LogiCORE IP FIR Compiler v7.0

Product Guide for Vivado Design Suite

PG149 March 20, 2013

Table of Contents

Introduction

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

Features

- AXI4-Stream-compliant interfaces
- High-performance finite impulse response (FIR), polyphase decimator, polyphase interpolator, half-band, half-band decimator and half-band interpolator, Hilbert transform and interpolated filter 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
- Support for advanced interleaved data channel sequences
- 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
- Efficient multi-column structures for all filter implementations and optimizations

LogiCORE IP Facts Table	
Core Specifics	
Supported Device Family[1]	Zynq™-7000, Virtex®-7, Kintex™-7, Artix™-7
Supported User Interfaces	AXI4-Stream
Resources	See Table 2-1 and Table 2-2
Provided with Core	
Design Files	Encrypted RTL
Example Design	Not Provided
Test Bench	VHDL
Constraints File	Not Provided
Simulation Model	VHDL Behavioral VHDL or Verilog Structural C Model
Supported S/W Driver	N/A
Tested Design Flows[2]	
Design Entry	Vivado™ Design Suite System Generator for DSP
Simulation	Mentor Graphics Questa® SIM Vivado Simulator
Synthesis	Vivado Synthesis
Support	
Provided by Xilinx @ www.xilinx.com/support	

Notes:

1. For a complete listing of supported devices, see Vivado IP Catalog.
2. For the supported versions of the tools, see the Xilinx Design Tools: Release Notes Guide.

Overview

A wide range of filter types can be implemented in the Vivado™ Integrated Design Environment (IDE): 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 1-1, where *N* is the number of filter coefficients.

$$y(k) = \sum_{n=0}^{N-1} a(n)x(k-n) \qquad k = 0, 1, \ldots$$

Equation 1-1

Figure 1-1 shows 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.

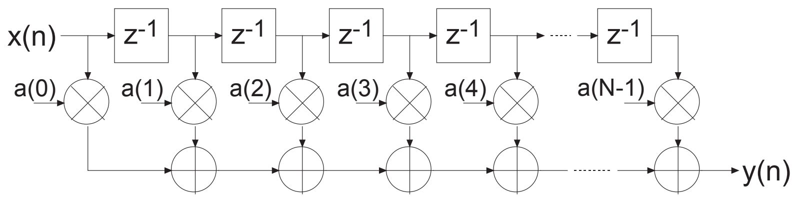


Figure 1-1: **Conventional Tapped Delay Line FIR Filter Representation**

Feature Summary

Table 1-1 and Table 1-2 show the features and filter configuration support for the FIR Compiler.

Feature Support Matrix

Table 1-1: **Feature Support Matrix**

Feature	Systolic Multiply-Accumulate	Transpose Multiply-Accumulate
Number of Coefficients	2–2048	2–2048
Coefficient Width(1)	2–49	2–49
Data Width(1,2)	2–49	2–49
Number of Channels	1–64	1
Parallel Datapaths(3)	1-16	1-16
Maximum Rate Change *Single Channel* *Multiple Channels*	 1024 512	 1024 N/A
Fractional Rate Support	Yes	No
Coefficient Reload	Yes	Yes
Coefficient Sets	1–256	1–256
Output Rounding	Yes	Yes

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 Vivado IDE might reduce further to ensure that the accumulator width does not exceed the maximum.
3. Maximum Parallel Datapaths reduces to 8 when Coefficient Width or Data Width is greater than 25-bits.

Table 1-2 shows the classes of filters that are supported for the FIR Compiler core.

Table 1-2: **Filter Configuration Support Matrix**

Filter Configuration	Supported
Conventional Single-rate FIR	Yes
Half-band FIR	Yes
Hilbert Transform [Ref 1]	Yes
Interpolated FIR [Ref 2] [Ref 3]	Yes
Polyphase Decimator	Yes
Polyphase Interpolator	Yes
Half-band Decimator	Yes
Half-band Interpolator	Yes

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

Notable Limitations

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

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

- Fractional Rate filters do not currently exploit coefficient symmetry.
- Non Half-band rate change filters utilizing the advanced channel sequence feature do not 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.

Licensing and Ordering Information

This Xilinx LogiCORE™ IP module is provided under the terms of the Xilinx Core License Agreement. The module is shipped as part of the Vivado Design Suite. For full access to all core functionalities in simulation and in hardware, you must purchase a license for the core. Contact your local Xilinx sales representative for information about pricing and availability.

For more information, visit the FIR Compiler product web page.

Information about other Xilinx LogiCORE IP modules is available at the Xilinx Intellectual Property page. For information on pricing and availability of other Xilinx LogiCORE IP modules and tools, contact your local Xilinx sales representative.

Product Specification

Performance

Maximum Frequencies

See Table 2-1 and Table 2-2.

Latency

The core latency is dependent on many of the core parameters. The Implementation Details Tab on the core GUI displays the core latency value, in clock cycles, given the current configuration.

Throughput

The core's throughput is completely configurable; from full throughput, one clock cycle per input sample, through to a completely over-sampled implementation. Refer to Hardware Oversampling Specification on the Channel Specification Screen of the core GUI for details.

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.

IMPORTANT: The figures in Table 2-1 and Table 2-2 were obtained from the ISE® Design Suite; Vivado™ Design Suite figures are expected to be similar.

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

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, using a different version of Xilinx tools, and other factors

Resource Utilization for Virtex-7 FPGA

Table 2-1 provides characterization data for Virtex®-7 FPGAs using a XC7VX330T-1FF1157 and ISE software speed specification version 'ADVANCED 1.01c 2011-08-29.' 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 figures are for 18K blocks. However these can be amalgamated to 36K blocks depending on the surrounding circuitry. This should be considered if comparing the table values to BRAM resource counts for a particular configuration.

Table 2-1: **XC7VX330T-1FF1157 FPGA Resource Estimates**

Filter Type	Rate	# Coefficients	Symmetric	Half-band	Reloadable	Channels	Clocks/Sample / Channel	Input Width	Coefficient Width	Area/Speed	DSP Slice	Block RAM	LUT-FF pairs	Clock F_{max} (MHz)
SingleRate	1	366				1	366	18	18	A	1	1	65	458
SingleRate	1	4				4	1	18	18	A	4	0	58	547
SingleRate	1	20				1	5	18	18	A	5	0	157	524
SingleRate	1	20				3	5	18	18	A	5	0	175	537
SingleRate	1	27				1	1	18	18	A	27	0	314	501
SingleRate	1	21	Y			2	1	17	18	A	11	0	261	523
Decimation	6	34	Y			1	3	16	16	A	1	0	146	534
Decimation	2	69	Y			1	18	16	16	A	1	0	194	521
SingleRate	1	19	Y			6	1	16	16	A	10	0	171	492
SingleRate	1	32				1	32	16	16	A	1	0	104	543
SingleRate	1	32				1	4	16	16	A	9	0	193	547
SingleRate	1	32				1	1	16	16	A	32	0	304	506
SingleRate	1	32	Y			1	32	16	16	A	1	0	125	547

Table 2-1: **XC7VX330T-1FF1157 FPGA Resource Estimates *(Cont'd)***

Filter Type	Rate	# Coefficients	Symmetric	Half-band	Reloadable	Channels	Clocks/Sample / Channel	Input Width	Coefficient Width	Area/Speed	DSP Slice	Block RAM	LUT-FF pairs	Clock F_{max} (MHz)
SingleRate	1	32	Y			1	4	16	16	A	5	0	249	524
SingleRate	1	32	Y			1	1	16	16	A	16	0	502	492
SingleRate	1	32				3	4	16	16	A	9	0	207	547
SingleRate	1	32				3	1	16	16	A	32	0	304	515
SingleRate	1	32	Y			3	4	16	16	A	5	0	259	542
SingleRate	1	32	Y			3	1	16	16	A	16	0	276	473
Interpolation	5	32				1	20	16	16	A	3	0	105	547
Interpolation	5	32				3	20	16	16	A	3	0	155	547
Interpolation	5	61	Y			3	5	16	16	A	8	0	344	517
Interpolation	5	61	Y			3	20	16	16	A	3	0	243	524
Interpolation	2	31	Y	Y		1	8	16	16	A	2	0	154	547
Interpolation	5/3	64				3	10	16	16	A	4	0	212	499
Decimation	5	32				1	4	16	16	A	3	0	136	538
Decimation	5	32				3	4	16	16	A	3	0	257	525
Decimation	5	64				3	1	16	16	A	8	0	392	507
Decimation	5	64				3	4	16	16	A	3	0	431	530
Decimation	5	64				3	13	16	16	A	1	1	225	458
Decimation	2	31	Y	Y		1	3	16	16	A	3	0	160	526
Decimation	3/5	64				3	10	16	16	A	3	0	228	469
Interpolation	16	288			Y	16	16	18	18	A	18	3	890	451
Interpolation	8	144			Y	8	32	18	18	A	6	0	640	476
Interpolation	36/25	144				2	6	18	18	A	1	3	111	458
Interpolation	2	11	Y	Y		2	6	17	18	A	1	0	166	546
Interpolation	2	15	Y	Y		2	12	16	18	A	1	0	167	503
Interpolation	2	251	Y			2	24	16	18	A	7	0	530	520
Single Rate[(4)]	1	32				1	4	16	16	A	8	0	120	534
Interpolation[(4)]	5	32				1	20	16	16	A	2	0	108	547

Table 2-1: **XC7VX330T-1FF1157 FPGA Resource Estimates *(Cont'd)***

Filter Type	Rate	# Coefficients	Symmetric	Half-band	Reloadable	Channels	Clocks/Sample / Channel	Input Width	Coefficient Width	Area/Speed	DSP Slice	Block RAM	LUT-FF pairs	Clock F_{max} (MHz)
Decimation(4)	5	32				1	4	16	16	A	2	0	133	533

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 Kintex-7 FPGA

Table 2-2 provides characterization data for Kintex-7 FPGAs using a XC7K160T-1FBG676 and ISE software speed specification version 'ADVANCED 1.02a 2011-08-29.' 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 figures are for 18K blocks. However these can be amalgamated to 36K blocks depending on the surrounding circuitry. This should be considered if comparing the table values to BRAM resource counts for a particular configuration.

Table 2-2: **XC7K160T-1FBG676 FPGA Resource Estimates**

Filter Type	Rate	# Coefficients	Symmetric	Half-band	Reloadable	Channels	Clocks/Sample / Channel	Input Width	Coefficient Width	Area/Speed	DSP Slice	Block RAM	LUT-FF pairs	Clock F_{max} (MHz)
SingleRate	1	366				1	366	18	18	A	1	1	63	458
SingleRate	1	4				4	1	18	18	A	4	0	58	547
SingleRate	1	20				1	5	18	18	A	5	0	157	547
SingleRate	1	20				3	5	18	18	A	5	0	180	547
SingleRate	1	27				1	1	18	18	A	27	0	314	499
SingleRate	1	21	Y			2	1	17	18	A	11	0	268	506
Decimation	6	34	Y			1	3	16	16	A	1	0	151	547
Decimation	2	69	Y			1	18	16	16	A	1	0	196	521
SingleRate	1	19	Y			6	1	16	16	A	10	0	171	531

Table 2-2: **XC7K160T-1FBG676 FPGA Resource Estimates *(Cont'd)***

Filter Type	Rate	# Coefficients	Symmetric	Half-band	Reloadable	Channels	Clocks/Sample / Channel	Input Width	Coefficient Width	Area/Speed	DSP Slice	Block RAM	LUT-FF pairs	Clock F_{max} (MHz)
SingleRate	1	32				1	32	16	16	A	1	0	104	547
SingleRate	1	32				1	4	16	16	A	9	0	192	547
SingleRate	1	32				1	1	16	16	A	32	0	302	496
SingleRate	1	32	Y			1	32	16	16	A	1	0	125	536
SingleRate	1	32	Y			1	4	16	16	A	5	0	239	547
SingleRate	1	32	Y			1	1	16	16	A	16	0	472	497
SingleRate	1	32				3	4	16	16	A	9	0	207	547
SingleRate	1	32				3	1	16	16	A	32	0	303	496
SingleRate	1	32	Y			3	4	16	16	A	5	0	264	520
SingleRate	1	32	Y			3	1	16	16	A	16	0	276	494
Interpolation	5	32				1	20	16	16	A	3	0	108	547
Interpolation	5	32				3	20	16	16	A	3	0	154	518
Interpolation	5	61	Y			3	5	16	16	A	8	0	363	533
Interpolation	5	61	Y			3	20	16	16	A	3	0	244	520
Interpolation	2	31	Y	Y		1	8	16	16	A	2	0	158	547
Interpolation	5/3	64				3	10	16	16	A	4	0	204	522
Decimation	5	32				1	4	16	16	A	3	0	137	545
Decimation	5	32				3	4	16	16	A	3	0	257	513
Decimation	5	64				3	1	16	16	A	8	0	392	496
Decimation	5	64				3	4	16	16	A	3	0	431	500
Decimation	5	64				3	13	16	16	A	1	1	218	458
Decimation	2	31	Y	Y		1	3	16	16	A	3	0	149	547
Decimation	3/5	64				3	10	16	16	A	3	0	229	485
Interpolation	16	288			Y	16	16	18	18	A	18	3	889	458
Interpolation	8	144			Y	8	32	18	18	A	6	0	644	497
Interpolation	36/25	144				2	6	18	18	A	1	3	111	458
Interpolation	2	11	Y	Y		2	6	17	18	A	1	0	165	530
Interpolation	2	15	Y	Y		2	12	16	18	A	1	0	164	547
Interpolation	2	251	Y			2	24	16	18	A	7	0	535	503
Single Rate(4)	1	32				1	4	16	16	A	8	0	120	538
Interpolation(4)	5	32				1	20	16	16	A	2	0	111	547

Table 2-2: **XC7K160T-1FBG676 FPGA Resource Estimates *(Cont'd)***

Filter Type	Rate	# Coefficients	Symmetric	Half-band	Reloadable	Channels	Clocks/Sample / Channel	Input Width	Coefficient Width	Area/Speed	DSP Slice	Block RAM	LUT-FF pairs	Clock F_{max} (MHz)
Decimation(4)	5	32				1	4	16	16	A	2	0	133	547

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.

Port Descriptions

Figure 2-1 shows the schematic symbol for the interface pins for the FIR Compiler core.

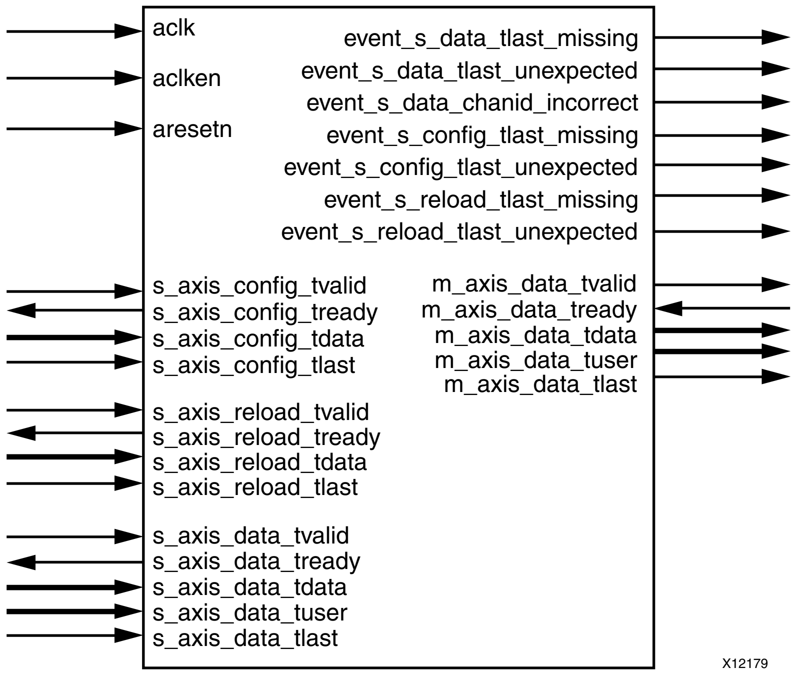


Figure 2-1: **FIR Compiler Core Pinout**

Table 2-3 defines the FIR filter port names and port functional descriptions.

Table 2-3: **Core Signal Pinout**

Name	Direction	Optional	Description
aclk	Input	no	Rising-edge clock
aclken	Input	yes	Active-High clock enable (optional).
aresetn	Input	yes	Active-Low synchronous clear (optional, always take priority over aclken). A minimum aresetn active pulse of two cycles is required
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.
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. See TDATA of the RELOAD Channel for internal structure and width.
s_axis_reload_tlast	Input	yes	TLAST for RELOAD channel. Indicates the last transfer of a packet of coefficients.
s_axis_data_tvalid	Input	no	TVALID for input DATA channel. Asserted by external master to indicate data is available for transfer.
s_axis_data_tready	Output	no	TREADY for input DATA channel. Asserted by core to indicate core is ready to accept data.
s_axis_data_tdata	Input	no	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.

Table 2-3: **Core Signal Pinout *(Cont'd)***

Name	Direction	Optional	Description
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	no	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	no	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.

Designing with the Core

This chapter includes guidelines and additional information to facilitate designing with the core.

Clocking

The core uses a single clock, `aclken`, which is common to all the AX4-Stream interfaces and event signals.

The optional clock enable signal, `aclken`, is used to qualify `aclk`. When `aclken` is de-asserted the core state and outputs are halted. Asserting `aclken` allows the core to continue processing.

Resets

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 data vector of the core. Selecting data vector reset can result in the core using more FPGA logic 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, indicates when the filter data memory has been completely flushed and can be used as additional qualification of the `m_axis_data_tdata` bus.

AXI4-Stream Considerations

The 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 on 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 to 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 [Ref 4] and [Ref 5].

Basic Handshake

Figure 3-1 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.

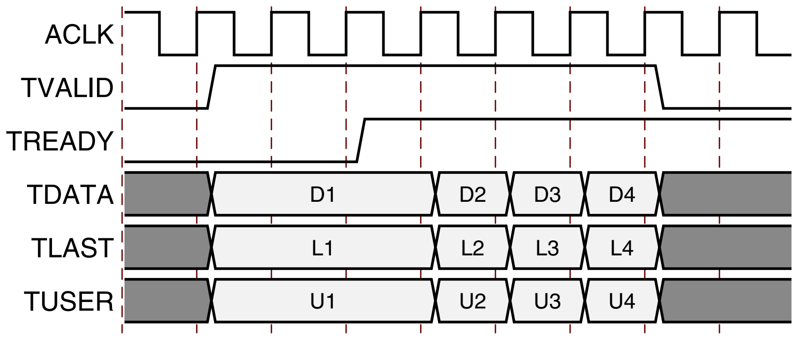


Figure 3-1: **Data Transfer in an AXI4-Stream Channel**

Input and Output DATA Channels

The basic operation of the FIR is for samples to enter through the input DATA channel (`s_axis_data_t*`) and exit through 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 (Parallel Data Channel Filters) 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 3-2 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 indexes shown in the diagram are as follows: A= 31, B = 26, C=15, D = 10, E = 31, F = 28, G=15, H = 12.

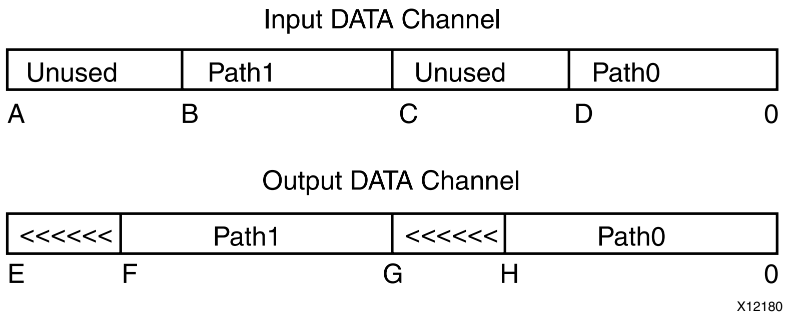


Figure 3-2: **TDATA Structure for Input and Output DATA Channels**

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 Resets for 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.

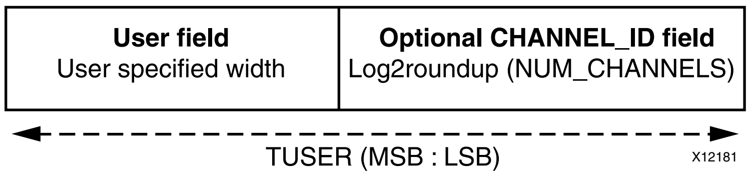


Figure 3-3: **TUSER**

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 the current channel sequence value. 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 has been specified to support advanced channel sequences, the channel pattern value specifies which channel sequence is to be used.
- 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 first transaction in the packet also includes the channel sequence ID, if required for the core configuration. If the core is configured to support configurable channel sequences but not multiple filter sets, then the packet length is 1.
- 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. This transaction also includes the channel pattern value, if required for the core configuration.

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 you can specify the core to synchronize the CONFIG 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 option 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

Each field of the TDATA bus is zero padded to an 8-bit boundary.

Field A = Filter Select; size log2roundup(NUM_FILTS)

Field B = Channel pattern; log2roundup(NUM_PATTERNS).

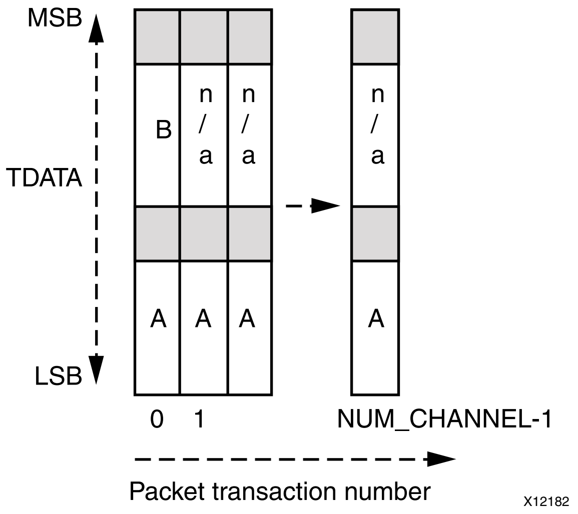


Figure 3-4: **TDATA structure for CONFIG channel**

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 several 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 used. See sections Coefficient Reload and Tab 4: Coefficient Reload 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 in CONFIG Channel.

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.

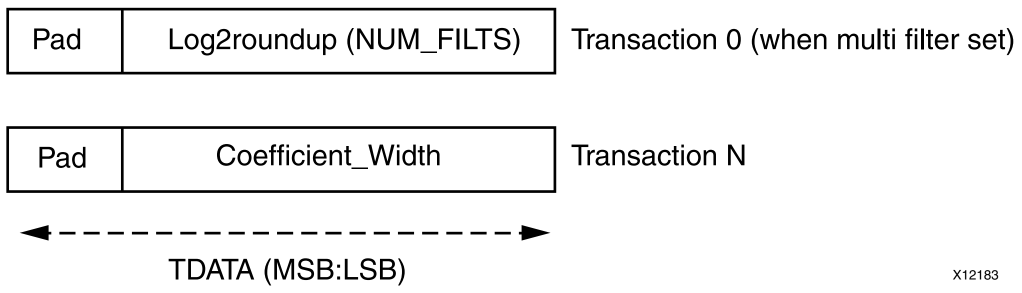


Figure 3-5: **TDATA Format**

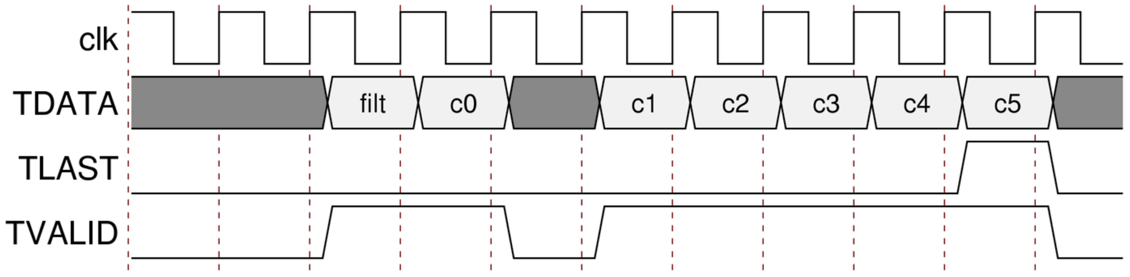


Figure 3-6: **TDATA Example Timing**

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.

