XILINX®

LogiCORE IP FIR Compiler v6.2

DS795 March 1, 2011 **Product Specification**

Overview

The Xilinx® LogiCORE™ IP FIR Compiler core provides a common interface for users to generate highly parameterizable, area-efficient high-performance FIR filters.

Features

- Drop-in module for Virtex®-7 and Kintex™-7, Virtex-6 and Spartan®-6 FPGAs
- AXI4-Stream-compliant interfaces
- High-performance finite impulse response (FIR), polyphase decimator, polyphase interpolator, half-band decimator and half-band interpolator implementations
- Support for up to 256 sets of coefficients, with 2 to 2048 coefficients per set
- Input data up to 49-bit precision
- Filter coefficients up to 49-bit precision
- Support for up to 64 interleaved data channels
- Interpolation and decimation factors of up to 64 generally and up to 1024 for single channel filters
- Support for multiple parallel datapaths with shared control logic
- Online coefficient reload capability
- User-selectable output rounding
- Use with Xilinx CORE Generator™ tool and Xilinx System Generator for DSP 13.1

LogiCORE IP Facts	
Core Specifics	
Supported Device Family(1)	Kintex-7, Virtex-7 Virtex-6, Spartan-6
Supported User Interfaces	AXI4-Stream
Configuration	See Table 9, Table 10
Provided with Core	
Documentation	Product Specification
Design Files	Netlist
Example Design	Not Provided
Test Bench	VHDL
Constraints File	N/A
Simulation Model	VHDL and Verilog
Tested Design Tools	
Design Entry Tools	CORE Generator tool 13.1 System Generator for DSP 13.1
Simulation	Mentor Graphics ModelSim 6.6d Cadence Incisive Enterprise Simulator (IES) 10.2 Synopsys VCS and VCS MX 2010.06 ISIM 13.1
Synthesis Tools	N/A
Support	
Provided by Xilinx, Inc.	

1. For the complete list of supported devices, see the release notes for this core.

© Copyright 2010 - 2011 Xilinx, Inc. XILINX, the Xilinx logo, Kintex, Virtex, Spartan, ISE and other designated brands included herein are trademarks of Xilinx in the United States and other countries. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. ARM is a registered trademark of ARM in the EU and other countries. The AMBA trademark is a registered trademark of ARM Limited. All other trademarks are the property of their respective owners.

Functional Description

Overview

A wide range of filter types can be implemented in the Xilinx CORE Generator tool: single-rate, polyphase decimators and interpolators and half-band decimators and interpolators. Structure in the coefficient set is exploited to produce area-efficient FPGA implementations. Sufficient arithmetic precision is employed in the internal datapath to avoid the possibility of overflow.

The conventional single-rate FIR version of the core computes the convolution sum defined in Equation 1, where N is the number of filter coefficients.

$$y(k) = \sum_{n=0}^{N-1} a(n)x(k-n) \quad k = 0, 1, ..$$ *Equation 1*

Figure 1 illustrates the conventional tapped delay line realization of this inner-product calculation, and although the illustration is a useful conceptualization of the computation performed by the core, the actual FPGA realization is quite different.

One or more time-shared multiply-accumulate (MAC) functional units are used to service the N sum-of-product calculations in the filter. The core automatically determines the minimum number of MAC engines required to meet user-specified throughput.

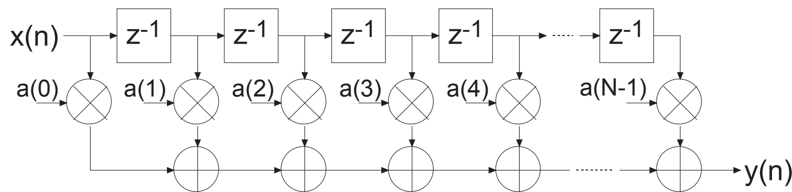


Figure 1: **Conventional Tapped Delay Line FIR Filter Representation**

Feature Support Matrix

Table 1: **Feature Support Matrix**

Feature	Systolic Multiply-Accumulate		Transpose Multiply-Accumulate	
	Virtex-6, Virtex-7, Kintex-7 FPGAs	Spartan-6 FPGAs	Virtex-6, Virtex-7, Kintex-7 FPGAs	Spartan-6 FPGAs
Number of Coefficients	2–2048	2–2048	2–2048	2–2048
Coefficient Width(1)	2–49	2–35	2–49	2–35
Data Width(1,2)	2–49	2–35	2–49	2–35
Number of Channels	1–64	1–64	1	1
Parallel Datapaths(3)	1-16	1-16	1-16	1-16
Maximum Rate Change *Single Channel* *Multiple Channels*	 1024 512	 1024 512	 1024 N/A	 1024 N/A
Fractional Rate Support	✔	✔	✖	✖
Coefficient Reload	✔	✔	✔	✔
Coefficient Sets	1–256	1–256	1–256	1–256
Output Rounding	✔	✔	✔	✔

Notes:

1. Maximum Coefficient Width reduces by one when the Coefficients are signed. Similarly for Maximum Data Width when the Data values are signed.
2. The allowable range for the Data Width field in the GUI might reduce further in Virtex-6/Virtex-7/Kintex-7 devices to ensure that the accumulator width does not exceed maximum.
3. Maximum Parallel Datapaths reduces to 8 when Coefficient Width or Data Width is greater than 25-bits for Virtex-6/Virtex-7/Kintex-7 FPGAs or 18-bits for other families.

Table 2 shows the classes of filters that are supported for the FIR Compiler core.

Table 2: **Filter Configuration Support Matrix**

Filter Configuration	Supported
Conventional Single-rate FIR	✔
Half-band FIR	✖
Hilbert Transform [Ref 3]	✖
Interpolated FIR [Ref 4] [Ref 5]	✖
Polyphase Decimator	✔
Polyphase Interpolator	✔
Half-band Decimator	✔
Half-band Interpolator	✔
Polyphase Filter Bank	✖

The unsupported filter configurations will be re-introduced in a subsequent release of the FIR Compiler core.

The supported filter configurations are described in separate sections within this document.

Notable Limitations

In conjunction with Table 1 and Table 2, it is important to note some further limitations inherent in the core.

When selecting the Systolic Multiply-Accumulate architecture, the limitations are as follows:

- Symmetry is not exploited in configurations requiring multiple columns of DSP slices.
  - See the appropriate DSP48 user guide for device column information.
    - *Virtex-6 FPGA DSP48E1 Slice User Guide* (UG369)
    - *7 Series FPGAs DSP48E1 Slice User Guide* (UG479)
    - *Spartan-6 FPGA DSP48A1 Slice User Guide* (UG389)
  - The column length information is also available in the CORE Generator GUI.
- Fractional Rate filters do not currently exploit coefficient symmetry.

When selecting the Transpose Multiply-Accumulate architecture, the limitations are as follows:

- Symmetry is not exploited.
- Multiple interleaved channels are not supported.

Pinout

Figure 2 displays the schematic symbol for the interface pins to the FIR Compiler module.

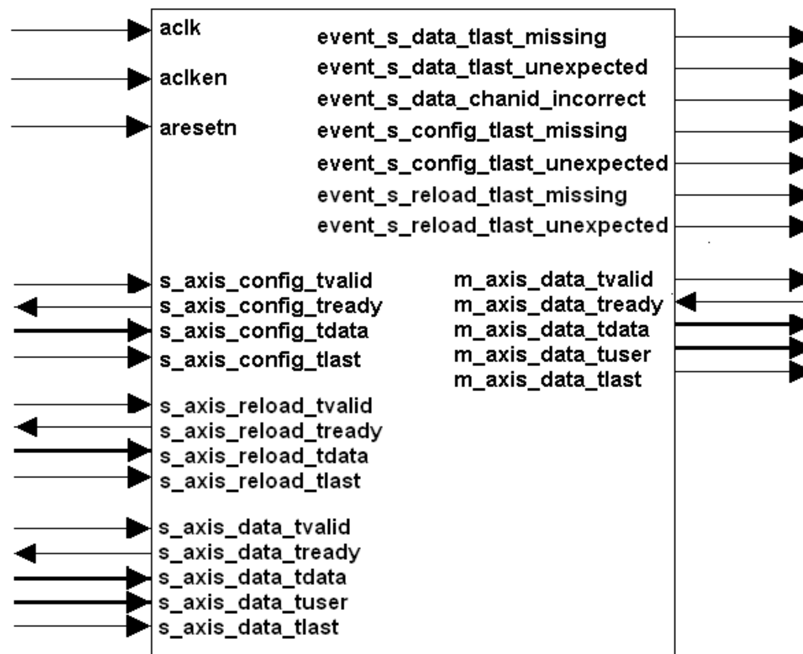


Figure 2: **FIR Compiler Core Pinout**

Filter input data is supplied on the `s_axis_data_tdata` port (N bits wide, extended if necessary to fit a byte boundary), subject to the `s_axis_data_tvalid` and `s_axis_data_tready` handshake, and filter output

samples are presented on the m_axis_data_tdata port (R bits wide, extended if necessary to fit a byte boundary), subject to the m_axis_data_tvalid and m_axis_data_tready handshake. The maximum output width R is the sum of the data bit width N and the bit growth of the filter; see the Output Width and Bit Growth section for more details. The output width can also be reduced further under user control by truncation or rounding. See Input and Output DATA Channels for full details of how input and output fields map to the TDATA port of each channel.

The aclk signal is the system clock for the core, where the clock rate can be greater than or equal to the input signal sample frequency.

The ND, RDY, and RFD signals of previous versions have been replaced by AXI interfaces. See AXI4-Stream Considerations for dataflow protocol.

For interleaved data channel implementations an optional field, channel_id, of the m_axis_data_tuser port specifies which interleaved data channel the current transaction relates to. A similar optional field is available on the s_axis_data_tuser port. The input value is checked against the core's internal state and a warning is generated on the event_s_data_chanid_incorrect port if the value does not match the state of the core. The channel_id field is C bits wide, where C is the required bit width to represent the maximum channel value.

Where multiple coefficient sets are specified in the .coe file, the CONFIG channel (s_axis_config_t*) is used to select the active filter set via the fsel field of the s_axis_config_tdata port. The core can be configured to use a single filter selection for all interleaved data channels or have a unique filter selection for each interleaved data channel. The fsel field is F bits wide; F is the required bit width to represent the maximum filter set value. See CONFIG Channel, page 21 for full details.

The RELOAD channel (s_axis_reload_t*) is used to load new filter coefficients into the filter from an external source. The CONFIG channel (s_axis_config_t*) is then used to enable the use of the newly load filter coefficients. See RELOAD Channel, page 22 and CONFIG Channel, page 21 for full details.

Resetting the core is achieved by driving the aresetn pin which is active low, for a minimum of two cycles; it does not require the assertion of clock enable (aclken). A clock enable (aclken) pin also optional is available.

Table 3 defines the FIR filter port names and port functional descriptions.

Table 3: **Core Signal Pinout**

Name	Direction	Optional	Description
aclk	Input	no	Rising-edge clock
aclken	Input	yes	Active-high clock enable (optional). Available for MAC-based FIR implementations.
aresetn	Input	yes	Active-low synchronous clear (optional, always take priority over aclken).
s_axis_config_tvalid	Input	yes	TVALID for CONFIG channel. Asserted by external master to indicate data is available for transfer.
s_axis_config_tready	Output	yes	TREADY for CONFIG channel. Asserted by core to indicate core is ready to accept data.
s_axis_config_tdata[A-1:0]	Input	yes	TDATA for CONFIG channel. See TDATA of CONFIG Channel for internal structure and width.
s_axis_config_tlast	Input	yes	TLAST for CONFIG channel. Indicates the last transfer of a reconfiguration packet.
s_axis_reload_tvalid	Input	yes	TVALID for RELOAD channel. Asserted by external master to indicate data is available for transfer.

Table 3: **Core Signal Pinout *(Cont'd)***

Name	Direction	Optional	Description
s_axis_reload_tready	Output	yes	TREADY for RELOAD channel. Asserted by core to indicate core is ready to accept data.
s_axis_reload_tdata	Input	yes	TDATA for RELOAD channel. Conveys the coefficient data stream.
s_axis_reload_tlast	Input	yes	TLAST for RELOAD channel. Indicates the last transfer of a packet of coefficients.
s_axis_data_tvalid	Input	yes	TVALID for input DATA channel. Asserted by external master to indicate data is available for transfer.
s_axis_data_tready	Output	yes	TREADY for input DATA channel. Asserted by core to indicate core is ready to accept data.
s_axis_data_tdata	Input	yes	TDATA for input DATA channel. Conveys the data stream to be filtered. See TDATA Structure for internal structure.
s_axis_data_tuser	Input	yes	TUSER for input DATA channel. Conveys ancillary data to be passed through the core with latency equal to the input DATA to output DATA datapath and or a chan ID field to identify which Time Division Multiplexed (TDM) channel the current sample belongs to.
s_axis_data_tlast	Input	yes	TLAST for input DATA channel.This optionally indicates the last of a cycle of TDM channels or can indicate the end of an arbitrary packet in which case it is conveyed to the output with latency equal to the main data stream.
m_axis_data_tvalid	Output	yes	TVALID for output DATA channel. Asserted by core to indicate data is available for transfer.
m_axis_data_tready	Input	yes	TREADY for output DATA channel. Asserted by external slave to indicate the slave is ready to accept data.
m_axis_data_tdata	Output	yes	TDATA for the output DATA channel. This is the filtered data stream. See TDATA Structure for internal structure.
m_axis_data_tuser	Output	yes	TUSER for the output DATA channel. Optionally conveys a user field from the input DATA TUSER port and/or a chan ID field to identify which TDM channel the current sample belongs to.
m_axis_data_tlast	Output	yes	TLAST for the output DATA channel. Optionally indicates the last sample of a cycle of TDM channels (vector framing) or the TLAST passed through the core from the input DATA channel (packet framing)
event_s_data_tlast_missing	Output	yes	Indicates that the input DATA TLAST was not asserted when expected by an internal channel counter.
event_s_data_tlast_unexpected	Output	yes	Indicates that the input DATA TLAST was asserted when not expected by an internal channel counter.
event_s_data_chanid_incorrect	Output	yes	Indicates that the chan ID field of the input DATA TUSER port did not match the value of an internal counter.
event_s_reload_tlast_missing	Output	yes	Indicates that the RELOAD TLAST was not asserted when expected by an internal counter.
event_s_reload_tlast_unexpected	Output	yes	Indicates that the RELOAD TLAST was asserted when not expected by an internal counter.
event_s_config_tlast_missing	Output	yes	Indicates that the CONFIG TLAST was not asserted when expected by an internal counter.
event_s_config_tlast_unexpected	Output	yes	Indicates that the CONFIG TLAST was asserted when not expected by an internal counter.

Parallel Datapaths

Up to 16 parallel datapaths are supported. These parallel datapaths are mapped into `s_axis_data_tdata` and `m_axis_data_tdata`. See Input and Output DATA Channels, page 19 for TDATA internal structure.

CORE Generator Graphical User Interface

The FIR Compiler GUI contains four pages used to configure the core plus four informational/analysis tabs.

Tool Tips appear when hovering the mouse over each parameter and a brief description appears, as well as feedback about how their values or ranges are affected by other parameter selections. For example, the Coefficient Structure Tool Tip displays the inferred structure when Inferred is selected from the drop-down list.

Tab 1: IP Symbol

The IP Symbol tab illustrates the core pinout.

Tab 2: Freq. Response

The Freq. Response tab (Figure 3), the default tab when the CORE Generator software is started, displays the filter frequency response (magnitude only). The content of the tab can be adjusted to fit the entire window or un-docked (as shown) into a separate window.

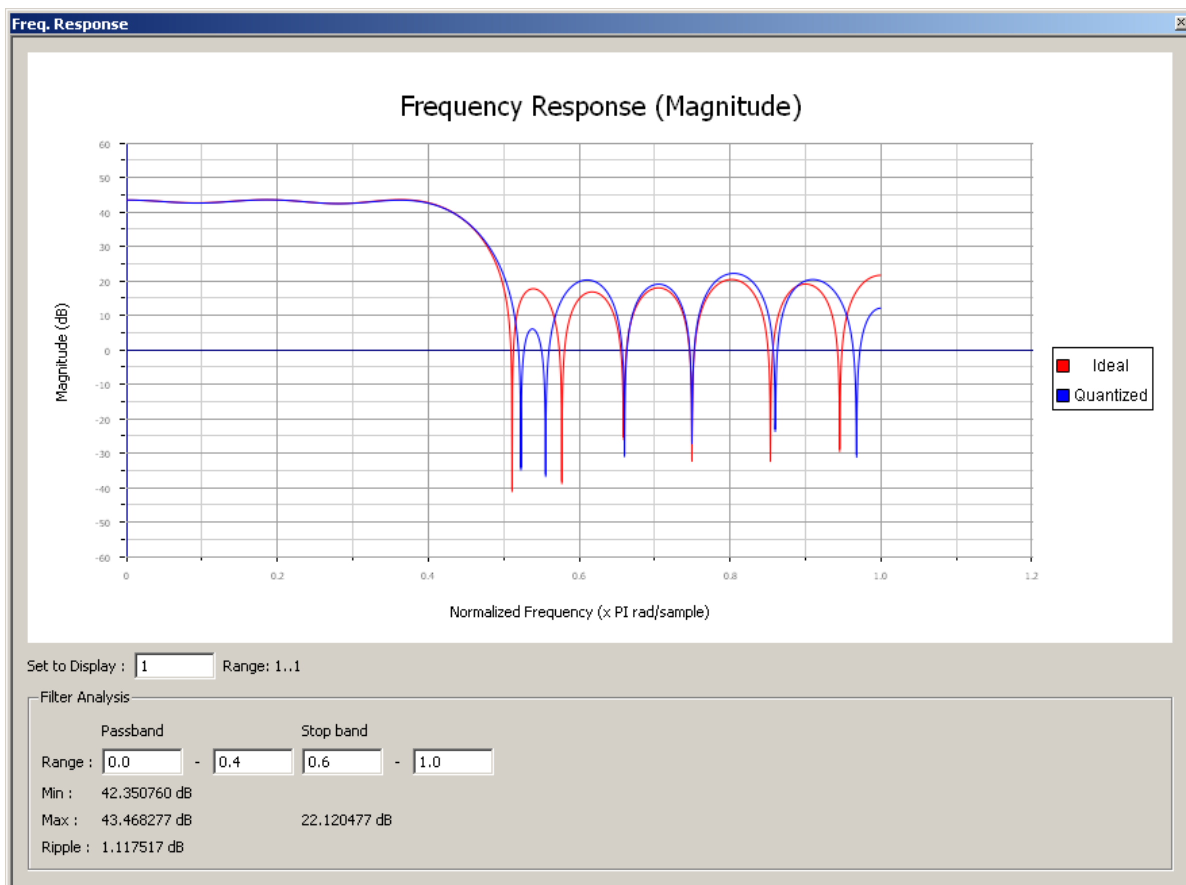


Figure 3: **Freq. Response Tab**

The frequency response of the currently selected coefficient set is plotted against normalized frequency. Where the Quantization option is set to Integer Coefficients, there is only a single plot based on the specified coefficient values. Where the Quantization option has been set to Quantize Only, an ideal plot is displayed based on the provided values alongside a Quantized plot based on a set of coefficient values quantized according to the specified coefficient bit width. Where the Quantization option is set to Maximize Dynamic Range, the coefficients are first scaled to take full advantage of the available dynamic range, then quantized according to the specified coefficient bit width. The quantized coefficients are summed to determine the resulting gain factor over the provided real coefficient set, and the resulting scale factor is used to correct the filter response of the quantized coefficients such that the gain is factored out. The scale factor is reported in the legend text of the frequency response plot and on the Summary page. See the Coefficient Quantization section for more details.

- **Set to Display:** This selects which of multiple coefficient sets (if applicable) is displayed in the Frequency Response Graph.
- **Passband Range:** Two fields are available to specify the passband range, the left-most being the minimum value and the right-most the maximum value. The values are specified in the same units as on the graph x-axis (for example, normalized to pi radians per second). For the specified range the passband maximum, minimum and ripple values are calculated and displayed (in dB).
- **Stopband Range:** Two fields are available to specify the stopband range, the left-most being the minimum value and the right-most the maximum value. The values are specified in the same units as on the graph x-axis (for example, normalized to pi radians per second). For the specified range the stopband maximum value is calculated and displayed (in dB).

The user can specify any range for the passband or stopband, allowing closer analysis of any region of the response. For example, examination of the transition region can be done to more accurately examine the filter roll-off.

Tab 3: Implementation Details

The Implementation Details tab displays Resource Estimation information, core latency, actual calculated coefficients, and the internal structure of AXI4-Stream TDATA and TUSER ports.

The number of DSP slices/Multipliers is displayed in addition to a count of the number of block RAM elements required to implement the design. Usage of general slice logic is not currently estimated.

It should be noted that the results presented in the Resource Estimation are estimates only using equations that model the expected core implementation structure. The Resource Utilization option within the CORE Generator software should be used after generating the core to get a more accurate report on all resource usage. It is not guaranteed that the resource estimates provided in the GUI match the results of a mapped core implementation.

For some configurations, the number of coefficients calculated by the core might be greater than specified. In this circumstance, the user can increase the number coefficients used to specify the filter at little or no cost in resource usage.

The AXI4-Stream Channel Sub-Field Details pane describes fields internal to the AXI4-Stream ports. The individual fields cannot be broken out as individual ports with application-specific names because of AXI4-Stream rules. This pane allows the user to see how individual fields map to the indices of the compound port as a whole.

Tab 4: Coefficient Reload

The Coefficient Reload tab provides the facility to generate re-ordered filter coefficient files for use with the RELOAD channel. The tab also displays the coefficient reload order.

The coefficient reload order is displayed when “Use Reloadable Coefficients” has been selected and “Display Reload Order” is checked. This information is also contained in the <component_name>_reload_order.txt file produced during core generation. See the Coefficient Reload section for more details.

Reload Coefficients MIF File Generation pane is enabled when “Use Reloadable Coefficients” has been selected. Reload files can be generated for the coefficients used to specify the filter configuration (“Coefficient Vector” or “Coefficient File”) or for coefficients specified via the “Reload Coefficient File” parameter. It uses the same COE format as the “Coefficient File” parameter. See Filter Coefficient Data, page 44 for more details. The reload filter coefficient characteristics must match those of the coefficients used to specify the filter configuration.

The re-ordered coefficients are output in a multiple binary text files formatted to the width of the `s_axis_reload_tdata` port.

The output file names have the following format given their source:

Filter Specification Coefficients: <*component_name*>_rld_src_<*x*>.txt

Reload Coefficient File: <*component_name*>_rld_coe_<*x*>.txt

where *x* specifies the coefficient set.

Filter Specification Screen

The Filter Specification screen is used to define the basic configuration and performance of the filter.

- **Component Name**: The user-defined filter component instance name.

