



# **FFT MegaCore Function**

---

## **User Guide**



101 Innovation Drive  
San Jose, CA 95134  
[www.altera.com](http://www.altera.com)

MegaCore Version: 9.0  
Document Date: March 2009

Copyright © 2009 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

## Chapter 1. About This MegaCore Function

Release Information .....	1-1
Device Family Support .....	1-1
Features .....	1-2
General Description .....	1-3
Fixed Transform Size Architecture .....	1-3
Variable Streaming Architecture .....	1-3
OpenCore Plus Evaluation .....	1-4
DSP Builder Support .....	1-4
Performance and Resource Utilization .....	1-4
Cyclone III Devices .....	1-4
Stratix III Devices .....	1-8
Stratix IV Devices .....	1-11

## Chapter 2. Getting Started

Design Flow .....	2-1
FFT Walkthrough .....	2-2
Create a New Quartus II Project .....	2-2
Launch IP Toolbench .....	2-3
Step 1: Parameterize .....	2-4
Step 2: Set Up Simulation .....	2-8
Step 3: Generate .....	2-9
Simulate the Design .....	2-12
Simulate in the MATLAB Software .....	2-12
Fixed Transform Architectures .....	2-12
Variable Streaming Architecture .....	2-13
Simulate with IP Functional Simulation Models .....	2-14
Simulating in Third-Party Simulation Tools Using NativeLink .....	2-14
Compile the Design .....	2-15
Fixed Transform Architecture .....	2-15
Variable Streaming Architecture .....	2-16
Program a Device .....	2-16
Set Up Licensing .....	2-16

## Chapter 3. Functional Description

Buffered, Burst, & Streaming Architectures .....	3-1
Variable Streaming Architecture .....	3-2
The Avalon Streaming Interface .....	3-3
OpenCore Plus Time-Out Behavior .....	3-4
FFT Processor Engine Architectures .....	3-4
Radix-2 <sup>2</sup> Single Delay Feedback Architecture .....	3-5
Quad-Output FFT Engine Architecture .....	3-5
Single-Output FFT Engine Architecture .....	3-6

I/O Data Flow Architectures .....	3-7
Streaming .....	3-7
Streaming FFT Operation .....	3-7
Enabling the Streaming FFT .....	3-8
Variable Streaming .....	3-9
Change the Block Size .....	3-9
Enabling the Variable Streaming FFT .....	3-10
Dynamically Changing the FFT Size .....	3-10
The Effect of I/O Order .....	3-11
Buffered Burst .....	3-12
Burst .....	3-13
Parameters .....	3-14
Signals .....	3-16

## **Appendix A. Block Floating Point Scaling**

Introduction .....	A-1
Block Floating Point .....	A-1
Calculating Possible Exponent Values .....	A-2
Implementing Scaling .....	A-2
Achieving Unity Gain in an IFFT+FFT pair .....	A-4

## **Additional Information**


Revision History .....	1-1
How to Contact Altera .....	1-1
Typographic Conventions .....	1-1

## Release Information

Table 1–1 provides information about this release of the Altera® FFT MegaCore® function.

**Table 1–1.** Product Name Release Information

Item	Description
Version	9.0
Release Date	March 2009
Ordering Code	IP-FFT
Product ID	0034
Vendor ID	6AF7

 For more information about this release, refer to the [MegaCore IP Library Release Notes and Errata](#).

Altera verifies that the current version of the Quartus® II software compiles the previous version of each MegaCore® function. The [MegaCore IP Library Release Notes and Errata](#) report any exceptions to this verification. Altera does not verify compilation with MegaCore function versions older than one release."

## Device Family Support

MegaCore functions provide either full or preliminary support for target Altera device families:

- *Full support* means the MegaCore function meets all functional and timing requirements for the device family and may be used in production designs
- *Preliminary support* means the MegaCore function meets all functional requirements, but may still be undergoing timing analysis for the device family; it may be used in production designs with caution.

Table 1–2 shows the level of support offered by the FFT MegaCore function to each of the Altera device families.

**Table 1–2.** Device Family Support (Part 1 of 2)

Device Family	Support
Arria™ GX	Full
Arria II GX	Preliminary
Cyclone®	Full
Cyclone II	Full
Cyclone III	Full
HardCopy® II	Full

**Table 1–2.** Device Family Support (Part 2 of 2)

Device Family	Support
HardCopy III	Preliminary
HardCopy IV E	Preliminary
Stratix®	Full
Stratix II	Full
Stratix II GX	Full
Stratix III	Full
Stratix IV	Preliminary
Stratix GX	Full

## Features

- Bit-accurate MATLAB models
- DSP Builder ready
- Enhanced variable streaming FFT:
  - Single precision floating point or fixed point representation
  - Input and output orders include:
    - Natural order
    - Bit reversed
    - $-N/2$  to  $N/2$
  - Reduced memory requirements
  - Support for 8 to 32 bit data and twiddle width
  - DSP Builder fast functional simulation model
- Radix-4 and mixed radix-4/2 implementations
- Block floating-point architecture—maintains the maximum dynamic range of data during processing (not for variable streaming)
  - Uses embedded memory
  - Maximum system clock frequency >300 MHz
  - Optimized to use Stratix series DSP blocks and TriMatrix™ memory architecture
  - High throughput quad-output radix 4 FFT engine
  - Support for multiple single-output and quad-output engines in parallel
  - Multiple I/O data flow modes: streaming, buffered burst, and burst
- Avalon® Streaming (ST) compliant input and output interfaces
- Parameterization-specific VHDL and Verilog HDL testbench generation
- Transform direction (FFT/IFFT) specifiable on a per-block basis
- Easy-to-use IP Toolbench interface

- IP functional simulation models for use in Altera-supported VHDL and Verilog HDL simulators



For more information on Avalon ST interfaces, refer to the [Avalon Interface Specifications](#).

## General Description

The FFT MegaCore function is a high performance, highly-parameterizable Fast Fourier transform (FFT) processor. The FFT MegaCore function implements a complex FFT or inverse FFT (IFFT) for high-performance applications.

The FFT MegaCore function implements one of the following architectures:

- Fixed transform size architecture
- Variable streaming architecture

### Fixed Transform Size Architecture

The fixed transform architecture FFT implements a radix-2/4 decimation-in-frequency (DIF) FFT fixed-transform size algorithm for transform lengths of  $2^m$  where  $6 \leq m \leq 14$ . This architecture uses block-floating point representations to achieve the best trade-off between maximum signal-to-noise ratio (SNR) and minimum size requirements.

The fixed transform architecture accepts as an input, a two's complement format complex data vector of length  $N$ , where  $N$  is the desired transform length in natural order; the function outputs the transform-domain complex vector in natural order. An accumulated block exponent is output to indicate any data scaling that has occurred during the transform to maintain precision and maximize the internal signal-to-noise ratio. Transform direction is specifiable on a per-block basis via an input port.

### Variable Streaming Architecture

The variable streaming architecture FFT implements a radix-2<sup>2</sup> single delay feedback architecture, which you can configure during runtime to perform FFT algorithm for transform lengths of  $2^m$  where  $4 \leq m \leq 16$ . This architecture uses either a fixed-point representation or a single precision floating point representation. The fixed-point representation grows the data widths naturally from input through to output thereby maintaining a high SNR at the output. The single precision floating point representation allow a large dynamic range of values to be represented while maintaining a high SNR at the output.

For more information on radix-2<sup>2</sup> single delay feedback architecture, refer to *S. He and M. Torkelson, A New Approach to Pipeline FFT Processor, Department of Applied Electronics, Lund University, IPPS 1996*.

The order of the input data vector of size  $N$  can be natural, bit reversed, or DC-centered. The architecture outputs the transform-domain complex vector in natural or bit-reversed order. The transform direction is specifiable on a per-block basis using an input port.

## OpenCore Plus Evaluation

With Altera's free OpenCore Plus evaluation feature, you can perform the following actions:

- Simulate the behavior of a megafunction (Altera MegaCore function or AMPP<sup>SM</sup> megafunction) within your system
- Verify the functionality of your design, as well as evaluate its size and speed quickly and easily
- Generate time-limited device programming files for designs that include megafunctions
- Program a device and verify your design in hardware

You only need to purchase a license for the megafunction when you are completely satisfied with its functionality and performance, and want to take your design to production.



For more information on OpenCore Plus hardware evaluation using the FFT MegaCore function, see [“OpenCore Plus Time-Out Behavior” on page 3–4](#) and AN 320: *OpenCore Plus Evaluation of Megafunctions*.

## DSP Builder Support

Altera's DSP Builder shortens DSP design cycles by helping you create the hardware representation of a DSP design in an algorithm-friendly development environment.

You can combine existing MATLAB/Simulink blocks with Altera DSP Builder/MegaCore blocks to verify system level specifications and perform simulation. After installing this MegaCore function, a Simulink symbol of this MegaCore function appears in the Simulink library browser in the MegaCore library from the Altera DSP Builder blockset.

## Performance and Resource Utilization

Performance varies depending on the FFT engine architecture and I/O data flow. All data represents the geometric mean of a three seed Quartus II synthesis sweep.

### Cyclone III Devices

[Table 1–3](#) shows the streaming data flow performance, using the 4 mults/2 adders complex multiplier structure, for width 16, for Cyclone III (EP3C10F256C6) devices.

**Table 1–3.** Performance with the Streaming Data Flow Engine Architecture—Cyclone III Devices

Points	Combinational LUTs	Logic Registers	Memory (Bits)	Memory (M9K)	9 × 9 Mults	f <sub>MAX</sub> (MHz)	Clock Cycle Count	Transform Time (μs)
256	3,425	3,880	39,168	20	24	236	256	1.08
1,024	3,837	4,575	155,904	20	24	237	1,024	4.33
4,096 (1)	5,941	6,313	622,848	76	48	232	4,096	17.68

**Note to Table 1–3:**

(1) EP3C40F780C6 device.



Table 1–4 shows the variable streaming data flow performance, with in order inputs and bit-reversed outputs, for width 16 (32 for floating point), for Cyclone III (EP3C16F484C6) devices.



The variable streaming with fixed-point number representation uses natural word growth, therefore the multiplier requirement is larger compared with the equivalent streaming FFT with the same number of points.

If you want to significantly reduce M9K memory utilization, set a lower  $f_{\text{MAX}}$  target.

**Table 1–4.** Performance with the Variable Streaming Data Flow Engine Architecture—Cyclone III Devices

Point	Points	Combinational LUTs	Logic Registers	Memory (Bits)	Memory (M9K)	9 × 9 Mults	$f_{\text{MAX}}$ (MHz)	Clock Cycle Count	Transform Time ( $\mu\text{s}$ )
Fixed	256	3,976	4,173	10,309	17	48	190	256	1.35
Fixed	1,024	5,392	5,549	42,605	24	64	181	1,024	5.66
Fixed	4,096	6,865	6,873	172,006	46	80	176	4,096	23.22
Floating (1)	256	27,323	19,619	22,132	66	96	113	256	2.27
Floating (2)	1,024	34,508	24,436	80,912	89	128	114	1,024	8.99
Floating (2)	4,096	41,774	29,294	311,724	135	160	113	4,096	36.38

**Note to Table 1–4:**

(1) EP3C40F780C6 device.

(2) EP3C55F780C6 device.

Table 1–5 lists resource usage with buffered burst data flow architecture, using the 4 mults/2 adders complex multiplier structure, for data and twiddle width 16, for Cyclone III (EP3C25F324C6) devices.

**Table 1–5.** Resource Usage with Buffered Burst Data Flow Architecture—Cyclone III Devices (Part 1 of 2)

Points	Number of Engines (1)	Combinational LUTs	Logic Registers	Memory (Bits)	Memory (M9K)	9 × 9 Mults	$f_{\text{MAX}}$ (MHz)
256 (2)	1	3,118	3,738	30,976	16	24	254
1,024 (2)	1	3,208	3,930	123,136	16	24	238
4,096	1	3,287	4,108	491,776	60	24	234
256 (3)	2	5,114	5,892	30,976	31	48	244
1,024 (3)	2	4,207	5,023	175,392	20	24	220
4,096	2	5,299	6,280	491,776	60	48	231
256	4	8,904	10,620	30,976	60	96	215
1,024	4	9,030	10,839	123,136	60	96	206

**Table 1-5.** Resource Usage with Buffered Burst Data Flow Architecture—Cyclone III Devices (Part 2 of 2)

Points	Number of Engines (1)	Combinational LUTs	Logic Registers	Memory (Bits)	Memory (M9K)	9 × 9 Mults	f <sub>MAX</sub> (MHz)
4,096	4	9,144	11,039	491,776	60	96	207

**Notes to Table 1-5:**

- (1) When using the buffered burst architecture, you can specify the number of quad-output FFT engines in the FFT wizard.
- (2) EP3C10F256C6 device.
- (3) EP3C16F484C6 device.

Table 1-6 lists performance with buffered burst data flow architecture, using the 4 mults/2 adders complex multiplier structure, for data and twiddle width 16, for Cyclone III (EP3C25F324C6) devices.

**Table 1-6.** Performance with the Buffered Burst Data Flow Architecture—Cyclone III Devices

Points	Number of Engines (1)	fMAX (MHz)	Transform Calculation Time (2)		Data Load & Transform Calculation		Block Throughput (3)	
			Cycles	Time (μs)	Cycles	Time (μs)	Cycles	Time (μs)
256 (4)	1	254	235	0.93	491	1.93	331	1.3
1,024 (4)	1	238	1,069	4.49	2,093	8.8	1,291	5.43
4,096	1	234	5,167	22.04	9,263	39.51	6157	26.26
256 (5)	2	244	162	0.66	397	1.63	299	1.23
1,024 (5)	2	220	557	2.53	1,581	7.18	1,163	5.28
4,096	2	231	2,607	11.28	6,703	29.01	5,133	22.22
256	4	215	118	0.55	347	1.61	283	1.32
1,024	4	206	340	1.65	1,364	6.61	1,099	5.33
4,096	4	207	1,378	6.64	5,474	26.38	4,633	22.33

**Notes to Table 1-6:**

- (1) When using the buffered burst architecture, you can specify the number of quad-output engines in the FFT wizard. You may choose from one, two, or four quad-output engines in parallel.
- (2) In a buffered burst data flow architecture, transform time is defined as the time from when the N-sample input block is loaded until the first output sample is ready for output. Transform time does not include the additional N-1 clock cycle to unload the full output data block.
- (3) Block throughput is the minimum number of cycles between two successive start-of-packet (sink\_sop) pulses.
- (4) EP3C10F256C6 device.
- (5) EP3C16F484C6 device.

Table 1-7 lists resource usage with burst data flow architecture, using the 4 mults/2 adders complex multiplier structure, for data and twiddle width 16, for Cyclone III (EP3C10F256C6) devices.

**Table 1-7.** Resource Usage with the Burst Data Flow Architecture—Cyclone III Devices (Part 1 of 2)

Points	Engine Architecture	Number of Engines (2)	Combinational LUTs	Logic Registers	Memory (Bits)	Memory (M9K)	9 × 9 Mults
256	Quad Output	1	3,110	3,672	14,592	8	24
1,024	Quad Output	1	4,207	5,023	175,392	20	24
4,096	Quad Output	1	3,278	4,022	229,632	28	24
256	Quad Output	2	5,093	5,824	14,592	15	48

**Table 1-7.** Resource Usage with the Burst Data Flow Architecture—Cyclone III Devices (Part 2 of 2)

Points	Engine Architecture	Number of Engines (2)	Combinational LUTs	Logic Registers	Memory (Bits)	Memory (M9K)	9 × 9 Mults
1,024	Quad Output	2	5,189	6,016	57,600	15	48
4,096	Quad Output	2	5,270	6,192	229,632	28	48
256	Quad Output	4	8,906	10,556	14,592	28	96
1,024	Quad Output	4	9,017	10,765	57,600	28	96
4,096	Quad Output	4	9,128	10,955	229,632	28	96
256	Single Output	1	1,465	1,495	9,472	3	8
1,024	Single Output	1	1,528	1,541	37,120	6	8
4,096	Single Output	1	1,620	1,587	147,712	19	8
256	Single Output	2	2,079	2,406	14,592	9	16
1,024	Single Output	2	2,131	2,482	57,600	11	16
4,096	Single Output	2	2,194	2,558	229,632	28	16

**Notes to Table 1-7:**

- (1) When using the burst data flow architecture, you can specify the number of engines in the FFT wizard. You may choose from one to two single-output engines in parallel, or from one, two, or four quad-output engines in parallel.

