Fast Fourier Transform v9.1

LogiCORE IP Product Guide

Vivado Design Suite

PG109 June 17, 2020

XILINX

Table of Contents

Introduction

The Xilinx® LogiCORE™ IP Fast Fourier Transform (FFT) core implements the Cooley-Tukey FFT algorithm, a computationally efficient method for calculating the Discrete Fourier Transform (DFT).

Features

- Forward and inverse complex FFT, run time configurable
- Transform sizes $N = 2^m$, $m = 3 – 16$
- Data sample precision $b_x = 8 – 34$
- Phase factor precision $b_w = 8 – 34$
- Arithmetic types:
  - Unscaled (full-precision) fixed-point
  - Scaled fixed-point
  - Block floating-point
- Fixed-point or floating-point interface
- Rounding or truncation after the butterfly
- Block RAM or Distributed RAM for data and phase-factor storage
- Optional run time configurable transform point size
- Run time configurable scaling schedule for scaled fixed-point cores
- Bit/digit reversed or natural output order
- Optional cyclic prefix insertion for digital communications systems
- Four architectures offer a trade-off between core size and transform time
- Bit accurate C model and MEX function for system modeling available for download

LogiCORE IP Facts Table	
Core Specifics	
Supported Device Family(1)	UltraScale+™ UltraScale™ Zynq®-7000 SoC 7 Series
Supported User Interfaces	AXI4-Stream
Resources	Performance and Resource Utilization web page
Provided with Core	
Design Files	Encrypted RTL
Example Design	Not Provided
Test Bench	VHDL
Constraints File	Not Provided
Simulation Model	Encrypted VHDL C Model
Supported S/W Driver	N/A
Tested Design Flows(2)	
Design Entry	Vivado® Design Suite System Generator for DSP
Simulation	For supported simulators, see the Xilinx Design Tools: Release Notes Guide.
Synthesis	Vivado Synthesis
Support	
Release Notes and Known Issues	Master Answer Record: 54501
All Vivado IP Change Logs	Master Vivado IP Change Logs: 72775
Xilinx Support web page	

Notes:

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

Overview

Core Overview

The FFT core computes an N-point forward DFT or inverse DFT (IDFT) where N can be 2^m, m = 3–16.

For fixed-point inputs, the input data is a vector of N complex values represented as dual b_x-bit twos-complement numbers, that is, b_x bits for each of the real and imaginary components of the data sample, where b_x is in the range 8 to 34 bits inclusive. Similarly, the phase factors b_w can be 8 to 34 bits wide.

For single-precision floating-point inputs, the input data is a vector of N complex values represented as dual 32-bit floating-point numbers with the phase factors represented as 24- or 25-bit fixed-point numbers.

All memory is on-chip using either block RAM or distributed RAM. The N element output vector is represented using b_y bits for each of the real and imaginary components of the output data. Input data is presented in natural order and the output data can be in either natural or bit/digit reversed order. The complex nature of data input and output is intrinsic to the FFT algorithm, not the implementation.

Three arithmetic options are available for computing the FFT:

- Full-precision unscaled arithmetic
- Scaled fixed-point, where you provide the scaling schedule
- Block floating-point (run time adjusted scaling)

The point size N, the choice of forward or inverse transform, the scaling schedule and the cyclic prefix length are run time configurable. Transform type (forward or inverse), scaling schedule and cyclic prefix length can be changed on a frame-by-frame basis. Changing the point size resets the core.

Four architecture options are available: Pipelined Streaming I/O, Radix-4 Burst I/O, Radix-2 Burst I/O, and Radix-2 Lite Burst I/O. For detailed information about each architecture, see Architecture Options.

The FFT is a computationally efficient algorithm for computing a Discrete Fourier Transform (DFT) of sample sizes that are a positive integer power of 2. The DFT $X(k),\ k = 0,\mathrm{K}\ ,N-1$ of a sequence $x(n),\ n = 0,\mathrm{K}\ ,N-1$ is defined as

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-jnk2\pi/N} \quad k = 0,\mathrm{K}\ ,N-1 \qquad \textit{Equation 1-1}$$

where N is the transform size and $j = \sqrt{-1}$. The inverse DFT (IDFT) is given by

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k) e^{jnk2\pi/N} \quad n = 0,\mathrm{K}\ ,N-1 \qquad \textit{Equation 1-2}$$

Algorithm

The FFT core uses the Radix-4 and Radix-2 decompositions for computing the DFT. For Burst I/O architectures, the decimation-in-time (DIT) method is used, while the decimation-in-frequency (DIF) method is used for the Pipelined Streaming I/O architecture. When using Radix-4 decomposition, the N-point FFT consists of $\log_4(N)$ stages, with each stage containing $N/4$ Radix-4 butterflies. Point sizes that are not a power of 4 need an extra Radix-2 stage for combining data. An N-point FFT using Radix-2 decomposition has $\log_2(N)$ stages, with each stage containing $N/2$ Radix-2 butterflies.

The inverse FFT (IFFT) is computed by conjugating the phase factors of the corresponding forward FFT. The FFT core does not implement the $1/N$ scaling for inverse FFT. The scaling is therefore as per forward FFT, simply with conjugated phase factors (twiddle factors).

Licensing and Ordering

This Xilinx® LogiCORE IP module is provided at no additional cost with the Xilinx Vivado® Design Suite under the terms of the Xilinx End User License.

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

Product Specification

Resource Utilization

For details about resource utilization, visit Performance and Resource Utilization.

Port Descriptions

This section describes the core ports as shown in Figure 2-1 and described in Table 2-1.

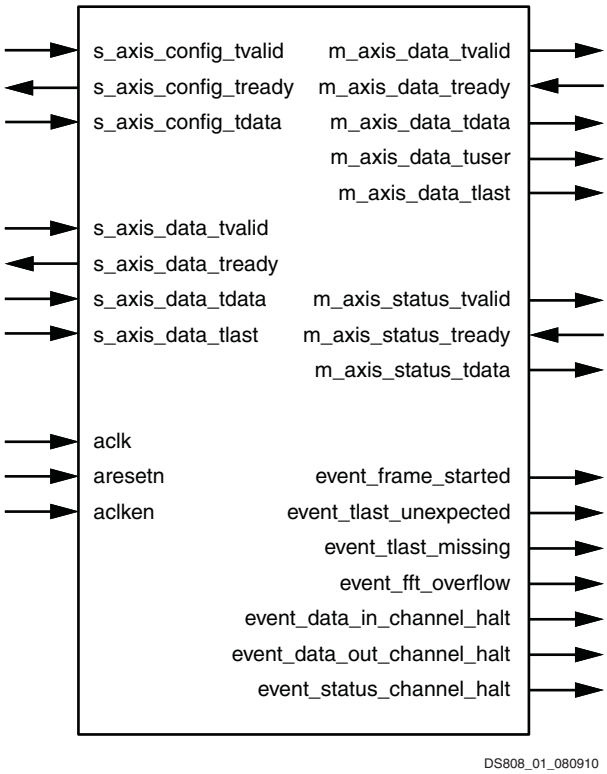

Figure 2-1: **Core Schematic Symbol**

Table 2-1: **Core Signal Pinout**

Name	I/O	Optional	Description
aclk	I	No	Rising-edge clock.
aclken	I	Yes	Active-High clock enable (optional).
aresetn	I	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	I	No	TVALID for the Configuration channel. Asserted by the external master to signal that it is able to provide data.
s_axis_config_tready	O	No	TREADY for the Configuration channel. Asserted by the core to signal that it is ready to accept data.
s_axis_config_tdata	I	No	TDATA for the Configuration channel. Carries the configuration information: CP_LEN, FWD/INV, NFFT and SCALE_SCH. See Run Time Transform Configuration.
s_axis_data_tvalid	I	No	TVALID for the Data Input channel. Used by the external master to signal that it is able to provide data.
s_axis_data_tready	O	No	TREADY for the Data Input channel. Used by the core to signal that it is ready to accept data.
s_axis_data_tdata	I	No	TDATA for the Data Input channel. Carries the unprocessed sample data: XN_RE and XN_IM. See Data Input Channel.
s_axis_data_tlast	I	No	TLAST for the Data Input channel. Asserted by the external master on the last sample of the frame. This is not used by the core except to generate the events event_tlast_unexpected and event_tlast_missing events
m_axis_data_tvalid	O	No	TVALID for the Data Output channel. Asserted by the core to signal that it is able to provide sample data.
m_axis_data_tready	I	No	TREADY for the Data Output channel. Asserted by the external slave to signal that it is ready to accept data. Only present in Non-Realtime mode.
m_axis_data_tdata	O	No	TDATA for the Data Output channel. Carries the processed sample data XK_RE and XK_IM. See Data Output Channel.
m_axis_data_tuser	O	No	TUSER for the Data Output channel. Carries additional per-sample information, such as XK_INDEX, OVFLO and BLK_EXP. See Data Output Channel.
m_axis_data_tlast	O	No	TLAST for the Data Output channel. Asserted by the core on the last sample of the frame.

Table 2-1: **Core Signal Pinout *(Cont'd)***

Name	I/O	Optional	Description
m_axis_status_tvalid	O	No	TVALID for the Status channel. Asserted by the core to signal that it is able to provide status data.
m_axis_status_tready	I	No	TREADY for the Status channel. Asserted by the external slave to signal that it is ready to accept data. Only present in Non-Realtime mode
m_axis_status_tdata	O	No	TDATA for the Status channel. Carries the status data: BLK_EXP or OVFLO. See Status Channel.
event_frame_started	O	No	Asserted when the core starts to process a new frame. See event_frame_started.
event_tlast_unexpected	O	No	Asserted when the core sees s_axis_data_tlast High on a data sample that is not the last one in a frame. See event_tlast_unexpected.
event_tlast_missing	O	No	Asserted when s_axis_data_tlast is Low on the last data sample of a frame. See event_tlast_missing.
event_fft_overflow	O	No	Asserted when an overflow is seen in the data samples being unloaded from the Data Output channel. Only present when overflow is a valid option. See event_fft_overflow.
event_data_in_channel_halt	O	No	Asserted when the core requests data from the Data Input channel and none is available. See event_data_in_channel_halt.
event_data_out_channel_halt	O	No	Asserted when the core tries to write data to the Data Output channel and it is unable to do so. Only present in Non-Realtime mode. See event_data_out_channel_halt.
event_status_channel_halt	O	No	Asserted when the core tries to write data to the Status channel and it is unable to do so. Only present in Non-Realtime mode. See event_status_channel_halt.

Notes:

1. All AXI4-Stream port names are lowercase, but for ease of visualization, uppercase is used in this document when referring to port name suffixes, such as TDATA or TLAST.

Designing with the Core

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

Clocking

The core uses a single clock, called **aclk**. All input and output interfaces and internal state are subject to this single clock.

aclken (Clock Enable)

If the clock enable (**aclken**) pin is present on the core, driving the pin Low pauses the core in its current state. All logic within the core is paused. Driving the **aclken** pin High allows the core to continue processing. Note that **aclken** can reduce the maximum frequency at which the core runs.

Resets

aresetn (Synchronous Clear)

If the **aresetn** pin is present on the core, driving the pin Low causes all output pins, internal counters, and state variables to be reset to their initial values. The initial values described in Table 3-1 are also the default values that the circuit adopts on power-on, regardless of whether the core is configured for **aresetn** or not. All pending load processes, transform calculations, and unload processes stop and are re-initialized. NFFT is set to the largest FFT point size permitted (the Transform Length value set in the Vivado Integrated Design Environment (IDE)). The scaling schedule is set to 1/N. For the Radix-4 Burst I/O and Pipelined Streaming I/O architectures with a non-power-of-four point size, the last stage has a scaling of 1, and the rest have a scaling of 2. See Table 3-1.

Table 3-1: **Synchronous Clear Reset Values for Configuration Information**

Signal	Initial/Reset Value
NFFT	maximum point size = *N*
FWD_INV	Forward = 1
SCALE_SCH	1/*N* [10 10... 10] for Radix-4 Burst I/O or Pipelined Streaming I/O architectures when *N* is a power of 4. [01 10... 10] for Radix-4 Burst I/O or Pipelined Streaming I/O architectures when *N* is not a power of 4. [01 01... 01] for Radix-2 Burst I/O or Radix-2 Lite Burst I/O architectures

The **aresetn** pin takes priority over **aclken**. If **aresetn** is asserted, reset occurs regardless of the value of **aclken**. A minimum **aresetn** active pulse of two cycles is required, because the signal is internally registered for performance. A pulse of one cycle resets the core, but the response to the pulse is not in the cycle immediately following.

Event Signals

The core provides some real-time non-AXI signals to report information about the core status. These event signals are updated on a clock cycle by clock cycle basis, and are intended for use by reactive components such as interrupt controllers. These signals are not optionally configurable from the IDE, but are removed by synthesis tools if left unconnected.

event_frame_started

This event signal is asserted for a single clock cycle when the core starts to process a new frame. This signal is provided to allow you to count frames and to synchronize the configuration of the core to a particular frame if required.

event_tlast_missing

This event signal is asserted for a single clock cycle when **s_axis_data_tlast** is Low on a last incoming data sample of a frame. This shows a configuration mismatch between the core and the upstream data source with regard to the frame size, and indicates that the upstream data source is configured to a larger point size than the core.

This is only calculated when the core starts processing a frame, so the event can lag the missing **s_axis_data_tlast** by a large number of clock cycles.

event_tlast_unexpected

This event signal is asserted for a single clock cycle when the core sees **s_axis_data_tlast** High on any incoming data sample that is not the last one in a frame. This shows a configuration mismatch between the core and the upstream data source with regard to the frame size, and indicates that the upstream data source is configured to a smaller point size than the core. This is only calculated when the core starts processing a frame, so the event can lag the unexpected High on **s_axis_data_tlast** by a large number of clock cycles.

If there are multiple unexpected highs on **s_axis_data_tlast** for a frame, then this is asserted for each of them.

event_fft_overflow

This event signal is asserted on every clock cycle when an overflow is seen in the data samples being transferred on **m_axis_data_tdata**.

It is only possible to get FFT overflows when scaled arithmetic or single-precision floating-point I/O is used. In all other configurations the pin is removed from the core.

event_data_in_channel_halt

This event is asserted on every cycle where the core needs data from the Data Input channel and no data is available.

- In Realtime Mode the core continues processing the frame even though it is unrecoverably corrupted.
- In Non-Realtime Mode, core processing halts and only continues when data is written to the Data Input channel. The frame is not corrupted.

In both modes the event remains asserted until data is available in the Data Input channel.

event_data_out_channel_halt

This event is asserted on every cycle where the core needs to write data to the Data Output channel but cannot because the buffers in the channel are full. When this occurs, the core processing is halted and all activity stops until space is available in the channel buffers. The frame is not corrupted.

The event pin is only available in Non-Realtime mode.

event_status_channel_halt

This event is asserted on every cycle where the core needs to write data to the Status channel but cannot because the buffers on the channel are full. When this occurs, the core

processing is halted, and all activity stops until space is available in the channel buffers. The frame is not corrupted. The event pin is only available in Non-Realtime mode.

AXI4-Stream Considerations

The conversion to AXI4-Stream interfaces brings standardization and enhances interoperability of Xilinx IP LogiCORE solutions. Other than general control signals such as **aclk**, **aclken** and **aresetn**, and event signals, all inputs and outputs to the core are conveyed on AXI4-Stream channels. A channel always consists of TVALID and TDATA plus additional ports (such as TREADY, TUSER and TLAST) when required and optional fields. Together, TVALID and TREADY perform a handshake to transfer a message, where the payload is TDATA, TUSER and TLAST. The core operates on the operands contained in the TDATA fields and outputs the result in the TDATA field of the output channel.

For further details on AXI4-Stream Interfaces see the *AMBA® AXI4-Stream Protocol Specification* (ARM IHI 0051A) [Ref 1] and the *Xilinx Vivado AXI Reference Guide* (UG1037) [Ref 2].

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.

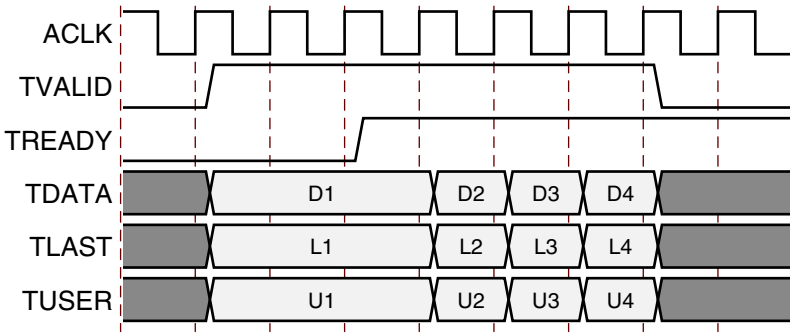

Figure 3-1: **Data Transfer in an AXI-Stream Channel**

AXI Channel Rules

Note that all of the AXI channels follow the same rules:

- All TDATA and TUSER fields are packed in little endian format. That is, bit 0 of a sub-field is aligned to the same side as bit 0 of TDATA or TUSER.
- Fields are not included in TDATA or TUSER unless the core is configured in such a way that it needs the fields to be present. For example, if the core is configured to have a fixed-point size, no bits are allocated to the NFFT field that specifies the point size.
- All TDATA and TUSER vectors are multiples of 8 bits. When all fields in a TDATA or TUSER vector have been concatenated, the overall vector is padded to bring it up to an 8-bit boundary.

Configuration Channel

Table 3-2 shows the Configuration channel pinout.

Table 3-2: **Configuration Channel Pinout**

Port Name	Port Width	I/O	Description
s_axis_config_tdata	Variable. See the Vivado® IDE when configuring the core.	I	Carries the configuration information, **CP_LEN**, **FWD/INV**, **NFFT** and **SCALE_SCH**. See Run Time Transform Configuration for more information
s_axis_config_tvalid	1	I	Asserted by the external master to signal that it is able to provide data.
s_axis_config_tready	1	O	Asserted by the core to signal that it is able to accept data.

TDATA Fields

The Configuration channel (**s_axis_config**) is an AXI channel that carries the fields in Table 3-3 in its TDATA vector.

Table 3-3: **Configuration Channel TDATA Fields**

Field Name	Width	Padded	Description
NFFT	5	Yes	Point size of the transform: NFFT can be the size of the maximum transform or any smaller point size. For example, a 1024-point FFT can compute point sizes 1024, 512, 256, and so on. The value of NFFT is $\log_2$ (point size). This field is only present with run time configurable transform point size. For more information, see Transform Size.
CP_LEN	$\log_2$ (maximum point size)	Yes	Cyclic prefix length: The number of samples from the end of the transform that are initially output as a cyclic prefix, before the whole transform is output. CP_LEN can be any number from zero to one less than the point size. This field is only present with cyclic prefix insertion. For more information, see Cyclic Prefix Insertion.
FWD_INV	1 bit per FFT data channel	No	Indicates if a forward FFT transform or an inverse FFT transform is performed. When FWD_INV = 1, a forward transform is computed. If FWD_INV = 0, an inverse transform is computed. The field contains 1 bit per FFT data channel, bit 0 (LSB) representing channel 0, bit 1 representing channel 1, etc. For more information, see Forward/Inverse and Scaling Schedule.
SCALE_SCH	$2\times ceil\left(\frac{NFFT}{2}\right)$ for Pipelined Streaming I/O and Radix-4 Burst I/O architectures or $2 \times NFFT$ for Radix-2, Burst I/O and Radix-2 Lite Burst I/O architectures where NFFT is $\log_2$ (maximum point size) or the number of stages	No	Scaling schedule: For Burst I/O architectures, the scaling schedule is specified with two bits for each stage, with the scaling for the first stage given by the two LSBs. The scaling can be specified as 3, 2, 1, or 0, which represents the number of bits to be shifted. An example scaling schedule for N =1024, Radix-4 Burst I/O is [1 0 2 3 2] (ordered from last to first stage). For N =128, Radix-2 Burst I/O or Radix-2 Lite Burst I/O, one possible scaling schedule is [1 1 1 1 0 1 2] (ordered from last to first stage). For Pipelined Streaming I/O architecture, the scaling schedule is specified with two bits for every pair of Radix-2 stages, starting at the two LSBs. For example, a scaling schedule for N = 256 could be [2 2 2 3]. When N is not a power of 4, the maximum bit growth for the last stage is one bit. For example, [0 2 2 2 2] or [1 2 2 2 2] are valid scaling schedules for N = 512, but [2 2 2 2 2] is invalid. For this transform length the two MSBs of SCALE_SCH can only be 00 or 01. This field is only available with scaled arithmetic (not unscaled, block floating-point or single precision floating-point). For more information, see Transform Size.

All fields with padding should be extended to the next 8-bit boundary if they do not already finish on an 8-bit boundary. The core ignores the value of the padding bits, so they can be driven to any value. Connecting them to constant values might help reduce device resource usage.

TDATA Format

The configuration fields are packed into the **`s_axis_config_tdata`** vector in the following order (starting from the LSB):

1. (optional) NFFT plus padding
2. (optional) CP_LEN plus padding
3. FWD/INV
4. (optional) SCALE_SCH

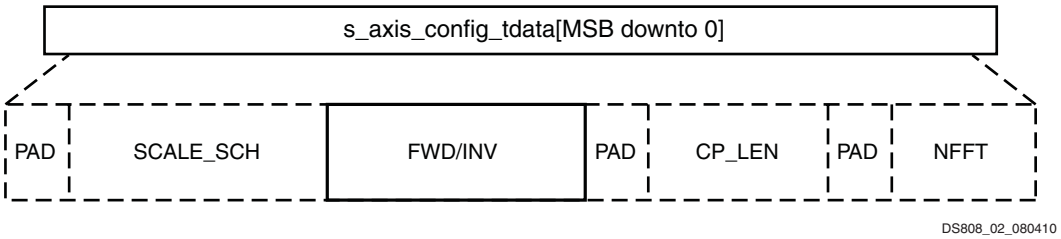

Figure 3-2: **Configuration Channel TDATA (s_axis_config_tdata) Format**

Optional fields are shown as dotted.

TDATA Example

A core has a configurable transform size with a maximum size of 128 points, cyclic prefix insertion and 3 FFT channels. The core needs to be configured to do an 8 point transform, with an inverse transform performed on channels 0 and 1, and a forward transform performed on channel 2. A 4 point cyclic prefix is required. The fields take on the values in Table 3-4.