FIlter Coefficients

- **Coefficient Source:** Specifies which coefficient input method to use, directly in the GUI via the Coefficient Vector parameter or from a .coe file specified by the Coefficient File parameter.
- **Coefficient Vector:** Used to specify the filter coefficients directly in the GUI. The filter coefficients are specified in decimal using a comma delimited list as for the “coefdata” field in the Filter Coefficient Data file. As with the .coe file, the filter coefficients can be specified using non-integer real numbers which the FIR Compiler quantizes appropriately, given the user requirements. See the Coefficient Quantization section for more details.
- **Coefficients File**: Coefficient file name. This is the file of filter coefficients. The file has a .coe extension, and the file format is described in theFilter Coefficient Data section. The file can be selected through the dialog box activated by the Browse.
- **Show Coefficients:** Selecting this button displays the filter coefficient data defined in the specified Coefficient file in a pop-up window.
- **Number of Coefficient Sets**: The number of sets of filter coefficients to be implemented. The value specified must divide without remainder into the number of coefficients derived from the .coe file or Coefficient Vector.
- **Number of Coefficients (per set):** The number of filter coefficients per filter set. This value is automatically derived from the specified coefficient data and the specified number of coefficient sets.

Filter Specification

- **Filter Type**: Three filter types are currently supported: Single-rate FIR, Interpolating FIR, Decimating FIR.
- **Rate Change Type:** This field is applicable to Interpolation and Decimation filter types. Used to specify an Integer or Fixed Fractional rate change.
- **Interpolation Rate Value:** This field is applicable to all Interpolation filter types and Decimation filter types for Fractional Rate Change implementations. The value provided in this field defines the up-sampling factor, or P for Fixed Fractional Rate (P/Q) resampling filter implementations.

- **Decimation Rate Value:** This field is applicable to the all Decimation and Interpolation filter types for Fractional Rate Change implementations. The value provided in this field defines the down-sampling factor, or Q for Fixed Fractional Rate (P/Q) resampling filter implementations.
- **Number of Channels:** The maximum number of interleaved data channels to be processed by the filter.

Hardware Oversampling Specification

- **Select format:** Selects which format is used to specify the hardware oversampling rate, the number of clock cycles available to the core to process an input sample and generate an output. This value directly affects the level of parallelism in the core implementation and resources used. When "Frequency Specification" is selected, the user specifies the Input Sampling Frequency and Clock Frequency. The ratio between these values along with other core parameters determine the hardware oversampling rate. When "Sample Period" is selected, the user specifies the integer number of clock cycles between input samples.
- **Input/Output Sample Period:** Integer number of clock cycles between input samples. When the multiple channels have been specified, this value should be the integer number of clock cycles between the time division multiplexed input sample data stream. When a fixed fractional decimation filter has been specified, this parameter specifies the integer number of clock cycles between output samples. Specifying the output sample period enables a more efficient use of the available clock cycles.
- **Input Sampling Frequency:** This field can be an integer or real value; it specifies the sample frequency for one channel. The upper limit is set based on the clock frequency and filter parameters such as Interpolation Rate and number of channels.
- **Clock Frequency:** This field can be an integer or real value. The limits are set based on the sample frequency, interpolation rate, and number of channels. **This field influences architecture choices only; the specified clock rate might not be achievable by the final implementation.**

Implementation Options Screen

The Implementation Options screen is used to define which filter architecture and coefficient structure to use and to configure the various datapath and coefficient options.

- **Filter Architecture:** Two filter architectures are supported: Systolic Multiply-Accumulate and Transpose Multiply-Accumulate.

Coefficient Options

- **Use Reloadable Coefficients**: When the "Reloadable" option is selected, a coefficient reload interface is provided on the core.
- **Coefficient Type:** The coefficient data can be specified as either signed or unsigned. When the signed option is selected, conventional two's complement representation is assumed.
- **Quantization:** Specifies the quantization method to be used. Available options are Integer Coefficients, Quantize Only, or Maximize Dynamic Range.
  - The Integer Coefficients option is only available when the filter coefficients have been specified using only integer values.
  - The Quantize Only option rounds the provided values to the nearest quantum using a simple rounding towards zero algorithm.
  - The Maximize Dynamic Range option scales all coefficients such that the maximum coefficient is equal to the maximum representable number in the specified bit width, thus maximizing the dynamic range of the filter (with the current implementation, overflow is not possible, as the accumulator width is automatically set to accommodate maximum bit growth within the filter). See the Coefficient Quantization section for more information.
- **Coefficient Width:** The bit precision of the filter coefficients. This field can be used with the filter response graph to explore the possibilities for more efficient implementation by limiting coefficient bit width to the minimum required to meet the user's target specification for the filter.

- **Best Precision Fraction Length:** When selected, the coefficient fractional width is automatically set to maximize the precision of the specified filter coefficients. See the Best Precision Fractional Length section for further information.
- **Coefficient Fractional Bits:** Specifies the number of coefficient bits that are used to represent the fractional portion of the provided filter coefficients. The maximum value it supports is the Coefficient Width value minus the required integer bit width. The integer bit width value is static and is automatically determined by calculating the integer bit width required to represent the maximum value contained in the provided coefficient sets. When the coefficient width is less than the required integer bit width, this field reports zero. When the required integer bit width is zero, this parameter can take a value greater than the Coefficient Width. See the Coefficient Quantization section for more information.
- **Coefficient Structure:** Four coefficient structures are supported: Non-symmetric, Symmetric, Negative Symmetric, and Half-band. The structure can also be inferred from the coefficient file directly (default setting), or specified directly. The inference algorithm only analyses the first 2048 coefficients. Only valid structure options, based on analysis of the provided coefficient file, are available for the user to specify directly.

Datapath Options

- **Number of Paths:** Specifies the number of parallel datapaths the filter is to process. Each parallel datapath is extended to a byte boundary, for both the input and output widths selected. The padding can be signed extended or set to zero.
- **Input Data Type:** The filter input data can be specified as either signed or unsigned. The signed option employs conventional two's complement arithmetic.
- **Input Data Width:** The precision (in bits) of the filter input data samples.
- **Input Data Fractional Bits:** The number of Input Data Width bits used to represent the fractional portion of the filter input data samples. This field is for information only. It is used in conjunction with Coefficient Fractional Bits to calculate the filter Output Fractional Bits value.
- **Output Rounding Mode:** Specifies the type of rounding to be applied to the output of the filter.
- **Output Width:** When using Full Precision, this field is disabled and indicates the output precision (in bits) of the filter output data samples, including bit growth. When using any other Rounding Mode, this field allows the user to specify the desired output sample width.
- **Output Fractional Bits:** This field reports the number Output Width bits used to represent the fractional portion of the filter output samples.

Detailed Implementation Options Screen

The Detailed Implementation Options screen is used to configure various control and implementation options.

- **Optimization Goal:** Specifies if the core is required to operate at maximum possible speed (Speed option) or minimum area (Area option). The Area option is the recommended default and normally achieves the best speed and area for the design; however in certain configurations, the Speed setting might be required to improve performance at the expense of overall resource usage. (This setting normally adds pipeline registers in critical paths.)

Memory Options

The memory type for MAC implementations can either be user-selected or chosen automatically to suit the best implementation options. Choosing "Distributed" can result in shift register implementation where appropriate to the filter structure. Forcing the RAM selection to be either "Block" or "Distributed" should be used with caution, as inappropriate use can lead to inefficient resource usage. The default "Automatic" mode is recommended for most users.

- **Data Buffer Type:** Specifies the type of RAM to be used to store data within a MAC element. Users can select either "Block" or "Distributed RAM" options, or select "Automatic" to allow the core to choose the memory type appropriately.
- **Coefficient Buffer Type:** Specifies the type of RAM to be used to store coefficients within a MAC element. Users can select either "Block" or "Distributed RAM" options, or select "Automatic" to allow the core to choose the memory type appropriately.
- **Input Buffer Type:** Specifies the type of RAM to be used to implement the data input buffer, where present. Users can select either "Block" or "Distributed RAM" options, or select "Automatic" to allow the core to choose the memory type appropriately.
- **Output Buffer Type:** Specifies the type of RAM to be used to implement the data output buffer, where present. Users can select either "Block" or "Distributed RAM" options, or select "Automatic" to allow the core to choose the memory type appropriately.
- **Preference for Other Storage:** Specifies the type of RAM to be used to implement general storage in the datapath. Users can select either "Block" or "Distributed RAM" options, or select "Automatic" to allow the core to choose the memory type appropriately. Because this covers several different types of storage, it is recommended that users specify this type of memory directly only if they really need to steer the core away from using a particular memory resource (for example, if they are short of block RAMs in their overall design).

DSP Slice Column Options

- **Multi-column Support:** For device families with XtremeDSP™ slices, implementations of large high speed filters might require chaining of DSP slice elements across multiple columns. Where applicable (the feature is only enabled for multi-column devices), the user can select the method of folding of the filter structure across the multiple columns, which can be "Automatic" (based on the selected device for the project) or "Custom" (user specifies the length of each column). The Multiple Column Filter implementation section describes the multi-column implementation in more detail.
- **Device Column Lengths:** Information only. Displays the column length pattern in a comma delimited list for the selected project device.
- **Column Configuration:** Specifies the individual column lengths in a comma delimited list. When "Automatic" has been selected, the column lengths are determined by the GUI starting with the first column in the device column pattern. When "Custom" is selected, the user can specify the desired column pattern. The number of columns might not exceed that available in the selected device, and the individual column lengths must sum to the number of DSP slices utilized by current filter configuration. When the selected device has various column lengths, it might be desirable to skip a particular column; this can be done by specifying a zero column length, for example 10,0,22. **The specified column configuration does not guarantee that the downstream tools place the columns in the desired sequence.**
- **Inter-column Pipe Length:** Pipeline stages are required to connect between the columns, with the level of pipelining required being dependent upon the required system clock rate, the chosen device, and other system-level parameters. Choice of this parameter is always left for the user to specify.

Data Channel Options

- **TLAST:** TLAST can either be Not Required, in which case the core does not have the port, or Vector Framing, where TLAST is expected to denote the last sample of an interleaved cycle of data channels, or Packet Framing, where the core does not interpret TLAST, but passes the signal to the output DATA channel TLAST with the same latency as the datapath.
- **Output TREADY:** This field enables the `m_axis_data_tready` port. With this port enabled, the core supports back-pressure. Without the port, back-pressure is not supported, but resources are saved and performance is likely to be higher.
- **Input FIFO:** Selects a FIFO interface for the S_AXIS_DATA channel. When the FIFO has been selected data can be transferred in a continuous burst up to the size of the FIFO (default 16) or, if greater, the number of interleaved data channels. The FIFO requires additional fabric resources.

- **TUSER Input:** The input TUSER port can independently and optionally convey a User Field and/or a Chan ID Field, giving four options.
- **TUSER Output:** The output TUSER port can optionally carry a User Field and/or a Chan ID Field. The presence of a User field in this port is coupled to the presence of a User Field in the TUSER input selection, because the User Field, if present, is not interpreted by the core, but conveyed from input DATA channel to Output Channel with the same latency as the datapath to ease system design.
- **User Field Width: ra**nge 1 to 256 bits.

See TUSER Options, page 20 of the Input and Output DATA Channels, page 19 for further details.

Configuration Channel Options

The `CONFIG` channel is used to select the active filter coefficient set. The channel is also used to apply newly reload filter coefficients. See CONFIG Channel, page 21 for full details.

- **Synchronization Mode:**
  - **On Vector:** Configuration packets, when available, are consumed and their contents applied when the first sample of an interleaved data channel sequence is processed by the core. When the core is configured to process a single data channel configuration packets are consumed every processing cycle of the core.
  - **On Packet:** Further qualifies the consumption of configuration packets. Packets are only consumed after the core has received a transaction on the `s_axis_data` channel where `s_axis_data_tlast` has been asserted.
- **Configuration Method**
  - **Single:** A single coefficient set is used to process all interleaved data channels.
  - **By Channel:** A unique coefficient set is specified for each interleaved data channel.

Reload Channel Options

- **Reload Slots:** Range 1 to 256. Specifies the number of coefficient sets that can be loaded in advance. Reloaded coefficients are only applied to the core after a configuration packet has been consumed. See RELOAD Channel, page 22 and CONFIG Channel, page 21 for more details.

Control Signals

- **aclken:** Determines if the core has the `aclken` pin.
- **aresetn:** Determines if the core has the `aresetn` pin. `aresetn` is active low and it is recommended that when asserted, it is asserted for a minimum of 2 clock cycles.
- **Reset data vector:** Specifies if aresetn resets the data vector and the control signals or just the control signals. Data vector reset requires additional fabric resources. When no data vector reset has been selected an additional data_valid field is present in the m_axis_data_tuser bus which can be used as further qualification of the core's output data. See Resetting the Core, page 24 and Input and Output DATA Channels TUSER Options, page 20 for more details.

Summary Screen

The Summary screen provides a summary of core options selected.

Summary: The final page provides summary information about the core parameters selected, which includes information on the actual number of calculated coefficients, including padding; the inferred or specified coefficient structure; the additional gain incurred as data passes through the filter due to maximizing the coefficient dynamic range during quantization; the specified output width along with the full precision width for comparison; the calculated cycle-latency value; and the latency delta from the previous major revision of the core.

Using the FIR Compiler IP Core

The CORE Generator software GUI performs error-checking on all input parameters. Resource estimation, implementation details, and filter analysis are also available.

Several files are produced when a core is generated, and customized instantiation templates for Verilog and VHDL design flows are provided in the .veo and .vho files, respectively. For detailed instructions, see the CORE Generator software documentation.

Simulation Models

The core has a number of options for simulation models:

- VHDL behavioral model in the xilinxcorelib library
- VHDL UniSim-based structural simulation model
- Verilog UniSim-based structural simulation model

The models required can be selected in the CORE Generator software project options.

Xilinx recommends that simulations utilizing UniSim-based structural models are run using a resolution of 1 ps. Some Xilinx library components require a 1 ps resolution to work properly in either functional or timing simulation. The UniSim-based structural simulation models might produce incorrect results if simulated with a resolution other than 1 ps. See the “Register Transfer Level (RTL) Simulation Using Xilinx Libraries” section in *Chapter 6 of the Synthesis and Simulation Design Guide* for more information. This document is part of the ISE® Software Manuals set available at www.xilinx.com/support/software_manuals.htm.

XCO Parameters

Table 4 defines valid entries for the XCO parameters. Parameters are not case sensitive. Default values are displayed in bold. Xilinx strongly suggests that XCO parameters are not manually edited in the XCO file; instead, use the CORE Generator software GUI to configure the core and perform range and parameter value checking. The XCO parameters are helpful in defining the interface to other Xilinx tools.

Table 4: **XCO Parameters**

XCO Parameter	Valid Values
component_name	ASCII text using characters: a..z, 0..9 and "_" starting with a letter
CoefficientSource	**Vector**, COE_File
CoefficientVector	ASCII text using characters: 0..9, "." and ","
Coefficient_File	Valid file path
Coefficient_Sets	1 - 256
Filter_Type	**Single_Rate**, Interpolation, Decimation
Rate_Change_Type	**Integer**, Fixed_Fractional
Interpolation_Rate	1 - 1024
Decimation_Rate	1 - 1024
Number_Channels	1 - 64
RateSpecification	**Frequency_Specification**, Sample_Period
SamplePeriod	1 - 10000000
Sample_Frequency	0.000001 - 600.0

Table 4: **XCO Parameters *(Cont'd)***

XCO Parameter	Valid Values
Clock_Frequency	0.000001 - 600.0
Filter_Architecture	Systolic_Multiply_Accumulate, Transpose_Multiply_Accumulate
Coefficient_Reload	**false**, true
Coefficient_Sign	**Signed**, Unsigned
Quantization	**Integer_Coefficients**, Quantize_Only, Maximize_Dynamic_Range
Coefficient_Width	1 - 49; Default is **16**
BestPrecision	**true**, false
Coefficient_Fractional_Bits	0 - 49
Coefficient_Structure	**Inferred**, Non_Symmetric, Symmetric, Negative_Symmetric, Half_Band
Data_Sign	**Signed**, Unsigned
Data_Width	1 - 49; Default is **16**
Data_Fractional_Bits	0 - 49
Number_Paths	1 - 16; Default is **1**
Output_Rounding_Mode	**Full_Precision**, Truncate_LSBs, Non_Symmetric_Rounding_Down, Non_Symmetric_Rounding_Up, Symmetric_Rounding_to_Zero, Symmetric_Rounding_to_Infinity, Convergent_Rounding_to_Even, Convergent_Rounding_to_Odd
Output_Width	1 - 89
Optimization_Goal	**Area**, Speed
Data_Buffer_Type	**Automatic**, Block, Distributed, Not_Applicable
Coefficient_Buffer_Type	**Automatic**, Block, Distributed, Not_Applicable
Input_Buffer_Type	**Automatic**, Block, Distributed, Not_Applicable
Output_Buffer_Type	**Automatic**, Block, Distributed, Not_Applicable
Preference_For_Other_Storage	**Automatic**, Block, Distributed, Not_Applicable
Multi_Column_Support	**Disabled**, Automatic, Custom
Inter_Column_Pipe_Length	1 - 16; Default is **4**
ColumnConfig	ASCII text using characters: 0..9 and ","
DATA_Has_TLAST	**Not_Required**, Vector_Framing, Packet_Framing
M_DATA_Has_TREADY	**false**, true
S_DATA_Has_FIFO	**true**, false
S_DATA_Has_TUSER	**Not_Required**, User_Field, Chan_ID_Field, User_and_Chan_ID_Field
M_DATA_Has_TUSER	**Not_Required**, User_Field, Chan_ID_Field, User_and_Chan_ID_Field
DATA_TUSER_Width	Range **1** to 256
S_CONFIG_Sync_Mode	**On_Vector,** On_Packet
S_CONFIG_Method	**Single**, By_Channel
Num_Reload_Slots	Range **1** to 256.
Has_ACLKEN	**false**, true
Has_ARESETn	**false**, true
Reset_Data_Vector	**true**, false

Demonstration Test Bench

When the core is generated using the CORE Generator tool, a demonstration test bench is created. This is a simple VHDL test bench that exercises the core.

The demonstration test bench source code is one VHDL file: `demo_tb/tb_<component_name>.vhd` in the CORE Generator software output directory. The source code is comprehensively commented.

Using the Demonstration Test Bench

The demonstration test bench instantiates the generated FIR Compiler core. Either the behavioral model or the netlist can be simulated within the demonstration test bench.

- Behavioral model: Ensure that the CORE Generator software project options are set to generate a behavioral model. After generation, this creates a behavioral model wrapper named `<component_name>.vhd`. Compile this file into the work library (see your simulator documentation for more information on how to do this).
- Netlist: If the CORE Generator software project options were set to generate a structural model, a VHDL or Verilog netlist named `<component_name>.vhd` or `<component_name>.v` was generated. If this option was not set, generate a netlist using the netgen program, for example:

  ```
  netgen -sim -ofmt vhdl <component_name>.ngc <component_name>_netlist.vhd
  ```

  Compile the netlist into the work library (see your simulator documentation for more information on how to do this).

Compile the demonstration test bench into the work library. Then simulate the demonstration test bench. View the test bench's signals in your simulator's waveform viewer to see the operations of the test bench.

The Demonstration Test Bench in Detail

The demonstration test bench performs the following tasks:

- Instantiate the core
- Generate a clock signal
- Drive the core's input signals to demonstrate core features (see below for details)
- Checks that the core's output signals obey AXI protocol rules (data values are not checked to keep the test bench simple)
- Provide signals showing the separate fields of AXI TDATA and TUSER signals

The demonstration test bench drives the core's input signals to demonstrate the features and modes of operation of the core. An impulse is used as input data in all operations; the corresponding output of the core is therefore the impulse response of the filter, that is, the filter coefficients.

The operations performed by the demonstration test bench are appropriate for the configuration of the generated core, and are a subset of the following operations:

- Drive an impulse
- Drive an impulse, demonstrating AXI handshaking signals by modifying the input data rate using slave data channel TVALID, and modifying the output data rate using master data channel TREADY (if present)
- Drive an impulse, during which deassert clock enable (if present), then assert reset (if present) and drive a new impulse
- For multiple paths: drive a set of impulses of different magnitudes on each path
- For multiple channels: drive a set of impulses of different magnitudes on each channel

- For multiple filter coefficient sets: select a different coefficient set (a different set for each channel, if supported); drive an impulse (on each channel, if there are multiple channels)
- For reloadable coefficients: load a new coefficient set; drive an impulse (on each channel, if there are multiple channels)

Customizing the Demonstration Test Bench

It is possible to modify the demonstration test bench to drive the core's inputs with different data or to perform different operations.

All operations performed by the demonstration test bench to drive the core's inputs are done in the `stimuli` process. This process also contains procedures to simplify driving input data. The `drive_data` procedure drives one or more input data samples with the specified data, controlling AXI signals to adhere to the AXI protocol and keep to the core's configured input sample rate. The `drive_impulse` procedure drives an impulse input, with enough zero-valued samples to allow time for the impulse response to emerge on the core's output data channel. To drive input data other than an impulse, either use the `drive_data` procedure repeatedly with specific input data values, or copy and modify the `drive_impulse` procedure.

The stimuli process is comprehensively commented, to explain clearly what is being done. New data, configuration and reload operations can be added by copying and modifying sections of this process.

The clock frequency of the core can be modified by changing the `CLOCK_PERIOD` constant.

System Generator for DSP Graphical User Interface

This section describes each tab of the System Generator GUI and details the parameters that differ from the CORE Generator software GUI. See CORE Generator Graphical User Interface for detailed information about all other parameters.

Tab 1: Filter Specification

The Filter Specification tab is used to define the basic filter configuration as on the Filter Specification Screen of the CORE Generator software GUI.

- **Coefficients:** This field is used to specify the coefficient vector as a single MATLAB® software row vector. The number of taps is inferred from the length of the MATLAB software row vector. It is possible to enter these coefficients using the MATLAB software FDATool block. Multiple coefficient sets must be concatenated into a single vector as described in the Multiple Coefficient Sets section.
- **Hardware Oversampling Specification format:** Selects which method is used to specify the hardware oversampling rate and determines the level of control and rate abstraction utilized by the core. This value directly affects the level of parallelism of the core implementation and resources used.

  When "Maximum Possible" is selected, the core uses the maximum oversampling given the sample period of the signal connected to `s_data_tdata` port. The `s_data_tvalid` handshake signal is abstracted and automatically driven by System Generator and the core propagates the data streams sample period.

  When "Hardware Oversampling Rate" is selected, the user can specify the oversampling rate relative to the input sample period of the core. As with "Maximum Possible" the handshake and sample period are managed automatically by System Generator.

  When "Sample Period" is selected there is no automatic handshaking, `s_data_tvalid` is exposed, or rate abstraction, all core ports are considered as having a normalized sample period 1. The core clock is connected to the system clock. The core must be controlled using the full AXI4-Stream protocol, see AXI4-Stream Considerations, page 18 for full details.

- **Sample Period**: Specifies the input sample period supported by the core.
- **Hardware Oversampling Rate**: Specifies the hardware oversampling rate to be applied to the core.

See Filter Specification Screen for information about the other parameters on this tab.

Tab 2: Implementation

The Implementation tab is used to define implementation options; see the Implementation Options Screen of the CORE Generator software GUI for details of all the core parameters on this tab.

- **FPGA Area Estimation:** See the System Generator documentation for detailed information about this section.