Table 1-8 lists performance with burst data flow architecture, using the 4 mults/2 adders complex multiplier structure, for data and twiddle width 16, for Cyclone III (EP3C10F256C6) devices.

**Table 1-8.** Performance with the Burst Data Flow Architecture—Cyclone III Devices (Part 1 of 2)

Points	Engine Architecture	Number of Engines (1)	fMAX (MHz)	Transform Calculation Time (2)		Data Load & Transform Calculation		Block Throughput (3)	
				Cycles	Time (μs)	Cycles	Time (μs)	Cycles	Time (μs)
256	Quad Output	1	243	235	0.97	491	2.02	331	1.36
1,024	Quad Output	1	233	1,069	4.59	2,093	8.98	1,291	5.54
4,096	Quad Output	1	235	5,167	21.97	9,263	39.39	6,157	26.18
256	Quad Output	2	232	162	0.7	397	1.71	299	1.29
1,024	Quad Output	2	221	557	2.52	1,581	7.14	1,163	5.25
4,096	Quad Output	2	234	2,607	11.13	6,703	28.61	5,133	21.91
256	Quad Output	4	223	118	0.53	374	1.68	283	1.27
1,024	Quad Output	4	214	340	1.59	1,364	6.36	1,099	5.12
4,096	Quad Output	4	210	1,378	6.55	5,474	26.01	4,633	22.02
256	Single Output	1	261	1,115	4.28	1,371	5.26	1,628	6.25
1,024	Single Output	1	237	5,230	22.02	6,344	26.72	7,279	30.65
4,096	Single Output	1	236	24,705	104.62	28,801	121.97	32,898	139.32
256	Single Output	2	246	585	2.38	841	3.42	1,098	4.47
1,024	Single Output	2	240	2,652	11.03	3,676	15.29	4,701	19.55

**Table 1-8.** Performance with the Burst Data Flow Architecture—Cyclone III Devices (Part 2 of 2)

Points	Engine Architecture	Number of Engines (1)	fMAX (MHz)	Transform Calculation Time (2)		Data Load & Transform Calculation		Block Throughput (3)	
				Cycles	Time (μs)	Cycles	Time (μs)	Cycles	Time (μs)
4,096	Single Output	2	241	12,329	51.18	16,495	68.47	20,605	85.53

**Notes to Table 1-8:**

- (1) In the burst I/O data flow architecture, you can specify the number of engines in the FFT wizard. You may choose from one to two single-output engines in parallel, or from one, two, or four quad-output engines in parallel.
- (2) Transform time is the time frame when the input block is loaded until the first output sample (corresponding to the input block) is output. Transform time does not include the time to unload the full output data block.
- (3) Block throughput is defined as the minimum number of cycles between two successive start-of-packet (*sink\_sop*) pulses.

## Stratix III Devices

Table 1-9 shows the streaming data flow performance, using the 4 mults/2 adders complex multiplier structure, for data and twiddle width 16, for Stratix III (EP3SE50F780C2) devices.

**Table 1-9.** Performance with the Streaming Data Flow Engine Architecture—Stratix III Devices

Points	Combinational ALUTs	Logic Registers	Memory (Bits)	Memory (M9K)	18 × 18 Mults	f <sub>MAX</sub> (MHz)	Clock Cycle Count	Transform Time (μs)
256	2,141	3,688	39,168	20	12	397	256	0.64
1,024	2,434	4,383	155,904	20	12	406	1,024	2.52
4,096	3,732	5,929	622,848	76	24	361	4,096	11.35

Table 1-10 shows the variable streaming data flow performance, with in order inputs and bit-reversed outputs, for width 16 (32 for floating point), for Stratix III (EP3SE50F780C2) devices.



The variable streaming with fixed-point number representation uses natural word growth, therefore the multiplier requirement is larger compared with the equivalent streaming FFT with the same number of points.

If you want to significantly reduce M9K memory utilization, set a lower f<sub>MAX</sub> target.

**Table 1-10.** Performance with the Variable Streaming Data Flow Engine Architecture—Stratix III Devices

Point	Points	Combinational ALUTs	Logic Registers	Memory (Bits)	Memory (M9K)	18 × 18 Mults	f <sub>MAX</sub> (MHz)	Clock Cycle Count	Transform Time (μs)
Fixed	256	2,539	3,910	10,193	14	24	335	256	0.76
Fixed	1,024	3,513	5,231	42,377	21	32	333	1,024	3.08
Fixed	4096	4,540	6,545	171,611	40	40	304	4,096	13.46
Floating	256	17,724	19,941	24,959	66	48	215	256	1.19
Floating	1,024	22,342	24,758	84,798	87	64	214	1,024	4.78

**Table 1-10.** Performance with the Variable Streaming Data Flow Engine Architecture—Stratix III Devices

Point	Points	Combinational ALUTs	Logic Registers	Memory (Bits)	Memory (M9K)	18 × 18 Mults	f <sub>MAX</sub> (MHz)	Clock Cycle Count	Transform Time (μs)
Floating (1)	4,096	27,325	29,709	316,579	136	80	214	4,096	19.12

**Note to Table 1-10:**

(1) EP3SL70F780C2 device.

Table 1-11 lists resource usage with buffered burst data flow architecture, using the 4 mults/2 adders complex multiplier structure, for data and twiddle width 16, for Stratix III (EP3SE50F780C2) devices.

**Table 1-11.** Resource Usage with Buffered Burst Data Flow Architecture—Stratix III Devices

Points	Number of Engines (1)	Combinational ALUTs	Logic Registers	Memory (Bits)	Memory (M9K)	18 × 18 Mults	f <sub>MAX</sub> (MHz)
256	1	1,996	3,546	30,976	16	12	406
1,024	1	2,040	3,738	123,136	16	12	404
4,096	1	2,082	3,917	491,776	60	12	392
256	2	3,345	5,508	30,976	31	24	378
1,024	2	3,383	5,711	123,136	31	24	379
4,096	2	3,425	5,896	491,776	60	24	380
256	4	5,871	9,854	30,976	60	48	347
1,024	4	5,935	10,071	123,136	60	48	340
4,096	4	6,009	10,271	491,776	60	48	325

**Notes to Table 1-11:**

(1) When using the buffered burst architecture, you can specify the number of quad-output FFT engines in the FFT wizard.

Table 1-12 lists performance with buffered burst data flow architecture, using the 4 mults/2 adders complex multiplier structure, for data and twiddle width 16, for Stratix III (EP3SE50F780C2) devices.

**Table 1-12.** Performance with the Buffered Burst Data Flow Architecture—Stratix III Devices (Part 1 of 2)

Points	Number of Engines (1)	fMAX (MHz)	Transform Calculation Time (2)		Data Load & Transform Calculation		Block Throughput (3)	
			Cycles	Time (μs)	Cycles	Time (μs)	Cycles	Time (μs)
256	1	406	235	0.58	491	1.21	331	0.81
1,024	1	404	1,069	2.64	2,093	5.17	1,291	3.19
4,096	1	392	5,167	13.19	9,263	23.65	6157	15.72
256	2	378	162	0.43	397	1.05	299	0.79
1,024	2	379	557	1.47	1,581	4.17	1,163	3.07
4,096	2	380	2,607	6.86	6,703	17.64	5,133	13.51
256	4	347	118	0.34	347	1	283	0.82
1,024	4	340	340	1	1,364	4.01	1,099	3.23

**Table 1-12.** Performance with the Buffered Burst Data Flow Architecture—Stratix III Devices (Part 2 of 2)

Points	Number of Engines (1)	fMAX (MHz)	Transform Calculation Time (2)		Data Load & Transform Calculation		Block Throughput (3)	
			Cycles	Time (μs)	Cycles	Time (μs)	Cycles	Time (μs)
4,096	4	325	1,378	4.24	5,474	16.85	4,633	14.26

**Notes to Table 1-12:**

- (1) When using the buffered burst architecture, you can specify the number of quad-output engines in the FFT wizard. You may choose from one, two, or four quad-output engines in parallel.
- (2) In a buffered burst data flow architecture, transform time is defined as the time from when the N-sample input block is loaded until the first output sample is ready for output. Transform time does not include the additional N-1 clock cycle to unload the full output data block.
- (3) Block throughput is the minimum number of cycles between two successive start-of-packet (sink\_sop) pulses.

Table 1-13 lists resource usage with burst data flow architecture, using the 4 mults/2 adders complex multiplier structure, for data and twiddle width 16, for Stratix III (EP3SE50F780C2) devices.

**Table 1-13.** Resource Usage with the Burst Data Flow Architecture—Stratix III Devices

Points	Engine Architecture	Number of Engines (2)	Combinational ALUTs	Logic Registers	Memory (Bits)	Memory (M9K)	18 × 18 Mults
256	Quad Output	1	1,849	3,480	14,592	8	12
1,024	Quad Output	1	1,891	3,662	57,600	8	12
4,096	Quad Output	1	1,926	3,830	229,632	28	12
256	Quad Output	2	3,065	5,440	14,592	15	24
1,024	Quad Output	2	3,107	5,632	57,600	15	24
4,096	Quad Output	2	3,154	5,808	229,632	28	24
256	Quad Output	4	5,355	9,788	14,592	28	48
1,024	Quad Output	4	5,410	9,997	57,600	28	48
4,096	Quad Output	4	5,483	10,187	229,632	28	48
256	Single Output	1	712	1,431	9,472	3	4
1,024	Single Output	1	749	1,477	37,120	6	4
4,096	Single Output	1	816	1,523	147,712	19	4
256	Single Output	1	1,024	2,278	14,592	9	8
1,024	Single Output	2	1,037	2,354	57,600	11	8
4,096	Single Output	1	1,075	2,430	229,632	28	8

**Notes to Table 1-13:**

- (1) Represents data and twiddle factor precision.
- (2) When using the burst data flow architecture, you can specify the number of engines in the FFT wizard. You may choose from one to two single-output engines in parallel, or from one, two, or four quad-output engines in parallel.

Table 1-14 lists performance with burst data flow architecture, using the 4 mults/2 adders complex multiplier structure, for data and twiddle width 16, for Stratix III (EP3SE50F780C2) devices.

**Table 1-14.** Performance with the Burst Data Flow Architecture—Stratix III Devices

Points	Engine Architecture	Number of Engines (1)	fMAX (MHz)	Transform Calculation Time (2)		Data Load & Transform Calculation		Block Throughput (3)	
				Cycles	Time (μs)	Cycles	Time (μs)	Cycles	Time (μs)
256	Quad Output	1	421	235	0.56	491	1.17	331	0.79
1,024	Quad Output	1	409	1,069	2.62	2,093	5.12	1,291	3.16
4,096	Quad Output	1	381	5,167	13.57	9,263	24.32	6,157	16.17
256	Quad Output	2	368	162	0.44	397	1.08	299	0.81
1,024	Quad Output	2	386	557	1.44	1,581	4.09	1,163	3.01
4,096	Quad Output	2	379	2,607	6.88	6,703	17.7	5,133	13.55
256	Quad Output	4	346	118	0.34	374	1.08	283	0.82
1,024	Quad Output	4	341	340	1	1,364	4	1,099	3.22
4,096	Quad Output	4	327	1,378	4.22	5,474	16.76	4,633	14.19
256	Single Output	1	414	1,115	2.69	1,371	3.31	1,628	3.93
1,024	Single Output	1	407	5,230	12.85	6,344	15.58	7,279	17.88
4,096	Single Output	1	416	24,705	59.32	28,801	69.16	32,898	79
256	Single Output	2	413	585	1.42	841	2.04	1,098	2.66
1,024	Single Output	2	374	2,652	7.09	3,676	9.83	4,701	12.58
4,096	Single Output	2	391	12,329	31.53	16,495	42.18	20,605	52.69

**Notes to Table 1-14:**

- (1) In the burst I/O data flow architecture, you can specify the number of engines in the FFT wizard. You may choose from one to two single-output engines in parallel, or from one, two, or four quad-output engines in parallel.
- (2) Transform time is the time frame when the input block is loaded until the first output sample (corresponding to the input block) is output. Transform time does not include the time to unload the full output data block.
- (3) Block throughput is defined as the minimum number of cycles between two successive start-of-packet (*sink\_sop*) pulses.

## Stratix IV Devices

Table 1-15 shows the streaming data flow performance, using the 4 mults/2 adders complex multiplier structure, for data and twiddle width 16, for Stratix IV (EP4SGX70DF29C2X) devices.

**Table 1-15.** Performance with the Streaming Data Flow Engine Architecture—Stratix IV Devices

Points	Combinational ALUTs	Logic Registers	Memory (Bits)	Memory (M9K)	18 × 18 Mults	f <sub>MAX</sub> (MHz)	Clock Cycle Count	Transform Time (μs)
256	2,142	3,688	39,168	20	12	420	256	0.61
1,024	2,435	4,384	155,904	20	12	396	1,024	2.59
4,096	3,732	5,929	622,848	76	24	358	4,096	11.43

Table 1-16 shows the variable streaming data flow performance, with in order inputs and bit-reversed outputs, for width 16 (32 for floating point), for Stratix IV (EP4SGX70DF29C2X) devices.



The variable streaming with fixed-point number representation uses natural word growth, therefore the multiplier requirement is larger compared with the equivalent streaming FFT with the same number of points.

If you want to significantly reduce M9K memory utilization, set a lower  $f_{\text{MAX}}$  target.

**Table 1-16.** Performance with the Variable Streaming Data Flow Engine Architecture—Stratix IV Devices

Point	Points	Combinational ALUTs	Logic Registers	Memory			18 × 18 Mults	$f_{\text{MAX}}$ (MHz)	Clock Cycle Count	Transform Time (μs)
				Bits	ALUTs	M9K				
Fixed	256	2,572	3,996	10,193	29	10	24	304	256	0.84
Fixed	1,024	3,601	5,433	42,377	75	14	32	303	1,024	3.38
Fixed	4096	4,693	6,857	171,611	134	32	40	293	4,096	14
Floating	256	17,948	20,457	24,959	225	53	48	227	256	1.13
Floating	1,024	22,342	24,758	84,798	—	87	64	228	1,024	4.49
Floating	4,096	27,322	29,708	316,579	—	136	80	225	4,096	18.18

Table 1-17 lists resource usage with buffered burst data flow architecture, using the 4 mults/2 adders complex multiplier structure, for data and twiddle width 16, for Stratix IV (EP4SGX70DF29C2X) devices.

**Table 1-17.** Resource Usage with Buffered Burst Data Flow Architecture—Stratix IV Devices

Points	Number of Engines (1)	Combinational ALUTs	Logic Registers	Memory (Bits)	Memory (M9K)	18 × 18 Mults	$f_{\text{MAX}}$ (MHz)
256	1	1,995	3,546	30,976	16	12	412
1,024	1	2,039	3,738	123,136	16	12	430
4,096	1	2,082	3,917	491,776	60	12	388
256	2	3,344	5,508	30,976	31	24	383
1,024	2	3,380	5,710	123,136	31	24	369
4,096	2	3,426	5,897	491,776	60	24	372
256	4	5,868	9,852	30,976	60	48	361
1,024	4	5,931	10,071	123,136	60	48	367
4,096	4	6,007	10,271	491,776	60	48	363

**Notes to Table 1-11:**

- (1) When using the buffered burst architecture, you can specify the number of quad-output FFT engines in the FFT wizard.

Table 1-18 lists performance with buffered burst data flow architecture, using the 4 mults/2 adders complex multiplier structure, for data and twiddle width 16, for Stratix IV (EP4SGX70DF29C2X) devices.



**Table 1-18.** Performance with the Buffered Burst Data Flow Architecture—Stratix IV Devices

Points	Number of Engines (1)	fMAX (MHz)	Transform Calculation Time (2)		Data Load & Transform Calculation		Block Throughput (3)	
			Cycles	Time (μs)	Cycles	Time (μs)	Cycles	Time (μs)
256	1	412	235	0.57	491	1.19	331	0.8
1,024	1	430	1,069	2.49	2,093	4.87	1,291	3
4,096	1	388	5,167	13.32	9,263	23.88	6157	15.87
256	2	383	162	0.42	397	1.04	299	0.78
1,024	2	369	557	1.51	1,581	4.28	1,163	3.15
4,096	2	372	2,607	7	6,703	18	5,133	13.78
256	4	361	118	0.33	347	0.96	283	0.78
1,024	4	367	340	0.93	1,364	3.72	1,099	3
4,096	4	363	1,378	3.8	5,474	15.09	4,633	12.78

**Notes to Table 1-12:**

- (1) When using the buffered burst architecture, you can specify the number of quad-output engines in the FFT wizard. You may choose from one, two, or four quad-output engines in parallel.
- (2) In a buffered burst data flow architecture, transform time is defined as the time from when the N-sample input block is loaded until the first output sample is ready for output. Transform time does not include the additional N-1 clock cycle to unload the full output data block.
- (3) Block throughput is the minimum number of cycles between two successive start-of-packet (sink\_sop) pulses.