Table 3-4: **Configuration Channel TDATA Example**

Field Name	Padding	Value	Notes
NFFT	000	00011	3 gives an 8-point FFT
CP_LEN	0	1000000	The core selects the top NFFT bits of the CP_LEN field (not including the padding) to determine the cyclic prefix length. To get a Cyclic Prefix length of 4, and NFFT is 3, the field has to be set to 64
FWD_INV	N/A	100	Channel 2: Forward Channel 1: Inverse Channel 0: Inverse

This gives a vector length of 19 bits. As all AXI channels must be aligned to byte boundaries, 5 padding bits are required, giving an **`s_axis_config_tdata`** length of 24 bits.

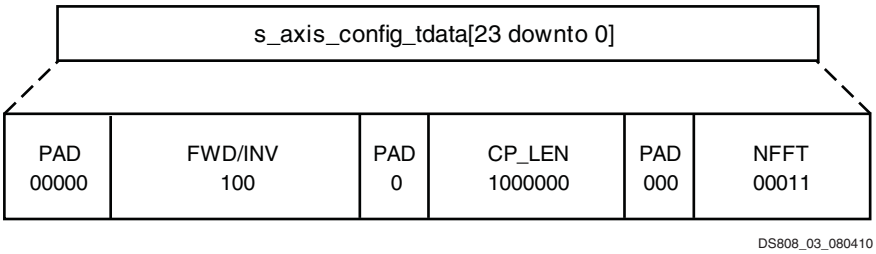

Figure 3-3: **Configuration Channel TDATA Example**

Data Input Channel

The Data Input channel contains the real and imaginary sample data to be transformed.

Pinout

Table 3-5: **Data Input Channel Pinout**

Port Name	Port Width	I/O	Description
s_axis_data_tdata	Variable. See the Vivado IDE when configuring the core.	I	Carries the sample data: XN_RE and XN_IM
s_axis_data_tvalid	1	I	Asserted by the upstream master to signal that it is able to provide data
s_axis_data_tlast	1	I	Asserted by the upstream master on the last sample of the frame. This is not used by the core except to generate the events: event_tlast_unexpected event_tlast_missing events
s_axis_data_tready	1	O	Used by the core to signal that it is ready to accept data

TDATA Fields

The Data Input channel (**`s_axis_data`**) is an AXI channel that carries the fields in Table 3-6 in its TDATA vector.

Table 3-6: **Data Input Channel TDATA Fields**

Field Name	Width	Padded	Description
XN_RE	b_{xn}	Yes	Real component (b_{xn} = 8 - 34) in twos complement or single precision floating-point format.
XN_IM	b_{xn}	Yes	Imaginary component (b_{xn} = 8 - 34) in twos complement or single precision floating-point format.

All fields with padding should be extended to the next 8-bit boundary if they do not already finish on an 8-bit boundary. The core ignores the value of the padding bits, so they can be driven to any value. Connecting them to constant values can help reduce device resource usage.

These fields are then repeated for each channel in the design.

TDATA Format

The data fields are packed into the **`s_axis_data_tdata`** vector in the following order (starting from the LSB):

1. XN_RE plus padding for channel 0
2. XN_IM plus padding for channel 0
3. (optional) XN_RE plus padding for channel 1
4. (optional) XN_IM plus padding for channel 1
5. (optional) XN_RE plus padding for channel 2
6. (optional) XN_IM plus padding for channel 2
7. etc., up to channel 11

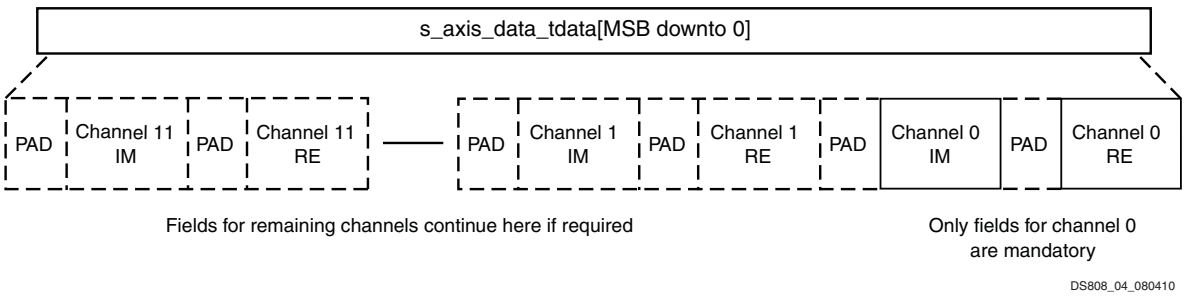

Figure 3-4: **Data Input Channel TDATA (s_axis_data_tdata) Format**

Optional fields are shown as dotted.

TDATA Example

The core has been configured to have two FFT data channels with 12-bit data. Channel 0 has the following sample value:

- Re = 0010 1101 1001
- IM = 0011 1110 0110

Channel 1 has the following sample value:

- Re = 0111 0000 0000
- IM = 0000 0000 0000

The fields take on the values in Table 3-7.

Table 3-7: **Data Input Channel TDATA Example**

Field Name	Padding	Value
XN_RE (channel 0)	0000	0010 1101 1001
XN_IM (channel 0)	0000	0011 1110 0110
XN_RE (channel 1)	0000	0111 0000 0000
XN_IM (channel 1)	0000	0000 0000 0000

This gives a vector length of 64 bits.

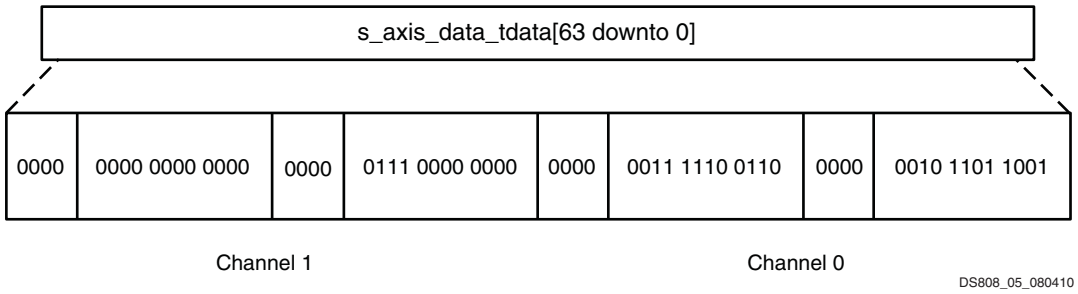

Figure 3-5: **Data Input Channel TDATA Example**

Data Output Channel

The Data Output channel contains the real and imaginary results of the transform, which are carried on TDATA. In addition, TUSER carries per-sample status information relating to the sample data on TDATA. This status information is intended for use by downstream slaves that directly process data samples. It cannot get out of synchronization with the data as it is transferred in the same channel. The following information is classed as per-sample status:

1. **XK_INDEX**
2. Block Exponent (**BLK_EXP**) for each FFT channel
3. Overflow (**OVFLO**) for each FFT channel

Pinout

Table 3-8: **Data Output Channel Pinout**

Port Name	Port Width	I/O	Description
m_axis_data_tdata	Variable. See the Vivado IDE when configuring the core.	O	Carries the sample data: **XK_RE** and **XK_IM**.
m_axis_data_tuser	Variable. See the Vivado IDE when configuring the core.	O	Carries additional per-sample: **XK_RE** and **XK_IM**.
m_axis_data_tvalid	1	O	Asserted by the core to signal that it is able to provide sample data
m_axis_data_tlast	1	O	Asserted by the core on the last sample of the frame
m_axis_data_tready	1	I	Asserted by the external slave to signal that it is ready to accept data

TDATA Fields

The Data Output channel (**m_axis_data**) is an AXI channel that carries the fields in Table 3-9 in its TDATA vector.

Table 3-9: **Data Output Channel TDATA Fields**

Field Name	Width	Padded	Description
XK_RE	b_{xk}	Yes - sign extended	Output data: Real component in twos complement or floating-point format. (For scaled arithmetic and block floating-point arithmetic, $b_{xk} = b_{xn}$. For unscaled arithmetic, $b_{xk} = b_{xn} + \log_2$ (maximum point size) +1. For single precision floating-point $b_{xk} = 32$).
XK_IM	b_{xk}	Yes - sign extended	Output data: Imaginary component in twos complement or single precision floating-point format. (For scaled arithmetic and block floating-point arithmetic, $b_{xk} = b_{xn}$. For unscaled arithmetic, $b_{xk} = b_{xn} + \log_2$ (maximum point size) +1. For single precision floating-point $b_{xk} = 32$)

All fields are sign extended to the next 8-bit boundary if they do not already finish on an 8-bit boundary.

These fields are then repeated for each FFT channel in the design.

TDATA Format

The data fields are packed into the **`s_axis_data_tdata`** vector in the following order (starting from the LSB):

1. **`XK_RE`** plus padding for channel 0
2. **`XK_IM`** plus padding for channel 0
3. (optional) **`XK_RE`** plus padding for channel 1
4. (optional) **`XK_IM`** plus padding for channel 1
5. (optional) **`XK_RE`** plus padding for channel 2
6. (optional) **`XK_IM`** plus padding for channel 2
7. etc., up to channel 11

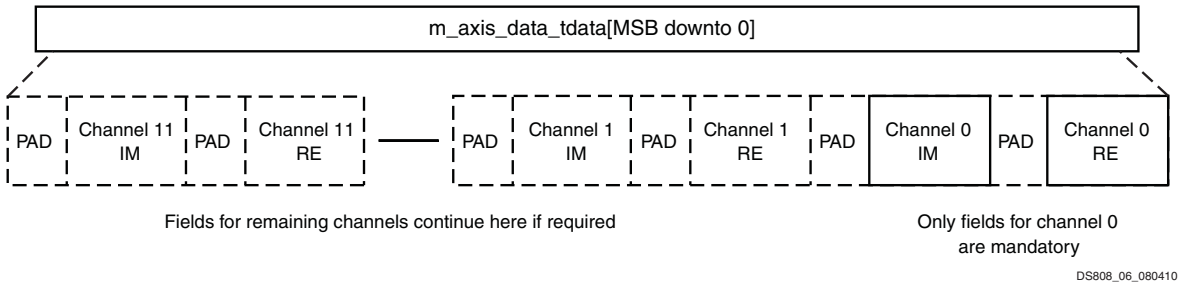

Figure 3-6: **Data Output Channel TDATA (m_axis_data_tdata) Format**

Optional fields are shown as dotted.

TDATA Example

The core has been configured to have two FFT data channels with 12-bit output data. The FFT produces the following sample result for channel 0:

- Re = 0010 1101 1001
- IM = 1011 1110 0110

The FFT produces the following sample result for channel 1:

- Re = 0111 0000 0000
- IM = 1000 0000 0000

The fields take on the values in Table 3-10.

Table 3-10: **Data Output Channel TDATA Example**

Field Name	Padding	Value
XK_RE (channel 0)	0000	0010 1101 1001
XK_IM (channel 0)	1111	1011 1110 0110
XK_RE (channel 1)	0000	0111 0000 0000
XK_IM (channel 1)	1111	1000 0000 0000

This gives a vector length of 64 bits.

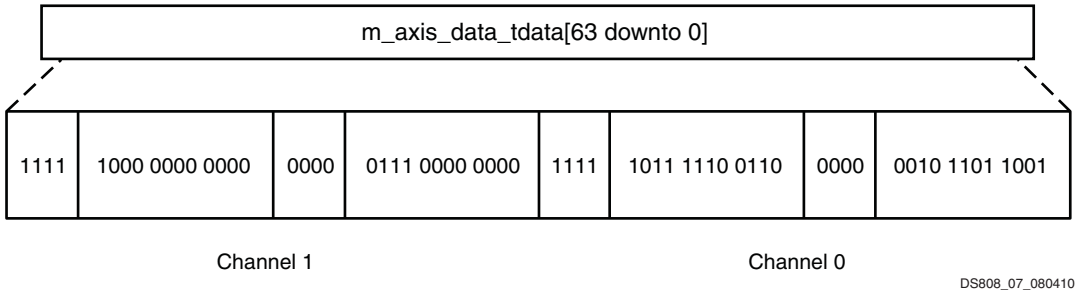

Figure 3-7: **Data Output Channel TDATA Example**

TUSER Fields

The Data Output channel carries the fields in Table 3-11 in its TUSER vector.

Table 3-11: **Data Output Channel TUSER Fields**

Field Name	Width	Padded	Description
XK_INDEX	$\log_2$ (maximum point size)	Yes - zero extended	Index of output data (unsigned 2's complement). This field is optional, and only included when XK_INDEX is enabled in the IDE.

Table 3-11: **Data Output Channel TUSER Fields *(Cont'd)***

Field Name	Width	Padded	Description
BLK_EXP	8	Yes - zero extended	Block exponent (unsigned 2's complement): The amount of scaling applied (i.e., the number of bits by which the otherwise unscaled output value has been shifted down). A separate BLK_EXP field is included for each FFT channel that the core has. Available only when block floating-point is used. For more information on BLK_EXP, see Block Exponent.
OVFLO	1	No	Arithmetic overflow indicator (single bit, active-High): **OVFLO** is High during result unloading if any value in the data frame overflowed. The **OVFLO** signal is reset at the beginning of a new frame of data. A separate **OVFLO** field is included for each FFT channel that the core has. This port is optional and only available with scaled arithmetic or single precision floating-point I/O. For more information on **OVFLO**, see Overflow.

All fields with padding should be 0 extended to the next 8-bit boundary if they do not already finish on an 8-bit boundary.

TUSER Format

The data fields are packed into the **m_axis_data_tuser** vector in the following order (starting from the LSB):

1. (optional) **XK_INDEX** plus padding
2. (optional) **BLK_EXP** plus padding for channel 0
3. (optional) **BLK_EXP** plus padding for channel 1
4. etc.
5. (optional) **OVFLO** for channel 0
6. (optional) **OVFLO** for channel 1
7. etc.
8. Padding to make TUSER 8-bit aligned. Only needed when **OVFLO** is present

Note that the core cannot be configured to have both BLK_EXP and **OVFLO**.

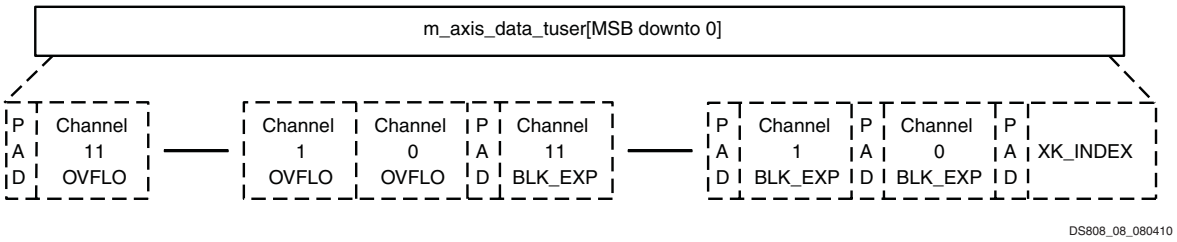

Figure 3-8: **Data Output Channel TUSER (m_axis_data_tuser) Format**

Optional fields are shown as dotted. As all fields are optional, it is possible to configure the core such that TUSER would have no fields. In this case it is automatically removed from the core interface.

TUSER Examples

Example 1

The core has been configured to have two FFT data channels, a 128 point transform size, overflow, and **`XK_INDEX`**. The third sample (**`XK_INDEX`** = 3) has an overflow on channel 0 but not on channel 1. **`XK_INDEX`** is 7 bits long.

The fields take on the values in Table 3-12.

Table 3-12: **Data Output Channel TUSER Example 1**

Field Name	Padding	Value
XK_INDEX	0	000 0011
OVFLO (channel 0)	None	1
OVFLO (channel 1)	None	0

This gives a vector length of 10 bits. As all AXI channels must be aligned to byte boundaries, 6 padding bits are required, giving an **`m_axis_data_tuser`** length of 16 bits.

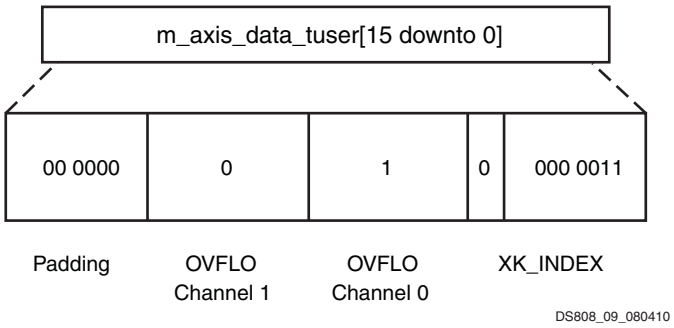

Figure 3-9: **Data Output Channel TUSER Example 1**

Example 2

The core has been configured to have two FFT data channels, block exponent, but no **XK_INDEX**. The output sample for channel 0 has a block exponent of 4, and the output sample for channel 1 has a block exponent of 31.

The fields take on the values in Table 3-13.

Table 3-13: **Data Output Channel TUSER Example 2**

Field Name	Padding	Value
BLK_EXP (channel 0)	000	0 0100
BLK_EXP (channel 1)	000	1 1111

This gives a vector length of 16 bits, so no more padding is required.

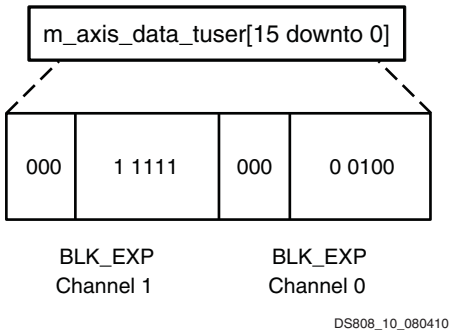

Figure 3-10: **Data Output Channel TUSER Example 2**

Status Channel

The Status channel contains per-frame status information, that is, information that relates to an entire frame of data. This is intended for downstream slaves that do not operate on the data directly but might need to know the information to control another part of the system. The exact position in the frame where the status is sent depends on the nature of the status information. The following information is classed as per-frame status:

1. **BLK_EXP** for each channel
2. **OVFLO** for each channel

Note that the core cannot be configured to have both **BLK_EXP** and **OVFLO**.

BLK_EXP status information is sent at the start of the frame and **OVFLO** status information is sent at the end of the frame.

Pinout

Table 3-14: **Status Channel Pinout**

Port Name	Port Width	I/O	Description
m_axis_status_tdata	Variable. See the Vivado IDE when configuring the core.	O	Carries the status data: BLK_EXP or OVFLO
m_axis_status_tvalid	1	O	Asserted by the core to signal that it is able to provide status data
m_axis_status_tready	1	I	Asserted by the external slave to signal that it is ready to accept data

TDATA Fields

The Status channel carries the fields in Table 3-15 in its TDATA vector.

Table 3-15: **Status Channel TDATA Fields**

Field Name	Width	Padded	Description
BLK_EXP	5	Yes - zero extended	Block exponent: The amount of scaling applied. A separate **BLK_EXP** field is included for each FFT channel that the core has. Available only when block floating-point is used. For more information on **BLK_EXP**, see Block Exponent.
OVFLO	1	No	Arithmetic overflow indicator (active-High): **OVFLO** is High during result unloading if any value in the data frame overflowed. The **OVFLO** signal is reset at the beginning of a new frame of data. A separate **OVFLO** field is included for each FFT channel that the core has. This port is optional and only available with scaled arithmetic or single precision floating-point I/O. For more information on **OVFLO**, see Overflow.

All fields with padding should be 0 extended to the next 8-bit boundary if they do not already finish on an 8-bit boundary.

TDATA Format

The data fields are packed into the **m_axis_status_tdata** vector in the following order (starting from the LSB):

1. (optional) **BLK_EXP** plus padding for channel 0
2. (optional) **BLK_EXP** plus padding for channel 1
3. etc.
4. (optional) **OVFLO** for channel 0
5. (optional) **OVFLO** for channel 1
6. etc.

7. Padding to make **TDATA** 8-bit aligned. Only needed when **OVFLO** is present

Note that the core cannot be configured to have both **BLK_EXP** and **OVFLO**.

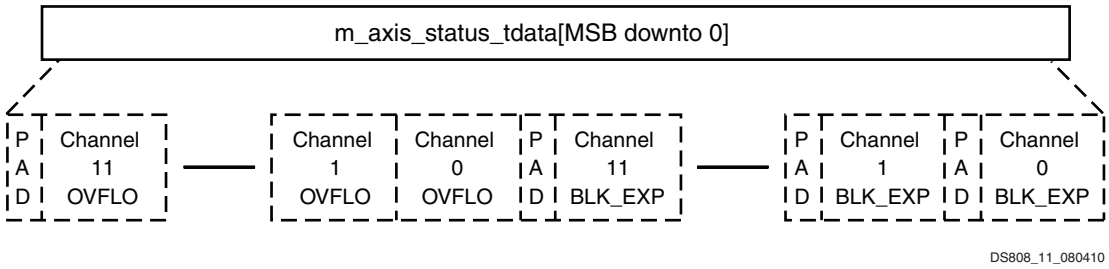

Figure 3-11: **Status channel TDATA (m_axis_status_tdata) Format**

Optional fields are shown as dotted. As all fields are optional, it is possible to configure the core such that TDATA would have no fields. In this case the entire Status channel is automatically removed from the core interface.

TDATA Example

Example 1

The core has been configured to have four FFT data channels and overflow. The current frame contains an overflow in channels 2 and 3.

Table 3-16: **Status Channel TDATA Example 1**

Field Name	Padding	Value
OVFLO (channel 0)	None	0
OVFLO (channel 1)	None	0
OVFLO (channel 2)	None	1
OVFLO (channel 3)	None	1

This gives a vector length of 4 bits. As all AXI channels must be aligned to byte boundaries, 4 padding bits are required, giving an **m_axis_status_tdata** length of 8 bits.

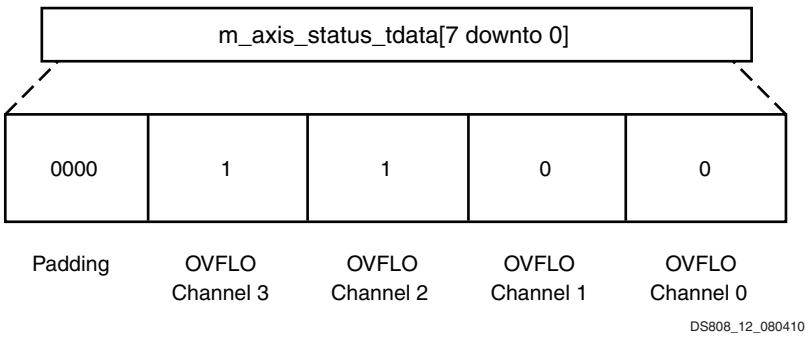

Figure 3-12: **Status Channel TDATA Example 1**

Example 2

The core has been configured to have one FFT data channel and overflow. The current frame contains no overflow.