Interface Timing

Figure 3-7 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.

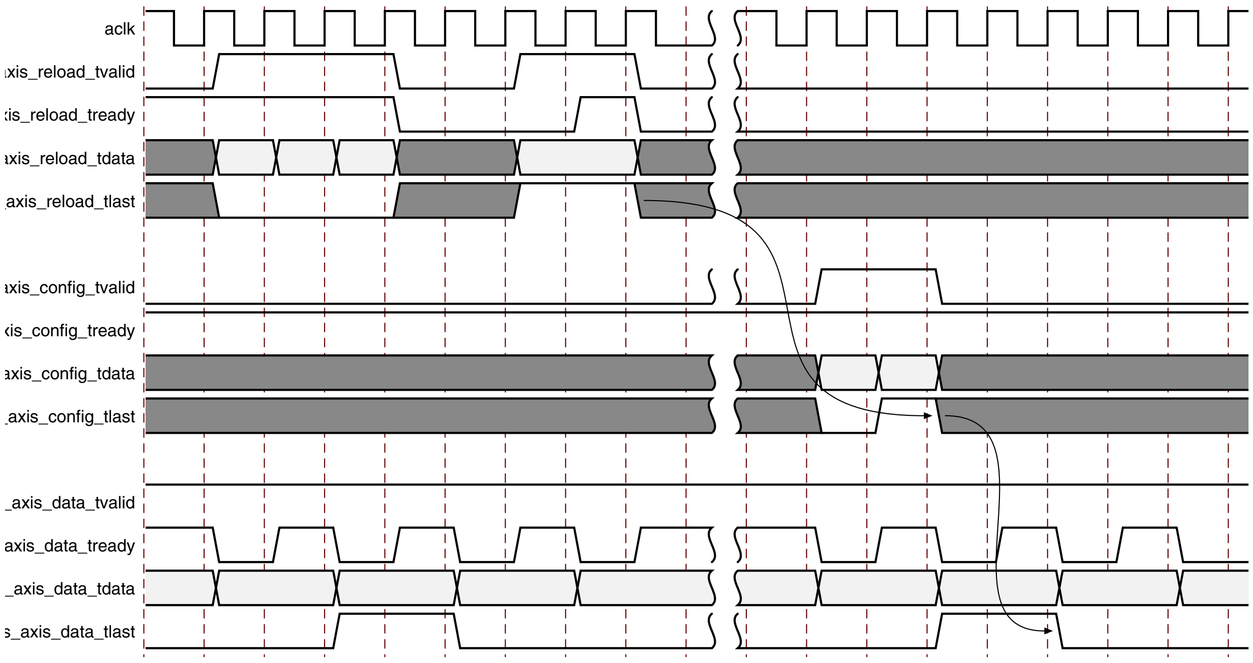


Figure 3-7: **Interface Timing**

Core Features

Filter Architectures

Multiply-Accumulate

Figure 3-8 shows a simplified view of a Multiply-Accumulate (MAC)-based FIR utilizing a single MAC engine.

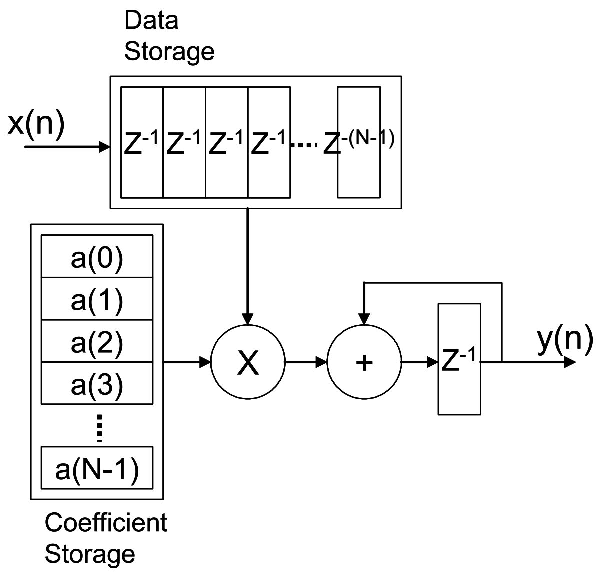


Figure 3-8: **Single MAC Engine Block Diagram**

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).

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.

Two MAC architectures are available in the FIR Compiler: one that implements a Systolic filter structure and the other a Transpose filter structure

Systolic Multiply-Accumulate

Figure 3-9 shows the Systolic Multiply-Accumulate architecture implementing a pipelined Direct-Form filter.

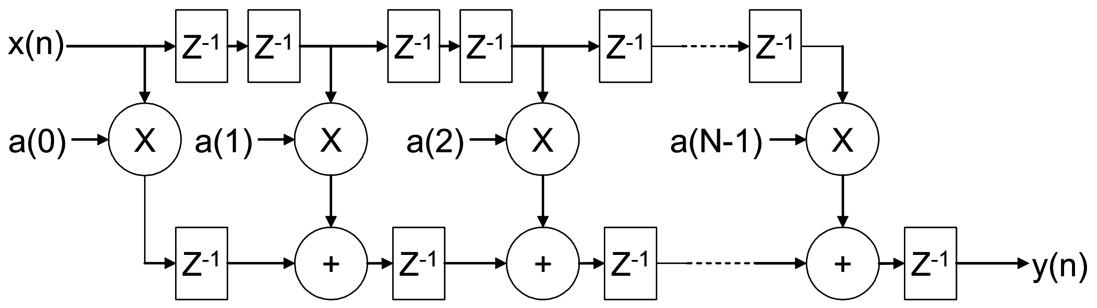


Figure 3-9: **Pipelined Direct - Form**

Figure 3-10 shows a multi-MAC implementation for this architecture.

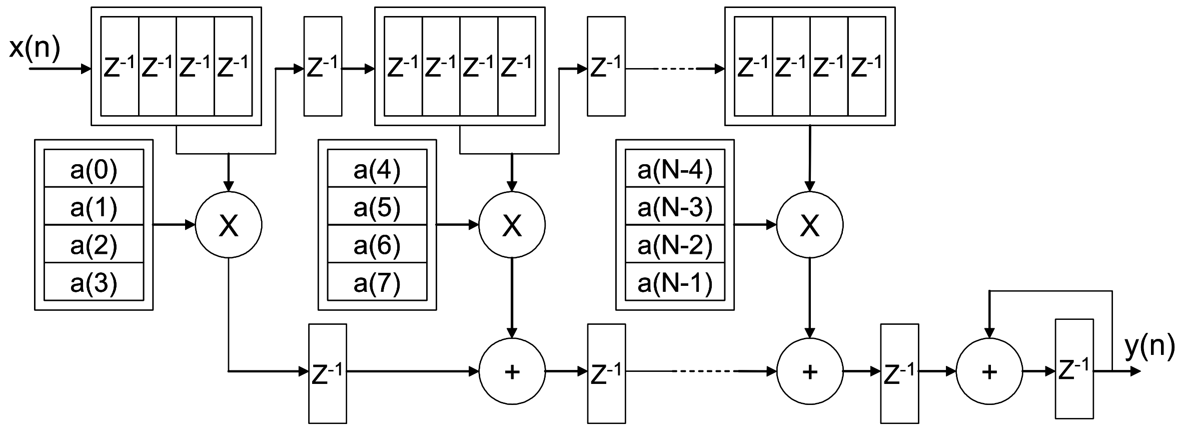


Figure 3-10: **Systolic Multi - MAC Implementation**

The architecture is directly supported by the DSP 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 3-11 shows the Transpose Multiply-Accumulate architecture implementing a Transposed Direct-Form filter.

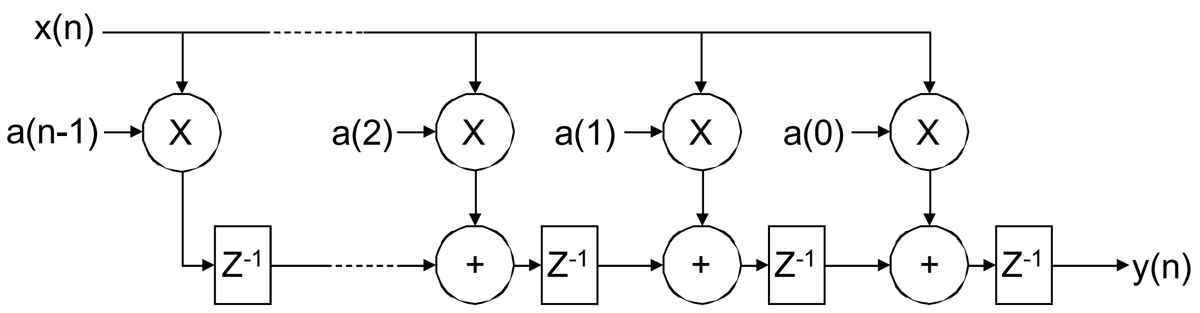


Figure 3-11: **Transpose Direct - Form**

Figure 3-12 shows a multi-MAC implementation for this architecture.

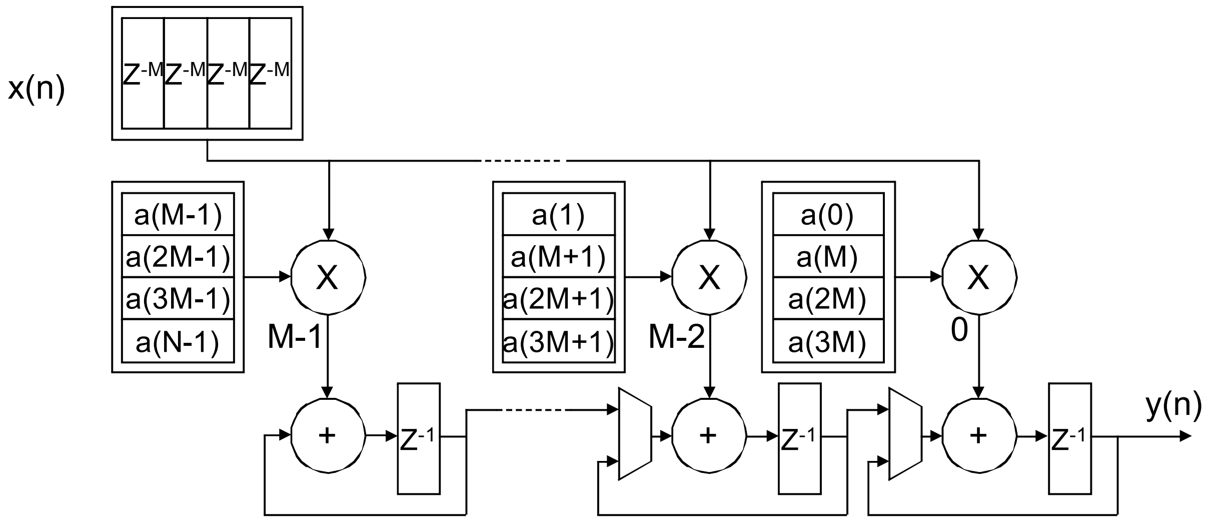


Figure 3-12: **Transpose Multi - MAC Implementation**

This architecture is also directly supported by the DSP 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

This section describes the filters and how to optimize their use in the FIR Compiler. The topics covered include:

- Filter Symmetry
- Single-rate FIR Filter
- Half-band FIR Filter
- Hilbert Transform
- Polyphase Decimator
- Polyphase Interpolator
- Half-band Decimator
- Half-band Interpolator
- Small Non-zero Even Terms in a Half-band Filter Impulse Response
- Fixed Fractional Rate Resampling Filters

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 3-13 shows the impulse response for a 9-tap symmetric FIR filter.

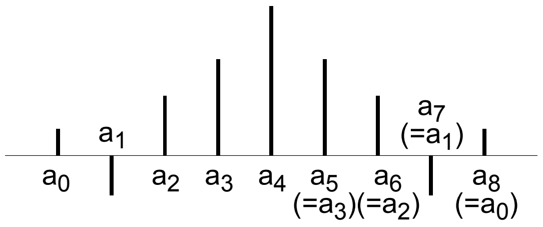


Figure 3-13: **Symmetric FIR – Odd Number of Terms**

Instead of implementing this filter using the architecture shown in Figure 1-1, the more efficient signal flow-graph in Figure 3-14 can be used. In general, the former approach requires *N* multiplications and (*N*-1) additions. In contrast, the architecture in Figure 3-14 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.

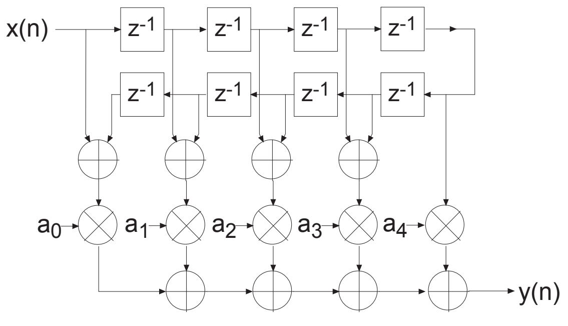


Figure 3-14: **Exploiting Coefficient Symmetry – Odd Number of Filter Taps**

Coefficient symmetry for an even number of terms can be exploited as shown in Figure 3-15.

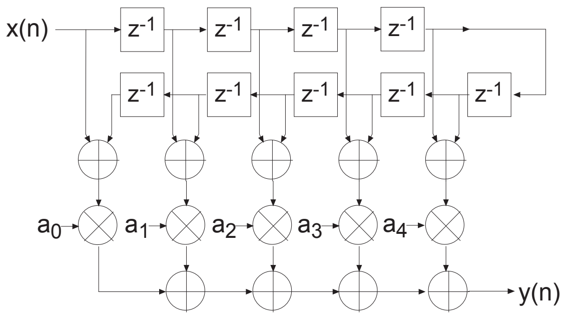


Figure 3-15: **Exploiting Coefficient Symmetry – Even Number of Filter Taps**

Figure 3-16 shows the impulse response for a negative, or odd, symmetric filter.

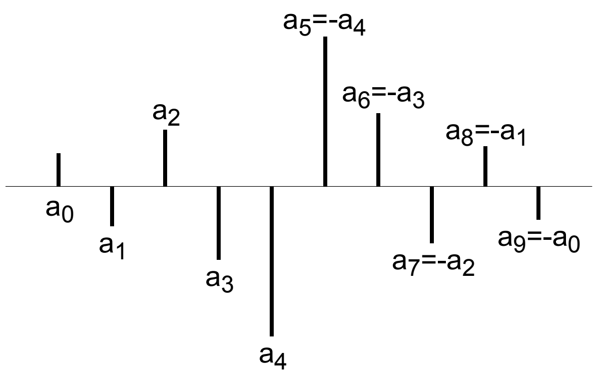


Figure 3-16: **Negative Symmetric Impulse Response**

This symmetry is exploited in a manner similar to that shown in Figure 3-14 and Figure 3-15. In this case, the middle layer of adders are replaced by subtracters, as shown in Figure 3-17.

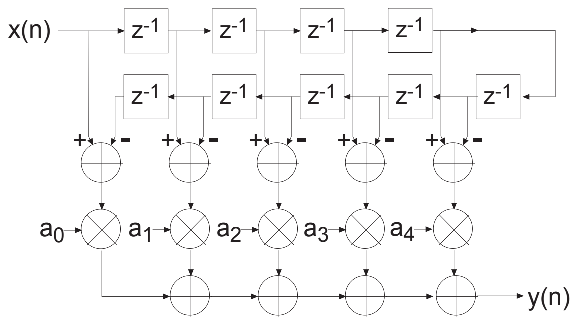


Figure 3-17: **FIR Architecture Exploiting Negative Symmetry**

Filter coefficient symmetry is inferred by the core GUI from the coefficient definition file. You can override this inferred value. 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, you 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 shown in Figure 3-18.

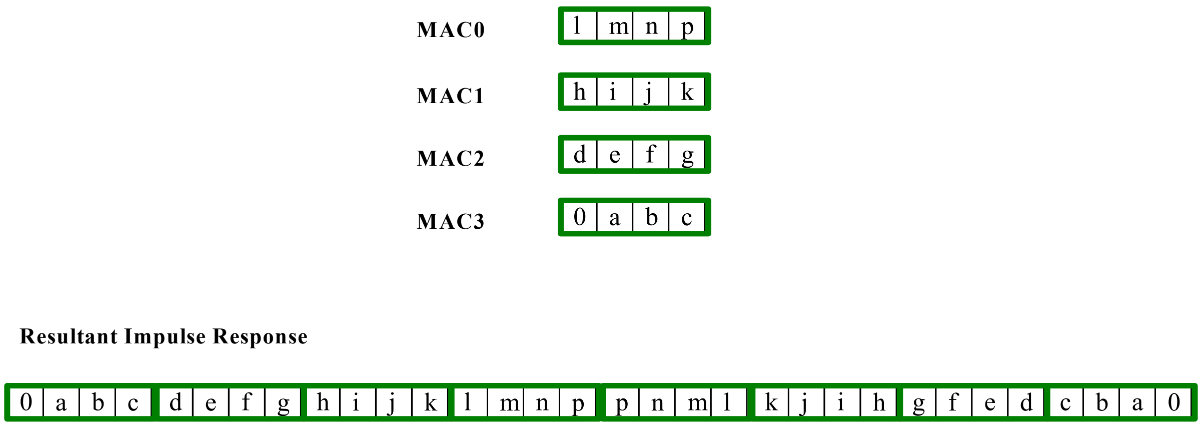

MAC0
l m n p
MAC1
h i j k
MAC2
d e f g
MAC3
0 a b c
Resultant Impulse Response
0 a b c d e f g h i j k l m n p p n m l k j i h g f e d c b a 0

Figure 3-18: **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, you 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, you 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 Vivado IDE displays the actual number of coefficients calculated on the Implementation Details tab. You can use this information to determine if you can increase the number of coefficients used by your 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 Vivado IDE.

Half-band FIR Filter

Figure 3-19 shows the general frequency response for a half-band filter.

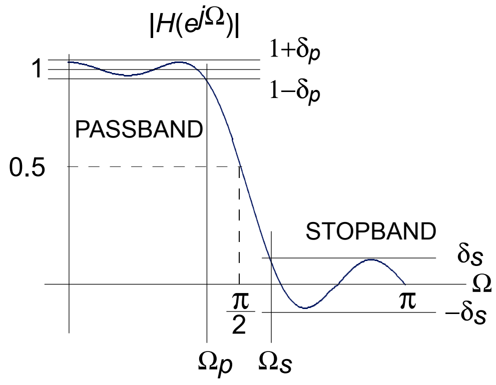


Figure 3-19: **Half-band Filter Magnitude Frequency Response**

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 3] that approximately half of the filter coefficients are zero for an odd number of taps, as shown in Figure 3-20 for an 11-tap half-band filter.

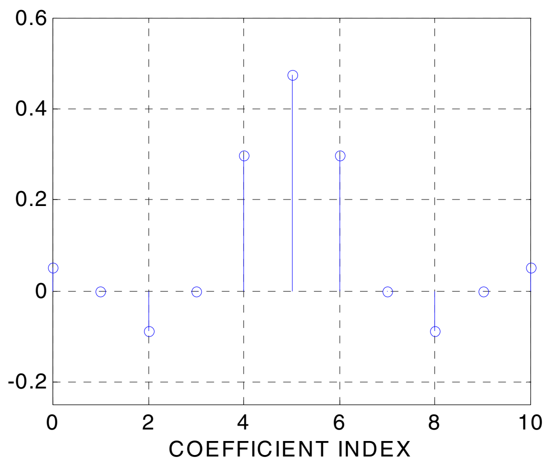


Figure 3-20: **Half-band Filter Impulse Response**

The interleaved zero values in the coefficient data can be exploited to realize an efficient realization, as shown in Figure 3-21.

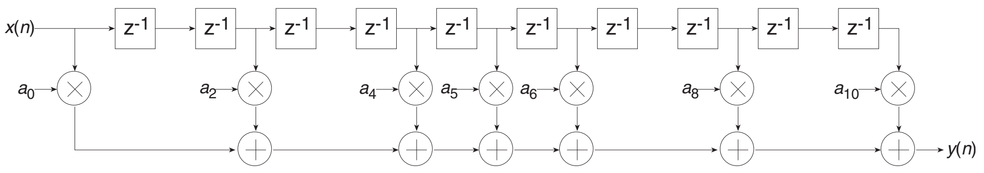


Figure 3-21: **Half-band Filter Impulse Response**

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.

IMPORTANT: You 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 Filter Coefficient Data.

Hilbert Transform

Hilbert transformers [Ref 3] are used in several ways in digital communication systems. An ideal Hilbert transform provides a phase shift of 90 degrees for positive frequencies and -90 degrees for negative frequencies. It can be shown [Ref 3] that the impulse response corresponding to this frequency domain characteristic is odd-symmetric and has interleaved zeros as shown in Figure 3-21. Both the alternating zero-valued coefficients and the negative symmetry can be utilized to produce an efficient hardware realization.

A Hilbert transformer accepts a real-valued signal and produces a complex (I,Q) output signal. The quadrature (Q) component of the output signal is produced by a FIR filter with an impulse response like that shown in Figure 3-22. The in-phase (I) component is the input signal delayed by an appropriate amount to compensate for the phase delay of the FIR process employed for generating the Q output. This is efficiently achieved by accessing the center tap of the sample history delay of the Q channel FIR filter as shown in Figure 3-23. In this figure, x(n) is the real-valued input signal, and $y_I(n)$ and $y_Q(n)$ are the in-phase and quadrature outputs, respectively.

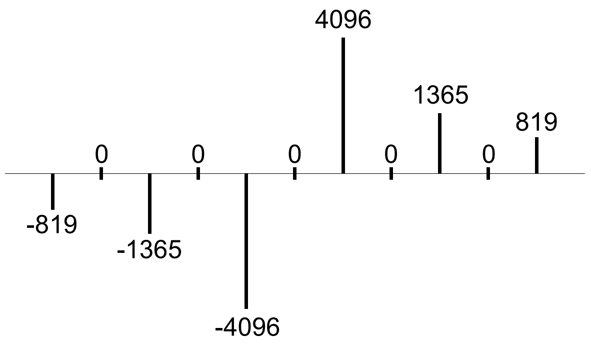


Figure 3-22: **Hilbert Transformer Impulse Response**

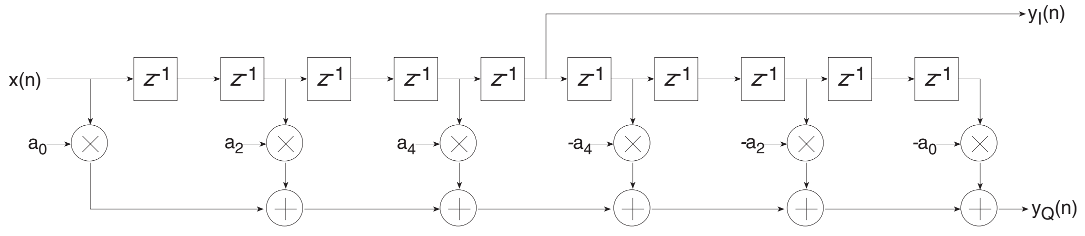


Figure 3-23: **Hilbert Transformer FIR Filter Realization**

Figure 3-24 shows the architecture for a Hilbert transformer that exploits both the zero-valued and the negative symmetry characteristics of the impulse response.