Table 1-19 lists resource usage with burst data flow architecture, using the 4 mults/2 adders complex multiplier structure, for data and twiddle width 16, for Stratix IV (EP4SGX70DF29C2X) devices.

**Table 1-19.** Resource Usage with the Burst Data Flow Architecture—Stratix IV Devices (Part 1 of 2)

Points	Engine Architecture	Number of Engines (2)	Combinational ALUTs	Logic Registers	Memory (Bits)	Memory (M9K)	18 × 18 Mults
256	Quad Output	1	1,849	3,480	14,592	8	12
1,024	Quad Output	1	1,891	3,663	57,600	8	12
4,096	Quad Output	1	1,926	3,830	229,632	28	12
256	Quad Output	2	3,065	5,440	14,592	15	24
1,024	Quad Output	2	3,106	5,632	57,600	15	24
4,096	Quad Output	2	3,154	5,808	229,632	28	24
256	Quad Output	4	5,353	9,788	14,592	28	48
1,024	Quad Output	4	5,408	9,998	57,600	28	48
4,096	Quad Output	4	5,482	10,187	229,632	28	48
256	Single Output	1	712	1,431	9,472	3	4
1,024	Single Output	1	749	1,477	37,120	6	4
4,096	Single Output	1	817	1,523	147,712	19	4
256	Single Output	1	1,024	2,278	14,592	9	8
1,024	Single Output	2	1,038	2,354	57,600	11	8

**Table 1–19.** Resource Usage with the Burst Data Flow Architecture—Stratix IV Devices (Part 2 of 2)

Points	Engine Architecture	Number of Engines (2)	Combinational ALUTs	Logic Registers	Memory (Bits)	Memory (M9K)	18 × 18 Mults
4,096	Single Output	1	1,075	2,430	229,632	28	8

**Notes to Table 1–13:**

- (1) Represents data and twiddle factor precision.
- (2) When using the burst data flow architecture, you can specify the number of engines in the FFT wizard. You may choose from one to two single-output engines in parallel, or from one, two, or four quad-output engines in parallel.

Table 1–20 lists performance with burst data flow architecture, using the 4 mults/2 adders complex multiplier structure, for data and twiddle width 16, for Stratix IV (EP4SGX70DF29C2X) devices.

**Table 1–20.** Performance with the Burst Data Flow Architecture—Stratix IV Devices

Points	Engine Architecture	Number of Engines (1)	fMAX (MHz)	Transform Calculation Time (2)		Data Load & Transform Calculation		Block Throughput (3)	
				Cycles	Time (μs)	Cycles	Time (μs)	Cycles	Time (μs)
256	Quad Output	1	418	235	0.56	491	1.17	331	0.79
1,024	Quad Output	1	433	1,069	2.47	2,093	4.84	1,291	2.98
4,096	Quad Output	1	391	5,167	13.22	9,263	23.69	6,157	15.75
256	Quad Output	2	398	162	0.41	397	1	299	0.75
1,024	Quad Output	2	404	557	1.38	1,581	3.91	1,163	2.88
4,096	Quad Output	2	379	2,607	6.88	6,703	17.7	5,133	13.56
256	Quad Output	4	363	118	0.33	374	1.03	283	0.78
1,024	Quad Output	4	367	340	0.93	1,364	3.72	1,099	3
4,096	Quad Output	4	346	1,378	3.99	5,474	15.84	4,633	13.41
256	Single Output	1	421	1,115	2.65	1,371	3.26	1,628	3.87
1,024	Single Output	1	414	5,230	12.63	6,344	15.32	7,279	17.58
4,096	Single Output	1	396	24,705	62.35	28,801	72.69	32,898	83.03
256	Single Output	2	402	585	1.46	841	2.09	1,098	2.73
1,024	Single Output	2	423	2,652	6.27	3,676	8.69	4,701	11.11
4,096	Single Output	2	405	12,329	30.47	16,495	40.77	20,605	50.93

**Notes to Table 1–14:**

- (1) In the burst I/O data flow architecture, you can specify the number of engines in the FFT wizard. You may choose from one to two single-output engines in parallel, or from one, two, or four quad-output engines in parallel.
- (2) Transform time is the time frame when the input block is loaded until the first output sample (corresponding to the input block) is output. Transform time does not include the time to unload the full output data block.
- (3) Block throughput is defined as the minimum number of cycles between two successive start-of-packet (*sink\_sop*) pulses.

### Design Flow

To evaluate the FFT MegaCore® function using the OpenCore Plus feature, include these steps in your design flow:

1. Obtain and install the FFT MegaCore function.

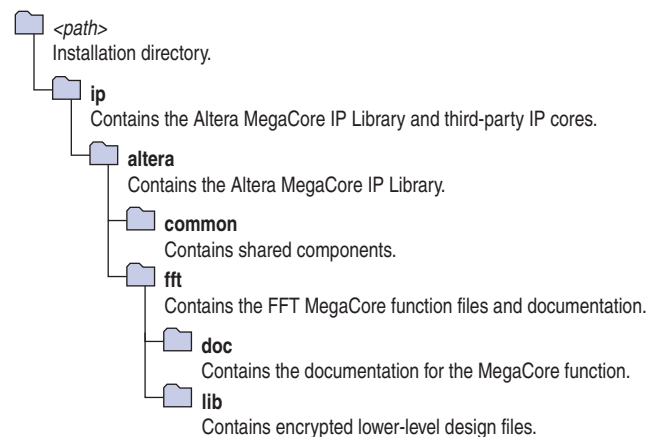
The FFT MegaCore function is part of the MegaCore IP Library, which is distributed with the Quartus® II software and downloadable from the Altera® website, [www.altera.com](http://www.altera.com).



For system requirements and installation instructions, refer to *Quartus II Installation & Licensing for Windows and Linux Workstations*.

Figure 2–1 shows the directory structure after you install the FFT MegaCore function, where *<path>* is the installation directory. The default installation directory on Windows is `c:\altera\90`; on Linux it is `/opt/altera90`.

**Figure 2–1.** Directory Structure



2. Create a custom variation of the FFT MegaCore function using IP Toolbench.



IP Toolbench is a toolbar from which you quickly and easily view documentation, specify parameters, and generate all of the files necessary for integrating the parameterized MegaCore function into your design.

3. Implement the rest of your design using the design entry method of your choice.
4. Use the IP functional simulation model, generated by IP Toolbench, to verify the operation of your system.



For more information on IP functional simulation models, refer to the *Simulating Altera IP in Third-Party Simulation Tools* chapter in volume 3 of the *Quartus II Handbook*.

5. Use the Quartus II software to compile your design.



You can also generate an OpenCore Plus time-limited programming file, which you can use to verify the operation of your design in hardware for a limited time.



For more information on OpenCore Plus hardware evaluation using the FFT MegaCore functions, see “OpenCore Plus Time-Out Behavior” on page 3–4, and *AN 320: OpenCore Plus Evaluation of Megafunctions*.

6. Purchase a license for the FFT MegaCore function.

After you have purchased a license for the FFT, follow these additional steps:

1. Set up licensing.
2. Generate a programming file for the Altera device(s) on your board.
3. Program the Altera device(s) with the completed design.

## FFT Walkthrough

This walkthrough explains how to create a custom variation of the FFT MegaCore function using IP Toolbench and the Quartus II software. As you go through the wizard, each step is described in detail. When you finish generating a custom variation of the FFT MegaCore function, you can incorporate it into your overall project.

This walkthrough requires the following steps:


- [Create a New Quartus II Project](#)
- [Launch IP Toolbench](#)
- [Step 1: Parameterize](#)
- [Step 2: Set Up Simulation](#)
- [Step 3: Generate](#)

### Create a New Quartus II Project


You need to create a new Quartus II project with the **New Project Wizard**, which specifies the working directory for the project, assigns the project name, and designates the name of the top-level design entity. To create a new project follow these steps:

1. Choose **Programs > Altera > Quartus II <version>** (Windows Start menu) to run the Quartus II software. Alternatively, you can use the Quartus II Web Edition software.
2. Choose **New Project Wizard** (File menu).

3. Click **Next** in the **New Project Wizard Introduction** page (the introduction page does not display if you turned it off previously).
4. In the **New Project Wizard: Directory, Name, Top-Level Entity** page, enter the following information:
  - a. Specify the working directory for your project. For example, this walkthrough uses the `c:\altera\projects\fft_project` directory.

 The Quartus II software automatically specifies a top-level design entity that has the same name as the project. This walkthrough assumes that the names are the same.

- b. Specify the name of the project. This walkthrough uses **example** for the project name.
5. Click **Next** to close this page and display the **New Project Wizard: Add Files** page.

 When you specify a directory that does not already exist, a message asks if the specified directory should be created. Click **Yes** to create the directory.


6. Click **Next** to close this page and display the **New Project Wizard: Family & Device Settings** page.
7. On the **New Project Wizard: Family & Device Settings** page, choose the target device family in the Family list.
8. The remaining pages in the **New Project Wizard** are optional. Click **Finish** to complete the Quartus II project.

You have finished creating your new Quartus II project.

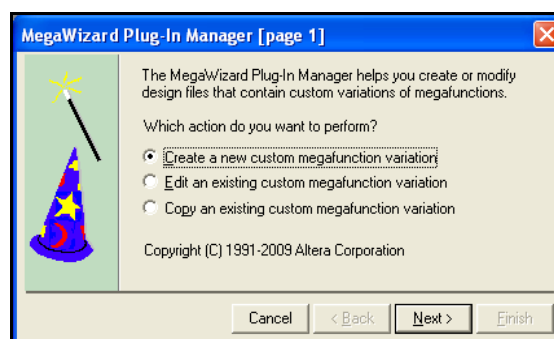
## Launch IP Toolbench

To launch IP Toolbench in the Quartus II software, follow these steps:

1. Start the MegaWizard® Plug-In Manager by choosing **MegaWizard Plug-In Manager** (Tools menu). The **MegaWizard Plug-In Manager** dialog box displays (see [Figure 2-2](#)).

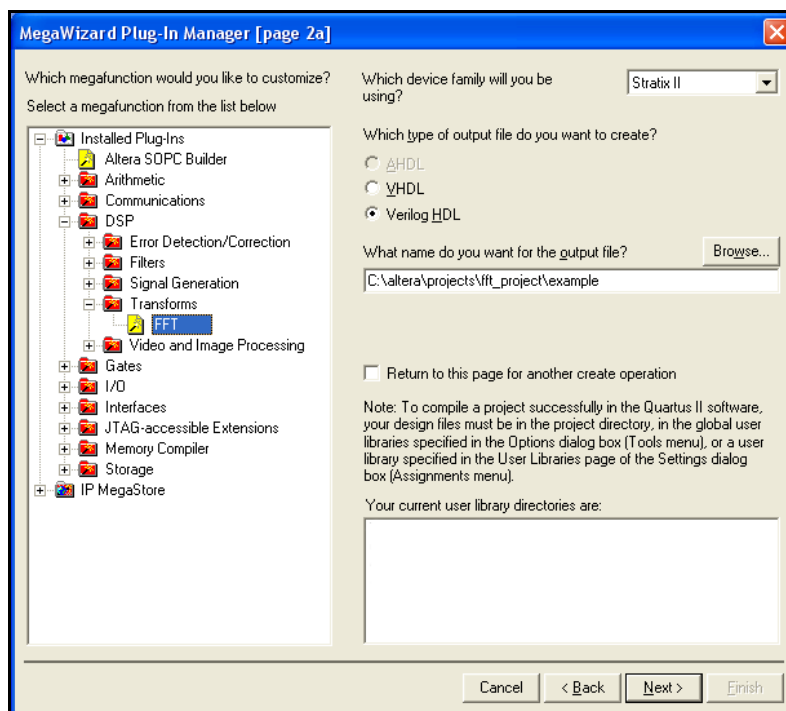
 Refer to Quartus II Help for more information on how to use the MegaWizard Plug-In Manager.

**Figure 2-2.** MegaWizard Plug-In Manager



2. Specify that you want to create a new custom megafunction variation and click **Next**.
3. Expand the **DSP > Transforms** directory and click **FFT**.
4. Choose the output file type for your design; the wizard supports, VHDL and Verilog HDL.
5. The MegaWizard Plug-In Manager shows the project path that you specified in the **New Project Wizard**. Append a variation name for the MegaCore function output files `<project path>\<variation name>`. [Figure 2-3](#) shows the wizard after you have made these settings.

**Figure 2-3.** Select the MegaCore Function



6. Click **Next** to launch IP Toolbench.

## Step 1: Parameterize

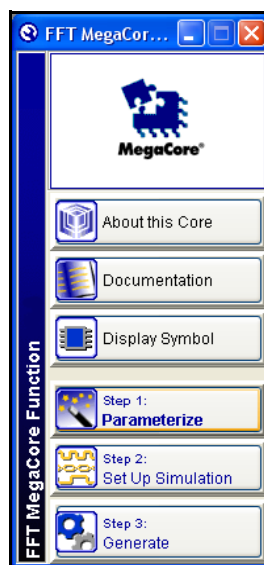
To create a custom variation of the FFT MegaCore function, follow these steps:



For more information on the parameters, see [“Parameters” on page 3-14](#).

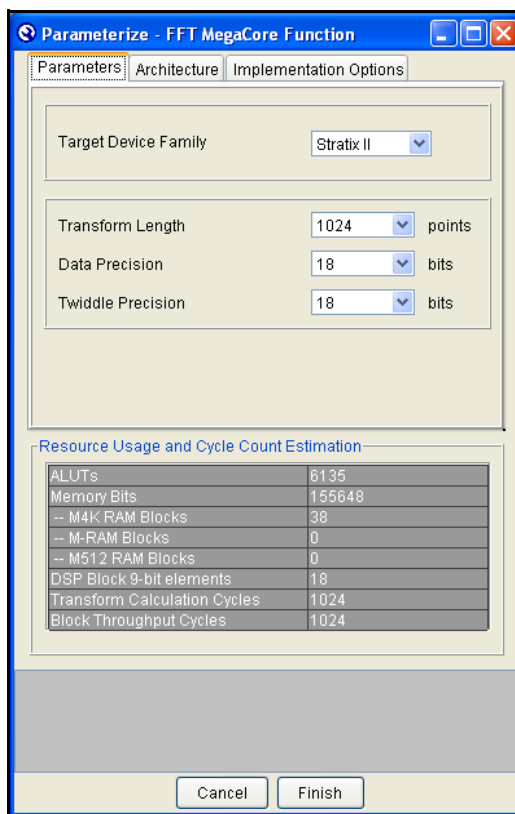
1. Click the **Step 1: Parameterize** button in IP Toolbench (see [Figure 2-4](#)).

**Figure 2-4.** IP Toolbench—Parameterize






2. Do not change the **Target Device Family**. The device family is automatically set to the value that was specified in your Quartus II project and the generated HDL for your MegaCore function variation may be incorrect if this value is changed (see [Figure 2-5](#)).

**Figure 2-5.** Parameters Tab

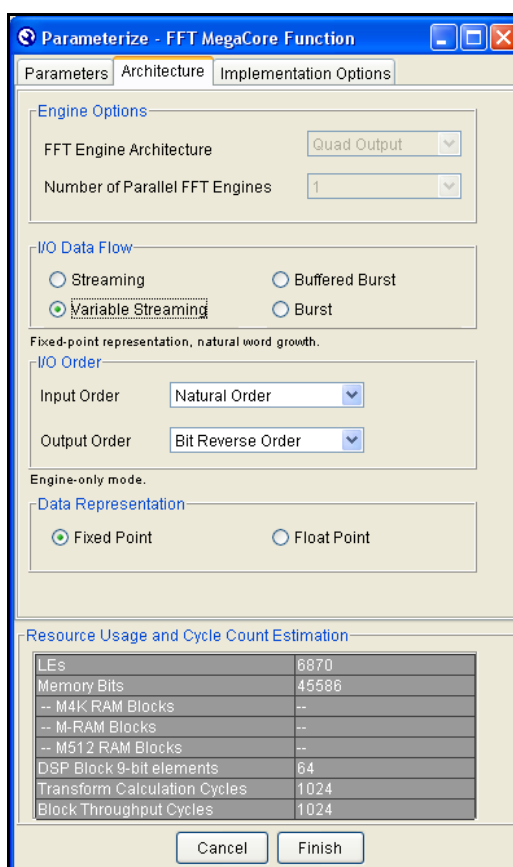


- Choose the **Transform length**, **Data precision**, and **Twiddle precision**.