Table 3-17: **Status Channel TDATA Example 2**

Field Name	Padding	Value
OVFLO (channel 0)	None	0

This gives a vector length of 1 bit. As all AXI channels must be aligned to byte boundaries, 7 padding bits are required, giving an **`m_axis_status_tdata`** length of 8 bits.

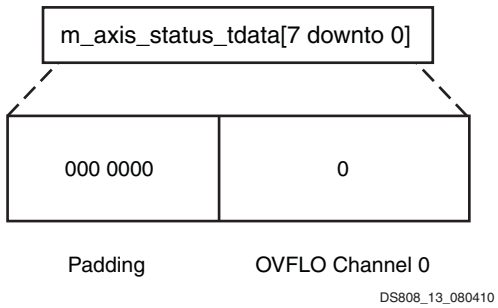

Figure 3-13: **Status Channel TDATA Example 2**

Theory of Operation

Finite Word Length Considerations

The Burst I/O architectures process an array of data by successive passes over the input data array. On each pass, the algorithm performs Radix-4 or Radix-2 butterflies, where each butterfly picks up four or two complex numbers, respectively, and returns four or two complex numbers to the same memory. The numbers returned to memory by the core are potentially larger than the numbers picked up from memory. A strategy must be employed to accommodate this dynamic range expansion. A full explanation of scaling strategies and their implications is beyond the scope of this document; for more information about this topic; see *A Simple Fixed-Point Error Bound for the Fast Fourier Transform* [Ref 3] and *Theory and Application of Digital Signal Processing* [Ref 4].

For a Radix-4 DIT FFT, the values computed in a butterfly stage can experience growth by a factor of up to $1 + 3\sqrt{2} \approx 5.242$. This implies a bit growth of up to 3 bits.

For Radix-2, the growth is by a factor of up to $1+\sqrt{2} \approx 2.414$. This implies a bit growth of up to 2 bits. This bit growth can be handled in three ways:

- Performing the calculations with no scaling and carrying all significant integer bits to the end of the computation
- Scaling at each stage using a fixed-scaling schedule
- Scaling automatically using block floating-point

All significant integer bits are retained when using full-precision unscaled arithmetic. The width of the datapath increases to accommodate the bit growth through the butterfly. The growth of the fractional bits created from the multiplication are truncated (or rounded) after the multiplication. The width of the output is (input width + $\log_2$(transform length) + 1). This accommodates the worst case scenario for bit growth.

Consider an unscaled Radix-2 DIT FFT: the datapath in each stage must grow by 1 bit as the adder and subtracter in the butterfly might add/subtract two full-scale values and produce a sample which has grown in width by 1 bit. This yields the $\log_2$(transform length) part of the increase in the output width relative to the input width. The complex multiplier preserves the magnitude of an input (as it applies a rotation on the complex plane), but can theoretically produce bit-growth when the magnitude of the input is greater than 1 (for example, 1+j has a magnitude of 1.414). This means that the complex multiplier bit growth must only be considered once in the entire FFT process, yielding the additional +1 increase in the output width relative to the input width. For example, a 1024-point transform with an input of 16 bits consisting of 1 integer bit and 15 fractional bits has an output of 27 bits with 12 integer bits and 15 fractional bits. Note that the core does not have a specific location for the binary point. The output maintains the same binary point location as the input. For the preceding example, a 16-bit input with 3 integer bits and 13 fractional bits would have an unscaled output of 27 bits with 14 integer bits and 13 fractional bits.

When using scaling, a scaling schedule is used to divide by a factor of 1, 2, 4, or 8 in each stage. If scaling is insufficient, a butterfly output might grow beyond the dynamic range and cause an overflow. As a result of the scaling applied in the FFT implementation, the transform computed is a scaled transform. The scale factor s is defined as

$$s = 2^{\sum_{i=0}^{\log(N-1)} b_i}$$

Equation 3-1

where b_i is the scaling (specified in bits) applied in stage i.

The scaling results in the final output sequence being modified by the factor $1/s$. For the forward FFT, the output sequence $X'(k)$, $k = 0,\ldots,N-1$ computed by the core is defined as

$$X'(k) = \frac{1}{s}X(k) = \frac{1}{s}\sum_{n=0}^{N-1} x(n)e^{-jnk2\pi/N} \quad k = 0,\mathrm{K}\ ,N-1$$

Equation 3-2

For the inverse FFT, the output sequence is

$$x(n) = \frac{1}{s}\sum_{k=0}^{N-1} X(k)e^{jnk2\pi/N} \qquad n = 0,\mathrm{K}\ ,N-1$$

Equation 3-3

If a Radix-4 algorithm scales by a factor of 4 in each stage, the factor of $1/s$ is equal to the factor of $1/N$ in the inverse FFT equation (Equation 1-2). For Radix-2, scaling by a factor of 2 in each stage provides the factor of $1/N$.

With block floating-point, each stage applies sufficient scaling to keep numbers in range, and the scaling is tracked by a block exponent.

As with unscaled arithmetic, for scaled and block floating-point arithmetic, the core does not have a specific location for the binary point. The location of the binary point in the output data is inherited from the input data and then shifted by the scaling applied.

Floating-Point Considerations

The FFT core optionally accepts data in IEEE-754 single-precision format with 32-bit words consisting of a 1-bit sign, 8-bit exponent, and 23-bit fraction. The construction of the word matches that of the Xilinx Floating-Point Operator core.

Implementing full floating-point on an FPGA can be expensive in terms of the resources required. The floating-point option in the FFT core uses a higher precision fixed-point FFT internally to achieve similar noise performance to a full floating-point FFT, with significantly fewer resources. Figure 3-14 illustrates the two levels of noise performance possible by selecting either 24 bits or 25 bits for the phase factor width. By increasing the phase factor width to 25 bits, more resources might be required, depending on the target device.

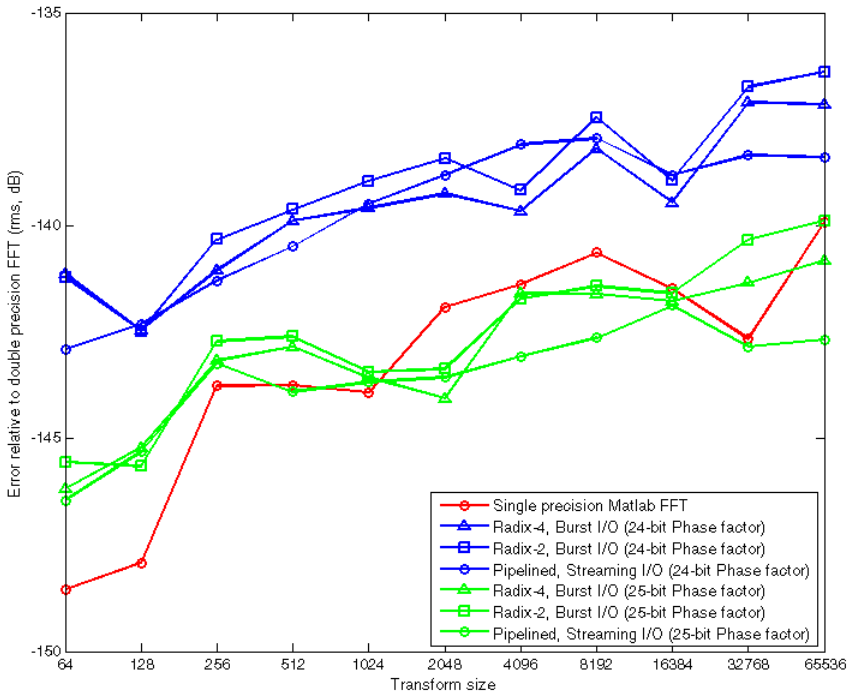

Figure 3-14: **Comparison of Two Levels of Noise Performance**

Figure 3-14 shows the ratio of the RMS difference between various models and the double-precision MATLAB® FFT to the data set peak amplitude. The models shown are the single-precision MATLAB FFT function (calculated by casting the input data to single-precision floating-point type), the FFT core using a 24-bit phase factor width, and the FFT core using a 25-bit phase factor width. To calculate the error signal, a randomized impulse (in magnitude and time) was used as the input signal, with the RMS error averaged over five simulation runs.

When comparing results against third party models, for example, MATLAB, it should be noted that a scaling factor is usually required to ensure that the results match. The scaling factor is data-dependent because the input data dictates the level of normalization required prior to the internal fixed-point core. Because the core does not provide this scaling factor in floating-point mode, you can apply scaling after the output of the core, if necessary.

RECOMMENDED: Xilinx recommends using the FFT C model and MEX function when evaluating floating-point datasets.

All optimization options (memory types and DSP slice optimization) remain available when floating-point input data is selected, allowing you to trade off resources with transform time.

Transform time for Burst I/O architectures is increased by approximately N, the number of points in the transform, due to the input normalization requirements. For the Pipelined Streaming I/O architecture, the initial latency to fill the pipeline is increased, but data still streams through the core with no gaps.

Denormalized Numbers

The floating-point interface to the FFT core does not support denormalized numbers. To match the behavior of the Xilinx Floating-Point Operator core, the core treats denormalized operands as zero, with a sign taken from the denormalized number.

NaNs and ± Infinity

If the core detects a NaN or ± Infinity value on the input, all output samples associated with the current input frame are set to NaN. The sign bit is set to zero and all exponent and fraction bits are set to 1.

Real-Valued Input Data

The FFT core accepts complex data samples, but can perform a transform on real-valued data by setting all imaginary input samples to zero.

Due to the finite wordlength effects described previously, noise is introduced during the transform, resulting in the output data not being perfectly symmetric. The DIT and DIF FFT algorithms have different noise effects due to the different calculation order.

For a thorough treatment of this topic, see *Limited Dynamic Range of Spectrum Analysis Due To Round off Errors Of The FFT* [Ref 5] and *Influence of Digital Signal Processing on Precision of Power Quality Parameters Measurement* [Ref 6].

The asymmetry between the two halves of the result is more noticeable at larger point sizes. In addition, the noise is more prominent in the lower frequency bins. Therefore, Xilinx recommends that the upper half (N/2+1 to N points) of the output data is used when performing a real-valued FFT.

Rounding Implementation

An option is available, in all architectures, to apply convergent rounding to the data after the butterfly stage. However, selecting this option does not apply convergent rounding to all points in the datapath where wordlength reduction occurs.

In particular, the outputs of all complex multipliers in the FFT datapath are truncated to reduce datapath width (while still maintaining adequate precision) and a simple rounding constant added to the fractional bits. This constant implements non-symmetric, round-towards-minus-infinity rounding, and can introduce a small bias to the FFT results over a large number of samples.

Dynamic Range Characteristics

The dynamic range characteristics are shown by performing *slot noise* tests. First, a frame of complex Gaussian noise data samples is created. An FFT is taken to acquire the spectrum of the data. To create the slot, a range of frequencies in the spectra is set to zero. To create the input slot noise data frame, the inverse FFT is taken, then the data is quantized to use the full input dynamic range. Because of the quantization, if a perfect FFT is done on the frame, the noise floor on the bottom of the slot is non-zero. The input data figures, which basically represent the dynamic range of the input format, display this.

This slot noise input data frame is fed to the FFT core to see how shallow the slot becomes due to the finite precision arithmetic. The depth of the slot shows the dynamic range of the FFT.

Figure 3-15 through Figure 3-24 show the effect of input data width on the dynamic range. All FFTs have the same bit width for both data and phase factors. Block floating-point arithmetic is used with rounding after the butterfly. The figures show the input data slot and the output data slot for bit widths of 24, 20, 16, 12, and 8.

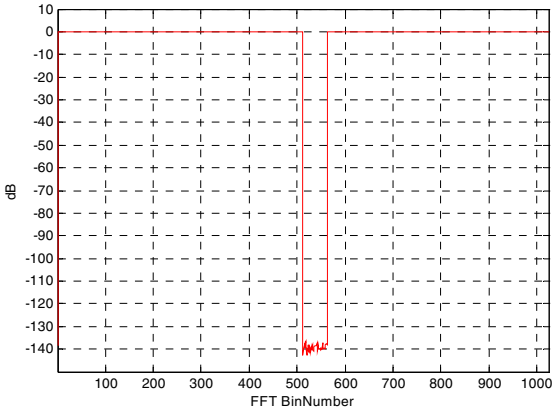

Figure 3-15: **Input Data: 24 Bits**

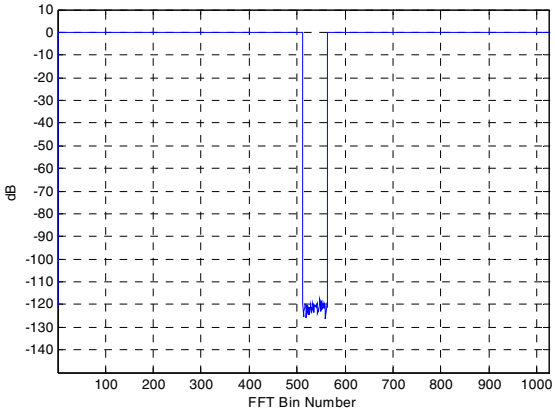

Figure 3-16: **FFT Core Results: 24 Bits**

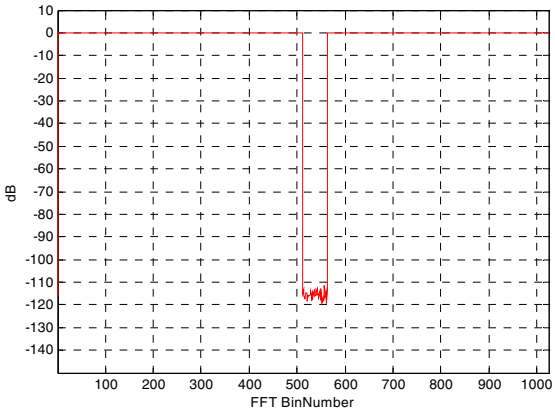

Figure 3-17: **Input Data: 20 Bits**

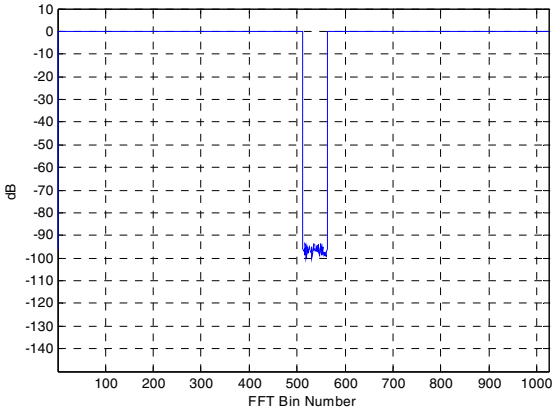

Figure 3-18: **FFT Core Results: 20 Bits**

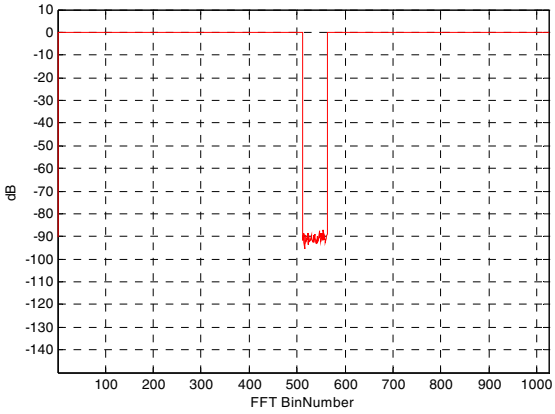

Figure 3-19: **Input Data: 16 Bits**

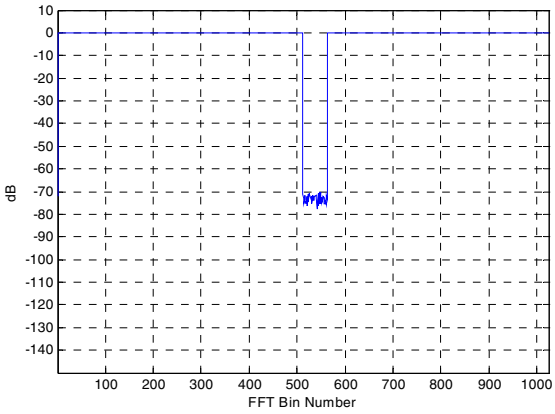

Figure 3-20: **FFT Core Results: 16 Bits**

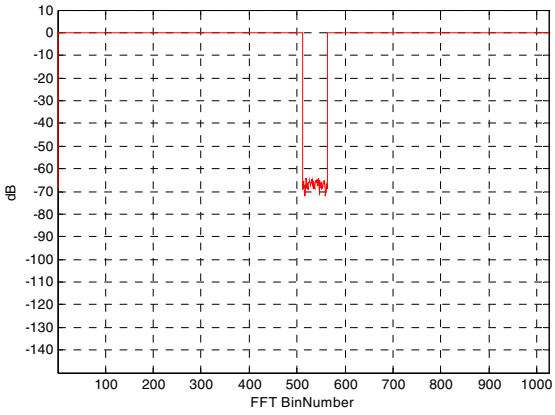

Figure 3-21: **Input Data: 12 Bits**

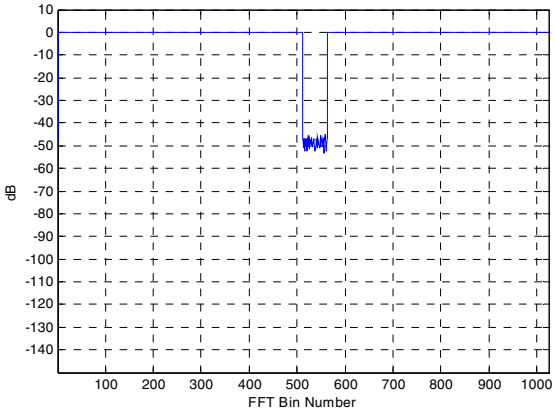

Figure 3-22: **FFT Core Results: 12 Bits**

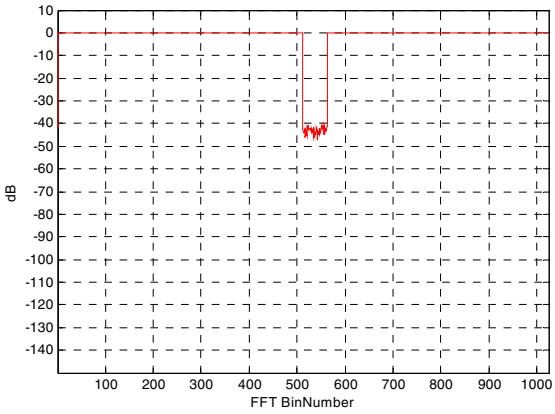

Figure 3-23: **Input Data: 8 Bits**

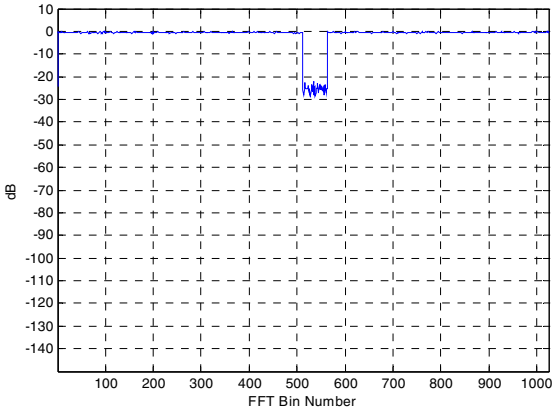

Figure 3-24: **FFT Core Results: 8 Bits**

There are several options available that also affect the dynamic range. Consider the arithmetic type used.

Figure 3-25, Figure 3-26, and Figure 3-27 display the results of using unscaled, scaled (scaling of 1/1024), and block floating-point. All three FFTs are 1024 point, Radix-4 Burst I/O transforms with 16-bit input, 16-bit phase factors, and convergent rounding.

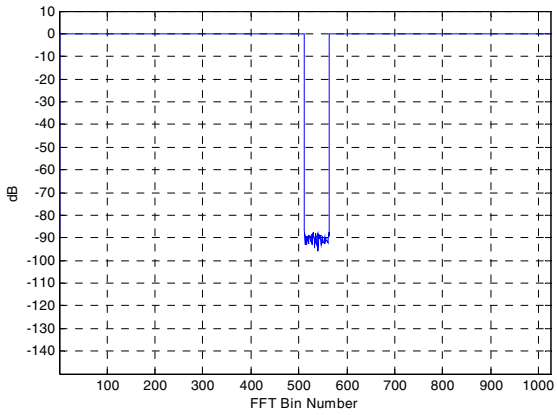

Figure 3-25: **Full-Precision Unscaled Arithmetic**

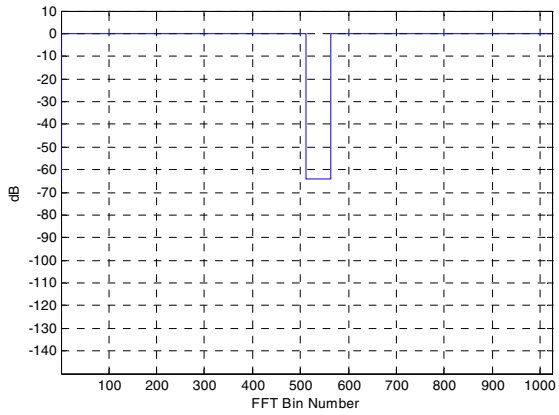

Figure 3-26: **Scaled (scaling of 1/N) Arithmetic**

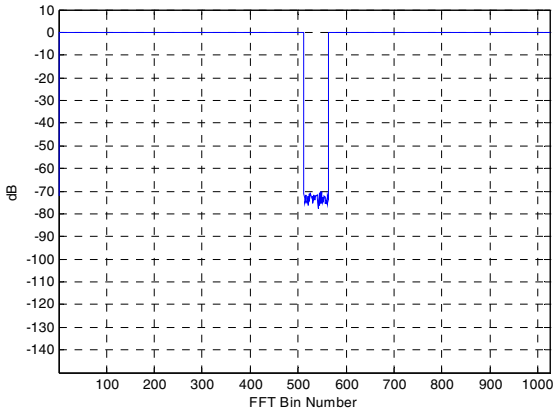

Figure 3-27: **Block Floating-Point Arithmetic**

After the butterfly computation, the LSBs of the datapath can be truncated or rounded. The effects of these options are shown in Figure 3-28 and Figure 3-29. Both transforms are 1024 points with 16-bit data and phase factors using block floating-point arithmetic.