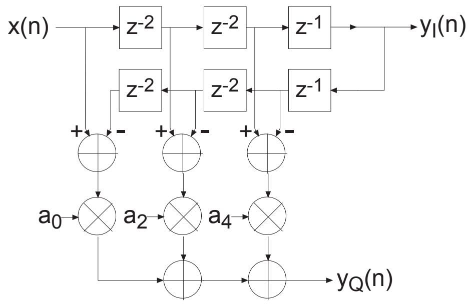


Figure 3-24: **Hilbert Transformer Exploiting Zero-valued Filter Coefficients and Negative Symmetry**

Interpolated FIR Filter

An *interpolated FIR* (IFIR) filter [Ref 2] has a similar architecture to a conventional FIR filter, but with the unit delay operator replaced by *k*-1 units of delay. *k* is referred to as the zero-packing factor. Figure 3-25 shows a *N*-tap IFIR filter. This architecture is functionally equivalent to inserting *k*-1 zeros between the coefficients of a prototype filter coefficient set.

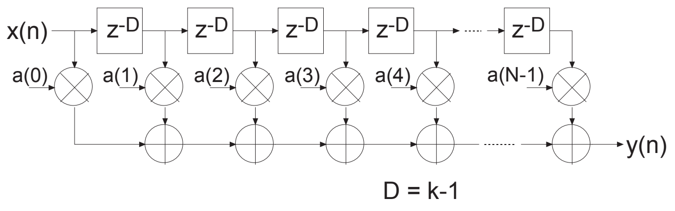


Figure 3-25: **Interpolated FIR (IFIR) - Zero-packing Factor is *k***

Interpolated filters are useful for realizing efficient implementations of both narrow-band and wide-band filters. A filter system based on an IFIR approach requires not only the IFIR but also an image rejection filter. References [Ref 2] and [Ref 6] provide the details of how these systems are realized, and how to design the IFIR and the image rejection filters.

The IFIR filter implementation takes advantage of the *k*-1 zeros in the impulse response to realize an area-efficient FPGA implementation. The FPGA area required by an IFIR filter is not a strong function of the zero-packing factor.

The interpolated FIR should not be confused with an interpolation filter. Interpolated filters are single-rate systems employed to produce efficient realizations of narrow-band filters and, with some minor enhancements, wide-band filters can be accommodated. There is no inherent rate change when using an interpolated filter – the input rate is the same as the output rate.

Polyphase Decimator

Figure 3-26 shows the polyphase decimation filter option which implements the computationally efficient *M*-to-1 polyphase decimating filter.

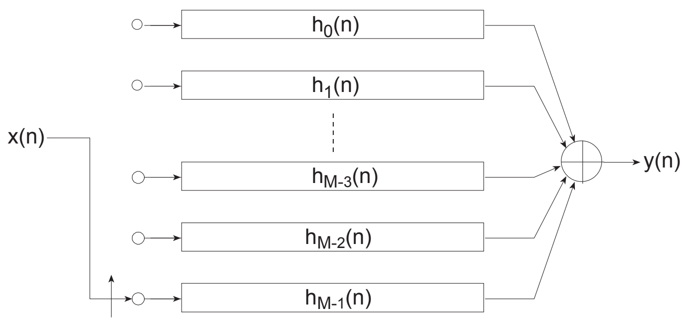


Figure 3-26: ***M*-to-1 Polyphase Decimating Filter**

A set of *N* prototype filter coefficients $a_0, a_1, \ldots, a_{N-1}$ is mapped to the *M* polyphase subfilters $h_0(n), h_1(n), \ldots, h_{M-1}(n)$ according to Equation 3-1.

$$h_i(r) = a(i + Mr) \qquad i = 0, 1, \ldots, M-1 \qquad r = 0, 1, \ldots, \frac{N}{M}$$ *Equation 3-1*

The polyphase segments are accessed by delivering the input samples *x*(*n*) to their inputs using 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 $(n), n = 0, 1, 2, \ldots$.

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 3-27 shows the polyphase interpolation filter option which implements the computationally efficient 1-to-*P* interpolation filter.

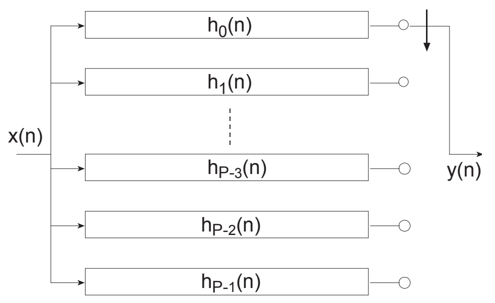


Figure 3-27: **1-to-*P* Polyphase Interpolator**

A set of *N* prototype filter coefficients $a_0, a_1, \ldots, a_{N-1}$ is mapped to the P polyphase subfilters $h_0(n), h_1(n), \ldots, h_{p-1}(n)$ according to Equation 3-1, 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 3-27.

The output sample f_s' rate is $f_s' = f_s P$ where f_s is the sample rate of the input data stream (n), $n = 0, 1, 2, \ldots$ 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 7] 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 3-28 shows the resulting structure.

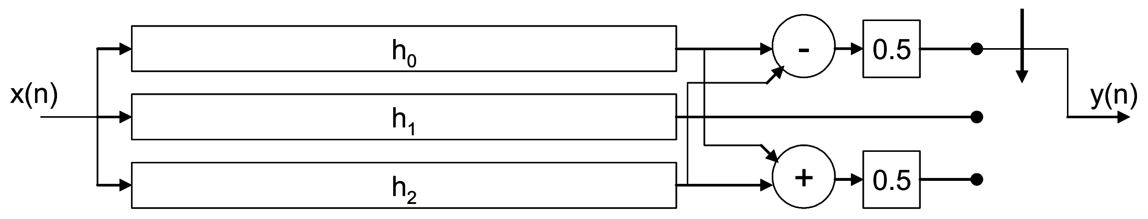


Figure 3-28: **Symmetric Pairs**

Note: For some configurations an extra DSP Slice is required to implement the recombination of the phases.

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 several 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 your application system, are also similar to the general symmetry case; you can either force non-symmetric structure implementation or make use of the extra coefficients which can be supported in the structure. Figure 3-29 shows several example cases in and is extensible to larger filters.

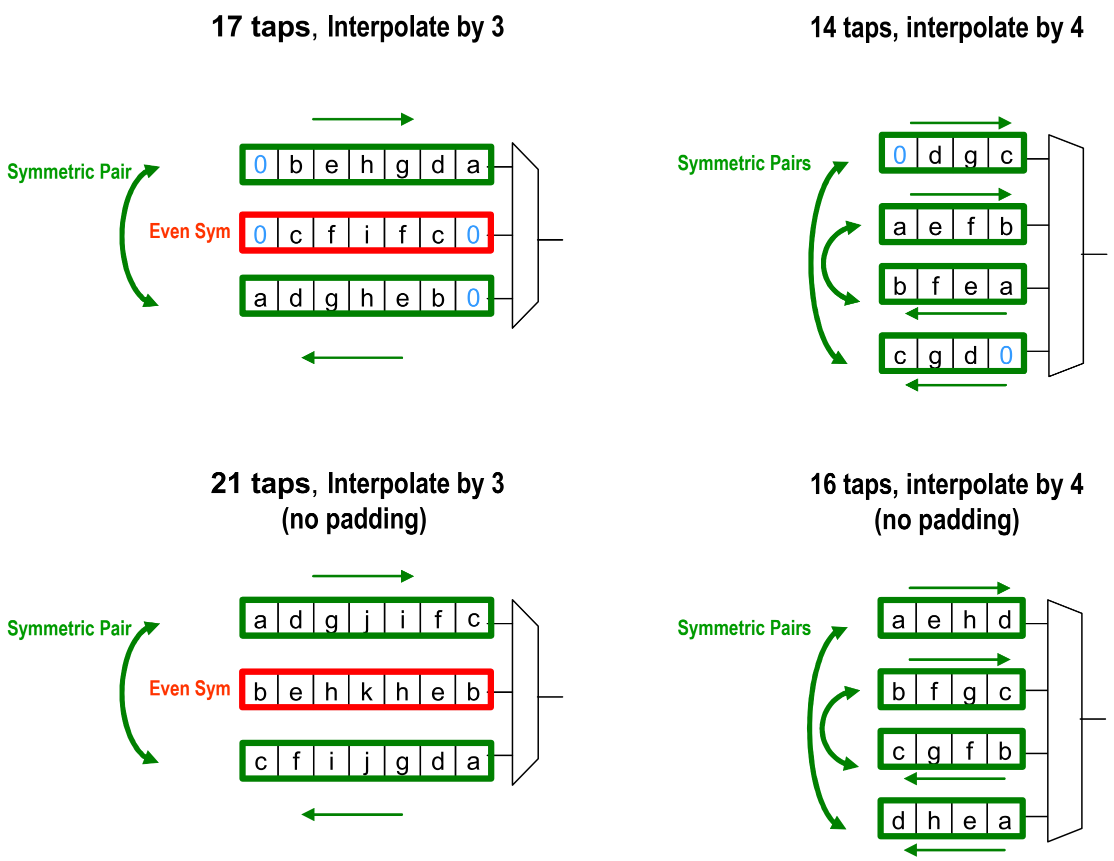


Figure 3-29: **Filter Padding to Facilitate Symmetric Pairing**

Half-band Decimator

The half-band decimator is a polyphase filter with an embedded 2-to-1 down-sampling of the input signal. Figure 3-30 shows the structure.

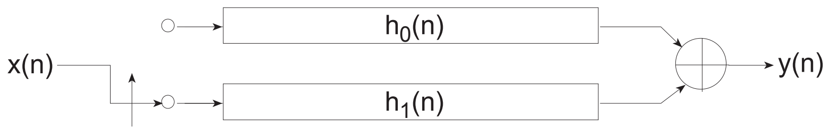


Figure 3-30: **Half-band Decimation Filter**

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 3-19 and Figure 3-20, respectively. Observe the alternating zero-valued coefficients in the impulse response. Figure 3-30 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 3-30. Figure 3-31

(a) is the filter impulse response ($a_1 = 0 = a_5$). Figure 3-31 (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 3-31 (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.

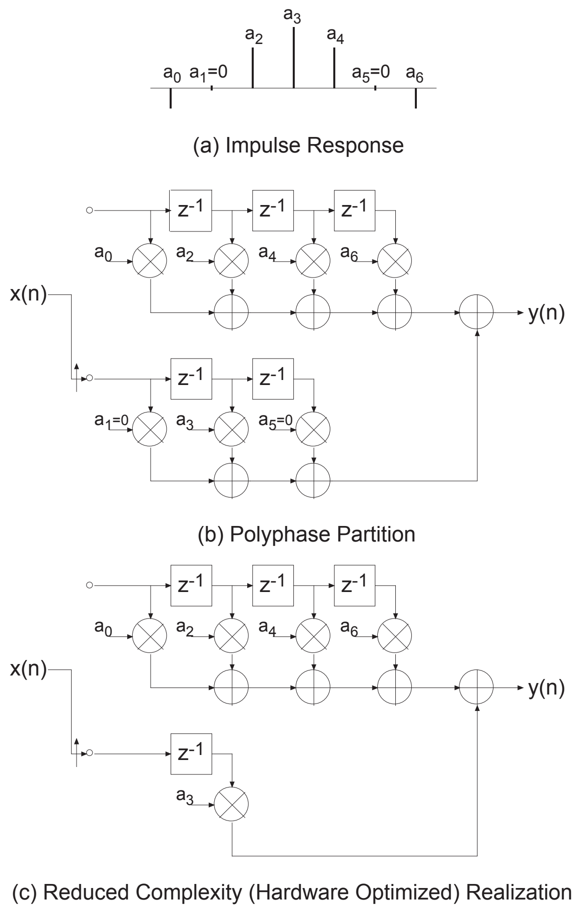


Figure 3-31: **7-Tap Half-band Decimation Filter**

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 3-32 shows the half-band interpolator.

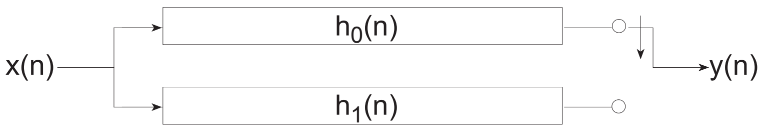


Figure 3-32: **Half-band Interpolation Filter**

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 3-33. Figure 3-33(a) is the impulse response, Figure 3-33(b) shows the polyphase partition, and Figure 3-33(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.

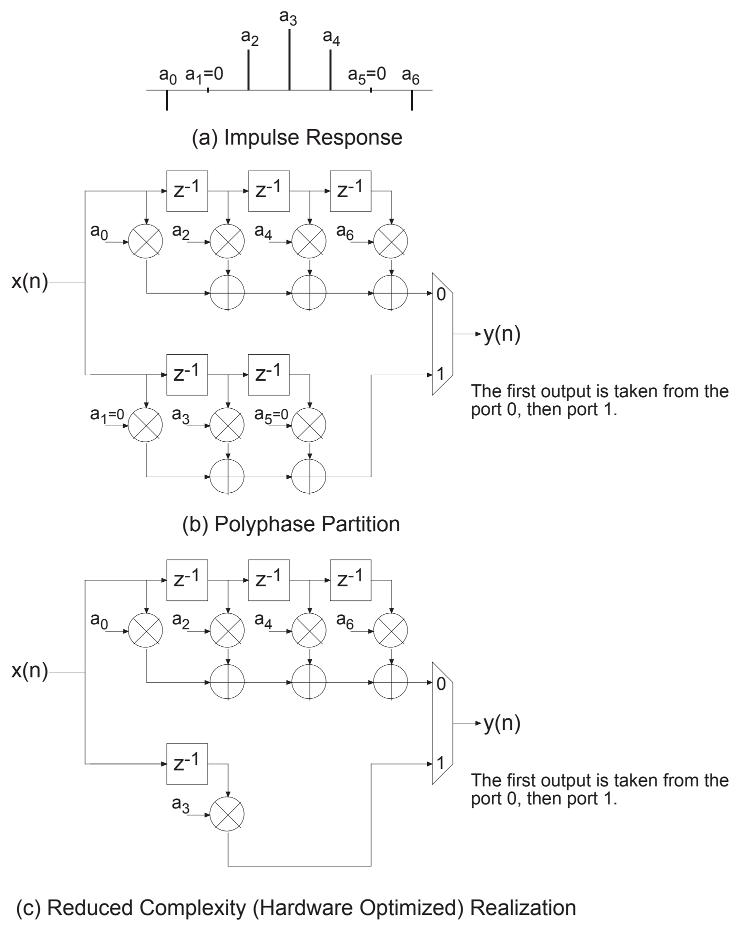

a_0 a_1=0 a_2 a_3 a_4 a_5=0 a_6
(a) Impulse Response
x(n)
z^{-1}
a_0 a_2 a_4 a_6
a_1=0 a_3 a_5=0
0
1
y(n)
The first output is taken from the port 0, then port 1.
(b) Polyphase Partition
a_3
(c) Reduced Complexity (Hardware Optimized) Realization

Figure 3-33: **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 3-34, 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.

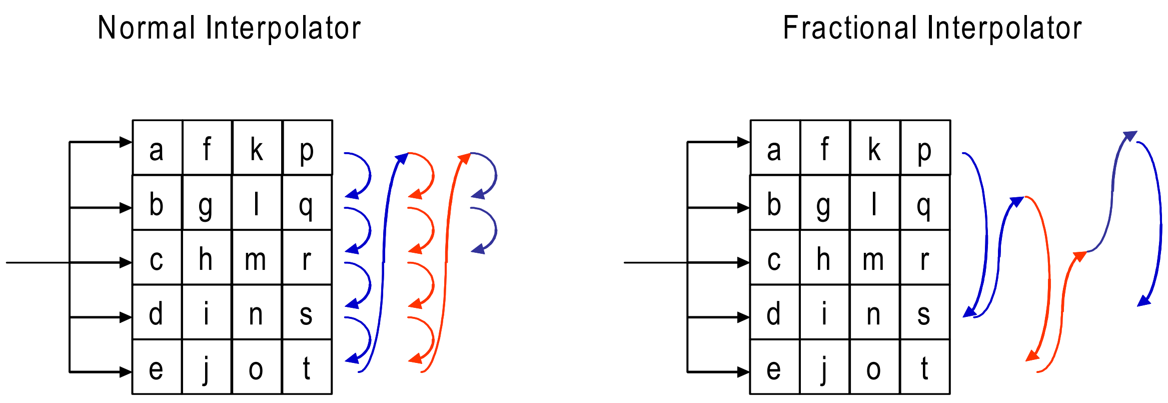


Figure 3-34: **Interpolation Filters for Integer and Fractional Rates**

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 3-35 shows 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 shown on the right is P/Q=3/5.

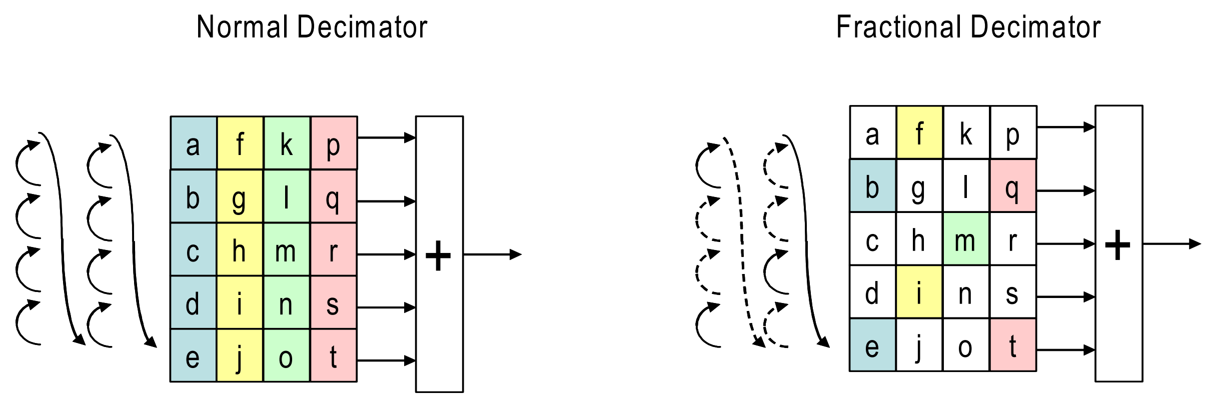


Figure 3-35: **Decimation Filters for Integer and Fractional Rates**

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

- Single-rate FIR
- Half-band Filter
- Hilbert Transform
- Interpolated Filter
- Multiple Coefficient Sets
- Coefficient Specification Using Non-integer Real Numbers

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 3-36 for an *N*-tap filter.

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

Figure 3-36: **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 you enter 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 3-37.

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

Figure 3-37: **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 3-38:

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

Figure 3-38: **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 3-39 is presented in Figure 3-40.

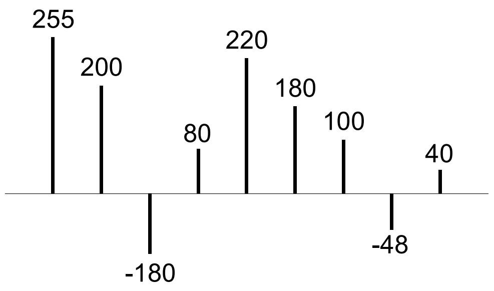


Figure 3-39: **Non-symmetric Impulse Response**

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

Figure 3-40: **Coefficient File for the Non-symmetric Impulse Response**

The coefficient file for the negative-symmetric filter characterized by the impulse response in Figure 3-41 is shown in Figure 3-42.

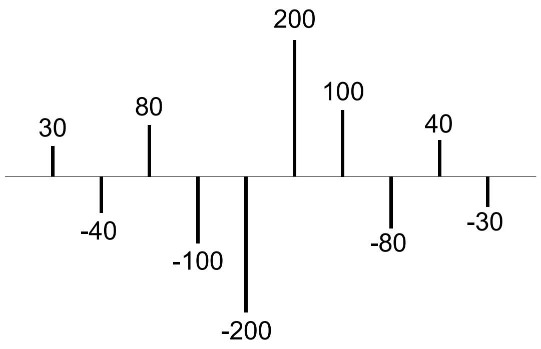


Figure 3-41: **Negative Symmetric Impulse Response**

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

Figure 3-42: **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 3-43 is shown in Figure 3-44.

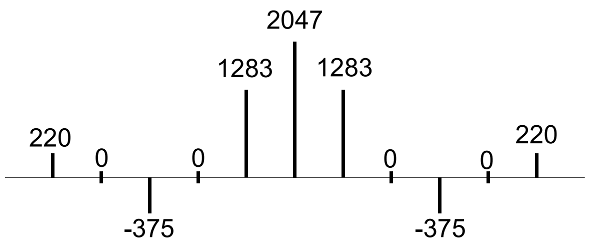


Figure 3-43: **11-Tap Half-band Filter Impulse Response**

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

Figure 3-44: **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 3-45. However, observe that these values do not contribute to the result.

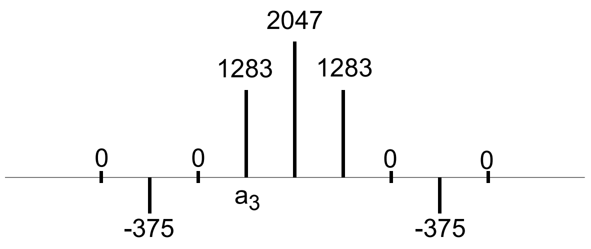


Figure 3-45: **9-Tap Half-band Filter Impulse Response**

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.

Hilbert Transform

The impulse response for a 10-term approximation to a Hilbert transformer is shown in Figure 3-46. The odd-symmetry and zero-valued coefficients are both exploited to generate an efficient FPGA realization. The coefficient data file for the Hilbert transform must contain the zero-valued entries. For example, the .coe file corresponding to Figure 3-46 is shown in Figure 3-47.

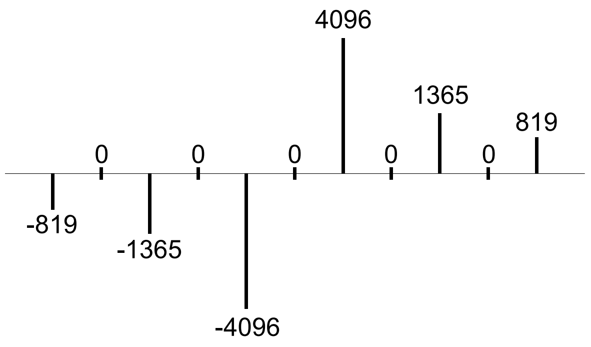


Figure 3-46: **Hilbert Transform Impulse Response**

```
radix=10;
coefdata=-819,0,-1365,0,-4096,0,4096,0,1365,0,819;
```

Figure 3-47: **Coefficient File for the Hilbert Transformer Impulse Response**

In practice, some optimization methods used for designing a Hilbert transform can lead to the presence of small even-numbered coefficients. If the *Hilbert Transform* filter class is used in the FIR Compiler, you must force these terms to zero.