-  Twiddle factor precision must be less than or equal to data precision.
-  If the variable streaming option is chosen (see **Architecture** Tab), the **Transform length** represents the maximum transform length that can be performed. All transforms of length  $2^m$  where  $6 \leq m \leq \log_2(\text{transform length})$  can be performed at runtime.
-  On the **Architecture** tab, if you select **Variable Streaming** and **floating point**, the precision is set to 32.

- Click the Architecture tab (see [Figure 2-6](#)).


**Figure 2-6.** Architecture Tab



- Choose the FFT Engine Architecture, Number of Parallel FFT Engines, and select the I/O Data Flow.

If you select the **Streaming** I/O data flow, the FFT MegaCore function automatically generates a design with a **Quad Output** FFT engine architecture and the minimum number parallel FFT engines for the required throughput.



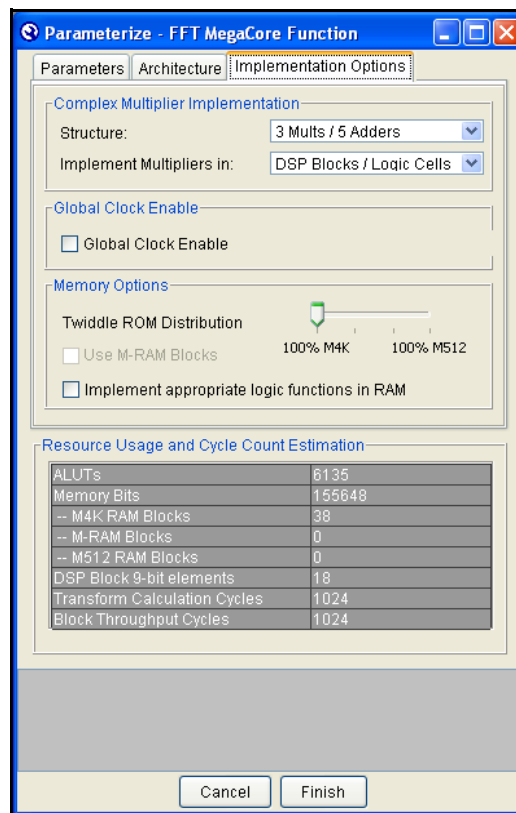
 A single FFT engine architecture provides enough performance for up to a 1,024-point streaming I/O data flow FFT.

If you select **Variable Streaming** I/O data flow, the **I/O Order** and **Data Representation** options are visible. The **Input Order** options allow you to select the order in which the samples are presented to the FFT. If you select **Natural Order**, the FFT expects the order of the input samples to be sequential (1, 2 ...,  $n - 1$ ,  $n$ ) where  $n$  is the size of the current transform. For **Bit Reverse Order**, the FFT expects the input samples to be in bit-reversed order. For  $-N/2$  to  $N/2$ , the FFT expects the input samples to be in the order  $-N/2$  to  $(N/2) - 1$ , otherwise known as the DC centered order. Similarly the **Output Order** option specifies the order in which the FFT generates the output.

In addition, you can select **fixed point** or **floating point** data representation.

6. Click the Implementation **Options** tab (see [Figure 2-7](#)).

**Figure 2-7.** Implementation Options Tab



7. Choose the complex multiplier structure.
8. Choose how you want to implement the multipliers.
9. Turn on **Global Clock Enable**, if you want to add a global clock enable to your design.
10. Specify the memory options.
11. Click **Finish** when the implementation options are set.



The implementation options are not available for the variable streaming architecture.

## Step 2: Set Up Simulation

An IP functional simulation model is a cycle-accurate VHDL or Verilog HDL model produced by the Quartus II software. The model allows for fast functional simulation of IP using industry-standard VHDL and Verilog HDL simulators.

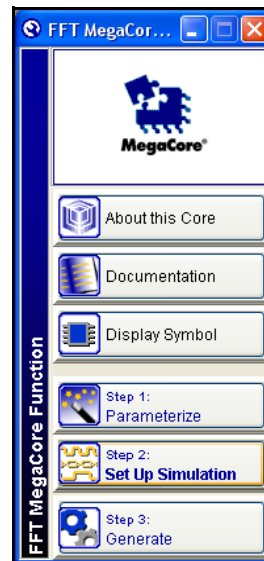


You may only use these simulation model output files for simulation purposes and expressly not for synthesis or any other purposes. Using these models for synthesis creates a nonfunctional design.

To generate an IP functional simulation model for your MegaCore function, follow these steps:

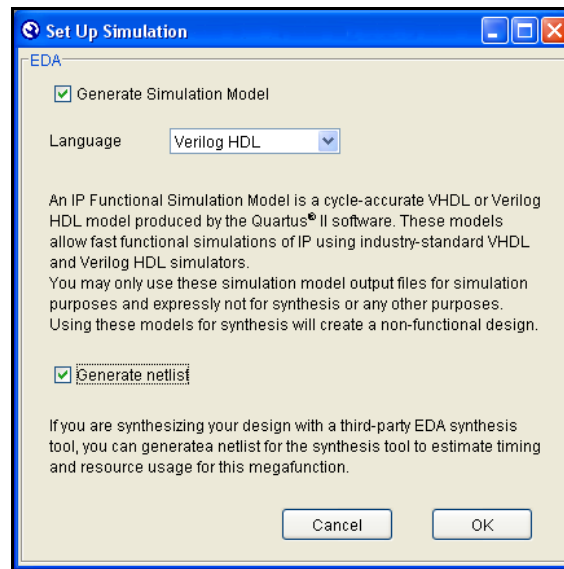
1. Click **Step 2: Set Up Simulation** in IP Toolbench (see [Figure 2-8](#)).

**Figure 2-8.** IP Toolbench—Set Up Simulation



2. Turn on **Generate Simulation Model** (see [Figure 2-9](#)).

**Figure 2-9.** Generate Simulation Model



3. Choose the language in the **Language** list.
4. Some third-party synthesis tools can use a netlist that contains only the structure of the MegaCore function, but not detailed logic, to optimize performance of the design that contains the MegaCore function. If your synthesis tool supports this feature, turn on **Generate netlist**.
5. Click **OK**.

### Step 3: Generate

To generate your MegaCore function, follow these steps:

1. Click **Step 3: Generate** in IP Toolbench (see [Figure 2-10](#)).

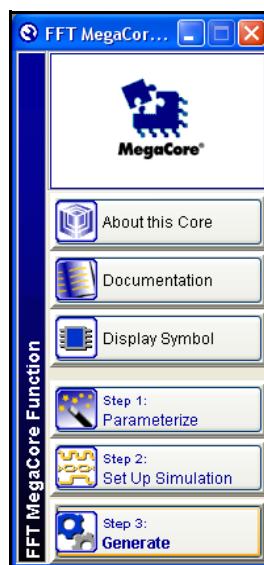
**Figure 2-10.** IP Toolbench—Generate

Figure 2-11 shows the generation report.

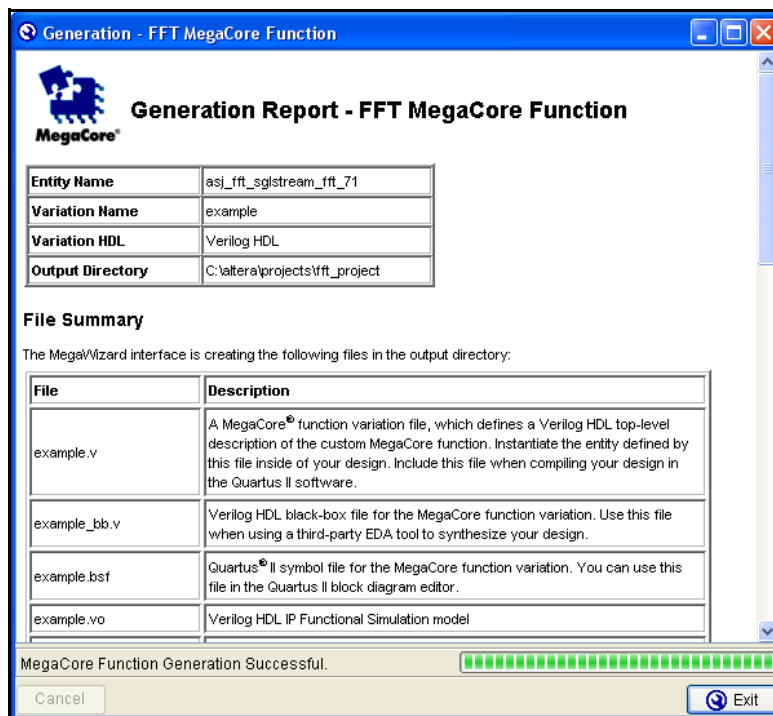
**Figure 2-11.** Generation Report

Table 2-1 describes the generated files and other files that may be in your project directory. The names and types of files specified in the IP Toolbench report vary based on whether you created your design with VHDL or Verilog HDL

**Table 2-1.** Generated Files (Part 1 of 2) *(Note 1) & (2)*

Filename	Description
<b>imag_input.txt</b>	The text file contains input imaginary component random data. This file is read by the generated VHDL or Verilog HDL MATLAB testbenches.
<b>real_input.txt</b>	Test file containing real component random data. This file is read by the generated VHDL or Verilog HDL and MATLAB testbenches.
<b>&lt;variation name&gt;.bsf</b>	Quartus II symbol file for the MegaCore function variation. You can use this file in the Quartus II block diagram editor.
<b>&lt;variation name&gt;.vo or .vho</b>	VHDL or Verilog HDL IP functional simulation model.
<b>&lt;variation name&gt;.vhd, or .v</b>	A MegaCore function variation file, which defines a VHDL or Verilog HDL top-level description of the custom MegaCore function. Instantiate the entity defined by this file inside of your design. Include this file when compiling your design in the Quartus II software.
<b>&lt;variation name&gt;_bit_reverse_top.vhd</b>	Example top-level VHDL design with bit-reversal module (variable streaming FFT engine-only mode only). This file shows how the bit-reversal operation can be external to the MegaCore architecture. For example, when there is an opportunity to combine the bit-reversal operation with another user-specified operation.
<b>&lt;variation name&gt;_1n1024cos.hex, &lt;variation name&gt;_2n1024cos.hex, &lt;variation name&gt;_3n1024cos.hex</b>	Intel hex-format ROM initialization files (not generated for variable streaming FFT).
<b>&lt;variation name&gt;_1n1024sin.hex, &lt;variation name&gt;_2n1024sin.hex, &lt;variation name&gt;_3n1024sin.hex</b>	Intel hex-format ROM initialization files (not generated for variable streaming FFT).
<b>&lt;variation name&gt;_fft.fsi</b>	A DSP Builder fast functional simulation model parameter description file (variable streaming only).
<b>&lt;variation name&gt;_model.m</b>	MATLAB m-file describing a MATLAB bit-accurate model.
<b>&lt;variation name&gt;_tb.m</b>	MATLAB testbench.
<b>&lt;variation name&gt;_syn.v or &lt;variation name&gt;_syn.vhd</b>	A timing and resource netlist for use in some third-party synthesis tools.
<b>&lt;variation name&gt;_tb.v or &lt;variation name&gt;_tb.vhd</b>	Verilog HDL or VHDL testbench file.
<b>&lt;variation name&gt;_nativelink.tcl</b>	Tcl Script that sets up NativeLink in the Quartus II software to natively simulate the design using selected EDA tools see <a href="#">“Simulating in Third-Party Simulation Tools Using NativeLink” on page 2-14.</a>

**Table 2-1.** Generated Files (Part 2 of 2) (Note 1) & (2)

Filename	Description
twr1_opt.hex, twi1_opt.hex, twr2_opt.hex, twi2_opt.hex, twr3_opt.hex, twi3_opt.hex, twr4_opt.hex, twi4_opt.hex,	Intel hex-format ROM initialization files (variable streaming FFT only).
<b>Notes to Table 2-1:</b> (1) These files are variation dependent, some may be absent or their names may change. (2) <variation name> is a prefix variation name supplied automatically by IP Toolbench.	

- After you review the generation report, click **Exit** to close IP Toolbench and click **Yes** on the **Quartus II IP Files** message.



The Quartus II IP File (.qip) is a file generated by the MegaWizard interface that contains information about a generated IP core. You are prompted to add this .qip file to the current Quartus II project at the time of file generation. In most cases, the .qip file contains all of the necessary assignments and information required to process the core or system in the Quartus II compiler. Generally, a single .qip file is generated for each MegaCore function.

You can now integrate your custom MegaCore function variation into your design and simulate and compile.

## Simulate the Design

This section describes the following simulation techniques:

- [Simulate in the MATLAB Software](#)
- [Simulate with IP Functional Simulation Models](#)
- [Simulating in Third-Party Simulation Tools Using NativeLink](#)

### Simulate in the MATLAB Software

This section discusses fixed-transform and variable streaming architecture simulations.

#### Fixed Transform Architectures

The FFT MegaCore function outputs a bit-accurate MATLAB model <variation name>\_model.m, which you can use to model the behavior of your custom FFT variation in the MATLAB software. The model takes a complex vector as input and it outputs the transform-domain complex vector and corresponding block exponent values. The length and direction of the transform (FFT/IFFT) are also passed as inputs to the model.

If the input vector length is an integral multiple of  $N$ , the transform length, the length of the output vector(s) is equal to the length of the input vector. However, if the input vector is not an integral multiple of  $N$ , it is zero-padded to extend the length to be so.



For additional information on exponent values, refer to the application note, [AN 404: FFT/IFFT Block Floating Point Scaling](#).

The wizard also creates the MATLAB testbench file `<variation name>_tb.m`. This file creates the stimuli for the MATLAB model by reading the input complex random data from IP Toolbench-generated.

If you selected **Floating point** data representation, the input data is generated in hexadecimal format.

To model your fixed-transform architecture FFT MegaCore function variation in the MATLAB software, follow these steps:

1. Run the MATLAB software.
2. In the MATLAB command window, change to the working directory for your project.
3. Perform the simulation:
  - a. Type `help <variation name>_model` at the command prompt to view the input and output vectors that are required to run the MATLAB model as a standalone M-function. Create your input vector and make a function call to `<variation name>_model`. For example:

```
N=2048;
INVERSE = 0; % 0 => FFT 1=> IFFT
x = (2^12)*rand(1,N) + j*(2^12)*rand(1,N);
[y,e] = <variation name>_model(x,N,INVERSE);
```

or

- b. Run the provided testbench by typing the name of the testbench, `<variation name>_tb` at the command prompt.



For more information on MATLAB and Simulink, refer to the MathWorks web site at [www.mathworks.com](http://www.mathworks.com).

## Variable Streaming Architecture

The FFT MegaCore function outputs a bit-accurate MATLAB model `<variation name>_model.m`, which you can use to model the behavior of your custom FFT variation in the MATLAB software. The model takes a complex vector as input and it outputs the transform-domain complex vector. The lengths and direction of the transforms (FFT/IFFT) (specified as one entry per block) are also passed as an input to the model.

You must ensure that the length of the input vector is at least as large as the sum of the transform sizes for the model to function correctly.

The wizard also creates the MATLAB testbench file `<variation name>_tb.m`. This file creates the stimuli for the MATLAB model by reading the input complex random data from files generated by IP Toolbench.

To model your variable streaming architecture FFT MegaCore function variation in the MATLAB software, follow these steps:

1. Run the MATLAB software.
2. In the MATLAB command window, change to the working directory for your project.
3. Perform the simulation:
  - a. Type `help <variation name>_model` at the command prompt to view the input and output vectors that are required to run the MATLAB model as a standalone M-function. Create your input vector and make a function call to `<variation name>_model`. For example:

```
nps=[256,2048];
inverse = [0,1]; % 0 => FFT 1=> IFFT
x = (2^12)*rand(1,sum(nps)) + j*(2^12)*rand(1,sum(nps));
[y] = <variation name>_model(x,nps,inverse);
```

or

- b. Run the provided testbench by typing the name of the testbench, `<variation name>_tb` at the command prompt.