See the Implementation Options Screen for information about the other parameters on this tab.

Tab 3: Detailed Implementation

See Detailed Implementation Options Screen for the corresponding CORE Generator software GUI screen.

The TUSER User Field width parameter is abstracted by System Generator and is defined by the signal connected to the core.

Data vector reset is always selected to ensure the simulation model and implementation remain bit and cycle accurate.

Core Use through System Generator for DSP

The FIR Compiler core is available through Xilinx System Generator for DSP, a design tool that enables the use of The MathWorks model-based design environment Simulink® software for FPGA design. The FIR Compiler core is one of the DSP building blocks provided in the Xilinx blockset for Simulink software. The core can be found in the Xilinx Blockset in the DSP section. The block is called "FIR Compiler v6.2." See the System Generator for DSP User Manual for more information.

AXI4-Stream Considerations

The conversion to AXI4-Stream interfaces brings standardization and enhances interoperability of Xilinx LogiCORE IP solutions. Other than general control signals such as `aclk`, `aclken` and `aresetn` and the event outputs all inputs and outputs to the FIR Compiler are conveyed via AXI4-Stream channels. A channel consists of TVALID and TDATA always, plus several optional ports. In the FIR Compiler, the optional ports supported are TREADY, TLAST and TUSER. Together, TVALID and TREADY perform a handshake to transfer a message, where the payload is TDATA, TUSER and TLAST. The FIR Compiler operates on the data contained in the input DATA channel TDATA port (`s_axis_data_tdata`) and outputs the result in the TDATA field of the output DATA channel (`m_axis_data_tdata`). The FIR Compiler optionally uses the TUSER and TLAST fields to indicate the phase of a cycle of time-multiplexed channels. The core also provides the facility to convey a user field within TUSER and the TLAST signal from input DATA channel to the output DATA channel with the same latency as for TDATA. This facility is intended to ease the use of the FIR Compiler in a system. For example, the FIR Compiler can be used to filter packetized data. In this example, the TLAST has no bearing on the FIR, but the core can be configured pass the TLAST of the packetized data channel saving the system designer the effort of constructing a bypass path for this information.

For further details on AXI4-Stream Interfaces see the Xilinx AXI Design Reference Guide (UG761) and the AMBA 4 AXI4-Stream Protocol Version: 1.0 Specification.

Basic Handshake

Figure 4 shows the transfer of data in an AXI4-Stream channel. TVALID is driven by the source (master) side of the channel and TREADY is driven by the receiver (slave). TVALID indicates that the value in the payload fields (TDATA, TUSER and TLAST) is valid. TREADY indicates that the slave is ready to receive data. When both TVALID and TREADY are true in a cycle, a transfer occurs. The master and slave set TVALID and TREADY respectively for the next transfer appropriately. Some channels can be configured to have no TREADY, in which case the channel behaves as through there was an implicit, permanently asserted TREADY.

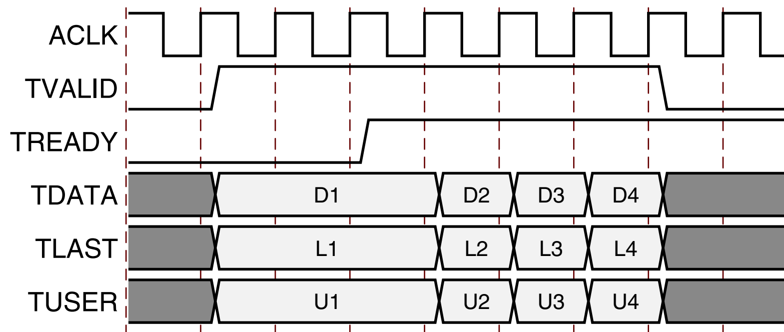


Figure 4: **Data Transfer in an AXI4-Stream Channel**

Input and Output DATA Channels

The basic operation of the FIR is for samples to enter via the input DATA channel (`s_axis_data_t*`) and exit via the output DATA channel (`m_axis_data_t*`) duly filtered. The output channel optionally supports TREADY which allows a resource/behavior trade-off. In circumstances where downstream slave can be guaranteed to accept the maximum bandwidth of the FIR,TREADY can be deselected to save resources. The input DATA channel always supports TREADY.

TREADY and TVALID

All AXI4-Stream channels support TVALID. The input DATA channel also always supports TREADY. The output channel optionally supports TREADY. Back-pressure from the output channel eventually propagates to the input DATA channel to ensure that no data is dropped.

TDATA Structure

The input DATA and output DATA channels share a common TDATA structure format, though can have different bit widths. All parallel datapaths, see Parallel Data Channel Filters, page 48 for more details, are contained in the TDATA bus, with each path being sign extended to an 8-bit boundary. The extra bits on the input TDATA are not used by the core.

Figure 5 shows the TDATA structure for a case with 2 parallel paths (data streams). In this case, bit growth is experienced between input and output. For a path width of 11 bits on input growing to 13 bits on output, the values of the various bus indices shown in the diagram are as follows: A= 31, B = 26, C=15, D = 10, E = 31, F = 28, G=15, H = 12.

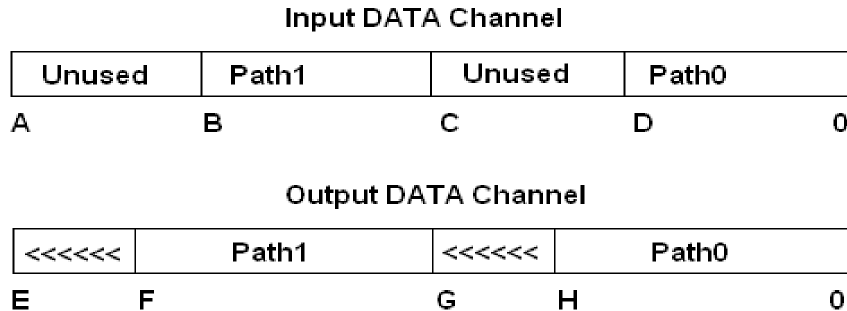


Figure 5: **TDATA Structure for Input and Output DATA Channels**

TLAST Options

On the input DATA channel and output DATA channel, TLAST can optionally be used to indicate the last sample in a cycle of interleaved data channels. This use is termed 'vector-based'. The input DATA and output DATA channels also support a mode in which the TLAST is passed from input to output with latency equivalent to the TDATA samples. This mode is termed 'packet-based' and is intended to ease system design.

TUSER Options

The input DATA channel and output DATA channels optionally support a TUSER field. For each, the TUSER field can be used to convey a User Field and/or a Channel ID field. When both are selected, they are concatenated, with Channel ID in the least significant bit positions. When User Field is selected on the input channel it is automatically selected for the output channel, as this User Field, like 'packet-based' TLAST is a facility whereby the User Field is passed through the core, but subject to the same latency delay as the TDATA path from input to output. This is intended to ease system design. The User Field has user-selected width.

The Channel ID field has the minimum width required to describe the number of channels in a time-division multiplex cycle (log2roundup(number_of_channels)), for example. with 13 channels, channel ID is 4 bits wide.

The output DATA channel also includes a Data Valid field when aresetn has been selected without Data vector reset being selected. This field can be used for additional validation of the m_axis_data_tdata bus. See Resetting the Core, page 24for more details. The Data Valid field occupies the LSB of m_axis_data_tuser with the other TUSER fields, when selected, being shifted up the bus.

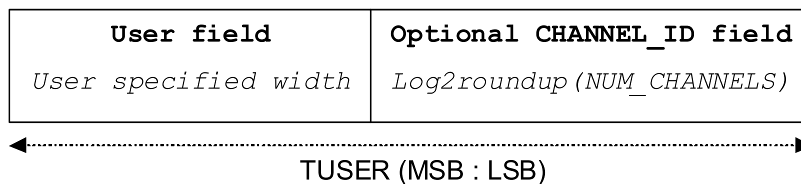


Figure 6: **TUSER**

When the core has been configured to implement a rate change the following rules are applied to TUSER and TLAST.

- When the core is configured with no rate change TUSER and TLAST propagate through the core unmodified.
- When the core is configured to up convert by X the input TUSER and TLAST are duplicated on the last sample of the corresponding block of X output samples. TUSER is undefined for the other X-1 output samples.
- When the core is configured to down convert by X the TUSER value for a given output sample is taken from the TUSER value of the first input sample of the corresponding X input samples. TLAST is OR'd over X input samples with the result being used for the TLAST of the corresponding output sample.

CONFIG Channel

This control channel specifies the filter select value for each (or all) interleaved data channels and it also activates reloaded filter coefficients.

- When the core has been specified to support multiple filter coefficients, the filter select value selects which filter should be used for each of the interleaved data channels.
- When the core is specified to support reloadable filter coefficients, receipt of a filter configuration packet updates to (or switches in) any reloaded filter coefficient sets since the previous update.

  Note: When the core is specified to full rate and no rate change, care must be taken to give the filter an opportunity to acknowledge/store the reloaded filters. If the Filter Configuration Channel is continuously updated, there is no opportunity to store the reloaded filters and the reload channel is blocked when all the reload slots are full. The time required to process a single input vector (block of interleaved channels) is sufficient to update the reload filters.
- The channel can be configured to have a packet of length of "Number of Channels" where each transaction in the packet specifies the filter select value of the corresponding interleaved channel.
- The channel can also be configured to have a packet length of 1 where the single transaction specifies the filter select value for all of the interleaved channels.

Blocking Behavior

- The channel is non-blocking to the data channel. The data channel is not halted if no new configuration data is present.
- The channel is blocking to the reload channel. When all the reload slots are full the reload channel is blocked until a configuration packet is received and processed.

Packet Consumption Rate and Synchronization

When a complete packet has been received the user can specify the core to synchronize the Configuration channel to the input Data channel in two methods:

- **Vector Synchronization (On Vector)**: Configuration packets, when available, are consumed and their contents used when the first sample of an interleaved data channel sequence is processed by the core. When the core is configured to process a single data channel configuration, packets are consumed every processing cycle of the core.
  - For down sampling (decimation) implementations configuration packets are only consumed on the first phase of a down sampling period.
- **Packet Synchronization (On Packet)**: Further qualifies the consumption of configuration packets. Packets are only consumed when the core has received a transaction on the `s_axis_data` channel where `s_axis_data_tlast` has been asserted. This options ties the rate at which configuration packets are consumed to the input DATA channel rather than to the rate at which the configuration packets are provided to the core, that is, configuration packets can be queued in advance and then used at a rate controlled by the input DATA channel.

TREADY

Inputs to the CONFIG channel are stored in a buffer until consumed. When this buffer is almost full, TREADY is deasserted in accordance with AXI4-Stream protocol.

TLAST Options

TLAST must be asserted to indicate the last transaction in the configuration packet. If the packet is of length 1 then TLAST is not required and is disabled. In this case each transaction is considered to be a complete packet. If TLAST last is incorrectly asserted a warning is reported on the event interface.

TDATA

TDATA bus is zero padded to an 8-bit boundary.

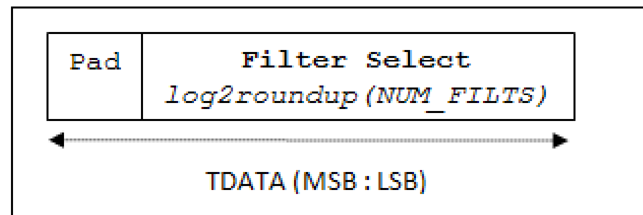


Figure 7: **TDATA (MSB:LSB)**

RELOAD Channel

This channel is used to sequentially load a new filter. When the core is configured to have multiple filter sets, the first transaction defines which filter is to be reloaded. At generate time the core is configured to support a number of reload slots. This defines how many filter sets can be reloaded before a synchronization event occurs and applies the new filter sets to the core. Consumption of a configuration packet on the CONFIG channel (S_AXIS_CONFIG) is used to, synchronize or, update to the newly reloaded filter sets.

The RELOAD channel packet length is derived from the number of coefficients specified at core generation time and the filter implementation utilized. See sections Coefficient Reload, page 48 and Tab 4: Coefficient Reload, page 8 for details on how to generate the content for the channel. As with the CONFIG channel, the last sample of the packet must be qualified by an asserted TLAST. The set of data loaded into the RELOAD channel does not take action until triggered by a reconfiguration synchronization event as described above.

TREADY

When all the reload filter slots are nearly full, TREADY is deasserted in accordance with AXI4-Stream protocol to prevent data loss.

TLAST

As with the CONFIG channel, TLAST on the RELOAD channel is associated with two event ports (`event_s_reload_tlast_missing` and `event_s_reload_tlast_unexpected`) which likewise indicate for a single cycle TLAST missing or TLAST asserted when not expected anomalies respectively.

TDATA

The TDATA bus is zero padded to an 8-bit boundary. As this is an input, the pad bits are ignored.

The following diagrams shows the format and example timing of TDATA into the RELOAD channel.

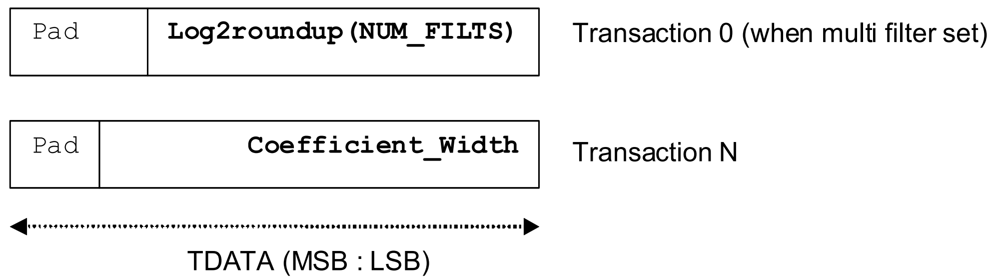


Figure 8: **TDATA Format**

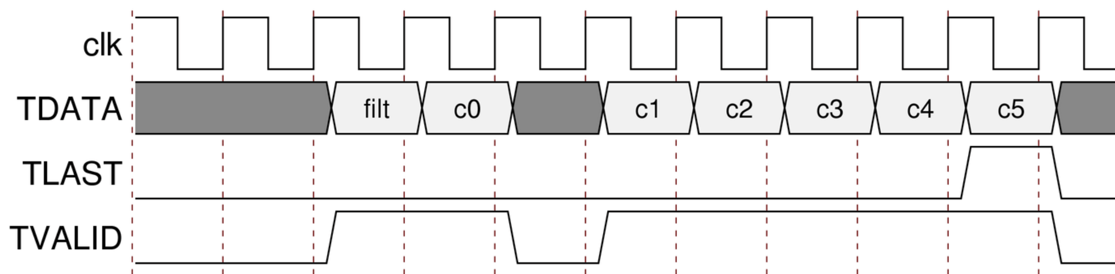


Figure 9: **TDATA Example Timing**

Event Interface

The event interface is a collection of individual pins, each of which is asserted for a single clock cycle to give external notice of an internal event. These events can be considered as errors or ignored by the external system. The individual event signals are:

`event_s_data_tlast_missing:` enabled when TLAST is set to vector-based for the input DATA channel; this event signal is asserted on the last transaction of an incoming vector is `s_axis_data_tlast` is not asserted.

`event_s_data_tlast_unexpected:` enabled when TLAST is set to vector-based or packet-based when down converting for the input DATA channel; this event signal is asserted on any transaction when `s_axis_data_tlast` asserted unexpectedly.

`event_s_data_chanid_incorrect:` enabled when the TUSER mode selects TUSER to have a chan ID field; this is asserted on every transaction when the `s_axis_data_tuser` Channel ID field does not match the value expected by the core.

`event_s_config_tlast_missing:` enabled when the CONFIG channel is enabled; this signal is asserted on the last transaction of an incoming vector if `s_axis_config_tlast` is not seen asserted.

`event_s_config_tlast_unexpected:` enabled when the CONFIG channel is enabled, this signal is asserted when s_axis_config_tlast is seen asserted unexpectedly.

`event_s_reload_tlast_missing:` enabled when the RELOAD channel is enabled; this signal is asserted on the last transaction of an incoming vector if `s_axis_config_tlast` is not seen asserted.

`event_s_reload_tlast_unexpected:` enabled when the RELOAD channel is enabled; this signal is asserted when `s_axis_config_tlast` is seen asserted unexpectedly.

Resetting the Core

The `aresetn` port is an optional active low input port which, when asserted for a minimum of two cycles, forces the internal control logic to the initialized condition and optionally clears the core's data vector. Selecting data vector reset can result in the core using more fabric resources.

When data vector reset has **not** been selected no internal data is cleared from the filter memories during the reset process. The filter output remains dependent on the prior input samples and as a result the m_axis_data_tdata bus of the behavioral simulation file (rather than a structural simulation file) might not match the generated netlist until the filter data memory is completely flushed. The data_valid field of the m_axis_data_tuser bus, see TUSER Options, page 20, indicates when the filter data memory has been completely flushed and can be used as additional qualification of the m_axis_data_tdata bus.

Interface Timing

Figure 10 shows the sequence of events from a packet of reload data being written to the RELOAD channel (start of first arrow), which is triggered for use on the arrival and consumption of a packet on the CONFIG channel (end of first arrow and start of second arrow), and on to the data stream.

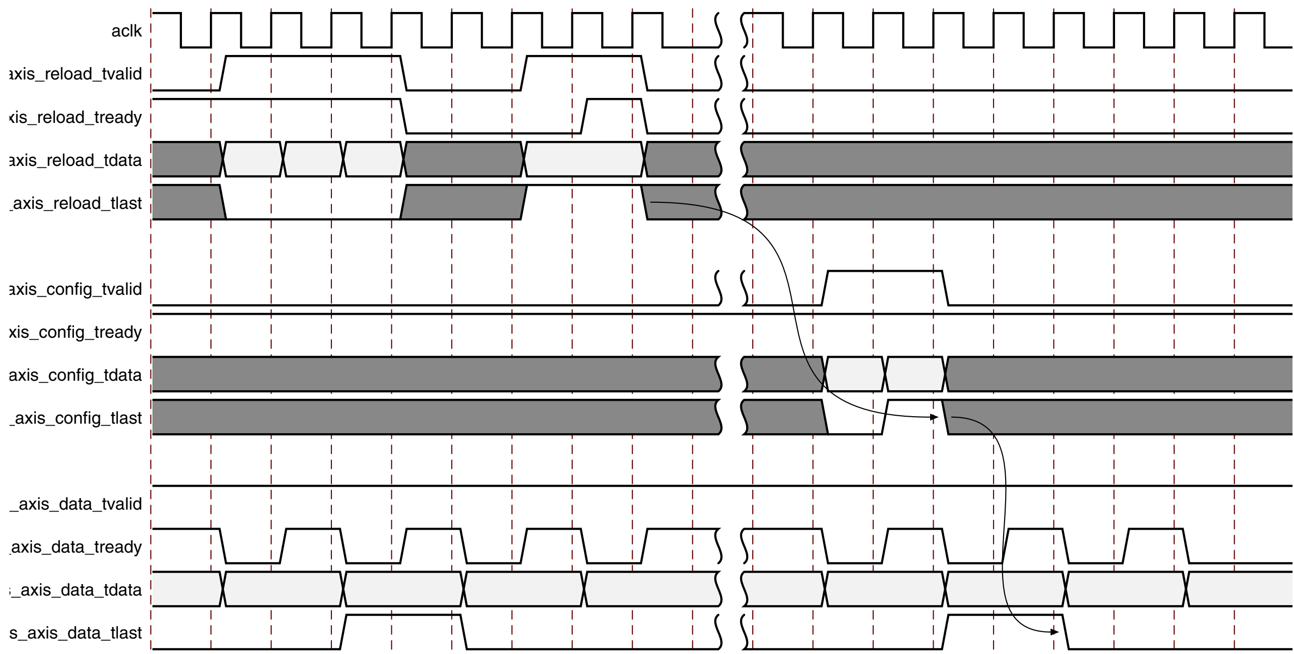


Figure 10: **Interface Timing**

Migrating to FIR Compiler v6.2 from Earlier Versions

Updating from FIR Compiler v6.0 and v6.1

There are no XCO parameter, port or latency changes between v6.2, v6.1 and v6.0 of the FIR Compiler, only additional parameters.

Updating from FIR Compiler v5.0

XCO Parameter Changes

The CORE Generator core update functionality can be used to update an existing XCO file from v5.0 to FIR Compiler v6.2, but it should be noted that the update mechanism alone does not create a core compatible with v5.0. See Instructions for Minimum Change Migration. FIR Compiler v6.2 has additional AXI4-Stream parameters. The following table shows the changes to XCO parameters from version 5.0 to version 6.2.

Table 5: **XCO Parameter Changes from v5.0 to v6.2**

Version v5.0	Version 6.2	Notes
component_name	component_name	Unchanged
CoefficientSource	CoefficientSource	Unchanged
CoefficientVector	CoefficientVector	Unchanged
Coefficient_File	Coefficient_File	Unchanged
Coefficient_Sets	Coefficient_Sets	Unchanged
Filter_Type	Filter_Type	Unchanged
Rate_Change_Type	Rate_Change_Type	Unchanged
Interpolation_Rate	Interpolation_Rate	Unchanged
Decimation_Rate	Decimation_Rate	Unchanged
Zero_Pack_Factor	Zero_Pack_Factor	Deprecated
Number_Channels	Number_Channels	Unchanged
RateSpecification	RateSpecification	Unchanged
SamplePeriod	SamplePeriod	Unchanged
Sample_Frequency	Sample_Frequency	Unchanged
Clock_Frequency	Clock_Frequency	Unchanged
Filter_Architecture	Filter_Architecture	Unchanged
Coefficient_Reload	Coefficient_Reload	Unchanged
Coefficient_Sign	Coefficient_Sign	Unchanged
Quantization	Quantization	Unchanged
Coefficient_Width	Coefficient_Width	Unchanged
BestPrecision	BestPrecision	Unchanged
Coefficient_Fractional_Bits	Coefficient_Fractional_Bits	Unchanged
Coefficient_Structure	Coefficient_Structure	Unchanged
Data_Sign	Data_Sign	Unchanged
Data_Width	Data_Width	Unchanged
Data_Fractional_Bits	Data_Fractional_Bits	Unchanged

Table 5: **XCO Parameter Changes from v5.0 to v6.2** ***(Cont'd)***

Version v5.0	Version 6.2	Notes
Number_Paths	Number_Paths	Unchanged
Output_Rounding_Mode	Output_Rounding_Mode	Unchanged
Output_Width	Output_Width	Unchanged
Allow_Rounding_Approximation		Deprecated
Registered_Output		Deprecated
Optimization_Goal	Optimization_Goal	Unchanged
Has_SCLR	Has_ARESETn	Name change. aresetn is active low.
Has_CE	Has_ACLKEN	Name change.
Has_ND		Deprecated. These options pertain to signals which have been replaced in the move to AXI4-Stream interfaces.
Has_Data_Valid		
SCLR_Deterministic		
UseChan_in_adv		
Chan_in_adv		
Data_Buffer_Type	Data_Buffer_Type	Unchanged
Coefficient_Buffer_Type	Coefficient_Buffer_Type	Unchanged
Input_Buffer_Type	Input_Buffer_Type	Unchanged
Output_Buffer_Type	Output_Buffer_Type	Unchanged
Preference_For_Other_Storage	Preference_For_Other_Storage	Unchanged
Multi_Column_Support	Multi_Column_Support	Unchanged
Inter_Column_Pipe_Length	Inter_Column_Pipe_Length	Unchanged
ColumnConfig	ColumnConfig	Unchanged
	DATA_Has_TLAST	Pertains to AXI4-Stream interfaces.
	M_DATA_Has_TREADY	Pertains to AXI4-Stream interfaces.
	S_DATA_Has_FIFO	New to version 6.2. Pertains to AXI4-Stream interfaces.
	S_DATA_Has_TUSER	Pertains to AXI4-Stream interfaces.
	M_DATA_Has_TUSER	Pertains to AXI4-Stream interfaces.
	DATA_TUSER_Width	Pertains to AXI4-Stream interfaces.
	S_CONFIG_Sync_Mode	Pertains to AXI4-Stream interfaces.
	S_CONFIG_Method	Pertains to AXI4-Stream interfaces.
	Num_Reload_Slots	Pertains to the coefficient reload feature.
	Reset_Data_Vector	New to version 6.2.