Just like the half-band filter, the zero-valued entries for a Hilbert transformer can occur at the filter impulse response extremities. However, these values do not contribute to the result.

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 Hilbert transformer must obey N=3 + 4n, where n = 0, 1, 2, 3,.... For example, a Hilbert transform filter can have 11, 15, 19, and 23 taps, but not 9, 13, 17, or 21 taps.

Interpolated Filter

A previous section explained that an IFIR filter is similar to a conventional FIR, but with the unit delay operator replaced by *k*-1 units of delay. *k* is referred to as the *zero-packing factor*. One way to realize this substitution is by the insertion of *k*-1 zeros between the coefficient values of a prototype filter. When specifying an IFIR architecture, the full set of prototype coefficients is supplied in the coefficient file, without the zeros implied by the zero-packing factor. The zero-packing factor is defined through the filter user interface. For example, consider the filter coefficient data in the .coe file shown in Figure 3-48.

```
radix=10;
coefdata=-200,1200,2047,1200,-200;
```

Figure 3-48: **Prototype Coefficient Data for IFIR Example**

If a zero-packing factor of *k*=2 is specified, the equivalent filter impulse response is shown in Figure 3-49.

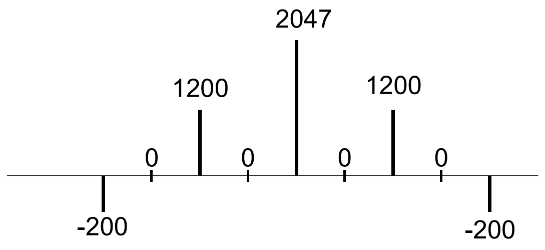


Figure 3-49: **Equivalent IFIR Impulse Response for the Coefficient Data Shown in Figure 3-48 with a Zero-packing Factor *k*=2**

If the zero-packing factor is changed to *k*=3, the impulse response is as shown in Figure 3-50.

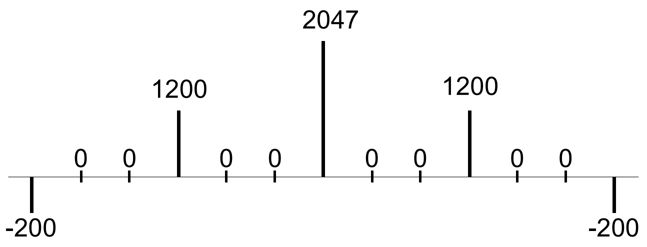


Figure 3-50: **Equivalent IFIR Impulse Response for the Coefficient Data Shown in Figure 3-48 with a Zero-packing Factor *k*=3**

These examples use a symmetrical prototype impulse response; this is not a restriction of the filter core. The prototype filter coefficient set can be symmetrical, non-symmetrical, or negative-symmetric.

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, you 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 your specified coefficient bit width. This allows you 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 Coefficient Quantization 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 project directory. The MIF file is always in binary format.

Interleaved Data Channel Filters

- Basic
- Advanced

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 3-51. Here a complex base-band signal $(n) = x_I(n) + jx_Q(n)$ is applied to a matched filter M(z). The in-phase and quadrature components are processed by the same filter.

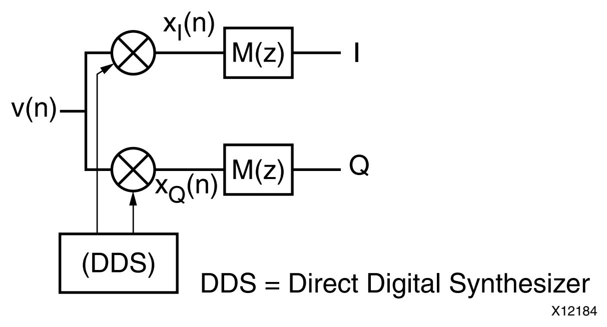


Figure 3-51: **Digital Down Converter**

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 Customize IP dialog box. The FIR Compiler supports two multichannel implementation: Basic and Advanced.

Basic

The basic implementation processes interleaved data channels sequentially; channel 0, channel 1, channel 2, ..., channel N-1, where N = Number of Channels. This implementation uses minimal resources.

Advanced

The advanced implementation provides a list of predefined interleaved data channel sequences, or patterns, from which multiple patterns can be selected during core customization. The specified patterns can then be selected during core operation using the CONFIG Channel.

When the core is configured to support one channel with a sample frequency of f_S the same hardware resources (DSP Slice and Memory) can support two channels with a sample frequency of $f_S/2$, 4 channels with a sample frequency of $f_S/4$ or 1 channel with a sample frequency $f_S/2$ and 2 channels with a sample frequency $f_S/4$. The Advanced implementation supports each of these configurations with an associated interleaved channel sequence that can then be selected, dynamically, during core operation through the CONFIG Channel.

Table 3-1 list all the supported interleaved channel patterns. The full pattern list is also displayed on the GUI.

Although the hardware resources (DSP Slice and Memory) remain the same as the equivalent Basic implementation the Advanced Implementation requires additional logic resources. For the patterns highlighted in Table 3-1 the memory requirements might also increase and further logic resources might be required.

Table 3-1: **Advanced Interleaved Data Channel Patterns**

No. Chans.	Seq. ID	Description	Interleaved Channel Pattern
4	P4-0	1 Channel at fs	0 0 0 0
4	P4-1	2 Channels at 1/2fs	0 1 0 1
4	P4-2	1 Channel at 3/4fs, 1 Channel at 1/4fs	0 0 0 1
4	P4-3	1 Channel at 1/2fs, 2 Channels at 1/4fs	0 1 0 2
4	P4-4	4 Channels at 1/4fs	0 1 2 3
6	P6-0	1 Channel at fs	0 0 0 0 0 0
6	P6-1	2 Channels at 1/2fs	0 1 0 1 0 1
6	P6-2	1 Channel at 2/3fs, 1 Channel at 1/3fs	0 0 0 0 1 1
6	P6-3	3 Channels at 1/3fs	0 1 2 0 1 2
6	P6-4	1 Channel at 2/3fs, 2 Channels at 1/6fs	0 0 0 0 1 2
6	P6-5	1 Channel at 1/2fs, 3 Channels at 1/6fs	0 1 0 2 0 3
6	P6-6	2 Channels at 1/3fs, 2 Channels at 1/6fs	0 1 2 0 1 3
6	P6-7	1 Channel at 1/3fs, 3 Channels at 1/6fs	0 1 2 0 3 4
6	P6-8	6 Channels at 1/6fs	0 1 2 3 4 5
8	P8-0	1 Channel at fs	0 0 0 0 0 0 0 0
8	P8-1	1 Channel at 3/4fs, 1 Channel at 1/4fs	0 0 0 0 0 0 1 1
8	P8-2	2 Channels at 1/2fs	0 1 0 1 0 1 0 1
8	P8-3	1 Channel at 3/4fs, 2 Channels at 1/8fs	0 0 0 0 0 0 1 2

Table 3-1: **Advanced Interleaved Data Channel Patterns *(Cont'd)***

No. Chans.	Seq. ID	Description	Interleaved Channel Pattern
8	P8-4	1 Channel at 1/2fs, 2 Channels at 1/4fs	0 1 0 2 0 1 0 2
8	P8-5	4 Channels at 1/4fs	0 1 2 3 0 1 2 3
8	P8-6	1 Channel at 1/2fs, 1 Channel at 1/4fs, 2 Channels at 1/8fs	0 1 0 2 0 1 0 3
8	P8-7	2 Channels at 3/8fs, 2 Channels at 1/8fs	0 1 0 1 0 1 2 3
8	P8-8	1 Channel at 1/2fs, 4 Channels at 1/8fs	0 1 0 2 0 3 0 4
8	P8-9	3 Channels at 1/4fs, 2 Channels at 1/8fs	0 1 2 3 0 1 2 4
8	P8-10	2 Channels at 1/4fs, 4 Channels at 1/8fs	0 2 1 3 0 4 1 5
8	P8-11	1 Channel at 1/4fs, 6 Channels at 1/8fs	0 1 2 3 0 4 5 6
8	P8-12	8 Channels at 1/8fs	0 1 2 3 4 5 6 7
12	P12-0	1 Channel at fs	0 0 0 0 0 0 0 0 0 0 0 0
12	P12-1	2 Channels at 1/2fs	0 1 0 1 0 1 0 1 0 1 0 1
12	P12-2	1 Channel at 2/3fs, 1 Channel at 1/3fs	0 0 0 0 0 0 0 0 1 1 1 1
12	P12-3	1 Channel at 3/4fs, 1 Channel at 1/4fs	0 0 0 0 0 0 0 0 0 1 1 1
12	P12-4	1 Channel at 2/3fs, 2 Channels at 1/6fs	0 0 0 0 0 0 0 0 1 2 1 2
12	P12-5	1 Channel at 1/2fs, 2 Channels at 1/4fs	0 1 0 2 0 1 0 2 0 1 0 2
12	P12-6	3 Channels at 1/3fs	0 1 2 0 1 2 0 1 2 0 1 2
12	P12-7	1 Channel at 1/2fs, 3 Channels at 1/6fs	0 1 0 2 0 3 0 1 0 2 0 3
12	P12-8	2 Channels at 1/3fs, 2 Channels at 1/6fs	0 1 2 0 1 3 0 1 2 0 1 3
12	P12-9	4 Channels at 1/4fs	0 1 2 3 0 1 2 3 0 1 2 3
12	P12-10	1 Channel at 2/3fs, 4 Channels at 1/12fs	0 0 0 0 0 0 0 0 1 2 3 4
12	P12-11	1 Channel at 1/2fs, 2 Channels at 1/6fs, 2 Channels at 1/12fs	0 1 0 2 0 3 0 1 0 2 0 4
12	P12-12	2 Channels at 1/3fs, 1 Channel at 1/6fs, 2 Channels at 1/12fs	0 1 2 0 1 3 0 1 2 0 1 4

Table 3-1: **Advanced Interleaved Data Channel Patterns *(Cont'd)***

No. Chans.	Seq. ID	Description	Interleaved Channel Pattern
12	P12-13	1 Channel at 1/3fs, 4 Channels at 1/6fs	0 1 2 0 3 4 0 1 2 0 3 4
12	P12-14	2 Channels at 1/3fs, 4 Channels at 1/12fs	0 1 2 0 1 3 0 1 4 0 1 5
12	P12-15	3 Channels at 1/4fs, 3 Channels at 1/12fs	0 1 2 3 0 1 2 4 0 1 2 5
12	P12-16	1 Channel at 1/3fs, 3 Channels at 1/6fs, 2 Channels at 1/12fs	0 1 3 0 2 4 0 1 3 0 2 5
12	P12-17	6 Channels at 1/6fs	0 1 2 3 4 5 0 1 2 3 4 5
12	P12-18	1 Channel at 1/2fs, 6 Channels at 1/12fs	0 1 0 2 0 3 0 4 0 5 0 6
12	P12-19	1 Channel at 1/3fs, 2 Channels at 1/6fs, 4 Channels at 1/12fs	0 1 3 0 2 4 0 1 5 0 2 6
12	P12-20	5 Channels at 1/6fs, 2 Channels at 1/12fs	0 1 2 3 4 5 0 1 2 3 4 6
12	P12-21	1 Channel at 1/3fs, 1 Channel at 1/6fs, 6 Channels at 1/12fs	0 1 2 0 3 4 0 1 5 0 6 7
12	P12-22	2 Channels at 1/4fs, 6 Channels at 1/12fs	0 1 2 3 0 1 4 5 0 1 6 7
12	P12-23	4 Channels at 1/6fs, 4 Channels at 1/12fs	0 1 2 3 4 5 0 1 2 3 6 7
12	P12-24	3 Channels at 1/6fs, 6 Channels at 1/12fs	0 1 2 3 4 5 0 1 2 6 7 8
12	P12-25	2 Channels at 1/6fs, 8 Channels at 1/12fs	0 1 2 3 4 5 0 1 6 7 8 9
12	P12-26	12 Channel at 1/12fs	0 1 2 3 4 5 6 7 8 9 10 11
16	P16-0	1 Channel at fs	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
16	P16-1	1 Channel at 3/4fs, 1 Channel at 1/4fs	0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
16	P16-2	2 Channels at 1/2fs	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
16	P16-3	1 Channel at 3/4fs, 2 Channels at 1/8fs	0 0 0 0 0 0 0 0 0 0 0 0 1 2 1 2
16	P16-4	1 Channel at 1/2fs, 2 Channels at 1/4fs	0 1 0 2 0 1 0 2 0 1 0 2 0 1 0 2
16	P16-5	4 Channels at 1/4fs	0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
16	P16-6	1 Channel at 1/2fs, 1 Channel at 1/4fs, 2 Channels at 1/8fs	0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 3

Table 3-1: **Advanced Interleaved Data Channel Patterns *(Cont'd)***

No. Chans.	Seq. ID	Description	Interleaved Channel Pattern
16	P16-7	2 Channels at 3/8fs, 2 Channels at 1/8fs	0 1 0 1 0 1 0 1 0 1 0 1 2 3 2 3
16	P16-8	1 Channel at 1/2fs, 4 Channels at 1/8fs	0 1 0 2 0 3 0 4 0 1 0 2 0 3 0 4
16	P16-9	3 Channels at 1/4fs, 2 Channels at 1/8fs	0 1 2 3 0 1 2 4 0 1 2 3 0 1 2 4
16	P16-10	2 Channels at 1/4fs, 4 Channels at 1/8fs	0 2 1 3 0 4 1 5 0 2 1 3 0 4 1 5
16	P16-11	1 Channel at 1/4fs, 6 Channels at 1/8fs	0 1 2 3 0 4 5 6 0 1 2 3 0 4 5 6
16	P16-12	8 Channels at 1/8fs	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
16	P16-13	2 Channels at 3/8fs, 4 Channels at 1/16fs	0 1 0 1 0 1 0 1 0 1 0 1 2 3 4 5
16	P16-14	2 Channels at 1/4fs, 4 Channels at 1/8fs	0 2 1 3 0 4 1 5 0 2 1 3 0 4 1 5
16	P16-15	2 Channels at 1/4fs, 2 Channels at 1/8fs, 4 Channels at 1/16fs	0 2 1 4 0 3 1 5 0 2 1 6 0 3 1 7
16	P16-16	4 Channels at 3/16fs, 4 Channels at 1/16fs	0 1 2 3 0 1 2 3 0 1 2 3 4 5 6 7
16	P16-17	2 Channels at 1/4fs, 8 Channels at 1/16fs	0 2 1 3 0 4 1 5 0 6 1 7 0 8 1 9
16	P16-18	6 Channels at 1/8fs, 4 Channels at 1/16fs	0 1 2 6 3 4 5 7 0 1 2 8 3 4 5 9
16	P16-19	4 Channels at 1/8fs, 8 Channels at 1/16fs	0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11
16	P16-20	2 Channels at 1/8fs, 12 Channel at 1/16fs	0 2 3 4 1 5 6 7 0 8 9 10 1 11 12 13
16	P16-21	16 Channel at 1/16fs	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
24	P24-0	2 Channels at 1/2fs	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
24	P24-1	4 Channels at 1/4fs	0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
24	P24-2	2 Channels at 1/3fs, 2 Channels at 1/6fs	0 1 2 0 1 3 0 1 2 0 1 3 0 1 2 0 1 3 0 1 2 0 1 3
24	P24-3	2 Channels at 3/8fs, 2 Channels at 1/4fs	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 2 3 2 3 2 3
24	P24-4	2 Channels at 1/3fs, 4 Channels at 1/12fs	0 1 2 0 1 3 0 1 4 0 1 5 0 1 2 0 1 3 0 1 4 0 1 5
24	P24-5	2 Channels at 1/4fs, 4 Channels at 1/8fs	0 2 1 3 0 4 1 5 0 2 1 3 0 4 1 5 0 2 1 3 0 4 1 5
24	P24-6	6 Channels at 1/6fs	0 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5

Table 3-1: **Advanced Interleaved Data Channel Patterns *(Cont'd)***

No. Chans.	Seq. ID	Description	Interleaved Channel Pattern
24	P24-7	2 Channels at 1/4fs, 6 Channels at 1/12fs	0 2 1 3 0 4 1 5 0 6 1 7 0 2 1 3 0 4 1 5 0 6 1 7
24	P24-8	4 Channels at 1/6fs, 4 Channels at 1/12fs	0 1 2 3 4 5 0 1 2 3 6 7 0 1 2 3 4 5 0 1 2 3 6 7
24	P24-9	8 Channels at 1/8fs	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
24	P24-10	2 Channels at 1/3fs, 8 Channels at 1/24fs	0 1 2 0 1 3 0 1 4 0 1 5 0 1 6 0 1 7 0 1 8 0 1 9
24	P24-11	2 Channels at 1/4fs, 4 Channels at 1/12fs, 4 Channels at 1/24fs	0 2 1 3 0 4 1 5 0 6 1 7 0 2 1 3 0 4 1 5 0 8 1 9
24	P24-12	4 Channels at 1/6fs, 2 Channels at 1/12fs, 4 Channels at 1/24fs	0 1 2 3 4 6 0 1 2 3 5 7 0 1 2 3 4 8 0 1 2 3 5 9
24	P24-13	2 Channels at 1/6fs, 8 Channels at 1/12fs	0 2 3 1 4 5 0 6 7 1 8 9 0 2 3 1 4 5 0 6 7 1 8 9
24	P24-14	4 Channels at 1/6fs, 8 Channels at 1/24fs	0 1 2 3 4 5 0 1 2 3 6 7 0 1 2 3 8 9 0 1 2 3 10 11
24	P24-15	6 Channels at 1/8fs, 6 Channels at 1/24fs	0 1 2 3 4 5 6 7 0 1 2 3 4 5 8 9 0 1 2 3 4 5 10 11
24	P24-16	2 Channels at 1/6fs, 6 Channels at 1/12fs, 4 Channels at 1/24fs	0 2 3 1 4 5 0 6 7 1 8 9 0 2 3 1 4 5 0 6 7 1 10 11
24	P24-17	12 Channel at 1/12fs	0 1 2 3 4 5 6 7 8 9 10 11 0 1 2 3 4 5 6 7 8 9 10 11
24	P24-18	2 Channels at 1/4fs, 12 Channel at 1/24fs	0 2 1 3 0 4 1 5 0 6 1 7 0 8 1 9 0 10 1 11 0 12 1 13
24	P24-19	2 Channels at 1/6fs, 4 Channels at 1/12fs, 8 Channels at 1/24fs	0 2 6 1 3 7 0 4 8 1 5 9 0 2 10 1 3 11 0 4 12 1 5 13
24	P24-20	10 Channel at 1/12fs, 4 Channels at 1/24fs	0 1 2 3 4 5 6 7 8 9 10 11 0 1 2 3 4 5 6 7 8 9 12 13
24	P24-21	2 Channels at 1/6fs, 2 Channels at 1/12fs, 12 Channel at 1/24fs	0 2 4 1 5 6 0 3 7 1 8 9 0 2 10 1 11 12 0 3 13 1 14 15
24	P24-22	4 Channels at 1/8fs, 12 Channel at 1/24fs	0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11 0 12 1 13 2 14 3 15
24	P24-23	8 Channels at 1/12fs, 8 Channels at 1/24fs	0 1 2 3 4 5 6 7 8 9 10 11 0 1 2 3 4 5 6 7 12 13 14 15
24	P24-24	6 Channels at 1/12fs, 12 Channel at 1/24fs	0 6 1 7 2 8 3 9 4 10 5 11 0 12 1 13 2 14 3 15 4 16 5 17
24	P24-25	4 Channels at 1/12fs, 16 Channel at 1/24fs	0 4 5 1 6 7 2 8 9 3 10 11 0 12 13 1 14 15 2 16 17 3 18 19
24	P24-26	24 Channels at 1/24fs	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

Table 3-1: **Advanced Interleaved Data Channel Patterns *(Cont'd)***

No. Chans.	Seq. ID	Description	Interleaved Channel Pattern
32	P32-0	2 Channels at 1/2fs	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
32	P32-1	2 Channels at 3/8fs, 2 Channels at 1/8fs	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 2 3 2 3 2 3 2 3
32	P32-2	4 Channels at 1/4fs	0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
32	P32-3	2 Channels at 3/8fs, 4 Channels at 1/16fs	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 2 3 4 5 2 3 4 5
32	P32-4	2 Channels at 1/4fs, 4 Channels at 1/8fs	0 2 1 3 0 4 1 5 0 2 1 3 0 4 1 5 0 2 1 3 0 4 1 5 0 2 1 3 0 4 1 5
32	P32-5	8 Channels at 1/8fs	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
32	P32-6	2 Channels at 1/4fs, 2 Channels at 1/8fs, 4 Channels at 1/16fs	0 2 1 4 0 3 1 5 0 2 1 6 0 3 1 7 0 2 1 4 0 3 1 5 0 2 1 6 0 3 1 7
32	P32-7	4 Channels at 3/16fs, 4 Channels at 1/16fs	0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 4 5 6 7 4 5 6 7
32	P32-8	2 Channels at 1/4fs, 8 Channels at 1/16fs	0 2 1 3 0 4 1 5 0 6 1 7 0 8 1 9 0 2 1 3 0 4 1 5 0 6 1 7 0 8 1 9
32	P32-9	6 Channels at 1/8fs, 4 Channels at 1/16fs	0 1 2 3 4 5 6 7 0 1 2 3 4 5 8 9 0 1 2 3 4 5 6 7 0 1 2 3 4 5 8 9
32	P32-10	4 Channels at 1/8fs, 8 Channels at 1/16fs	0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11 0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11
32	P32-11	2 Channels at 1/8fs, 12 Channel at 1/16fs	0 2 3 4 1 5 6 7 0 8 9 10 1 11 12 13 0 2 3 4 1 5 6 7 0 8 9 10 1 11 12 13
32	P32-12	16 Channel at 1/16fs	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
32	P32-13	4 Channels at 3/16fs, 8 Channels at 1/32fs	0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 4 5 6 7 8 9 10 11
32	P32-14	4 Channels at 1/8fs, 8 Channels at 1/16fs	0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11 0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11
32	P32-15	4 Channels at 1/8fs, 4 Channels at 1/16fs, 8 Channels at 1/32fs	0 4 1 8 2 5 3 9 0 6 1 10 2 7 3 11 0 4 1 12 2 5 3 13 0 6 1 14 2 7 3 15
32	P32-16	8 Channels at 3/32fs, 8 Channels at 1/32fs	0 1 2 6 3 4 5 7 0 1 2 6 3 4 5 7 0 1 2 6 3 4 5 7 8 9 10 11 12 13 14 15
32	P32-17	4 Channels at 1/8fs, 16 Channel at 1/32fs	0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11 0 12 1 13 2 14 3 15 0 16 1 17 2 18 3 19
32	P32-18	12 Channel at 1/16fs, 8 Channels at 1/32fs	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 16 17 18 19
32	P32-19	8 Channels at 1/16fs, 16 Channel at 1/32fs	0 8 1 9 2 10 3 11 4 12 5 13 6 14 7 15 0 16 1 17 2 18 3 19 4 20 5 21 6 22 7 23
32	P32-20	4 Channels at 1/16fs, 24 Channels at 1/32fs	0 4 5 6 1 7 8 9 2 10 11 12 3 13 14 15 0 16 17 18 1 19 20 21 2 22 23 24 3 25 26 27