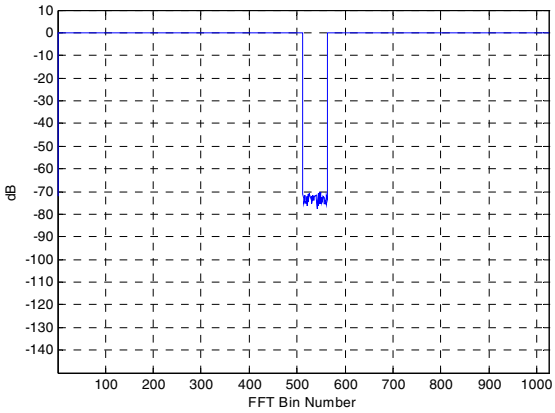

Figure 3-28: **Convergent Rounding**

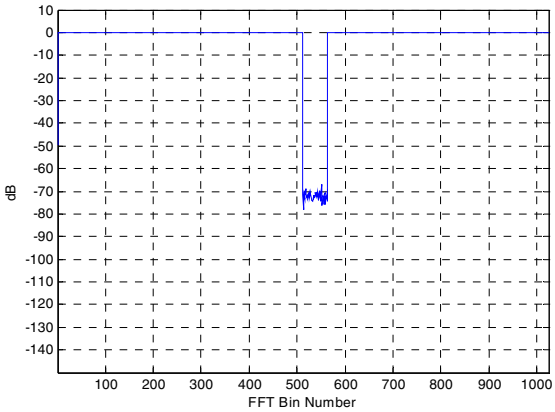

Figure 3-29: **Truncation**

For illustration purposes, the effect of point size on dynamic range is displayed Figure 3-30 through Figure 3-32. The FFTs in these figures use 16-bit input and phase factors along with convergent rounding and block floating-point arithmetic.

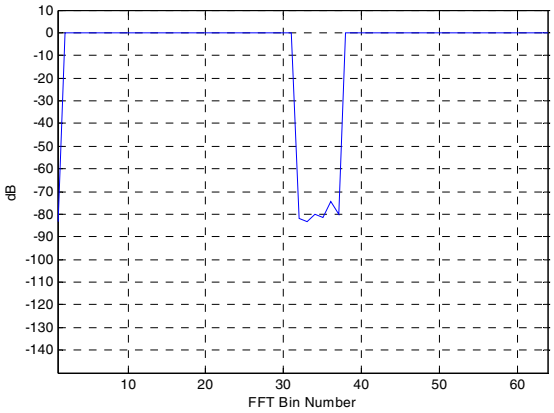

Figure 3-30: **64-point Transform**

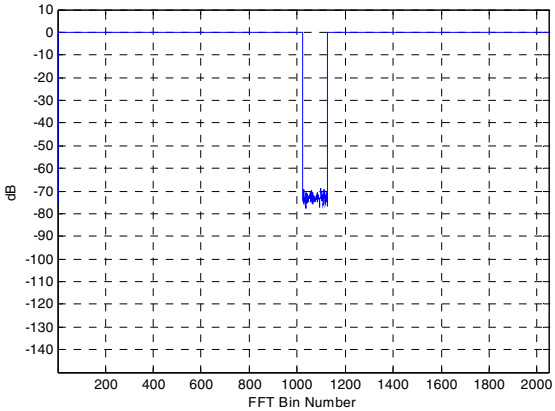

Figure 3-31: **2048-point Transform**

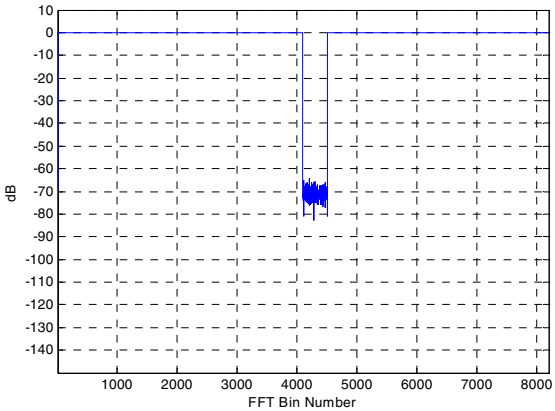

Figure 3-32: **8192-point Transform**

All of the preceding dynamic range plots show the results for the Radix-4 Burst I/O architecture. Figure 3-33 and Figure 3-34 show two plots for the Radix-2 Burst I/O architecture. Both use 16-bit input and phase factors along with convergent rounding and block floating-point.

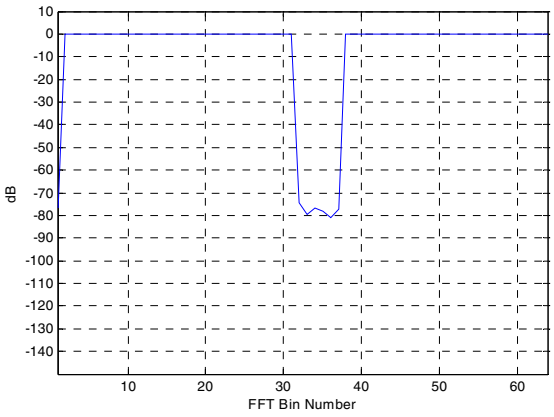

Figure 3-33: **64-point Radix-2 Transform**

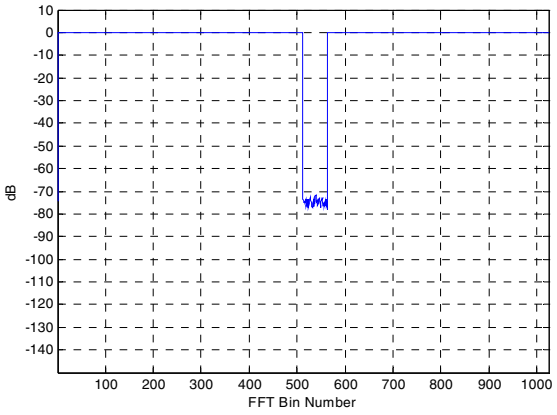

Figure 3-34: **1024-point Radix-2 Transform**

Architecture Options

The FFT core provides four architecture options to offer a trade-off between core size and transform time.

- Pipelined Streaming I/O **–** Allows continuous data processing.
- Radix-4 Burst I/O **–** Loads and processes data separately, using an iterative approach. It is smaller in size than the pipelined solution, but has a longer transform time.
- Radix-2 Burst I/O **–** Uses the same iterative approach as Radix-4, but the butterfly is smaller. This means it is smaller in size than the Radix-4 solution, but the transform time is longer.

- Radix-2 Lite Burst I/O – Based on the Radix-2 architecture, this variant uses a time-multiplexed approach to the butterfly for an even smaller core, at the cost of longer transform time.

Figure 3-35 illustrates the trade-off of throughput versus resource use for the four architectures. As a rule of thumb, each architecture offers a factor of 2 difference in resource from the next architecture. The example is for an even power of 2 point size. This does not require the Radix-4 architecture to have an additional Radix-2 stage.

All four architectures can be configured to use a fixed-point interface with one of three fixed-point arithmetic methods (unscaled, scaled or block floating-point) or might instead use a floating-point interface.

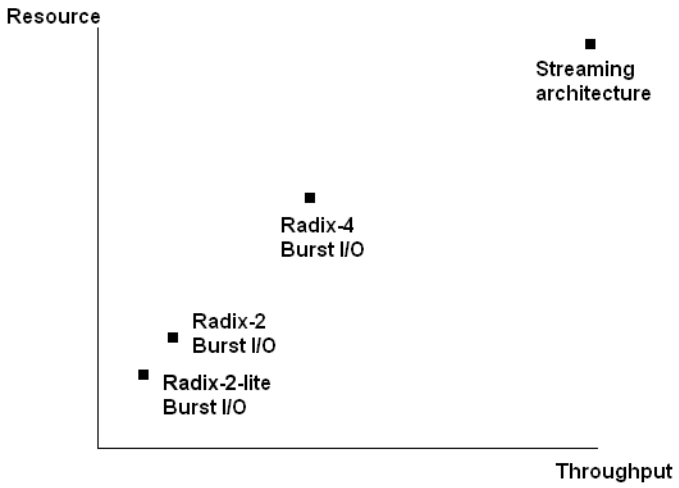

Figure 3-35: **Resource versus Throughput for Architecture Options**

Bit and Digit Reversal

Each architecture offers the option of natural or reversed ordering of output data, with data being input in natural order. The FFT algorithm reorders the samples during processing such that data input in natural order is output in reversed order. The core can optionally output the data in natural order. However, this imposes a cost on each architecture. For the Burst I/O architectures, this imposes a time penalty, because unloading the data cannot take place at the same time as loading input data for the next frame, so separate unload and load phases are required. In the pipelined architecture, it requires additional RAM storage to perform the reordering.

In the Radix-2 Burst I/O, Radix-2 Lite Burst I/O, and Pipelined Streaming I/O architectures, the Bit Reverse order is simple to calculate by taking the index of the data point, written in binary, and reversing the order of the digits. Hence, 0000, 0001, 0010, 0011, 0100,...(0, 1, 2, 3, 4,...) becomes 0000, 1000, 0100, 1100, 0010,...(0, 8, 4, 12, 2,...).

In the case of the Radix-4 Burst I/O architecture, the reversal applies to *digits* and, therefore, is called Digit Reversal. A digit in Radix-4 is two bits. Hence, 0000, 0001, 0010, 0011, 0100,...(0, 1, 2, 3, 4,...) becomes 0000, 0100, 1000, 1100, 0001,...(0, 4, 8, 12, 1,...), as the pairs of digits are reversed. Where the transform size requires an odd number of index bits, the odd digit in the least significant place is moved to the most significant place, so 00000, 00001, 00010, 00011, 00100,... (0, 1, 2, 3, 4,...) becomes 00000, 10000, 00100, 10100, 01000,...(0, 16, 4, 20, 8,...)

Note: The core can optionally output a data point index along with the data. See XK Index for more information.

Pipelined Streaming I/O

The Pipelined Streaming I/O solution pipelines several Radix-2 butterfly processing engines to offer continuous data processing. Each processing engine has its own memory banks to store the input and intermediate data (Figure 3-36). The core has the ability to simultaneously perform transform calculations on the current frame of data, load input data for the next frame of data, and unload the results of the previous frame of data. You can continuously stream in data and, after the calculation latency, can continuously unload the results. If preferred, this design can also calculate one frame by itself or frames with gaps in between.

IMPORTANT: *Continually streaming data does not imply that AXI4-Stream waitstates from the FFT core can be ignored. There are situations where the FFT core might have to insert waitstates to pause the incoming sample data.*

In the scaled fixed-point mode, the data is scaled after every pair of Radix-2 stages. The block floating-point mode might use significantly more resources than the scaled mode, as it must maintain extra bits of precision to allow dynamic scaling without impacting performance. Therefore, if the input data is well understood and is unlikely to exhibit large amplitude fluctuation, using scaled arithmetic (with a suitable scaling schedule to avoid overflow in the known worst case) is sufficient, and resources might be saved.

The input data is presented in natural order. The unloaded output data can either be in bit reversed order or in natural order. When natural order output data is selected, additional memory resource is utilized.

This architecture covers point sizes from 8 to 65536. You have the flexibility to select the number of stages to use block RAM for data and phase factor storage. The remaining stages use distributed memory.

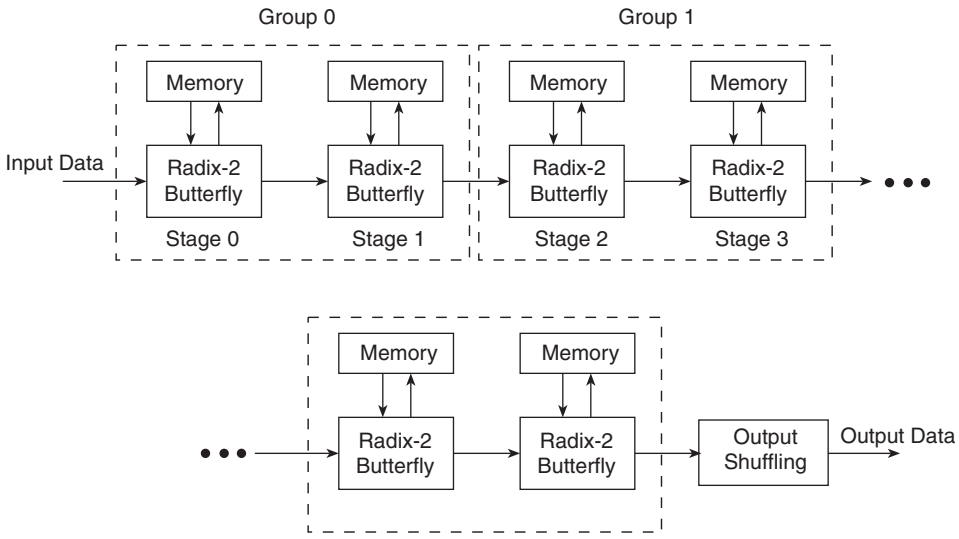

Figure 3-36: **Pipelined Streaming I/O**

Radix-4 Burst I/O

With the Radix-4 Burst I/O solution, the FFT core uses one Radix-4 butterfly processing engine (Figure 3-37). It loads and/or unloads data separately from calculating the transform. Data I/O and processing are not simultaneous. When the FFT is started, the data is loaded. After a full frame has been loaded, the core computes the transform. When the computation has finished, the data can be unloaded, but cannot be loaded or unloaded during the calculation process. The data loading and unloading processes can be overlapped if the data is unloaded in digit reversed order.

This architecture has lower resource usage than the Pipelined Streaming I/O architecture, but a longer transform time, and supports point sizes from 64 to 65536. Data and phase factors can be stored in block RAM or in distributed RAM (the latter for point sizes less than or equal to 1024).

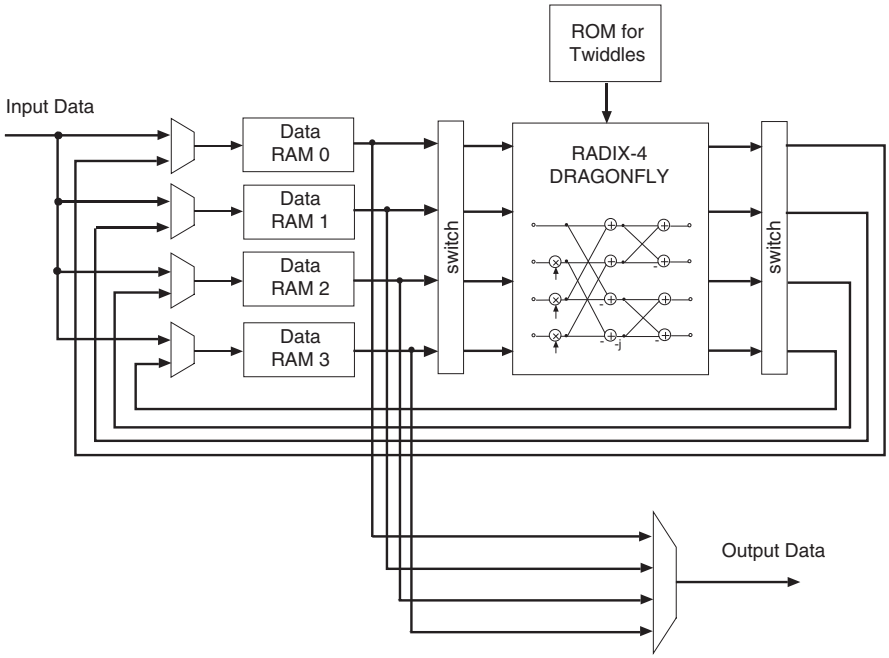

Figure 3-37: **Radix-4 Burst I/O**

Radix-2 Burst I/O

The Radix-2 Burst I/O architecture uses one Radix-2 butterfly processing engine (Figure 3-38). After a frame of data is loaded, the input data stream must halt until the transform calculation is completed. Then, the data can be unloaded. As with the Radix-4 Burst I/O architecture, data can be simultaneously loaded and unloaded when the output samples are in bit reversed order. This solution supports point sizes from 8 to 65536. Both the data memories and phase factor memories can be in either block RAM or distributed RAM (the latter for point sizes less than or equal to 1024).

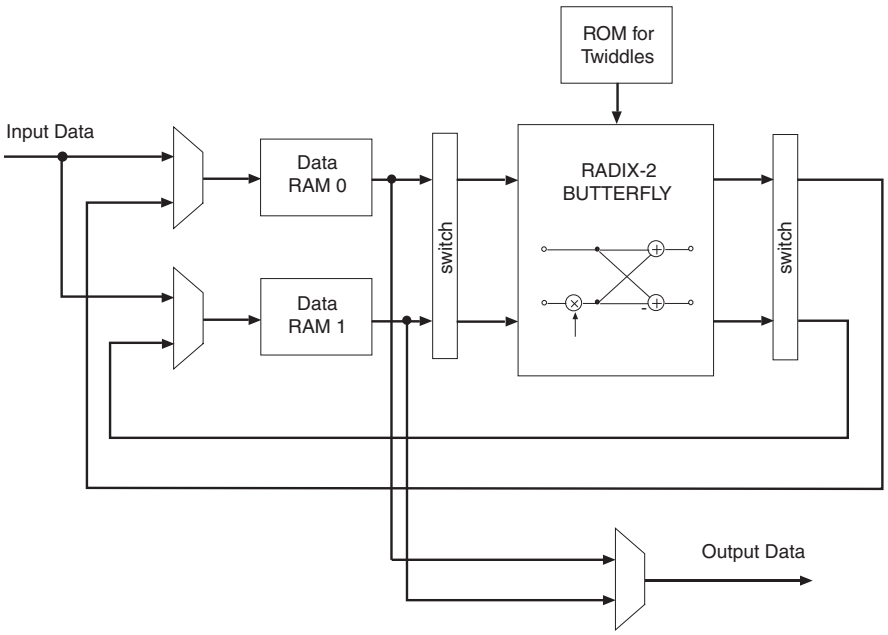

Figure 3-38: **Radix-2 Burst I/O**

Radix-2 Lite Burst I/O

This architecture differs from the Radix-2 Burst I/O in that the butterfly processing engine uses one shared adder/subtractor, hence reducing resources at the expense of an additional delay per butterfly calculation. Again, as with the Radix-4 and Radix-2 Burst I/O architectures, data can be simultaneously loaded and unloaded only if the output samples are in bit reversed order. This solution supports point sizes from 8 to 65536. See Figure 3-39.

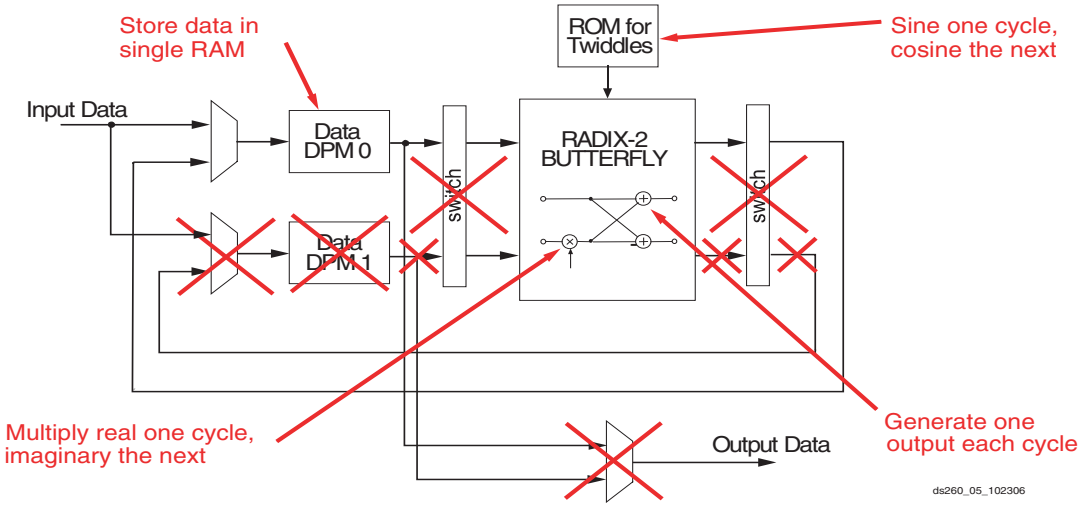

Figure 3-39: **Radix-2 Lite Burst I/O**

Run Time Transform Configuration

All run time configuration options discussed in this section are programed using the Configuration channel. See Configuration Channel for more information.

Transform Size

The transform point size can be set through the NFFT field in the Configuration channel if the run time configurable transform length option is selected. Valid settings and the corresponding transform sizes are provided in Table 3-18. If the NFFT value entered is too large, the core sets itself to the largest available point size (selected in the IDE). If the value is too small, the core sets itself to the smallest available point size: 64 for the Radix-4 Burst I/O architecture and 8 for the other architectures.

Table 3-18: **Valid NFFT Settings**

NFFT[4:0]	Transform size (*N*)
00011	8
00100	16
00101	32
00110	64
00111	128
01000	256
01001	512
01010	1024
01011	2048
01100	4096
01101	8192
01110	16384
01111	32768
10000	65536

Forward/Inverse and Scaling Schedule

The transform type (forward or inverse) and the scaling schedule can be set frame-by-frame without interrupting frame processing. Both the transform type and the scaling schedule can be set independently for each FFT channel in a multichannel core. Each FFT data channel has an assigned FWD_INV field and SCALE_SCH field in the Configuration channel. Setting the FWD_INV field to 0 produces an inverse FFT, and setting the FWD_INV field to 1 creates the forward transform.

A scaling schedule is not required (SCALE_SCH is ignored) when the FFT core is configured to process floating-point data. Normalization and scaling are handled internally for floating-point data.

Burst I/O Architectures

The scaling performed during successive stages can be set using the appropriate **SCALE_SCH** field in the Configuration channel. For the Radix-4, Burst I/O and Radix-2 architectures, the value of the **SCALE_SCH** field is used as pairs of bits [... N4, N3, N2, N1, N0], each pair representing the scaling value for the corresponding stage. Stages are computed starting with stage 0 as the two LSBs. There are $\log_4$(point size) stages for Radix-4 and $\log_2$(point size) stages for Radix-2. In each stage, the data can be shifted by 0, 1, 2, or 3 bits, which corresponds to **SCALE_SCH** values of 00, 01, 10, and 11. For example, for Radix-4, when N = 1024, [01 10 00 11 10] translates to a right shift by 2 for stage 0, shift by 3 for stage 1, no shift for stage 2, a shift of 2 for stage 3, and a shift of 1 for stage 4 (there are $\log_4(1024)$ = 5 Radix-4 stages). This scaling schedule scales by a total of 8 bits which gives a scaling factor of 1/256. The conservative schedule **SCALE_SCH** = [10 10 10 10 11] completely avoids overflows in the Radix-4, Burst I/O architecture. For the Radix-2, Burst I/O and Radix-2 Lite, Burst I/O architectures, the conservative scaling schedule of [01 01 01 01 01 01 01 01 01 10] prevents overflow for N = 1024 (there are $\log_2(1024)$ = 10 Radix-2 stages).