For more information on this upgrade feature, see the CORE Generator software documentation.

Port Changes

Table 6 details the changes to port naming, additional or deprecated ports and polarity changes from v5.0 to v6.2.

Table 6: **Port Changes from Version 5.0 to Version 6.2**

Version 5.0	Version 6.2	Notes
CLK	aclk	Rename only
CE	aclken	Rename only
SCLR	aresetn	Rename and change of sense (now active low)
ND	s_axis_data_tvalid	Equivalent to s_axis_data_tvalid. See s_axis_data_t* below.
FILTER_SEL		Replaced by CONFIG channel. See s_axis_config_t* below
COEF_LD		Replaced by RELOAD channel (see s_axis_reload_t* below)
COEF_WE		
COEF_DIN		
COEF_FILTER_SEL		
RFD	s_axis_data_tready	
RDY	m_axis_data_tvalid	
DATA_VALID		Deprecated, see s_axis_data_t* below
CHAN_IN		Deprecated. Function performed by s_axis_data_tuser (chan ID field) or s_axis_data_tlast (vector-based).
CHAN_OUT		Deprecated. Function performed by m_axis_data_tuser (chan ID field) or m_axis_data_tlast (vector-based)
DIN		Deprecated. Now exists as a field within s_axis_data_tdata
DOUT		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_I		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_Q		Deprecated. Now exists as a field within m_axis_data_tdata.
DIN_1		Deprecated. Now exists as a field within s_axis_data_tdata
DIN_2		Deprecated. Now exists as a field within s_axis_data_tdata
DIN_3		Deprecated. Now exists as a field within s_axis_data_tdata
DIN_4		Deprecated. Now exists as a field within s_axis_data_tdata
DIN_5		Deprecated. Now exists as a field within s_axis_data_tdata
DIN_6		Deprecated. Now exists as a field within s_axis_data_tdata
DIN_7		Deprecated. Now exists as a field within s_axis_data_tdata
DIN_8		Deprecated. Now exists as a field within s_axis_data_tdata
DIN_9		Deprecated. Now exists as a field within s_axis_data_tdata
DIN_10		Deprecated. Now exists as a field within s_axis_data_tdata
DIN_11		Deprecated. Now exists as a field within s_axis_data_tdata
DIN_12		Deprecated. Now exists as a field within s_axis_data_tdata
DIN_13		Deprecated. Now exists as a field within s_axis_data_tdata
DIN_14		Deprecated. Now exists as a field within s_axis_data_tdata
DIN_15		Deprecated. Now exists as a field within s_axis_data_tdata
DIN_16		Deprecated. Now exists as a field within s_axis_data_tdata
DOUT_1		Deprecated. Now exists as a field within m_axis_data_tdata.

Table 6: **Port Changes from Version 5.0 to Version 6.2 *(Cont'd)***

Version 5.0	Version 6.2	Notes
DOUT_I_1		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_Q_1		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_2		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_I_2		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_Q_2		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_3		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_I_3		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_Q_3		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_4		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_I_4		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_Q_4		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_5		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_I_5		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_Q_5		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_6		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_I_6		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_Q_6		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_7		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_I_7		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_Q_7		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_8		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_I_8		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_Q_8		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_9		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_I_9		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_Q_9		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_10		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_I_10		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_Q_10		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_11		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_I_11		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_Q_11		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_12		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_I_12		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_Q_12		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_13		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_I_13		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_Q_13		Deprecated. Now exists as a field within m_axis_data_tdata.

Table 6: **Port Changes from Version 5.0 to Version 6.2** ***(Cont'd)***

Version 5.0	Version 6.2	Notes
DOUT_14		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_I_14		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_Q_14		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_15		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_I_15		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_Q_15		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_16		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_I_16		Deprecated. Now exists as a field within m_axis_data_tdata.
DOUT_Q_16		Deprecated. Now exists as a field within m_axis_data_tdata.
	s_axis_data_tvalid	TVALID for input DATA channel
	s_axis_data_tready	TREADY for input DATA channel
	s_axis_data_tdata	TDATA for input DATA channel. Replaces all DIN ports. See TDATA Structure for internal structure.
	s_axis_data_tuser	TUSER for input DATA channel. Optionally replaces CHAN_IN.
	s_axis_data_tlast	TLAST for input DATA channel. Optionally compared to internal channel counter (replacement for CHAN_IN) with discrepancies indicated on event_s_axis_*
	s_axis_reload_tvalid	TVALID for input RELOAD channel
	s_axis_reload_tready	TREADY for input RELOAD channel
	s_axis_reload_tdata	
	s_axis_reload_tlast	
	s_axis_config_tvalid	TVALID for input CONFIG channel
	s_axis_config_tready	TREADY for input CONFIG channel
	s_axis_config_tdata	
	s_axis_config_tlast	
	m_axis_data_tvalid	TVALID for output DATA channel
	m_axis_data_tready	TREADY for output DATA channel
	m_axis_data_tdata	TDATA for output DATA channel. Replaces all DOUT ports. See TDATA Structure for internal structure.
	m_axis_data_tuser	TUSER for output DATA channel. Optionally replaces CHAN_OUT
	m_axis_data_tlast	TLAST for output DATA channel. Optionally replaces function performed by CHAN_OUT.

Latency Changes

The latency of FIR Compiler v6.2 is different compared to v5.0 The update process cannot account for this and guarantee equivalent performance.

When in Blocking Mode (`m_data_tready` in use), the latency of the core is variable, so only the minimum possible latency can be determined. When in Non-Blocking Mode (no `m_data_tready`), the latency of the core might only be slightly greater than that for the equivalent configuration of v5.0. See the latency information in the GUI Summary page.

Instructions for Minimum Change Migration

To configure the FIR Compiler v6.2 to most closely mimic the behavior of v5.0 the translation is as follows:

Parameters

Output TREADY (Data Channel Options): Set to false. Disables back-pressure facility and guarantees fixed latency.

Input FIFO (Data Channel Options): Set to false. Disables the input FIFO on the S_AXIS_DATA channel and minimizes fabric resources.

Synchronization Mode (Configuration Channel Options): Set to "On Vector". This ensures the filter select values is updated on every processing cycle.

Configuration Method (Configuration Channel Options): Set to "By Channel" when applicable. This ensures a unique filter select value can be set for every interleaved data channel.

Reload Slots (Reload Channel Options): Set to the number of coefficient sets specified.

Data Vector Reset (Control Signals): Set to false. Minimizes fabric resources and matches FIR Compiler v5.0 reset behavior.

Ports

Input / Output Data Channels

ND is mapped to `s_data_tvalid`

RFD is mapped to `s_data_tready`

RDY is mapped to `m_data_tvalid`

Configuration Channel

`FILTER_SEL` is mapped to the filter select field of the `s_config_tdata` bus

Drive `s_config_tvalid` with the same signal driving `s_data_tvalid`

Tie `s_config_tlast` to 0 and ignore `event_s_config_*`

Reload Channel

The format of the reload channel has changed such that `COEF_FILTER_SEL` is now pre-pended to the reload packet on the `s_reload_tdata` bus.

`COEF_DIN` is mapped to `s_reload_tdata` bus

`COEF_WE` is mapped to `s_reload_tvalid`

`COEF_LD` is mapped to `s_reload_tlast` but is now asserted at the end of a reload packet

Core Features

Filter Architectures

Multiply-Accumulate

Figure 11 illustrates a simplified view of a MAC-based FIR utilizing a single MAC engine.

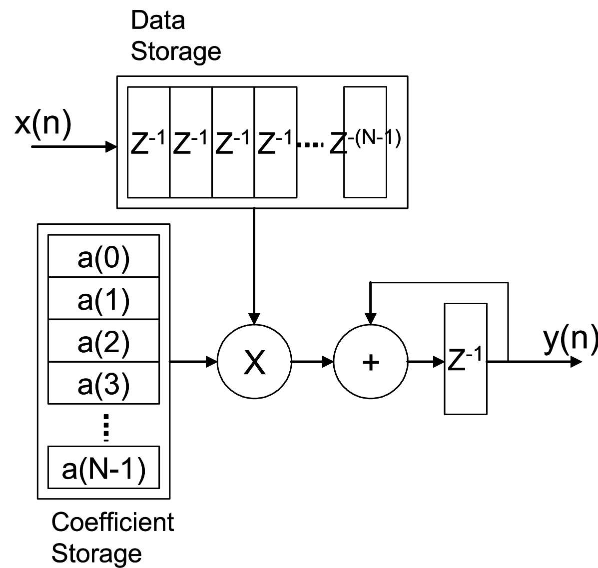


Figure 11: **Single MAC Engine Block Diagram**

The single implementation is extensible to multi-MAC implementations for use in achieving higher performance filter specifications (larger numbers of coefficients, higher sample rates, more channels, etc.).

The number of multipliers required to implement a filter is determined by calculating the number of multiplies required to perform the computation (taking into account symmetrical and half-band coefficient structures and sample rate changes) and then dividing by the number of clocks available to process each input sample. The available clock cycles value is always rounded down and the number of multipliers rounded up to the nearest integer. If there is a non-zero remainder, some of the MAC engines calculate fewer coefficients than others, and the coefficients are padded with zeros to accommodate the excess cycles.

The output samples reflect the padding of the coefficient vector; for this reason, the response to an applied impulse contains a certain number of zero outputs before the first coefficient of the specified impulse response appears at the output. The core automatically generates an implementation that meets the user-defined performance requirements based on the system clock rate, the sample rate, the number of taps and channels, and the rate change. The core inserts one or more multipliers to meet the overall throughput requirements.

The current version of the FIR Compiler only supports the Systolic Multiply Accumulate filter architecture.

Systolic Multiply-Accumulate

Figure 12 illustrates the Systolic Multiply-Accumulate architecture implementing a pipelined Direct-Form filter.

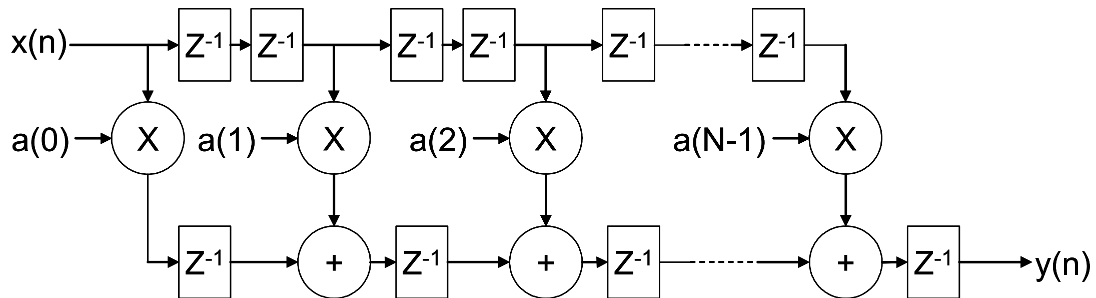


Figure 12: **Pipelined Direct-Form**

Figure 13 illustrates a multi-MAC implementation for this architecture.

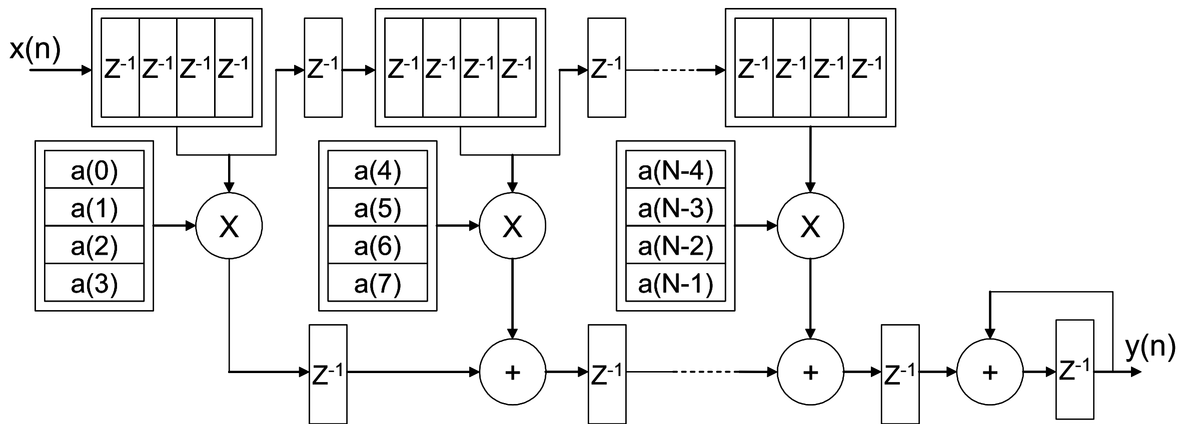


Figure 13: **Systolic Multi-MAC Implementation**

The architecture is directly supported by the XtremeDSP Slice and results in area-efficient and high performance filter implementations. The structure also extends to exploit coefficient symmetry offering further resource savings.

Transpose Multiply-Accumulate

Figure 14 illustrates the Transpose Multiply-Accumulate architecture implementing a Transposed Direct-Form filter.

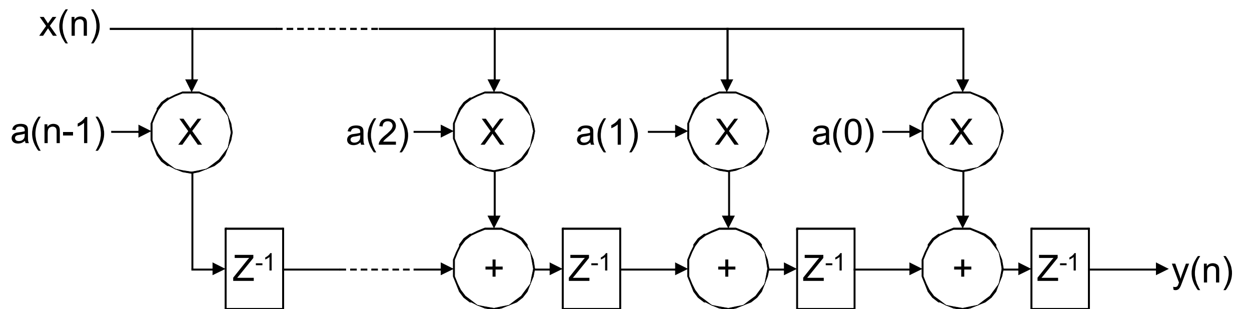


Figure 14: **Transpose Direct-Form**

Figure 15 illustrates a multi-MAC implementation for this architecture.

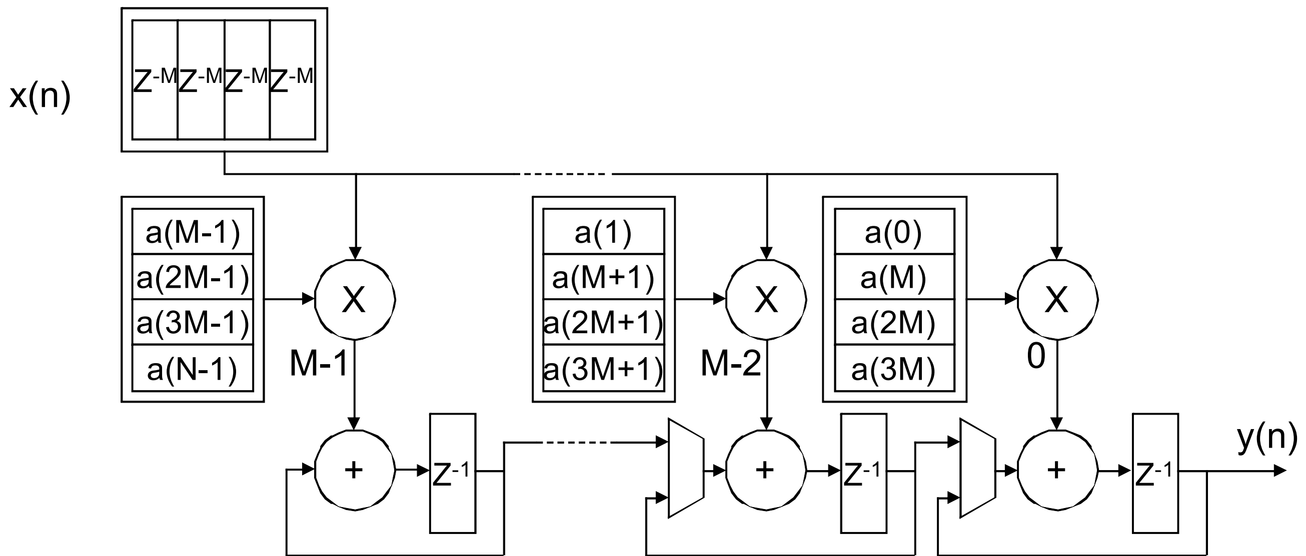


Figure 15: **Transpose Multi-MAC Implementation**

This architecture is also directly supported by the XtremeDSP Slice. This structure offers a low latency implementation, and for some configurations can also offer extra resource savings over the Systolic structure. It does not require an accumulator and can use fewer data memory resources, although it does not exploit coefficient symmetry.

Filter Structures and Optimizations

Filter Symmetry

The impulse response for many filters possesses significant symmetry. This symmetry can generally be exploited to minimize arithmetic requirements and produce area-efficient filter realizations. Figure 16 shows the impulse response for a 9-tap symmetric FIR filter.

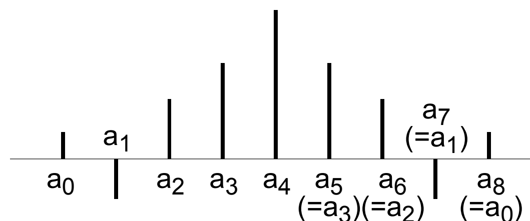


Figure 16: **Symmetric FIR – Odd Number of Terms**

Instead of implementing this filter using the architecture shown in Figure 1, the more efficient signal flow-graph in Figure 17 can be used. In general, the former approach requires N multiplications and (N-1) additions. In contrast,

the architecture in Figure 17 requires only [N/2] multiplications and approximately *N* additions. This significant reduction in the computation workload can be exploited to generate efficient filter hardware implementations.

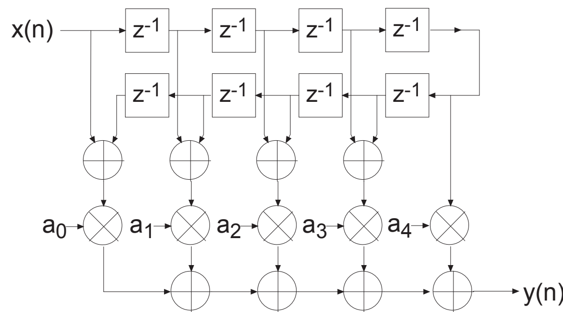


Figure 17: **Exploiting Coefficient Symmetry – Odd Number of Filter Taps**

Coefficient symmetry for an even number of terms can be exploited as shown in Figure 18.

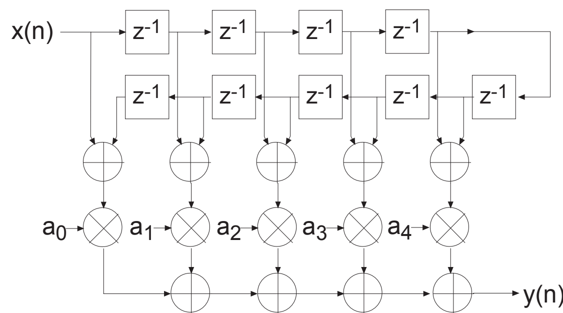


Figure 18: **Exploiting Coefficient Symmetry – Even Number of Filter Taps**

Figure 19 shows the impulse response for a negative, or odd, symmetric filter.

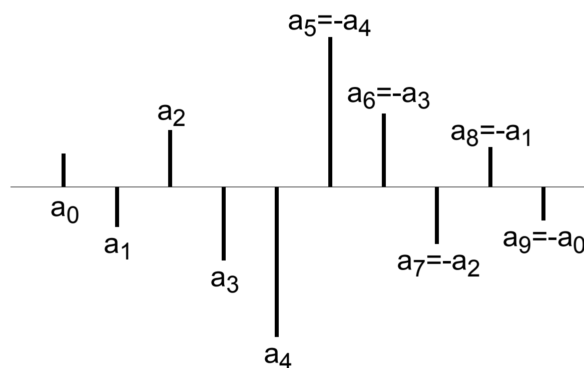


Figure 19: **Negative Symmetric Impulse Response**

This symmetry is exploited in a manner similar to that shown in Figure 17 and Figure 18. In this case, the middle layer of adders are replaced by subtracters, as illustrated in Figure 20.

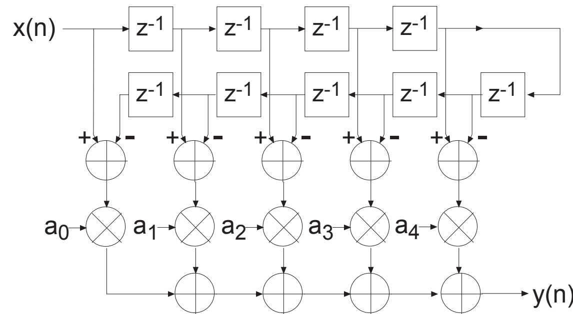
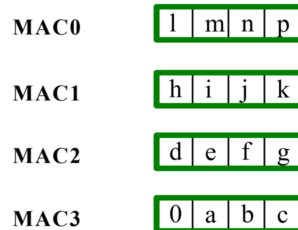


Figure 20: **FIR Architecture Exploiting Negative Symmetry**

Filter coefficient symmetry is inferred by the core GUI from the coefficient definition file. This inferred value can be overridden by the user. When the structure is inferred, the inferred setting is displayed in the Summary page and in the ToolTip for the Coefficient Structure field.

Coefficient Padding

When implementing a filter with symmetric coefficients using the Multiply-Accumulate architecture, users must be aware that the core reorganizes the filter coefficients if required to exploit symmetry, and this **might alter the filter response**. This is only necessary if the core is configured such that all processing cycles are not utilized. For example, when the core has four cycles to process each sample for a 30-tap symmetric response filter, the core pads the coefficient storage out as illustrated in Figure 21.