If you selected bit-reversed output order, you can reorder the data with the following MATLAB code:

```
y = y(bit_reverse(0:(FFTSIZE-1), log2(FFTSIZE)) + 1);
```

where `bit_reverse` is:

```
function y = bit_reverse(x, n_bits)
y = bin2dec(fliplr(dec2bin(x, n_bits)));
```

## Simulate with IP Functional Simulation Models

To simulate your design, use the IP functional simulation models generated by IP Toolbench. The IP functional simulation model is the `.vo` or `.vho` file generated as specified in “[Step 2: Set Up Simulation](#)” on page 2-8. Compile the `.vo` or `.vho` file in your simulation environment to perform functional simulation of your custom variation of the MegaCore function.



For more information on IP functional simulation models, refer to the *Simulating Altera in Third-Party Simulation Tools* chapter in volume 3 of the *Quartus II Handbook*.

## Simulating in Third-Party Simulation Tools Using NativeLink

You can perform a simulation in a third-party simulation tool from within the Quartus II software, using NativeLink.



For more information on NativeLink, refer to the *Simulating Altera IP in Third-Party Simulation Tools* chapter in volume 3 of the *Quartus II Handbook*.

You can use the Tcl script file `<variation name>_nativelink.tcl` to assign default NativeLink testbench settings to the Quartus II project.



To set up simulation in the Quartus II software using NativeLink, follow these steps:

1. Create a custom variation but ensure you specify your variation name to match the Quartus II project name.
2. Check that the absolute path to your third-party simulator executable is set. On the Tools menu click **Options** and select **EDA Tools Options**.
3. On the Processing menu, point to **Start** and click **Start Analysis & Elaboration**.
4. On the Tools menu click **Tcl scripts**. Select the *<variation name>\_nativelink.tcl* Tcl script and click **Run**. Check for a message confirming that the Tcl script was successfully loaded.
5. On the Assignments menu click **Settings**, expand **EDA Tool Settings** and select **Simulation**. Select a simulator under **Tool Name** and in **NativeLink Settings**, select **Test Benches**.
6. On the Tools menu point to **EDA Simulation Tool** and click **Run EDA RTL Simulation**.

## Compile the Design

Use the Quartus II software to synthesize and place and route your design. Refer to Quartus II Help for instructions on performing compilation.

### Fixed Transform Architecture

To compile your fixed-transform architecture design, follow these steps:

1. If you are using the Quartus II software to synthesize your design, skip to step 2-15. If you are using a third-party synthesis tool to synthesize your design, follow these steps:
  - a. Set a black box attribute for your FFT MegaCore function custom variation before you synthesize the design. Refer to Quartus II Help for instructions on setting black-box attributes per synthesis tool.
  - b. Run the synthesis tool to produce an EDIF Netlist File (.edf) or Verilog Quartus Mapping (VQM) file (.vqm) for input to the Quartus II software.
  - c. Add the EDIF or VQM file to your Quartus II project.
2. Choose **Add/Remove Files in Project** (Project menu).
3. If the **fft\_pack\_fft\_72.vhd** file is not listed, browse to the \lib directory and choose the VHDL package, **fft\_pack\_fft\_72.vhd**.
4. Click **Open** to add the **fft\_pack\_fft\_72.vhd** file to your Quartus II project.



Ensure the **fft\_pack\_fft\_72.vhd** file is at the top of the list in the **File Name list** window. If the **fft\_pack\_fft\_72.vhd** file is not at the top of the **File Name list**, choose **fft\_pack\_fft\_72.vhd** and click **Up**.

5. Choose **Start Compilation** (Processing menu).

## Variable Streaming Architecture

To compile your variable streaming architecture design, follow these steps:

1. If you are using the Quartus II software to synthesize your design, skip to step 2–15. If you are using a third-party synthesis tool to synthesize your design, follow these steps:
  - a. Set a black-box attribute for your FFT MegaCore function custom variation before you synthesize the design. Refer to Quartus II Help for instructions on setting black-box attributes per synthesis tool.
  - b. Run the synthesis tool to produce an EDIF Netlist File (.edf) or Verilog Quartus Mapping (VQM) file (.vqm) for input to the Quartus II software.
  - c. Add the EDIF or VQM file to your Quartus II project.
2. Choose **Add/Remove Files in Project** (Project menu).
3. You should see a list of files in the project. If there are no files listed browse to the \lib, then select and add all files with the prefix **auk\_dspip\_r22sdf** and **auk\_dspip\_bit\_reverse**. Browse to the <project> directory and select all files with prefix **auk\_dspip**.
4. Chose **Start Compilation** (Processing menu).

## Program a Device

After you have compiled your design, program your targeted Altera device, and verify your design in hardware.

With Altera's free OpenCore Plus evaluation feature, you can evaluate the FFT MegaCore function before you purchase a license. OpenCore Plus evaluation allows you to generate an IP functional simulation model, and produce a time-limited programming file.



For more information on IP functional simulation models, refer to the *Simulating Altera in Third-Party Simulation Tools* chapter in volume 3 of the *Quartus II Handbook*.

You can simulate the FFT in your design, and perform a time-limited evaluation of your design in hardware.



For more information on OpenCore Plus hardware evaluation using the FFT, see “[OpenCore Plus Time-Out Behavior](#)” on page 3–4 and AN 320: *OpenCore Plus Evaluation of Megafunctions*.

## Set Up Licensing

You need to purchase a license for the MegaCore function only when you are completely satisfied with its functionality and performance and want to take your design to production.

After you purchase a license for FFT, you can request a license file from the Altera website at [www.altera.com/licensing](http://www.altera.com/licensing) and install it on your computer. When you request a license file, Altera emails you a **license.dat** file. If you do not have Internet access, contact your local Altera representative.

The discrete Fourier transform (DFT), of length  $N$ , calculates the sampled Fourier transform of a discrete-time sequence at  $N$  evenly distributed points  $\omega k = 2\pi k/N$  on the unit circle.

Equation 1 shows the length- $N$  forward DFT of a sequence  $x(n)$ :

$$X[k] = \sum_{n=0}^{N-1} x(n)e^{-j2\pi nk/N}$$

where  $k = 0, 1, \dots, N-1$

Equation 2 shows the length- $N$  inverse DFT:

$$x(n) = (1/N) \sum_{k=0}^{N-1} X[k]e^{j2\pi nk/N}$$

where  $n = 0, 1, \dots, N-1$

The complexity of the DFT direct computation can be significantly reduced by using fast algorithms that use a nested decomposition of the summation in equations one and two—in addition to exploiting various symmetries inherent in the complex multiplications. One such algorithm is the Cooley-Tukey radix- $r$  decimation-in-frequency (DIF) FFT, which recursively divides the input sequence into  $N/r$  sequences of length  $r$  and requires  $\log_r N$  stages of computation.

Each stage of the decomposition typically shares the same hardware, with the data being read from memory, passed through the FFT processor and written back to memory. Each pass through the FFT processor is required to be performed  $\log_r N$  times. Popular choices of the radix are  $r = 2, 4$ , and  $16$ . Increasing the radix of the decomposition leads to a reduction in the number of passes required through the FFT processor at the expense of device resources.



The MegaCore function does not apply the scaling factor  $1/N$  required for a length- $N$  inverse DFT. You must apply this factor externally.

## Buffered, Burst, & Streaming Architectures

A radix-4 decomposition, which divides the input sequence recursively to form four-point sequences, has the advantage that it requires only trivial multiplications in the four-point DFT and is the chosen radix in the Altera® FFT MegaCore® function. This results in the highest throughput decomposition, while requiring non-trivial complex multiplications in the post-butterfly twiddle-factor rotations only. In cases where  $N$  is an odd power of two, the FFT MegaCore automatically implements a radix-2 pass on the last pass to complete the transform.

To maintain a high signal-to-noise ratio throughout the transform computation, the FFT MegaCore function uses a block-floating-point architecture, which is a trade-off point between fixed-point and full-floating point architectures.

In a fixed-point architecture, the data precision needs to be large enough to adequately represent all intermediate values throughout the transform computation. For large FFT transform sizes, an FFT fixed-point implementation that allows for word growth can make either the data width excessive or can lead to a loss of precision.

In a floating-point architecture each number is represented as a mantissa with an individual exponent—while this leads to greatly improved precision, floating-point operations tend to demand increased device resources.

In a block-floating point architecture, all of the values have an independent mantissa but share a common exponent in each data block. Data is input to the FFT function as fixed point complex numbers (even though the exponent is effectively 0, you do not enter an exponent).

The block-floating point architecture ensures full use of the data width within the FFT function and throughout the transform. After every pass through a radix-4 FFT, the data width may grow up to  $\log_2(4\sqrt{2}) = 2.5$  bits. The data is scaled according to a measure of the block dynamic range on the output of the previous pass. The number of shifts is accumulated and then output as an exponent for the entire block. This shifting ensures that the minimum of least significant bits (LSBs) are discarded prior to the rounding of the post-multiplication output. In effect, the block-floating point representation acts as a digital automatic gain control. To yield uniform scaling across successive output blocks, you must scale the FFT function output by the final exponent.



In comparing the block-floating point output of the Altera FFT MegaCore function to the output of a full precision FFT from a tool like MATLAB, the output should be scaled by  $2^{(-\text{exponent\_out})}$  to account for the discarded LSBs during the transform (see “Block Floating Point Scaling” on page A-1).



For more information on exponent values, refer to the application note, [AN 404: FFT/IFFT Block Floating Point Scaling](#).

## Variable Streaming Architecture

The variable streaming architecture uses a radix  $2^2$  single delay feedback architecture, which is a fully pipelined architecture. For a length  $N$  transform there are  $\log_4(N)$  stages concatenated together. The radix  $2^2$  algorithm has the same multiplicative complexity of a fully pipelined radix-4 architecture, however the butterfly unit retains a radix-2 architecture. The butterfly units use the DIF decomposition.

The variable streaming architecture uses either fixed point or single precision floating point data representation. Fixed point representation allows for natural word growth through the pipeline. The maximum growth of each stage is  $\log_2(4\sqrt{2}) = 2.5$  bits, which is accommodated in the design by growing the pipeline stages by either 2 bits or 3 bits. After the complex multiplication the data is rounded down to the expanded data size using convergent rounding.

The floating point internal data representation is single precision floating point (32 bit, IEEE 754 representation). Floating point operations are costly in terms of hardware resources. To reduce the amount of logic required for floating point operations, the variable streaming FFT uses "fused" floating point kernels. The reduction in logic occurs by fusing together several floating point operations and reducing the number of normalizations that need to occur.

You can select input and output orders generated by the FFT. Table 3–1 shows the input and output order options.

**Table 3–1.** Input & Output Order Options

Input Order	Output Order	Mode	Comments
Natural	Bit reversed	Engine-only	Requires minimum memory and minimum latency.
Bit reversed	Natural		
DC-centred	Bit-reversed		
Natural	Natural	Engine with bit-reversal	At the output, requires an extra $N$ complex memory words and an additional $N$ clock cycles latency, where $N$ is the size of the transform.
Bit reversed	Bit reversed		
DC-centred	Natural		

Some applications for the FFT require an FFT > user operation > IFFT chain. In this case, choosing the input order and output order carefully can lead to significant memory and latency savings. For example, consider where the input to the first FFT is in natural order and the output is in bit-reversed order (FFT is operating in engine-only mode). In this example, if the IFFT operation is configured to accept bit-reversed inputs and produces natural order outputs (IFFT is operating in engine-only mode), only the minimum amount of memory is required, which provides a saving of  $N$  complex memory words, and a latency saving of  $N$  clock cycles, where  $N$  is the size of the current transform.

## The Avalon Streaming Interface

The Avalon® Streaming (Avalon-ST) interface is an evolution of the Atlantic™ interface. The Avalon-ST interface defines a standard, flexible, and modular protocol for data transfers from a source interface to a sink interface and simplifies the process of controlling the flow of data in a datapath. The Avalon-ST interface signals can describe traditional streaming interfaces supporting a single stream of data without knowledge of channels or packet boundaries. Such interfaces typically contain data, ready, and valid signals. The Avalon-ST interface can also support more complex protocols for burst and packet transfers with packets interleaved across multiple channels. The Avalon-ST interface inherently synchronizes multi-channel designs, which allows you to achieve efficient, time-multiplexed implementations without having to implement complex control logic.

The Avalon-ST interface supports backpressure, which is a flow control mechanism, where a sink can signal to a source to stop sending data. The sink typically uses backpressure to stop the flow of data when its FIFO buffers are full or when there is congestion on its output. When designing a datapath, which includes the FFT MegaCore function, you may not need backpressure if you know the downstream components can always receive data. You may achieve a higher clock rate by driving the source ready signal `source_ready` of the FFT high, and not connecting the sink ready signal `sink_ready`.

The FFT MegaCore function has a `READY_LATENCY` value of zero.



For more information on the Avalon-ST interface, refer to the [Avalon Interface Specifications](#).

## OpenCore Plus Time-Out Behavior

OpenCore Plus hardware evaluation can support the following two modes of operation:

- Untethered—the design runs for a limited time
- Tethered—requires a connection between your board and the host computer. If tethered mode is supported by all megafunctions in a design, the device can operate for a longer time or indefinitely

All megafunctions in a device time out simultaneously when the most restrictive evaluation time is reached. If there is more than one megafunction in a design, a specific megafunction's time-out behavior may be masked by the time-out behavior of the other megafunctions.



For MegaCore functions, the untethered time-out is one hour; the tethered time-out value is indefinite.

The signals `source_real`, `source_imag`, and `source_exp` are forced low when the evaluation time expires.



For more information on OpenCore Plus hardware evaluation, see [“OpenCore Plus Evaluation” on page 1-4](#) and *AN 320: OpenCore Plus Evaluation of Megafunctions*.

## FFT Processor Engine Architectures

The FFT MegaCore function can be parameterized to use either quad-output or single-output engine architecture. To increase the overall throughput of the FFT MegaCore function, you may also use multiple parallel engines of a variation. This section discusses the following topics:

- Radix  $2^2$  single-delay feedback architecture
- Quad-output FFT engine architecture
- Single-output FFT engine architecture

## Radix-2<sup>2</sup> Single Delay Feedback Architecture

Radix-2<sup>2</sup> single delay feedback architecture is a fully pipelined architecture for calculating the FFT of incoming data. It is similar to radix-2 single delay feedback architectures. However, the twiddle factors are rearranged such that the multiplicative complexity is equivalent to a radix-4 single delay feedback architecture.

There are  $\log_2(N)$  stages with each stage containing a single butterfly unit and a feedback delay unit that delays the incoming data by a specified number of cycles, halved at every stage. These delays effectively align the correct samples at the input of the butterfly unit for the butterfly calculations. Every second stage contains a modified radix-2 butterfly whereby a trivial multiplication by  $-j$  is performed before the radix-2 butterfly operations. The output of the pipeline is in bit-reversed order.

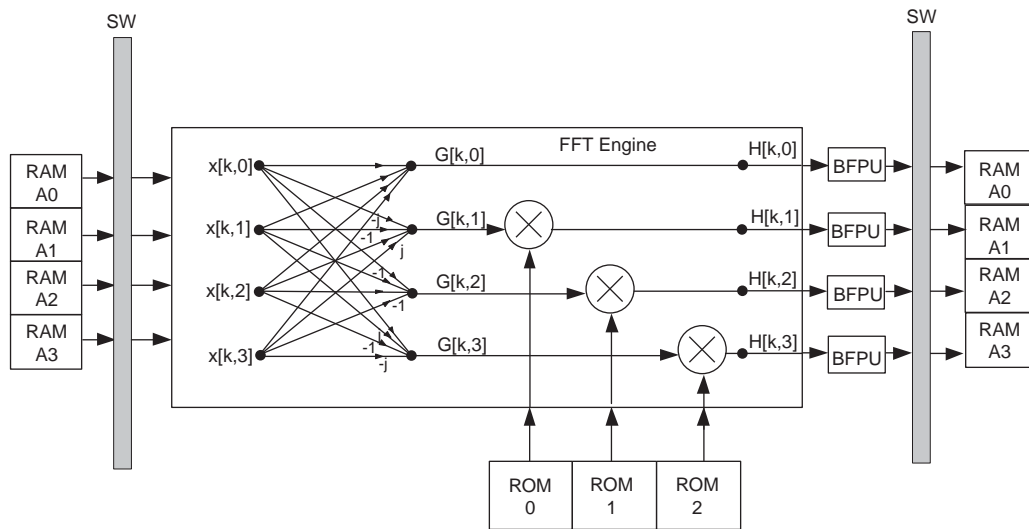
The following scheduled operations in the pipeline for an FFT of length  $N = 16$  occur.

