qdesigns}\backslash\text{tutorial}\backslash\text{chiptrip.gdf}). Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword (\text{SUBDESIGN})), and logic function names (for example, (\text{TRI})).</td>
</tr>
<tr>
<td>1., 2., 3., and a., b., c., and so on.</td>
<td>Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.</td>
</tr>
<tr>
<td>■ ■</td>
<td>Bullets indicate a list of items when the sequence of the items is not important.</td>
</tr>
<tr>
<td>■</td>
<td>The hand points to information that requires special attention.</td>
</tr>
<tr>
<td>CAUTION ■</td>
<td>A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.</td>
</tr>
<tr>
<td>WARNING ■</td>
<td>A warning calls attention to a condition or possible situation that can cause you injury.</td>
</tr>
<tr>
<td>♬</td>
<td>The angled arrow instructs you to press the Enter key.</td>
</tr>
<tr>
<td>♬</td>
<td>The feet direct you to more information about a particular topic.</td>
</tr>
</tbody>
</table>