Resultant Impulse Response

Figure 21: **Filter Padding to Facilitate Symmetric Structure Exploitation**

The appended zeroes after the non-zero coefficients do not affect the filter response, but the prepended zero coefficients do alter the phase response of the filter implementation when compared to the ideal coefficients. There are two ways to avoid this issue: First, and simplest, the user can force the Coefficient Structure to be Non-Symmetric. This avoids the issue of prepending zero coefficients to the coefficient vector, and only appended zeroes are used to pad out the filter response to the required number of cycles. Second, and more efficient, the user can increase the number of taps implemented by the filter **at little or no cost in resource usage.** In the previous example, the filter could process 32 taps in the same time, with the same hardware resources, and with the same cycle latency as the 30-tap implementation, and the phase response of the 32-tap filter would be unaltered.

The core GUI displays the actual number of coefficients calculated on the Implementation Details tab. Users can use this information to determine if they can increase the number of coefficients used by their filter definition.

Single-rate FIR Filter

The basic FIR filter core is a single-rate (input sample rate = output sample rate) finite impulse response filter. This is the simplest of filter types and is the default at the start of parametrization in the CORE Generator software.

Half-band FIR Filter

Figure 22 illustrates the general frequency response for a half-band filter.

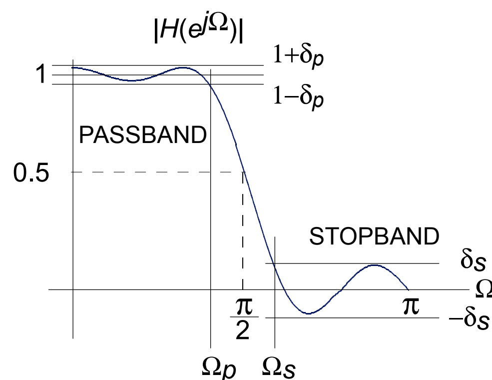


Figure 22: **Half-band Filter Magnitude Frequency Response**

The magnitude frequency response is symmetrical about quarter sample frequency π/2 radians. The sample rate is normalized to 2π radians/sec. The passband and stopband frequencies are positioned such that

$$\Omega_p = \pi - \Omega_s$$

The passband and stopband ripple, δ_p and δ_s respectively, are equal $\delta_p = \delta_s$. These properties are reflected in the filter impulse response. It can be shown [Ref 5] that approximately half of the filter coefficients are zero for an odd number of taps, as illustrated in Figure 23 for an 11-tap half-band filter.

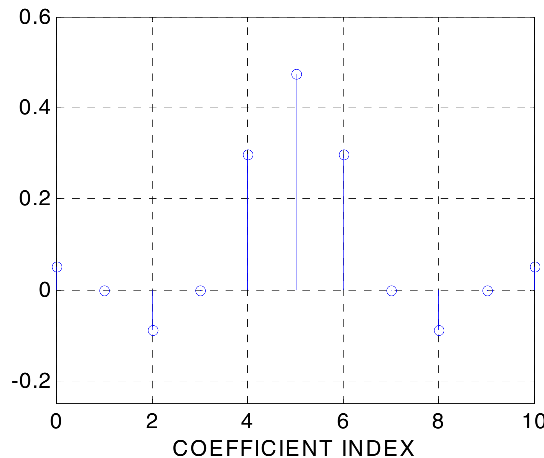


Figure 23: **Half-band Filter Impulse Response**

The interleaved zero values in the coefficient data can be exploited to realize an efficient realization, as shown in Figure 24.

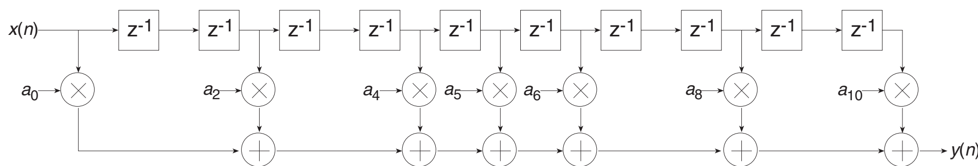


Figure 24: **Half-band Filter Impulse Response**

The half-band filter selection in the compiler is intended for this purpose. This filter is available in the *Coefficient Structure* field of the user interface. The user must supply the complete list of filter coefficients, including the 0 value samples, when using the half-band filter. The filter coefficient file format is discussed in greater detail in the Filter Coefficient Data section.

Polyphase Decimator

Figure 25 illustrates the polyphase decimation filter option which implements the computationally efficient *M*-to-1 polyphase decimating filter.

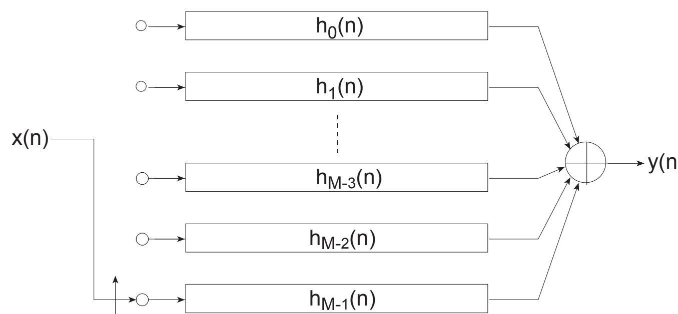


Figure 25: ***M*-to-1 Polyphase Decimating Filter**

A set of N prototype filter coefficients $a_0, a_1, .., a_{N-1}$ is mapped to the M polyphase subfilters $h_0(n), h_1(n), .., h_{M-1}(n)$ according to Equation 2.

$$h_i(r) = a(i + Mr) \quad i = 0, 1, .., M-1 \quad r = 0, 1, .., \frac{N}{M} \qquad \text{Equation 2}$$

The polyphase segments are accessed by delivering the input samples $x(n)$ to their inputs via an input commutator which starts at the segment index $i = M-1$ and decrements to index 0. After the commutator has executed one cycle and delivered M input samples to the filter, a single output is taken as the summation of the outputs from the polyphase segments. The output sample f_s' rate is $f_s' = \frac{f_s}{M}$ where f_s is the sample rate of the input data stream $x(n), n = 0, 1, 2, ..$

Observe that each of the polyphase segments is operating at the low output sample rate f_s' (compared to the high input sample rate f_s), and a total of N operations is performed per output point.

Polyphase Interpolator

Figure 26 illustrates the polyphase interpolation filter option which implements the computationally efficient 1-to-P interpolation filter.

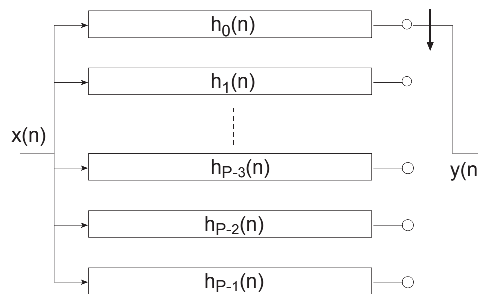


Figure 26: **1-to-*P* Polyphase Interpolator**

A set of N prototype filter coefficients $a_0, a_1, .., a_{N-1}$ is mapped to the P polyphase subfilters $h_0(n), h_1(n), .., h_{p-1}(n)$ according to Equation 2, as in the decimation case.

Each new input sample $x(n)$ engages all of the polyphase segments in parallel. For each input sample delivered to the filter, P output samples, one from each segment, are delivered to the filter output port, as indicated by the commutator in Figure 26.

The output sample f_s' rate is $f_s' = f_s P$ where f_s is the sample rate of the input data stream $x(n), n = 0, 1, 2, ..$. Observe each of the polyphase segments operating at the low input sample rate f_s (compared to the high output sample rate f_s) and a total of N operations performed per output point.

Polyphase Interpolator Exploiting Symmetric Pairs

The *symmetric pairs* technique [Ref 8] is used to exploit coefficient symmetry when implementing an Interpolation filter in the Systolic Multiply-Accumulator architecture. When P polyphase subfilters are generated from symmetric filter coefficients, not all the subfilters contain a set of coefficients that are themselves symmetric. The symmetric pairs technique observes that adding and subtracting two corresponding non-symmetric phases produces two new phases containing symmetric coefficients.

The following example demonstrates this technique for a 15-tap interpolate by 3 filter. The filter coefficients:

a, b, c, d, e, f, g, h, g, f, e, d, c, b, a

Produce the following subfilters:

h_0 = a, d, g, f, c

h_1 = b, e, h, e, b

h_2 = c, f, g, d, a

Subfilters h_0 and h_2 are not symmetric. Applying the symmetric pairs technique produces the following subfilters:

h_0 = a+c, d+f, d,g, f+d, c+a

h_1 = b, e, h, e, b

h_2 = c-a, f-d, g-g, d-f, a-c

Now both h_0 and h_2 are symmetric with h_2 being negative symmetric. The filter can now be implemented utilizing symmetry, giving the associated resource savings. The output from subfilters h_0 and h_2 must be added and subtracted and then scaled by a factor of 0.5 to produce the original filter output. Figure 27 illustrates the resulting structure.

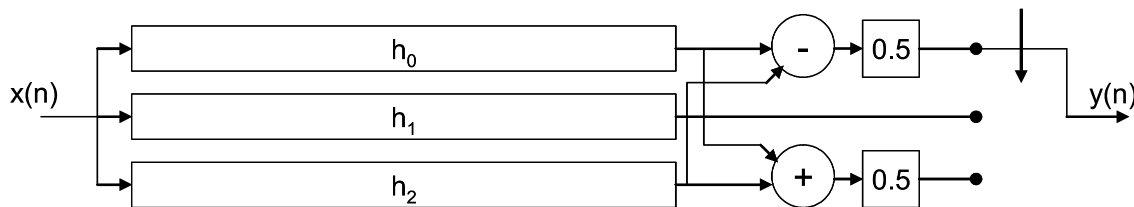


Figure 27: **Symmetric Pairs**

Note: When interpolating by 2 with an odd number of symmetric coefficients, this technique is not required as the resulting polyphase subfilters are symmetric.

Coefficient Padding

As with the general symmetric filter case, if the combination of rate and number of filter taps results in a subfilter which is not fully populated with coefficients, the reorganization of the filter coefficients results in a change in the phase response of the filter. The impulse response is shifted by a number of output samples as a result. In the 14 tap, interpolate by 4 case, padding a zero coefficient to the front of the coefficient response would be required to align the phases such that symmetry can be exploited, resulting in a smaller implementation, but this results in a different phase response for the filter. The methods to avoid this change in response, if such a change cannot be accommodated in the user's application system, are also similar to the general symmetry case; the user can either force non-symmetric structure implementation or make use of the extra coefficients which can be supported in the structure. Figure 28 illustrates several example cases in and is extensible to larger filters.

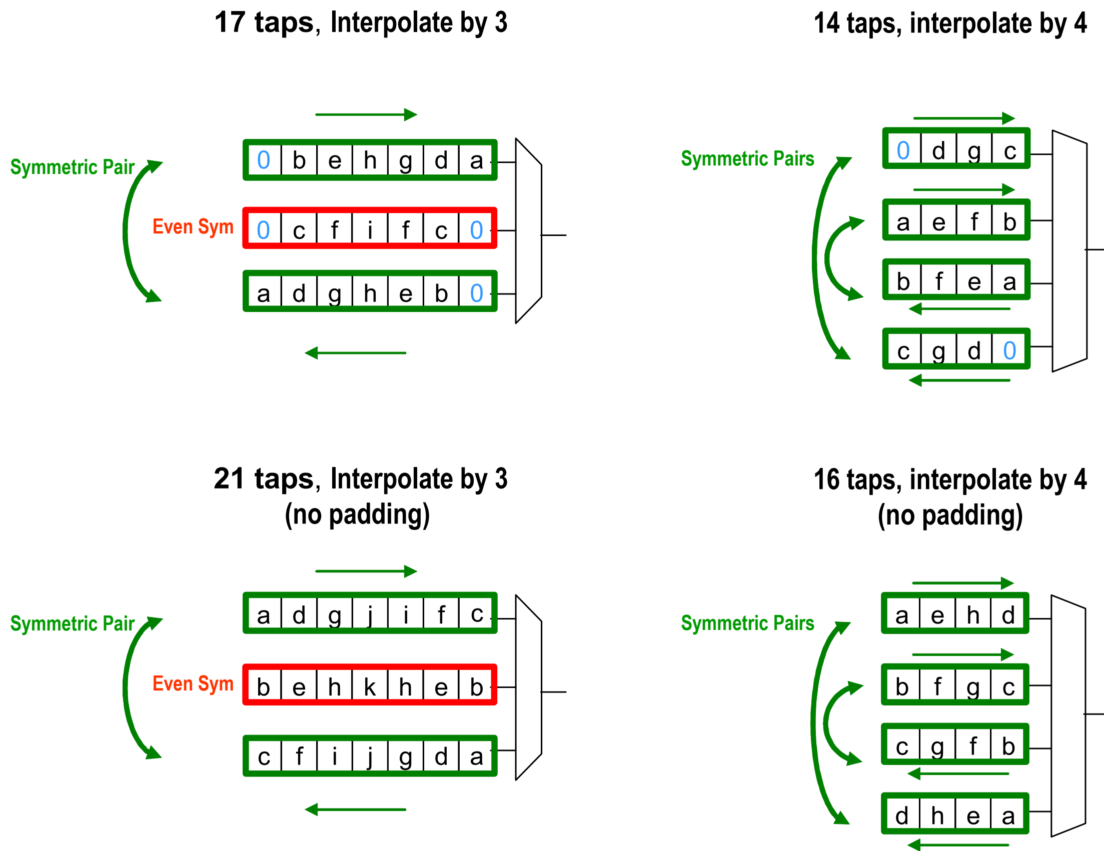


Figure 28: **Filter Padding to Facilitate Symmetric Pairing**

Half-band Decimator

The half-band decimator is a polyphase filter with an embedded 2-to-1 down-sampling of the input signal. Figure 29 illustrates the structure.

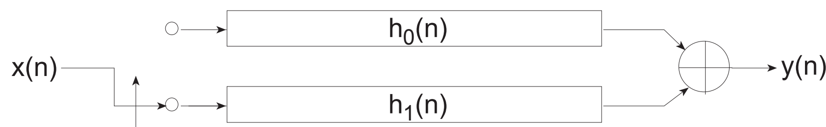


Figure 29: **Half-band Decimation Filter**

The filter is very similar to the polyphase decimator described in Polyphase Decimator with the decimation factor set to *M*=2. However, there is a subtle difference in the implementation that makes the half-band decimator a more area-efficient 2-to-1 down-sampling filter when the frequency response reflects a true half-band characteristic.

The frequency and time response of a half-band filter are shown in Figure 22 and Figure 23, respectively. Observe the alternating zero-valued coefficients in the impulse response. Figure 29 details a 7-tap half-band polyphase filter when the coefficients are allocated to the two polyphase segments $h_0(n)$ and $h_1(n)$ shown in Figure 29. Figure 30 (a) is the filter impulse response; note that $a_1 = 0 = a_5$. Figure 30 (b) provides a detailed illustration of the polyphase subfilters and shows how the filter coefficients are allocated to the two polyphase arms.

In the bottom arm, $h_1(n)$, the only non-zero coefficient, is the center value of the impulse response a_3. Figure 30 (c) shows the optimized architecture when the redundant multipliers and adders are removed. The final structure has a reduced computation workload in contrast to a more general 2:1 down-sampling filter.

The number of multiply-accumulate (MAC) operations required to compute an output sample has been lowered by a factor of approximately two. In this figure, the high density of zero-valued filter coefficients is exploited in the FPGA realization to produce a minimal area implementation.

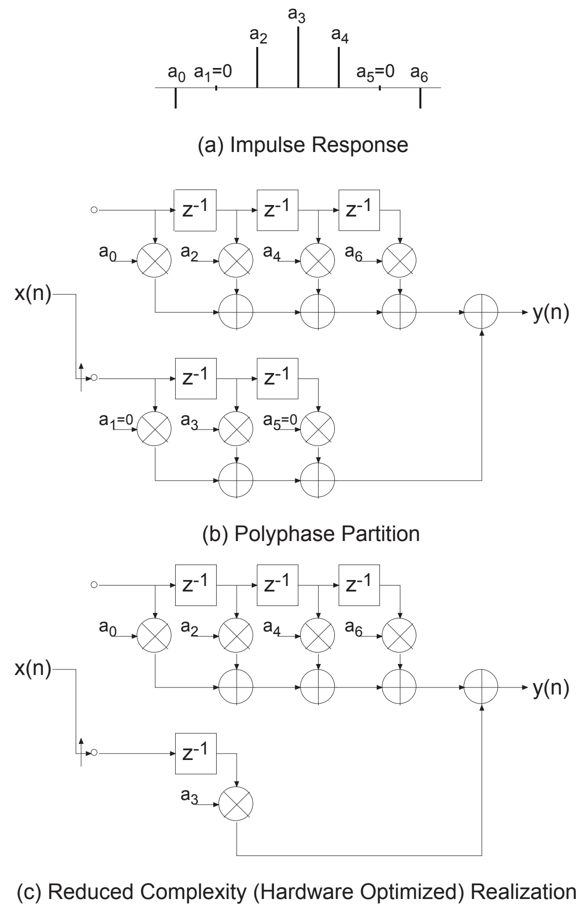


Figure 30: **7-Tap Half-band Decimation Filter**

Half-band Interpolator

Just as the half-band decimator is an optimized version of the more general polyphase decimation filter, the half-band interpolator is a special case of a polyphase interpolator. Figure 31 displays the half-band interpolator.

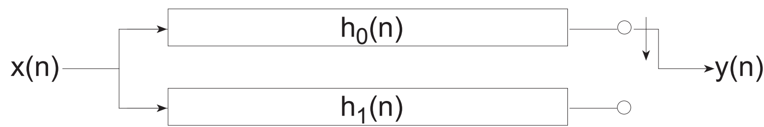


Figure 31: **Half-band Interpolation Filter**

The coefficient set for a true half-band interpolator is identical to that of a half-band decimator with the same specifications. The large number of zero entries in the impulse response is exploited in exactly the same manner as with the half-band decimator to produce hardware-optimized half-band interpolators. The process is presented in Figure 32. Figure 32(a) is the impulse response, Figure 32(b) shows the polyphase partition, and Figure 32(c) is the optimized architecture that has taken full advantage of the 0 entries in the coefficient data.

The high density of zero-valued filter coefficients is exploited in the FPGA realization to produce a minimal area implementation.

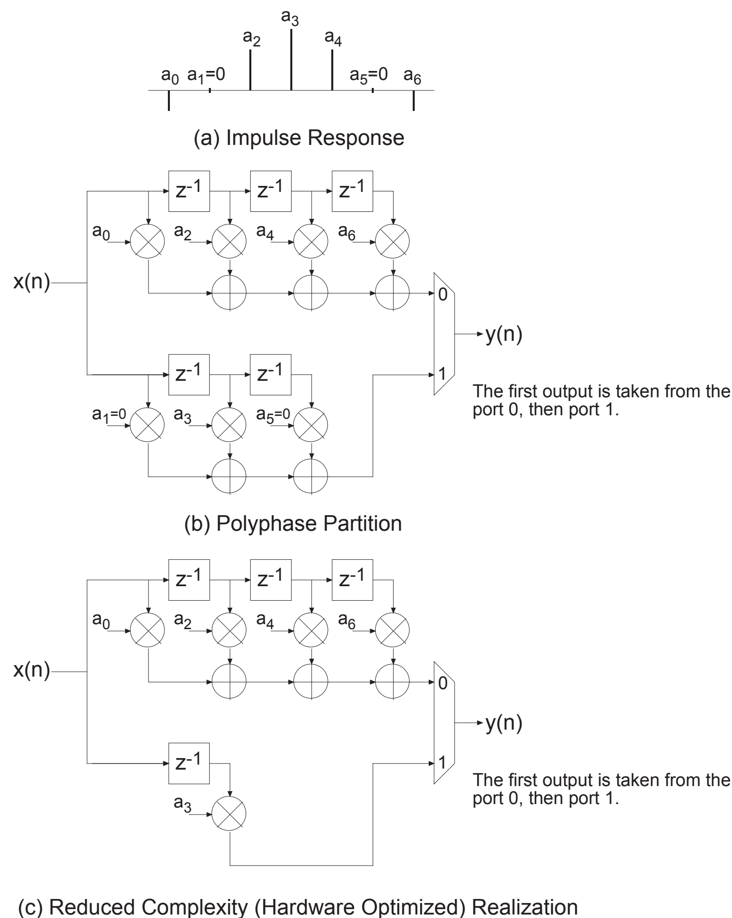


Figure 32: **7-Tap Half-band Interpolation Filter**

Small Non-zero Even Terms in a Half-band Filter Impulse Response

Certain filter design software can result in small non-zero values for the odd terms in the half-band filter impulse response. In this situation, it can be useful to force these values to 0 and re-evaluate the frequency response to assess if it is still acceptable for the intended application. If the odd terms are not identically zero, the hardware optimizations described previously are not possible. If the small non-zero value terms cannot be ignored, the general polyphase decimator or interpolator described in Polyphase Decimator and Polyphase Interpolator, using a rate change of two, is more appropriate.

Fixed Fractional Rate Resampling Filters

FIR filters that implement resampling of a data stream at a fixed fractional rate P/Q, where P and Q are integers up to 64, are available for the Systolic Multiply-Accumulate architecture. In Figure 33, the operation of an interpolation filter with interpolation rate P=5 is contrasted conceptually with the operation of a fixed fractional rate filter with rate P/Q=5/3.

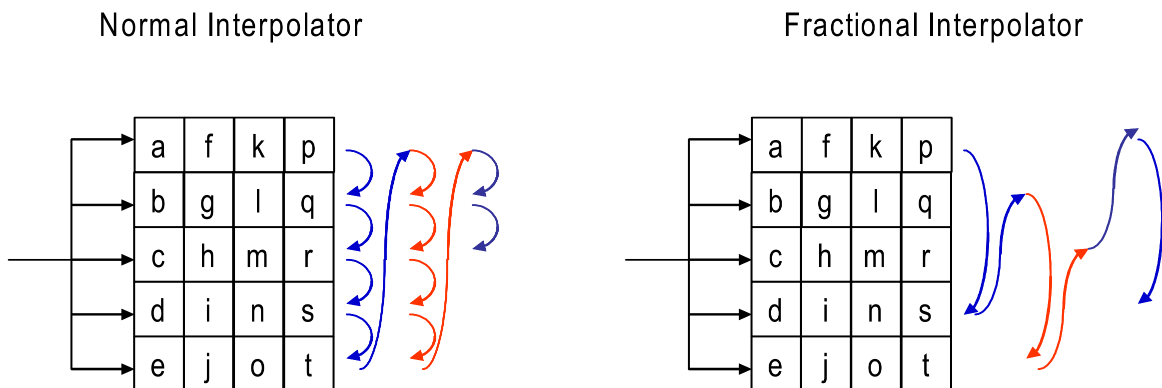


Figure 33: **Interpolation Filters for Integer and Fractional Rates**

The normal (integer rate) interpolator passes the input sample to all P phases and then produces an output from each of the phase arms of the polyphase filter structure. In the fractional rate version, the output is taken from a phase arm which varies according to a stepping sequence with step size Q.