1. For the first 8 clock cycles, the samples are fed unmodified through the butterfly unit to the delay feedback unit.
2. The next 8 clock cycles perform the butterfly calculation using the data from the delay feedback unit and the incoming data. The higher order calculations are sent through to the delay feedback unit while the lower order calculations are sent to the next stage.
3. The next 8 clock cycles feeds the higher order calculations stored in the delay feedback unit unmodified through the butterfly unit to the next stage.

Subsequent data stages use the same principles. However, the delays in the feedback path are adjusted accordingly.

## Quad-Output FFT Engine Architecture

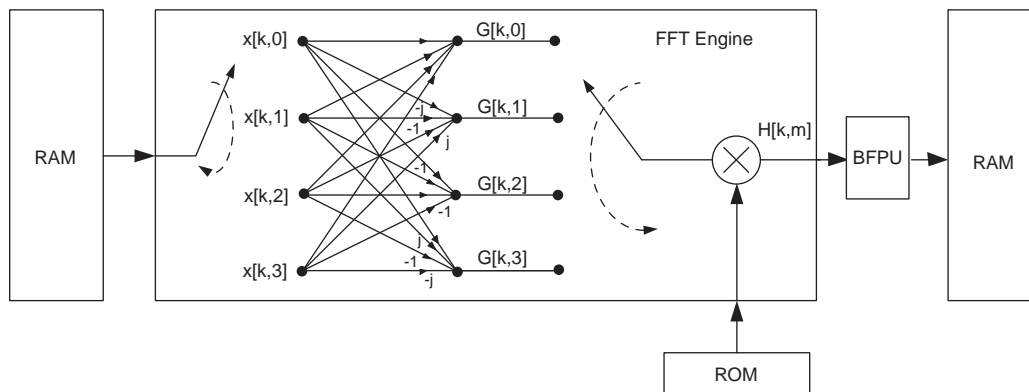
For applications where transform time is to be minimized, a quad-output FFT engine architecture is optimal. The term quad-output refers to the throughput of the internal FFT butterfly processor. The engine implementation computes all four radix-4 butterfly complex outputs in a single clock cycle. [Figure 3-1](#) shows a diagram of the quad-output FFT engine.

**Figure 3-1.** Quad-Output FFT Engine

Complex data samples  $x[k,m]$  are read from internal memory in parallel and re-ordered by switch (SW). Next, the ordered samples are processed by the radix-4 butterfly processor to form the complex outputs  $G[k,m]$ . Because of the inherent mathematics of the radix-4 DIF decomposition, only three complex multipliers are required to perform the three non-trivial twiddle-factor multiplications on the outputs of the butterfly processor. To discern the maximum dynamic range of the samples, the four outputs are evaluated in parallel by the block-floating point units (BFPU). The appropriate LSBs are discarded and the complex values are rounded and re-ordered before being written back to internal memory.

## Single-Output FFT Engine Architecture

For applications where the minimum-size FFT function is desired, a single-output engine is most suitable. The term single-output again refers to the throughput of the internal FFT butterfly processor. In the engine architecture, a single butterfly output is computed per clock cycle, requiring a single complex multiplier (see [Figure 3-2](#)).

**Figure 3-2.** Single-Output FFT Engine Architecture



## I/O Data Flow Architectures

This section describes and illustrates the following I/O data flow architectural options supported by the FFT MegaCore function:

- Streaming
- Variable Streaming
- Buffered Burst
- Burst

For information on setting the architectural parameters in IP Toolbench, refer to “[Step 1: Parameterize](#)” on page 2-4.

### Streaming

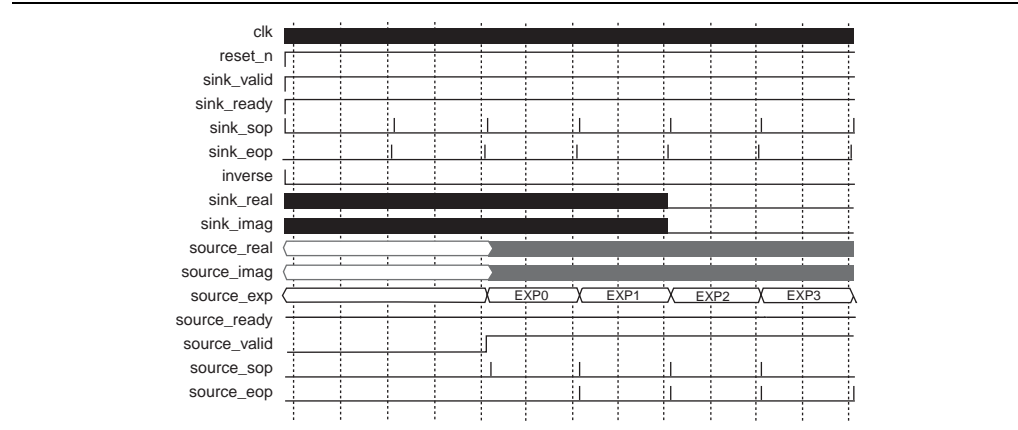
The streaming I/O data flow FFT architecture allows continuous processing of input data, and outputs a continuous complex data stream without the requirement to halt the data flow in or out of the FFT function.

#### Streaming FFT Operation

[Figure 3-3](#) shows an example simulation waveform.

For more information on the signals, see [Table 3-4](#) on page 3-16.

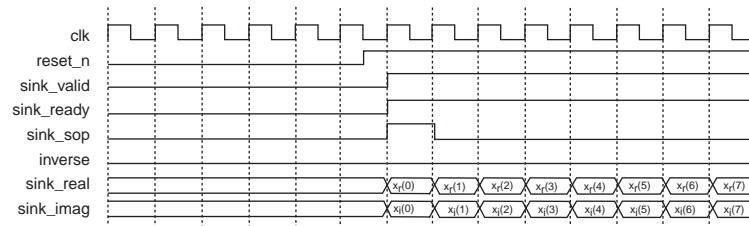
**Figure 3-3.** FFT Streaming Data Flow Architecture Simulation Waveform



Following the de-assertion of the system reset, the data source asserts `sink_valid` to indicate to the FFT function that valid data is available for input. A successful data transfer occurs when both the `sink_valid` and the `sink_ready` are asserted.

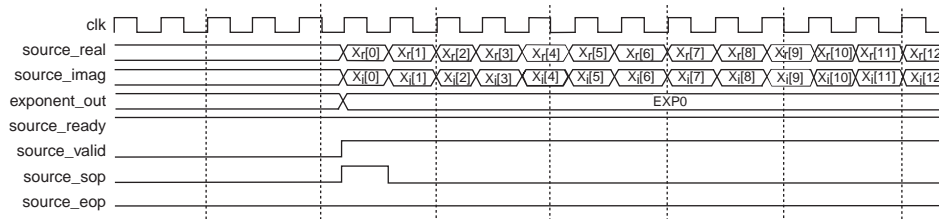
For more information on the Avalon-ST interface, refer to the [Avalon Interface Specifications](#).

When the data transfer is complete, `sink_sop` is de-asserted and the data samples are loaded in natural order. [Figure 3-4](#) shows the input flow control. When the final sample is loaded, the source asserts `sink_eop` and `sink_valid` for the last data transfer.

**Figure 3-4.** FFT Streaming Data Flow Architecture Input Flow Control

To change direction on a block-by-block basis, assert or deassert `inverse` (appropriately) simultaneously with the application of the `sink_sop` pulse (concurrent with the first input data sample of the block).

When the FFT has completed the transform of the input block, it asserts `source_valid` and outputs the complex transform domain data block in natural order. The FFT function asserts `source_sop` to indicate the first output sample. [Figure 3-5](#) shows the output flow control.

**Figure 3-5.** FFT Streaming Data Flow Architecture Output Flow Control

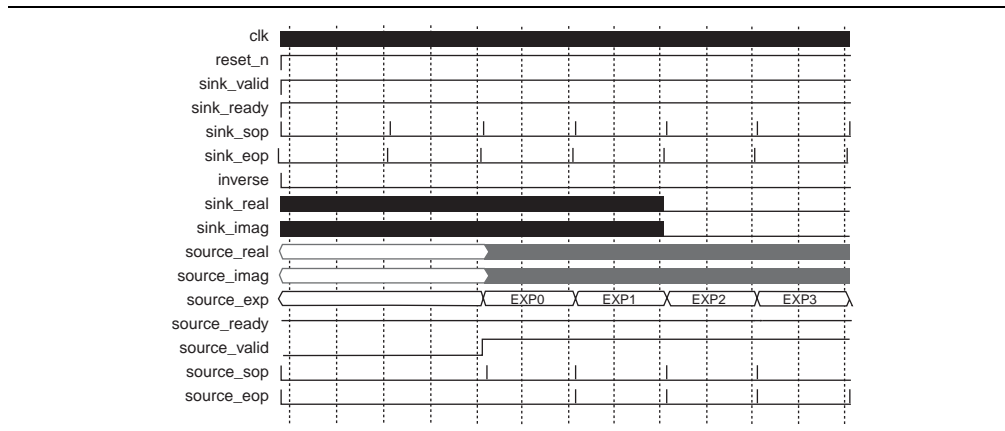
After  $N$  data transfers, `source_eop` is asserted to indicate the end of the output data block (see [Figure 3-3](#)).

### Enabling the Streaming FFT

[Figure 3-6](#) shows enabling of the streaming FFT.

The `sink_valid` signal must be asserted for `source_valid` to be asserted (and a valid data output). To extract the final frames of data from the FFT, you need to provide several frames where the `sink_valid` signal is asserted and apply the `sink_sop` and `sink_eop` signals in accordance with the Avalon-ST specification.

**Figure 3-6.** FFT Streaming—Enable



## Variable Streaming

The variable streaming architecture allows continuous streaming of input data and produces a continuous stream of output data similar to the streaming architecture.

### Change the Block Size

You change the size of the FFT on a block-by-block basis by changing the value of the `fftpts` simultaneously with the application of the `sink_sop` pulse (concurrent with the first input data sample of the block). `fftpts` uses a binary representation of the size of the transform, therefore for a block with maximum transfer size of 1,024.

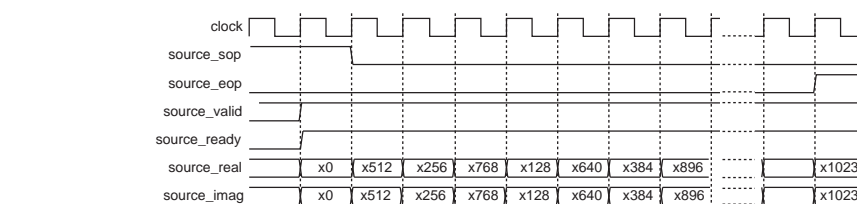
Table 3-2 shows the value of the `fftpts` signal and the equivalent transform size.

**Table 3-2.** `fftpts` & Transform Size

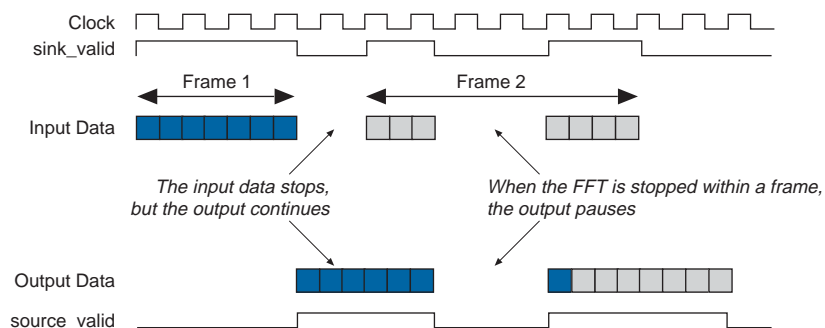
<code>fftpts</code>	Transform Size
1000000000	1,024
0100000000	512
0010000000	256
0001000000	128
0000100000	64

To change direction on a block-by-block basis, assert or de-assert `inverse` (appropriately) simultaneously with the application of the `sink_sop` pulse (concurrent with the first input data sample of the block).

When the FFT has completed the transform of the input block, it asserts `source_valid` and outputs the complex transform domain data block. The FFT function asserts the `source_sop` to indicate the first output sample. The order of the output data depends on the output order that you select in IP Toolbench. The output of the FFT may be in natural order or bit-reversed order. Figure 3-7 shows the output flow control when the output order is bit-reversed. If the output order is natural order, data flow control remains the same, but the order of samples at the output is in sequential order 1..*N*.

**Figure 3-7. Output Flow Control—Bit Reversed Order****Enabling the Variable Streaming FFT**

The FFT processes data when there is valid data transferred to the module (**sink\_valid** asserted). Figure 3-8 shows the FFT behavior when **sink\_valid** is deasserted.

**Figure 3-8. FFT Behavior When **sink\_valid** is Deasserted**

When **sink\_valid** is deasserted during a frame, the FFT stalls and no data is processed until **sink\_valid** is reasserted. This implies that any previous frames that are still in the FFT also stall.

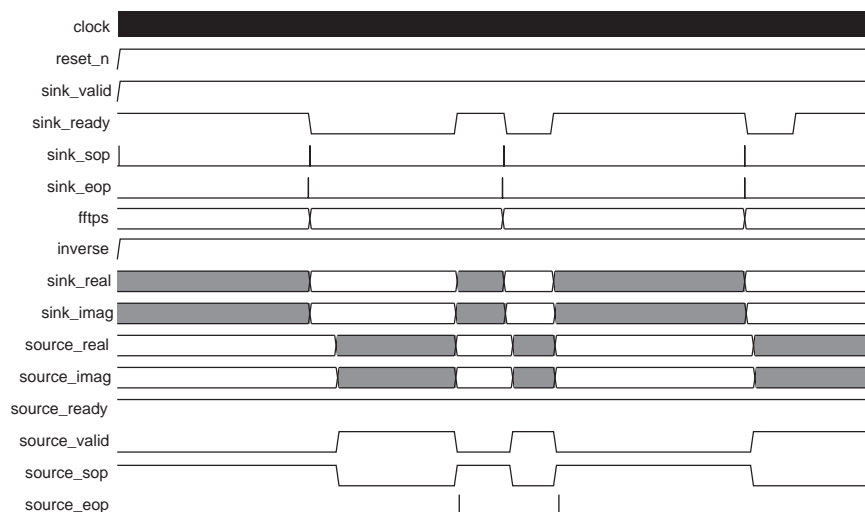
If **sink\_valid** is deasserted between frames, the data currently in the FFT continues to be processed and transferred to the output. Figure 3-8 shows the FFT behavior when **sink\_valid** is deasserted between frames and within a frame.

The FFT may optionally be disabled by deasserting the **clk\_en** signal.

**Dynamically Changing the FFT Size**

When the size of the incoming FFT changes, the FFT stalls the incoming data (deasserts the **sink\_ready** signal) until all of the previous FFT frames of the previous FFT size have been processed and transferred to the output. Figure 3-9 shows dynamically changing the FFT size for engine-only mode.

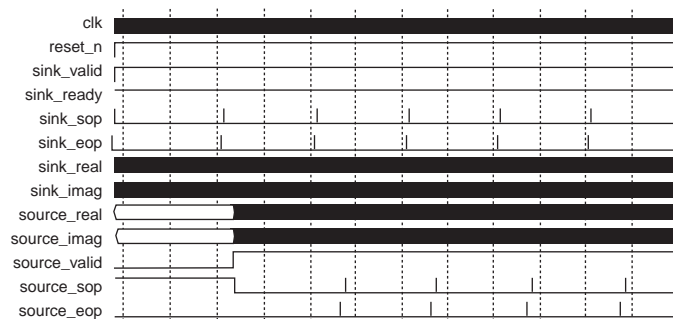
**Figure 3-9.** Dynamically Changing the FFT Size

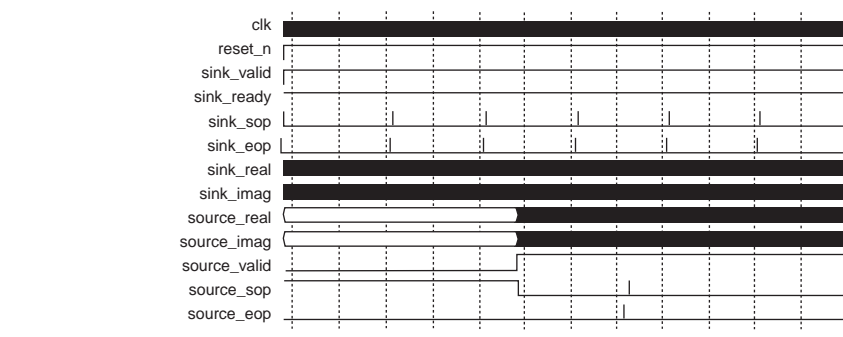


### The Effect of I/O Order

The order of samples entering and leaving the FFT is determined by the wizard selection in the I/O order panel. This selection also determines if the FFT is operating in engine-only mode or engine with bit-reversal mode. If the FFT operates in engine-only mode, the output data is available after approximately  $N + \text{latency}$  clocks cycles after the first sample was input to the FFT. Latency represents a small latency through the FFT core and is dependant on the transform size. For engine with bit-reversal mode, the output is available after approximately  $2N + \text{latency}$  cycles. [Figure 3-10](#) and [3-12](#) show the data flow output when the FFT is operating in engine-only mode and engine with bit-reversal mode respectively.