Table 3-1: **Advanced Interleaved Data Channel Patterns *(Cont'd)***

No. Chans.	Seq. ID	Description	Interleaved Channel Pattern
32	P32-21	32 Channels at 1/32fs	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
48	P48-0	4 Channels at 1/4fs	0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
48	P48-1	8 Channels at 1/8fs	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
48	P48-2	4 Channels at 1/6fs, 4 Channels at 1/12fs	0 1 2 3 4 5 0 1 2 3 6 7 0 1 2 3 4 5 0 1 2 3 6 7 0 1 2 3 4 5 0 1 2 3 6 7 0 1 2 3 4 5 0 1 2 3 6 7
48	P48-3	4 Channels at 3/16fs, 4 Channels at 1/16fs	0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 4 5 6 7 4 5 6 7 4 5 6 7
48	P48-4	4 Channels at 1/6fs, 8 Channels at 1/24fs	0 1 2 3 4 5 0 1 2 3 6 7 0 1 2 3 8 9 0 1 2 3 10 11 0 1 2 3 4 5 0 1 2 3 6 7 0 1 2 3 8 9 0 1 2 3 10 11
48	P48-5	4 Channels at 1/8fs, 8 Channels at 1/16fs	0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11 0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11 0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11
48	P48-6	12 Channel at 1/12fs	0 1 2 3 4 5 6 7 8 9 10 11 0 1 2 3 4 5 6 7 8 9 10 11 0 1 2 3 4 5 6 7 8 9 10 11 0 1 2 3 4 5 6 7 8 9 10 11
48	P48-7	4 Channels at 1/8fs, 12 Channel at 1/24fs	0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11 0 12 1 13 2 14 3 15 0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11 0 12 1 13 2 14 3 15
48	P48-8	8 Channels at 1/12fs, 8 Channels at 1/24fs	0 1 2 3 4 5 6 7 8 9 10 11 0 1 2 3 4 5 6 7 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 0 1 2 3 4 5 6 7 12 13 14 15
48	P48-9	16 Channel at 1/16fs	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
48	P48-10	4 Channels at 1/6fs, 16 Channel at 1/48fs	0 1 2 3 4 5 0 1 2 3 6 7 0 1 2 3 8 9 0 1 2 3 10 11 0 1 2 3 12 13 0 1 2 3 14 15 0 1 2 3 16 17 0 1 2 3 18 19
48	P48-11	4 Channels at 1/8fs, 8 Channels at 1/24fs, 8 Channels at 1/48fs	0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11 0 12 1 13 2 14 3 15 0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11 0 16 1 17 2 18 3 19
48	P48-12	8 Channels at 1/12fs, 4 Channels at 1/24fs, 8 Channels at 1/48fs	0 1 2 3 4 5 6 7 8 9 10 11 0 1 2 3 4 5 6 7 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 0 1 2 3 4 5 6 7 16 17 18 19
48	P48-13	4 Channels at 1/12fs, 16 Channel at 1/24fs	0 4 5 1 6 7 2 8 9 3 10 11 0 12 13 1 14 15 2 16 17 3 18 19 0 4 5 1 6 7 2 8 9 3 10 11 0 12 13 1 14 15 2 16 17 3 18 19
48	P48-14	8 Channels at 1/12fs, 16 Channel at 1/48fs	0 1 2 3 4 5 6 7 8 9 10 11 0 1 2 3 4 5 6 7 12 13 14 15 0 1 2 3 4 5 6 7 16 17 18 19 0 1 2 3 4 5 6 7 20 21 22 23
48	P48-15	12 Channel at 1/16fs, 12 Channel at 1/48fs	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 16 17 18 19 0 1 2 3 4 5 6 7 8 9 10 11 20 21 22 23
48	P48-16	4 Channels at 1/12fs, 12 Channel at 1/24fs, 8 Channels at 1/48fs	0 4 5 1 6 7 2 8 9 3 10 11 0 12 13 1 14 15 2 16 17 3 18 19 0 4 5 1 6 7 2 8 9 3 10 11 0 12 13 1 14 15 2 20 21 3 22 23
48	P48-17	24 Channels at 1/24fs	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
48	P48-18	4 Channels at 1/8fs, 24 Channels at 1/48fs	0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11 0 12 1 13 2 14 3 15 0 16 1 17 2 18 3 19 0 20 1 21 2 22 3 23 0 24 1 25 2 26 3 27

Table 3-1: **Advanced Interleaved Data Channel Patterns *(Cont'd)***

No. Chans.	Seq. ID	Description	Interleaved Channel Pattern
48	P48-19	4 Channels at 1/12fs, 8 Channels at 1/24fs, 16 Channel at 1/48fs	0 4 12 1 5 13 2 6 14 3 7 15 0 8 16 1 9 17 2 10 18 3 11 19 0 4 20 1 5 21 2 6 22 3 7 23 0 8 24 1 9 25 2 10 26 3 11 27
48	P48-20	20 Channels at 1/24fs, 8 Channels at 1/48fs	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 24 25 26 27
48	P48-21	4 Channels at 1/12fs, 4 Channels at 1/24fs, 24 Channels at 1/48fs	0 4 8 1 9 10 2 5 11 3 12 13 0 6 14 1 15 16 2 7 17 3 18 19 0 4 20 1 21 22 2 5 23 3 24 25 0 6 26 1 27 28 2 7 29 3 30 31
48	P48-22	8 Channels at 1/16fs, 24 Channels at 1/48fs	0 8 1 9 2 10 3 11 4 12 5 13 6 14 7 15 0 16 1 17 2 18 3 19 4 20 5 21 6 22 7 23 0 24 1 25 2 26 3 27 4 28 5 29 6 30 7 31
48	P48-23	16 Channel at 1/24fs, 16 Channel at 1/48fs	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 24 25 26 27 28 29 30 31
48	P48-24	12 Channel at 1/24fs, 24 Channels at 1/48fs	0 12 1 13 2 14 3 15 4 16 5 17 6 18 7 19 8 20 9 21 10 22 11 23 0 24 1 25 2 26 3 27 4 28 5 29 6 30 7 31 8 32 9 33 10 34 11 35
48	P48-25	8 Channels at 1/24fs, 32 Channels at 1/48fs	0 8 9 1 10 11 2 12 13 3 14 15 4 16 17 5 18 19 6 20 21 7 22 23 0 24 25 1 26 27 2 28 29 3 30 31 4 32 33 5 34 35 6 36 37 7 38 39
48	P48-26	48 Channels at 1/48fs	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
64	P64-0	4 Channels at 1/4fs	0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
64	P64-1	4 Channels at 3/16fs, 4 Channels at 1/16fs	0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 4 5 6 7 4 5 6 7 4 5 6 7 4 5 6 7
64	P64-2	8 Channels at 1/8fs	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
64	P64-3	4 Channels at 3/16fs, 8 Channels at 1/32fs	0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 4 5 6 7 8 9 10 11 4 5 6 7 8 9 10 11
64	P64-4	4 Channels at 1/8fs, 8 Channels at 1/16fs	0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11 0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11 0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11 0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11
64	P64-5	16 Channel at 1/16fs	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
64	P64-6	4 Channels at 1/8fs, 4 Channels at 1/16fs, 8 Channels at 1/32fs	0 4 1 8 2 5 3 9 0 6 1 10 2 7 3 11 0 4 1 12 2 5 3 13 0 6 1 14 2 7 3 15 0 4 1 8 2 5 3 9 0 6 1 10 2 7 3 11 0 4 1 12 2 5 3 13 0 6 1 14 2 7 3 15
64	P64-7	8 Channels at 3/32fs, 8 Channels at 1/32fs	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 8 9 10 11 12 13 14 15
64	P64-8	4 Channels at 1/8fs, 16 Channel at 1/32fs	0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11 0 12 1 13 2 14 3 15 0 16 1 17 2 18 3 19 0 4 1 5 2 6 3 7 0 8 1 9 2 10 3 11 0 12 1 13 2 14 3 15 0 16 1 17 2 18 3 19
64	P64-9	12 Channel at 1/16fs, 8 Channels at 1/32fs	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 16 17 18 19 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 16 17 18 19

Table 3-1: **Advanced Interleaved Data Channel Patterns *(Cont'd)***

No. Chans.	Seq. ID	Description	Interleaved Channel Pattern
64	P64-10	8 Channels at 1/16fs, 16 Channel at 1/32fs	0 8 1 9 2 10 3 11 4 12 5 13 6 14 7 15 0 16 1 17 2 18 3 19 4 20 5 21 6 22 7 23 0 8 1 9 2 10 3 11 4 12 5 13 6 14 7 15 0 16 1 17 2 18 3 19 4 20 5 21 6 22 7 23
64	P64-11	4 Channels at 1/16fs, 24 Channels at 1/32fs	0 4 5 6 1 7 8 9 2 10 11 12 3 13 14 15 0 16 17 18 1 19 20 21 2 22 23 24 3 25 26 27 0 4 5 6 1 7 8 9 2 10 11 12 3 13 14 15 0 16 17 18 1 19 20 21 2 22 23 24 3 25 26 27
64	P64-12	32 Channels at 1/32fs	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
64	P64-13	8 Channels at 3/32fs, 16 Channel at 1/64fs	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
64	P64-14	8 Channels at 1/16fs, 16 Channel at 1/32fs	0 8 1 9 2 10 3 11 4 12 5 13 6 14 7 15 0 16 1 17 2 18 3 19 4 20 5 21 6 22 7 23 0 8 1 9 2 10 3 11 4 12 5 13 6 14 7 15 0 16 1 17 2 18 3 19 4 20 5 21 6 22 7 23
64	P64-15	8 Channels at 1/16fs, 8 Channels at 1/32fs, 16 Channel at 1/64fs	0 8 1 16 2 9 3 17 4 10 5 18 6 11 7 19 0 12 1 20 2 13 3 21 4 14 5 22 6 15 7 23 0 8 1 24 2 9 3 25 4 10 5 26 6 11 7 27 0 12 1 28 2 13 3 29 4 14 5 30 6 15 7 31
64	P64-16	16 Channel at 3/64fs, 16 Channel at 1/64fs	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
64	P64-17	8 Channels at 1/16fs, 32 Channels at 1/64fs	0 8 1 9 2 10 3 11 4 12 5 13 6 14 7 15 0 16 1 17 2 18 3 19 4 20 5 21 6 22 7 23 0 24 1 25 2 26 3 27 4 28 5 29 6 30 7 31 0 32 1 33 2 34 3 35 4 36 5 37 6 38 7 39
64	P64-18	24 Channels at 1/32fs, 16 Channel at 1/64fs	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 32 33 34 35 36 37 38 39
64	P64-19	16 Channel at 1/32fs, 32 Channels at 1/64fs	0 16 1 17 2 18 3 19 4 20 5 21 6 22 7 23 8 24 9 25 10 26 11 27 12 28 13 29 14 30 15 31 0 32 1 33 2 34 3 35 4 36 5 37 6 38 7 39 8 40 9 41 10 42 11 43 12 44 13 45 14 46 15 47
64	P64-20	8 Channels at 1/32fs, 48 Channels at 1/64fs	0 8 9 10 1 11 12 13 2 14 15 16 3 17 18 19 4 20 21 22 5 23 24 25 6 26 27 28 7 29 30 31 0 32 33 34 1 35 36 37 2 38 39 40 3 41 42 43 4 44 45 46 5 47 48 49 6 50 51 52 7 53 54 55
64	P64-21	64 Channels at 1/64fs	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

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 for details of the TDATA format.

This feature can be used in conjunction with the Interleaved Data Channel Filters 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. The Vivado IDE (see GUI) offers the facility to generate re-ordered coefficient files for use with the RELOAD channel and during core generation delivers an informational text file to the project area named `<component_name>_reload_order.txt`, which lists the indexes of the coefficients, Coefficient x, in the order they should be reloaded into the filter through the reload channel Reload index x.

Reload Order File

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, you must pad the filter response at the beginning and the end with (actual - specified)/2 zeros before applying the reload order. Figure 3-18 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 indexes. 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 3-52 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 3-52: **Reload Order Text File Format Example 1**

Figure 3-53 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 3-53: **Reload Order Text File Format Example 2**

Figure 3-54 contains an example with the same filter configuration as in Figure 3-53, 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 3-54: **Reload Order Text File Format Example 3**

Coefficient Quantization

- Integer Coefficients
- Quantize Only
- Maximize Dynamic Range
- Best Precision Fractional Length

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 you wish 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 you can reduce it. 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 you to explore the trade-off between filter performance and resources by varying the coefficient width parameter.

Maximize Dynamic Range

You 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 Vivado IDE.

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 your 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 Vivado IDE.

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.

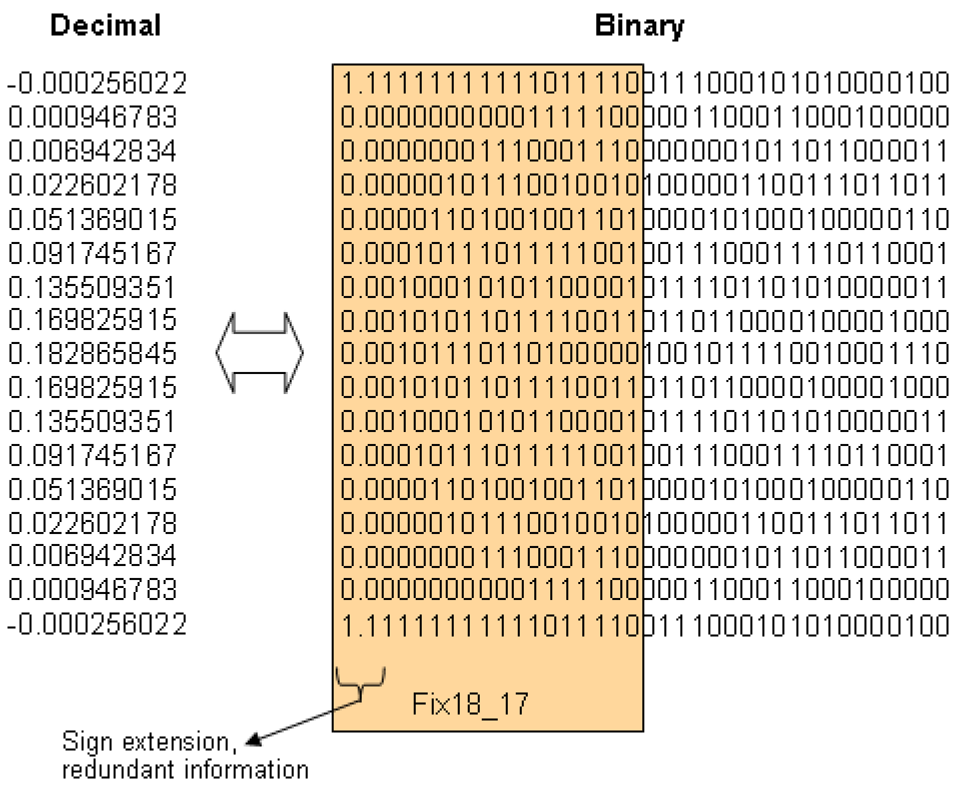
An integer coefficient width of 18 bits gives a maximum positive value of 131071 and a maximum negative number of -131072. Considering the fractional bits, this gives a maximum representable positive number of 131071/(2^19)=0.249998092 and a maximum representable negative number of 131072/(2^19)=0.25. 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).

IMPORTANT: While some performance improvement can be achieved by using the full dynamic range of the coefficient bit width, you must be satisfied that any changes are acceptable using the frequency response plot. You must also account 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 you do not want to introduce any additional scaling into the design, select *Quantize Only*.

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

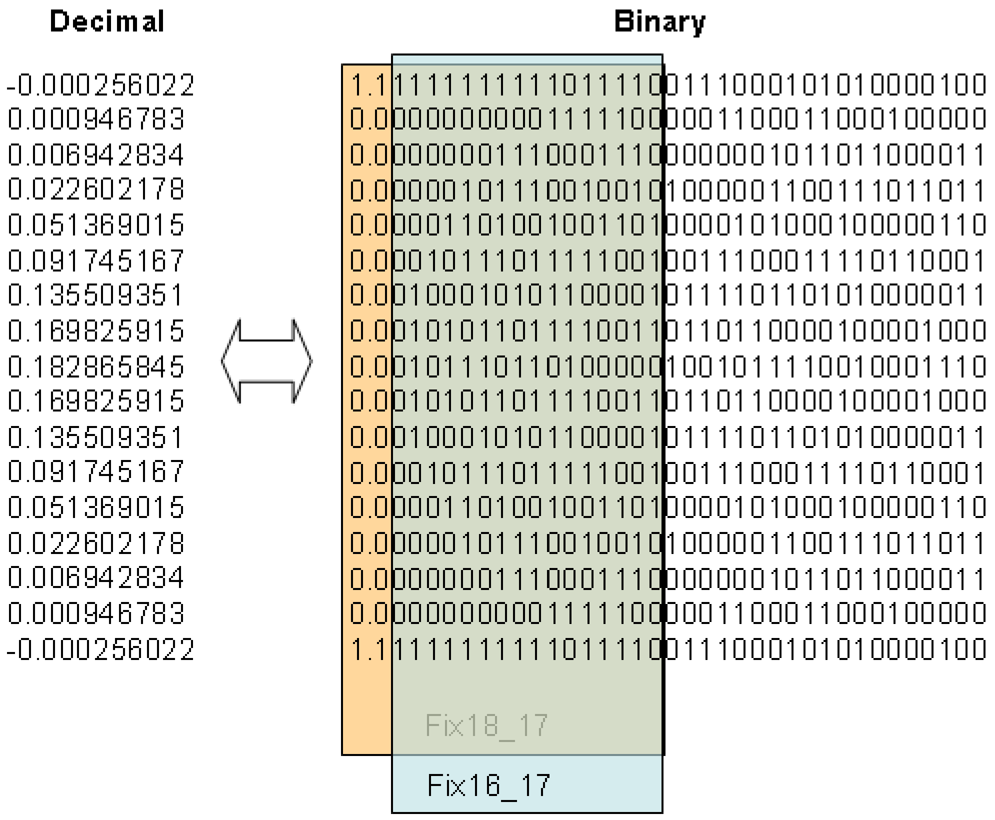
Quantize Only section, the FIR Compiler analyzes the filter coefficients to determine how many bits are required to represent the integer portion of the coefficient values. All the remaining coefficient bits are then allocated to represent the fractional portion of the coefficients. When all the specified 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.



Decimal	Binary
-0.000256022	1.111111111110111100111000101010000100
0.000946783	0.000000000011111000001100011000100000
0.006942834	0.000000011100011100000001011011000011
0.022602178	0.000001011100100101000001100111011011
0.051369015	0.000011010010011010000101000100000110
0.091745167	0.000101110111110010011100011110110001
0.135509351	0.001000101011000010111101101010000011
0.169825915	0.001010110111100110110110000100001000
0.182865845	0.001011101101000001001011110010001110
0.169825915	0.001010110111100110110110000100001000
0.135509351	0.001000101011000010111101101010000011
0.091745167	0.000101110111110010011100011110110001
0.051369015	0.000011010010011010000101000100000110
0.022602178	0.000001011100100101000001100111011011
0.006942834	0.000000011100011100000001011011000011
0.000946783	0.000000000011111000001100011000100000
-0.000256022	1.111111111110111100111000101010000100

Figure 3-55: **Coefficient Quantization Fix18_17**

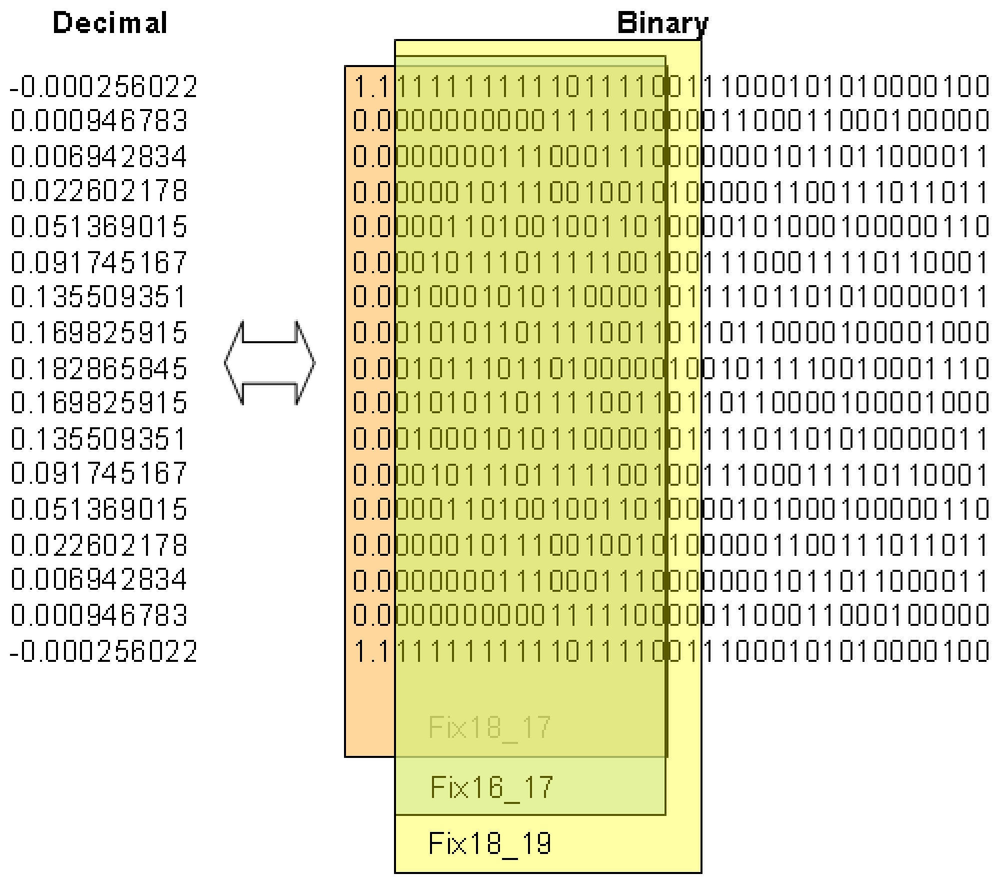
In Figure 3-55 the coefficient values are represented using 18 bits. The binary point is positioned such that 17 bits are used to represent the fractional portion of the number. An analysis of the coefficients reveals that no value has a magnitude greater than 0.25; therefore, the upper two MSBs are a sign extension and contain redundant information.