Figure 34 illustrates a similar conceptual method for implementing fractional rate decimators. The integer decimation rate for the left-hand diagram is Q=5, while the fractional-rate illustrated on the right is P/Q=3/5.

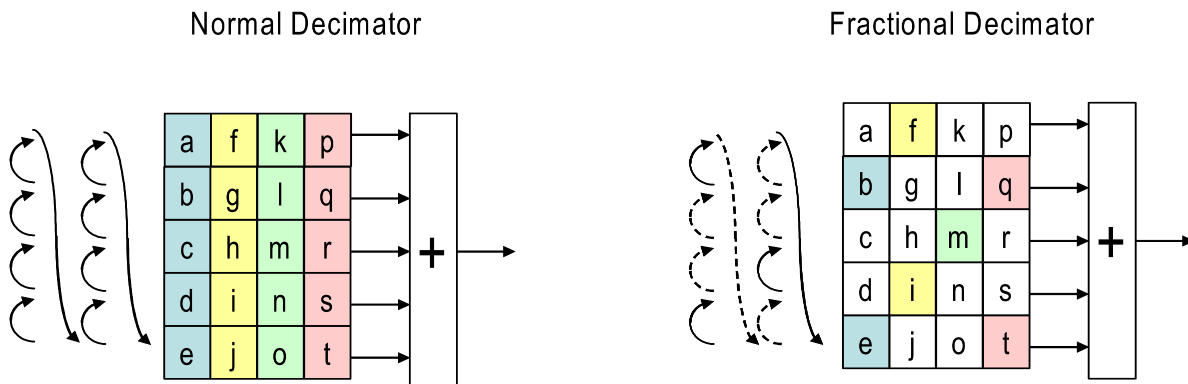


Figure 34: **Decimation Filters for Integer and Fractional Rates**

The integer rate decimator passes the input samples in sequence to each of the Q phase arms in turn, with the data being shifted through the filter, and the output is generated from the summation of the outputs from each phase arm of the polyphase filter. For the fractional rate implementation, the filter passes the input samples to phases in a stepping sequence based on a step size of P, with zero samples being placed into the skipped phases. The summation across the various phase arms remains the same, but is based on fewer actual calculations. The implementation details differ somewhat from these conceptual illustrations, but the resulting behavior of the filter is the same. Symmetry is not currently exploited when using the fractional rate structures.

Filter Coefficient Data

The filter coefficients are supplied to the FIR Compiler using a coefficient file with a .coe extension. This is an ASCII text file with a single-line header that defines the radix of the number representation used for the coefficient data, followed by the coefficient values themselves. This is shown in Figure 35 for an *N*-tap filter.

```
radix=coefficient_radix;
coefdata=
a(0),
a(1),
a(2),
….
a(N-1);
```

Figure 35: **Filter Coefficient File Format**

The filter coefficients can be supplied as integers in either base-10, base-16, or base-2 representation. This corresponds to *coefficient_radix*=10, *coefficient_radix*=16, and *coefficient_radix*=2 respectively. Alternatively, the coefficients can be entered as real numbers (specified to a minimum of one decimal place) in base-10 only. If the user enters signed negative symmetric hexadecimal coefficients, each value should be sign-extended to the boundary of the most significant nibble or hex character. This ensures that coefficient structure inference can be performed correctly (this includes Hilbert transform filter types, which are also negative symmetric).

The coefficient values can also be placed on a single line as shown in Figure 36.

```
radix=coefficient_radix;
coefdata=a(0),a(1),a(2),...,a(N-1);
```

Figure 36: **Filter Coefficient File Format – Coefficient Data on a Single Line**

Single-rate FIR

The coefficient file for the single-rate FIR filter is straightforward and consists of a one-line header followed by the filter coefficient data. For example, the filter coefficient file for an 8-tap filter using a base-10 representation for the coefficient values is shown in Figure 37:

```
radix=10;
coefdata=20,-256,200,255,255,200,-256,20;
```

Figure 37: **Filter Coefficient File – 8-Tap Filter, Base-10 Coefficient Values**

Irrespective of the filter possessing positive or negative symmetry, the coefficient file should contain the complete set of coefficient values. The filter coefficient file for the non-symmetric impulse response shown in Figure 38 is presented in Figure 39.

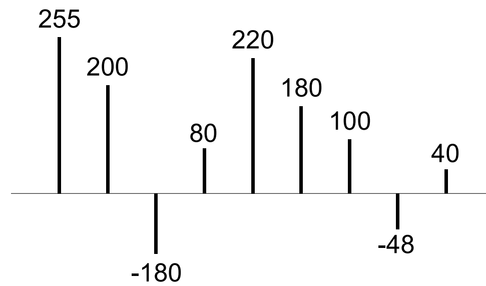


Figure 38: **Non-symmetric Impulse Response**

```
radix=10;
coefdata=255,200,-180,80,220,180,100,-48,40;
```

Figure 39: **Coefficient File for the Non-symmetric Impulse Response**

The coefficient file for the negative-symmetric filter characterized by the impulse response in Figure 40 is shown in Figure 41.

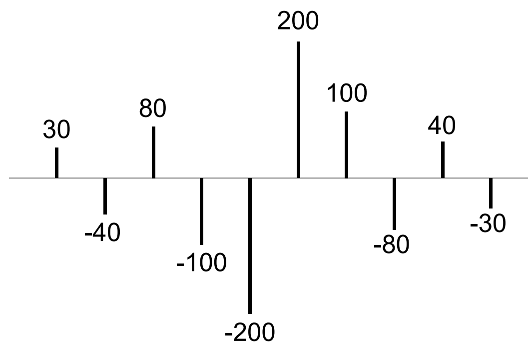


Figure 40: **Negative Symmetric Impulse Response**

```
radix=10;
coefdata=30,-40,80,-100,-200,200,100,-80,40,-30;
```

Figure 41: **Coefficient File for the Negative Symmetric Impulse Response**

Half-band Filter

As previously described, every second filter coefficient for a half-band filter with an odd number of terms is zero. When specifying the filter coefficient data for this filter class, the zero value entries must be included in the coefficient file. For example, the filter coefficient file that specifies the filter impulse response in Figure 42 is shown in Figure 43.

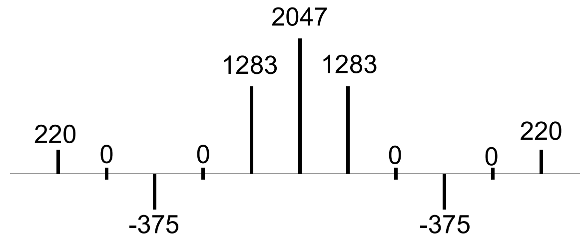


Figure 42: **11-Tap Half-band Filter Impulse Response**

```
radix=10;
coefdata=220,0,-375,0,1283,2047,1283,0,-375,0,220;
```

Figure 43: **Coefficient File for the Half-band Filter Impulse Response**

The filter coefficient set is parsed by the FIR Compiler. If either the alternating zero entries are absent or the coefficient set is not even-symmetric, this condition is flagged as an error and the filter is not generated. A dialog box is presented to indicate the issue under these circumstances.

Technically, the zero-valued entries for a half-band filter can occur at the filter impulse response extremities as shown in Figure 44. However, observe that these values do not contribute to the result.

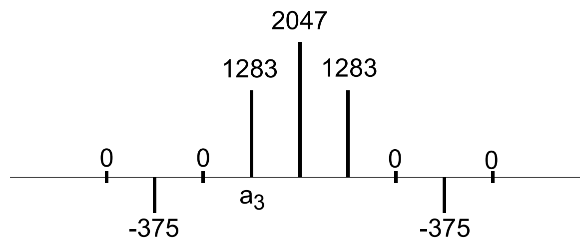


Figure 44: **9-Tap Half-band Filter Impulse Response**

This condition is detected when the filter is specified. If the number of taps is such that the zero-valued coefficients form the first and last entry of the impulse response, the filter length is reported as an invalid value. The number of taps N for a half-band filter must obey N=3 + 4n, where n=0,1,2,3,…. For example, a half-band filter can have 11, 15, 19, and 23 taps, but not 9, 13, 17, or 21 taps.

Multiple Coefficient Sets

For multiple coefficient filters, a single .coe file is used to specify the coefficient sets. Each coefficient set should be appended to the previous set of coefficients.

For example, if a 2-coefficient set, 10-tap symmetric filter was being designed and coefficient set #0 was: `coef data = -1, -2, -3, 4, 5, 5, 4, -3, -2, -1;`

and coefficient set #1 was:

coefdata = -9, -10, -11, 12, 13, 13, 12, -11, -10, -9;

then the .coe file for the entire filter would be:

radix = 10;

coefdata = -1, -2, -3, 4, 5, 5, 4, -3, -2, -1, -9, -10, -11, 12, 13, 13, 12, -11, -10, -9;

All coefficients sets in a multiple set implementation must exhibit the same symmetry. For example, if even one set of a multi-set has non-symmetric coefficient structure, then all sets are implemented using that structure. All coefficient sets must also be of the same vector length. If one coefficient set has fewer coefficients, it must be zero padded – either appended with zeros when non-symmetric or prepended and appended with an equal number of zeros when symmetric. See the Coefficient Padding section for further information.

Coefficient Specification Using Non-integer Real Numbers

As indicated previously, the user can specify the coefficient values as non-integer real numbers, with the radix set to 10. For example:

radix = 10;

coefdata = 0.08659436542927, 0.00579513928555, -0.06734424313287, -0.04031582111240;

The coefficients are then quantized by the core to produce the binary coefficient values used in the filter, based on the user's specified coefficient bit width. This allows the user to supply floating-point values derived from a chosen filter design tool and explore the costs and benefits between performance and resource usage by altering the coefficient bit width and observing the alteration in the quantified frequency response in comparison to the ideal response. The basic quantization function is selected by setting the Quantization field to Quantize_Only. See the Coefficient Quantization section for further details.

The integer values used in the filter implementation can be determined by examining the main core MIF file (<component_name>.mif) which is generated in the CORE Generator software project directory. The MIF file is always in binary format.

Interleaved Data Channel Filters

The FIR Compiler core provides support for processing multiple input sample streams using the same implementation. Each input stream is filtered using the same filter configuration (rate change, etc.) using the currently selected filter coefficient set.

In many applications, the same filter must be applied to several data streams. A common example is the simple digital down converter shown in Figure 45. Here a complex base-band signal $x(n) = x_I(n) + jx_Q(n)$ is applied to a matched filter M(z). The in-phase and quadrature components are processed by the same filter.

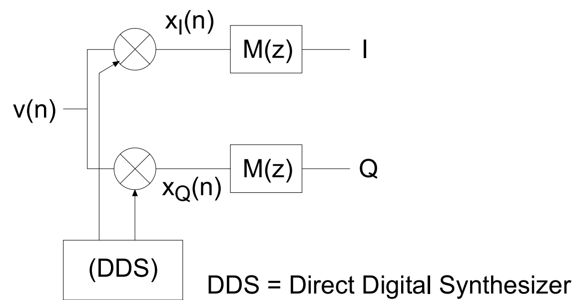


Figure 45: **Digital Down Converter**

One solution to this issue is to employ two separate filters; however, this can waste logic resources. A more efficient design can be realized using a filter architecture that shares logic resources between multiple time division multiplexed (TDM) sample streams. As more channels are processed by the core, the sample throughput is commensurately reduced. For example, if the sample rate for a single-channel filter is f_S, a two-channel version of the same filter processes two sample streams, each with a sample rate of $f_S/2$. A three-channel version of the filter processes three data streams and supports a sample rate of $f_S/3$ for each of the streams.

A multichannel filter implementation is very efficient in resource utilization. A filter with two or more channels can be realized using a similar amount of logic resources to a single-channel version of the same filter, with proportionate increase in data memory requirements. The trade-off that needs to be addressed when using multichannel filters is one of sample rate versus logic requirements. As the number of channels is increased, the logic area remains approximately constant, but the sample rate for an individual input stream decreases. The number of channels supported by a filter core is specified in the filter customization GUI.

Parallel Data Channel Filters

The FIR Compiler provides support for processing multiple parallel datapaths with the same filter coefficients. This feature differs from a multiple-channel implementation when it is necessary to time division multiplex (TDM) the individual channels onto a single data stream. When processing parallel datapaths, the FIR Compiler allocates a field of the `s_axis_data_tdata` and `m_axis_data_tdata` port to each individual datapath. See Input and Output DATA Channels, page 19 for details of the TDATA format.

This feature can be used in conjunction with the Interleaved Data Channel Filters, page 47 feature such that multiple data stream can be shared across multiple parallel paths and interleaved channels. For example, six data streams can be shared across two parallel datapaths each implementing three interleaved data channels. Each parallel datapath exhibits the same interleaved data sequence and the Channel ID field of the `s_data_tuser` and `m_data_tuser` buses is shared across all paths.

In this configuration, the FIR Compiler can share control logic and coefficient memory resources between the parallel datapaths. This offers significant resource savings over using one FIR Compiler instance per parallel datapath.

Coefficient Reload

To minimize the resources required to implement the coefficient reload feature, it is necessary for users to re-order the coefficients that are to be reloaded to correctly pass each coefficient to its correct storage location in the filter structure. When "Reloadable Coefficients" have been selected, the CORE Generator software delivers an informational text file to the project area named <component_name>_reload_order.txt, which lists the indices of the coefficients, "Coefficient *x*," in the order they should be reloaded into the filter via the reload port "Reload index *x*."

The core now also offers the facility to generate a re-ordered coefficient file, see Tab 4: Coefficient Reload, page 8.

Care must be take to correctly interpret the reload order, as it is based on the actual number of coefficients calculated by the filter. The Coefficient Padding section of Filter Symmetry discusses how the FIR Compiler sometimes implements a filter with more coefficients than specified. The actual coefficients calculated are displayed on the Implementation Details tab. When the filter is configured to utilize coefficient symmetry, the user must pad the filter response at the beginning and the end with (actual - specified)/2 zeros before applying the reload order. Figure 21 demonstrates a padded filter response. When the filter is non-symmetric, the coefficient set must be padded with (actual - specified) zeros at the end of the filter response before applying the reload order.

In the case of a polyphase interpolating filter utilizing coefficient symmetry, where the Symmetric Pairs technique has been used, the coefficients must be preprocessed before being loaded into the filter. The combination of the non-symmetric subfilters are defined as the sum or difference of two coefficient indices. When the filter configuration requires multiple DSP slices to implement a single Multiply-Accumulate unit, the definition is extended to include bit ranges of the source coefficients.

Figure 46 contains an example of the _reload_order.txt file, for a non-symmetric 16-tap single rate filter where the clock rate is four times the input sample rate.

```
Reload index 0 = Coefficient 12
Reload index 1 = Coefficient 13
Reload index 2 = Coefficient 14
Reload index 3 = Coefficient 15
Reload index 4 = Coefficient 8
Reload index 5 = Coefficient 9
Reload index 6 = Coefficient 10
Reload index 7 = Coefficient 11
Reload index 8 = Coefficient 4
Reload index 9 = Coefficient 5
Reload index 10 = Coefficient 6
Reload index 11 = Coefficient 7
Reload index 12 = Coefficient 0
Reload index 13 = Coefficient 1
Reload index 14 = Coefficient 2
Reload index 15 = Coefficient 3
```

Figure 46: **Reload Order Text File Format Example 1**

Figure 47 contains an example for a symmetric 15-tap interpolate by 3 filter where the clock rate is six times the input sample rate and a coefficient width of 16 bits.

```
Reload index 0 = Coefficient 7
Reload index 1 = Coefficient 10
Reload index 2 = Coefficient 6 - Coefficient 8
Reload index 3 = Coefficient 9- Coefficient 11
Reload index 4 = Coefficient 6 + Coefficient 8
Reload index 5 = Coefficient 9 + Coefficient 11
Reload index 6 = Coefficient 1
Reload index 7 = Coefficient 4
Reload index 8 = Coefficient 0 - Coefficient 2
Reload index 9 = Coefficient 3 - Coefficient 5
Reload index 10 = Coefficient 0 + Coefficient 2
Reload index 11 = Coefficient 3 + Coefficient 5
```

Figure 47: **Reload Order Text File Format Example 2**

Figure 48 contains an example with the same filter configuration as in Figure 47, but with a coefficient width of 30 bits (the width of the reload port is extended when the Symmetric Pairs technique is used, so in this example, the reload port is 33 bits wide).

Contact Xilinx Technical Support if you need any assistance or guidance in implementing the reload coefficient ordering for your specific filter implementation.

```
Reload index 0 (17 downto 0) = "00" & Coefficient 7 (15 downto 0)
Reload index 0 (32 downto 18) = Coefficient 7 (29) & Coefficient 7 (29 downto 16)
Reload index 1 (17 downto 0) = "00" & Coefficient 10 (15 downto 0)
Reload index 1 (32 downto 18) = Coefficient 10 (29) & Coefficient 10 (29 downto 16)
Reload index 2 (17 downto 0) = "00" & Coefficient 6 (15 downto 0) -
                "00" & Coefficient 8 (15 downto 0)
Reload index 2 (32 downto 18) = Coefficient 6 (29) & Coefficient 6 (29 downto 16) -
                Coefficient 8 (29) & Coefficient 8 (29 downto 16)
Reload index 3 (17 downto 0) = "00" & Coefficient 9 (15 downto 0) -
                "00" & Coefficient 11 (15 downto 0)
Reload index 3 (32 downto 18) = Coefficient 9 (29) & Coefficient 9 (29 downto 16) -
                Coefficient 11 (29) & Coefficient 11 (29 downto 16)
Reload index 4 (17 downto 0) = "00" & Coefficient 6 (15 downto 0) +
                "00" & Coefficient 8 (15 downto 0)
Reload index 4 (32 downto 18) = Coefficient 6 (29) & Coefficient 6 (29 downto 16) +
                Coefficient 8 (29) & Coefficient 8 (29 downto 16)
Reload index 5 (17 downto 0) = "00" & Coefficient 9 (15 downto 0) +
                "00" & Coefficient 11 (15 downto 0)
Reload index 5 (32 downto 18) = Coefficient 9 (29) & Coefficient 9 (29 downto 16) +
                Coefficient 11 (29) & Coefficient 11 (29 downto 16)
Reload index 6 (17 downto 0) = "00" & Coefficient 1 (15 downto 0)
Reload index 6 (32 downto 18) = Coefficient 1 (29) & Coefficient 1 (29 downto 16)
Reload index 7 (17 downto 0) = "00" & Coefficient 4 (15 downto 0)
Reload index 7 (32 downto 18) = Coefficient 4 (29) & Coefficient 4 (29 downto 16)
Reload index 8 (17 downto 0) = "00" & Coefficient 0 (15 downto 0) -
                "00" & Coefficient 2 (15 downto 0)
Reload index 8 (32 downto 18) = Coefficient 0 (29) & Coefficient 0 (29 downto 16) -
                Coefficient 2 (29) & Coefficient 2 (29 downto 16)
Reload index 9 (17 downto 0) = "00" & Coefficient 3 (15 downto 0) -
                "00" & Coefficient 5 (15 downto 0)
Reload index 9 (32 downto 18) = Coefficient 3 (29) & Coefficient 3 (29 downto 16) -
                Coefficient 5 (29) & Coefficient 5 (29 downto 16)
Reload index 10 (17 downto 0) = "00" & Coefficient 0 (15 downto 0) +
                "00" & Coefficient 2 (15 downto 0)
Reload index 10 (32 downto 18) = Coefficient 0 (29) & Coefficient 0 (29 downto 16) +
                 Coefficient 2 (29) & Coefficient 2 (29 downto 16)
Reload index 11 (17 downto 0) = "00" & Coefficient 3 (15 downto 0) +
                "00" & Coefficient 5 (15 downto 0)
Reload index 11 (32 downto 18) = Coefficient 3 (29) & Coefficient 3 (29 downto 16) +
                 Coefficient 5 (29) & Coefficient 5 (29 downto 16)
```

Figure 48: **Reload Order Text File Format Example 3**

Coefficient Quantization

The FIR Compiler offers three coefficient quantization options: Integer Coefficient, Quantize Only, and Maximize Dynamic Range. When the coefficients are specified using Radix 2 (binary) and 16 (hexadecimal), only the "Integer Coefficients" option is available, as the coefficients are considered to have already been quantized. When the coefficients are specified using integer numbers, all of the quantization options are available. When the coefficients are specified using non-integer decimal numbers (containing fractional information), only the "Quantize Only" and "Maximize Dynamic Range" options are available.

Integer Coefficients

The "Integer Coefficients" quantization option analyzes the coefficients and determines the minimum number of bits required to represent the coefficients. The coefficient width must be equal to or greater than this value. When more bits are specified than required, the coefficients are sign extended. If the user wishes to truncate the coefficients, the "Quantize Only" option must be used.

Quantize Only

Primarily for use when the filter coefficients have been specified using non-integer real numbers, this option quantizes the coefficients to the specified coefficient bit width. The coefficient values are rounded to the nearest quantum using a simple round towards zero algorithm. The coefficient word is split into integer and fractional bits. The integer width is determined by analyzing the filter coefficients to find the maximum integer value. The remaining bits are allocated to represent the fractional portion of the coefficient values. When the specified coefficient bit width is less than the required integer bit width, coefficients are appropriately rounded. The default value for the Coefficient Fractional Bits parameter is set to maximize the precision of the coefficients, but it can be reduced by the user. In this circumstance, more bits are allocated to the integer portion of the word, and the coefficient values are sign extended appropriately. When all the specified coefficients are between 1 and -1, only a single integer bit is required (to convey sign information), with the remainder of the coefficient word being used for fractional bits. When the coefficient range reduces further, the fractional bit width can be specified to a value greater than or equal to the coefficient width. See the Best Precision Fractional Length section for further explanation.

The frequency response of the quantized filter coefficients are compared to the ideal response on the Frequency Response Tab. This enables the user to explore the trade-off between filter performance and resources by varying the coefficient width parameter.

Maximize Dynamic Range

The user can also choose to scale the coefficients to utilize the full dynamic range provided by the coefficient bit width by selecting the Maximize Dynamic Range option. If selected, this results in the filter coefficients being scaled up by a common factor such that the largest coefficient (usually the center tap) is equal to the maximum representable value using the chosen bit width, then quantized. The overall scale factor is calculated as the ratio of the sum of the scaled and quantized coefficients to the sum of the original (ideal) coefficients. This value is calculated by the FIR Compiler and is presented (in dB) as part of the legend text on the filter response graph, or on the Summary page in the CORE Generator software GUI.

The filter response plot for the quantized coefficients is scaled down by the scale factor for easy comparison against the ideal coefficients.

Scaling the coefficients introduces a gain which should be taken into account in the user design.

Example 1

For this example the coefficients are signed with a coefficient width of 10 bits and a coefficient fractional width of 5 bits (using the Mathworks Fix format notation Fix10_5). The specified coefficients range between -12.34 and +13.88.