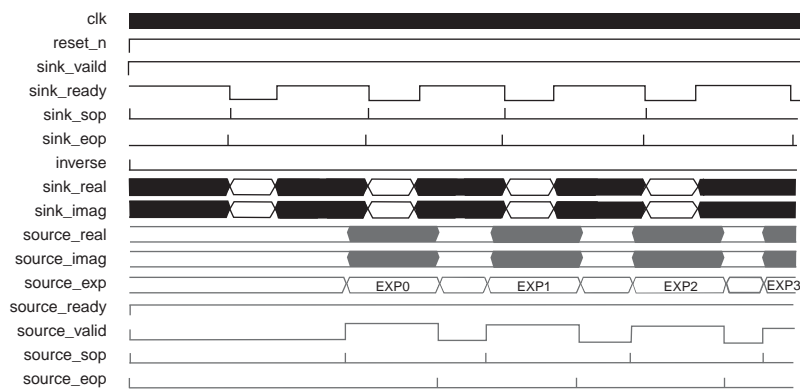
**Figure 3-10.** Data Flow—Engine-Only Mode



**Figure 3–11.** Data Flow—Engine with Bit-Reversal Mode

## Buffered Burst

The buffered burst I/O data flow architecture FFT requires fewer memory resources than the streaming I/O data flow architecture, but the tradeoff is an average block throughput reduction. [Figure 3–12](#) shows an example simulation waveform.

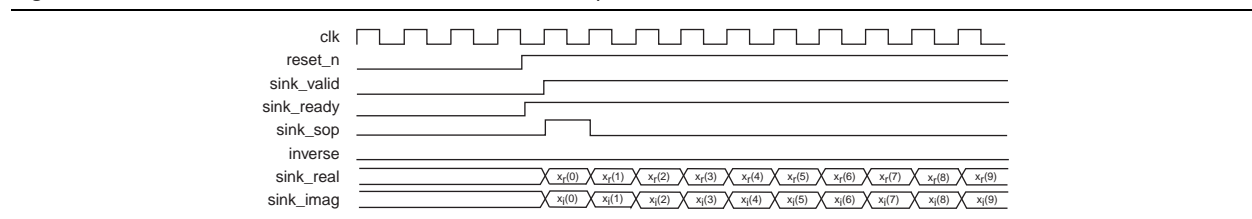
**Figure 3–12.** FFT Buffered Burst Data Flow Architecture Simulation Waveform

Following the de-assertion of the system reset, the data source asserts `sink_valid` to indicate to the FFT function that valid data is available for input. A successful data transfer occurs when both the `sink_valid` and the `sink_ready` are asserted.

The data source loads the first complex data sample into the FFT function and simultaneously asserts `sink_sop` to indicate the start of the input block. On the next clock cycle, `sink_sop` is de-asserted and the following  $N - 1$  complex input data samples should be loaded in natural order. On the last complex data sample, `sink_eop` should be asserted.

When the input block is loaded, the FFT function begins computing the transform on the stored input block. The `sink_ready` signal is held high as you can transfer the first few samples of the subsequent frame into the small FIFO at the input. If this FIFO is filled, the core deasserts the `sink_ready` signal. It is not mandatory to transfer samples during `sink_ready` cycles. [Figure 3–13](#) shows the input flow control.

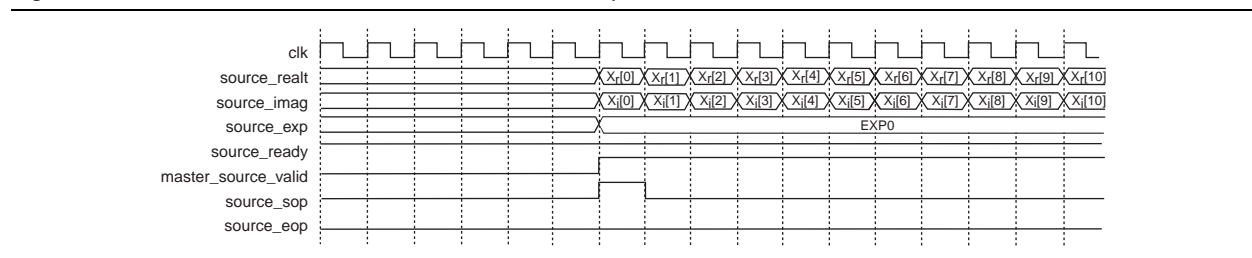
**Figure 3-13.** FFT Buffered Burst Data Flow Architecture Input Flow Control



Following the interval of time where the FFT processor reads the input samples from an internal input buffer, it re-asserts `sink_ready` indicating it is ready to read in the next input block. The beginning of the subsequent input block should be demarcated by the application of a pulse on `sink_sop` aligned in time with the first input sample of the next block.

As in all data flow architectures, the logical level of `inverse` for a particular block is registered by the FFT function at the time of the assertion of the start-of-packet signal, `sink_sop`. When the FFT has completed the transform of the input block, it asserts the `source_valid` and outputs the complex transform domain data block in natural order (see Figure 3-14).

**Figure 3-14.** FFT Buffered Burst Data Flow Architecture Output Flow Control



Signals `source_sop` and `source_eop` indicate the start-of-packet and end-of-packet for the output block data respectively (see Figure 3-12).



The `sink_valid` signal must be asserted for `source_valid` to be asserted (and a valid data output). You must therefore leave `sink_valid` signal asserted at the end of data transfers to extract the final frames of data from the FFT.

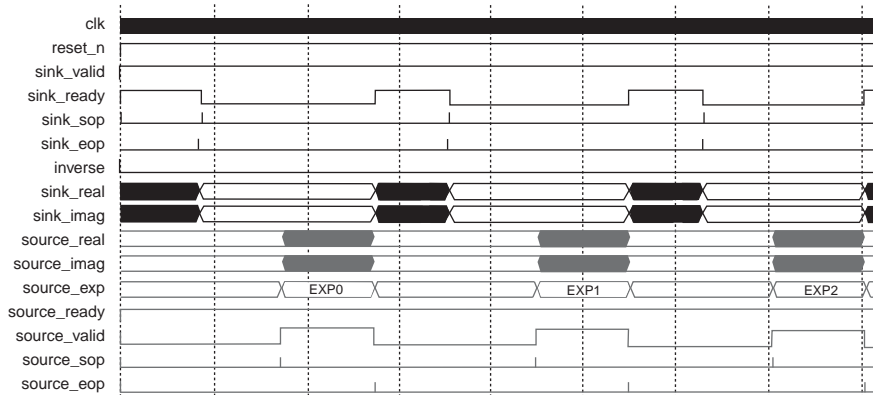


For information on enabling the buffered burst FFT, see “[Enabling the Streaming FFT](#)” on page 3-8.

## Burst

The burst I/O data flow architecture operates similarly to the buffered burst architecture, except that the burst architecture requires even lower memory resources for a given parameterization at the expense of reduced average throughput.

Figure 3-15 shows the simulation results for the burst architecture. Again, the signals `source_valid` and `sink_ready` indicate, to the system data sources and slave sinks either side of the FFT, when the FFT can accept a new block of data and when a valid output block is available on the FFT output.

**Figure 3-15.** FFT Burst Data Flow Architecture Simulation Waveform

In a burst I/O data flow architecture, the core can process a single input block only. There is a small FIFO buffer at the sink of the block and `sink_ready` is not deasserted until this FIFO buffer is full. Thus you can provide a small number of additional input samples associated with the subsequent input block. It is not mandatory to provide data to the FFT during `sink_ready` cycles. The burst architecture can load the rest of the subsequent FFT frame only when the previous transform has been fully unloaded.



For information on enabling the buffered burst FFT, see “[Enabling the Streaming FFT](#)” on [page 3-8](#).

## Parameters

[Table 3-3](#) shows the FFT MegaCore function’s parameters.

**Table 3-3.** Parameters (Part 1 of 3)

Parameter	Value	Description
Target device family	<device family>	Displays the target device family. The device family is normally preselected by the project specified in the Quartus II software.  The generated HDL for your MegaCore function variation may be incorrect if this value does not match the value specified in the Quartus II project.  The device family must be the same as your Quartus® II project device family.
Transform length	64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384. Variable streaming also allows 16, 32, 32768, and 65536.	The transform length. For variable streaming, this value is the maximum FFT length.
Data precision	8, 10, 12, 14, 16, 18, 20, 24, 28, 32	The data precision. The values 28 and 32 are available for variable streaming only.



**Table 3-3.** Parameters (Part 2 of 3)

Parameter	Value	Description
Twiddle precision	8, 10, 12, 14, 16, 18, 20, 24, 28, 32	The twiddle precision. Twiddle factor precision must be less than or equal to data precision.
FFT engine architecture	Quad output, single output	For both the buffered burst and burst I/O data flow architectures, you can choose between one, two, and four quad-output FFT engines working in parallel. Alternatively, if you have selected a single-output FFT engine architecture, you may choose to implement one or two engines in parallel. Multiple parallel engines reduce the FFT MegaCore function's transform time at the expense of device resources—which allows you to select the desired area and throughput trade-off point.  For more information on device resource and transform time trade-offs, refer to <a href="#">“Parameters” on page 3-14</a> . Not available for variable streaming.
Number of parallel FFT engines	1, 2, 4	
I/O data flow	Streaming Variable streaming Buffered burst Burst	Choose the FFT architecture.
I/O order	Bit reverse order, Natural order, $-N/2$ to $N/2$	The input and output order for data entering and leaving the FFT (variable streaming architecture only).
Data representation	Fixed point or floating point	The internal data representation type (variable streaming architecture only), either fixed point with natural bit-growth or single precision floating point.
Structure	3 mults/5 adders 4 mults/2 adders	You can implement the complex multiplier structure with four real multipliers and two adders/subtractors, or three multipliers, five adders, and some additional delay elements. The 4 mults/2 adders structure uses the DSP block structures to minimize logic usage, and maximize the DSP block usage. This option may also improve the push button $f_{MAX}$ . The 5 mults/3 adders structure requires fewer DSP blocks, but more LEs to implement. It may also produce a design with a lower $f_{MAX}$ . Not available for variable streaming.
Implement multipliers in	DSP block/logic cells Logic cells only DSP blocks only	Each real multiplication can be implemented in DSP blocks or LEs only, or using a combination of both. If you use a combination of DSP blocks and LEs, the FFT MegaCore function automatically extends the DSP block $18 \times 18$ multiplier resources with LEs as needed. Not valid for variable streaming.
Global clock enable	On or off	Turn on <b>Global Clock Enable</b> , if you want to add a global clock enable to your design.

**Table 3–3.** Parameters (Part 3 of 3)

Parameter	Value	Description
Twiddle ROM distribution	100% M4K to 100% M512 or 100% M9K to 100% MLAB	<p>High-throughput FFT parameterizations can require multiple shallow ROMs for twiddle factor storage. If your target device family supports M512 RAM blocks (or MLAB blocks in Stratix III devices), you can choose to distribute the ROM storage requirement between M4K (M9K) RAM and M512 (MLAB) RAM blocks by adjusting the <b>Twiddle ROM Resource Distribution</b> slider bar. Set the slider bar to the far left to implement the ROM storage completely in M4K (M9K) RAM blocks; set the slider bar to the far right to implement the ROM completely in M512 (MLAB) RAM blocks.</p> <p>Implementing twiddle ROM in M512 (MLAB) RAM blocks can lead to a more efficient device internal memory bit usage. Alternatively, this option can be used to conserve M4K (M9K) RAM blocks used for the storage of FFT data or other storage requirements in your system.</p> <p>Not available for variable streaming.</p>
Use M-RAM or M144K blocks	On or off	<p>Implements suitable data RAM blocks within the FFT MegaCore function in M-RAM (M144K in Stratix III devices) to reduce M4K (M9K) RAM block usage, in device families that support M-RAM blocks.</p> <p>Not available for variable streaming.</p>
Implement appropriate logic functions in RAM	On or off	<p>Uses embedded RAM blocks to implement internal logic functions, for example, tapped delay lines in the FFT MegaCore function. This option reduces the overall LE count.</p> <p>Not available for variable streaming.</p>

## Signals

Table 3–4 shows the Avalon-ST interface signals.



For more information on the Avalon-ST interface, refer to the *Avalon Streaming Interface Specification*.

**Table 3–4.** Avalon-ST Signals (Part 1 of 3)

Signal Name	Direction	Avalon-ST Type	Size	Description
clk	Input	clk	1	Clock signal that clocks all internal FFT engine components.
reset_n	Input	reset_n	1	Active-low asynchronous reset signal.
sink_eop	Input	endofpacket	1	Indicates the end of the incoming FFT frame.

**Table 3-4. Avalon-ST Signals (Part 2 of 3)**

Signal Name	Direction	Avalon-ST Type	Size	Description
sink_error	Input	error	2	<p>Indicates an error has occurred in an upstream module, because of an illegal usage of the Avalon-ST protocol. The following errors are defined (see <a href="#">Table 3-6</a>):</p> <ul style="list-style-type: none"> <li>■ 00 = no error</li> <li>■ 01 = missing SOP</li> <li>■ 10 = missing EOP</li> <li>■ 11 = unexpected EOP</li> </ul> <p>If this signal is not used in upstream modules, set to zero.</p>
sink_imag	Input	data	<i>data precision width</i>	Imaginary input data, which represents a signed number of data precision bits.
sink_ready	Output	ready	1	Asserted by the FFT engine when it can accept data. It is not mandatory to provide data to the FFT during ready cycles.
sink_real	Input	data	<i>data precision width</i>	Real input data, which represents a signed number of data precision bits.
sink_sop	Input	startofpacket	1	Indicates the start of the incoming FFT frame.
sink_valid	Input	valid	1	Asserted when data on the data bus is valid. When <code>sink_valid</code> and <code>sink_ready</code> are asserted, a data transfer takes place. See <a href="#">“Enabling the Variable Streaming FFT” on page 3-10</a> .
source_eop	Output	endofpacket	1	Marks the end of the outgoing FFT frame. Only valid when <code>source_valid</code> is asserted.
source_error	Output	error	2	<p>Indicates an error has occurred either in an upstream module or within the FFT module (logical OR of <code>sink_error</code> with errors generated in the FFT).</p> <p>The following errors are defined (see <a href="#">Table 3-6</a>):</p> <ul style="list-style-type: none"> <li>■ 00 = no error</li> <li>■ 01 = missing SOP</li> <li>■ 10 = unexpected EOP</li> <li>■ 11 = other error</li> </ul>
source_exp	Output	data	6	Streaming, burst, and buffered burst architectures only. Signed block exponent: Accounts for scaling of internal signal values during FFT computation.

**Table 3-4. Avalon-ST Signals (Part 3 of 3)**

Signal Name	Direction	Avalon-ST Type	Size	Description
source_imag	Output	data	(data precision width + growth) (1)	Imaginary output data. For burst, buffered burst, and streaming FFTs, the output data width is equal to the input data width. For variable streaming FFTs, the size of the output data is dependent on the number of stages defined for the FFT and is approximately 2.5 bits per radix 2 <sup>2</sup> stage.
source_ready	Input	ready	1	Asserted by the downstream module if it is able to accept data.
source_real	Output	data	(data precision width + growth) (1)	Real output data. For burst, buffered burst, and streaming FFTs, the output data width is equal to the input data width. For variable streaming FFTs, the size of the output data is dependent on the number of stages defined for the FFT and is approximately 2.5 bits per radix 2 <sup>2</sup> stage.
source_sop	Output	startofpacket	1	Marks the start of the outgoing FFT frame. Only valid when source_valid is asserted.
source_valid	Output	valid	1	Asserted by the FFT when there is valid data to output.