Pipelined Streaming I/O Architecture

For the Pipelined Streaming I/O architecture, consider every pair of adjacent Radix-2 stages as a group. That is, group 0 contains stage 0 and 1, group 1 contains stage 2 and 3, and so on. The value of the **SCALE_SCH** field is also used as pairs of bits [... N4, N3, N2, N1, N0]. Each pair represents the scaling value for the corresponding group of two stages. Groups are computed starting with group 0 as the two LSBs. In each group, the data can be shifted by 0, 1, 2, or 3 bits which corresponds to SCALE_SCH values of 00, 01, 10, and 11. For example, when N = 1024, [10 10 00 01 11] translates to a right shift by 3 for group 0 (stages 0 and 1), shift by 1 for group 1 (stages 2 and 3), no shift for group 3 (stages 4 and 5), a shift of 2 in group 3 (stages 6 and 7), and a shift of 2 for group 4 (stages 8 and 9). The conservative schedule **SCALE_SCH** = [10 10 10 10 11] completely avoids overflows in the Pipelined Streaming I/O architecture. When the point size is not a power of 4, the last group only contains one stage, and the maximum bit growth for the last group is one bit. Therefore, the two MSBs of the scaling schedule can only be 00 or 01. A conservative scaling schedule for N = 512 is **SCALE_SCH** = [01 10 10 10 11].

The initial value and reset value of the FWD_INV field is forward = 1. The scaling schedule is set to $1/N$. That translates to [10 10 10 10... 10] for the Radix-4, Burst I/O and Pipelined Streaming I/O architectures, and [01 01... 01] for the Radix-2 architectures. The core uses the (2*number of stages) LSBs for the scaling schedule. So, when the point size decreases, the leftover MSBs are ignored. However, all bits are programmed into the core and are used in later transforms if the point size increases.

Cyclic Prefix Insertion

Cyclic prefix insertion takes a section of the output of the FFT and prefixes it to the beginning of the transform. The resultant output data consists of the cyclic prefix (a copy of

the end of the output data) followed by the complete output data, all in natural order. Cyclic prefix insertion is only available when output ordering is Natural Order.

When cyclic prefix insertion is used, the length of the cyclic prefix can be set frame-by-frame without interrupting frame processing. The cyclic prefix length can be any number of samples from zero to one less than the point size. The cyclic prefix length is set by the **CP_LEN** field in the Configuration channel. For example, when *N* = 1024, the cyclic prefix length can be from 0 to 1023 samples, and a **CP_LEN** value of 0010010110 produces a cyclic prefix consisting of the last 150 samples of the output data.

The initial value and reset value of **CP_LEN** is 0 (no cyclic prefix). The core uses the $\log_2$(point size) MSBs of **CP_LEN** for the cyclic prefix length. So, when the point size decreases, the leftover LSBs are ignored. This effectively scales the cyclic prefix length with the point size, keeping them in approximately constant proportion. However, all bits of **CP_LEN** are programmed into the core and are used in later transforms if the point size increases.

Transform Status

Overflow

Fixed-Point Data

The Overflow (**OVFLO**) field in the Data Output and Status channels is only available when the Scaled arithmetic is used. **OVFLO** is driven High during unloading if any point in the data frame overflowed. For a multichannel core, there is a separate **OVFLO** field for each channel.

When an overflow occurs in the core, the data is wrapped rather than saturated, resulting in the transformed data becoming unusable for most applications.

Floating-Point Data

The Overflow field is used to indicate an exponent overflow when the FFT is processing floating-point data. The output sample which overflowed is set to ±∞, depending on the sign of the internal result. The Overflow field is not asserted when a NaN value is present on the output. NaN values can only occur at the FFT output when the input data frame contains NaN or ±∞ samples.

Block Exponent

The Block Exponent (**BLK_EXP**) field in the Data Output and the Status channels (used only with the block floating-point option) contains the block exponent. For a multichannel core, there is a separate **BLK_EXP** field for each channel. The value present in the field represents the total number of bits the data was scaled during the transform. For example, if **BLK_EXP** has a value of 00101 = 5, this means the associated output data (**XK_RE**, **XK_IM**) was scaled by 5 bits (shifted right by 5 bits), or in other words, was divided by 32, to fully use the available dynamic range of the output datapath without overflowing. Because block scaling

is performed based on the maximum value at each stage of processing, the **BLK_EXP** value may differ from one architecture to another, even with identical input data, due to the different inherent scaling performed per stage of processing in each architecture.

XK Index

The **XK_INDEX** field (if present in the Data Output channel) gives the sample number of the **XK_RE/XK_IM** data being presented at the same time. In the case of natural order outputs, **XK_INDEX** increments from 0 to (point size) -1. When bit reversed outputs are used, **XK_INDEX** covers the same range of numbers, but in a bit (or digit) reversed manner.

For example, when you have an 8 point FFT, **XK_INDEX** takes on the values in Table 3-19.

Table 3-19: **XK _INDEX values for 8 point FFT**

XK_INDEX with Natural Outputs	XK_INDEX with Bit Reversed Outputs
0 ('b000)	0 ('b000)
1 ('b001)	4 ('b100)
2 ('b010)	2 ('b010)
3 ('b011)	6 ('b110)
4 ('b100)	1 ('b001)
5 ('b101)	5 ('b101)
6 ('b110)	3 ('b011)
7 ('b111)	7 ('b111)

If cyclic prefix insertion is used, the cyclic prefix is unloaded first and **XK_INDEX** counts from (point_size) - (cyclic prefix length) up to (point size) -1. After the cyclic prefix has been unloaded, or if the cyclic prefix length is zero, the whole frame of output data is unloaded. **XK_INDEX** counts from 0 up to (point size) -1 as before. Cyclic Prefix Insertion is only possible with natural order outputs.

Controlling the FFT Core

Symbol data to be processed is loaded into the core using the Data Input channel. Processed symbol data is unloaded using the Data Output channel. Both of these use the AXI4-Stream protocol. Figure 3-40 shows the basics of this protocol.

TVALID is driven by the Master component to show that it has data to transfer, and TREADY is driven by the Slave component to show that it is ready to accept data. When both TVALID and TREADY are High, a transfer takes place. Points A in the diagram show clock cycles where no data is transferred because neither the Master or the Slave is ready. Point B shows two clock cycles where data is not transferred because the Master does not have any data to transfer. This is known as a Master Waitstate. Point C shows a clock cycle where no data is transferred because the Slave is not ready to accept data. This is known as a Slave Waitstate. Master and Slave waitstates can extend for any number of clock cycles.

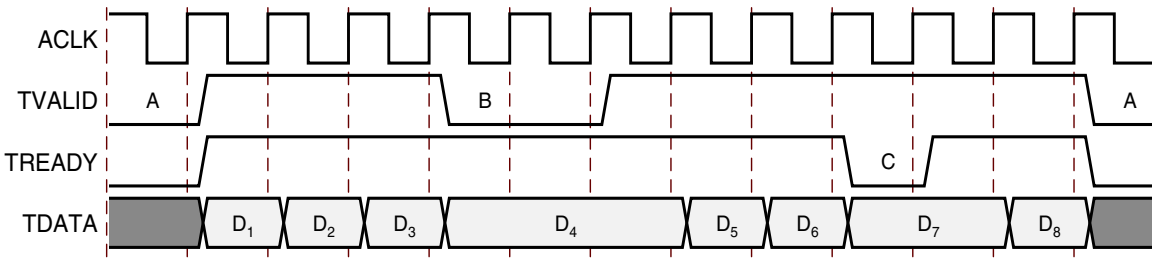

Figure 3-40: **AXI Transfers and Terminology**

After the master asserts TVALID High, it must remain asserted (and the associated data remain stable) until the slave asserts TREADY High.

To load a frame into the core, the upstream master supplying the **XN_RE** and **XN_IM** data has to send it when it is ready. If the core can accept it (which is when **s_axis_data_tready** = 1) then it is buffered by the core until it can be processed. If the core cannot accept it (which is when **s_axis_data_tready** = 0), a slave waitstate exists in the AXI channel and the master is stalled. Figure 3-40 shows the loading of the sample data for an 8 point FFT. The upstream master drives TVALID and the core drives TREADY. In this case, both the master and the core insert waitstates.

Unloading a frame works in a similar manner, except that the core is the master in this case. When it has **XK_RE** and **XK_IM** data to unload, it asserts its TVALID signal (**m_axis_data_tvalid** = 1). The downstream slave that consumes the processed sample data can then accept the data (**m_axis_data_tready** = 1) or not (**m_axis_data_tready** = 0). Figure 3-40 also shows the unloading of the sample data for an 8 point FFT (with no cyclic prefix). The core drives TVALID and the downstream slave drives TREADY. In this case, both the core and the slave insert waitstates.

The previous description only applies when the core is configured to use Non-Realtime mode. The situation is different in Realtime mode, which is used to create a smaller and faster design at the expense of flexibility in loading and unloading data. When the core is configured to use Realtime mode, the following occurs:

1. The TREADY signal on the Data Output channel (**m_axis_data_tready**) is removed
2. The TREADY signal on the Status channel (**m_axis_status_tready**) is removed
3. The TVALID signal on the Data Input channel is ignored when the loading of a frame has begun

The first two points mean that neither the downstream slave that consumes processed data, or the downstream slave that consumes status information, can insert waitstates using TREADY (**m_axis_data_tready** and **m_axis_status_tready**, respectively) as the pins are not present on the core. Both slaves must be able to respond immediately on every clock cycle where the core is producing data (**m_axis_data_tvalid** asserted High or **m_axis_status_tvalid** asserted High). If the slave cannot respond immediately, then data is lost.

The third point is slightly more complex as TVALID (**s_axis_data_tvalid**) cannot be removed. The upstream master still controls the start of a frame with TVALID. The core does not try to load a frame until the upstream master has asserted TVALID to provide the first symbol and there is no requirement for the master to supply the first sample of a frame at any particular time. However, when this has occurred, TVALID is then ignored by the core and it assumes that the master provides symbol data immediately on every clock cycle where TREADY is High. If the master does not provide data when requested, the data from the last provided symbol is reused and the **event_data_in_channel_halt** is asserted to show that the timing requirements have been violated. Note that the core can still insert waitstates when in Realtime mode. It is only the response to externally induced waitstates that changes.

Figure 3-41 shows the upstream master inserting waitstates while loading an 8 point frame in Realtime mode. At point A, the master has sent one sample to the Data Input channel. The core then inserts a waitstate while it waits for the FFT processing core to start the transform. This is shown as one cycle here, but it could be longer in certain cases. At point B, the master inserts two waitstates using TVALID. However, the core ignores them and uses the previous data (D_3) for the missing data. It is likely that the processed frame will be corrupted.

At point C, the master starts supplying the last samples of the frame (D_7 and later D_8) but the core has already started processing the frame and inserts a waitstate. The Master and the core are now out of synchronisation. When the core finishes processing the frame and is ready for a new frame, it sees D_7 as the first symbol of the new frame and starts to consume another 8 samples.

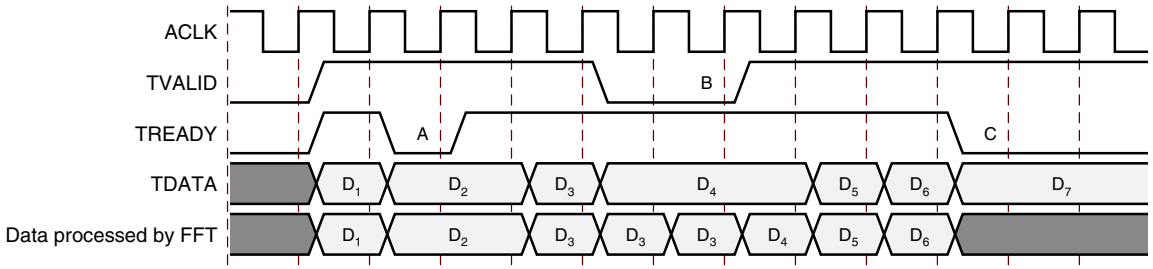

Figure 3-41: **Incorrect Transfer in Realtime mode**

IMPORTANT: *It is important that Realtime mode is only selected when the appropriate external masters and slaves can meet the timing requirements on supplying and consuming data.*

Transform Timing

The core starts to process a frame as soon as a) the upstream master asks it to by supplying data to process, and b) when it is able to. The chosen architecture and cyclic prefix insertion are the major configuration options that affect when the core is able to process a new frame.

The following timing diagrams are generalizations of actual behavior used to show the broad phases the core moves through when processing frames, and how these phases can (or cannot) overlap. The lengths of the various phases are not to scale, and the processing time might be much longer than the time required to input or output a frame.

In particular, the behavior of TREADY on the input data channel is not fully accurate because the Data Input channel buffers the data (16 symbols in Non-Realtime mode and 1 symbol in Realtime mode). However, this data waits in the buffer until the FFT processing core is ready for it. The Data Input channel TREADY in these diagrams is used as an indication of when the FFT processing core wants data rather than when the AXI channel (with its buffer) wants data.

Pipelined Streaming I/O with no Cyclic Prefix Insertion

When Pipelined Streaming I/O is selected and no cyclic prefix is used, the core can overlap the loading of a frame with the processing and unloading of earlier frames. If the upstream master supplies the first symbol for a new frame immediately after the last symbol for the previous frame, the core starts loading it immediately.

Figure 3-42 shows the general timing for back-to-back frames in the Pipelined Streaming architecture.

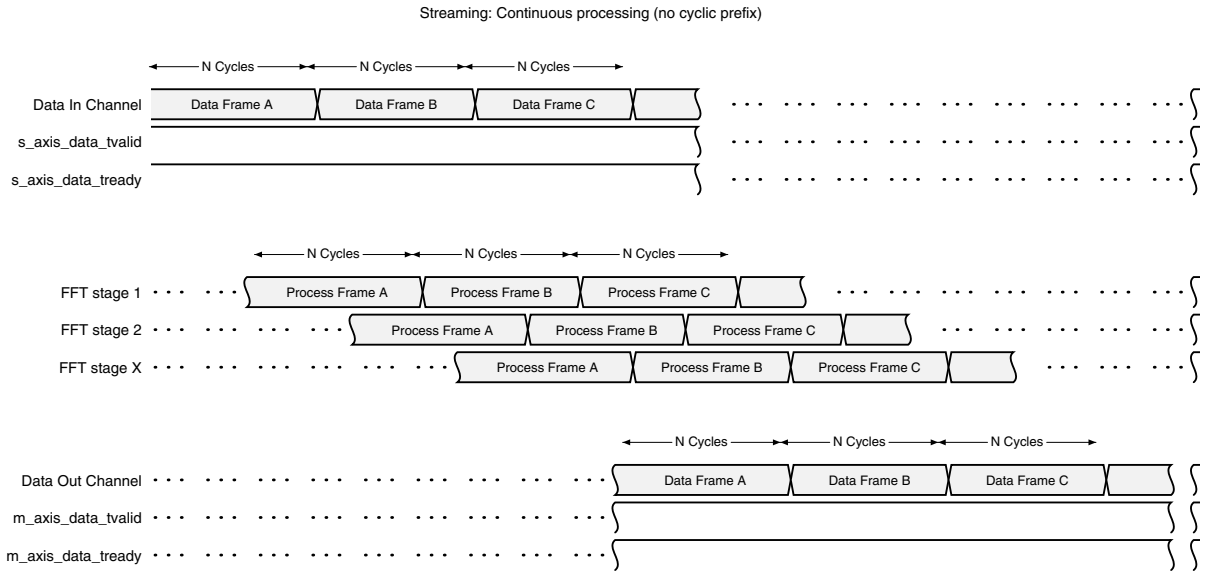

Figure 3-42: **Transform Timing for Entire Frames in Pipelined Streaming I/O with no Cyclic Prefix Insertion**

Note that there is a latency between a frame being loaded and the processed data for that frame being available. This latency depends on the options chosen in the Vivado IDE to parameterize the core. However, when that latency has passed, processed frames appear back-to-back.

Pipelined Streaming I/O with Cyclic Prefix Insertion

If cyclic prefix insertion is used, more samples are unloaded from the core than are loaded. Therefore, the core cannot continuously stream frames, but must insert a gap of cyclic prefix length clock cycles in between each frame of input data to accommodate the additional clock cycles required to unload the cyclic prefix (Figure 3-43). This is indicated by the TREADY signal on the Data Input channel. This goes Low to allow the core time to unload the cyclic prefix.

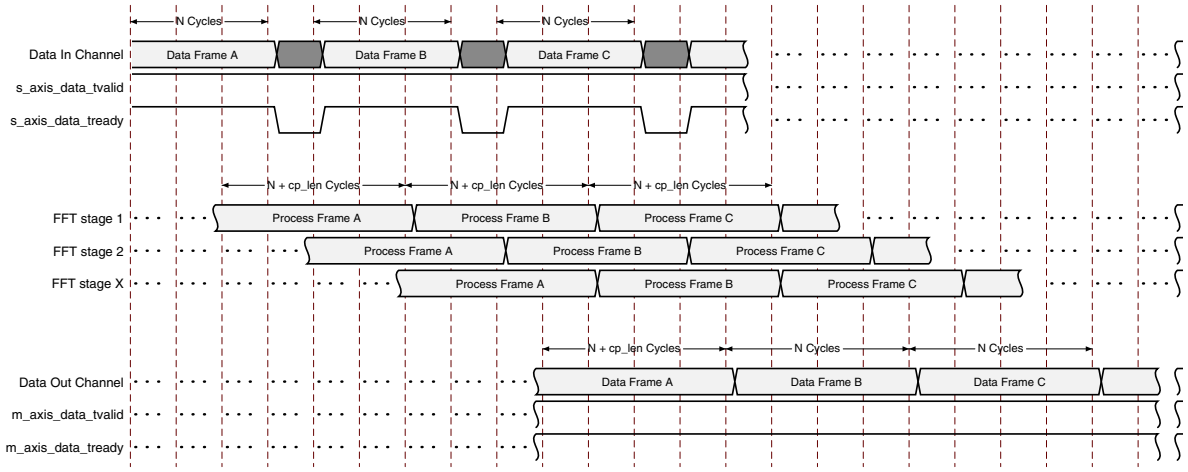

Figure 3-43: **Transform Timing for Entire Frames in Pipelined Streaming I/O with Cyclic Prefix Insertion**

Burst I/O Architectures

The Burst I/O architectures do not allow frame overlapping to the same degree as the Pipelined Streaming I/O architecture. When natural ordered outputs are used, a frame has to be processed and unloaded before the core can start to load the following frame.

Note: This refers to the FFT processing core. As the Data In channel has a 16 element deep buffer on its input, it can start to pre-buffer a frame while a frame is still being processed. In the case of 8 and 16 point FFTs, it can pre-buffer entire frames. However, this buffered data waits in the buffer until the FFT engine has finished dealing with the current frame.

When bit-reversed outputs are used, the core only unloads data when a new frame is loaded. This means that the loading of frame N+1 overlaps with (and actually causes) the unloading of frame N. However, if the upstream master does not supply data to the core when it is ready to start unloading a frame, the core will flush the frame out manually. If this occurs, the loading and unloading phases do not overlap.

Figure 3-44 shows the general transform timing for a Burst I/O architecture with natural ordered outputs. This requires distinct load, process and unload phases. The upstream master is constantly attempting to stream data as is the downstream slave. These examples do not show the effect of a cyclic prefix, which is to extend the unloading phase.

The Upstream Master loads all of the data for Frame A into the Data Input channel of the FFT. As the FFT is loading this data to process it, the buffer in the channel never fills. However, the master immediately starts sending data for Frame B. At point A in the waveform, the buffer in the Data Input channel fills, because the FFT is processing frame A and no longer draining the buffer. This can be seen externally as **s_axis_data_tready** going Low. The Data Input channel remains in a slave waitstate situation, where the FFT cannot accept data from the upstream Master, until point B. Now the FFT has unloaded frame A and started loading Frame B into the processing core. This drains the buffer in the Data Input channel, which unblocks the Upstream Master and allows it to send the remaining data for Frame B. The situation then repeats itself with Frame C.

The important points here are:

1. Activity on the AXI interface to the Data Input channel does not necessarily correlate to the activity inside the FFT. For example, just before point A, the channel loads sample data for frame B yet the FFT is internally processing Frame A.
2. The Upstream Master cannot always stream frame data without reference to **s_axis_data_tready**.
3. The FFT unloads a frame before loading the subsequent frame.

Figure 3-45 is similar to Figure 3-44, except that the FFT is configured to have bit reversed outputs. As the upstream master is always supplying data, the loading and unloading of frames can overlap.

Figure 3-46 is similar to Figure 3-45, except that the upstream master does not supply data for Frame B until the core has started flushing out Frame A. As the core has already started flushing Frame A, it completes this before loading Frame B. The loading and unloading of frames do not overlap.

In this example, **s_axis_data_tready** remains High at Point A. Loading Frame A into the core drained the buffer in the Data Input channel, and because the Upstream Master did not send any new data, the buffer is empty. The core is ready to accept new frame data at point A although it is not able to do anything with it at this point. At point B the Upstream Master starts to send data from Frame B. This fills the buffer in the Data Input channel, but because the core is committed to flushing Frame A, the buffer fills and the core stalls the Upstream Master with waitstates. At point C, the core has started loading Frame B to process it, so the buffer drains and more data can be accepted to finish off Frame B.

The key difference between the situation in Figure 3-45 and Figure 3-46 is that the master in Figure 3-45 has provided new frame data during the processing phase of the previous frame. As a result, the core knows there is a new frame coming so when processing finishes, it starts to load the new frame as this flushes the old frame out. In Figure 3-46, the master did not provide data (and therefore did not tell the core that there would be a new frame) during the processing phase, so when the core finishes processing the frame, it moves to a flushing phase where it is no longer possible to load a new frame. Even if the master provides a sample for the new frame a cycle after unloading has begun, that sample is not loaded until the core is finished unloading the old frame.