Considering the coefficient bit width as integer only, 10 bits give a maximum positive value of 511 and a maximum negative value of -512. The fractional bit width is 5 bits; this gives a maximum representable positive number of 511/(2^5)=15.96875 and a maximum representation negative number of -512/(2^5)=-16. All coefficients are scaled by the factor 15.96875/13.88=1.1504863 (=+1.2176dB) prior to quantization. The overall scaling factor is calculated as defined previously and displayed in the core GUI.

Example 2

For this example the coefficients are signed with a coefficient width of 18 bits and a coefficient fractional width of 19 bits, or Fix18_19. The specified coefficients range between -0.000256022 and +0.182865845.

An integer coefficient width of 18 bits gives a maximum positive value of 131071 and a maximum negative number of -131072. Considering the fractional bits, this gives a maximum representable positive number of 131071/(2^19)=0.249998092 and a maximum representable negative number of 131072/(2^19)=0.25. The scaling factor is determined by dividing the maximum value that can be represented (for the specified number of coefficient bits) by the maximum coefficient value. In this case 0.249998092/0.182865845=1.367112009 (=+2.716081962dB).

Note: While an appreciable improvement in performance can be achieved by making use of the full dynamic range of the coefficient bit width, this is not always the case, and the user must be satisfied that any changes are acceptable via the frequency response plot. The user must also account appropriately for any additional gain introduced by coefficient scaling elsewhere in the application system. In many systems, signal scaling can be arbitrary and no gain compensation is required; where scaling is necessary, it is often desirable to amalgamate gains inherent in a signal processing chain and compensate or adjust for these gains either at the front end (for example, in an Automatic Gain Control circuit) or the back end (for example, in a Constellation Decoder unit) of the chain. If the user wishes not to introduce any additional scaling into the design, “Quantize Only” should be chosen.

Best Precision Fractional Length

When the “Best Precision Fractional Length” option is selected, the coefficient fractional width is set to maximize the precision of the specified filter coefficients. As discussed in the Quantize Only section, the FIR Compiler analyzes the filter coefficients to determine how many bits are required to represent the integer portion of the coefficient values. All the remaining coefficient bits are then allocated to represent the fractional portion of the coefficients. When all the specified coefficients are between 1 and -1, only a single integer bit is required. The reminder of the coefficient word is then used for fractional bits. When the coefficient range reduces further, the fractional bit width is specified to a value greater than or equal to the coefficient width; otherwise the coefficient values contains redundant information that does not need to be explicitly stored. The available coefficient bits can then be better used to increase the precision of the coefficient values. This section goes on to illustrate this concept further. The MathWorks Fix Format notation is used, **Fixword length_fractional length**. The word length is specified by the Coefficient Width parameter, and the fractional length is specified by the Coefficient Fractional Bits parameter.

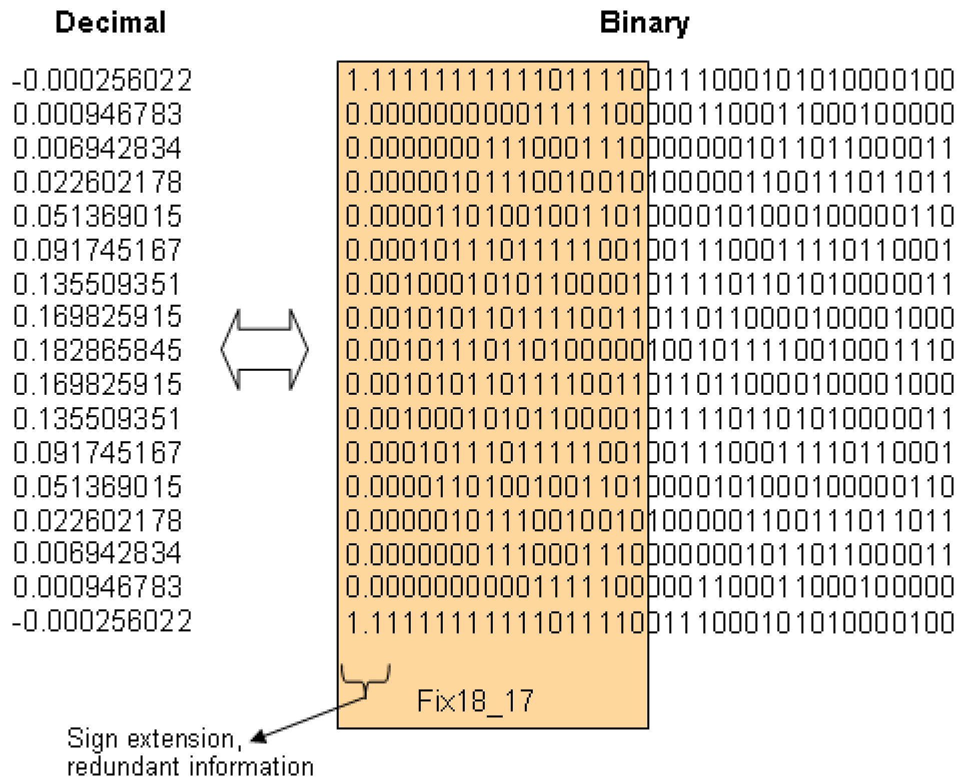


Figure 49: **Coefficient Quantization Fix18_17**

In Figure 49 the coefficient values are represented using 18 bits. The binary point is positioned such that 17 bits are used to represent the fractional portion of the number. An analysis of the coefficients reveals that no value has a magnitude greater than 0.25; therefore, the upper two MSBs are a sign extension and contain redundant information.

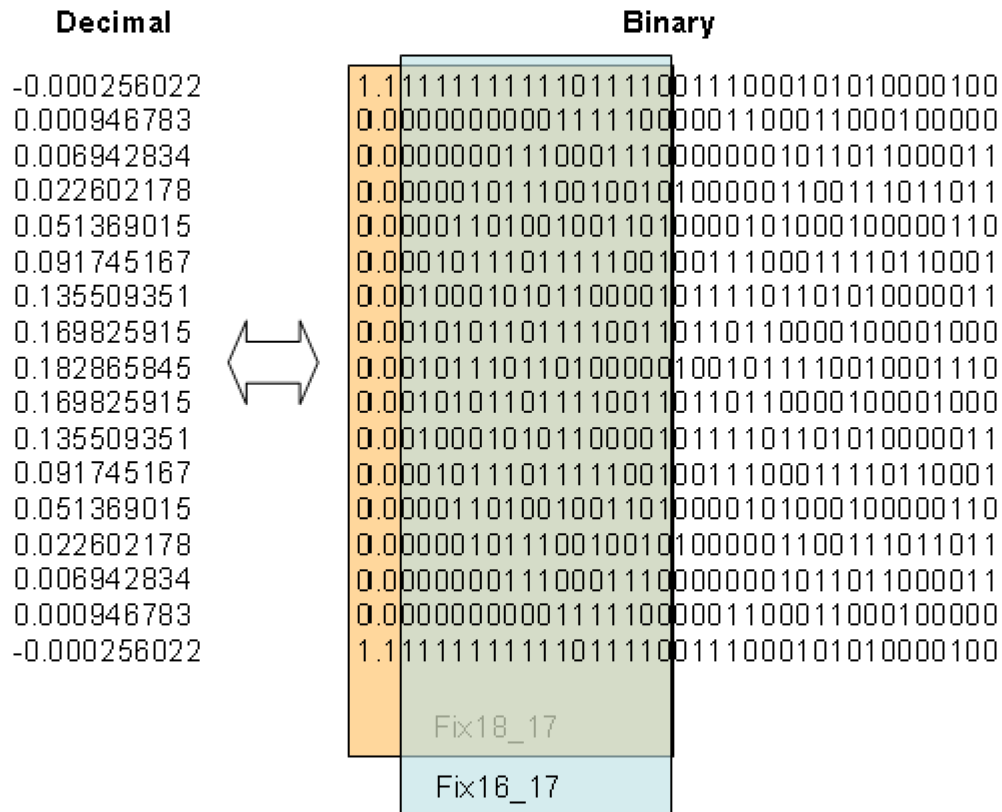


Figure 50: **Coefficient Quantization Fix16_17**

In Figure 50, 16 bits are used to represent the same coefficient values to the same precision. The redundant information has been removed, reducing the resources required to store the filter coefficients. The binary point position has not moved. 17 bits are still effectively used to represent the fractional portion of the number, but one of them does not need to be explicitly stored, as it is a sign extension of the stored MSB.

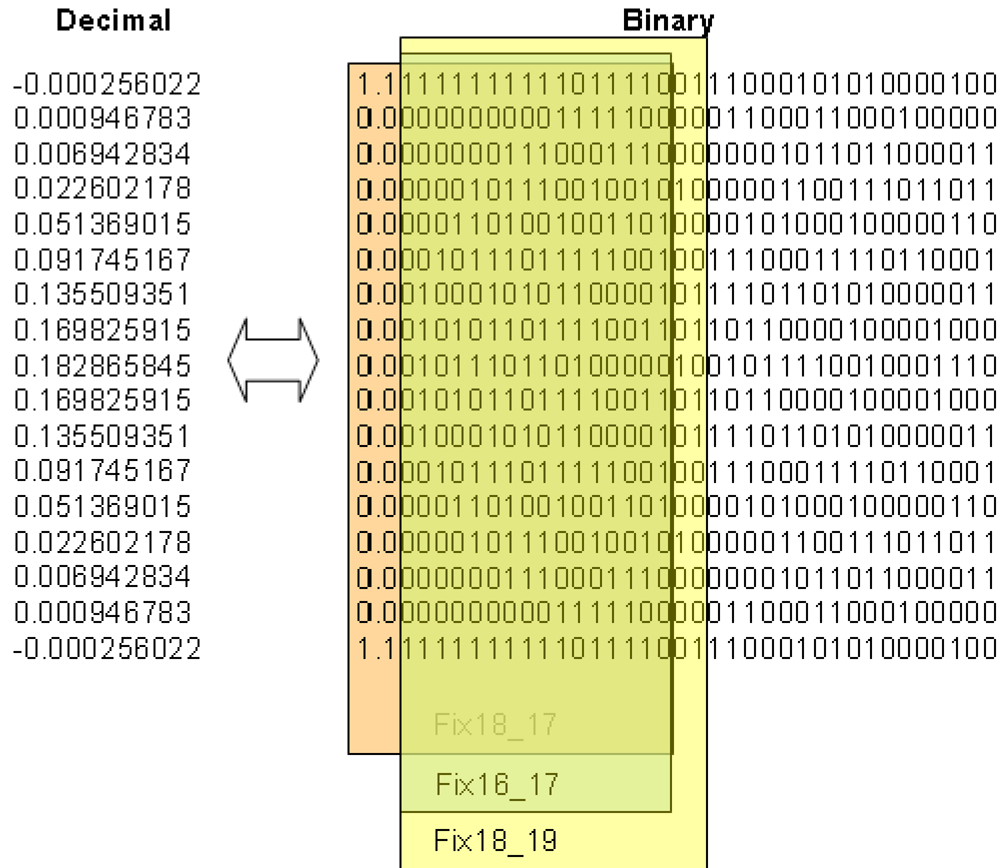


Figure 51: **Coefficient Quantization Fix18_19**

In Figure 51 18 bits are specified for the coefficient width. The two additional bits can be used to increase the precision. The binary point position has still not moved, but now, 19 bits are effectively used to represent the fractional portion of the number, which results in an increase of the filter precision.

Output Width and Bit Growth

The full precision output width can be defined as the input data width plus the bit growth due to the application of the filter coefficients. Bit growth from the original sample width occurs as a result of the many multiplications and additions that form the basic function of the filter. Therefore, the accumulator result width is significantly larger than the original input sample width. Limiting the accumulator width is desirable to save resources, both in the filter output path (such as output buffer memory, if present) and in any subsequent blocks in the signal processing chain. The worst case bit growth can be obtained by adding the coefficient width to the base 2 logarithm of the number of non-zero multiplications required (rounded up); however, this does not take into account the actual coefficient values. Taking the base 2 logarithm of the sum of the absolute value of all filter coefficients reveals the true maximum bit growth for a fixed coefficient filter, and this can be used to limit the required accumulator width. The following equation demonstrates this calculation, where B is the calculated bit growth, N is the number for filter coefficients, and a_n is n^{th} filter coefficient.

$$B = \mathrm{ceil}\left[\log_2\left(\sum_{n=0}^{(N-1)} |a_n|\right)\right]$$

For MAC implementations the FIR Compiler automatically calculates the bit growth based on the actual coefficient values. For reloadable filters the worst case bit growth is used.

The Coefficient (and Data) fractional width does not affect the output width calculation. The core determines the output width without considering fractional bits. The core determines the full precision output as previously described and then determines the output fractional width by summing the data and coefficient fractional bit width.

Output Rounding

As mentioned in Output Width and Bit Growth, it is desirable to limit the output sample width of the filter to minimize resource utilization in downstream blocks in a signal processing chain. For MAC implementations the FIR Compiler includes features to limit the output sample width and round the result to the nearest representable number within that bit width. Several rounding modes are provided to allow the user to select the preferred trade-off between resource utilization, rounding precision, and rounding bias:

- Full Precision
- Truncation (removal of LSBs)
- Non-symmetric rounding (towards positive or negative)
- Symmetric rounding (towards zero or infinity)
- Convergent rounding (towards odd or even)

In the following descriptions, the variable x is the fractional number to be rounded, with n representing the output width (that is, the integer bits of the accumulator result) and m representing the truncated LSBs (that is, the difference between the accumulator width and the output width). In Figure 52 through Figure 54, the direction of inflexion on the red midpoint markers indicates the direction of rounding.

Full Precision

In Full Precision mode, no output sample bit width reduction is performed (n=accumulator width, m=0). This is the default option.

Truncation

In Truncation mode, the m LSBs are removed from the accumulator result to reduce it to the specified output width; the effect is the same as the MATLAB software function *floor(x)*. This has the advantage that it can be implemented with zero resource cost, but has the disadvantage of being biased towards the negative by 0.5.

Non-symmetric Rounding to Positive

In this rounding mode, a binary value corresponding to 0.5 is added to the accumulator result and the m LSBs are removed; this is equivalent to the MATLAB software function *floor(x+0.5)*. The addition can usually be done in most filter configurations with little or no resource cost in hardware using the DSP slice features. It has the disadvantage of being biased towards the positive by $2^{-(m+1)}$.

Non-symmetric Rounding to Negative

In a modification of the preceding technique, a binary value corresponding to 0.499... is added to the accumulator result and the m LSBs are removed; this is equivalent to the MATLAB software function *ceil(x-0.5)*. The resource usage advantage is the same, but the bias in this case is towards the negative by $2^{-(m+1)}$.

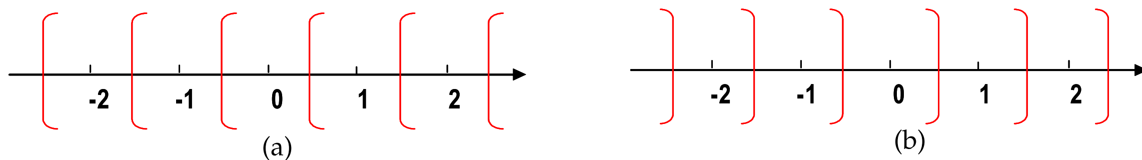


Figure 52: **Non-symmetric Rounding (a) to Positive (b) to Negative**

Symmetric Rounding to Highest Magnitude

The bias incurred during non-symmetric rounding occurs because rounding decisions at the midpoints always go in the same direction. In symmetric rounding, the decision on which direction to round is based on the sign of the number. For rounding towards highest magnitude, a binary value corresponding to 0.499 is added to the accumulator result, and the inverse of the accumulator sign bit is added as a carry-in before removal of the m LSBs. As is generally the case, there are as many positive as negative numbers; the result should not be biased in either direction. This rounding mode is commonly used in general applications, mainly due to the fact that it is equivalent to the MATLAB software function *round(x)*.

Symmetric Rounding to Zero

The implementation difference for this mode from round to highest magnitude is that the sign bit is used directly as the carry-in (see Figure 53). There is no direct MATLAB software equivalent of this operation. One minor advantage of rounding towards zero is that it does not cause overflow situations.

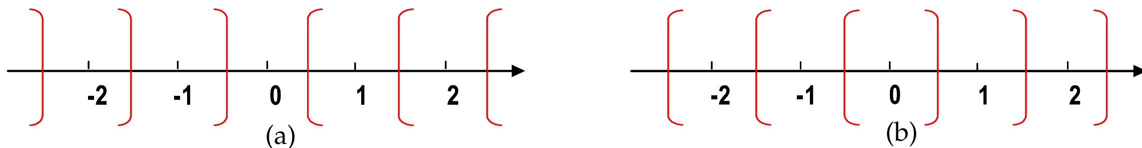


Figure 53: **Symmetric Rounding (a) to Highest Magnitude (b) to Zero**

Approximation of Symmetric Rounding

One important point to note about symmetric rounding mode is that to achieve the correct result, the sign of the accumulator must be known before the addition of the rounding constant to generate the correct carry-in. This requires an additional processing cycle to be available. When the additional cycle is not available and the user wishes to maintain full accuracy, a separate rounding unit must be used. (FIR Compiler calculates whether or not this is required automatically.)

An alternative technique is available to users who wish to employ symmetric rounding but do not have a spare cycle available, if they are willing to accept some inaccuracies. The rounding constant can be added on the initial loading of the accumulator, and the sign bit can be checked on the penultimate accumulation cycle and added on the final accumulation. This normally achieves the same result, but there is a small risk that the accumulated result changes sign between the penultimate and final accumulation cycles, which causes the midpoint decision to go in the wrong direction occasionally.

It is important to note that while some implementations of this approximation technique rearrange the calculation order of coefficients and data such that the smallest coefficient is used last, the FIR Compiler does not perform any rearrangement of coefficients and data. This is significant for symmetric filters, as the centre coefficient is the final coefficient calculated. For non-symmetric filters, the final coefficient is often very small and would be unlikely to affect the sign of the final result. It is also important that the risk of the sign changing between the penultimate and final accumulation cycles increases as the level of parallelism employed in the core increases. This is due to the contribution added to the accumulation on each cycle increasing as the number of cycles per output decreases. Therefore, it is important that users consider carefully the coefficient structure and level of parallelism they intend to use before deciding on whether to employ approximation of symmetric rounding.

Convergent Rounding

Convergent rounding chooses the rounding direction for midpoints as either toward odd or even numbers, rather than toward positive or negative (Figure 54). This can be advantageous, as the balance of rounding direction decisions for midpoints is based on the probability of occurrence of odd or even numbers, which is generally equal in most scenarios, even when the mean of the input signal moves away from zero. The function is achieved by adding a rounding constant, as in other modes, but then checking for a particular pattern on the LSBs to detect a midpoint and forcing the LSB to be either zero (for round to even) or one (for round to odd) when a midpoint occurs.

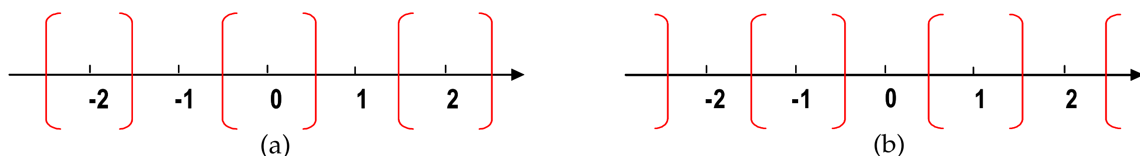


Figure 54: **Convergent Rounding (a) to Even (b) to Odd**

Resource Implications of Rounding

The implications with regard to resource utilization of selecting a particular rounding mode should be considered by users. Generally, the FIR Compiler IP core attempts to integrate rounding functions with existing functions, which usually means the accumulator portion of the circuit. However, this is not always possible. In certain combinations of rounding mode, filter type and device family, an additional DSP slice must be used to implement the rounding function. The most important factor to consider is the inherent hardware support for each mode in each of the device families, but filter type and configuration also play a role. Convergent rounding requires pattern detection support, and, therefore, this mode is only available in Virtex-6, Virtex-7 and Kintex-7 devices.

Table 7 indicates the combinations of filter type and rounding type for which no extra DSP slice is likely to be required. Where all three DSP slice enabled device families are likely to support that combination of rounding mode and filter type without an additional DSP slice, a tick mark is entered; where none of the three is likely to support the combination without the additional DSP slice, a check mark is entered; where there is a list of families provided, the list refers to those families that support the combination without an extra DSP slice. Support for symmetric rounding assumes that either there is a spare cycle available, or approximation is allowed. If this is not the case, an additional DSP slice is always required for symmetric rounding modes, regardless of filter type or family.

It is important to note that the table is indicative only, and certain combinations for which hardware support is indicated actually require the extra DSP, and vice versa. Notable exceptions to the table include parallel multichannel decimation with symmetric rounding (approximated), which requires an additional DSP slice.

Table 7: **Indicative Table of Hardware Support for Rounding Modes for Particular Filter Types**

Filter Type	Non-symmetric	Symmetric (Infinity)	Symmetric (Zero)	Convergent
Single Rate	✔	Virtex-6 Virtex-7 Kintex-7	Virtex-6 Virtex-7 Kintex-7	Virtex-6 Virtex-7 Kintex-7
Half-band	✔	Virtex-6 Virtex-7 Kintex-7	Virtex-6 Virtex-7 Kintex-7	Virtex-6 Virtex-7 Kintex-7
Interpolating without Symmetry	✔	Virtex-6 Virtex-7 Kintex-7	Virtex-6 Virtex-7 Kintex-7	Virtex-6 Virtex-7 Kintex-7
Interpolate by 2, Odd Symmetry	✔	Virtex-6 Virtex-7 Kintex-7	Virtex-6 Virtex-7 Kintex-7	Virtex-6 Virtex-7 Kintex-7
Interpolating with Symmetry (others)	✘	✘	✘	✘
Interpolating Half-band	✔	Virtex-6 Virtex-7 Kintex-7	✘	Virtex-6 Virtex-7 Kintex-7
Decimating, Single-channel	✔	Virtex-6 Virtex-7 Kintex-7	Virtex-6 Virtex-7 Kintex-7	Virtex-6 Virtex-7 Kintex-7
Decimating, Multichannel	✔	Virtex-6 Virtex-7 Kintex-7	Virtex-6 Virtex-7 Kintex-7	Virtex-6 Virtex-7 Kintex-7
Decimating Half-band	✔	Virtex-6 Virtex-7 Kintex-7	Virtex-6 Virtex-7 Kintex-7	Virtex-6 Virtex-7 Kintex-7

Multiple Column Filter implementation

The FIR Compiler can build filter implementations that span multiple DSP slice columns. The multi-column implementation is only required when the filter parameters, specifically the number of filter coefficients and the hardware oversampling rate (Sample Frequency to Clock Frequency ratio), result in an implementation that requires more DSP slices than are available in a single column of the select device. Figure 55 illustrates the structures implemented.