Decimal	Binary
-0.000256022	1.111111111110111100111000101010000100
0.000946783	0.000000000011111000001100011000100000
0.006942834	0.000000001110001110000000101101100011
0.022602178	0.000000101110010010100000110011101101 1
0.051369015	0.000001101001001101000010100010000011 0
0.091745167	0.000010111011111001001110001111011000 1
0.135509351	0.001000101011000010111101101010000011
0.169825915	0.001010110111100110110110000100001000
0.182865845	0.001011101101000001001011110010001110
0.169825915	0.001010110111100110110110000100001000
0.135509351	0.001000101011000010111101101010000011
0.091745167	0.000010111011111001001110001111011000 1
0.051369015	0.000001101001001101000010100010000011 0
0.022602178	0.000000101110010010100000110011101101 1
0.006942834	0.000000001110001110000000101101100011
0.000946783	0.000000000011111000001100011000100000
-0.000256022	1.111111111110111100111000101010000100

Figure 3-56: **Coefficient Quantization Fix16_17**

In Figure 3-56, 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.



Decimal	Binary
-0.000256022	1.1111111111101111100111000101010000100
0.000946783	0.0000000000011111000000110001100010000 00
0.006942834	0.0000000011100011100000001011011000011
0.022602178	0.000001011100100101000001100111011011
0.051369015	0.000011010010011010000101000100000110
0.091745167	0.000101110111110010011100011110110001
0.135509351	0.001000101011000010111110110101000011
0.169825915	0.001010110111100110110110000100001000
0.182865845	0.001011101101000001000101111001000110
0.169825915	0.001010110111100110110110000100001000
0.135509351	0.001000101011000010111110110101000011
0.091745167	0.000101110111110010011100011110110001
0.051369015	0.000011010010011010000101000100000110
0.022602178	0.000001011100100101000001100111011011
0.006942834	0.0000000011100011100000001011011000011
0.000946783	0.0000000000011111000000110001100010000 00
-0.000256022	1.1111111111101111100111000101010000100

Figure 3-57: **Coefficient Quantization Fix18_19**

In Figure 3-57 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. Equation 3-2 demonstrates this calculation, where B is the calculated bit growth, N is the number for filter coefficients and C_w is the coefficient width

$$B = C_w + ceil[\log_2 N] \qquad \textit{Equation 3-2}$$

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. Equation 3-3 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.

$$B = ceil\left[\log_2\left(\sum_{n=0}^{(N-1)} |a_n|\right)\right] \quad \text{Equation 3-3}$$

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

Equation 3-4 gives the cores full precision output width, where B is the calculated bit growth (given by Equation 3-2 or Equation 3-3), D_w is the data width and A_w is the full precision output width.

$$A_w = D_w + B \quad \text{Equation 3-4}$$

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. This value is then reduced by any output rounding. Equation 3-5 demonstrates this calculation, where O_w = output width, O_{fw}=output fractional width, D_{fw}=data fractional width, C_{fw}=coefficient fractional width and A_w=full precision output width.

$$O_{fw} = D_{fw} + C_{fw} - max(0, A_w - O_w) \quad \text{Equation 3-5}$$

Output Rounding

As mentioned in Symmetric Rounding to Highest Magnitude, 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 you to select the preferred trade-off between resource utilization, rounding precision, and rounding bias:

- Full Precision
- Truncation (removal of LSBs)
- Non-symmetric Rounding to Positive
- Non-symmetric Rounding to Negative
- Symmetric Rounding to Highest Magnitude
- Symmetric Rounding to Zero
- Convergent Rounding
- Resource Implications of Rounding

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 3-58 through Figure 3-60, 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 $2^{-(m+1)}$.

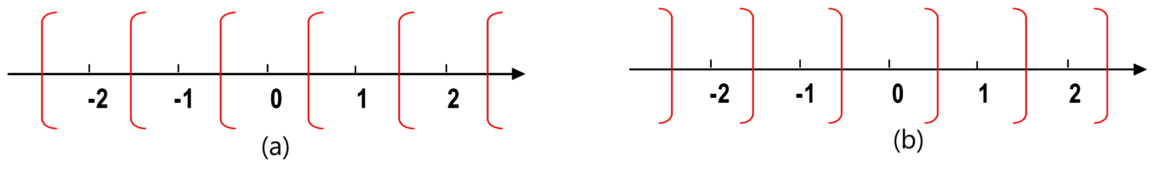


Figure 3-58: **Non-symmetric Rounding (a) to Positive (b) to Negative**

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 because 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 3-59). 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.

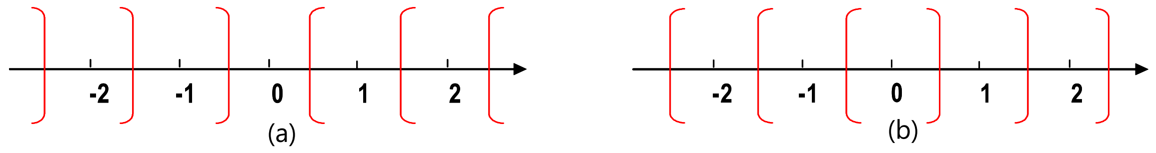


Figure 3-59: **Symmetric Rounding (a) to Highest Magnitude (b) to Zero**

Convergent Rounding

Convergent rounding chooses the rounding direction for midpoints as either toward odd or even numbers, rather than toward positive or negative (Figure 3-60). 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.

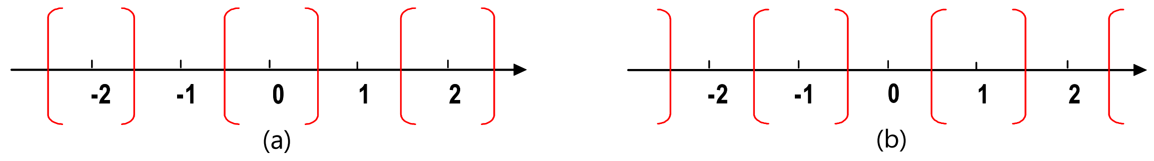


Figure 3-60: **Convergent Rounding (a) to Even (b) to Odd**

Resource Implications of Rounding

Ensure that you consider the implications of selecting a particular rounding mode on resource utilization. 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.

Table 3-2 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.

Table 3-2 is indicative only, and certain combinations for which hardware support is indicated actually require the extra DSP, and vice versa.

Table 3-2: **Indicative Table of Hardware Support for Rounding Modes for Particular Filter Types**

Filter Type	Non-symmetric	Symmetric (Infinity)	Symmetric (Zero)	Convergent
Single Rate	Yes	Yes	Yes	Yes
Half-band	Yes	Yes	Yes	Yes
Interpolating without Symmetry	Yes	Yes	Yes	Yes
Interpolate by 2, Odd Symmetry	Yes	Yes	Yes	Yes
Interpolating with Symmetry (others)	No	No	No	No
Interpolating Half-band	Yes	Yes	No	Yes
Decimating, Single-channel	Yes	Yes	Yes	Yes
Decimating, Multichannel	Yes	Yes	Yes	Yes
Decimating Half-band	Yes	Yes	Yes	Yes

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 to chain together more DSP slices than are available in a single column of the select device. Figure 3-61 shows the structures implemented.

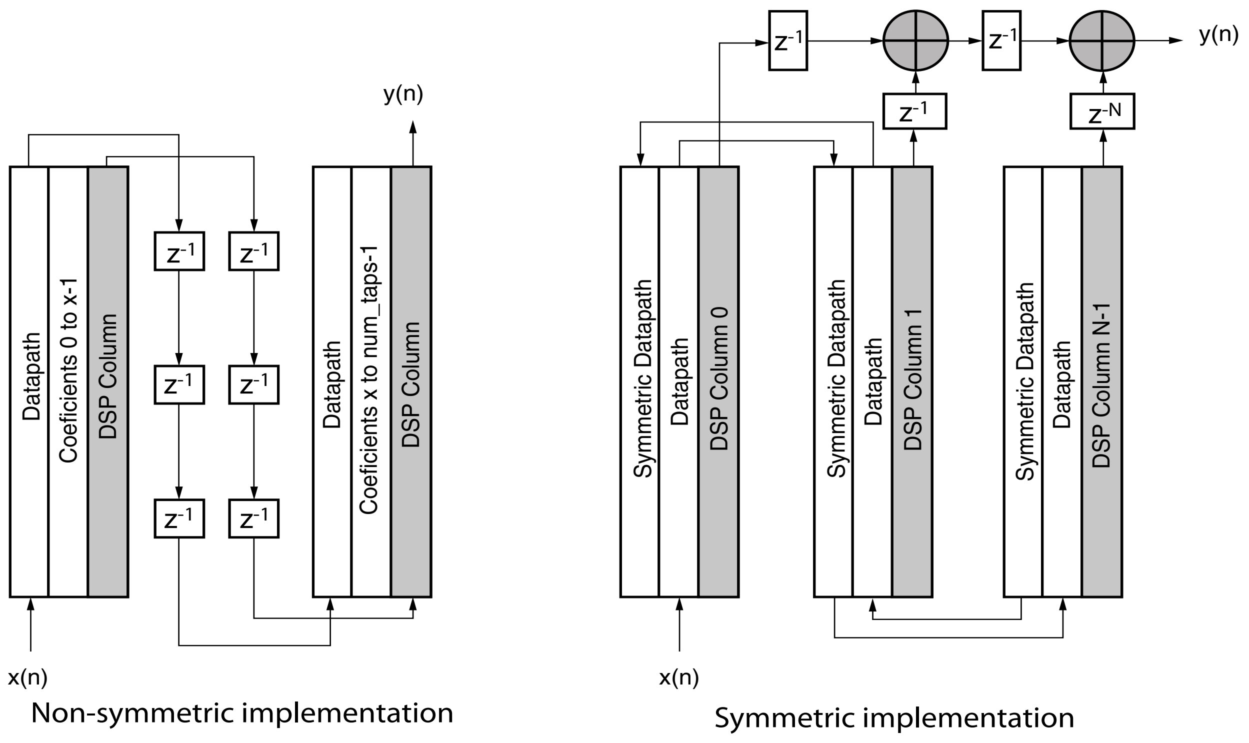


Figure 3-61: **Multi-Column Implementations**

The DSP column lengths are displayed on the Details Implementation Options page of the Vivado IDE. The implemented column lengths can be determined automatically, *Multi-column Support: Automatic*, or user-specified, *Multi-column Support: Custom*. The length of each implemented DSP column can be specified using the Column Configuration parameter. See Detailed Implementation Tab 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). 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 IDE displays the core DSP slice usage given all the core parameters.

Data and Coefficient Bit Width

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, the core uses multiple DSP slice columns to implement the filter. Table 3-3 provides a guide to the number of DSP columns that are required for various combinations of data and coefficient widths.

Table 3-3: **DSP Slice Column Usage for Given Data and Coefficient Widths**

Data Width		Coefficient Width		Number of DSP Slice Columns
Unsigned	Signed	Unsigned	Signed	
<=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

Note: The data/coefficient widths at which 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.

The Data Width threshold is further reduced by a bit when coefficient symmetry is being utilized by the core, see Filter Symmetry.

The Coefficient Width threshold is further reduced by a bit when symmetric pairs are being utilized by the core, see Polyphase Interpolator Exploiting Symmetric Pairs.

Output Rounding Selection

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

Multiple Channel versus Parallel Datapaths

The Interleaved Data Channel 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.

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, shown 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.

C Model Reference

The Xilinx LogiCORE™ IP FIR Compiler v7.0 core bit accurate C model is a self-contained, linkable, shared library that models the functionality of this core with finite precision arithmetic. This model provides a bit accurate representation of the various modes of the FIR Compiler v7.0 core, and it is suitable for inclusion in a larger framework for system-level simulation or core-specific verification.

This chapter provides information about the Xilinx LogiCORE IP FIR Compiler v7.0 bit accurate C model for 32-bit and 64-bit Linux, and 32-bit and 64-bit Windows platforms.

The model consists of a set of C functions that reside in a shared library. Example C code is provided to demonstrate how these functions form the interface to the C model. Full details of this interface are given in C Model Interface.

The model is bit accurate but not cycle-accurate; it performs exactly the same operations as the core. However, it does not model the core latency or its interface signals.

The C model ZIP files are delivered in the `cmodel` directory of a generated core.

Unpacking and Model Contents

There are separate ZIP files containing all the files necessary for use with a specific computing platform. Each ZIP file contains:

- The C model shared library
- Multiple Precision Integers and Rationals (MPIR) [Ref 8] shared libraries and header files.
- The C model header file
- The example code showing customers how to call the C model
- Documentation

Note: The C model uses MPIR libraries, which is provided in the ZIP files. MPIR is an interface-compatible version of the GNU Multiple Precision (GMP) [Ref 9] library, with greater support for Windows platforms. MPIR has been compiled using its GMP compatibility option, so the MPIR library and header file use GMP file names.

Table 4-1: **Example C Model ZIP File Contents - Linux**

File	Description
fir_compiler_v7_0_bitacc_cmodel.h	Header file which defines the C model API
libIp_fir_compiler_v7_0_bitacc_cmodel.so	Model shared object library
libstlport.so.5.1	STL portability library
libgmp.so.7	MPIR library, used by the C model
libgmpxx.so.1	MPIR Class library, used internally by the C model
gmp.h	MPIR header file, used by the C model
run_bitacc_cmodel.c	Example program for calling the C model
README.txt	Release notes
fir_compiler_v7_0_bitacc_mex.cpp	MATLAB® MEX function source
make_fir_compiler_v7_0_mex.m	MATLAB MEX function compilation script
run_fir_compiler_v7_0_mex.m	MATLAB MEX function example script
@fir_compiler_v7_0_bitacc	MATLAB MEX function class directory

Table 4-2: **Example C Model ZIP File Contents - Windows**

File	Description
fir_compiler_v7_0_bitacc_cmodel.h	Header file which defines the C model API
libIp_fir_compiler_v7_0_bitacc_cmodel.dll	Model dynamically linked library
libIp_fir_compiler_v7_0_bitacc_cmodel.lib	Model .lib file for compiling
stlport.5.1.dll	STL portability library
libgmp.dll	MPIR library, used by the C model
libgmp.lib	MPIR .lib file for compiling
gmp.h	MPIR header file, used by the C model
run_bitacc_cmodel.c	Example program for calling the C model
README.txt	Release notes
fir_compiler_v7_0_bitacc_mex.cpp	MATLAB MEX function source
make_fir_compiler_v7_0_mex.m	MATLAB MEX function compilation script
run_fir_compiler_v7_0_mex.m	MATLAB MEX function example script
@fir_compiler_v7_0_bitacc	MATLAB MEX function class directory

Installation

Linux

- Unpack the contents of the ZIP file.
- Ensure that the directory where the `libIp_fir_compiler_v7_0_bitacc_cmodel.so`, `libgmp.so.7` and `libgmpxx.so.1` files reside is included in the path of the environment variable LD_LIBRARY_PATH.

Windows

- Unpack the contents of the ZIP file.
- Ensure that the directory where the `libIp_fir_compiler_v7_0_bitacc_cmodel.dll` and `libgmp.dll` files reside is
  a. included in the path of the environment variable PATH or
  b. the directory in which the executable that calls the C model is run.

C Model Interface

The Application Programming Interface (API) of the C model is defined in the header file `fir_compiler_v7_0_bitacc_cmodel.h`. The interface consists of data structures and functions as described in the sections:

- Constants
- Type Definitions
- Dynamic Arrays
- Structures
- Functions

An example C file, `run_bitacc_cmodel.c`, is included with the C libraries. This file demonstrates how to call the C model.

Constants

Table 4-3: **Constants**

Name	Value
Error codes	
XIP_STATUS_OK	0
XIP_STATUS_ERROR	1
Filter Types	
XIP_FIR_SINGLE_RATE	0
XIP_FIR_INTERPOLATION	1
XIP_FIR_DECIMATION	2
XIP_FIR_HILBERT	3
XIP_FIR_INTERPOLATED	4
Rate Change	
XIP_FIR_INTEGER_RATE	0
XIP_FIR_FRACTIONAL_RATE	1
Channel Sequence	
XIP_FIR_BASIC_CHAN_SEQ	0
XIP_FIR_ADVANCED_CHAN_SEQ	1
Quantization	
XIP_FIR_INTEGER_COEFF	0
XIP_FIR_QUANTIZED_ONLY	1
XIP_FIR_MAXIMIZE_DYNAMIC_RANGE	2
Output Rounding	
XIP_FIR_FULL_PRECISION	0
XIP_FIR_TRUNCATE_LSBS	1
XIP_FIR_SYMMETRIC_ZERO	2
XIP_FIR_SYMMETRIC_INF	3
XIP_FIR_CONVERGENT_EVEN	4
XIP_FIR_CONVERGENT_ODD	5
XIP_FIR_NON_SYMMETRIC_DOWN	6
XIP_FIR_NON_SYMMETRIC_UP	7
Configuration Method	
XIP_FIR_CONFIG_SINGLE	0
XIP_FIR_CONFIG_BY_CHANNEL	1

Type Definitions

Table 4-4: **Type Definitions**

Field Name	Description
General types	
typedef double xip_real	Scalar type alias
typedef unsigned integer xip_uint	Integer type alias
typedef struct xip_complex	Complex data type
typedef mpz_t xip_mpz	MPIR [Ref 8] integer type alias
typedef struct xip_mpz_complex	Complex type based on xip_mpz
typedef int xip_status	Result code from API functions
typedef struct _xip_fir_v7_0 xip_fir_v7_0	Handle type to refer to an instance of the model
typdef enum xip_fir_v7_0_pattern	Defines enumerated values for all the advance channel patterns
Handler function signatures	
typedef void (*xip_msg_handler)(void* handle, int error,const char* msg)	Interface to a message handler function
typedef void (*xip_array_real_handler)(const xip_array_real* data, void* handle, void* opt_arg)	Interface to data packet handler function
Structures	
typedef struct xip_fir_v7_0_config	Model configuration structure, Table 4-6
typedef struct xip_fir_v7_0_cnfg_packet	Configuration packet structure, Table 4-7
typedef struct xip_fir_v7_0_rld_packet	Reload packet structure, Table 4-8
Array types	
typedef struct xip_array_uint	See Dynamic Arrays
typedef struct xip_array_real	
typedef struct xip_array_complex	
typedef struct xip_array_mpz	

Dynamic Arrays

The C model represents input and output data using multi-dimensional dynamic arrays. The `xip_array_<type>` structure is used to specify a multi-dimensional dynamic array containing elements of type `xip_<type>`. Several utility functions are provided that allow creation, allocation and destruction of array instances.

For each array type, the DECLARE_XIP_ARRAY(<type>) macro can be used to declare the structure and utility function prototypes. The C model header already contains declarations for the following array types:

- `xip_array_real` for arrays of `xip_real`
- `xip_array_complex` for arrays of `xip_complex`
- `xip_array_uint` for arrays of `xip_uint`
- `xip_array_mpz` for arrays of `xip_mpz`
- `xip_array_mpz_complex` for arrays of `xip_mpz_complex`

The utility functions for each array type can be defined using the DEFINE_XIP_ARRAY(<type>) macro. The utility function must be defined somewhere within user code before the functions can be used; see the `run_bitacc_cmodel.c` file for examples.

Further utility functions, specific to the FIR Compiler C Model, can be declared and defined using the DECLARE_FIR_XIP_ARRAY(<type>) and DEFINE_FIR_XIP_ARRAY(<type>) macros. The C model header already contains declarations for the following array types:

- `xip_array_real`
- `xip_array_complex`
- `xip_array_mpz`
- `xip_array_mpz_complex`

Structure

The `xip_array_<type>` structure is used to specify a multi-dimensional array of data with type <type>. The content is summarized in Table 4-5.

Table 4-5: **xip_array_<type>**

Field Name	Type	Description
data	xip_<type>*	Pointer to array of data
data_size	size_t	Current number of elements in the data array
data_capacity	size_t	Max number of elements in the data array
dim	size_t*	Pointer to dimension array
dim_size	size_t	Current number of elements in the dimension array, dim
dim_capacity	size_t	Max number of elements in dim array
owner	unsigned int	Ownership control. A value of 0 indicates that the structure and associated memory (for the data and dim fields) is allocated and owned by the xip_array_<type>_* functions, in which case the model can automatically resize arrays as required. Any other value indicates that the memory is owned by the user, and the model must report an error if an array is of insufficient capacity.

This data structure is defined for types:

- `xip_real`
- `xip_complex`
- `xip_uint`
- `xip_mpz`
- `xip_mpz_complex`

General Functions

Create Array

```
xip_array_<type>*
xip_array_<type>_create();
```

This function allocates and initializes an empty array for holding values of type <type>. The function returns a pointer to the created structure, or null if the structure cannot be created. The structure fields are all initialized to zero indicating an empty array, with ownership associated with the `xip_array_<type>_*` functions.

Reserve Data Memory

```
xip_status
xip_array_<type>_reserve_data(
   xip_array_<type>* p,
   size_t max_nels
);
```

This function ensures that array `p` has sufficient space to store up to `max_nels` elements of data. If the current `data_capacity` is insufficient and the current owner is zero, the function attempts to allocate or reallocate space to meet the request. The function returns XIP_STATUS_OK if the array capacity is now sufficient or XIP_STATUS_ERROR if memory could not be allocated.

Note: This function does not change the data or dimensions held within the array in any way; the contents of the array after calling the function are equivalent to the contents before calling the function, even if memory is reallocated. Also, this function never reduces memory allocation; use xip_array_<type>_destroy to release memory.

Reserve Dimension Memory

```
xip_status
xip_array_<type>_reserve_dim(
   xip_array_<type>* p,
   size_t max_nels
);
```

This function ensures that array `p` has sufficient space to store up to `max_ndims` dimensions. If the current `dim_capacity` is insufficient and the current owner is zero, the function attempts to allocate or reallocate space to meet the request. The function returns XIP_STATUS_OK if the array capacity is now sufficient or XIP_STATUS_ERROR if memory could not be allocated.

Note: This function does not change the data or dimensions held within the array in any way; the contents of the array after calling the function are equivalent to the contents before calling the function, even if memory is reallocated. Also, this function never reduces memory allocation; use xip_array_<type>_destroy to release memory.

Destroy Array

```
xip_array_<type>*
xip_array_<type>_create(
    xip_array_<type>* p
);
```

This function attempts to release all memory associated with array `p`. If the owner field is zero, the function releases the memory associated with `data`, `dim` and `p`, and returns null indicating success. If owner is non-zero the function returns `p`, indicating failure.

FIR Compiler Specific Functions

The following functions have been added to aid the use of the array types with the FIR Compiler C Model and, specifically, the advanced channel patterns.

Set Channel

```
xip_status
xip_array_<type>_set_chan(
    xip_array_<type>* p
    const <type> value,
    size_t path,
    size_t chan,
    size_t index
    xip_fir_v7_0_pattern pattern
);
```

This function maps an array index for one channel, specified by `path` and `chan`, onto the 3-D structure of `xip_array_<type>` structure expected by the `xip_fir_v7_0_data_send` (see Send DATA Packet) and `xip_fir_v7_0_data_get` (see Get DATA Packet) functions of the model.

This function should be particularly useful for the Advanced Interleaved Channels feature; where locations in the input array are remapped to duplicate entries for some channels. Figure 4-2 shows this requirement.

`pattern` should be set to `P_BASIC` for a Basic Interleaved Channel model configuration and set to the current pattern ID for an Advanced Interleaved Channel model configuration. See Table 3-1 for a list of all the supported patterns and see `fir_compiler_v7_0_bitacc_cmodel.h` for the enumerated pattern IDs.

Note: If the value of index exceeds the current capacity (data_capacity) of p then the function issues a XIP_STATUS_ERROR. If the value of index exceeds number of elements (data_size) of p then the function sets the new size of the array.