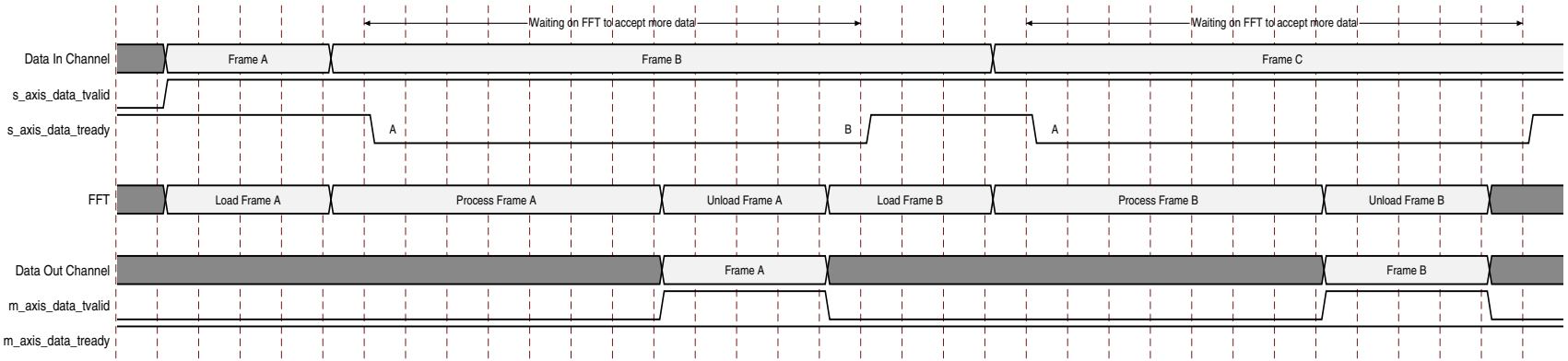

Figure 3-44: **Transform Timing for Entire Frames in Burst I/O Mode with Natural Ordered Outputs**

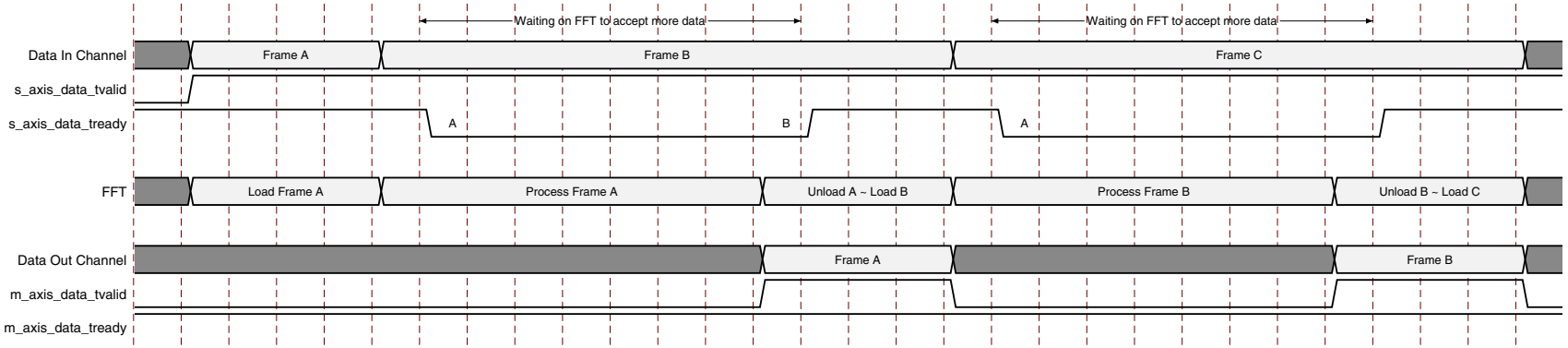

Figure 3-45: **Transform Timing for Entire Frames in Burst I/O Mode with Bit-Reversed Outputs**

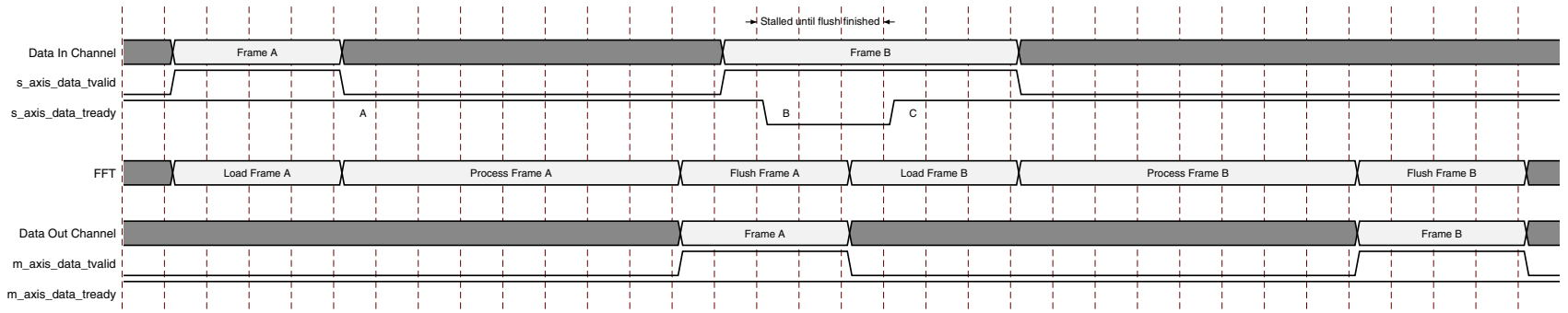

Figure 3-46: **Transform Timing for Entire Frames In Burst I/O Mode with Bit-Reversed Outputs (Core Flushes Frame)**

Configuring the FFT

FFT transforms are configured using the Configuration channel. The configuration information carried in this channel, and how it is packed, is discussed in more detail in Configuration Channel. When the core is ready to load a new frame for processing, it checks to see if a new configuration has been supplied on the Configuration channel. If it has, the FFT processing core is configured using that information before the frame is loaded. If no new configuration information has been supplied then the core processes the frame using the last configuration it had. If no configuration has ever been supplied, then the core defaults described in Resets are used.

The process of applying configuration data to a particular frame depends on the current status of the core:

1. To apply a configuration to the very first frame after power on or after an idle period
2. To apply the configuration to the next frame in a sequence of frames

Applying a New Configuration While Idle

If the core is idle (that is, it is not loading, processing or unloading any frames), it waits for either frame data or configuration data to decide what action to take next. If new frame data is seen by the core control module without new configuration information being seen, then the core starts to process a frame using the existing configuration. If configuration information is seen before frame data, or on the same clock edge as frame data, then the configuration is applied to that frame.

To ensure that the configuration data is applied before the frame is processed, the configuration information should be written to the Configuration channel where the write of configuration data to the Configuration channel must complete at least 1 clock cycle before the write of the first Data Input channel. Failure to do so can result in the frame being processed with the previous configuration options in use.

Perhaps the easiest way to satisfy this in a system context is to configure the core before enabling the upstream data master.

Applying a New Configuration While Streaming Frames

When the upstream master is active and sending frame data to the core, it becomes difficult to use the previous technique to synchronize configuration with particular frames because data for a new frame might have already been loaded into the Data Input channel. The recommended way of synchronizing configuration to frames is to use the **`event_frame_started`** signal.

This signal is asserted High when the core starts to load data for a frame into the FFT processing core. This is a known safe point to send configuration information for the next frame. Configuration data sent after this might or might not be applied to the subsequent

frame, depending on the frame size and the latency between `event_frame_started` asserting and the configuration write occurring.

How Changing the Configuration Can Change Transform Timing

There are two situations where changing the configuration can temporarily reduce the throughput of the core:

1. A Pipelined Streaming FFT is processing frames and the transform size (NFFT) is changed.
2. A Burst I/O core with bit reversed outputs is processing a frame, and the master supplies frame data in time to avoid the core automatically flushing the frame, and the transform size (NFFT) is changed.

Both the Pipelined Streaming architecture and the Burst I/O architectures (when bit reversed outputs are used) implement pipelining to achieve better throughput. In the case of the Pipelined Streaming architecture, it pipelines the loading, processing and unloading of entire frames (see Figure 3-42). In Burst I/O architectures when bit reversed outputs are used, the core implements a partial pipeline to overlap the loading on one frame with the unloading of another (see Figure 3-45).

However, a change to the transform size can only be applied when the pipeline is empty. Changing the transform size when the pipeline is not empty would result in data loss, so the core prevents this. When new configuration information is sent to the Configuration channel, and that information contains a change in transform size, the core does not load more frames until all frames already in the pipeline are processed and unloaded.

This is all handled automatically by the core, allowing you to send the configuration information at any time. However, throughput drops until the pipeline is fully flushed. This behavior only occurs if the transform size is to change. All other configuration options can be applied without waiting for the core pipeline to empty.

Design Flow Steps

This chapter describes customizing and generating the core, constraining the core, and the simulation, synthesis and implementation steps that are specific to this IP core. More detailed information about the standard Vivado® design flows and the IP integrator can be found in the following Vivado Design Suite user guides:

- *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) [Ref 7]
- *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 8]
- *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 9]
- *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 10]

Customizing and Generating the Core

This section includes information about using Xilinx® tools to customize and generate the core in the Vivado® Design Suite.

If you are customizing and generating the core in the Vivado IP integrator, see the *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) [Ref 7] for detailed information. IP integrator might auto-compute certain configuration values when validating or generating the design. To check whether the values do change, see the description of the parameter in this chapter. To view the parameter value you can run the **`validate_bd_design`** command in the Tcl Console.

You can customize the IP for use in your design by specifying values for the various parameters associated with the IP core using the following steps:

1. Select the IP from the IP catalog.
2. Double-click the selected IP or select the Customize IP command from the toolbar or right-click menu.

For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 8] and the *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 9].

The Vivado Integrated Design Environment (IDE) provides several FFT core customization screens with fields to set the parameter values for the particular instantiation required. A description of each field follows:

- **Component Name**: The name of the core component to be instantiated. The name must begin with a letter and be composed of the following characters: a to z, A to Z, 0 to 9, and "_".

Configuration Tab

- **Channels**: Select the number of channels from 1 to 12. Multichannel operation is available for the three Burst I/O architectures.
- **Transform Length**: Select the desired point size. All powers of two from 8 to 65536 are available.
- **Implementation Options**: Select an implementation option, as described in Architecture Options.
  - The Pipelined Streaming I/O, Radix-2 Burst I/O, and Radix-2 Lite Burst I/O architectures support point sizes 8 to 65536.
  - The Radix-4 Burst I/O architecture supports point sizes 64 to 65536.
  - Check Automatically Select to choose the smallest implementation that meets the specified Target Data Throughput, provided the specified Target Clock Frequency is achieved when the FFT core is implemented on an FPGA.
  - Target Clock Frequency and Target Data Throughput are only used to automatically select an implementation and to calculate latency. The core is not guaranteed to run at the specified target clock frequency or target data throughput.
- **Transform Length Options**: Select the transform length to be run time configurable or not. The core uses fewer logic resources and has a faster maximum clock speed when the transform length is not run time configurable.

Implementation Tab

- **Data Format**: Select whether the input and output data samples are in Fixed-Point format, or in IEEE-754 single precision (32-bit) Floating-Point format. Floating-Point format is not available when the core is in a multichannel configuration.
- **Precision Options**: Input data and phase factors can be independently configured to widths from 8 to 34 bits, inclusive. When the Data Format is Floating-Point, the input data width is fixed at 32 bits and the phase factor width can be set to 24 or 25 bits depending on the noise performance required and available resources.

- **Scaling Options**: Three options are available, for all architectures:
  - Unscaled
    - All integer bit growth is carried to the output. This can use more FPGA resources.
  - Scaled
    - A user-defined scaling schedule determines how data is scaled between FFT stages.
  - Block Floating-Point
    - The core determines how much scaling is necessary to make best use of available dynamic range, and reports the scaling factor as a block exponent.
- **Control Signals**: Clock Enable (**aclken**) and Synchronous Clear (**aresetn**) are optional pins. Synchronous Clear overrides Clock Enable if both are selected. If an option is not selected, some logic resources can be saved and a higher clock frequency might be attainable.
- **Optional Output Fields**: **XK_INDEX** is an optional field in the Data Output Channel. **OVFLO** is an optional field in both the Data Output channel and Status Channel.
- **Throttle Schemes**: Select trade-off between performance and data timing requirements. Realtime mode typically gives a smaller and faster design, but has strict constraints on when data must be provided and consumed. Non-Realtime mode has no such constraints, but the design might be larger and slower. See Controlling the FFT Core for more details.
- **Rounding Modes**: At the output of the butterfly, the LSBs in the datapath need to be trimmed. These bits can be truncated or rounded using convergent rounding, which is an unbiased rounding scheme. When the fractional part of a number is equal to exactly one-half, convergent rounding rounds up if the number is odd, and rounds down if the number is even. Convergent rounding can be used to avoid the DC bias that would otherwise be introduced by truncation after the butterfly stages. Selecting this option increases slice usage and yields a small increase in transform time due to additional latency.
- **Output Ordering**: Output data selections are either Bit/Digit Reversed Order or Natural Order. The Radix-2 based architectures (Pipelined Streaming I/O, Radix-2 Burst I/O and Radix-2 Lite Burst I/O) offer bit-reversed ordering, and the Radix-4 based architecture (Radix-4 Burst I/O) offers digit-reversed ordering. For the Pipelined Streaming I/O architecture, selecting natural order output ordering results in an increase in memory used by the core. For Burst I/O architectures, selecting natural order output increases the overall transform time because a separate unloading phase is required.
  - Cyclic Prefix Insertion can be selected if the output ordering is Natural Order. Cyclic Prefix Insertion is available for all architectures, and is typically used in OFDM wireless communications systems.

Detailed Implementation Tab

- **Memory Options**:
  - **Data And Phase Factors (Burst I/O architectures)**: For Burst I/O architectures, either block RAM or distributed RAM can be used for data and phase factor storage. Data and phase factor storage can be in distributed RAM for all point sizes up to and including 1024 points.
  - **Data And Phase Factors (Pipelined Streaming I/O)**: In the Pipelined Streaming I/O solution, the data can be stored partially in block RAM and partially in distributed RAM. Each pipeline stage, counting from the input side, uses smaller data and phase factor memories than preceding stages. You can select the number of pipeline stages that use block RAM for data and phase factor storage. Later stages use distributed RAM. The default displayed on the IDE offers a good balance between both. If output ordering is Natural Order, the memory used for the reorder buffer can be either block RAM or distributed RAM. The reorder buffer can use distributed RAM for point sizes less than or equal to 1024.
    - When block floating-point is selected for the Pipelined Streaming I/O architecture, a RAM buffer is required for natural order *and* bit reversed order output data. In this case, the reorder buffer options remain available and distributed RAM can be selected for all point sizes below 2048.
  - **Hybrid Memories**: Where data, phase factor, or reorder buffer memories are stored in block RAM, if the size of the memory is greater than one block RAM, the memory can be constructed from a hybrid of block RAMs and distributed RAM, where the majority of the data is stored in block RAMs and a few bits that are left over are stored in distributed RAM. This Hybrid Memory is an alternative to constructing the memory entirely from multiple block RAMs. It provides a reduction in the block RAM count, at the cost of an increase in the number of slices used. Hybrid Memories are only available when block RAM is used for one or more memories and the number of slices required for a Hybrid Memory implementation is below an internal threshold of 256 LUTs per memory. If these conditions are met, Hybrid Memories are made available and can be selected.
- **Optimize Options**:
  - **Complex Multipliers**: Three options are available for customization of the complex multiplier implementation:
    - **Use CLB logic**: All complex multipliers are constructed using slice logic. This is appropriate for target applications that have low performance requirements, or target devices that have few DSP Slices.
    - **Use 3-multiplier structure (resource optimization)**: All complex multipliers use a three real multiply, five add/subtract structure, where the multipliers use DSP Slices. This reduces the DSP Slice count, but uses some slice logic. This structure can make use of the DSP Slice pre-adder to reduce or remove the need for extra slice logic, and improve performance.

- **Use 4-multiplier structure (performance optimization)**: All complex multipliers use a four real multiply, two add/subtract structure, utilizing DSP Slices. This structure yields the highest clock performance at the expense of more dedicated multipliers. In devices with DSP Slices, the add/subtract operations are implemented within the DSP Slices.

Note: The core might override the complex multiplier implementation internally to ensure the fewest number of DSP Slices are used, without impacting performance. For this reason, some core configurations might show no difference in DSP Slice usage when toggling between the 3-multiplier and 4-multiplier options. If **Use CLB logic** is selected, however, slice logic is always used.

- **Butterfly Arithmetic**: Two options are available for customization of the butterfly implementation:
  - **Use CLB logic**: All butterfly stages are constructed using slice logic.
  - **Use XtremeDSP Slices:** For devices with DSP Slices, this option forces all butterfly stages to be implemented using the adder/subtracters in DSP Slices.

Information Tabs

- Implementation Details:
  - **Implementation**: This field displays the currently selected architecture. This is useful to see the result of automatic architecture selection.
  - **Transform Size**: When the transform length is run time configurable, the core has the ability to reprogram the point size while the core is running; that is, the core can support the selected point size and any smaller point size. This field displays the supported point sizes based on the Transform Length, Transform Length Options, and the Implementation Options selected.
  - **Output Data Width**: The output data width equals the input data width for scaled arithmetic and block floating-point arithmetic. With unscaled arithmetic, the output data width equals (input data width + $\log_2$(point size) + 1).
  - **Resource Estimates**: Based on the options selected, this field displays the DSP Slice count and 18K block RAM numbers (9K block RAM numbers for Spartan-6 devices). The resource numbers are just an estimate. For exact resource usage, and slice/LUT-FlipFlop pair information, a MAP report should be consulted.
  - **AXI4-Stream Port Structure:** This section shows how the FFT fields are mapped to the AXI channels.
- **Latency:**
  - This tab shows the latency of the FFT core in clock cycles and microseconds (µs) for each point size supported. The latency is from the Upstream Master supplying the first sample of a frame to the last sample of output data coming out of the core, assuming that the FFT core was idle and neither the Upstream Master or the Downstream Slave inserted wait states. This is not the minimum number of cycles between starting consecutive frames, as frames might overlap in some cases. The latency in microseconds is based on the target clock frequency.

User Parameters

Table 4-1 shows the relationship between the fields in the Vivado IDE and the User Parameters (which can be viewed in the Tcl Console).

Table 4-1: **Vivado IDE Parameter to User Parameter Relationship**

Vivado IDE Parameter/Value[1]	User Parameter/Value[1]	Default Value
Number of Channels	channels	1
Transform Length	transform_length	1024
Target Clock Frequency (MHz)	target_clock_frequency	250
Architecture Choice	implementation_options	
Automatically Select	automatically_select	Automatically_Select
Pipelined, Streaming I/O	pipelined_streaming_io	
Radix-2Lite, Burst I/O	radix_2_lite_burst_io	
Radix-2, Burst I/O	radix_2_burst_io	
Radix-4 Burst I/O	radix_4_burst_io	
Target Data Throughput	target_data_throughput	50
Run Time Configurable Transform Length	run_time_configurable_transform_length	False
Data Format	data_format	Fixed_point
Fixed-Point	fixed_point	
Floating-Point	floating_point	
Input Data Width	input_width	16
Phase Factor Width	phase_factor_width	16
Scaling Options	scaling_options	Scaled
Block Floating-Point	block_floating_point	
Scaled	scaled	
Unscaled	unscaled	
Rounding Modes	rounding_modes	Truncation
Convergent Rounding	convergent_rounding	
Truncation	truncation	
ACLKEN	aclken	False
ARESETn	aresetn	False
OVFLO	ovflo	False
XK_INDEX	xk_index	False
Throttle Scheme	throttle_scheme	Nonrealtime
Non Real Time	nonrealtime	
Real Time	realtime	

Table 4-1: **Vivado IDE Parameter to User Parameter Relationship *(Cont'd)***

Vivado IDE Parameter/Value[1]	User Parameter/Value[1]	Default Value
Output Ordering	output_ordering	Bit_reversed_order
Bit/Digit Reversed Order	bit_reversed_order	
Natural Order	natural_order	
Cyclic Prefix Insertion	cyclic_prefix_insertion	False
Memory Options: Data	memory_options_data	Block_ram
Block RAM	block_ram	
Distributed RAM	distributed_ram	
Memory Options: Phase Factors	memory_options_phase_factors	Block_ram
Block RAM	block_ram	
Distributed RAM	distributed_ram	
Reorder Buffer	memory_options_reorder	Block_ram
Block RAM	block_ram	
Distributed RAM	distributed_ram	
Number of stages using Block RAM for Data and Phase Factors	number_of_stages_using_block_ram_for_data_and_phase_factors	1
Optimize Block RAM Count Using Hybrid Memories	memory_options_hybrid	False
Complex Multipliers	complex_mult_type	Use_mult_resources
Use 3-multiplier structure (resource optimization)	use_mults_resources	
Use 4-multiplier structure (performance optimization)	use_mults_performance	
Use CLB Logic	use_luts	
Butterfly Arithmetic	butterfly_type	Use_luts
Use CLB Logic	use_luts	
Use XtremeDsp Slices	use_xtremedsp_slices	

Notes:

1. Parameter values are listed in the table where the IDE parameter value differs from the user parameter value. Such values are shown in this table as indented below the associated parameter.

Output Generation

For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 8].

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 IDE. See Customizing and Generating the Core for more detailed information about all other parameters.

Tab 1: Basic

The Basic tab is used to specify the transform configuration and architecture in a similar way to page 1 of the Vivado IDE.

Implementation Options: Select an implementation option as described in Architecture Options.

- The Pipelined Streaming I/O, Radix-2 Burst I/O, and Radix-2 Lite Burst I/O architectures support point sizes 8 to 65536.
- The Radix-4 Burst I/O architecture supports point sizes 64 to 65536.

System Generator supports only single-channel implementation of the FFT and, hence, Channels is not available as a GUI option.

Tab 2: Advanced

The Advanced tab is used to specify phase factor precision, scaling, rounding, optional output fields, throttle scheme, and optional port options in a similar way to page 2 of the Vivado IDE.

System Generator can optionally shorten the AXI4-Stream signal names on the symbol by removing the **`m_axis_`** or **`s_axis_`** prefixes.

System Generator automatically sets the Input Data Width parameter based on the signal properties of the **`XN_RE`** and **`XN_IM`** ports.

Tab 3: Implementation

The Implementation tab is used to specify memory and optimization options in a similar way to page 3 of the Vivado IDE.