**Note to Table 3-4:**(1) Variable streaming FFT only. Growth is  $2.5 \times (\text{number of stages}) = 2.5 \times (\log_4(\text{MAX}(\text{fftpts})))$ 

Table 3-5 shows the component specific signals.

**Table 3-5. Component Specific Signals**

Signal Name	Direction	Size	Description
fftpts_in	Input	$\log_2(\text{maximum number of points})$	The number of points in this FFT frame. If this value is not specified, the FFT can not be a variable length. The default behavior is for the FFT to have fixed length of maximum points. Only sampled at SOP.
fftpts_out	Output	$\log_2(\text{maximum number of points})$	The number of points in this FFT frame synchronized to the Avalon-ST source interface. Variable streaming only.
inverse	Input	1	Inverse FFT calculated if asserted. Only sampled at SOP.
clk_ena	Input	1	Active-high global clock enable input. When de-asserted, the FFT is disabled.

Incorrect usage of the Avalon-ST interface protocol on the sink interface results in an error on source\_error. Table 3-6 defines the behavior of the FFT when an incorrect Avalon-ST transfer is detected. If an error occurs, the behavior of the FFT is undefined and you must reset the FFT with reset\_n.

**Table 3-6. Error Handling Behavior**

Error	source_error	Description
Missing SOP	01	Missing SOP is asserted when valid goes high, but there is no start of frame.

**Table 3-6.** Error Handling Behavior

Error	source_error	Description
Missing EOP	10	Missing EOP is asserted if the FFT accepts <i>N</i> valid samples of an FFT frame, but it is not completed with an EOP signal.
Unexpected EOP	11	When EOP is asserted before <i>N</i> valid samples are accepted.



## Introduction

The FFT MegaCore® function uses block-floating-point (BFP) arithmetic internally to perform calculations. BFP architecture is a trade-off between fixed-point and full floating-point architecture.

Unlike an FFT block that uses floating point arithmetic, a block-floating-point FFT block does not provide an input for exponents. Internally, a complex value integer pair is represented with a single scale factor that is typically shared among other complex value integer pairs. After each stage of the FFT, the largest output value is detected and the intermediate result is scaled to improve the precision. The exponent records the number of left or right shifts used to perform the scaling. As a result, the output magnitude relative to the input level is:

$$\text{output} * 2^{-\text{exponent}}$$

For example, if  $\text{exponent} = -3$ , the input samples are shifted right by three bits, and hence the magnitude of the output is  $\text{output} * 2^3$ .

## Block Floating Point

After every pass through a radix-2 or radix-4 engine in the FFT core, the addition and multiplication operations cause the data bits width to grow. In other words, the total data bits width from the FFT operation grows proportionally to the number of passes. The number of passes of the FFT/IFFT computation depends on the logarithm of the number of points. [Table A-1 on page A-2](#) shows the possible exponents for corresponding bit growth.

A fixed-point architecture FFT needs a huge multiplier and memory block to accommodate the large bit width growth to represent the high dynamic range. Though floating-point is powerful in arithmetic operations, its power comes at the cost of higher design complexity such as a floating-point multiplier and a floating-point adder. BFP arithmetic combines the advantages of floating-point and fixed-point arithmetic. BFP arithmetic offers a better signal-to-noise ratio (SNR) and dynamic range than does floating-point and fixed-point arithmetic with the same number of bits in the hardware implementation.

In a block-floating-point architecture FFT, the radix-2 or radix-4 computation of each pass shares the same hardware, with the data being read from memory, passed through the core engine, and written back to memory. Before entering the next pass, each data sample is shifted right (an operation called "scaling") if there is a carry-out bit from the addition and multiplication operations. The number of bits shifted is based on the difference in bit growth between the data sample and the maximum data sample detected in the previous stage. The maximum bit growth is recorded in the exponent register. Each data sample now shares the same exponent value and data bit

width to go to the next core engine. The same core engine can be reused without incurring the expense of a larger engine to accommodate the bit growth. The output SNR depends on how many bits of right shift occur and at what stages of the radix core computation they occur. In other words, the signal-to-noise ratio is data dependent and you need to know the input signal to compute the SNR.

## Calculating Possible Exponent Values

Depending on the length of the FFT/IFFT, the number of passes through the radix engine is known and therefore the range of the exponent is known. The possible values of the exponent are determined by the following equations:

$P = \text{ceil}\{\log_4 N\}$ , where  $N$  is the transform length

$R = 0$  if  $\log_2 N$  is even, otherwise  $R = 1$

Single output range =  $(-3P+R, P+R-4)$

Quad output range =  $(-3P+R+1, P+R-7)$

These equations translate to the values in [Table A-1](#).

**Table A-1.** Exponent Scaling Values for FFT / IFFT (Note 1)

N	P	Single Output Engine		Quad Output Engine	
		Max (2)	Min (2)	Max (2)	Min (2)
64	3	-9	-1	-8	-4
128	4	-11	1	-10	-2
256	4	-12	0	-11	-3
512	5	-14	2	-13	-1
1,024	5	-15	1	-14	-2
2,048	6	-17	3	-16	0
4,096	6	-18	2	-17	-1
8,192	7	-20	4	-19	1
16,384	7	-21	3	-20	0

**Note to Table A-1:**

- (1) This table lists the range of exponents, which is the number of scale events that occurred internally. For IFFT, the output must be divided by  $N$  externally. If more arithmetic operations are performed after this step, the division by  $N$  must be performed at the end to prevent loss of precision.
- (2) The max and min values show the number of times the data is shifted. A negative value indicates shifts to the left, while a positive value indicates shifts to the right.

## Implementing Scaling

To implement the scaling algorithm, follow these steps:

1. Determine the length of the resulting full scale dynamic range storage register. To get the length, add the width of the data to the number of times the data is shifted (the max value in [Table A-1](#)). For example, for a 16-bit data, 256-point Quad Output FFT/IFFT, max = -11 and min = -3. The max value indicates 11 shifts to the left, so the resulting full scaled data width is  $16 + 11$ , or 27 bits.



2. Map the output data to the appropriate location within the expanded dynamic range register based upon the exponent value. To continue the above example, the 16-bit output data [15..0] from the FFT/IFFT is mapped to [26..11] for an exponent of  $-11$ , to [25..10] for an exponent of  $-10$ , to [24..9] for an exponent of  $-9$ , and so on.
3. Sign extend the data within the full scale register.

A sample of Verilog HDL code that illustrates the scaling of the output data (for exponents  $-11$  to  $-9$ ) with sign extension is shown in the following example:

```
case (exp)
  6'b110101 : //-11 Set data equal to MSBs
    begin
      full_range_real_out[26:0] <= {real_in[15:0],11'b0};
      full_range_imag_out[26:0] <= {imag_in[15:0],11'b0};
    end
  6'b110110 : //-10 Equals left shift by 10 with sign extension
    begin
      full_range_real_out[26] <= {real_in[15]};
      full_range_real_out[25:0] <= {real_in[15:0],10'b0};
      full_range_imag_out[26] <= {imag_in[15]};
      full_range_imag_out[25:0] <= {imag_in[15:0],10'b0};
    end
  6'b110111 : //-9 Equals left shift by 9 with sign extension
    begin
      full_range_real_out[26:25] <= {real_in[15],real_in[15]};
      full_range_real_out[24:0] <= {real_in[15:0],9'b0};
      full_range_imag_out[26:25] <= {imag_in[15],imag_in[15]};
      full_range_imag_out[24:0] <= {imag_in[15:0],9'b0};
    end
  .
  .
  .
endcase
```

In this example, the output provides a full scale 27-bit word. You need to choose how many and which bits should be carried forward in the processing chain. The choice of bits determines the absolute gain relative to the input sample level.

**Figure A-1** demonstrates the effect of scaling for all possible values for the 256-point quad output FFT with an input signal level of 5000H. The output of the FFT is 280H when the exponent =  $-5$ . The figure illustrates all cases of valid exponent values of scaling to the full scale storage register [26..0]. Since the exponent is  $-5$ , you need to look at the register values for that column. This data is shown in the last two columns in the figure. Note that the last column represents the gain compensated data after the scaling (0005000H), which agrees with the input data as expected. If you want to keep 16 bits for subsequent processing, you can choose the bottom 16 bits that result in 5000H. However, if you choose a different bit range, such as the top 16 bits, the result is 000AH. Therefore, the choice of bits affects the relative gain through the processing chain.

Because this example has 27 bits of full scale resolution and 16 bits of output resolution, choose the bottom 16 bits to maintain unity gain relative to the input signal. Choosing the LSBs is not the only solution or the correct one for all cases. The choice depends on which signal levels are important. One way to empirically select the proper range is by simulating test cases that implement expected system data. The output of the simulations should tell what range of bits to use as the output register. If the full scale data is not used (or just the MSBs), you must saturate the data to avoid wraparound problems.

Figure A-1. Scaling of Input Data Sample = 5000H

				Exponent										Looking at Exponent = -5	
Bit	Input	Output Data		-11	-10	-9	-8	-7	-6	-5	-4	-3		Taking All Bits	Sign Extend / Pad
	5000 H	280 H													
26				0											0
25				0	0										0
24				0	0	0									0
23				0	0	0	0								0
22				0	0	0	0	0							0
21				0	0	0	0	0	0						0
20				1	0	0	0	0	0	0				0	0
19				0	1	0	0	0	0	0	0			0	0
18				1	0	1	0	0	0	0	0	0		0	0
17				0	1	0	1	0	0	0	0	0		0	0
16				0	0	1	0	1	0	0	0	0		0	0
15	0	0		0	0	0	1	0	1	0	0	0		0	0
14	1	0		0	0	0	0	1	0	1	0	0		1	1
13	0	0		0	0	0	0	0	1	0	1	0		0	0
12	1	0		0	0	0	0	0	0	1	0	1		1	1
11	0	0		0	0	0	0	0	0	0	1	0		0	0
10	0	0			0	0	0	0	0	0	0	1		0	0
9	0	1				0	0	0	0	0	0	0		0	0
8	0	0					0	0	0	0	0	0		0	0
7	0	1						0	0	0	0	0		0	0
6	0	0							0	0	0	0		0	0
5	0	0								0	0	0		0	0
4	0	0									0	0		0	0
3	0	0										0		0	0
2	0	0												0	0
1	0	0													0
0	0	0													0

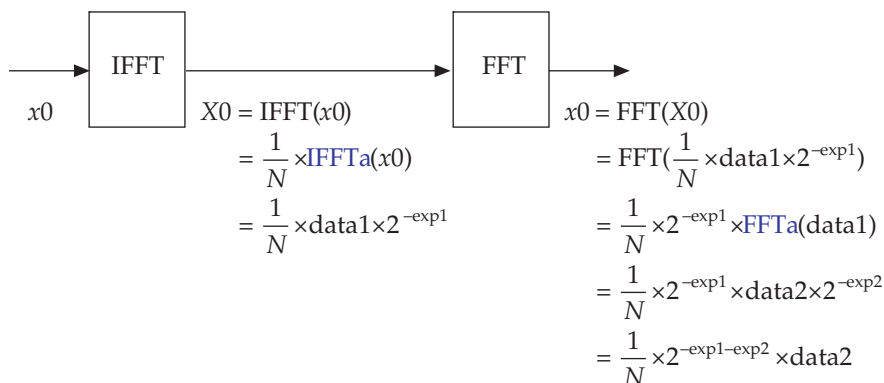
## Achieving Unity Gain in an IFFT+FFT pair

Given sufficiently high precision, such as with floating-point arithmetic, it is theoretically possible to obtain unity gain when an IFFT and FFT are cascaded. However, in BFP arithmetic, special attention must be paid to the exponent values of the IFFT/FFT blocks to achieve the unity gain. This section explains the steps required to derive a unity gain output from an Altera IFFT/FFT MegaCore pair, using BFP arithmetic.

Because BFP arithmetic does not provide an input for the exponent, you must keep track of the exponent from the IFFT block if you are feeding the output to the FFT block immediately thereafter and divide by  $N$  at the end to acquire the original signal magnitude.

Figure A-2 shows the operation of IFFT followed by FFT and derives the equation to achieve unity gain.

**Figure A-2.** Derivation to Achieve IFFT/FFT Pair Unity Gain



where:

$x0$  = Input data to IFFT

$X0$  = Output data from IFFT

$N$  = number of points

$\text{data1}$  = IFFT output data and FFT input data

$\text{data2}$  = FFT output data

$\text{exp1}$  = IFFT output exponent

$\text{exp2}$  = FFT output exponent

IFFTa = IFFT

FFTa = FFT

Any scaling operation on  $X0$  followed by truncation loses the value of  $\text{exp1}$  and does not result in unity gain at  $x0$ . Any scaling operation must be done on  $X0$  only when it is the final result. If the intermediate result  $X0$  is first padded with  $\text{exp1}$  number of zeros and then truncated or if the data bits of  $X0$  are truncated, the scaling information is lost.

One way to keep unity gain is by passing the  $\text{exp1}$  value to the output of the FFT block. The other way is to preserve the full precision of  $\text{data1} \times 2^{-\text{exp1}}$  and use this value as input to the FFT block. The disadvantage of the second method is a large size requirement for the FFT to accept the input with growing bit width from IFFT operations. The resolution required to accommodate this bit width will, in most cases, exceed the maximum data width supported by the core.



For more information, refer to the *FFT/IFFT Unity Gain* design example at [www.altera.com](http://www.altera.com).



## Revision History

The following table shows the revision history for this user guide.

Date	Version	Changes Made
March 2009	9.0	Added Arria® II GX device support
November 2008	8.1	No changes.
May 2008	8.0	<ul style="list-style-type: none"> <li>Added Stratix® IV device support</li> <li>Changed descriptions of the behavior of <code>sink_valid</code> and <code>sink_ready</code></li> </ul>
October 2007	7.2	<ul style="list-style-type: none"> <li>Corrected timing diagrams</li> <li>Added single precision floating point data representation information</li> </ul>
May 2007	7.1	<ul style="list-style-type: none"> <li>Added support for Arria™ GX devices</li> <li>Added new generated files</li> </ul>
December 2006	7.0	Added support for Cyclone® III devices.
December 2006	6.1	<ul style="list-style-type: none"> <li>Changed interface information.</li> <li>Added variable streaming information</li> </ul>

## How to Contact Altera

For the most up-to-date information about Altera® products, see the following table.






Contact <i>(Note 1)</i>	Contact Method	Address
Technical support	Website	<a href="http://www.altera.com/support">www.altera.com/support</a>
Technical training	Website	<a href="http://www.altera.com/training">www.altera.com/training</a>
	Email	<a href="mailto:custrain@altera.com">custrain@altera.com</a>
Altera literature services	Email	<a href="mailto:literature@altera.com">literature@altera.com</a>
Non-technical support (General) (Software Licensing)	Email	<a href="mailto:nacomp@altera.com">nacomp@altera.com</a>
	Email	<a href="mailto:authorization@altera.com">authorization@altera.com</a>

**Note:**

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

## Typographic Conventions

The following table shows the typographic conventions that this document uses.

Visual Cue	Meaning
<b>Bold Type with Initial Capital Letters</b>	Indicates command names, dialog box titles, dialog box options, and other GUI labels. For example, <b>Save As</b> dialog box.
<b>bold type</b>	Indicates directory names, project names, disk drive names, file names, file name extensions, and software utility names. For example, <b>\qdesigns</b> directory, <b>d:</b> drive, and <b>chiptrip.gdf</b> file.
<i>Italic Type with Initial Capital Letters</i>	Indicates document titles. For example: <i>AN 519: Stratix IV Design Guidelines</i> .
<i>Italic type</i>	Indicates variables. For example, $n + 1$ . Variable names are enclosed in angle brackets (<>). For example, <file name> and <project name>.pof file.
Initial Capital Letters	Indicates keyboard keys and menu names. For example, Delete key and the Options menu.
"Subheading Title"	Quotation marks indicate references to sections within a document and titles of Quartus II Help topics. For example, "Typographic Conventions."
Courier type	Indicates signal, port, register, bit, block, and primitive names. For example, data1, tdi, and input. Active-low signals are denoted by suffix n. Example: resetn. Indicates command line commands and anything that must be typed exactly as it appears. For example, c:\qdesigns\tutorial\chiptrip.gdf. Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword SUBDESIGN), and logic function names (for example, TRI).
1., 2., 3., and a., b., c., and so on.	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■	Bullets indicate a list of items when the sequence of the items is not important.
	The hand points to information that requires special attention.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
	A warning calls attention to a condition or possible situation that can cause you injury.
	The angled arrow instructs you to press the enter key.
	The feet direct you to more information about a particular topic.