Get Channel

```
xip_status
xip_array_<type>_get_chan(
    xip_array_<type>* p
<type>* value,
    size_t path,
    size_t chan,
    size_t index
    xip_fir_v7_0_pattern pattern
);
```

This function is the reciprocal of the `xip_array_<type>_set_chan` function and extracts the value of an individual channel for a given index, path and channel. The function issues an XIP_STATUS_ERROR if the index exceeds the array capacity or size.

Structures

The `xip_fir_v7_0_config` structure contains all parameters that affect the filter configuration. Most are duplicates of the core XCO parameters. The filter coefficients of the model are provided using the `coeff` field and are quantized (if specified) in the same manner as by the core GUI.

Table 4-6: **xip_fir_v7_0_config**

Field Name	Type	Description
name	const char*	
filter_type	unsigned int	Select from: XIP_FIR_SINGLE_RATE XIP_FIR_INTERPOLATION XIP_FIR_DECIMATION XIP_FIR_HILBERT XIP_FIR_INTERPOLATED
rate_change	unsigned int	Select from: XIP_FIR_INTEGER_RATE XIP_FIR_FRACTIONAL_RATE
interp_rate	unsigned int	Specifies the interpolation (or up-sampling) factor
decim_rate	unsigned int	Specifies the decimation (or down-sampling) factor
zero_pack_factor	unsigned int	Specifies the zero packing factor for Interpolated filters
coeff	const double*	Pointer to coefficient array
coeff_padding	unsigned int	Specifies the amount of zero padding added to the front of the filter. ***Note:*** The core GUI reports this value for a given core configuration.

Table 4-6: **xip_fir_v7_0_config *(Cont'd)***

Field Name	Type	Description
num_coeffs	unsigned int	Specifies the number of coefficients in one filter
coeff_sets	unsigned int	Specifies the number of coefficient sets in the coeff array
reloadable	unsigned int	Specifies if the coefficients are reloadable; 0 = No, 1 = Yes
is_halfband	unsigned int	Specifies if halfband coefficients have been specified; 0 = No, 1 = Yes
quantization	unsigned int	Select from: XIP_FIR_INTEGER_COEFF XIP_FIR_QUANTIZED_ONLY XIP_FIR_MAXIMIZE_DYNAMIC_RANGE
coeff_width	unsigned int	The model uses these parameters, if requested, to quantize the supplied coefficients
coeff_fract_width	unsigned int	
chan_seq	unsigned int	Select from: XIP_FIR_BASIC_CHAN_SEQ XIP_FIR_ADVANCED_CHAN_SEQ
num_channels	unsigned int	Specifies the number of data channels supported
init_pattern	xip_fir_v7_0_pattern	Specifies the initial channel pattern used by the model when Advanced Interleaved Channels have been selected
num_paths	unsigned int	Specifies the number of datapaths supported
data_width	unsigned int	The model uses these parameters to quantize the input samples of the model
data_fract_width	unsigned int	
output_rounding_mode	unsigned int	Select from: XIP_FIR_FULL_PRECISION XIP_FIR_TRUNCATE_LSBS XIP_FIR_SYMMETRIC_ZERO XIP_FIR_SYMMETRIC_INF XIP_FIR_CONVERGENT_EVEN XIP_FIR_CONVERGENT_ODD XIP_FIR_NON_SYMMETRIC_DOWN XIP_FIR_NON_SYMMETRIC_UP
output_width	unsigned int	Ignored when XIP_FIR_FULL_PRECISION
output_fract_width	unsigned int	**READ ONLY** Provides the number of fractional bits present in the output word
config_method	unsigned int	Select from: XIP_FIR_CONFIG_SINGLE XIP_FIR_CONFIG_BY_CHANNEL

The `xip_fir_v7_0_cnfg_packet` structure is supplied to the `xip_fir_v7_0_config_send` function (see Send CONFIG Packet) to update the channel pattern and coefficient set used by the model.

Table 4-7: **xip_fir_v7_0_cnfg_packet**

Field Name	Type	Description
chanpat	xip_fir_v7_0_pattern	Specifies the Advanced Interleaved Channel pattern to be used
fsel	xip_array_uint*	Filter set to use, 1-D array; specifies one value for all channels (XIP_FIR_CONFIG_SINGLE) or individually for each interleaved channel (XIP_FIR_CONFIG_BY_CHANNEL)

The `xip_fir_v7_0_rld_packet` structure is supplied to the `xip_fir_v7_0_reload_send` function (see Send RELOAD Packet) to update a given coefficient set with new filter coefficients. As with the core, a configuration packet must be processed by the model to apply any pending reload packets.

Table 4-8: **xip_fir_v7_0_rld_packet**

Field Name	Type	Description
fsel	Int	Filter set to reload
coeff	xip_array_real*	Pointer to an array containing the new coefficients to be loaded, 1-D array.

Functions

Model Configuration Functions

Get Version

```
const char* xip_fir_v7_0_get_version(void);
```

The function returns a string describing the version of the model.

Get Default Configuration

```
xip_status
xip_fir_v7_0_get_default_config(
    xip_fir_v7_0_config* config
)
```

This function populates the `xip_fir_v7_0_config` configuration structure pointed to by `config` with the default configuration of the FIR Compiler v7.0 core.

Create Model Object

```
xip_fir_v7_0
xip_fir_v7_0_create(
    const xip_fir_v7_0_config* config,
    xip_msg_handler msg_handler,
    void* msg_handle
)
```

This function creates a new model instance, based on the configuration data pointed to by `config`.

The msg_handler argument is a pointer to a function taking three arguments as previously defined in Type Definitions. This function pointer is retained by the model object and is called whenever the model wishes to issue a note, warning or error message. Its arguments are:

1. A generic pointer (void*). This is always the value that was passed in as the msg_handle argument to the create function.
2. An integer (int) indicating whether the message is an error (1) or a note or warning (0).
3. The message string itself.

If the handler argument is a null pointer, then the C model outputs no messages at all. Using this mechanism, you can choose whether to output messages to the console, log them to a file or ignore them completely.

The create function returns a pointer to the newly created object. If the object cannot be created, then a diagnostic error message is emitted using the supplied handler function (if any) and a null pointer is returned.

If the data and coefficient widths, number of coefficients and output precision result in an output precision greater than supported by the double (xip_real) data type then the model uses the mpz_t data type [Ref 8] (xip_mpz) and issues a warning indicating this requirement when this function is executed.

Get Model Configuration

```
xip_status
xip_fir_v7_0_get_config (
    xip_fir_v7_0* model,
    xip_fir_v7_0_config* config
)
```

This function returns the full configuration of the model. The function is intended to be primarily used to determine the output width and output fractional width of the model.

Note: The coeff pointer of the returned xip_fir_v7_0_config structure is set to NULL.

Reset Model Object

```
xip_status
xip_fir_v7_0_reset(
    xip_fir_v7_0* model
);
```

This function resets in the internal state of the FIR Compiler model object pointed to by model. A reset causes all data and pending configuration packets to be cleared from the model. As per the core, any pending reload packets are retained.

Destroy Model Object

```
xip_status
xip_fir_v7_0_destroy(
   xip_fir_v7_0* model
);
```

This function deallocates the model object pointed to by `model`. Any system resources or memory belonging to the model object are released on return from this function. The model object becomes undefined, and any further attempt to use it is an error.

Set Output Data Array

```
xip_status
xip_fir_v7_0_set_data_sink(
   xip_fir_v7_0* model,
   xip_array_real* data,
   xip_array_complex* cmplx_data
);
xip_status
xip_fir_v7_0_set_data_sink_mpz(
   xip_fir_v7_0* model,
   xip_array_mpz* data,
   xip_array_mpz_complex* cmplx_data
);
```

This function registers an array (the data sink), pointed to by `data` or `cmplx_data`, to push the generated filter output when the `xip_fir_v7_0_data_send` function is called. Only `data` or `cmplx_data` can be set, the other should be set to NULL (or 0).

If the data sink is undefined the filter output must be explicitly pulled using the `xip_fir_v7_0_data_get` function.

The array is automatically sized by the model given the size of the input request. The owner field of `xip_array_<type>` is ignored and forced to 0.

Note: The complex data sink is intended for the Hilbert filter type but is populated for other filter types with im set to 0.

Set Data Handler

```
xip_status
xip_fir_v7_0_set_data_handler(
   xip_fir_v7_0* model,
   xip_array_real_handler data_handler,
   void* handle,
   void* opt_arg
);
```

This function registers a data handler call back function that is called when the output data array is filled following a call to `xip_fir_v7_0_data_send`. The FIR Compiler C model API contains a function, `xip_fir_v7_0_data_send_handler` (see Send DATA Packet), to send data to an instance of the model whose signature matches that of a data handler.

The intention of this facility is to enable multiple instances of the model to be chained together such that only the first and last instance of the chain need to be directly controlled using the `xip_fir_v7_0_data_send` and `xip_fir_v7_0_data_get` functions.

The model only supports data handlers for output data arrays of type `xip_array_real` and the value passed to the (`*xip_array_real_handler`) function for the `data` argument is the value set by the `xip_fir_v7_0_set_data_sink` function. See Type Definitions for details of the data handler function signature. Its arguments are:

1. `data`: A pointer to the `xip_array_real` type containing the data to be processed. The array registered by the `xip_fir_v7_0_set_data_sink` function.
2. `handle`: A void pointer used to point to the next model instance in the filter chain.
3. `opt_arg`: An extra generic argument not currently used by the FIR Compiler C model.

Calculate Output Size

```
xip_status
xip_fir_v7_0_data_calc_size(
    xip_fir_v7_0* model,
    const xip_array_real* data_in,
    xip_array_real* data_out,
    xip_array_complex* cmplx_data_out
)
xip_status
xip_fir_v7_0_data_calc_size_mpz(
    xip_fir_v7_0* model,
    const xip_array_real* data_in,
    xip_array_mpz* data_out,
    xip_array_mpz_complex cmplx_data_out
)
```

This function calculates the size of an output packet/array given the size of the supplied input packet/array.

The `data_out` or `cmplx_data_out` array dimensions are modified to reflect the size of output the model produces, given the `data_in` array. The array dimensions, `dim` and `data_size` element are updated but the function does not allocate more space. Ensure that the correct amount of space is allocated for the `data` element of the array.

Note: Only one of data_out or cmplx_data_out can be set; the other should be set to NULL (or 0).

Model Operation Functions

Send CONFIG Packet

```
xip_status
xip_fir_v7_0_config_send(
    xip_fir_v7_0* model,
    const xip_fir_v7_0_cnfg_packet* cnfg_packet
)
```

This function passes a configuration packet, pointed to by `cnfg_packet` (see Table 4-7), to the model. The model implements an internal fifo/queue. A configuration packet is consumed from the queue for every data packet processed, that is, every call to `xip_fir_v7_0_data_send`.

Note: If the fsel field of the cnfg_packet is not sized correctly the function returns XIP_STATUS_ERROR.

Send RELOAD Packet

```
xip_status
xip_fir_v7_0_reload_send(
    xip_fir_v7_0* model,
    const xip_fir_v7_0_rld_packet* rld_packet
)
```

This function passes a reload packet, pointed to by `rld_packet` (see Table 4-8), to the model.

Note: If the coeff field of the rld_packet is not sized correctly the function returns XIP_STATUS_ERROR.

Send DATA Packet

```
xip_status
xip_fir_v7_0_data_send(
    xip_fir_v7_0* model,
    const xip_array_real* data
);
void
xip_fir_v7_0_data_send_handler(
    const xip_array_real* data,
    void* model,
    void* dummy
);
```

This function sends a new data packet, pointed to by `data`, to the model for processing.

The second version of the function, `xip_fir_v7_0_data_send_handler`, is supplied to be used as a (`*xip_array_real_handler`) call back function, see Set Data Handler for further details.

Input data is provided using the `xip_array_real` structure pointed to by `data` and is expected to be sized:
Number of paths x *Number of interleaved channels* x *number of input vectors*.

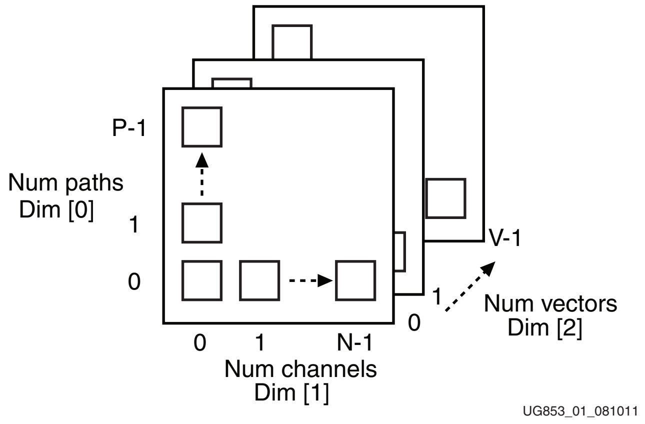


Figure 4-1: **Input and Output Data Packet Structure**

The 3-D structure shown in Figure 4-1 is translated to the 1-D array of the `xip_array_<type> data` element in the order; Paths, Channels, Vectors. The helper functions, `xip_array_<type>_set_chan` (Set Channel) and `xip_array_<type>_get_chan` (Get Channel) implement this translation.

The Advanced Channel implementation requires redundant channel positions to be remapped to higher rate channels. The helper functions, `xip_array_<type>_set_chan` (Set Channel) and `xip_array_<type>_get_chan` (Get Channel), simplify referencing each channel by presenting a flat index for each channel.

Figure 4-1 shows the remapping for three different pattern sequences:

- P4_4 (4 channels x 1/4fs);
- P4_3 (1 channel x 1/2fs and 2 channels x 1/4fs);
- P4_2 (1 channel x 3/4fs and 1 channel x 1/4fs).

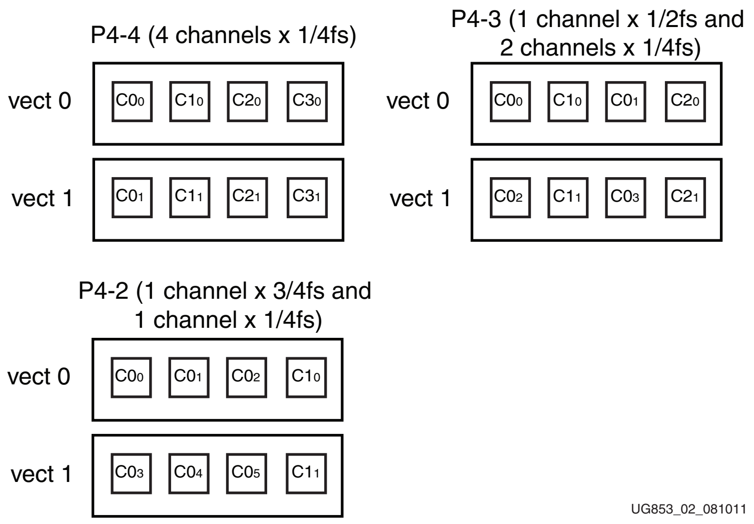


Figure 4-2: **Advanced Channel Pattern Data Packet Remapping**

Get DATA Packet

```
xip_status
xip_fir_v7_0_data_get(
    xip_fir_v7_0* model,
    xip_array_real* data,
    xip_array_complex* cmplx_data
);
xip_status
xip_fir_v7_0_data_get_mpz(
    xip_fir_v7_0* model,
    xip_array_mpz* data,
    xip_array_mpz_complex* cmplx_data
);
```

This function retrieves a filtered data packet from the model into the `xip_array_<type>` pointed to by `data` or `cmplx_data`. Only one of `data` or `cmplx_data` maybe set, the other should be set to NULL (or 0).

The size of the array `dim[2]` (Figure 4-1) determines how much data is fetched from the model. If the request is greater than available, then the array size is reduced to reflect this. The model does not modify the amount of space allocated. Both versions of the functions maybe used regardless of the internal implementation method of the model. If double data (`xip_real`) is requested when `mpz_t (xip_mpz)` has been used internally by the model the output data is truncated, as per the `mpz_get_d` function (see [Ref 9]).

`mpz_t (xip_mpz)` is an integer type so the model scales the input data and coefficients by their specified fractional width to use an integer representation. The output is also supplied as an integer value when `mpz_t` is requested. To correctly interpret the `mpz_t` output the model configuration, returned by the `xip_fir_v7_0_get_config` function (see Get Model Configuration), should be interrogated to determine the output fractional width.

Compiling

Compilation of user code requires access to the `fir_compiler_v7_0_bitacc_cmodel.h` header file and the header file of the MPIR [Ref 8] dependent library, `gmp.h`. The header files should be copied to a location where they are available to the compiler. Depending on the location chosen, the 'include' search path of the compiler might need to be modified.

The `fir_compiler_v7_0_bitacc_cmodel.h` header file includes the MPIR header file, so these do not need to be explicitly included in source code that uses the C model. When compiling on Windows, the symbol `NT` must be defined, either by a compiler option, or in user source code before the `fir_compiler_v7_0_bitacc_cmodel.h` header file is included.

Linking

To use the C model the user executable must be linked against the correct libraries for the target platform.

Note: The C model uses the MPIR library. Pre-compiled MPIR libraries are provided with the C model. It is also possible to use GMP or MPIR, libraries from other sources, for example, compiled from source code. For details, see Dependent Libraries.

Linux

The executable must be linked against the following shared object libraries:

- `libgmp.so.7`
- `libIp_fir_compiler_v7_0_bitacc_cmodel.so`

Using GCC, linking is typically achieved by adding the following command line options:

```
-L. -lgmp -lIp_fir_compiler_v7_0_bitacc_cmodel
```

This assumes the shared object libraries are in the current directory. If this is not the case, the `-L.` option should be changed to specify the library search path to use.

Using GCC, the provided example program `run_bitacc_cmodel.c` can be compiled and linked using the following command:

```
gcc run_bitacc_cmodel.c -o run_bitacc_cmodel -I. -L. -lgmp
-lIp_fir_compiler_v7_0_bitacc_cmodel
```

Note: The C model dynamically links to gmpxx.so.1 and therefore must be visible to the model while running.

Windows

The executable must be linked against the following dynamic link libraries:

- `libgmp.dll`
- `libIp_fir_compiler_v7_0_bitacc_cmodel.dll`

Depending on the compiler, the import libraries might also be required:

- `libgmp.lib`
- `libIp_fir_compiler_v7_0_bitacc_cmodel.lib`

Using Microsoft Visual Studio, linking is typically achieved by adding the import libraries to the Additional Dependencies entry under the Linker section of Project Properties.

Example

The `run_bitacc_cmodel.c` file contains example code to show the basic operation of the C model in various configurations.

MATLAB Interface

A MEX function and MATLAB® software class are provided to simplify the integration with MATLAB. The MEX function provides a low-level wrapper around the underlying C model, while the class file provides a convenient interface to the MEX function.

Compiling

Source code for a MATLAB MEX function is provided. This can be compiled within MATLAB by changing to the directory that contains the code and running the `make_fir_compiler_v7_0_bitacc_mex.m` script.

Installation

To use the MEX function, the compiled MEX function must be present on the MATLAB search path. This can be achieved in either of two ways:

1. Add the directory where the compiled MEX function is located to the MATLAB search path (see the MATLAB addpath function)

   or

2. Copy the files to a location already on the MATLAB search path.

As with all uses of the C model, the correct C model libraries also need to be present on the platform library search path (that is, PATH or LD_LIBRARY_PATH).

MATLAB Class Interface

The `@fir_compiler_v7_0_bitacc` class handles the create/destroy semantics on the C model. The class provides objects for each of the data, configuration and control structures, defined for the C model and previously described in Structures. All structure elements have MATLAB type double. MATLAB arrays are used with the mapping of types as in Table 4-9.

Table 4-9: **MATLAB to C Model Type Mapping**

C Model Type	MATLAB Type
xip_uint32	uint32

Table 4-9: **MATLAB to C Model Type Mapping**

C Model Type	MATLAB Type
xip_complex	complex double
xip_real	double

The class provides the methods:

Constructor

```
[model]=fir_compiler_v7_0_bitacc
[model]=fir_compiler_v7_0_bitacc(config)
[model]=fir_compiler_v7_0_bitacc(field, value [, field,value]*)
```

Note: * indicates an optional parameter.

The first version of the function call constructs a model object using the default configuration.

The second version constructs a model object from a structure that specifies the configuration parameter values to use.

The third version is the same as the second, but allows the configuration to be specified as a series of (parameter name, value) pairs rather than a single structure.

The names and valid values of configuration parameters are identical to those previously described for the C model in Structures.

The MATLAB configuration structure can contain an additional element, `PersistentMemory`. When the element is set to TRUE the internal data memory state of the model is retained following a call to the Filter function. Otherwise, the model is Reset after the filtered data is returned. `PersistentMemory` is set to FALSE by default.

Get Version

```
[version]=get_version(model)
```

This method returns the version string of the C model library used.

Get Configuration

```
[config]=get_configuration(model)
```

This method returns the current parameters structure of a model object. If the model object is empty, the method returns the default configuration. If the model object has been created, the method returns the configuration parameters that were used to create it.

Reset

```
[model]=reset(model)
```

This function resets the model, see Reset Model Object for further details.

Send CONFIG Packet

```
[model]=config_send(model,cnfg_packet)
```

This function passes a configuration packet (see Table 4-7), to the model. See Send CONFIG Packet for further details.

Send RELOAD Packet

```
[model]=reload_send(model,rld_packet)
```

This function passes a reload packet (see Table 4-8), to the model. See Send RELOAD Packet for further details.

Filter

```
[model,data_out]=filter(model,data_in)
```

This function passes a MATLAB double array to the model and returns the filtered output. `data_in` can be a 1, 2 or 3 dimensional array:

- A 1-D array is only supported by a single channel, single path filter configuration.
- A 2-D array is only supported by a multichannel, single path filter configuration.
- All filter configurations support a 3-D array.

See Send DATA Packet and Figure 4-1 for further details on the data array structure.

Example

The `run_fir_compiler_v7_0_bitacc_mex.m` file contains a MATLAB script with an example of how to run the C model using the MEX function.

To run the sample script:

1. Compile the MEX function with the `make_fir_compiler_v7_0_bitacc_mex.m` script (see Compiling).
2. Install the MEX function (see Installation).
3. Execute the `run_fir_compiler_v7_0_bitacc_mex.m` script.

Dependent Libraries

The C model uses MPIR libraries. Pre-compiled MPIR libraries are provided with the C model, using the following versions of the libraries:

- MPIR 2.2.1

Because MPIR is a compatible alternative to GMP, the GMP library can be used in place of MPIR. It is possible to use GMP or MPIR libraries from other sources, for example, compiled from source code.

GMP and MPIR in particular contain many low level optimizations for specific processors. The libraries provided are compiled for a generic processor on each platform, not using optimized processor-specific code. These libraries work on any processor, but run more slowly than libraries compiled to use optimized processor-specific code. For the fastest performance, compile libraries from source on the machine on which you run the executables.

Source code and compilation scripts are provided for the version of MPIR that were used to compile the provided libraries. Source code and compilation scripts for any version of the libraries can be obtained from the GMP [Ref 9] and MPIR [Ref 8] web sites.

Note: If compiling MPIR using its configure script (for example, on Linux platforms), use the --enable-gmpcompat option when running the configure script. This generates a libgmp.so library and a gmp.h header file that provide full compatibility with the GMP library.

Customizing and Generating the Core

This chapter includes information about using Xilinx tools to customize and generate the core in the Vivado™ Design Suite environment.