- **Number of stages using block RAM**: Specifies the number of stages for the Pipelined Streaming I/O architecture that uses block RAM for data and phase factor storage. As dynamic list boxes are not offered with the System Generator GUI, this option displays the full range (0 to 11) selection, but allows you to select only valid values as visible in the Vivado IDE.
- **FPGA Area Estimation**: See the *System Generator for DSP User Guide (UG640)* [Ref 11] for detailed information about this option.

Constraining the Core

This section contains information about constraining the core in the Vivado Design Suite.

Required Constraints

This section is not applicable for this IP core.

Device, Package, and Speed Grade Selections

This section is not applicable for this IP core.

Clock Frequencies

This section is not applicable for this IP core.

Clock Management

This section is not applicable for this IP core.

Clock Placement

This section is not applicable for this IP core.

Banking

This section is not applicable for this IP core.

Transceiver Placement

This section is not applicable for this IP core.

I/O Standard and Placement

This section is not applicable for this IP core.

Simulation

For comprehensive information about Vivado simulation components, as well as information about using supported third-party tools, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 10].

IMPORTANT: *For cores targeting 7 series or Zynq-7000 devices, UNIFAST libraries are not supported. Xilinx IP is tested and qualified with UNISIM libraries only.*

Synthesis and Implementation

For details about synthesis and implementation, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 8].

C Model

The Xilinx® LogiCORE™ IP Fast Fourier Transform (FFT) core has a bit-accurate C model designed for system modeling. A MATLAB® software MEX function for seamless MATLAB software integration is also available.

Features

- Bit accurate with FFT core
- Dynamic link library
- Available for 64-bit Linux and 64-bit Windows platforms
- MATLAB software MEX function
- Supports all features of the FFT core that affect numerical results
- Designed for rapid integration into a larger system model
- Example C++ and M code showing how to use the function is provided

Overview

The Xilinx LogiCORE IP FFT has a bit-accurate C model for 64-bit Linux and 64-bit Windows platforms. The model has an interface consisting of a set of C functions, which resides in a dynamic link library (shared library). Full details of the interface are given in FFT C Model Interface. An example piece of C++ code showing how to call the model is provided. The model is also available as a MATLAB software MEX function for seamless MATLAB software integration.

The model is bit accurate but not cycle accurate, so it produces exactly the same output data as the core on a frame-by-frame basis. However, it does not model the core latency or its interface signals. The C model is an optional output of the Vivado® Design Suite. For information about generating IP source outputs, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 8].

Unpacking and Model Contents

Unzip the FFT C model zip file. This produces the directory structure and files shown in Table 5-1.

Table 5-1: **Files for the FFT Bit-Accurate C Model**

File	Description
xfft_v9_1_bitacc_cmodel.h	Model header file
libIp_xfft_v9_1_bitacc_cmodel.so	Model shared object library (Linux platforms only)
libIp_xfft_v9_1_bitacc_cmodel.dll	Model dynamically linked library (Windows platforms only)
libIp_xfft_v9_1_bitacc_cmodel.lib	Model library file for static linking (Windows platforms only)
libgmp.so.11	MPIR library, used by the C model (Linux platforms only)
libgmp.dll	MPIR library, used by the C model (Windows platforms only)
libgmp.lib	MPIR.lib file for compiling (Windows platforms only)
gmp.h	MPIR header file, used by the C model
run_bitacc_cmodel.c	Example code calling the C model
xfft_v9_1_bitacc_mex.cpp	C++ wrapper for MEX function
make_xfft_v9_1_mex.m	MEX function compilation script
run_xfft_v9_1_mex.m	Example code calling the MEX function

Installation

On Linux, ensure that the directory in which the files **`libIp_xfft_v9_1_bitacc_cmodel.so`** and **`libgmp.so.11`** are located is in your $LD_LIBRARY_PATH environment variable.

On Windows, ensure that the directory in which the files **`libIp_xfft_v9_1_bitacc_cmodel.dll`** and **`libgmp.dll`** are located is either in your %PATH% environment variable, or is the directory in which you runs your executable that calls the FFT C model.

Software Requirements

The FFT C model and MEX function were compiled and tested with the software in Table 5-2.

Table 5-2: **Supported Software**

Platform	C++ Compiler	MATLAB Software
64-bit Linux	GCC 4.1.1(1)(2)	R2012b
64-bit Windows	Microsoft Visual Studio 2012(1)	R2012b

Notes:

1. M file scripts are provided in the zip file to allow the MEX function to be compiled for other versions of MATLAB software, and versions on different operating systems. See FFT MATLAB Software MEX Function for details.
2. MATLAB software requires at least GCC version 4.0.0 to build MEX functions for Linux platforms.

FFT C Model Interface

Note: An example C++ file, **`run_bitacc_cmodel.c`** is included that demonstrates how to call the FFT C model. See this file for examples of using the interface described below.

The C model is used through three functions, declared in the header file **`xfft_v9_1_bitacc_cmodel.h`**:

```
struct xilinx_ip_xfft_v9_1_state* xilinx_ip_xfft_v9_1_create_state
(
 struct xilinx_ip_xfft_v9_1_generics generics
);

int xilinx_ip_xfft_v9_1_bitacc_simulate
(
 struct xilinx_ip_xfft_v9_1_state* state,
 struct xilinx_ip_xfft_v9_1_inputs inputs,
 struct xilinx_ip_xfft_v9_1_outputs* outputs
);

void xilinx_ip_xfft_v9_1_destroy_state
(
 struct xilinx_ip_xfft_v9_1_state* state
);
```

To use the model, first create a state structure using the first function, **`xilinx_ip_xfft_v9_1_create_state`**. Then run the model using the second function, **`xilinx_ip_xfft_v9_1_bitacc_simulate`**, passing the state structure, an inputs structure, and an outputs structure to the function. Finally, free up memory allocated for the state structure using the third function, **`xilinx_ip_xfft_v9_1_destroy_state`**. Each of these functions is described fully in the following sections.

Create a State Structure

The first function, **`xilinx_ip_xfft_v9_1_create_state`**, creates a new state structure for the FFT C model, allocating memory to store the state as required, and returns a pointer to that state structure. The state structure contains all information required to define the FFT being modeled. The function is called with a structure containing the core generics; these are all of the parameters that define the bit-accurate numerical performance of the core, represented as integers, and are shown in Table 5-3.

Table 5-3: **FFT C Model Generics**

Generic	Description	Permitted Values	
C_NFFT_MAX	$\log_2$(maximum transform length)	3-16	
C_ARCH	Architecture	1 = Radix-4, Burst I/O 2 = Radix-2, Burst I/O 3 = Pipelined, Streaming I/O 4 = Radix-2 Lite, Burst I/O	
C_HAS_NFFT	Run time configurable transform length	0 = no, 1 = yes	
C_USE_FLT_PT	Core interface	0 = Fixed-Point	1 = Single-Precision Floating-Point
C_INPUT_WIDTH	Input data width (bits)	8-34	32
C_TWIDDLE_WIDTH	Phase factor width (bits)	8-34	24-25
C_HAS_SCALING	Scaling option: unscaled or determined by C_HAS_BFP	0 = unscaled, 1 = see C_HAS_BFP	0
C_HAS_BFP	Scaling option: Applicable if C_HAS_SCALING=1	0 = scaled, 1 = block floating-point	0
C_HAS_ROUNDING	Rounding:	0 = truncation, 1 = convergent rounding	0

Note: C_CHANNELS is not a generic used in the C model. The model is always single channel. To model multiple channels in a multichannel FFT, see Modeling Multichannel FFTs.

The **`xilinx_ip_xfft_v9_1_create_state`** function fails with an error message and returns a NULL pointer if any generic or combination of generics is invalid.

Simulate the FFT Core

After a state structure has been created, it can be used as many times as required to simulate the FFT core. A simulation is run using the second function, **`xilinx_ip_xfft_v9_1_bitacc_simulate`**. Call this function with the pointer to the existing state structure and structures that hold the inputs and outputs of the C model. The input structure members are shown in Table 5-4.

Table 5-4: **Members of the Inputs Structure**

Member	Type	Description
nfft	int	Transform length
xn_re	double*	Pointer to array of doubles: real part of input data
xn_re_size	int	Number of elements in xn_re array
xn_im	double*	Pointer to array of doubles: imaginary part of input data
xn_im_size	int	Number of elements in xn_im array
scaling_sch	int*	Pointer to array of ints containing scaling schedule
scaling_sch_size	int	Number of elements in scaling_sch array
direction	int	Transform direction: 1=forward FFT, 0=inverse FFT (IFFT)

The notes under the following headings apply to the inputs structure.

- General
- FFTs with Fixed-Point Interface
- FFTs with Floating-Point Interface

General

1. You are responsible for allocating memory for arrays in the inputs structure.
2. **nfft** input is only used with run time configurable transform length (that is, C_HAS_NFFT = 1). If the transform length is fixed (C_HAS_NFFT = 0), C_NFFT_MAX is used for **nfft**. In this case, **nfft** should be equal to C_NFFT_MAX, and a warning is printed if it is not (but the model continues, using C_NFFT_MAX for **nfft** and ignoring the **nfft** value in the inputs structure).
3. **xn_re** and **xn_im** must have 2^{nfft} elements. **xn_re_size** and **xn_im_size** must be set to 2^{nfft}.
4. **xn_re** and **xn_im** can be in natural or bit/digit-reversed sample index order. The C model produces samples in the same ordering format as they were input.

FFTs with Fixed-Point Interface

1. Data in **xn_re** and **xn_im** must all be in the range $-1.0 \leq \text{data} < +1.0$.
2. Data in **xn_re** and **xn_im** is of type double, but the model requires data in signed two's-complement, fixed-point format with precision given by C_INPUT_WIDTH. The data has a sign bit, then the binary point, and then (C_INPUT_WIDTH - 1) fractional bits. The model checks the input data to see if it fits within this format and precision. If not, it prints a warning, and internally rounds it using convergent rounding (halfway values are rounded to the nearest even number) to the required precision. To accurately model the FFT core, pre-quantize the input data to this required precision before passing it to the model.

3. **scaling_sch** and **scaling_sch_size** are ignored entirely unless fixed scaling is used (C_HAS_SCALING = 1 and C_HAS_BFP = 0).
4. **scaling_sch** is an array of integers, each of which indicates the scaling to be done in a stage of the FFT processing. **scaling_sch[0]** is the scaling in the first stage, **scaling_sch[1]** the scaling in the second stage, and so on. Note that this is the reverse of the scaling schedule vector applied to the IP core. The number of elements in the **scaling_sch** array and the value of **scaling_sch_size** must be equal to the number of stages in the FFT. This is dependent on the architecture, and on **nfft**, the point size of the transform:
   a. Radix-4, Burst I/O (C_ARCH = 1) or Pipelined, Streaming I/O (C_ARCH = 3): stages = ceil(nfft/2)
   b. Radix-2, Burst I/O (C_ARCH = 2) or Radix-2 Lite, Burst I/O (C_ARCH = 4): stages = nfft
5. If C_HAS_NFFT = 0, C_NFFT_MAX is used for **nfft**. The scaling in each stage is an integer in the range 0-3, which indicates the number of bits the intermediate result is shifted right. So 0 indicates no scaling, 1 indicates a division by 2, 2 indicates a division by 4, and 3 indicates a division by 8. Again, **scaling_sch[0]** is the scaling in the first stage, **scaling_sch[1]** the scaling in the second stage, and so on. Insufficiently large scaling results in overflow, indicated by the overflow output.

FFTs with Floating-Point Interface

1. Data in **xn_re** and **xn_im** must all be representable in IEEE-754 single-precision 32-bit format.
2. Data in **xn_re** and **xn_im** is of type double, but the model requires data in single-precision format, such that the values can be represented in the 32-bit float datatype. The double values are explicitly cast to the float datatype internally. No range checking is performed by the model prior to casting to float.
3. The model checks the input data for denormalized numbers, and if one is found, that sample is set to zero at the input to the model.
4. If an Infinity or Not A Number (NaN) value is detected in the input data, all outputs in that frame are invalidated and set to NaN in the output structure.

The outputs structure, a pointer which is passed to the **xilinx_ip_xfft_v9_1_bitacc_simulate** function, has the members shown in Table 5-5.

Table 5-5: **Members of the Outputs Structure**

Member	Type	Description
xk_re	double*	Pointer to array of doubles: real part of output data
xk_re_size	int	Number of elements in xk_re array
xk_im	double*	Pointer to array of doubles: imaginary part of output data

Table 5-5: **Members of the Outputs Structure *(Cont'd)***

Member	Type	Description
xk_im_size	int	Number of elements in xk_im array
blk_exp	int	Block exponent (if block floating-point is used)
overflow	int	Overflow occurred (if fixed scaling is used with a fixed-point interface, or if a floating-point interface is used)

The notes under the following headings apply to the outputs structure.

- General
- FFTs with Fixed-Point Interface
- FFTs with Floating-Point Interface

General

1. You are responsible for allocating memory for the outputs structure and for arrays in the outputs structure.
2. **xk_re** and **xk_im** must have at least 2^{nfft} elements. You must set **xk_re_size** and **xk_im_size** to indicate the number of elements in **xk_re** and **xk_im** before calling the FFT function. On exit, **xk_re_size** and **xk_im_size** are set to the number of elements that contain valid output data in **xk_re** and **xk_im**.
3. The C model produces data in the same ordering format as the input data. Hence, if **xn_re** and **xn_im** were provided in natural sample index order (0,1,2,3...), **xk_re** and **xk_im** samples will also be in natural sample index order.

FFTs with Fixed-Point Interface

1. Data in **xk_re** and **xk_im** has the correct precision to model the FFT:
   a. If the FFT is scaled or has block floating-point (C_HAS_SCALING = 1, C_HAS_BFP = 0 or 1, respectively), data in **xk_re** and **xk_im** is all in the range $-1.0 \leq$ data $< +1.0$, the precision is C_INPUT_WIDTH bits with C_INPUT_WIDTH-1 fractional bits. For example, if C_INPUT_WIDTH = 8, output data is precise to 2^{-7} = 0.0078125 and is in the range -1.0 to +0.9921875, and the binary representation of the output data has the format s.fffffff, where s is the sign bit and f is a fractional bit.
   b. If the FFT is unscaled (C_HAS_SCALING = 0), data in **xk_re** and **xk_im** grows beyond ± 1.0, such that the binary point remains in the same place and there are still (C_INPUT_WIDTH - 1) fractional bits after the binary point. In total, the output precision is (C_INPUT_WIDTH + C_NFFT_MAX + 1) bits. For example, if C_INPUT_WIDTH = 8 and C_NFFT_MAX = 3, output data is precise to 2^{-7} = 0.0078125 and is in the range -16.0 to +15.9921875, and the binary representation of the output data has the format siiii.fffffff, where s is the sign bit, i is an integer bit, and f is a fractional bit.

2. **blk_exp** is the integer block exponent. It is only valid (and non-zero) if block floating-point is used (C_HAS_SCALING = 1 and C_HAS_BFP = 1). It indicates the total number of bits that intermediate values were shifted right during the FFT processing. For example, if **blk_exp** = 5, the output data has been divided by 32 relative to the magnitude of the input data.
3. **overflow** indicates if overflow occurred during the FFT processing. It is only valid (and non-zero) if fixed scaling is used (C_HAS_SCALING = 1 and C_HAS_BFP = 0). A value of 0 indicates that overflow did not occur; a value of 1 indicates that overflow occurred at some stage in the FFT processing. To avoid overflow, increase the scaling at one or more stages in the scaling schedule (**scaling_sch** input).
4. If overflow occurred with the Pipelined, Streaming I/O architecture (C_ARCH = 3) due to differences between the FFT core and the model in the order of operations within the processing stage, the data in **xk_re** and **xk_im** might not match the XK_RE and XK_IM outputs of the FFT core. The **xk_re** and **xk_im** data must be ignored if the overflow output is 1. This is the only case where the model is not entirely bit accurate to the core.

FFTs with Floating-Point Interface

1. Data in **xk_re** and **xk_im** has the correct precision to model the FFT. The double-precision output can be cast to single-precision without introducing error.
2. Overflow indicates if floating-point exponent overflow occurred during the FFT processing. A value of 0 indicates that overflow did not occur; a value of 1 indicates that overflow occurred. Overflow is not set when a NaN value is present on the output. NaN values can only occur at the FFT output when the input data frame contains samples with value NaN or ± Infinity.
3. If overflow occurred, the output sample that overflowed is set to ± Infinity, depending on the sign of the internal result.

The **xilinx_ip_xfft_v9_1_bitacc_simulate** function checks the input and output structures for errors. If the model finds a problem, it prints an error message and returns a value **xilinx_ip_xfft_v9_1_bitacc_simulate** function as shown in Table 5-6.

Table 5-6: **xilinx_ip_xfft_v9_1_bitacc_simulate Function Return Values**

Return Value	Meaning
0	Success.
1	state structure is NULL.
2	outputs structure is NULL.
3	state structure is corrupted (Radix-4, Burst I/O architecture).
4	state structure is corrupted (Radix-2 [Lite], Burst I/O architecture).
5	state structure is corrupted (Pipelined, Streaming I/O architecture).
6	nfft in inputs structure out of range (Radix-4, Burst I/O architecture).
7	nfft in inputs structure out of range (other architectures).

Table 5-6: **xilinx_ip_xfft_v9_1_bitacc_simulate Function Return Values *(Cont'd)***

Return Value	Meaning
8	xn_re in inputs structure is a NULL pointer.
9	xn_re_size in inputs structure is incorrect.
10	data value in xn_re in inputs structure out of range -1.0 to < +1.0 (fixed-point input data only).
11	xn_im in inputs structure is a NULL pointer.
12	xn_im_size in inputs structure is incorrect.
13	data value in xn_im in inputs structure is out of range -1.0 to < +1.0 (fixed-point input data only).
14	scaling_sch in inputs structure is a NULL pointer.
15	scaling_sch_size in inputs structure is incorrect (Radix-4, Burst I/O or Pipelined, Streaming I/O architectures).
16	scaling_sch_size in inputs structure is incorrect (Radix-2, Burst I/O or Radix-2 Lite, Burst I/O architectures).
17	scaling value in scaling_sch in inputs structure out of range 0-3.
18	scaling value for final stage in scaling_sch in inputs structure out of range 0-1 when nfft is odd and architecture is Radix-4, Burst I/O or Pipelined, Streaming I/O.
19	direction in inputs structure is out of range 0-1.
20	xk_re in outputs structure is a NULL pointer.
21	xk_im in outputs structure is a NULL pointer.
22	xk_re_size in outputs structure is too small.
23	xk_im_size in outputs structure is too small.

If the **`xilinx_ip_xfft_v9_1_bitacc_simulate`** function returns 0 (zero), it completed successfully and the outputs of the model are in the outputs structure.

Destroy the State Structure

Finally, the state structure must be destroyed to free up memory used to store the state, using the third function, **`xilinx_ip_xfft_v9_1_destroy_state`**, called with the pointer to the existing state structure.

If the generics of the core need to be changed, destroy the existing state structure and create a new state structure using the new generics. There is no way to change the generics of an existing state structure.

C Model Example Code

An example C++ file, **run_bitacc_cmodel.c**, is provided. This demonstrates the steps required to run the model: set up generics, create a state structure, create inputs and outputs structures, simulate the FFT, use the outputs, and finally destroy the state structure. The code works for all legal combinations of generics; modify the const int declarations of generics at the start of function main(). The code also shows how to model a multichannel FFT; see Modeling Multichannel FFTs.

The example code can be used to test your compilation process. See Compiling with the FFT C Model.

Compiling with the FFT C Model

Place the header files, **xfft_v9_1_bitacc_cmodel.h** and **gmp.h**, with your other header files.

Compilation varies from platform to platform.

Linux

To compile the example code, **run_bitacc_cmodel.c**, first ensure that the directory in which the files **libIp_xfft_v9_1_bitacc_cmodel.so** and **libgmp.so.11** are located is present on your $LD_LIBRARY_PATH environment variable. These shared libraries are referenced during the compilation and linking process.

Place the header file and C++ source file in a single directory. Then in that directory, compile using the GNU C++ Compiler:

```
gcc -x c++ -I. -L. -lIp_xfft_v9_1_bitacc_cmodel -Wl,-rpath,. -o
run_bitacc_cmodel run_bitacc_cmodel.c
```

Windows

When compiling on Windows, the symbol NT must be defined either by a compiler option or in user source code before the **xfft_v9_1_bitacc_cmodel.h** header file is included.

Link to the import libraries **libIp_xfft_v9_1_bitacc_cmodel.lib** and **libgmp.lib**. For example, for Microsoft Visual Studio.NET, in **Project Properties**, under **Linker > Input**, for Additional Dependencies, specify **libIp_xfft_v9_1_bitacc_cmodel.lib** and **libgmp.lib**.

FFT MATLAB Software MEX Function

The FFT model is available as a MATLAB® software MEX function for seamless integration with MATLAB software. The FFT MEX function provides a MATLAB software interface to the FFT C model. The FFT MEX function and FFT C model produce identical results, and both are bit accurate to the FFT core.

Building the MEX Function

A C++ wrapper and compilation script are provided to allow the MEX function to be built for your MATLAB software version and operating system.

The FFT C model does not support the LCC compiler shipped with MATLAB software.

Xilinx has verified that GCC version 4.1.1 can successfully be used to build the MEX function on 64-bit Linux.

To build the MEX function:

1. Start the MATLAB software.
2. Change directory to the unzipped FFT C model installation.
3. Use the **`mex -setup`** command at the MATLAB software command line to set up the compiler. For more details on the mex command and the arguments it accepts, type **`help mex`** at the MATLAB software command line
4. Execute the **`make_xfft_v9_1_mex.m`** file in the MATLAB software to build the MEX function.
5. Verify that a file named **`xfft_v9_1_bitacc_mex.mex<suffix>`** is now present in the current directory. The **`<suffix>`** portion of the file name depends on the platform on which the function was compiled.

Installing and Running the MEX Function

To install the FFT MEX function, place the MEX file in your MATLAB software working directory, or in the MATLAB software, change directory to the directory containing the MEX file.

Note: For Windows platforms, the correct **`libIp_xfft_v9_1_bitacc_cmodel.dll`** and **`libgmp.dll`** files must be copied to the directory where the FFT MEX function has been installed before running the MEX function.

Note: For Linux platforms, the **`libIp_xfft_v9_1_bitacc_cmodel.so`** and **`libgmp.so.11`** files must be copied to the directory where the FFT MEX function has been installed before running the MEX function, or be visible using the $LD_LIBRARY_PATH environment variable. $LD_LIBRARY_PATH must be set correctly before starting MATLAB software.