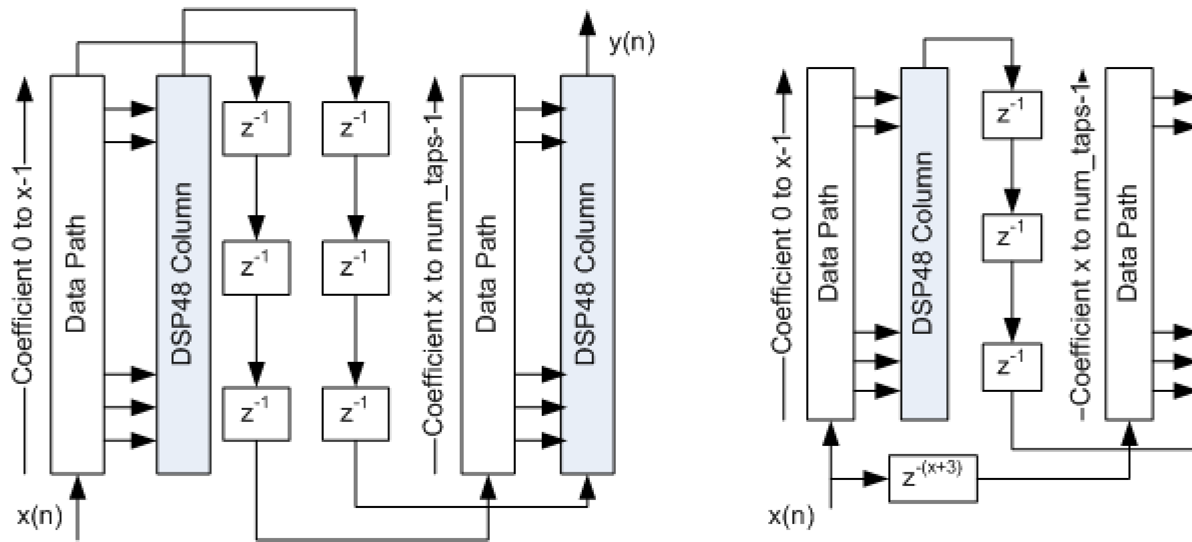


Figure 55: **Multi-column Implementations: Standard Implementation, Left; DSP Slice Data Cascade Port Implementation, Right**

This feature is only available when the Multiply-Accumulate filter architectures are selected on device families with more than one DSP slice column. Currently this feature is not supported for symmetric coefficient structures. To ensure this feature is available, set the Coefficient Structure parameter to “Non Symmetric.” See the Multiple Channels and Symmetric Filters section of Resource Considerations for further details on how to implement large symmetric filters.

The DSP column lengths are displayed on the Details Implementation Options page of the CORE Generator software GUI. The implemented column lengths can be determined automatically, Multi-column Support: Automatic, or specified by the user, Multi-column Support Automatic. The length of each implemented DSP column can be specified using the Column Configuration parameter. See the Detailed Implementation Options Screen section for more details.

Resource Considerations

The number of DSP slices utilized by the FIR Compiler is primarily determined by the number of coefficients, modified by any rate change, and the hardware oversampling rate per channel (defined by the Sample Period or the Sample frequency to Clock frequency ratio divided by the number of channels). Users should also be aware that Data and Coefficient Bit Width and Output Rounding Selection can also affect the DSP slice usage and are discussed in the following sections.

Tab 3: Implementation Details of the CORE Generator software GUI displays the core DSP slice usage given all the core parameters.

Data and Coefficient Bit Width

When the FIR Compiler is configured to implement the Multiply-Accumulate filter architectures, the DSP slice resource usage is influenced by the data and coefficient width specified. When the data and coefficient widths are specified to be greater than the input width of the DSP slice for the given device family, the core uses multiple DSP slice columns to implement the filter. Table 8 provides a guide to the number of DSP columns that are required for various combinations of data and coefficient widths. The widths used are that of the specification values and not that of the AXI-Stream data bus widths.

Table 8: **DSP Slice Column Usage for Given Data and Coefficient Widths**

Family	Data Width		Coefficient Width		Number of DSP Slice Columns
	Unsigned	Signed	Unsigned	Signed	
Spartan-6	<=17	<=18	<=17	<=18	1
	>17	>18	<=17	<=18	2
	<=17	<=18	>17	>18	2
	>17	>18	>17	>18	4
Virtex-6, Virtex-7, Kintex-7(1)	<=24	<=25	<=17	<=18	1
	<=17	<=18	<=24	<=25	1
	>24	>25	<=17	<=18	2
	<=17	<=18	>24	>25	2
	>17	>18	<=24	<=25	2
	<=24	<=25	>17	>18	2
	>24	>25	>17	>18	4
	>17	>18	>24	>25	4

1. **Note:** The data/coefficient widths at which Virtex-6, Virtex-7, Kintex-7 FPGA implementations transition to multi-column implementations might be lower than that shown based on the number of filter coefficients. This ensures that the accumulator width does not exceed 48 bits, thereby avoiding overflow.

Output Rounding Selection

The selected output rounding mode might cause additional DSP slice resources to be used. See the Output Rounding section for more details.

Multiple Channels and Symmetric Filters

When a filter is configured to use multiple channels, the number of clock cycles available to process the filter is reduced compared to implementing the same filter for a single channel. For example, a single-channel filter with a sample frequency of 1 MHz and a clock frequency of 4 MHz gives four clock cycles per input sample to generate an output. If the same sample and clock frequency is retained but two channels are processed, the core input is time-division multiplexed between the two channels, reducing the clock cycles per input sample to two. This results in a proportional increase in the number of DSP slices utilized.

An issue can arise when the FIR Compiler has detected that the specified filter coefficients have a symmetric coefficient structure (see the Filter Symmetry section for more details on utilizing coefficient symmetry), but the implementation still requires more DSP slices than are available in a single DSP slice column of the selected device. As symmetry is not supported by the multi-column implementation, it is not possible to generate this filter configuration.

To enable this feature, the coefficient structure would have to be set to "Non Symmetric," but the DSP slice resources required are doubled. When the filter has been configured to support multiple channels, an alternative implementation is possible.

Splitting the channels to be implemented across multiple parallel datapaths result in each datapath having more cycles available to process the filter coefficients, reducing the number of DSP slices required in a single column. It might then be possible to implement the filter configuration.

For example, a filter with 96 symmetric coefficients implementing four channels with a sample frequency of 1 MHz and a clock frequency of 4 MHz requires 48 DSP slices. If the selected device only has 32 DSP slices per column, this filter configuration cannot be generated. If the coefficient structure is set to "Non Symmetric," the implementation requires 96 DSP slices, but they can be split over three DSP slice columns. If the configuration is changed to implement two parallel datapaths with two time-division multiplexed channels per path, the core uses 25 DSP slices per parallel datapath (24 multiply-adds plus an accumulator), giving a total of 50 DSP slices. As the DSP slice column requirement is reduced from 48 to 25, the filter configuration can be generated.

Multiple Channel vs. Parallel Datapaths

The Multiple Channels and Symmetric Filters and Parallel Data Channel Filters features both offer the facility to process multiple input sample streams but using different interfaces. A multichannel interface requires the multiple input streams to be time division multiplexed (TDM) into a single core input, whereas the Parallel Datapaths interface provides an individual core input for each input stream. The choice of interface can influence the resources used by the core. In general, the multichannel implementation uses less DSP slice resources, but under some circumstances this is not the case. The following example demonstrates such a situation. It might also be desirable to consider the Parallel Datapaths implementation when implementing filter where a large number of DSP slices is required. This is discussed in the Multiple Channels and Symmetric Filters section.

Example 1

Consider an 8-tap single rate filter that is to process four 12.5 MHz input streams with a clock frequency of 100 MHz.

Multichannel implementation:

100 MHz/12.5 MHz=8 clock cycles per input sample. Shared between the four input streams, 8/4=2, gives a hardware oversampling rate of 2. The 8 filter coefficients must be processed in 2 clock cycles. This gives 8/2=4 DSP slices, where the filter processes the first 4 coefficients on the first clock cycle and the remaining 4 coefficients on the second clock cycle. The two partial products must be summed together, so an additional accumulator DSP slice is required. This gives a total of **5** DSP slices.

Parallel Datapaths:

100 MHz/12.5 MHz=8 clock cycles per input sample. Each input stream can use the full 8 clock cycles to process the 8 filter coefficients. This gives 8/8=1 multiply-accumulate DSP slice. The core provides four input streams, each using 1 DSP slice. This gives a total of **4** DSP slices.

This demonstrates that the Parallel Datapath implementation offers a more efficient implementation.

If the input sample frequency was increased to 25 MHz per channel, this would not be the case, illustrated as follows.

Multichannel implementation:

8 taps/(100 MHz/25 MHz/4)=**8** DSP slices, no accumulator required.

Parallel Datapaths:

8 taps/(100 MHz/25 MHz)=2 DSP slices, plus 1 accumulator DSP slice gives 3 DSP slices per path. A total of **12** DSP slices are required.

Performance and Resource Utilization

This section provides indicative resource utilization figures for limited example filters. To be concise, codes are used in these tables to indicate particular configuration options; these are detailed in the following sections.

The maximum clock frequency results were obtained by double-registering input and output ports (using IOB flip-flops) to reduce dependence on I/O placement. The inner level of registers used a separate clock signal to measure the path from the input registers to the first output register through the core.

The resource usage results do not include the preceding "characterization" registers and represent the true logic used by the core. LUT counts include SRL32s.

The map options used were: `"map -ol high"`

The par options used were: `"par -ol high"`

Clock frequency does not take clock jitter into account and should be derated by an amount appropriate to the clock source jitter specification.

The maximum achievable clock frequency and the resource counts can also be affected by other tool options, additional logic in the FPGA device, using a different version of Xilinx tools, and other factors.

Resource Utilization for MAC-based FIR Filters (Virtex-6 FPGA)

Table 9 provides characterization data for Virtex-6 FPGAs using a XC6VLX75T-1FF784 and ISE software speed file version "PRODUCTION 1.11b 2010-10-25." Generally the overall filter performance is within 10% of the DSP slice clock rating for the given device speed grade, and often reaches this clock rate (although the Speed setting might be required to achieve this in some cases). Some fully parallel cases can be slower due to routing congestion. Block RAM counts quoted are for 18k blocks, which are often amalgamated into pairs for mapping to 36k locations where possible; therefore customers should bear this in mind if comparing these values with map results for their particular configuration.

Table 9: **MAC-based FIR Resource Utilization in Virtex-6 FPGAs**

Filter Type	Rate	# Coefficients	Symmetric	Half-band	Reloadable	Channels	Clocks/Sample / Channel	Input Width	Coefficient Width	Area/Speed	DSP48	Block RAM	LUT-FF pairs	Clock Fmax (MHz)
SingleRate	1	366				1	366	18	18	A	1	1	68	450
SingleRate	1	4				4	1	18	18	A	4	0	65	452
SingleRate	1	20				1	5	18	18	A	5	0	128	452
SingleRate	1	20				3	5	18	18	A	5	0	137	452
SingleRate	1	27				1	1	18	18	A	27	0	316	452
SingleRate	1	21	✔			2	1	17	18	A	11	0	208	450
Decimation	6	34	✔			1	3	16	16	A	1	0	147	451
Decimation	2	69	✔			1	18	16	16	A	1	0	193	452
SingleRate	1	19	✔			6	1	16	16	A	10	0	172	447
SingleRate	1	32				1	32	16	16	A	1	0	64	452
SingleRate	1	32				1	4	16	16	A	9	0	154	452
SingleRate	1	32				1	1	16	16	A	32	0	293	450
SingleRate	1	32	✔			1	32	16	16	A	1	0	84	452
SingleRate	1	32	✔			1	4	16	16	A	5	0	186	452
SingleRate	1	32	✔			1	1	16	16	A	16	0	159	452
SingleRate	1	32				3	4	16	16	A	9	0	165	411
SingleRate	1	32				3	1	16	16	A	32	0	296	450
SingleRate	1	32	✔			3	4	16	16	A	5	0	219	452
SingleRate	1	32	✔			3	1	16	16	A	16	0	279	452
Interpolation	5	32				1	20	16	16	A	3	0	79	452
Interpolation	5	32				3	20	16	16	A	3	0	127	452
Interpolation	5	61	✔			3	5	16	16	A	8	0	310	452
Interpolation	5	61	✔			3	20	16	16	A	3	0	214	452
Interpolation	2	31	✔	✔		1	8	16	16	A	2	0	135	452

Table 9: **MAC-based FIR Resource Utilization in Virtex-6 FPGAs *(Cont'd)***

Filter Type	Rate	# Coefficients	Symmetric	Half-band	Reloadable	Channels	Clocks/Sample / Channel	Input Width	Coefficient Width	Area/Speed	DSP48	Block RAM	LUT-FF pairs	Clock Fmax (MHz)
Interpolation	5/3	64				3	10	16	16	A	4	0	177	452
Decimation	5	32				1	4	16	16	A	3	0	95	452
Decimation	5	32				3	4	16	16	A	3	0	213	452
Decimation	5	64				3	1	16	16	A	8	0	381	450
Decimation	5	64				3	4	16	16	A	3	0	384	450
Decimation	5	64				3	13	16	16	A	1	0	208	450
Decimation	2	31	✔	✔		1	3	16	16	A	3	0	156	452
Decimation	3/5	64				3	10	16	16	A	3	0	225	452
Interpolation	16	288			✔	16	16	18	18	A	18	1	871	452
Interpolation	8	144			✔	8	32	18	18	A	6	0	644	452
Interpolation	36/25	144				2	6	18	18	A	1	1	115	450
Interpolation	2	11	✔	✔		2	6	17	18	A	1	0	165	452
Interpolation	2	15	✔	✔		2	12	16	18	A	1	0	162	452
Interpolation	2	251	✔			2	24	16	18	A	7	0	88	452
Single Rate[4]	1	32				1	4	16	16	A	8	0	95	444
Interpolation[4]	5	32				1	20	16	16	A	2	0	102	452
Decimation[4]	5	32				1	4	16	16	A	2	0	68	452

Notes:

1. Clock rates determined using a -1 speed grade.
2. Clocks per sample per channel uses the input sample rate as the basis for all filter types.
3. Clock frequency does not take clock jitter into account and should be derated by an amount appropriate to the clock source jitter specification.
4. Implemented using the Transpose Multiply-Accumulate architecture.

Resource Utilization for MAC-based FIR Filters (Spartan-6 FPGA)

Table 10 provides characterization data for Spartan-6 FPGAs using a XC6SLX150-2FGG484 and ISE software speed file version "PRODUCTION 1.13a 2010-10-25." Generally the overall filter performance is within 10% of the DSP slice clock rating for the given device speed grade, and often reaches this clock rate (although the Speed setting might be required to achieve this in some cases). Some fully parallel cases can be slower due to routing congestion.

Table 10: **MAC-based FIR Resource Utilization in Spartan-6 FPGAs**

Filter Type	Rate	# Coefficients	Symmetric	Half-band	Reloadable	Channels	Clocks/Sample / Channel	Input Width	Coefficient Width	Area/Speed	DSP48	Block RAM	LUT-FF pairs	Clock Fmax (MHz)
SingleRate	1	366				1	366	18	18	A	1	1	94	251
SingleRate	1	4				4	1	18	18	A	4	0	61	251
SingleRate	1	20				1	5	18	18	A	5	0	126	251
SingleRate	1	20				3	5	18	18	A	5	0	166	251
SingleRate	1	27				1	1	18	18	A	27	0	314	251
SingleRate	1	21	✔			2	1	17	18	A	11	0	223	251
Decimation	6	34	✔			1	3	16	16	A	1	0	138	220
Decimation	2	69	✔			1	18	16	16	A	1	0	186	251
SingleRate	1	19	✔			6	1	16	16	A	10	0	185	251
SingleRate	1	32				1	32	16	16	A	1	0	79	250
SingleRate	1	32				1	4	16	16	A	9	0	160	251
SingleRate	1	32				1	1	16	16	A	32	0	298	221
SingleRate	1	32	✔			1	32	16	16	A	1	0	85	251
SingleRate	1	32	✔			1	4	16	16	A	5	0	204	251
SingleRate	1	32	✔			1	1	16	16	A	16	0	171	251
SingleRate	1	32				3	4	16	16	A	9	0	165	251
SingleRate	1	32				3	1	16	16	A	32	0	294	251
SingleRate	1	32	✔			3	4	16	16	A	5	0	187	251
SingleRate	1	32	✔			3	1	16	16	A	16	0	287	251
Interpolation	5	32				1	20	16	16	A	3	0	78	251
Interpolation	5	32				3	20	16	16	A	3	0	126	251
Interpolation	5	61	✔			3	5	16	16	A	8	0	325	251
Interpolation	5	61	✔			3	20	16	16	A	3	0	204	251
Interpolation	2	31	✔	✔		1	8	16	16	A	3	0	133	250
Interpolation	5/3	64				3	10	16	16	A	4	0	195	251
Decimation	5	32				1	4	16	16	A	3	0	104	251
Decimation	5	32				3	4	16	16	A	3	0	219	250
Decimation	5	64	✔			3	1	16	16	A	8	0	314	250
Decimation	5	64	✔			3	4	16	16	A	3	0	376	250

Table 10: **MAC-based FIR Resource Utilization in Spartan-6 FPGAs *(Cont'd)***

Filter Type	Rate	# Coefficients	Symmetric	Half-band	Reloadable	Channels	Clocks/Sample / Channel	Input Width	Coefficient Width	Area/Speed	DSP48	Block RAM	LUT-FF pairs	Clock Fmax (MHz)
Decimation	5	64	✔			3	13	16	16	A	1	1	193	251
Decimation	2	31	✔	✔		1	3	16	16	A	3	0	165	251
Decimation	3/5	64				3	10	16	16	A	3	0	224	251
Interpolation	16	288			✔	16	16	18	18	A	18	1	857	251
Interpolation	8	144			✔	8	32	18	18	A	6	0	643	251
Interpolation	36/25	144				2	6	18	18	A	1	1	104	251
Interpolation	2	11	✔	✔		2	6	17	18	A	1	0	125	250
Interpolation	2	15	✔	✔		2	12	16	18	A	1	0	120	251
Interpolation	2	251	✔			2	24	16	18	A	7	0	451	251
Single Rate[4]	1	32				1	4	16	16	A	8	0	87	251
Interpolation[4]	5	32				1	20	16	16	A	2	0	87	251
Decimation[4]	5	32				1	4	16	16	A	2	0	101	251

Notes:

1. Clock rates determined using a -2 speed grade.
2. Clocks per sample per channel uses the input sample rate as the basis for all filter types.
3. Clock frequency does not take clock jitter into account and should be derated by an amount appropriate to the clock source jitter specification.
4. Implemented using Transpose Multiply-Accumulate architecture.

References

1. Peled and B. Liu, *A New Hardware Realization of Digital Filters*, IEEE Trans. on Acoust., Speech, Signal Processing, vol. ASSP-22, pp. 456-462, Dec. 1974.
2. S. A. White, *Applications of Distributed Arithmetic to Digital Signal Processing*, IEEE ASSP Magazine, Vol. 6(3), pp. 4-19, July 1989.
3. C. H. Dick, *Implementing Area Optimized Narrow-Band FIR Filters Using Xilinx FPGAs*, SPIE International Symposium on Voice, Video and Data Communications—Configurable Computing: Technology an Applications Stream, Boston, Massachusetts USA, pp. 227-238, Nov 1-6, 1998
4. P.P. Vaidyanathan, *Multi-Rate Systems and Filter Banks,* Prentice Hall, Englewood Cliffs, New Jersey, 1993.
5. M. E. Frerking, *Digital Signal Processing in Communication Systems,* Van Nostrand Reinhold, New York, 1994.
6. Xilinx Inc., *XtremeDSP Design Manual,* Xilinx Inc., San Jose California, 2004.
7. Fred Harris, Chris Dick, and Michael Rice, *Digital Receivers and Transmitters Using Polyphase Filter Banks for Wireless Communications,* IEEE Trans. on Microwave Theory and Techniques, Vol. 51, No.4. 4 April 2003
8. Mou, Zhi-Jian, *Symmetry Exploitation in Digital Interpolators/Decimators*, IEEE Transactions on Signal Processing, Vol. 44 No. 10, Oct. 1996
9. *Xilinx AXI Design Reference Guide* (UG761)
10. AMBA 4 AXI4-Stream Protocol Version: 1.0 Specification

Support

Xilinx provides technical support at www.xilinx.com/support for this LogiCORE IP product when used as described in the product documentation. Xilinx cannot guarantee timing, functionality, or support of product if implemented in devices that are not defined in the documentation, if customized beyond that allowed in the product documentation, or if changes are made to any section of the design labeled *DO NOT MODIFY*.

Refer to the IP Release Notes Guide (XTP025) for further information on this core. On the first page there is a link to "All DSP IP." The relevant core can then be selected from the displayed list.

For each core, there is a master Answer Record that contains the Release Notes and Known Issues list for the core being used. The following information is listed for each version of the core:

- New Features
- Bug Fixes
- Known Issues

Ordering Information

This LogiCORE IP module is included at no additional cost with the Xilinx ISE Design Suite software and is provided under the terms of the Xilinx End User License Agreement. Use the CORE Generator software included with the ISE Design Suite to generate the core. For more information, please visit the core page.

Contact your local Xilinx sales representative for pricing and availability of additional Xilinx LogiCORE modules and software. Information about additional Xilinx LogiCORE modules is available on the Xilinx IP Center.

List of Acronyms

Acronym	Spelled Out
AXI	Advanced eXtensible Interface
DA	Distributed Arithmetic
dB	decibel
DSP	Digital Signal Processing
FDM	Frequency Division Multiplexed
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
I	In-Phase
IES	Incisive Enterprise Simulator
IFIR	Interpolated Finite Impulse Response
IP	Intellectual Property
ISE	Integrated Software Environment
ISIM	ISE Simulator
LSB	Least Significant Bit
LUT	Lookup Table

Acronym	Spelled Out
MAC	Multiply-Accumulate
MSB	Most Significant Bit
ND	New Data
PDA	Parallel Distributed Arithmetic
ps	picoseconds
PSC	Parallel-to-Serial Shift Register
Q	Quadrature
RAM	Random Access Memory
SDA	Serial Distributed Arithmetic
TDM	Time Division Multiplex or Time Domain Multiplex
TSB	Time-Skew Buffer
VHDL	VHSIC Hardware Description Language (VHSIC an acronym for Very High-Speed Integrated Circuits)
XCO	Xilinx CORE Generator core source file
XST	Xilinx Synthesis Technology

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
09/21/10	1.0	First release of the core with AXI interface support. The previous release of this document was ds534.
12/14/10	1.1	Release for 12.4 with additional features.
03/01/11	1.2	Support added for Virtex-7 and Kintex-7. ISE Design Suite 13.1.

Notice of Disclaimer

Xilinx is providing this product documentation, hereinafter "Information," to you "AS IS" with no warranty of any kind, express or implied. Xilinx makes no representation that the Information, or any particular implementation thereof, is free from any claims of infringement. You are responsible for obtaining any rights you may require for any implementation based on the Information. All specifications are subject to change without notice. XILINX EXPRESSLY DISCLAIMS ANY WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE INFORMATION OR ANY IMPLEMENTATION BASED THEREON, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF INFRINGEMENT AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Except as stated herein, none of the Information may be copied, reproduced, distributed, republished, downloaded, displayed, posted, or transmitted in any form or by any means including, but not limited to, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Xilinx.