GUI

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 shows the core pinout.

Tab 2: Freq. Response

The Freq. Response tab displays the filter frequency response (magnitude only).

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 Coefficient Quantization for more details.

The filter gain displayed is for a single rate implementation and does not take into account the zero insertion between output samples in the up-sampling processes in a interpolating filter. Therefore, following the zero insertion the average filter gain is reduced by the up-sampling rate.

- **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).

You 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, selected interleaved data channel sequences 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. 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, you can increase the number coefficients used to specify the filter at little or no cost in resource usage.

The AXI4-Port Structure pane describes fields internal to the AXI4-Stream ports and the number of bus transactions the core expects. This pane allows you to see how individual fields map to the indexes of the compound port as a whole.

The Interleaved Channel Pattern pane displays the enumerated list of channel sequences that have been selected. The enumerated value is used to select the desired pattern using the `chanpat` field of the `s_axis_config_tdata` port. See CONFIG Channel for details of the CONFIG channel.

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.

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 using the *Reload Coefficient File* parameter. It uses the same COE format as the *Coefficient File* parameter. See Filter Coefficient Data 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.

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 Coefficient Reload for more details.

Filter Options Tab

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 using 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 your requirements. See Coefficient Quantization 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.
- **Use Reloadable Coefficients**: When the *Reloadable* option is selected, a coefficient reload interface is provided on the core.

Filter Specification

- **Filter Type**: Five filter types are supported: Single-rate FIR, Interpolating FIR, Decimating FIR, Hilbert transform and Interpolated FIR.
- **Inferred Coefficient Structure(s)**: Displays the coefficient structures, that can be supported for the selected filter type, detected by the GUI in the specified coefficients. The inferred coefficient structure (the first item in the list) can be overridden using the Coefficient Structure parameter later in the GUI. Supported coefficient structures are: Non-symmetric, Symmetric, Negative Symmetric, Half-band and Hilbert.

The combination of Filter Type, Coefficient Structure and Filter Architecture selects the implementation used by the core.

- **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.
- **Zero Packing Factor:** This field is applicable to the interpolated filter only. The zero packing factor specifies the number of 0s inserted between the coefficient data

specified by you. A zero packing factor of *k* inserts *k*-1 zeros between the supplied coefficient values.

Channel Specification Tab

Interleaved Channel Specification

- **Channel Sequence**: This field selects between *Basic* and *Advanced* interleaved data channel sequences. The *Basic* implementation processes interleaved data channels starting at channel 0 incrementing in steps of 1 to Number of Channels - 1. The *Advanced* implementation can processes interleaved data channels in multiple pre-defined sequences. The desired sequences are specified using the Sequence ID List parameter. The CONFIG channel is used to select the active channel sequence. See Interleaved Data Channel Filters for more details.
- **Number of Channels:** The maximum number of interleaved data channels to be processed by the filter. For *Advanced* channel sequences this parameter specifies the channel sequence length, which also specifies the maximum number of interleave data channels.
- **Select Sequence:** This field can be used to select which of the supported channel sequences are to be implemented. Selecting *All* populates the Sequence ID List with all the available channel sequences. Similarly, *Clear All* removes all the sequences apart from default first channel sequence supported. Selecting a specific channel sequence toggles its entry in the Sequence ID List parameter.
- **Sequence ID List:** A comma delimited list that specifies which channel sequences are implemented by the core. The Interleaved Channel Pattern pane of Implementation Tab, Tab 3: Implementation, displays the enumerated list of selected patterns. The Select Sequence parameter can be used to populate the list. See Interleaved Data Channel Filters for details of the supported channel sequences.

Parallel Channel Specification

- **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.

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, you can specify 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, you can specify the integer number of clock cycles between input samples.

- **Input 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 Tab

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

Coefficient Options

- **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 Coefficient Quantization 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 your 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 Best Precision Fractional Length 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 Coefficient Quantization for more information.
- **Coefficient Structure:** Five coefficient structures are supported: Non-symmetric, Symmetric, Negative Symmetric, Half-band and Hilbert. 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 you to specify directly. If Hilbert has been specified as the Filter Type then Hilbert is forced for Coefficient Structure.

Datapath Options

- **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 you 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 Tab

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

- **Filter Architecture:** Two filter architectures are supported: Systolic Multiply-Accumulate and Transpose Multiply-Accumulate.
- **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.). When *Advanced* interleaved channels have been specified a further two options are available: Speed(Control only) and Speed(Data only).

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. Inappropriate use of forcing the RAM selection to be either *Block* or *Distributed* can lead to inefficient resource usage.

RECOMMENDED: The default *Automatic* mode is recommended for most implementations.

- **Data Buffer Type:** Specifies the type of RAM to be used to store data within a MAC element. You 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. You 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. You 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. You 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. You 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 you specify this type of memory directly only if you really need to steer the core away from using a particular memory resource (for example, if you are short of block RAMs in your overall design).

DSP Slice Column Options

The Vivado IDE displays the number of independent DSP chains, and their length, required to build the specified filter configuration.

- **Multi-column Support:** Implementations of large high speed filters might require chaining of DSP slice elements across multiple DSP columns. Where applicable (the feature is only enabled for multi-column devices), you 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 *Automatic* (user specifies the length of each column). Multiple Column Filter Implementation describes the multi-column implementation in more detail.
- **Device Column Lengths**: Displays the column length pattern in a comma delimited list for the selected project device.
- **Available Column Lengths**: Displays the column length pattern available for a single DSP chain. The GUI reduces the Device Columns Lengths given the number of independent DSP chains required by the filter configuration. The generated column pattern considers the Optimization Goal specified.
- **Column Configuration:** Specifies the individual column lengths, in a comma delimited list, that implement a single DSP chain. When *Automatic* has been selected, the column lengths are determined by the GUI starting with the first column in the available column pattern. When *Custom* is selected, you can specify the desired column pattern. The number and length of the columns cannot exceed the available column pattern and the column lengths must sum to the DSP chain length. When the available columns have various 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 (Non-symmetric filter implementations only), 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 you to specify.

Interface Tab

Data Channel Options

- **TLAST**: TLAST can either be **Not Required**, **Vector Framing** or **Packet Framing**. Selecting **Not Required** means that the core does not have the port; selecting **Vector Framing** means that TLAST is expected to denote the last sample of an interleaved cycle of data channels; selecting **Packet Framing** means that 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 FPGA logic 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:** range 1 to 256 bits.

See TUSER Options of the Input and Output DATA Channels 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 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 and CONFIG Channel for more details.

Control Signals

- **aclken:** Determines if the core has the `aclken` pin.
- **aresetn:** Determines if the core has the `aresetn` pin.

IMPORTANT: *`aresetn` is active-Low and when asserted, it should be asserted for a minimum of two 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 FPGA logic 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 output data of the core. See Resets and Input and Output DATA Channels TUSER Options for more details.

Summary Tab

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.

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 Vivado Integrated Design Environment (IDE). See GUI 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 Options Tab of the 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 Multiple Coefficient Sets.

Tab 2: Channel Specification

- **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, you 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).

- **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 Options Tab for information about the other parameters on this tab.

Tab 3: Implementation

The Implementation tab is used to define implementation options; see the Implementation Tab of the Vivado IDE 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 Tab for information about the other parameters on this tab.

Tab 5: Interface

See Detailed Implementation Tab for the corresponding IDE 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.

Chapter 6

Constraining the Core

There are no constraints associated with this core.

Detailed Example Design

This chapter contains information about the provided test bench in the Vivado™ Design Suite environment.

Demonstration Test Bench

When the core is generated using the Vivado IP catalog, a demonstration test bench is optionally 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 Vivado Design Suite 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.

After generating the demonstration test bench it must be set as the top-level simulation object. This is done using the *Sources* pane. Expand the Simulation sources folder and under the core instance the test bench object is visible as `tb_<component_name>`. Select the file, right-click and select **Set as Top**. Simulation can now be launched and the test bench is used to drive the core instance.

The Demonstration Test Bench in Detail

The demonstration test bench performs the following tasks:

- Instantiates the core
- Generates a clock signal
- Drives the input signals of the core to demonstrate core features
- Checks that the output signals of the core obey AXI4 protocol rules (data values are not checked to keep the test bench simple)
- Provides signals showing the separate fields of AXI4 TDATA and TUSER signals

The demonstration test bench drives the input signals of the core 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 AXI4 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 advanced interleaved data channel sequences: select a different channel pattern; drive an impulse 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 inputs with different data or to perform different operations.

All operations performed by the demonstration test bench to drive the core 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 AXI4 signals to adhere to the AXI4 protocol and keep to the configured input sample rate of the core. The *drive_impulse* procedure drives an impulse input, with enough zero-valued samples to allow time for the impulse response to emerge on the output data channel of the core. 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.

Simulation

To simulate the core, generate the core simulation model and demonstration test bench. Ensure that the demonstration test bench is the top level entity in the simulation options. Then select ‘Run Simulation’ in the Vivado IDE.

For full instructions on simulating your core, see *UG900, Vivado Design Suite User Guide: Logic Simulation* [Ref 10].

Migrating

This appendix describes migrating from older versions of the IP to the current IP release.

For information on migrating to the Vivado™ Design Suite, see UG911, *Vivado Design Suite Migration Methodology Guide* [Ref 12].

Parameter Changes

There are no parameter, port, or latency changes between v7.0 and v6.3.

Updating from FIR Compiler v6.0, v6.1 and v6.2

Multi-Column Support: *Disabled* is now deprecated. Automatic upgrade replaces this with a value of *Automatic*.

There are no other parameter, port or latency changes between v7.0 and v6.2, v6.1 and v6.0 of the FIR Compiler, only additional parameters. The additional parameters are: Channel Sequence, Select Pattern and Pattern List. See GUI for details.

Updating from FIR Compiler v5.0

Parameter Changes

The Vivado core update functionality can be used to import an existing XCO file from v5.0 and upgrade to FIR Compiler v7.0, 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 v7.0 has additional AXI4-Stream parameters. The following table shows the changes in parameters from v5.0 to v7.0.

Table A-1: **Parameter Changes from v5.0 to v7.0**

Version v5.0	Version 7.0	Notes
component_name	component_name	Unchanged
CoefficientSource	CoefficientSource	Unchanged
CoefficientVector	CoefficientVector	Unchanged
Coefficient_File	Coefficient_File	Unchanged

Table A-1: **Parameter Changes from v5.0 to v7.0 *(Cont'd)***

Version v5.0	Version 7.0	Notes
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
	Channel_Sequence	New to version 7.0. See the Advanced section of Interleaved Data Channel Filters.
Number_Channels	Number_Channels	Unchanged
	Select_Pattern	New to version 7.0. See the Advanced section of Advanced
	Pattern_List	New to version 7.0. See the Advanced section of Advanced.
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
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.

Table A-1: **Parameter Changes from v5.0 to v7.0 *(Cont'd)***

Version v5.0	Version 7.0	Notes
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	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	

Port Changes

Table A-2 details the changes to port naming, additional or deprecated ports and polarity changes from v5.0 to v7.0.

Table A-2: **Port Changes from Version 5.0 to Version 7.0**

Version 5.0	Version 7.0	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

Table A-2: **Port Changes from Version 5.0 to Version 7.0 *(Cont'd)***

Version 5.0	Version 7.0	Notes
FILTER_SEL		Replaced by CONFIG channel. See s_axis_config_t*.
COEF_LD		Replaced by RELOAD channel. See s_axis_reload_t*.
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*
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.
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.

Table A-2: **Port Changes from Version 5.0 to Version 7.0 *(Cont'd)***

Version 5.0	Version 7.0	Notes
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.

Table A-2: **Port Changes from Version 5.0 to Version 7.0 *(Cont'd)***

Version 5.0	Version 7.0	Notes
DOUT_Q_13		Deprecated. Now exists as a field within m_axis_data_tdata.
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.

Functionality Changes

Latency Changes

The latency of FIR Compiler v7.0 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 Vivado IDE Summary page.

Instructions for Minimum Change Migration

To configure the FIR Compiler v7.0 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 FPGA logic resources.

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

Configuration Method (CONFIG 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 FPGA logic resources and matches FIR Compiler v5.0 reset behavior.

Ports

Input / Output Data Channels

ND is mapped to `s_axis_data_tvalid`

RFD is mapped to `s_axis_data_tready`

RDY is mapped to `m_axis_data_tvalid`

Configuration Channel

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

Drive `s_axis_config_tvalid` with the same signal driving `s_axis_data_tvalid`.

Note: For decimation filters `s_axis_config_tvalid` must be driven at the output rate. Configuration packets are consumed at the lower output rate and if supplied at the input rate the Configuration Channel FIFO becomes full and `s_axis_config_tready` is deasserted and input packets ignored.

Tie `s_axis_config_tlast` to 0 and ignore `event_s_axis_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_axis_reload_tdata` bus.

`COEF_DIN` is mapped to `s_axis_reload_tdata` bus

`COEF_WE` is mapped to `s_axis_reload_tvalid`

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

Appendix B

Debugging

This appendix includes details about resources available on the Xilinx Support website and debugging tools. In addition, this appendix provides a step-by-step debugging process and a flow diagram to guide you through debugging the FIR Compiler core.

The following topics are included in this appendix:

- Finding Help on Xilinx.com
- Debug Tools
- Simulation Debug
- Interface Debug

Finding Help on Xilinx.com

To help in the design and debug process when using the FIR Compiler, the Xilinx Support web page (www.xilinx.com/support) contains key resources such as product documentation, release notes, answer records, information about known issues, and links for opening a Technical Support WebCase.

Documentation

This product guide is the main document associated with the FIR Compiler. This guide, along with documentation related to all products that aid in the design process, can be found on the Xilinx Support web page (www.xilinx.com/support) or by using the Xilinx Documentation Navigator.

Download the Xilinx Documentation Navigator from the Design Tools tab on the Downloads page (www.xilinx.com/download). For more information about this tool and the features available, open the online help after installation.

Known Issues

Answer Records include information about commonly encountered problems, helpful information on how to resolve these problems, and any known issues with a Xilinx product.

Answer Records are created and maintained daily ensuring that you have access to the most accurate information available.

Answer Records for this core are listed below, and can also be located by using the Search Support box on the main Xilinx support web page. To maximize your search results, use proper keywords such as

- Product name
- Tool message(s)
- Summary of the issue encountered

A filter search is available after results are returned to further target the results.

Master Answer Record for the FIR Compiler

AR 54502

Contacting Technical 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.

To contact Xilinx Technical Support:

1. Navigate to www.xilinx.com/support.
2. Open a WebCase by selecting the WebCase link located under Support Quick Links.

When opening a WebCase, include:

- Target FPGA including package and speed grade.
- All applicable Xilinx Design Tools and simulator software versions.
- Additional files based on the specific issue might also be required. See the relevant sections in this debug guide for guidelines about which file(s) to include with the WebCase.

Debug Tools

There are tools available to address FIR Compiler design issues. It is important to know which tools are useful for debugging various situations.

Test Bench

The FIR Compiler can generate with a test bench that can be simulated, Information about the test bench can be found in *Chapter 7, Detailed Example Design*.

Vivado Lab Tools

Vivado inserts logic analyzer and virtual I/O cores directly into your design. Vivado Lab Tools allows you to set trigger conditions to capture application and integrated block port signals in hardware. Captured signals can then be analyzed. This feature represents the functionality in the Vivado IDE that is used for logic debugging and validation of a design running in Xilinx FPGAs in hardware.

The Vivado logic analyzer is used to interact with the logic debug LogiCORE IP cores, including:

- ILA 2.0 (and later versions)
- VIO 2.0 (and later versions)

Reference Boards

Various Xilinx development boards support FIR Compiler. These boards can be used to prototype designs and establish that the core can communicate with the system.

- 7 series FPGA evaluation boards
  - KC705
  - KC724

C-Model Reference

See *Chapter 4, C Model Reference* in this guide for tips and instructions for using the provided C-Model files to debug your design.

License Checkers

If the IP requires a license key, the key must be verified. The Vivado design tools have several license check points for gating licensed IP through the flow. If the license check succeeds, the IP can continue generation. Otherwise, generation halts with error. License checkpoints are enforced by the following tools:

- Vivado design tools: Vivado Synthesis, Vivado Implementation, write_bitstream (Tcl command)

IMPORTANT: *IP license level is ignored at checkpoints. The test confirms a valid license exists. It does not check IP license level.*

Simulation Debug

The simulation debug flow for Questa® SIM is shown in Figure B-1. A similar approach can be used with other simulators.

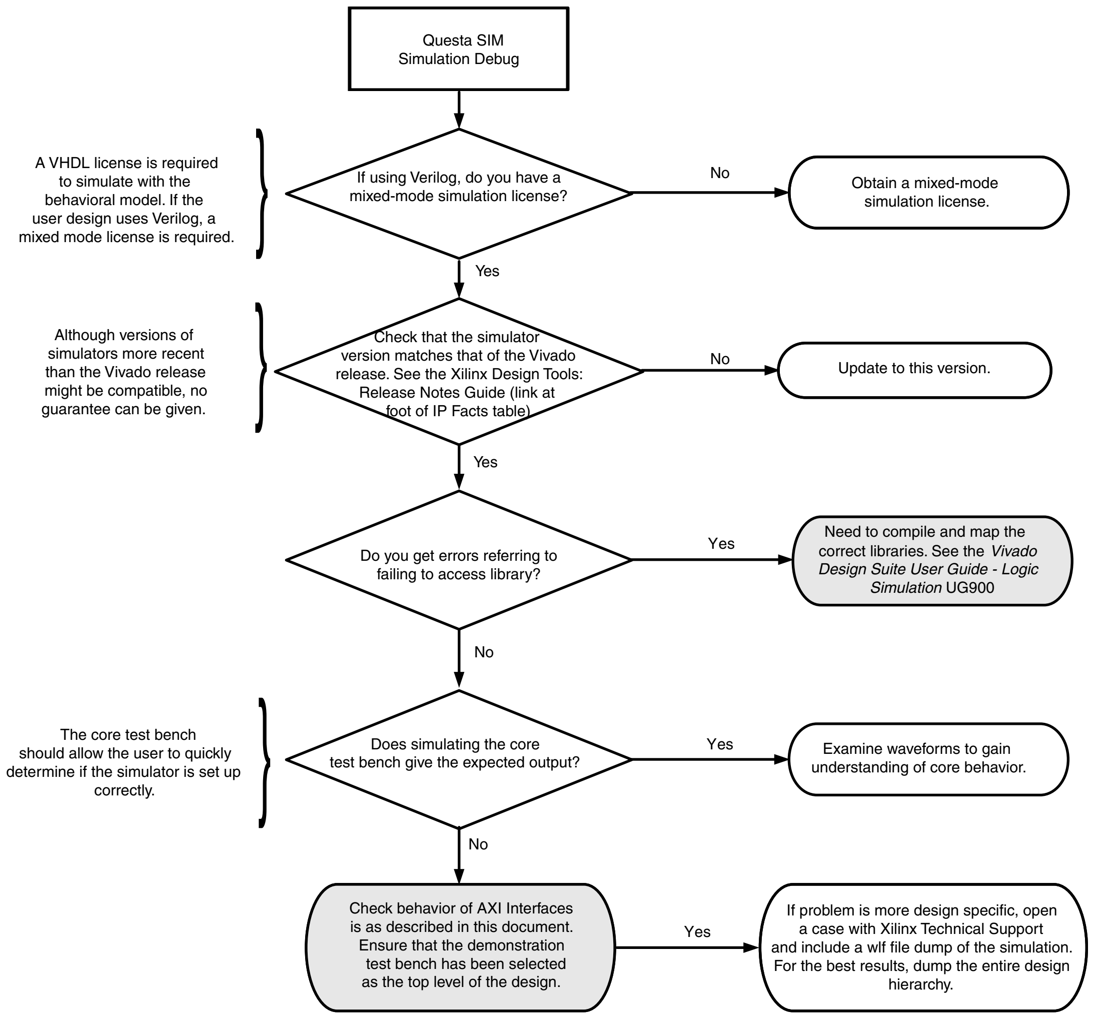

Questa SIM Simulation Debug
A VHDL license is required to simulate with the behavioral model. If the user design uses Verilog, a mixed mode license is required.
If using Verilog, do you have a mixed-mode simulation license?
No
Obtain a mixed-mode simulation license.
Yes
Although versions of simulators more recent than the Vivado release might be compatible, no guarantee can be given.
Check that the simulator version matches that of the Vivado release. See the Xilinx Design Tools: Release Notes Guide (link at foot of IP Facts table)
No
Update to this version.
Yes
Do you get errors referring to failing to access library?
Yes
Need to compile and map the correct libraries. See the Vivado Design Suite User Guide - Logic Simulation UG900
No
The core test bench should allow the user to quickly determine if the simulator is set up correctly.
Does simulating the core test bench give the expected output?
Yes
Examine waveforms to gain understanding of core behavior.
No
Check behavior of AXI Interfaces is as described in this document. Ensure that the demonstration test bench has been selected as the top level of the design.
Yes
If problem is more design specific, open a case with Xilinx Technical Support and include a wlf file dump of the simulation. For the best results, dump the entire design hierarchy.

Figure B-1: **Questa SIM Debug Flow**

Interface Debug

AXI4-Stream Interfaces

If data is not being transmitted or received, check the following conditions:

- If transmit `<`*`interface_name`*`>_tready` is stuck Low following the **`<`*`interface_name`*`>_tvalid`** input being asserted, the core cannot send data.
- If the receive `<`*`interface_name`*`>_tvalid` is stuck Low, the core is not receiving data.
- Check that the `ACLK` inputs are connected and toggling.
- Check that the AXI4-Stream waveforms are being followed (see Figure 3-1).
- Check core configuration.

Additional Resources

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see the Xilinx Support website at:

www.xilinx.com/support.

For a glossary of technical terms used in Xilinx documentation, see:

www.xilinx.com/company/terms.htm.

References

These documents provide supplemental material useful with this product guide:

1. 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
2. P.P. Vaidyanathan, *Multi-Rate Systems and Filter Banks,* Prentice Hall, Englewood Cliffs, New Jersey, 1993.
3. M. E. Frerking, *Digital Signal Processing in Communication Systems,* Van Nostrand Reinhold, New York, 1994.
4. AMBA® AXI4-Stream Protocol Specification (ARM IHI 0051A)
5. Xilinx AXI Reference Guide (UG761)
6. Xilinx Inc., *XtremeDSP Design Manual,* Xilinx Inc., San Jose California, 2004.
7. Mou, Zhi-Jian, *Symmetry Exploitation in Digital Interpolators/Decimators*, IEEE Transactions on Signal Processing, Vol. 44 No. 10, Oct. 1996
8. The Multiple Precision Integers and Rationals (MPIR) Library: www.mpir.org/
9. The GNU Multiple Precision Arithmetic (GMP) Library: gmplib.org/

10. Vivado™ Design Suite user documentation
11. System Generator for DSP User Guide (UG640)
12. Vivado Design Suite Migration Methodology Guide (UG911)
13. Vivado Design Suite User Guide: Designing with IP (UG896)
14. 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.
15. S. A. White, *Applications of Distributed Arithmetic to Digital Signal Processing*, IEEE ASSP Magazine, Vol. 6(3), pp. 4-19, July 1989.
16. 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

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
03/20/2013	1.0	Initial release as a Product Guide; replaces DS795 and UG853. There are no other document changes for this release.

Notice of Disclaimer