The FFT MEX function is called **`xfft_v9_1_bitacc_mex`**. Enter this function name without arguments at the MATLAB software command line to see usage information. The FFT MEX function syntax is:

```
[output_data, blk_exp, overflow] = xfft_v9_1_bitacc_mex(generics, nfft, input_data,
scaling_sch, direction)
```

The function inputs are shown in Table 5-7.

Table 5-7: **FFT MEX Function Inputs**

Input	Description	Permitted values	
generics	Core parameters. Single-element, 9-field structure containing all relevant generics defining the core		
generics.C_NFFT_MAX	$\log_2$(maximum transform length)	3-16	
generics.C_ARCH	Architecture	1 = Radix-4, Burst I/O, 2 = Radix-2, Burst I/O, 3 = Pipelined, Streaming I/O, 4 = Radix-2 Lite, Burst I/O	
generics.C_HAS_NFFT	Run time configurable transform length	0 = no, 1 = yes	
generics.C_USE_FLT_PT	Core interface	0 = fixed-point	1 = single-precision floating-point
generics.C_INPUT_WIDTH	Input data width	8-34 bits	32 bits
generics.C_TWIDDLE_WIDTH	Phase factor width	8-34 bits	24-25 bits
generics.C_HAS_SCALING	Type of scaling	0 = unscaled, 1 = other	0
generics.C_HAS_BFP	Type of scaling if C_HAS_SCALING = 1	0 = scaled, 1 = block floating-point	0
generics.C_HAS_ROUNDING	Type of rounding	0 = truncation, 1 = convergent rounding	0
nfft	$\log_2$(transform length) for this transform. Single integer.	Maximum value is C_NFFT_MAX. Minimum value is 6 for Radix-4, Burst I/O architecture, or 3 for other architectures.	Maximum value is C_NFFT_MAX. Minimum value is 6 for Radix-4, Burst I/O architecture, or 3 for other architectures.
input_data	Input data. 1D array of complex data with 2^{nfft} elements.	All components must be in the range of $-1.0 \leq$ data $< +1.0$.	All components must be representable in the MATLAB software single datatype (equivalent to a float in C++).

Table 5-7: **FFT MEX Function Inputs *(Cont'd)***

Input	Description	Permitted values	
scaling_sch	Scaling schedule. 1D array of integer values size S = number of stages. For Radix-4 and Streaming architectures, S = nfft/2, rounded up to the next integer. For Radix-2 and Radix-2 Lite architectures, S = nfft.	Each value corresponds to scaling to be performed by the corresponding stage and must be in the range 0 to 3. scaling_sch[1] is the scaling for the first stage.	N/A
direction	Transform direction. Single integer.	1 = forward FFT, 0 = inverse FFT (IFFT)	1 = forward FFT, 0 = inverse FFT (IFFT)

The following notes apply to the MEX function inputs.

1. **`nfft`** input is only used for run time configurable transform length (that is, generics.C_HAS_NFFT = 1). It is ignored otherwise and generics. C_NFFT_MAX is used instead.
2. For fixed-point input FFTs (that is, generics.C_USE_FLT_PT = 0), to ensure identical numerical behavior to the hardware, pre-quantize the input_data values to have precision determined by C_INPUT_WIDTH. This is achieved using the MATLAB software built-in quantize function.
3. **`scaling_sch`** input is only used for a fixed-point input, scaled FFT (that is, generics.C_USE_FLT_PT = 0, generics.C_HAS_SCALING = 1, and generics. C_HAS_BFP = 0). It is ignored otherwise.
4. **`input_data`** can be in natural or bit/digit-reversed sample index order. The MEX function produces samples in the same ordering format as they were input.

The function outputs are shown in Table 5-8.

Table 5-8: **FFT MEX Function Outputs**

Output	Description	Validity
output_data	Output data. 1D array of complex data with 2^{nfft} elements.	Always valid.
blk_exp	Block exponent. Single integer.	Only valid if using block floating-point (if generics.C_HAS_SCALING = 1 and C_HAS_BFP = 1). Zero otherwise.
overflow	Overflow. Single integer. 1 indicates overflow occurred, 0 indicates no overflow occurred.	Only valid with a scaled FFT (if generics.C_HAS_SCALING = 1 and generics.C_HAS_BFP = 0) or an FFT with floating-point interfaces (that is, generics.C_USE_FLT_PT = 1). Zero otherwise.

The notes under the following headings apply to the MEX function outputs.

- General
- FFTs with Fixed-Point Interface
- FFTs with Floating-Point Interface

General

1. There is no need to create and destroy state, as must be done with the C model; this is handled internally by the FFT MEX function.
2. The FFT MEX function performs extensive checking of its inputs. Any invalid input results in a message reporting the error and the function terminates.
3. The MEX function produces data in the same order as the input data. Hence, if input_data was provided in natural sample index order (0,1,2,3...), output_data samples will also be in natural sample index order.

FFTs with Fixed-Point Interface

1. Input data is an array of complex double-precision floating-point data, but the FFT core being modeled requires data in signed two's-complement, fixed-point format with precision given by C_INPUT_WIDTH. The data has a sign bit, then the binary point, then (C_INPUT_WIDTH - 1) fractional bits. The FFT MEX function checks the input data to see if it fits within this format and precision. If not, it prints a warning, and internally rounds it using convergent rounding (halfway values are rounded to the nearest even number) to the required precision. To accurately model the FFT core, pre-quantize the input data to this required precision before passing it to the model. This can be done using the MATLAB software built-in quantize function.

   Type **`help quantizer/quantize`** or **`doc quantize`** on the MATLAB software command line for more information.

2. Output data has the correct precision to model the FFT:
   a. If the FFT is scaled or has block floating-point (that is, C_HAS_SCALING = 1, C_HAS_BFP = 0 or 1, respectively), output data is all in the range $-1.0 \leq$ data $< +1.0$, the precision is C_INPUT_WIDTH bits, with C_INPUT_WIDTH-1 fractional bits. For example, if C_INPUT_WIDTH = 8, output data is precise to 2^{-7} = 0.0078125 and is in the range -1.0 to +0.9921875, and the binary representation of the output data has the format s.fffffff, where s is the sign bit and f is a fractional bit.
   b. If the FFT is unscaled (C_HAS_SCALING = 0), output data grows beyond ± 1.0, such that the binary point remains in the same place and there are still (C_INPUT_WIDTH - 1) fractional bits after the binary point. In total, the output precision is (C_INPUT_WIDTH + C_NFFT_MAX + 1) bits. For example, if C_INPUT_WIDTH = 8 and C_NFFT_MAX = 3, output data is precise to 2^{-7} = 0.0078125 and is in the range -16.0 to +15.9921875, and the binary representation of the output data has the format siiii.fffffff, where s is the sign bit, i is an integer bit, and f is a fractional bit.

3. **blk_exp** is the integer block exponent. It is only valid (and non-zero) if block floating-point is used (that is, C_HAS_SCALING = 1 and C_HAS_BFP = 1). It indicates the total number of bits that intermediate values were shifted right during the FFT processing. For example, if **blk_exp** = 5, the output data has been divided by 32 relative to the magnitude of the input data.
4. **overflow** indicates if overflow occurred during the FFT processing. It is only valid (and non-zero) if fixed scaling is used (that is, C_HAS_SCALING = 1 and C_HAS_BFP = 0). A value of 0 indicates that overflow did not occur; a value of 1 indicates that overflow occurred at some stage in the FFT processing. To avoid overflow, increase the scaling at one or more stages in the scaling schedule (**scaling_sch** input).
5. If overflow occurred with the Pipelined, Streaming I/O architecture (C_ARCH = 3) due to differences between the FFT core and the model in the order of operations within the processing stage, the output data might not match the XK_RE and XK_IM outputs of the FFT core. The output data must be ignored if the overflow output is 1. This is the only case where the model is not entirely bit accurate to the core.

FFTs with Floating-Point Interface

1. Input data is an array of complex double-precision floating-point data, but the FFT core being modeled requires values in single-precision (32-bit) format. The data must therefore be representable in the MATLAB software 'single' datatype, even if it is represented in the 'double' datatype. The value is explicitly cast to the C++ 'float' datatype inside the MEX function.
2. Output data has the correct precision to model the FFT. The double-precision output array contains single-precision values which are representable in the MATLAB software 'single' datatype without error.
3. **overflow** indicates if exponent overflow has occurred during the FFT processing. A value of 0 indicates that overflow did not occur; a value of 1 indicates that exponent overflow did occur.
4. If overflow occurred, the output sample that overflowed is set to ± Infinity, depending on the sign of the internal result.

MEX Function Example Code

An example M file, **run_xfft_v9_1_mex.m**, is provided. This demonstrates the steps required to run the MEX function: set up generics, create input data, simulate the FFT, and use the outputs. The code works for all legal combinations of generics. Modify the declarations of generics at the top of the file. The code also shows how to model a multichannel FFT; see Modeling Multichannel FFTs.

The example code can be used to test your MEX function compilation process. See Building the MEX Function.

Modeling Multichannel FFTs

The FFT C model and FFT MEX function are single-channel models that do not directly model multichannel FFTs. However, it is very simple to model multichannel FFTs.

By definition, a multichannel FFT is equivalent to multiple identical single-channel FFTs, each operating on different input data. Therefore a multichannel FFT can be modeled by running a single-channel model several times on the input data of each channel.

For the FFT C model, the example C++ code provided, **`run_bitacc_cmodel.c`**, demonstrates how to model a multichannel FFT. This example code creates the FFT state structure, then uses a loop to run the model on each channel's input data in turn, then finally destroys the state structure. For the FFT MEX function, call the function on the input data of each channel in turn.

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 2.6.0

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 12] and MPIR [Ref 13] 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.

Test Bench

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

Demonstration Test Bench

When the core is generated using the Vivado Design Suite, a demonstration test bench is created. This is a simple VHDL test bench that exercises the core.

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

Using the Demonstration Test Bench

The demonstration test bench instantiates the generated Fast Fourier Transform core. Compile the netlist and the demonstration test bench into the work library (see your simulator documentation for more information on how to do this). Then simulate the demonstration test bench. View the test bench signals in your simulator waveform viewer to see the operations of the test bench.

Demonstration Test Bench in Detail

The demonstration test bench performs the following tasks:

- Instantiates the core
- Generates an input data frame consisting of one or the sum of two complex sinusoids
- Generates a clock signal
- Drives the core input signals to demonstrate core features
- Checks that the core output signals obey AXI protocol rules (data values are not checked to keep the test bench simple)
- Provides signals showing the separate fields of AXI TDATA and TUSER signals

The demonstration test bench drives the core input signals to demonstrate the features and modes of operation of the core. This includes performing an FFT on a pre-generated input data frame. The input data frame consists of a complex sinusoid with a frequency of 2.6 times the frame size. The FFT of this input frame is a peak centred between output samples 2 and 3. For FFTs with a maximum point size of 64 or greater, the input data is modified by adding a second complex sinusoid with a frequency of 23.2 times the frame size and a quarter of the magnitude of the first sinusoid. This modifies the FFT by adding a smaller peak centred between output samples 23 and 24. The test bench captures this output frame and uses it as the input frame for an inverse transform. The output of this inverse transform is therefore the same as the original input frame (modified by the scaling and finite precision effects of the FFT core).

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:

- Frame 1: drive a frame of pre-generated input data
- Frame 2: configure an inverse transform; drive the output of frame 1 as a frame of input data
- Configure frame 3: a forward transform while the previous transform is running
- Frame 3: drive the output of frame 2 as a frame of input data; deassert AXI TVALID (and TREADY if present) signals occasionally to demonstrate AXI handshaking
- If ARESETn present: start another frame but reset the core before it completes
- Frames 4-7: run these back-to-back, as quickly as possible:
    - Queue up configurations for a forward transform (frame 4) followed by a reverse transform (frame 5), both with a smaller point size (if point size is configurable) and a short cyclic prefix (if available)
    - Frame 4: drive a frame of pre-generated input data
    - Frame 5: drive the output of frame 1 as a frame of input data; simultaneously configure frame 6: a forward transform with maximum point size, a longer cyclic prefix (if available) and a zero scaling schedule (if fixed scaling is used)
    - Frame 6: drive a frame of pre-generated input data; simultaneously configure frame 7: an inverse transform with maximum point size, no cyclic prefix and default scaling schedule (if fixed scaling is used)
    - Frame 7: drive the output of frame 1 as a frame of input data
- Wait until all frames are complete

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.

Input data is pre-generated in the **create_ip_table** function and stored in the **IP_DATA** constant. New input data frames can be added by defining new functions and constants. Make sure that each input data frame is of the **T_IP_TABLE** array type.

All operations performed by the demonstration test bench to drive the core's inputs are done in the **data_stimuli** process. This process also contains procedures to simplify driving a frame of input data. Configuration is requested in this process by setting **cfg_*** signals to the desired configuration and setting the **do_config** shared variable to either **IMMEDIATE** or **AFTER_START**. The configuration signals are actually driven by the **config_stimuli** process.

The **data_stimuli** process is comprehensively commented, to explain clearly what is being done. New configuration and data 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.

Upgrading

This appendix contains information about migrating a design from the ISE® Design Suite to the Vivado® Design Suite, and for upgrading to a more recent version of the IP core. For customers upgrading in the Vivado Design Suite, important details (where applicable) about any port changes and other impact to user logic are included.

Migrating to the Vivado Design Suite

For information about migrating to the Vivado Design Suite, *see the ISE to Vivado Design Suite Migration Guide* (UG911) [Ref 14].

Upgrading in the Vivado Design Suite

This section provides information about any changes to the user logic or port designations that take place when you upgrade to a more current version of this IP core in the Vivado Design Suite.

Parameter Changes

There are no parameter changes between versions 9.0 and 9.1.

Port Changes

There are no port changes between versions 9.0 and 9.1.

Functionality Changes

Latency Changes

The majority of configurations in version 9.1 have unchanged latency compared to version 9.0.

The exception is the Pipelined Streaming I/O architecture when configured in Block Floating-Point scaling mode. In this case, the latency will increase in version 9.1 by 1 cycle compared to version 9.0.

See Information Tabs for the definition of latency.

Numerical Behavior Changes

The majority of configurations in version 9.1 have unchanged numerical behavior compared to version 9.0.

The exception is the Pipelined Streaming I/O architecture when configured in Block Floating-Point scaling mode. In this case, the output data may no longer be bit accurate with previous versions due to the introduction of a rounding stage. The IP remains bit accurate with the C model, however user test vectors generated from earlier versions of the C model or IP may require updating to match the new behavior.

Debugging

This appendix includes details about resources available on the Xilinx Support website and debugging tools.

Finding Help on Xilinx.com

To help in the design and debug process when using the Fast Fourier Transform core, the Xilinx Support web page contains key resources such as product documentation, release notes, answer records, information about known issues, and links for obtaining further product support.

Documentation

This product guide is the main document associated with the Fast Fourier Transform core. This guide, along with documentation related to all products that aid in the design process, can be found on the Xilinx Support web page or by using the Xilinx® Documentation Navigator.

Download the Xilinx Documentation Navigator from the Downloads page. For more information about this tool and the features available, open the online help after installation.

Answer Records

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 users have access to the most accurate information available.

Answer Records for this core can be located by using the Search Support box on the main Xilinx support web page. To maximize your search results, use 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 Fast Fourier Transform

AR: 54501

Technical Support

Xilinx provides technical support at the Xilinx Support web page for this LogiCORE™ IP product when used as described in the product documentation. Xilinx cannot guarantee timing, functionality, or support if you do any of the following:

- Implement the solution in devices that are not defined in the documentation.
- Customize the solution beyond that allowed in the product documentation.
- Change any section of the design labeled DO NOT MODIFY.

To contact Xilinx Technical Support, navigate to the Xilinx Support web page.

Debug Tools

There are tools available to address Fast Fourier Transform design issues. It is important to know which tools are useful for debugging various situations.

Vivado Design Suite Debug Feature

The Vivado® Design Suite debug feature inserts logic analyzer and virtual I/O cores directly into your design. The debug feature also allows you to set trigger conditions to capture application and integrated block port signals in hardware. Captured signals can then be analyzed. This feature in the Vivado IDE is used for logic debugging and validation of a design running in Xilinx devices.

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

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

See the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 15].

Reference Boards

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

C Model Reference

See *Chapter 5, C Model* in this guide for tips and instructions for using the C Model files provided to debug your design.

Simulation Debug

The simulation debug flow for Mentor Graphics Questa Advanced Simulator is illustrated in Figure B-1. A similar approach can be used with other simulators.

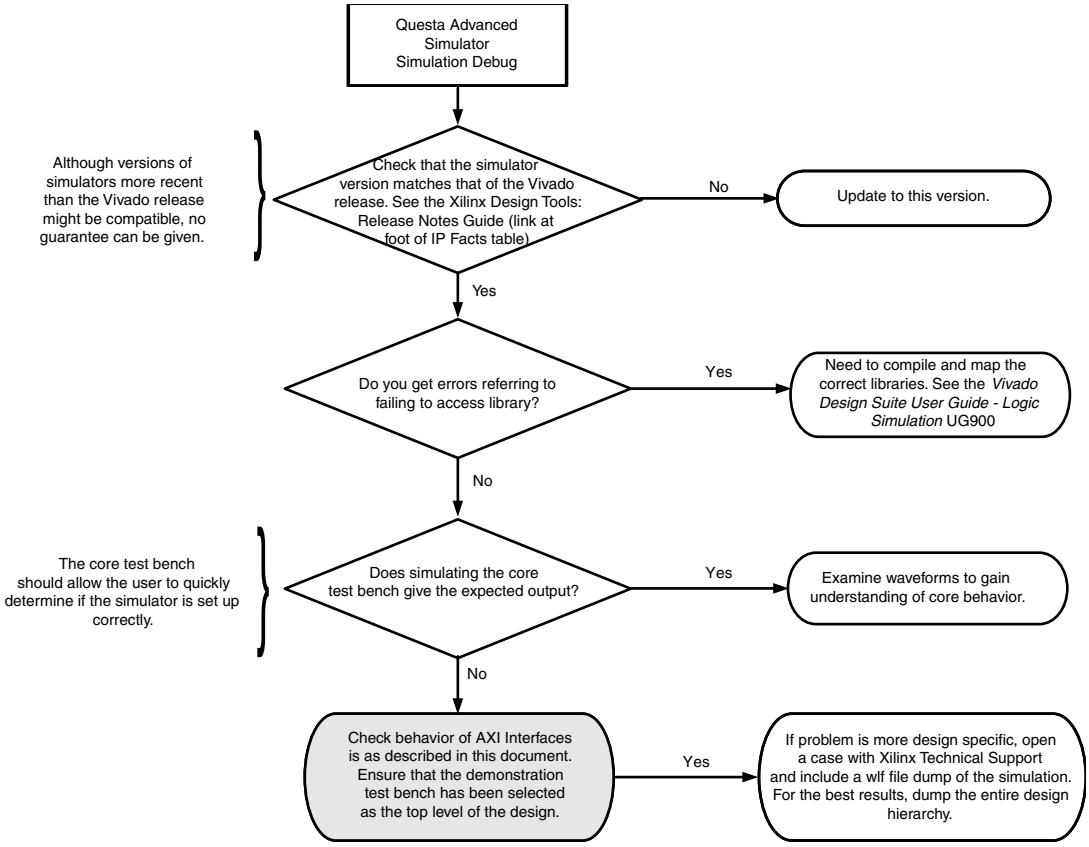

Figure B-1: **Questa Advanced Simulator Debug Flow Diagram**

AXI4-Stream Interface Debug

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 and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (DocNav):

- From the Vivado® IDE, select **Help > Documentation and Tutorials**.
- On Windows, select **Start > All Programs > Xilinx Design Tools > DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In the Xilinx Documentation Navigator, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

Note: For more information on Documentation Navigator, see the Documentation Navigator page on the Xilinx website.

References

These documents provide supplemental material useful with this product guide:

1. *AMBA® AXI4-Stream Protocol Specification* (Arm IHI 0051A)
2. *Vivado Design Suite AXI Reference Guide* (UG1037)
3. W. R. Knight and R. Kaiser, *A Simple Fixed-Point Error Bound for the Fast Fourier Transform*, IEEE Trans. Acoustics, Speech and Signal Proc., Vol. 27, No. 6, pp. 615-620, December 1979.
4. L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing,* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1975.
5. Quang Hung Nguyen and Istvan Kollar, *Limited Dynamic Range of Spectrum Analysis Due To Round off Errors Of The FFT*, available at: home.mit.bme.hu/~kollar/papers/IMTC-FFT.pdf
6. I. Szolik, K. Kovac, V. Smiesko, *Influence of Digital Signal Processing on Precision of Power Quality Parameters Measurement*, available at: www.measurement.sk/2003/S1/Szolik.pdf.
7. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
8. *Vivado Design Suite User Guide: Designing with IP* (UG896)
9. *Vivado Design Suite User Guide: Getting Started* (UG910)
10. *Vivado Design Suite User Guide - Logic Simulation* (UG900)
11. *System Generator for DSP User Guide* (UG640)
12. The GNU Multiple Precision Arithmetic (GMP) Library: gmplib.org/
13. The Multiple Precision Integers and Rationals (MPIR) Library: www.mpir.org/
14. *ISE® to Vivado Design Suite Migration Guide* (UG911)
15. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
06/17/2020	9.1	• Updated configuration data paragraph in Applying a New Configuration While Idle. • Removed libgmpxx.so.4.
05/22/2019	9.1	Minor changes to Algorithm section; Table 3-11 and Transform Status section.
04/04/2018	9.1	Version change only.
10/04/2017	9.0	Added scaling factor information.
11/18/2015	9.0	UltraScale+ device support added.
09/30/2015	9.0	• Updated C model chapter to include details of MPIR libraries required by latest C model enhancements.
06/24/2015	9.0	C Model supported software updated.
04/01/2015	9.0	• Removed 32-bit support for the C model. • Clarified the compiler support for the C model.
10/01/2014	9.0	• Updated document to correct cross-reference error and clarify Resets information.
04/02/2014	9.0	• Added link to resource utilization numbers. • Add User Parameter table (Table 4-1).
12/18/2013	9.0	• Added UltraScale™ architecture support.
10/02/2013	9.0	• Revision number advanced to 9.0 to align with core version number 9.0. • C Model updates.
03/20/2013	1.0	Initial release as a Product Guide; replaces DS808 and UG459. No other documentation changes.

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.